

O'REILLY®



# Myśl w języku Java!

Nauka programowania

Helion 

Allen B. Downey, Chris Mayfield

Tytuł oryginału: Think Java: How to Think Like a Computer Scientist

Tłumaczenie: Łukasz Suma

ISBN: 978-83-283-3006-1

© 2017 Helion SA

Authorized Polish translation of the English edition of Think Java, ISBN 9781491929568

© 2016 Allen B. Downey, Chris Mayfield

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto: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/mysjav>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

---

# Spis treści

<b>Wstęp .....</b>	<b>9</b>
<b>1. Droga programu .....</b>	<b>13</b>
Czym jest programowanie?	13
Czym jest informatyka?	14
Języki programowania	15
Program „Witaj, świecie!”	16
Wyświetlanie łańcuchów znakowych	17
Sekwencje ucieczki	18
Formatowanie kodu	19
Debugowanie kodu	20
Słownictwo	20
Ćwiczenia	22
<b>2. Zmienne i operatory .....</b>	<b>25</b>
Deklarowanie zmiennych	25
Przypisanie	26
Diagramy stanu	27
Wyświetlanie zmiennych	28
Operatory arytmetyczne	28
Liczby zmiennoprzecinkowe	29
Błędy zaokrągleń	31
Operatory działające na łańcuchach znakowych	32
Kompozycja	33
Typy błędów	33
Słownictwo	36
Ćwiczenia	38

<b>3. Wejście i wyjście .....</b>	<b>41</b>
Klasa System	41
Klasa Scanner	42
Struktura programu	43
Cale na centymetry	44
Literały i stałe	45
Formatowanie danych wyjściowych	45
Centymetry na cale	46
Operator modulo	47
Łączenie wszystkiego w całość	48
„Bug” w klasie Scanner	49
Słownictwo	50
Ćwiczenia	51
<b>4. Metody niezwracające wartości .....</b>	<b>55</b>
Metody matematyczne	55
Kompozycja raz jeszcze	56
Dodawanie nowych metod	57
Przepływ wykonania	59
Parametry i argumenty	60
Wiele parametrów	61
Diagramy stosu	62
Czytanie dokumentacji	62
Pisanie dokumentacji	65
Słownictwo	66
Ćwiczenia	67
<b>5. Warunki i operacje logiczne .....</b>	<b>69</b>
Operatory relacyjne	69
Operatory logiczne	70
Instrukcje warunkowe	71
Tworzenie łańcuchów i zagnieżdżanie	72
Zmienne flagi	73
Instrukcja return	74
Sprawdzanie danych wejściowych	74
Metody rekurencyjne	75
Rekurencyjne diagramy stosu	76
Liczby binarne	77
Słownictwo	79
Ćwiczenia	80

<b>6. Metody zwracające wartość .....</b>	<b>83</b>
Wartości zwracane	83
Pisanie metod	85
Kompozycja metody	87
Przeciążanie	88
Metody typu boolean	89
Znaczniki Javadoc	90
Więcej rekurencji	90
Akt wiary	92
Jeszcze jeden przykład	93
Słownictwo	94
Ćwiczenia	95
<b>7. Pętle .....</b>	<b>99</b>
Instrukcja while	99
Generowanie tablic	100
Hermetyzacja i uogólnianie	102
Więcej uogólniania	104
Instrukcja for	106
Pętla do-while	107
Instrukcje break i continue	108
Słownictwo	109
Ćwiczenia	110
<b>8. Tablice .....</b>	<b>113</b>
Tworzenie tablic	113
Dostęp do tablic	114
Wyświetlanie tablic	115
Kopiowanie tablic	116
Długość tablic	117
Przechodzenie przez tablice	117
Liczby losowe	118
Przechodzenie i zliczanie	119
Budowanie histogramu	120
Rozszerzona pętla for	121
Słownictwo	122
Ćwiczenia	123

<b>9. Łańcuchy znakowe i spółka .....</b>	<b>127</b>
Znaki	127
Niezmiennosc łańcuchów znakowych	128
Przechodzenie przez łańcuchy znakowe	129
Podłańcuchy znakowe	130
Metoda indexOf	131
Porównywanie łańcuchów znakowych	132
Formatowanie łańcuchów znakowych	133
Klasy opakowujące	133
Argumenty wiersza poleceń	134
Słownictwo	135
Ćwiczenia	136
<b>10. Obiekty .....</b>	<b>141</b>
Obiekty klasy Point	141
Atrybuty	142
Obiekty jako parametry	142
Obiekty jako wartości zwracane	143
Obiekty zmienne	144
Korzystanie z aliasów	145
Słowo kluczowe null	146
Oczyszczanie pamięci	147
Diagramy klas	147
Źródła biblioteki języka Java	148
Słownictwo	149
Ćwiczenia	150
<b>11. Klasy .....</b>	<b>155</b>
Klasa Time	155
Konstruktory	156
Więcej konstruktorów	157
Gettery i settery	158
Wyświetlanie obiektów	160
Metoda toString	161
Metoda equals	162
Dodawanie obiektów klasy Time	163
Czyste metody i modyfikatory	164
Słownictwo	165
Ćwiczenia	166

<b>12. Tablice obiektów .....</b>	<b>171</b>
Obiekty klasy Card	171
Metoda toString klasy Card	172
Zmienne klasy	174
Metoda compareTo	175
Niezmienność obiektów klasy Card	176
Tablica obiektów klasy Card	177
Wyszukiwanie sekwencyjne	179
Wyszukiwanie binarne	179
Śledzenie wykonania kodu	180
Wersja rekurencyjna	181
Słownictwo	182
Ćwiczenia	182
<b>13. Obiekty zawierające tablice .....</b>	<b>185</b>
Klasa Deck	185
Tasowanie talii kart	186
Sortowanie przez wybieranie	187
Sortowanie przez scalanie	188
Podtalie	188
Scalanie talii	190
Dodanie rekurencji	190
Słownictwo	191
Ćwiczenia	192
<b>14. Obiekty zawierające inne obiekty .....</b>	<b>195</b>
Talie i układy	195
Klasa CardCollection	196
Dziedziczenie	199
Rozdawanie kart	200
Klasa Player	201
Klasa Eights	203
Relacje pomiędzy klasami	206
Słownictwo	207
Ćwiczenia	208
<b>A Narzędzia programistyczne .....</b>	<b>211</b>
Instalacja programu DrJava	211
Panel Interactions programu DrJava	212
Interfejs wiersza poleceń	213

Testowanie w wierszu poleceń	214
Uruchamianie narzędzia Checkstyle	216
Śledzenie wykonania za pomocą debuggera	217
Testowanie przy użyciu narzędzia JUnit	218
Słownictwo	220
<b>B Grafika 2D w Javie .....</b>	<b>223</b>
Tworzenie grafiki	223
Metody graficzne	224
Przykładowy rysunek	226
Słownictwo	226
Ćwiczenia	227
<b>C Debugowanie .....</b>	<b>231</b>
Błędy czasu kompilacji	231
Błędy czasu wykonania	234
Błędy logiczne	237
<b>Skorowidz .....</b>	<b>243</b>



Komputery są często wykorzystywane do automatyzowania powtarzających się zadań. Powtarzanie czynności bez popełniania błędów to coś, z czym komputery radzą sobie świetnie, a ludzie raczej marnie.

Wykonywanie tego samego kodu wielokrotnie nosi nazwę **iteracji** (ang. *iteration*). Mieliśmy już do czynienia z metodami, takimi jak `countdown` oraz `factorial`, które powtarzają swoje działanie przy użyciu rekurencji. Choć jest to sposób bardzo elegancki i oferuje spore możliwości, wymaga pewnej wprawy, aby mógł być skutecznie stosowany. Język Java zapewnia konstrukcje, dzięki którym iteracja staje się znacznie prostsza: są to instrukcje `while` oraz `for`.

## Instrukcja `while`

Dzięki zastosowaniu instrukcji `while` możemy raz jeszcze napisać metodę `countdown`, tym razem w sposób przedstawiony poniżej:

```
public static void countdown(int n) {
    while (n > 0) {
        System.out.println(n);
        n = n - 1;
    }
    System.out.println("Odpalamy!");
}
```

Instrukcję `while` da się po angielsku przeczytać niemal tak, jakby była napisana w języku naturalnym: „Dopóki `n` jest większe od zera, wyświetlaj wartość `n`, a następnie zmniejsz ją o 1. Gdy osiągniesz wartość 0, wyświetl Odpalamy!”.

Wyrażenie ujęte w nawias okrągły nazywane jest warunkiem. Instrukcje w nawiasie klamrowym tworzą tzw. **ciało** (ang. *body*). Przepływ wykonania jest w przypadku instrukcji `while` następujący:

1. Sprawdzany jest warunek, w wyniku czego otrzymujemy wartość `true` lub `false`.
2. Gdy warunek nie jest spełniony (otrzymaliśmy wartość `false`), ciało instrukcji `while` jest pomijane i następuje przejście do kolejnej instrukcji.
3. Gdy warunek jest spełniony (otrzymaliśmy wartość `true`), ciało instrukcji jest wykonywane i następuje powrót do kroku 1.

Ten typ przepływu nosi nazwę **pętli** (ang. *loop*), ponieważ następuje tu cykliczne przejście z ostatniego kroku z powrotem do pierwszego, a więc mamy do czynienia z zapętleniem.

W ciele pętli powinna zachodzić zmiana jednej lub większej liczby zmiennych, aby ostatecznie warunek uzyskał wartość `false` i wykonywanie pętli mogło się zakończyć. W przeciwnym przypadku będzie ona powtarzana w nieskończoność; sytuacja taka określana jest mianem **pętli nieskończonej** (ang. *infinite loop*). Niewyczerpanym źródłem rozrywki jest dla informatyków czytanie instrukcji użycia wydrukowanych na etykietach szamponów, które zwykle każą „nanieść niewielką ilość na włosy, umyć, spłukać, czynność powtórzyć”, stanowią więc nieskończone pętle.

W przypadku metody `countdown` można udowodnić, że pętla się kończy, gdy wartość przechowywana przez zmienną `n` jest dodatnia. Ogólnie rzecz biorąc, nie zawsze da się tak łatwo stwierdzić, czy pętla zostanie kiedyś zakończona. Przedstawiona poniżej pętla wykonuje się do chwili, w której wartość zmiennej `n` będzie wynosiła 1 (co sprawia, że warunek ma wartość `false`):

```
public static void sequence(int n) {
    while (n != 1) {
        System.out.println(n);
        if (n % 2 == 0) { // n jest parzyste
            n = n / 2;
        } else { // n jest nieparzyste
            n = n * 3 + 1;
        }
    }
}
```

Przy każdym przejściu przez pętlę na ekranie wyświetlana jest wartość zmiennej `n`, a następnie sprawdzane jest, czy wartość ta jest parzysta, czy nieparzysta. Jeśli wartość ta jest parzysta, zostaje podzielona przez dwa. Jeśli jest nieparzysta, zostaje zastąpiona przez wartość  $3n+1$ . Na przykład gdy początkowa wartość (argument przekazywany metodzie `sequence`) wynosi 3, wynikowa sekwencja będzie miała postać: 3, 10, 5, 16, 8, 4, 2, 1.

Z uwagi na fakt, że wartość zmiennej `n` czasami wzrasta, a czasami maleje, nie istnieje żaden oczywisty dowód na to, że wartość ta osiągnie w którymś momencie 1 i program kiedykolwiek się skończy. Na przykład gdy wartość początkowa będzie potęgą dwójki, wówczas wartość zmiennej `n` będzie parzysta za każdym razem, gdy pętla zostanie wykonana — aż do momentu, w którym dojdziemy do wartości 1. Przedstawiony w poprzednim akapicie przykład kończy się właśnie takim ciągiem, który zaczyna się od liczby 16.

Trudnym pytaniem jest to, czy program ten zakończy swoje działanie dla *wszystkich* wartości `n`. Tak się składa, że do tej pory nikomu nie udało się tego udowodnić *ani* udowodnić, że tak nie będzie! Więcej informacji na ten temat znajdziesz pod adresem: [https://pl.wikipedia.org/wiki/Problem\\_Collatza](https://pl.wikipedia.org/wiki/Problem_Collatza).

## Generowanie tablic

Pętle nadają się doskonale do generowania i wyświetlania danych tabelarycznych. Zanim komputery stały się łatwo dostępne, ludzie musieli ręcznie obliczać wartości logarytmów, sinusów i cosinusów oraz innych powszechnie wykorzystywanych funkcji matematycznych. Aby ułatwić to zadanie, tworzono księgi zawierające tabele, w których można było odnaleźć wartości różnych funkcji. Ręczne

tworzenie tych tabel było zadaniem bardzo długotrwałym i nudnym, a wyniki były często obarczone wieloma błędami.

Gdy na scenie pojawiły się komputery, jedną z pierwszych reakcji było: „To wspaniale! Możemy skorzystać z komputera, aby generować tablice matematyczne, dzięki czemu unikniemy błędów”. Okazało się to prawdą (w większości przypadków), ale było krótkowzroczne. Niewiele później komputery upowszechniły się w takim stopniu, że drukowane tablice stały się reliktem przeszłości i po prostu przestały być potrzebne.

Mimo to w przypadku niektórych operacji komputery wykorzystują tabele wartości w celu uzyskania przybliżonej odpowiedzi, a następnie wykonują obliczenia, aby zwiększyć dokładność otrzymanego pierwotnie wyniku. Czasami okazywało się, że w tych podstawowych tabelach występowały pewne błędy, a najstraszniejszym przypadkiem tego rodzaju była tabela wykorzystywana przez oryginalne procesory Pentium firmy Intel w operacjach dzielenia liczb zmiennoprzecinkowych (więcej informacji na ten temat w języku angielskim znajdziesz pod adresem: [https://en.wikipedia.org/wiki/Pentium\\_FDIV\\_bug](https://en.wikipedia.org/wiki/Pentium_FDIV_bug)).

Choć tabela logarytmów nie jest już tak przydatna, jak niegdyś bywała, w dalszym ciągu nadaje się doskonale do zaprezentowania jako przykład wykorzystania iteracji. Przedstawiona poniżej pętla umożliwia wyświetlenie tabeli zawierającej w lewej kolumnie ciąg kolejnych liczb całkowitych oraz ich logarytmów naturalnych w prawej:

```
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + "    " + Math.log(x));
    i = i + 1;
}
```

Uruchomienie tej pętli spowoduje wyświetlenie na ekranie następujących danych:

```
1.0    0.0
2.0    0.6931471805599453
3.0    1.0986122886681098
4.0    1.3862943611198906
5.0    1.6094379124341003
6.0    1.791759469228055
7.0    1.9459101490553132
8.0    2.0794415416798357
9.0    2.1972245773362196
```

Metoda `Math.log` umożliwia obliczenie logarytmu naturalnego, a więc logarytmu o podstawie  $e$ . W przypadku zastosowań typowo informatycznych często używa się logarytmów o podstawie 2. Aby je obliczyć, możemy skorzystać z przedstawionego poniżej wzoru:

$$\log_2 x = \frac{\log_e x}{\log_e 2}$$

Naszą pętlę można zatem zmodyfikować tak, jak zostało to pokazane poniżej:

```
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + "    " + Math.log(x) / Math.log(2));
    i = i + 1;
}
```

Wynik działania programu będzie dzięki temu następujący:

```
1.0 0.0
2.0 1.0
3.0 1.5849625007211563
4.0 2.0
5.0 2.321928094887362
6.0 2.584962500721156
7.0 2.807354922057604
8.0 3.0
9.0 3.1699250014423126
```

Przy każdym przejściu przez pętlę do wartości przechowywanej przez zmienną `x` dodajemy jeden, dlatego otrzymujemy tu ciąg arytmetyczny. Gdybyśmy zamiast dodawania zastosowali mnożenie przez jakąś liczbę, otrzymalibyśmy ciąg geometryczny. Zmodyfikowana pętla ma następującą postać:

```
final double LOG2 = Math.log(2);
int i = 1;
while (i < 10) {
    double x = (double) i;
    System.out.println(x + " " + Math.log(x) / LOG2);
    i = i * 2;
}
```

W pierwszej linii widocznego powyżej kodu zmiennej `final` o nazwie `LOG2` została przypisana wartość zwrócona przez wywołanie metody `Math.log(2)`; zrobiliśmy to, aby uniknąć ciągłego obliczania tej wartości od nowa. W ostatniej linii w ciele pętli wykonywane jest mnożenie zmiennej `x` przez wartość 2. Wynik działania pętli będzie następujący:

```
1.0 0.0
2.0 1.0
4.0 2.0
8.0 3.0
16.0 4.0
32.0 5.0
64.0 6.0
```

Widoczna powyżej tabela przedstawia kolejne potęgi liczby dwa oraz wartości ich logarytmu o podstawie 2. Tablice logarytmów może i nie są już do niczego potrzebne, ale każdemu programiście na pewno bardzo przyda się znajomość potęg liczby dwa!

## Hermetyzacja i uogólnianie

W podrozdziale zatytułowanym „Pisanie metod”, wchodzącym w skład rozdziału 6., przedstawiliśmy sposób pisania programów nazywany programowaniem przyrostowym. W niniejszym podrozdziale zaprezentujemy inny proces **tworzenia oprogramowania** (ang. *program development*), określany jako „hermetyzacja i uogólnianie”. Kroki tego procesu są następujące:

1. Napisz kilka linii kodu w metodzie `main` lub innej metodzie, a następnie przetestuj ich działanie.
2. Jeśli działają prawidłowo, opakuj je w nową metodę, a następnie ponownie przetestuj.
3. Jeśli się da, zastąp wartości literałów zmiennymi i parametrami.

Drugi z wymienionych powyżej kroków nazywany jest **hermetryzacją** (ang. *encapsulation*), a trzeci — **uogólnianiem** lub **generalizacją** (ang. *generalization*).

Aby zademonstrować przebieg tego procesu, opracujemy metody, za pomocą których można wyświetlić tabliczkę mnożenia. Pętla wyświetlająca wielokrotności liczby dwa w jednej linii została pokazana poniżej:

```
int i = 1;
while (i <= 6) {
    System.out.printf("%4d", 2 * i);
    i = i + 1;
}
System.out.println();
```

W pierwszej linii powyższego kodu zostaje zainicjalizowana zmienna o nazwie *i*, która spełnia tu rolę **zmiennej pętli** (ang. *loop variable*); wartość tej zmiennej rośnie od 1 do 6 wraz z każdym kolejnym wykonaniem pętli, a gdy osiąga 7, pętla zostaje przerwana.

Przy każdym przejściu przez pętlę na ekranie wyświetlana jest wartość  $2 * i$  z wypełnieniem znakami spacji, dzięki czemu zajmuje zawsze pole o szerokości czterech znaków. Z racji tego, że korzystamy tu z metody `System.out.printf`, cały komunikat wyjściowy wyświetlany jest w jednej linii.

Po instrukcji pętli dodane zostało wywołanie metody `println`, którego celem jest zakończenie linii i wyświetlenie znaku nowej linii. Pamiętaj, że w przypadku niektórych środowisk na ekranie nie pojawiają się żadne dane, dopóki linia nie zostanie zakończona.

Wykonanie powyższej pętli na tym etapie spowoduje wyświetlenie na ekranie następujących danych:

```
2  4  6  8 10 12
```

Kolejnym krokiem jest „enkapsulacja” naszego kodu w nowej metodzie. Może ona wyglądać tak, jak zostało to pokazane poniżej:

```
public static void printRow() {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", 2 * i);
        i = i + 1;
    }
    System.out.println();
}
```

Następnie zastępujemy stałą wartość 2 parametrem o nazwie *n*. Krok ten jest określany mianem „uogólniania”, ponieważ dzięki niemu metoda staje się bardziej ogólna (mniej specyficzna). Nowa postać metody została przedstawiona poniżej:

```
public static void printRow(int n) {
    int i = 1;
    while (i <= 6) {
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}
```

Wywołanie tej metody z argumentem 2 spowoduje wyświetlenie na ekranie dokładnie tego samego co wcześniej. W przypadku podania wartości 3 dane wyjściowe będą następujące:

```
3 6 9 12 15 18
```

Natomiast gdy podamy argument 4, na ekranie pojawi się następujący ciąg liczb:

```
4 8 12 16 20 24
```

Na tym etapie prawdopodobnie domyślasz się już, w jaki sposób wyświetlimy naszą tabliczkę mnożenia: zrobimy to, wielokrotnie wywołując metodę `printRow` z różnymi argumentami. W gruncie rzeczy posłużymy się tu kolejną pętlą w celu wykonania iteracji po wierszach tabliczki, tak jak zostało to pokazane poniżej:

```
int i = 1;
while (i <= 6) {
    printRow(i);
    i = i + 1;
}
```

Dane wyświetlone na ekranie będą miały następującą postać:

```
1 2 3 4 5 6
2 4 6 8 10 12
3 6 9 12 15 18
4 8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
```

Zastosowany w metodzie `printRow` specyfikator `%4d` powoduje, że dane wyjściowe są wyrównane w pionie niezależnie od tego, czy wyświetlane liczby są jedno-, czy dwucyfrowe.

Na koniec zajmiemy się hermetyzacją drugiej pętli w oddzielnej metodzie, tak jak zostało to zaprezentowane poniżej:

```
public static void printTable() {
    int i = 1;
    while (i <= 6) {
        printRow(i);
        i = i + 1;
    }
}
```

Jednym z większych wyzwań związanych z programowaniem, zwłaszcza w przypadku osób początkujących, okazuje się opracowanie sposobu podziału programu na metody. Proces hermetyzacji i uogólniania umożliwia Ci projektowanie rozwiązania wraz z jego rozwojem.

## Więcej uogólniania

Przedstawiona powyżej wersja metody `printTable` zawsze wyświetla sześć wierszy tabliczki mnożenia. Możemy ją uogólnić, zastępując literał 6 parametrem, tak jak zostało to przedstawione poniżej:

```
public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
```

```

        printRow(i);
        i = i + 1;
    }
}

```

Dane wyjściowe wygenerowane przez tę metodę wywołaną z argumentem o wartości 7 będą wyglądały następująco:

```

1  2  3  4  5  6
2  4  6  8 10 12
3  6  9 12 15 18
4  8 12 16 20 24
5 10 15 20 25 30
6 12 18 24 30 36
7 14 21 28 35 42

```

Jest już nieco lepiej, ale nadal widać tu pewien problem: program zawsze wyświetla tę samą liczbę kolumn. Możemy dokonać dalej idącego uogólnienia, dodając jeszcze jeden parametr do metody `printRow`, tak jak zostało to pokazane poniżej:

```

public static void printRow(int n, int cols) {
    int i = 1;
    while (i <= cols) {
        System.out.printf("%4d", n * i);
        i = i + 1;
    }
    System.out.println();
}

```

Teraz metoda `printRow` ma dwa parametry: `n` to wartość, której wielokrotności mają być wyświetlone, a `cols` to liczba kolumn. Jako że dodaliśmy parametr w definicji metody `printRow`, musimy też zmienić tę linię metody `printTable`, w której ją wywołujemy, jak zostało to przedstawione poniżej:

```

public static void printTable(int rows) {
    int i = 1;
    while (i <= rows) {
        printRow(i, rows);
        i = i + 1;
    }
}

```

Podczas wykonywania tej linii kodu sprawdzana jest zmienna `rows` i jej wartość (w omawianym powyżej przypadku jest to 7) jest przekazywana jako argument wywołania. W metodzie `printRow` wartość ta przypisywana jest do parametru `cols`. W wyniku tego liczba kolumn jest taka sama jak liczba wierszy, dzięki czemu otrzymujemy kwadratową tablicę o wymiarach 7×7 pozycji, jak widać to poniżej:

```

1  2  3  4  5  6  7
2  4  6  8 10 12 14
3  6  9 12 15 18 21
4  8 12 16 20 24 28
5 10 15 20 25 30 35
6 12 18 24 30 36 42
7 14 21 28 35 42 49

```

Gdy uogólnisz metodę we właściwy sposób, często odkrywasz, że oferuje ona możliwości, których oryginalnie nie planowałeś. Możesz na przykład zauważyć, że tabliczka mnożenia jest symetryczna;

z uwagi na to, że  $ab = ba$ , wszystkie pozycje pojawiają się w niej dwukrotnie. Możesz zaoszczędzić trochę atramentu niezbędnego do wydrukowania swojej tabliczki, eliminując powtórzenia. W tym celu musiałbyś zmienić tylko jedną linię w kodzie metody `printTable`, tak jak zostało to pokazane poniżej:

```
printRow(i, i);
```

Dzięki temu długość każdego wiersza stanie się taka sama jak jego numer. Wynikiem będzie zatem trójkątna tabliczka mnożenia o następującej postaci:

```
1
2  4
3  6  9
4  8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
```

Uogólnianie sprawia, że kod staje się bardziej uniwersalny, zwiększa się szansa jego ponownego wykorzystania, a czasami również łatwiej się go pisze.

## Instrukcja for

Pętle, które do tej pory pisaliśmy, mają kilka cech wspólnych. Zaczynają się od inicjalizacji zmiennej, mają warunek, który jest uzależniony od tej zmiennej, a wewnątrz pętli dzieje się coś, co powoduje zmianę jej wartości. Ten rodzaj pętli występuje tak powszechnie, że istnieje inna instrukcja umożliwiająca jej tworzenie w bardziej zwięzły sposób; jest nią pętla `for`.

Korzystając z niej, moglibyśmy na przykład zapisać metodę `printTable` następująco:

```
public static void printTable(int rows) {
    for (int i = 1; i <= rows; i = i + 1) {
        printRow(i, rows);
    }
}
```

W przypadku pętli `for` w nawiasie występują trzy elementy, które są rozdzielone znakami średnika. Są to: inicjalizator, warunek oraz aktualizacja.

1. *Inicjalizator* wykonywany jest tylko raz, na samym początku działania pętli.
2. *Warunek* jest sprawdzany w każdym przejściu pętli. Jeśli ma wartość `false`, pętla się kończy. W przeciwnym przypadku ciało pętli jest wykonywane (ponownie).
3. Na koniec każdej iteracji wykonywana jest *aktualizacja* i następuje powrót do kroku 2.

Pętlę `for` zwykle łatwiej się czyta, ponieważ w jej przypadku wszystkie instrukcje związane ze sterowaniem działaniem pętli umieszczane są w jej nagłówku.

Istnieje jeszcze jedna ważna różnica pomiędzy pętlami `for` i `while`: gdy zadeklarujesz zmienną w inicjalizatorze, zmienna ta istnieje tylko wewnątrz pętli `for`. Za przykład weźmy tu wersję metody `printRow`, w której wykorzystana została instrukcja pętli `for`, tak jak zostało to przedstawione poniżej:

```
public static void printRow(int n, int cols) {
    for (int i = 1; i <= cols; i = i + 1) {
```



```

        System.out.printf("%4d", n * i);
    }
    System.out.println(i);    // błąd kompilatora
}

```

W ostatniej linii zaprezentowanej powyżej metody podjęta zostaje próba wyświetlenia wartości zmiennej `i` (bez żadnego konkretnego powodu poza tym, że ma to posłużyć demonstracji problemu). Próba ta się nie powiedzie, ponieważ na tym etapie zmienna już nie istnieje. Jeśli zmienna pętli ma być wykorzystywana poza jej wnętrzem, trzeba ją również zadeklarować na zewnątrz, tak jak zostało to pokazane poniżej:

```

public static void printRow(int n, int cols) {
    int i;
    for (i = 1; i <= cols; i = i + 1) {
        System.out.printf("%4d", n * i);
    }
    System.out.println(i);
}

```

Przypisania w rodzaju widocznego powyżej `i = i + 1` nie pojawiają się zwykle w pętlach `for`, ponieważ język Java zapewnia bardziej zwięzły sposób zwiększania i zmniejszania wartości zmiennej o jeden. Chodzi tu konkretnie o operatory **inkrementacji**, czyli operator `++`, oraz **dekrementacji**, czyli operator `--`. Działanie pierwszego z nich jest równoważne zastosowaniu wyrażenia `i = i + 1`, zaś drugie odnosi dokładnie taki sam skutek jak wyrażenie `i = i - 1`.

Jeśli chcesz zinkrementować lub zdekrementować zmienną o wartość inną niż 1, możesz użyć operatorów `+=` oraz `-=`. Na przykład wyrażenie `i += 2` zwiększa wartość zmiennej `i` o 2.

## Pętla do-while

Instrukcje `while` oraz `for` umożliwiają tworzenie **pętli sprawdzających warunek na początku** (ang. *pretest loops*); oznacza to, że w ich przypadku warunek jest sprawdzany na samym początku działania i na początku każdego przejścia przez pętlę.

Język Java zapewnia również możliwość pisania **pętli sprawdzających warunek na końcu** (ang. *posttest loops*), którą realizuje się za pomocą instrukcji `do-while`. Ten rodzaj pętli przydaje się, gdy trzeba przynajmniej raz wykonać ciało pętli.

Na przykład w podrozdziale zatytułowanym „Sprawdzanie danych wejściowych”, wchodzącym w skład rozdziału 5., korzystaliśmy z instrukcji `return`, aby uniknąć wczytywania niepoprawnych danych wejściowych podanych przez użytkownika. Zamiast tego możemy zastosować instrukcję `do-while`, aby odczytywać dane wejściowe, dopóki są one poprawne, tak jak zostało to pokazane poniżej:

```

Scanner in = new Scanner(System.in);
boolean okay;
do {
    System.out.print("Wprowadź liczbę: ");
    if (in.hasNextDouble()) {
        okay = true;
    } else {
        okay = false;
    }
    String word = in.next();
}

```

```
        System.err.println(word + " nie jest liczbą");
    }
} while (!okay);
double x = in.nextDouble();
```

Choć kod ten wydaje się trochę bardziej skomplikowany, wykonywane są tu w gruncie rzeczy raptem trzy podstawowe kroki:

1. Na ekranie wyświetlana jest prośba o podanie danych.
2. Wprowadzone dane są sprawdzane; jeśli są niepoprawne, wyświetlany jest komunikat błędu i procedura zaczyna się od nowa.
3. Odczytywane są dane wejściowe.

W powyższym fragmencie kodu wykorzystywana jest flaga o nazwie `okay`, która wskazuje, czy ciało pętli należy powtórzyć. Gdy w wyniku wywołania metody `hasNextDouble()` zwrócona zostanie wartość `false`, niewłaściwe dane wejściowe zostają odrzucone za pomocą wywołania metody `next()`. Następnie na ekranie wyświetlany jest komunikat błędu przy użyciu strumienia `System.err`. Działanie pętli zostaje zakończone, gdy w wyniku wywołania metody `hasNextDouble()` otrzymana zostanie wartość `true`.

## Instrukcje `break` i `continue`

W niektórych sytuacjach nie sprawdzi się najlepiej ani sprawdzanie warunku na początku pętli, ani sprawdzanie go na jej końcu, ponieważ żaden z dostępnych rodzajów pętli nie będzie w stanie zaferować Ci dokładnie tego, czego oczekujesz. W ostatnim przykładzie przedstawionym w poprzednim podrozdziale sprawdzenie warunku musiało zachodzić w samym środku pętli. W związku z tym trzeba było zastosować specjalną flagę oraz zagnieżdżoną instrukcję warunkową `if-else`.

Prostszym sposobem poradzenia sobie z tego rodzaju problemem jest skorzystanie z instrukcji `break`. Gdy wykonanie osiągnie punkt, w którym występuje ta instrukcja, bieżąca pętla zostaje natychmiast opuszczona. Przyjrzyj się przedstawionemu poniżej fragmentowi kodu:

```
Scanner in = new Scanner(System.in);
while (true) {
    System.out.print("Wprowadź liczbę: ");
    if (in.hasNextDouble()) {
        break;
    }
    String word = in.next();
    System.err.println(word + " nie jest liczbą");
}
double x = in.nextDouble();
```

Użycie wartości `true` w roli warunku pętli `while` jest równoznaczne z utworzeniem pętli, która ma trwać wiecznie lub — jak jest w tym przypadku — wykonywać się aż do momentu wystąpienia instrukcji `break`.

Oprócz instrukcji `break`, którą można zastosować w obrębie pętli, język Java oferuje też instrukcję `continue`, która powoduje przeniesienie wykonania do kolejnej iteracji. Na przykład przedstawiony poniżej kod odczytuje z klawiatury wartości całkowite i oblicza ich bieżącą sumę. Zastosowana tu

instrukcja `continue` powoduje pominięcie wszelkich wartości ujemnych, które mogą zostać wprowadzone przez użytkownika:

```
Scanner in = new Scanner(System.in);
int x = -1;
int sum = 0;
while (x != 0) {
    x = in.nextInt();
    if (x <= 0) {
        continue;
    }
    System.out.println("Dodaję " + x);
    sum += x;
}
```

Choć instrukcje `break` i `continue` zapewniają Ci większą kontrolę nad przebiegiem wykonania pętli, mogą sprawić, że kod trudniej będzie zrozumieć i zdebugować. Dlatego staraj się z nich korzystać bardzo oszczędnie.

## Słownictwo

*iteracja:*

Wielokrotne wykonywanie sekwencji instrukcji.

*pętla:*

Instrukcja, która wielokrotnie wykonuje sekwencję instrukcji.

*ciało pętli:*

Instrukcje znajdujące się wewnątrz pętli.

*pętla nieskończona:*

Pętla, której warunek jest zawsze prawdziwy.

*tworzenie oprogramowania:*

Proces pisania programów. Do tej pory mieliśmy tu do czynienia z dwoma technikami pisania kodu: programowaniem przyrostowym oraz hermetyzacją i uogólnianiem.

*hermetyzacja:*

Opakowanie sekwencji instrukcji w metodę.

*uogólnianie:*

Zastępowanie czegoś niepotrzebnie specyficznego (takiego jak wartości stałe) czymś odpowiednio ogólnym (takim jak zmienna lub parametr).

*zmienna pętli:*

Zmienna, która jest inicjalizowana, sprawdzana i aktualizowana w celu sterowania wykonaniem pętli.

*inkrementacja:*

Zwiększenie wartości zmiennej.

*dekrementacja:*

Zmniejszenie wartości zmiennej.

*pętla sprawdzająca warunek na początku:*

Pętla, w przypadku której warunek sprawdzany jest przed każdą iteracją.

*pętla sprawdzająca warunek na końcu:*

Pętla, w przypadku której warunek sprawdzany jest po każdej iteracji.

## Ćwiczenia

Kod związany z niniejszym rozdziałem znajduje się w katalogu *r07* archiwum kodów dołączonego do tej książki. Informacje na temat tego, jak pobrać to archiwum, znajdziesz w należącym do wstępu podrozdziale „Używanie przykładowych kodów”. Zanim zaczniesz wykonywać przedstawione poniżej ćwiczenia, radzimy Ci skompilować i uruchomić przykłady zaprezentowane w tym rozdziale.

Jeśli nie zapoznałeś się jeszcze z treścią podrozdziału zatytułowanego „Uruchamianie narzędzia Checkstyle”, który wchodzi w skład dodatku A, nadszedł dobry moment, aby to zrobić. Opisany w nim został Checkstyle, czyli narzędzie umożliwiające przeprowadzanie wszechstronnych analiz kodu źródłowego.

### Ćwiczenie 7.1

Przyjrzyj się przedstawionym poniżej metodom, a następnie wykonaj polecenia i odpowiedz na pytanie zamieszczone pod kodem:

```
public static void main(String[] args) {
    loop(10);
}

public static void loop(int n) {
    int i = n;
    while (i > 1) {
        System.out.println(i);
        if (i % 2 == 0) {
            i = i / 2;
        } else {
            i = i + 1;
        }
    }
}
```

1. Utwórz tabelę przedstawiającą wartości zmiennych *i* oraz *n* w trakcie wykonywania metody *loop*. Tabela ta ma zawierać po jednej kolumnie dla każdej ze zmiennych oraz po jednym wierszu dla każdej iteracji.
2. Napisz, jak wyglądają dane wyświetlone przez ten program na ekranie.
3. Czy jesteś w stanie udowodnić, że przedstawiona powyżej pętla zakończy swoje działanie dla dowolnej dodatniej wartości zmiennej *n*?

### Ćwiczenie 7.2

Powiedzmy, że Twoim zadaniem jest znaleźć pierwiastek kwadratowy podanej liczby ( $a$ ). Jednym ze sposobów, aby to zrobić, jest rozpoczęcie od zgrubnego oszacowania wartości szukanej liczby ( $x_0$ ), a następnie poprawienie tego przybliżenia przy użyciu następującego wzoru:

$$x_1 = (x_0 + a/x_0)/2$$

Na przykład gdy chcemy znaleźć pierwiastek kwadratowy liczby 9 i zaczniemy od  $x_0 = 6$ , wówczas  $x_1 = (6+9/6)/2 = 3,75$ , co jest już wartością bardziej zbliżoną do właściwego wyniku. Procedurę tę możemy powtarzać, wykorzystując wartość  $x_1$  w celu obliczenia wartości  $x_2$  i tak dalej. W naszym przypadku  $x_2 = 3,075$ , a  $x_3 = 3,00091$ . Widać zatem, że wraz z kolejnymi wartościami szybko zbliżamy się do właściwej odpowiedzi.

Napisz metodę o nazwie `squareRoot`, która przyjmuje argument typu `double` i zwraca przybliżenie pierwiastka kwadratowego tej wartości przy użyciu opisanego powyżej sposobu. Nie możesz tu korzystać z metody `Math.sqrt`.

Jako swojego wstępnego przybliżenia powinieneś użyć wartości  $a/2$ . W metodzie powinieneś iterować aż do momentu, w którym dwa kolejne oszacowania będą się różniły o mniej niż 0,0001. Do obliczenia bezwzględnej wartości różnicy możesz użyć metody `Math.abs`.

### Ćwiczenie 7.3

Zadanie przedstawione w ćwiczeniu 6.9 polegało na napisaniu rekurencyjnej wersji metody `power`, która przyjmuje wartość typu `double` pod postacią parametru  $x$  oraz wartość całkowitą jako parametr  $n$  i zwraca wartość wyrażenia  $x^n$ . Teraz napisz iteracyjną wersję metody, która będzie wykonywała to samo obliczenie.

### Ćwiczenie 7.4

W podrozdziale „Więcej rekurencji”, wchodzącym w skład rozdziału 6., zaprezentowana została metoda, która oblicza wartość funkcji silnia dla danej liczby naturalnej. Napisz iteracyjną wersję metody `factorial`.

### Ćwiczenie 7.5

Sposobem umożliwiającym obliczenie wartości wyrażenia  $e^x$  jest skorzystanie z rozwinięcia nieskończonego szeregu o postaci:

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \dots$$

Element (wyraz) szeregu o indeksie  $i$  ma postać:  $x^i/i!$ .

1. Napisz metodę o nazwie `myexp`, która przyjmuje wartości za pośrednictwem parametrów  $x$  oraz  $n$  i szacuje wartość wyrażenia  $e^x$ , dodając pierwszych  $n$  wyrazów przedstawionego powyżej szeregu. Możesz tu wykorzystać metodę `factorial` przedstawioną w podrozdziale „Więcej rekurencji”, wchodzącym w skład rozdziału 6., lub swoją iteracyjną wersję tej metody, o której była mowa w poprzednim ćwiczeniu.

2. Możesz poprawić wydajność działania tej metody, gdy zdasz sobie sprawę z faktu, że licznik ułamka każdego z wyrazów jest równy licznikowi poprzedniego przemnożonego przez wartość zmiennej  $x$ , zaś jego mianownik jest równy mianownikowi poprzedniego wyrazu przemnożonego przez wartość zmiennej  $i$ . Skorzystaj z tej wskazówki, aby wyeliminować konieczność stosowania metod `Math.pow` oraz `factorial`, i sprawdź, czy uda Ci się w ten sposób uzyskać ten sam wynik.
3. Napisz metodę o nazwie `check`, która ma parametr  $x$  i wyświetla wartości  $x$ , `myexp(x)` oraz `Math.exp(x)`. Dane wyjściowe wygenerowane przez Twój program powinny wyglądać mniej więcej tak:
 

```
1.0 2.708333333333333 2.718281828459045
```

 Możesz tu zastosować sekwencję ucieczki `"\t"` w celu umieszczenia znaku tabulacji pomiędzy kolumnami tabeli.
4. Pozmieniaj liczbę wyrazów w szeregu (chodzi o drugi argument wywołania metody `myexp`, który jest jej przekazywany z poziomu metody `check`) i przyjrzyj się temu, jak wpływa to na dokładność otrzymywanych wyników. Dostrajaj tę wartość aż do momentu, gdy szacowana przez Ciebie wartość będzie całkowicie zgodna z prawidłową odpowiedzią dla  $x$  równego 1.
5. Napisz w metodzie `main` pętlę, w której wywoływana będzie metoda `check` z wartościami 0.1, 1.0, 10.0 oraz 100.0. Jak zmienia się dokładność wyniku wraz ze zmianą wartości  $x$ ? Porównuj raczej liczbę cyfr, które się zgadzają, niż samą różnicę między wartościami rzeczywistą i szacowaną.
6. Dodaj w metodzie `main` pętlę, w której sprawdzane są wyniki działania metody `myexp` dla wartości -0.1, -1.0, -10.0 oraz -100.0. Skomentuj uzyskaną dokładność.

### Ćwiczenie 7.6

Jednym ze sposobów obliczania wartości funkcji  $\exp(-x^2)$  jest skorzystanie z rozwinięcia nieskończonego szeregu o postaci:

$$\exp(-x^2) = 1 - x^2 + x^4/2 - x^6/6 + \dots$$

Wyraz tego szeregu o indeksie  $i$  to:  $(-1)^i x^{2i} / i!$ . Napisz metodę o nazwie `gauss`, która przyjmuje  $x$  oraz  $n$  jako argumenty i zwraca sumę pierwszych  $n$  wyrazów tego szeregu. Nie powinieneś przy tym korzystać z metod `factorial` ani `pow`.

## A

adres, 41, 50  
akt wiary, 92, 95  
akumulator, 118, 122  
algorytm, 14, 21  
    wyszukiwania, 182  
alias, 122, 145  
argument  
    funkcji, 55, 60, 66  
    wiersza poleceń, 134  
atrybut, 142, 149  
AWT, 223, 226

## B

biblioteka, 50, 148  
błąd  
    czasu kompilacji, 33, 37, 231  
    czasu wykonania, 34, 37, 234  
    logiczny, 35, 231, 237  
    zaokrąglenia, 31, 37  
boolean, 79  
breakpoint, 217  
budowanie histogramu, 120  
bug, 14, 49, 241

## C

ciało pętli, 109  
ciąg  
    Fibonacciego, 93  
    pseudolosowy, 122  
    znakowy, 18  
czysta metoda, 164, 166  
czytanie dokumentacji, 62

## D

dane wejściowe, 74  
debugger, 217, 220  
debugowanie, 14, 20, 231  
deklaracja, 36  
deklarowanie zmiennych, 25  
dekompozycja funkcjonalna, 88, 94  
dekrementacja, 107, 110  
diagram  
    klas UML, 148, 149, 207  
    stanu, 27, 36, 63, 162, 185  
    stanu tablicy, 114  
    stosu, 62, 66, 76  
długość tablic, 117  
dodawanie  
    metod, 57  
    obiektów, 163  
    rekurencji, 190  
dokumentacja, 62, 65  
    klasy Scanner, 63  
dostęp do tablic, 114  
dziedziczenie, 196, 199, 206

## E

edytor tekstu, 211, 220  
egzemplarz, 155  
element, 122  
elipsa, 225  
enkapsulacja danych, 155

## F

flaga, 73, 79  
formatowanie  
  danych wyjściowych, 45  
  kodu, 19  
  łańcuchów znakowych, 133

## G

gałąź, 71, 79  
generalizacja, 103  
generowanie tablic, 100  
getter, 158, 166  
grafika 2D, 223

## H

hermetyzacja, 102, 109, 155, 165  
histogram, 119, 122

## I

IDE, integrated development environment, 211, 220  
identyczność, 166  
indeks, 114, 122  
informatyka, 14, 21  
inicjalizacja, 36  
inkrementacja, 107, 109  
instalacja programu DrJava, 211  
instancja, 155, 165  
instrukcja, 16, 22  
  break, 108  
  continue, 108  
  do-while, 107  
  for, 106  
  print, 16, 22  
  return, 74, 83  
  while, 99  
instrukcje  
  importu, 42, 50  
  warunkowe, 71, 79  
interfejs wiersza poleceń, 213, 220  
interpretacja, 21  
interpreter, 15  
iteracja, 99, 109

## J

JAR, 220  
Javadoc, 65, 67

JDK, Java Development Kit, 211, 220  
języki  
  niskiego poziomu, 15, 21  
  wysokiego poziomu, 15, 21  
JVM, Java Virtual Machine, 16, 211, 220

## K

klasa, 17, 22, 155, 165  
  ArrayList, 198  
  Card, 171, 172, 176, 177  
  CardCollection, 196, 200  
  Deck, 185  
  Eights, 203  
  Player, 201  
  Point, 141  
  Scanner, 42  
  System, 41  
  Time, 155, 163  
klasy  
  opakowujące, 133, 135  
  wbudowane, 44  
klient, 166  
kod  
  bajtowy, 15, 22  
  obiektowy, 15, 21  
  wykonywalny, 15, 22  
  źródłowy, 21  
kolejność wykonywania działań, 32, 37  
kolekcja, 196, 207  
komentarze, 17, 22  
  dokumentujące, 65  
  Javadoca, 65  
kompilacja, 21  
kompilator, 15, 231  
kompilowanie programu, 233  
kompletność Turinga, 95  
kompozycja, 33, 37, 56  
  metody, 87  
komunikaty błędów, 231  
konkatenacja, 37  
konstruktor, 156, 166  
kopiowanie tablic, 116  
krótkie spięcia, 79

## L, ł

liczby  
  binarne, 77  
  losowe, 118



- magiczne, 45, 51
- zmiennoprzecinkowe, 29

literał, 45, 51

łańcuchy znakowe, 17, 22, 72, 127

- formatowanie, 46, 51, 133
- klasy opakowujące, 133
- metoda indexOf, 131
- niezmiennosc, 128
- porównywanie, 132
- przechodzenie, 129

## M

martwy kod, 84, 94

maszyna wirtualna Javy, 16

metoda, 17, 22

- compareTo, 175
- countdown, 99
- equals, 162
- fillOval, 225
- indexOf, 131
- last, 198
- main, 58
- newLine, 58
- nextInt, 50
- popCard, 198
- println, 60, 240
- printTime, 62
- printTwice, 60
- toString, 161, 172

metody

- graficzne, 224
- instancji, 161, 166
- matematyczne, 55
- niezwracające wartości, 55
- opakowujące, 197, 208
- pomocnicze, 187, 191
- rekurencyjne, 75, 79, 181
- typu boolean, 89
- typu void, 94
- zwracające wartość, 83, 94

modulo, 51

modyfikator, 164, 166

## N

nadklasa, 199, 208

narzędzia programistyczne, 211

narzędzie Javadoc, 65, 67

nieskończona

- pętla, 235
- rekurencja, 235

niezmiennosc, 135

- łańcuchów znakowych, 128
- obiektów, 176

notacja

- kropkowa, 142, 149
- wielbłądzia, 57

## O

obiekty, 135, 141

- jako parametry, 142
- jako wartości zwracane, 143
- klasy Card, 171
- klasy Point, 141
- zawierające inne obiekty, 195
- zawierające tablice, 185
- zmiennie, 144

obliczanie

- metodą krótkiego spięcia, 70
- metodą zwarcia, 70

obwiednia, 227

oczyszczanie pamięci, 147, 149

odświeżanie, 147

operand, 29, 37

operator, 36

- [], 114
- ==, 162
- modulo, 47, 51
- new, 157

operatory

- arytmetyczne, 28
- działające na łańcuchach znakowych, 32
- logiczne, 70, 79
- przekierowania, 215, 220
- relacyjne, 69, 79

## P

pakiet, 41, 50

- java.awt, 223
- junit.framework, 219

pamięć, 147

panel Interactions, 212

parametr, 57, 60, 66

parsowanie, 34, 37, 135

- pętla, 100, 109
  - do-while, 107
  - for, 106
  - for rozszerzona, 121
  - while, 99
- pętle
  - nieskończone, 100, 109
  - sprawdzające warunek na końcu, 110
  - sprawdzające warunek na początku, 110
- pierwszy program, 16
- piksel, 225, 227
- pisanie
  - dokumentacji, 65
  - metod, 85
- podklasa, 196, 207
- podłańcuchy znakowe, 130
- poła, 142
- pomoc, 240
- porównywanie łańcuchów znakowych, 132
- prawa De Morgana, 70, 79
- program, 13, 21
  - Checkstyle, 216
  - DrJava, 211, 212
  - JUnit, 218
  - WinMerge, 216
- programowanie, 13, 21
  - przyrostowe, 85, 94
  - z dołu do góry, 203, 208
  - z góry na dół, 187, 191
  - zorientowane obiektowo, 10, 145, 149
- programy
  - deterministyczne, 118, 122
  - niedeterministyczne, 122
  - przenośne, 15
- przechodzenie, 119, 122
  - przez łańcuchy znakowe, 129
  - przez tablice, 117
- przeciążanie, 88, 95
- przekazywanie parametrów, 60, 66
- przeliczanie stopni, 44, 46
- przenośność, 21
- przeptyw wykonania, 59, 66, 235
- przesłanianie metod, 161, 166
- przypadek podstawowy, 79
- przypisanie, 26, 36
- pseudokod, 187, 191
- punkt przerywania, 217, 220
- pusta tablica, 134, 135
- pusty
  - łańcuch znakowy, 130, 135
  - obiekt, 185

## R

- ramka, 66
- redukcja, 118, 122
- referencja, 114, 122
- rekurencja, 75, 79, 90, 181, 190
- rekurencyjne diagramy stosu, 76
- relacja
  - typu jest, 206, 208
  - typu posiada, 206, 208
- relacje pomiędzy klasami, 206
- RGB, 227
- rozdawanie kart, 200
- rozszerzona pętla for, 121, 123
- rozwiązywanie problemu, 21
- równoważność, 166
- rusztowanie, 86, 94
- rztowanie typów, 47, 51

## S, Ś

- scalanie, 188
  - talii, 190
- sekwencja ucieczki, 18
- setter, 158, 166
- silnia, 91, 95
- składnia, 37
- słowo kluczowe, 36
  - extends, 219
  - null, 146
- sortowanie
  - przez scalanie, 188, 192
  - przez wybieranie, 187, 191
- specyfikator formatu, 46, 51
- sprawdzanie danych wejściowych, 74
- stała, 45
- stan, 36
- standard Unicode, 128
- stos, 62, 76
  - wywołania, 221
- struktura programu, 43
- strumień znaków, 50
- sygnatura, 67
  - metody, 64
- symbol wieloznaczny, 220
- system binarny, 80
- śląd stosu, 236
- śledzenie wykonania, 180, 217

## T

- tablica obiektów, 171, 177, 178
- tablice, 100, 113, 122
  - dostęp, 114
  - kopiowanie, 116
  - przechodzenie, 117
  - rozmiar, 117
  - tworzenie, 113
  - wyświetlanie, 115
- talie, 195
- tasowanie talii kart, 186
- terminal, 213
- test jednostkowy, 218, 221
- testowanie
  - przy użyciu narzędzia JUnit, 218
  - w wierszu poleceń, 214
- token, 51
- tworzenie
  - grafiki, 223
  - łańcuchów, 72, 79
  - oprogramowania, 102, 109
  - tablic, 113
- typ danych boolean, 79
- typy
  - błędów, 33
  - parametrów, 158
  - podstawowe, 36, 127, 135
  - zmiennoprzecinkowe, 37
  - zwracanej wartości, 83, 94

## U

- układ współrzędnych
  - graficznych, 224
  - kartezjańskich, 224
- układy, 195
- ukrywanie informacji, 156, 166
- UML, Unified Modeling Language, 149
- Unicode, 128, 135
- uogólnianie, 102, 104, 109
- uruchamianie narzędzia Checkstyle, 216

## W

- wartość, 36
  - null, 146
  - zwracana, 83, 94

- warunek, 71
- wejście, 41
- wrapper, 133, 197
- współrzędna, 227
- wybieranie, 187
- wyjątek, 34
  - ArithmeticException, 236
  - ArrayIndexOutOfBoundsException, 236
  - FileNotFoundException, 236
  - InputMismatchException, 74
  - NullPointerException, 236
  - StackOverflowError, 236
  - StringIndexOutOfBoundsException, 129
- wyjście, 41
- wyrażenie, 29, 37
- wyszukiwanie, 117, 122
  - binarne, 179, 182
  - sekwencyjne, 179, 182
- wyświetlanie
  - łańcuchów znakowych, 17
  - obiektów, 160
  - tablic, 115
  - zmiennych, 28
- wywoływanie metod, 55, 66

## Z

- zagnieżdżanie, 72, 79
- zaokrąglenia, 31
- zasłanianie, 166
- zaśleпка, 86, 94
- zintegrowane środowisko programistyczne, IDE, 211, 220
- zliczanie, 119
- zmiennie, 25, 36
  - flagi, 73
  - instancji, 156, 165
  - klasy, 174, 182
  - lokalne, 66
  - pętli, 103, 109
  - tymczasowe, 84, 94
- znacznik, 95
  - Javadoc, 90
- znak nowej linii, 18, 22
- znaki, 127



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

## Java – już wkrótce zaczniesz myśleć jak informatyk!

Zacznij myśleć jak programista! Naucz się łączyć umiejętności płynące z różnych dziedzin: matematyki, inżynierii i nauk przyrodniczych. Różnorodna wiedza ułatwi Ci pokonywanie przeszkód w pracy programisty – dzięki niej zdefiniujesz problem i sformułujesz jasne i precyzyjne rozwiązanie. Jak się okazuje, te wszystkie trudne umiejętności można sobie przyswoić, ucząc się programowania. Znajomość języka Java będzie dodatkową korzyścią – jest to język o ugruntowanej pozycji, lubiany, dojrzały i najwycyzejniej bardzo przydatny.

Trzymasz w ręku zwięzły podręcznik programowania napisany z myślą o osobach mających niewielkie lub zerowe doświadczenie w tej materii. Przedstawiono w nim najbardziej podstawowe zagadnienia, a poszczególne koncepcje zaprezentowano w logicznej kolejności. Sam język Java nie został może wyczerpująco opisany, jednak ważniejsze konstrukcje, strategie i techniki programistyczne są omówione w bardzo zrozumiały i przystępny sposób. Poszczególne koncepcje zilustrowano praktycznymi przykładami kodu. Ten starannie przemyślany układ treści sprawi, że „myślenie jak informatyk” bardzo szybko przestanie być Twoim problemem!

**Allen B. Downey** jest profesorem informatyki na uczelni Olin College of Engineering. Prowadził zajęcia na uczelniach: Wellesley College, Colby College i U.C. Berkeley; na tej ostatniej uzyskał tytuł doktora. W latach 2009 – 2010 pracował jako zaproszony naukowiec w firmie Google.

**Chris Mayfield** jest doktorem informatyki. Pracuje na uczelni James Madison University, gdzie zajmuje się badaniami nad edukacją informatyczną oraz rozwojem zawodowym.

Dzięki tej książce:

- poznasz najważniejsze koncepcje informatyki i programowania
- nauczysz się używać różnych typów danych i sterować przebiegiem programu
- dowiesz się, czym są tablice, ciągi znakowe, i nauczysz się nimi posługiwać
- zrozumiesz zasady programowania obiektowego

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

**Helion**

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

Sprawdź najnowsze promocje:  
● <http://helion.pl/promocje>  
Książki najchętniej czytane:  
● <http://helion.pl/bestsellery>  
Zamów informacje o nowościach:  
● <http://helion.pl/novosci>

ISBN 978-83-283-3006-1



9 788328 330061

Informatyka w najlepszym wydaniu

cena: 49,00 zł