

>>> PROJEKTOWANIE  
OPROGRAMOWANIA DLA  
ZUPEŁNIE POCZĄTKUJĄCYCH

WYDANIE V



TONY GADDIS

Tytuł oryginału: Starting Out with Programming Logic and Design (5th Edition)

Tłumaczenie: Wojciech Moch z wykorzystaniem fragmentów książki „Projektowanie oprogramowania dla zupełnie początkujących. Owoce programowania. Wydanie IV” w tłumaczeniu Krzysztofa Braunera

ISBN: 978-83-283-5566-8

Authorized translation from the English language edition, entitled: STARTING OUT WITH PROGRAMMING LOGIC AND DESIGN, 5th Edition, by GADDIS TONY, published by Pearson Education, Inc, publishing as Pearson.

Copyright © 2019 by Pearson Education, Inc. or its affiliates.

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 Pearson Education, Inc.  
Polish language edition published by HELION S.A. Copyright © 2020.

Microsoft® Windows®, and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopowanie 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 właścicielami.

Autor oraz Helion SA 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 Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/pkpro5.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

[http://helion.pl/user/opinie/pkpro5\\_ebook](http://helion.pl/user/opinie/pkpro5_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

# Spis treści

<b>Wstęp .....</b>	<b>13</b>
<b>Podziękowania .....</b>	<b>19</b>
<b>O autorze .....</b>	<b>21</b>
<b>Rozdział 1. Wstępne informacje na temat komputerów i programowania .....</b>	<b>23</b>
1.1    Wstęp .....	23
1.2    Sprzęt .....	24
1.3    W jaki sposób komputer przechowuje dane .....	30
1.4    W jaki sposób działa program .....	35
1.5    Rodzaje oprogramowania .....	44
Pytania kontrolne .....	45
<b>Rozdział 2. Dane wejściowe, przetwarzanie i dane wyjściowe ....</b>	<b>51</b>
2.1    Projektowanie programu .....	51
2.2    Dane wejściowe, dane wyjściowe i zmienne .....	58
2.3    Przypisywanie wartości do zmiennych i wykonywanie obliczeń ....	69
<b>W CENTRUM UWAGI</b> Obliczanie opłat za dodatkowe minuty .....	73
<b>W CENTRUM UWAGI</b> Obliczanie procentów .....	74
<b>W CENTRUM UWAGI</b> Obliczanie średniej .....	79
<b>W CENTRUM UWAGI</b> Zamiana wzoru matematycznego na wyrażenie ...	82
2.4    Deklarowanie zmiennych i typy danych .....	84
2.5    Stałe nazwane .....	90
2.6    Ręczne śledzenie programu .....	91
2.7    Dokumentowanie programu .....	93
<b>W CENTRUM UWAGI</b> Korzystanie ze stałych nazwanych, konwencje zapisu i komentarze .....	94
2.8    Projektowanie pierwszego programu .....	97
2.9    Rzut oka na języki Java, Python i C++ .....	100

## 6 Spis treści

Pytania kontrolne .....	121
Ćwiczenia z wykrywania błędów .....	126
Ćwiczenia programistyczne .....	127
<b>Rozdział 3. Moduły .....</b>	<b>131</b>
3.1    Moduły — informacje wstępne .....	131
3.2    Definiowanie i wywoływanie modułów .....	134
<b>W CENTRUM UWAGI</b> Definiowanie i wywoływanie modułów .....	140
3.3    Zmienne lokalne .....	145
3.4    Przekazywanie argumentów do modułów .....	148
<b>W CENTRUM UWAGI</b> Przekazywanie argumentu do modułu .....	153
<b>W CENTRUM UWAGI</b> Przekazywanie argumentu przez referencję .....	159
3.5    Zmienne globalne i stałe globalne .....	162
<b>W CENTRUM UWAGI</b> Korzystanie ze stałych globalnych .....	165
3.6    Rzut oka na języki Java, Python i C++ .....	167
Pytania kontrolne .....	179
Ćwiczenia z wykrywania błędów .....	183
Ćwiczenia programistyczne .....	183
<b>Rozdział 4. Struktury warunkowe i logika boolowska .....</b>	<b>187</b>
4.1    Struktury warunkowe — informacje wstępne .....	187
<b>W CENTRUM UWAGI</b> Korzystanie z instrukcji If-Then .....	195
4.2    Struktury warunkowe podwójnego wyboru .....	198
<b>W CENTRUM UWAGI</b> Korzystanie z instrukcji If-Then-Else .....	199
4.3    Porównywanie ciągów znaków .....	202
4.4    Zagnieżdżone struktury warunkowe .....	208
<b>W CENTRUM UWAGI</b> Wielokrotne zagnieżdżenie struktur warunkowych .....	211
4.5    Struktura decyzyjna .....	215
<b>W CENTRUM UWAGI</b> Korzystanie ze struktury decyzyjnej .....	218
4.6    Operatory logiczne .....	221
4.7    Zmienne boolowskie .....	229
4.8    Rzut oka na języki Java, Python i C++ .....	230
Pytania kontrolne .....	242
Ćwiczenia z wykrywania błędów .....	246
Ćwiczenia programistyczne .....	247

Rozdział 5. <b>Struktury cykliczne .....</b>	<b>251</b>
5.1    Struktury cykliczne — wprowadzenie .....	251
5.2    Pętle warunkowe: While, Do-While i Do-Until .....	253
<b>W CENTRUM UWAGI</b> Projektowanie pętli While .....	257
<b>W CENTRUM UWAGI</b> Projektowanie pętli Do-While .....	266
5.3    Pętle licznikowe i instrukcja For .....	272
<b>W CENTRUM UWAGI</b> Projektowanie pętli licznikowej za pomocą instrukcji For .....	279
5.4    Obliczanie sumy bieżącej .....	289
5.5    Wartownik .....	293
<b>W CENTRUM UWAGI</b> Korzystanie z wartownika .....	293
5.6    Pętle zagnieżdżone .....	295
5.7    Rzut oka na języki Java, Python i C++ .....	298
Pytania kontrolne .....	308
Ćwiczenia z wykrywania błędów .....	311
Ćwiczenia programistyczne .....	312
Rozdział 6. <b>Funkcje .....</b>	<b>315</b>
6.1    Wprowadzenie do funkcji: generowanie liczb losowych .....	315
<b>W CENTRUM UWAGI</b> Korzystanie z liczb losowych .....	319
<b>W CENTRUM UWAGI</b> Wykorzystanie liczb losowych do reprezentowania innych wartości .....	321
6.2    Tworzenie własnych funkcji .....	322
<b>W CENTRUM UWAGI</b> Modularyzacja kodu z wykorzystaniem funkcji .....	330
6.3    Inne funkcje biblioteczne .....	338
6.4    Rzut oka na języki Java, Python i C++ .....	349
Pytania kontrolne .....	357
Ćwiczenia z wykrywania błędów .....	359
Ćwiczenia programistyczne .....	360
Rozdział 7. <b>Walidacja danych wejściowych .....</b>	<b>365</b>
7.1    Garbage In, Garbage Out .....	365
7.2    Pętla walidacji danych wejściowych .....	367
<b>W CENTRUM UWAGI</b> Projektowanie pętli walidacji danych wejściowych .....	369
7.3    Programowanie defensywne .....	374

7.4 Rzut oka na języki Java, Python i C++ .....	375
Pytania kontrolne .....	379
Ćwiczenia z wykrywania błędów .....	381
Ćwiczenia programistyczne .....	382
<b>Rozdział 8. Tablice .....</b>	<b>385</b>
8.1 Tablice — informacje podstawowe .....	385
<b>W CENTRUM UWAGI</b> Korzystanie z elementów tablicy w wyrażeniach matematycznych .....	392
8.2 Sekwencyjne przeszukiwanie tablicy .....	400
8.3 Przetwarzanie elementów tablicy .....	405
<b>W CENTRUM UWAGI</b> Przekazywanie tablicy .....	412
8.4 Tablice równoległe .....	419
<b>W CENTRUM UWAGI</b> Korzystanie z tablic równoległych .....	420
8.5 Tablice dwuwymiarowe .....	424
<b>W CENTRUM UWAGI</b> Korzystanie z tablic dwuwymiarowych .....	427
8.6 Tablice trój- i więcej wymiarowe .....	432
8.7 Rzut oka na języki Java, Python i C++ .....	434
Pytania kontrolne .....	444
Ćwiczenia z wykrywania błędów .....	447
Ćwiczenia programistyczne .....	448
<b>Rozdział 9. Sortowanie i przeszukiwanie tabel .....</b>	<b>453</b>
9.1 Algorytm sortowania bąbelkowego .....	453
<b>W CENTRUM UWAGI</b> Korzystanie z algorytmu sortowania bąbelkowego .....	460
9.2 Algorytm sortowania przez wybieranie .....	468
9.3 Algorytm sortowania przez wstawianie .....	473
9.4 Algorytm wyszukiwania binarnego .....	479
<b>W CENTRUM UWAGI</b> Korzystanie z algorytmu wyszukiwania binarnego .....	482
9.5 Rzut oka na języki Java, Python i C++ .....	485
Pytania kontrolne .....	497
Ćwiczenia z wykrywania błędów .....	500
Ćwiczenia programistyczne .....	501

Rozdział 10. <b>Pliki</b> .....	<b>503</b>
10.1 Odczyt i zapis do plików — informacje wstępne .....	503
10.2 Przetwarzanie plików za pomocą pętli .....	516
<b>W CENTRUM UWAGI</b> Korzystanie z plików .....	520
10.3 Korzystanie z plików i tablic .....	524
10.4 Przetwarzanie rekordów .....	525
<b>W CENTRUM UWAGI</b> Dodawanie i wyświetlanie rekordów .....	530
<b>W CENTRUM UWAGI</b> Wyszukiwanie rekordu .....	533
<b>W CENTRUM UWAGI</b> Modyfikowanie rekordów .....	535
<b>W CENTRUM UWAGI</b> Usuwanie rekordów .....	540
10.5 Separatory sterowania .....	543
<b>W CENTRUM UWAGI</b> Korzystanie z separatorów sterowania .....	544
10.6. Rzut oka na języki Java, Python i C++ .....	550
Pytania kontrolne .....	570
Ćwiczenia z wykrywania błędów .....	574
Ćwiczenia programistyczne .....	574
Rozdział 11. <b>Programy sterowane za pomocą menu</b> .....	<b>577</b>
11.1 Wprowadzenie do programów sterowanych za pomocą menu .....	577
11.2 Modularyzacja programu sterowanego za pomocą menu ....	587
11.3 Ponowne wyświetlanie menu za pomocą pętli .....	589
<b>W CENTRUM UWAGI</b> Projektowanie programu sterowanego za pomocą menu .....	596
11.4 Menu wielopoziomowe .....	610
11.5 Rzut oka na języki Java, Python i C++ .....	616
Pytania kontrolne .....	621
Ćwiczenia programistyczne .....	623
Rozdział 12. <b>Przetwarzanie tekstu</b> .....	<b>627</b>
12.1 Wstęp .....	627
12.2 Przetwarzanie poszczególnych znaków w ciągu .....	629
<b>W CENTRUM UWAGI</b> Sprawdzanie hasła .....	632
<b>W CENTRUM UWAGI</b> Formatowanie numeru telefonu i usuwanie formatowania .....	637
12.3 Rzut oka na języki Java, Python i C++ .....	642

Pytania kontrolne .....	649
Ćwiczenia z wykrywania błędów .....	651
Ćwiczenia programistyczne .....	652
<b>Rozdział 13. Rekurencja .....</b>	<b>657</b>
13.1 Wprowadzenie do rekurencji .....	657
13.2 Rozwiązywanie zadań za pomocą rekurencji .....	660
13.3 Przykłady algorytmów rekurencyjnych .....	664
13.4 Rzut oka na języki Java, Python i C++ .....	674
Pytania kontrolne .....	678
Ćwiczenia programistyczne .....	681
<b>Rozdział 14. Programowanie obiektowe .....</b>	<b>683</b>
14.1 Programowanie proceduralne i programowanie obiektowe .....	683
14.2 Klasy .....	687
14.3 Projektowanie klas za pomocą języka UML .....	698
14.4 Wyznaczanie klas i ich zakresu obowiązków w zadaniu .....	700
<b>W CENTRUM UWAGI</b> Wyznaczanie klas .....	701
<b>W CENTRUM UWAGI</b> Określanie zakresu obowiązków klasy .....	705
14.5 Dziedziczenie .....	711
14.6 Polimorfizm .....	718
14.7 Rzut oka na języki Java, Python i C++ .....	723
Pytania kontrolne .....	740
Ćwiczenia programistyczne .....	744
<b>Rozdział 15. Aplikacje z GUI i programowanie sterowane zdarzeniami .....</b>	<b>747</b>
15.1 Graficzny interfejs użytkownika .....	747
15.2 Projektowanie interfejsu użytkownika do programu wyposażonego w GUI .....	751
<b>W CENTRUM UWAGI</b> Projektowanie okna .....	755
15.3 Tworzenie procedury obsługi zdarzenia .....	758
<b>W CENTRUM UWAGI</b> Projektowanie procedury obsługi zdarzenia .....	761
15.4. Projektowanie aplikacji na urządzenia mobilne .....	764
15.5 Rzut oka na języki Java, Python i C++ .....	773
Pytania kontrolne .....	774
Ćwiczenia programistyczne .....	776

Dodatek A	<b>Tablica kodów ASCII/Unicode .....</b>	<b>779</b>
Dodatek B	<b>Symbole na schematach blokowych .....</b>	<b>781</b>
Dodatek C	<b>Przewodnik po pseudokodzie .....</b>	<b>783</b>
Dodatek D	<b>Zamiana liczb dziesiętnych na postać binarną .....</b>	<b>797</b>
Dodatek E	<b>Odpowiedzi do pytań z punktów kontrolnych .....</b>	<b>799</b>
	<b>Skorowidz .....</b>	<b>815</b>



# Wstęp

Witaj w książce *Projektowanie oprogramowania dla zupełnie początkujących. Owoce programowania. Wydanie V.* W niniejszej książce, aby pokazać Ci, jak działają programy i jak można za ich pomocą rozwiązywać zadania programistyczne, wykorzystałem metodę niewymagającą znajomości żadnego języka. Założyłem także, że czytelnik nie ma żadnego doświadczenia w projektowaniu i tworzeniu oprogramowania. Zdobywasz wiedzę dotyczącą projektowania programów komputerowych, korzystając z łatwego do zrozumienia pseudokodu, ze schematów blokowych i innych narzędzi, bez konieczności zaznajomienia się ze składnią danego języka.

W tej książce omówiłem zarówno podstawowe zagadnienia, takie jak typy danych, zmienne, dane wyjściowe i dane wyjściowe, struktury sterujące, moduły, funkcje, tablice oraz pliki, jak i zagadnienia związane z programowaniem obiektowym, tworzeniem graficznych interfejsów użytkownika i pisaniem programów sterowanych zdarzeniami. Podobnie jak w przypadku innych książek z serii *Owoce programowania* tekst jest napisany przyjaznym i zrozumiałym językiem, zachęcającym do dalszej lektury.

W każdym rozdziale prezentuję wiele przykładowych projektów programów. Są to zarówno krótkie przykłady mające na celu zaznajomić czytelnika z określonym zagadnieniem, jak i bardziej rozbudowane przykłady ilustrujące, jak można rozwiązać konkretne zadanie programistyczne. W każdym rozdziale znajduje się co najmniej jedna sekcja „W centrum uwagi”, w której krok po kroku analizuję dane zadanie i przedstawuję jego rozwiązanie.

Książka ta idealnie sprawdza się na kursach programowania, jeszcze przed rozpoczęciem nauki programowania w danym języku lub jako wstęp do nauki konkretnego języka programowania.

## Co nowego w wydaniu piątym?

Sposób przekazywania wiedzy, układ tekstu oraz klarowny język pozostały nie zmienione względem wydań poprzednich. W książce wprowadziłem wiele poprawek. Oto ich lista:

- W rozdziałach od 2. do 15. pojawia się nowy podrozdział, zatytułowany „Rzut oka na języki Java, Python i C++”. Opisuję w nim, w jaki sposób główne zagadnienia zawarte w poszczególnych rozdziałach można zaimplementować w językach Java, Python i C++. Znajdziesz tutaj wycinki kodu i kompletne programy napisane w każdym z tych języków. Jest to bardzo wartościowe narzędzie dla studentów, którzy chcieliby poznać koncepcje z danego rozdziału przedstawione w różnych językach programowania.
- W rozdziale 15. dodałem nowy podrozdział poświęcony modułowi INIT(). Jest to moduł startowy aplikacji z interfejsem graficznym, podobny do metody start znanej z aplikacji JavaFX albo do metody Form\_Load z aplikacji Windows Forms.

- W rozdziale 15. dodałem też podrozdział „Projektowanie aplikacji dla urządzeń mobilnych”. Opisuję w nim niektóre z typowych problemów, z jakimi muszą mierzyć się programiści tworzący aplikacje dla urządzeń mobilnych. Prezentuję tutaj także koncepcje programowania związane ze specjalnymi rozwiązaniami sprzętowymi urządzeń mobilnych, takimi jak wysyłanie i odbieranie wiadomości SMS, wykonywanie i odbieranie połączeń telefonicznych lub wyznaczanie pozycji urządzenia. Podaję również kilka przykładów programów w pseudokodzie działającym na symulowanym smartfonie.
- W kilku rozdziałach dodałem także nowe ćwiczenia motywacyjne.
- Zaktualizowałem też podawane w tej książce przewodniki po językach. Wszystkie przewodniki z tej książki dostępne są również na stronach wydawnictwa Helion, pod adresem <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

## Krótki przegląd poszczególnych rozdziałów

### Rozdział 1. „Wstępne informacje na temat komputerów i programowania”

Rozdział ten rozpoczynam od zwięzłego i przystępnego omówienia tego, jak działa komputer, jak zapisywane i przetwarzane są dane oraz dlaczego programy piszemy w językach wysokiego poziomu.

### Rozdział 2. „Dane wejściowe, przetwarzanie i dane wyjściowe”

Wysyłam, czym są proces tworzenia programu, typy danych, zmienne i struktury sterujące. Czytelnik, korzystając z pseudokodu i schematów blokowych, zaczyna projektować proste programy odczytujące dane, wykonujące działania matematyczne i wyświetlające wynik na ekranie.

### Rozdział 3. „Moduły”

W tym rozdziale przedstawiam zalety modularyzacji programów i metodę projektowania „od ogółu do szczegółu”. Czytelnik dowiaduje się, jak się definiuje i wywołuje moduły, przekazuje do nich argumenty i korzysta ze zmiennych lokalnych. Przedstawiam także schematy hierarchiczne jako jedno z narzędzi służących do projektowania.

### Rozdział 4. „Struktury warunkowe i logika boolowska”

Czytelnik poznaje operatory relacji i wyrażenia boolowskie, a także dowiaduje się, jak za pomocą struktur warunkowych można sterować wykonaniem programu. Przedstawiam instrukcje If-Then, If-Then-Else i If-Then-Else If. Omawiam także zagnieżdżone struktury warunkowe, operatory logiczne i strukturę decyzyjną.

## Rozdział 5. „Struktury cykliczne”

W tym rozdziale wyjaśniam, jak tworzy się pętle za pomocą struktur cyklicznych oraz jak działają pętle While, Do-While, Do-Until i For. Omawiam także zmienne licznikowe, akumulatory, sumy bieżące i wartowniki.

## Rozdział 6. „Funkcje”

Rozdział ten rozpoczynam od omówienia popularnych funkcji bibliotecznych, na przykład funkcji generującej liczbę losową. Wyjaśniam czytelnikowi, jak wywołuje się funkcje biblioteczne, jak funkcja zwraca wartość oraz jak się tworzy i wywołuje własne funkcje.

## Rozdział 7. „Walidacja danych wejściowych”

W tym rozdziale wyjaśniam, jak ważną rolę odgrywa walidacja danych wejściowych. Czytelnik uczy się tworzyć pętle walidacji danych pełniące rolę pułapek na błędy. Omawiam także programowanie defensywne oraz to, jakie znaczenie ma zdolność przewidywania oczywistych i nieoczywistych błędów.

## Rozdział 8. „Tablice”

W tym rozdziale czytelnik dowiaduje się, jak się tworzy i wykorzystuje tablice jedno- i dwuwymiarowe. Przedstawiam wiele przykładów ilustrujących, jak oblicza się sumę, średnią, największą i najmniejszą wartość elementów tablicy oraz jak sumuje się wiersze, kolumny i wszystkie elementy tablicy dwuwymiarowej. Wyjaśniam także, w jakich sytuacjach przydają się tablice równejległe.

## Rozdział 9. „Sortowanie i przeszukiwanie tabel”

W tym rozdziale czytelnik poznaje podstawy dotyczące porządkowania i wyszukiwania elementów w tablicy. Opisuję algorytmy sortowania bąbelkowego, sortowania przez wybieranie, sortowania przez wstawianie i wyszukiwania binarnego.

## Rozdział 10. „Pliki”

W tym rozdziale omawiam odczyt i zapis plików w trybie sekwencyjnym. Czytelnik dowiaduje się, jak się odczytuje i zapisuje duże zbiory danych, zapisuje dane w postaci rekordów oraz projektuje programy operujące na plikach i tablicach. Rozdział kończę omówieniem separatorów sterowania.

## Rozdział 11. „Programy sterowane za pomocą menu”

W tym rozdziale czytelnik uczy się projektować programy wyświetlające menu i wykonujące operacje wskazane przez użytkownika w menu. Wyjaśniam także, jak ważna jest modularyzacja programów sterowanych za pomocą menu.

## **Rozdział 12. „Przetwarzanie tekstu”**

W tym rozdziale opisuję szczegółowo tematykę związaną z przetwarzaniem danych tekstowych. Omawiam algorytmy, które śledzą po kolej i każdy znak w ciągu znaków, i opisuję kilka popularnych funkcji bibliotecznych do przetwarzania znaków.

## **Rozdział 13. „Rekurencja”**

W tym rozdziale omawiam rekurencję i jej zastosowanie. Graficznie prezentuję, jak następują po sobie kolejne wywołania rekurencyjne i jak działają aplikacje rekurencyjne. Przedstawiam wiele algorytmów rekurencyjnych, takich jak obliczanie silni liczby, obliczanie największego wspólnego dzielnika, sumowanie zakresu elementów tablicy oraz wyszukiwanie binarne. Omawiam także klasyczny przykład zadania rekurencyjnego, jakim są wieże Hanoi.

## **Rozdział 14. „Programowanie obiektowe”**

W tym rozdziale pokazuję różnicę między programowaniem proceduralnym i programowaniem obiektowym. Wyjaśniam podstawy dotyczące klas i obiektów — czym są pola, metody, modyfikatory dostępu, konstruktory, akcesory i mutatory. Czytelnik dowiaduje się, jak modelować klasy za pomocą języka UML i określać klasy potrzebne do rozwiązania konkretnego zadania.

## **Rozdział 15. „Aplikacje z GUI i programowanie sterowane zdarzeniami”**

W tym rozdziale omawiam proste zagadnienia związane z projektowaniem aplikacji wyposażonych w graficzny interfejs użytkownika. Omawiam tworzenie GUI za pomocą edytorów graficznych (takich jak Visual Studio czy NetBeans). Czytelnik dowiaduje się, czym są zdarzenia w aplikacji GUI i jak tworzy się procedury obsługi zdarzenia.

## **Dodatek A. „Tablica kodów ASCII/Unicode”**

W tym dodatku prezentuję listę znaków w kodowaniu ASCII, która wygląda tak samo jak lista pierwszych 127 znaków w kodowaniu Unicode.

## **Dodatek B. „Symbole na schematach blokowych”**

W tym dodatku wyjaśniam symbole na schematach blokowych, których używam w książce.

## **Dodatek C. „Przewodnik po pseudokodzie”**

Ten dodatek pełni rolę przewodnika, w którym opisuję polecenia i operatory pseudokodu, jakich używam w tej książce.

## **Dodatek D. „Zamiana liczb dziesiętnych na postać binarną”**

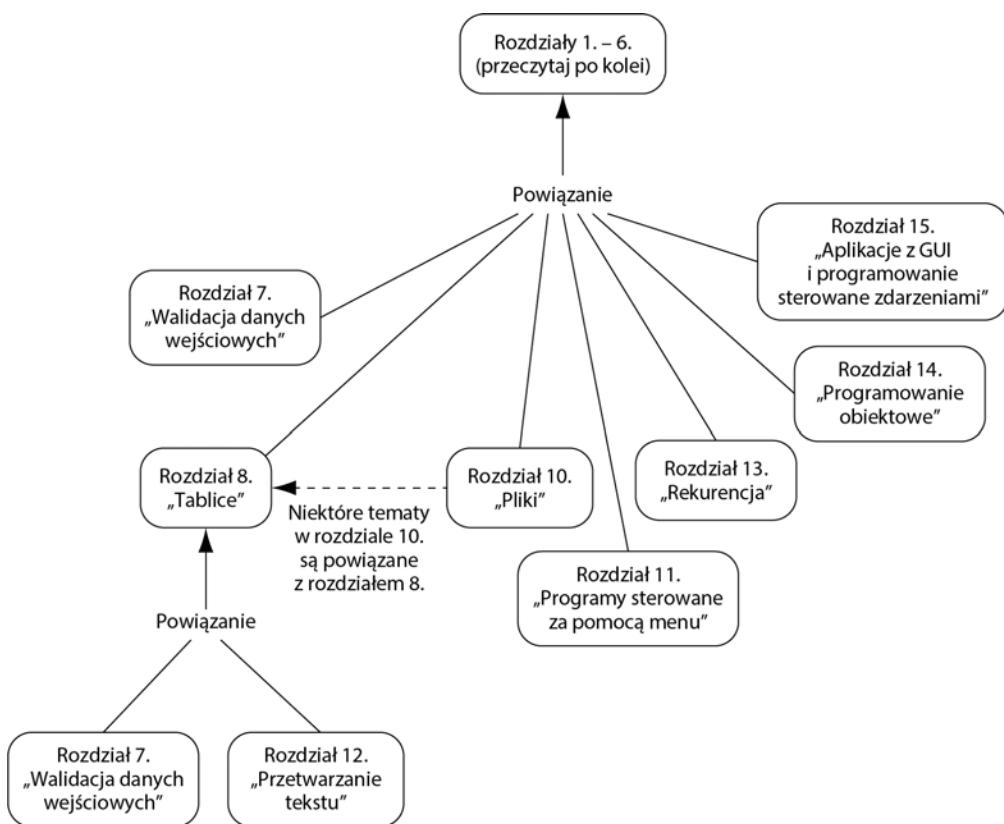
W dodatku tym zamieściłem prosty przykład ilustrujący zamianę liczby z postaci dziesiętnej na binarną.

## Dodatek E. „Odpowiedzi do pytań z punktów kontrolnych”

Dodatek ten zawiera odpowiedzi do pytań zamieszczonych w punktach kontrolnych.

## Organizacja tekstu

W niniejszej książce wyjaśniam krok po kroku, jak działają programy i jak się je projektuje. W każdym rozdziale omawiam pewne zagadnienia, a czytelnik stopniowo poszerza swoją wiedzę. Mimo że rozdziały powinno się czytać po kolej, istnieje także pewna dowolność. Na rysunku W.1 przedstawiłem powiązania między rozdziałami. Każdy blok oznacza rozdział lub grupę rozdziałów. Rozdział, na który skierowana jest strzałka, należy przeczytać przed rozdziałem, od którego strzałka wychodzi. Przerywana linia oznacza, że niektóre tematy omówione w rozdziale 10. związane są z tematyką opisaną w rozdziale 8.



**Rysunek W.1.** Powiązania między rozdziałami

## Elementy rozdziałów

**WYJAŚNIENIE.** Niemal każdy podrozdział rozpoczyna się od krótkiego wyjaśnienia zagadnienia, które będę w nim omawiał.

**PRZYKŁADOWE PROGRAMY.** W każdym rozdziale znajduje się wiele pełnych lub częściowych programów, mających na celu objaśnienie poruszanego zagadnienia. Przykładowe programy przedstawione są za pomocą pseudokodu, schematów blokowych i innych narzędzi.

**W CENTRUM UWAGI.** W każdym rozdziale znajduje się co najmniej jedna sekcja „W centrum uwagi”, w której prezentuję studium przypadku i szczegółową analizę zadania. Następnie pokazuję czytelnikowi, jak można dane zadanie rozwiązać.



**UWAGA:** Uwagi znajdują się w wielu miejscach tekstu. Wyjaśniają one interesujące lub często źle rozumiane zagadnienia związane z omawianym tematem.



**WSKAZÓWKA:** Wskazówki doradzają czytelnikowi, jakie techniki użyć, aby rozwiązać określone zadanie programistyczne.



**OSTRZEŻENIE!** Ostrzeżenia informują czytelnika, że pewne techniki programistyczne mogą przyczynić się do błędного działania programu lub utraty zapisanych danych.



**KOD TOWARZYSZĄCY PRZYKŁADOWI.** Wielu przykładowym programom zapisanym w książce za pomocą pseudokodu towarzyszy kod zapisany w językach Java, Python i Visual Basic. Kod ten można pobrać ze strony internetowej książki (<ftp://ftp.helion.pl/przyklady/pkpro5.zip>). Jeśli kodowi z danego listingu towarzyszy kod zapisany w języku Java, umieszczona jest przy nim odpowiednia ikonka.



**PUNKTY KONTROLNE.** Punkty kontrolne znajdują się w kilku miejscach każdego rozdziału i zawierają pytania. Mają one na celu sprawdzenie wiedzy czytelnika zaraz po zapoznaniu się z nowym zagadnieniem.

**PYTANIA KONTROLNE.** W każdym rozdziale zamieszczony jest rozbudowany zestaw pytań kontrolnych i ćwiczeń. Są to: „Test jednokrotnego wyboru”, „Prawda czy fałsz?”, „Krótka odpowiedź” i „Warsztat projektanta algorytmów”.

**ĆWICZENIA Z WYKRYWANIA BŁĘDÓW.** W większości rozdziałów zamieściłem zestaw ćwiczeń z wykrywania błędów. Prezentuję w nich pseudokod programu, a zadaniem czytelnika jest odszukanie w nim błędu.

**ĆWICZENIA PROGRAMISTYCZNE.** Na końcu każdego rozdziału zamieściłem zestaw ćwiczeń programistycznych mających na celu utrwalenie wiedzy zdobytej przez czytelnika podczas lektury rozdziału.



# Podziękowania

W pisaniu i publikacji tej książki pomogło mi bardzo wiele osób. Chciałbym podziękować następującym korektorom merytorycznym:

## **Korektorzy merytoryczni niniejszego wydania**

Tony Cantrell  
*Georgia Northwestern Technical College*  
Keith Hallmark  
*Calhoun Community College*  
Vai Kumar  
*Pensacola State College*

## **Korektorzy merytoryczni poprzednich wydań**

Reni Abraham  
*Houston Community College*  
Alan Anderson  
*Gwinnett Technical College*  
Cherie Aukland  
*Thomas Nelson Community College*  
Steve Browning  
*Freed Hardeman University*  
John P. Buerck  
*Saint Louis University*  
Jill Canine  
*Ivy Tech Community College of Indiana*  
Steven D. Carver  
*Ivy Tech Community College*  
Stephen Robert Cheskiewicz  
*Keystone College and Wilkes University*  
Katie Danko  
*Grand Rapids Community College*  
Richard J. Davison  
*College of the Albemarle*  
Sameer Dutta  
*Grambling State University*  
Norman P. Hahn  
*Thomas Nelson Community College*  
John Haley  
*Athens Technical College*

Ronald J. Harkins

*Miami University, OH*

Dianne Hill

*Jackson College*

Coronicca Oliver

*Coastal Georgia Community College*

Robert S. Overall, III

*Nashville State Community College*

Dale T. Pickett

*Baker College of Clinton Township*

Tonya Pierce

*Ivy Tech Community College*

J. Shawn Pope

*Tulsa Community College*

Maryam Rahnemoonfar

*Texas A&M University*

Linda Reeser

*Arizona Western College*

Homayoun Sharafi

*Prince George's Community College*

Emily Shepard

*Central Carolina Community College*

Larry Strain

*Ivy Tech Community College–Bloomington*

Donald Stroup

*Ivy Tech Community College*

John Thacher

*Gwinnett Technical College*

Jim Turney

*Austin Community College*

Scott Vanselow

*Edison State College*

Chciałbym także podziękować wszystkim pracownikom wydawnictwa Pearson, dzięki którym seria *Owoce programowania* okazała się tak dużym sukcesem. Moja współpraca z zespołem wydawnictwa Pearson była tak bliska, że traktuję już tych ludzi jak najbliższych przyjaciół. Jestem szczęściarzem, bo redaktorem tej książki był Matt Goldstein, a redaktorką pomocniczą była Meghan Jacoby. To oni pomogli mi podczas korekty tej, a także wielu innych książek. Cieszę się również, że menadżerami ds. marketingu był Demetrius Hall. Jego zaangażowanie jest niezwykle inspirujące — wykonał ogromną pracę, aby wypromować moje książki w środowisku akademickim. Aby ta książka ujrzała światło dzienne, niestrudzenie działał zespół redakcyjny pod kierunkiem Carole Snyder. Dziękuję Wam wszystkim!

# O autorze

**TONY GADDIS** jest głównym autorem podręczników z serii *Owoce programowania*. Tony ma dwudziestoletnie doświadczenie w dziedzinie prowadzenia kursów informatycznych, zwłaszcza w Haywood Community College. Jest wielce docenianym wykładowcą, który został nagrodzony tytułem Nauczyciela Roku na uczelni North Carolina Community College i otrzymał nagrodę Teaching Excellence przyznaną przez National Institute for Staff and Organizational Development. W ramach serii *Owoce programowania* wydawnictwo Pearson wydało dotychczas książki dotyczące nauki projektowania oprogramowania, a także języków C++, Java, Visual Basic, C#, Python, Alice oraz aplikacji App Inventor.



# Wstępne informacje na temat komputerów i programowania

## TEMATYKA

- |   |                                  |
|---|----------------------------------|
| 1.1 Wstęp                                   | 1.4 W jaki sposób działa program |
| 1.2 Sprzęt                                  | 1.5 Rodzaje oprogramowania       |
| 1.3 W jaki sposób komputer przechowuje dane |                                  |

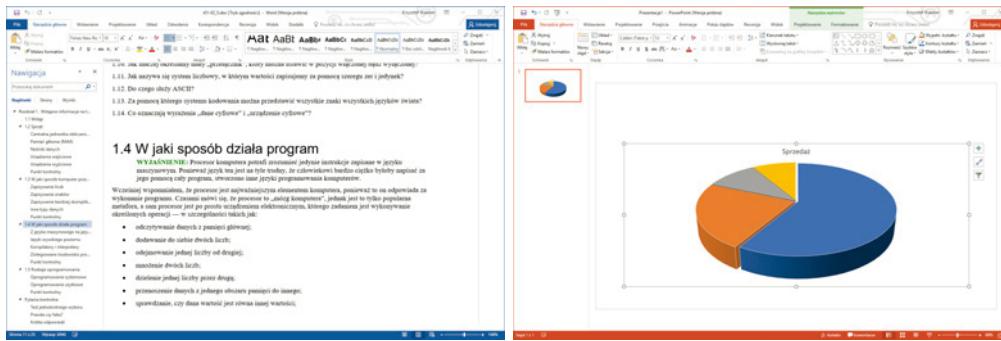
### 1.1

### Wstęp

Pomyśl przez chwilę, na jak wiele sposobów ludzie korzystają z komputerów. Uczniowie w szkole piszą na nich wypracowania, wyszukują artykuły, wysyłają wiadomości e-mail i biorą udział w zajęciach online. W pracy natomiast za pomocą komputerów analizuje się dane, tworzy prezentacje, przeprowadza transakcje biznesowe, nawiązuje kontakty z klientami i współpracownikami, steruje maszynami w fabrykach, a także wykonuje się wiele innych czynności. W domu wykorzystujemy komputery do takich czynności jak płacenie rachunków, zakupy w sieci, kontakty z przyjaciółmi i rodziną oraz granie w gry. Zwróć uwagę, że urządzenia takie jak na przykład smartfony, tablety, odtwarzacze MP3 czy systemy nawigacji samochodowej są także komputerami. W życiu codziennym używamy komputerów niemal na każdym kroku.

Komputery potrafią wykonywać tak wiele zadań dlatego, że można je w dowolny sposób zaprogramować. Oznacza to, że komputery nie zostały stworzone, aby wykonywać jedno, określone zadanie, ale po to, aby wykonywać zadanie, jakie wskaże im określony program. **Program** składa się z szeregu instrukcji, które komputer musi uruchomić, aby wykonać określone zadanie. Na rysunku 1.1 przedstawione są dwa popularne programy: Microsoft Word i Microsoft PowerPoint.

Programy to inaczej **oprogramowanie**. Z punktu widzenia komputera oprogramowanie jest rzeczą kluczową — bez niego nie byłby w stanie wykonać żadnej operacji. Programy, dzięki którym nasze komputery są dla nas tak przydatne, tworzą twórcy oprogramowania, czyli **programiści**. Osoby te posiadają wiedzę dotyczącą projektowania, tworzenia i testowania programów komputerowych. Bycie programistą jest



**Rysunek 1.1.** Popularne programy komputerowe (zdjęcie dzięki uprzejmości Microsoft Corporation)

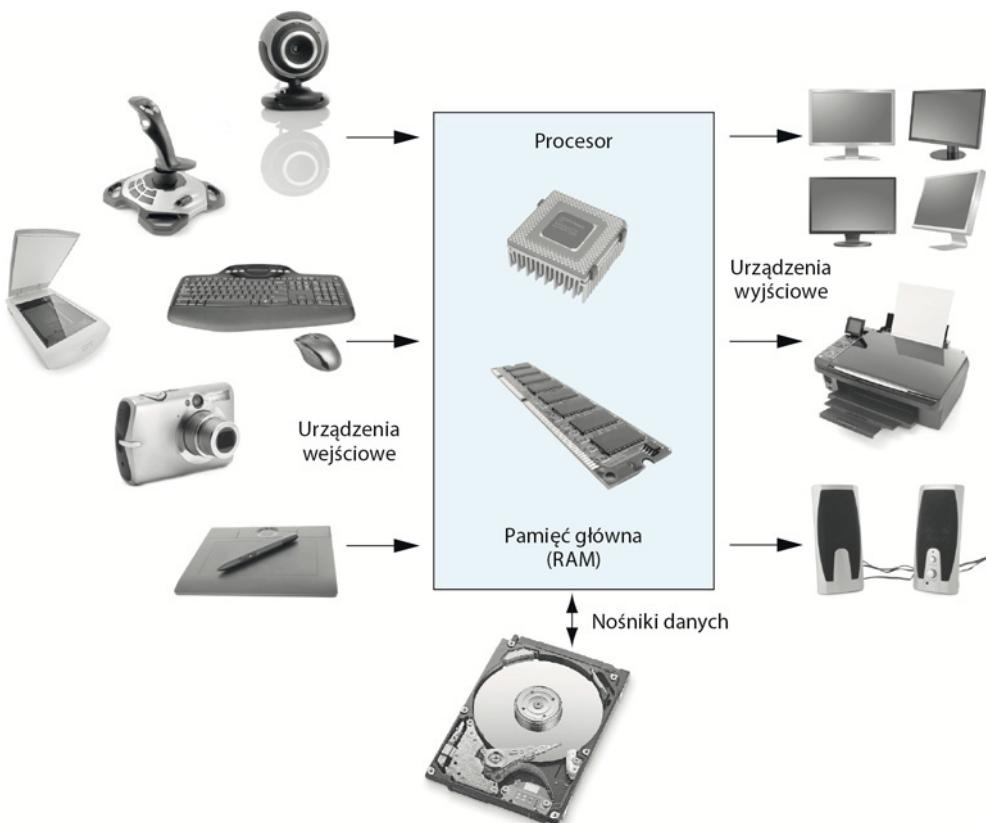
ekscytującym i satysfakcjonującym zajęciem. W dzisiejszych czasach programistów można znaleźć w bardzo wielu branżach: biznesie, medycynie, organach administracji publicznej, organach ścigania, rolnictwie, szkolnictwie, przemyśle rozrywkowym — niemal w każdej dziedzinie.

Dzięki tej książce poznasz podstawowe zagadnienia związane z programowaniem komputerów. Zanim jednak przejdziemy do omawiania tych zagadnień, musisz zrozumieć kilka prostych rzeczy dotyczących komputerów i ich działania. W tym rozdziale zdobędziesz wystarczającą wiedzę, która umożliwi Ci dalsze pogłębianie tematyki związanej z komputerami. Na początku omówię fizyczne elementy, z których składa się komputer. W kolejnym kroku przyjrzyzę się temu, w jaki sposób komputer przechowuje dane i uruchamia programy. Na koniec rozdziału przybliżę główne typy oprogramowania komputerowego.

## 1.2 Sprzęt

**WYJAŚNIENIE:** Sprzętem nazywamy wszystkie urządzenia fizyczne, z których zbudowany jest komputer. W zdecydowanej większości przypadków komputery składają się z bardzo podobnych urządzeń.

Przez słowo **sprzęt** (ang. *hardware*) rozumiemy wszystkie urządzenia fizyczne (komponenty), z których zbudowany jest komputer. Komputer nie jest więc pojedynczym urządzeniem, lecz systemem składającym się z wielu urządzeń, które ze sobą współpracują. Podobnie jak w przypadku poszczególnych instrumentów w orkiestrze, każde urządzenie odgrywa w komputerze określoną rolę. Jeśli kiedyś kupowałeś komputer, zauważyleś zapewne, że jest on opisany za pomocą listy komponentów takich jak procesor, pamięć, napędy, monitor, karta graficzna itp. Jeżeli nie masz dostatecznej wiedzy na temat komputerów lub nie znasz kogoś, kto ją posiada, zrozumienie takiego opisu może okazać się kłopotliwe. Na rysunku 1.2 przedstawiłem główne elementy, z jakich składa się typowy komputer:



**Rysunek 1.2.** Typowe elementy, z których składa się komputer  
(wszystkie zdjęcia © Shutterstock)

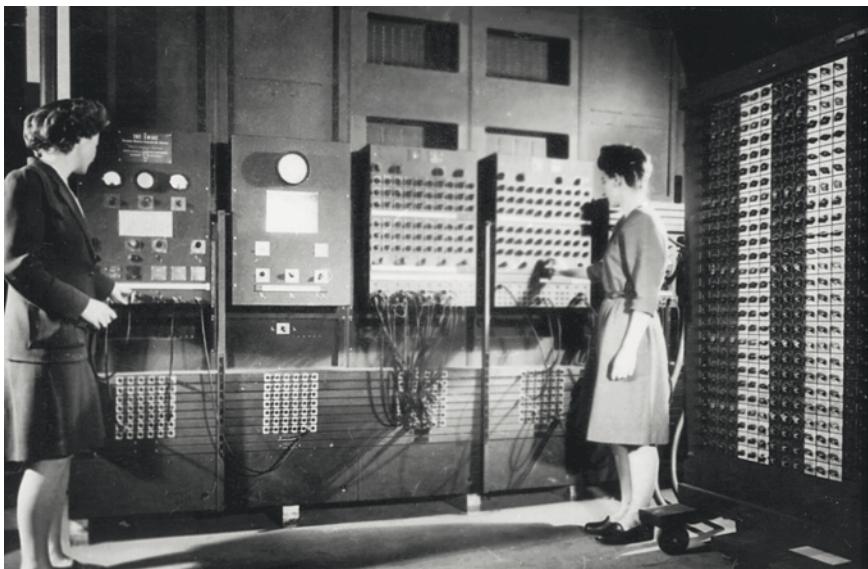
- centralna jednostka obliczeniowa (CPU),
- pamięć główna,
- nośniki danych,
- urządzenia wejściowe,
- urządzenia wyjściowe.

Przyjrzyjmy się teraz bliżej każdemu z wymienionych elementów.

## Centralna jednostka obliczeniowa (CPU)

Kiedy komputer wykonuje wskazane przez program działania, mówimy, że **uruchomił** lub **wykonuje** dany program. To właśnie **centralna jednostka obliczeniowa** (ang. *central processing unit* — CPU) jest elementem komputera odpowiedzialnym za wykonywanie programu. Zamiast CPU przeważnie używa się nazwy **procesor**. Jest on najważniejszym elementem komputera, ponieważ bez niego komputer nie mógłby wykonać żadnego programu.

Pierwsze komputery były wyposażone w ogromne procesory, zbudowane z elektrycznych i mechanicznych elementów, takich jak lampy elektronowe i przełączniki. Na rysunku 1.3 widoczny jest właśnie taki komputer — dwie kobiety przedstawione na zdjęciu obsługują historyczny komputer **ENIAC**. Zbudowany w 1945 roku i używany przez armię Stanów Zjednoczonych do wyliczania tablic balistycznych, ENIAC uważany jest za pierwszy na świecie programowalny komputer elektroniczny. Maszyna ta (będąca tak naprawdę jednym wielkim procesorem) mierzyła 2,4 metra wysokości i 30 metrów długości, a ważyła 30 ton.



**Rysunek 1.3.** Komputer ENIAC (zdjęcie dzięki uprzejmości US ARMY Center of Military History)

Obecnie CPU to małe chipy zwane także **mikroprocesorami**. Na rysunku 1.4 widoczny jest inżynier trzymający współczesny mikroprocesor. Poza tym, że dzisiejsze mikroprocesory są znacznie mniejsze od swoich wczesnych elektromechanicznych odpowiedników, mają również znacznie większą moc obliczeniową.

## Pamięć główna (RAM)

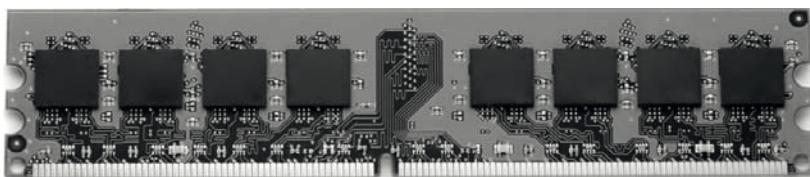
Wyobraź sobie, że **pamięć główna** (ang. *main memory*) to przestrzeń robocza komputera. Jest to miejsce, w którym komputer przechowuje uruchomiony program, jak również dane, na których ten program pracuje. Przykładowo założmy, że piszesz wypracowanie w edytorze tekstowym — w takim przypadku zarówno program w postaci edytora tekstowego, jak i samo wypracowanie zostaną umieszczone w pamięci głównej komputera.

Pamięć główną nazywa się także **pamięcią o dostępie swobodnym** (ang. *random-access memory* — RAM). Nazwa ta wynika z tego, że procesor musi mieć szybki dostęp do każdej informacji zapisanej w dowolnym miejscu pamięci. RAM jest pamięcią



**Rysunek 1.4.** Inżynier trzymający współczesny mikroprocesor (zdjęcie dzięki uprzejmości Chris Ryan/OJO Images/Getty Images)

**ulotną** i wykorzystuje się ją jako tymczasowe miejsce zapisywania informacji — tylko na czas działania programu. Kiedy komputer się wyłączy, wszystkie dane zostaną wymazane z pamięci głównej. Pamięć RAM ma postać płytka z chipami — takiej, jaką widać na rysunku 1.5.



**Rysunek 1.5.** Chipy pamięci (zdjęcie © Garsya/Shutterstock)



**UWAGA:** Inny rodzaj pamięci, która jest zbudowana z chipów, to **pamięć tylko do odczytu** (ang. *read-only memory* — ROM). Komputer potrafi odczytać zawartość takiej pamięci, ale nie może jej w żaden sposób zmodyfikować lub zapisać w niej nowych danych. ROM jest więc pamięcią **nieulotną**, co oznacza, że zapisane na niej dane nie zostaną usunięte nawet po wyłączeniu komputera. W pamięci ROM umieszcza się zazwyczaj programy konieczne do pracy systemu. Przykładem niech będzie program startowy, który uruchamia się zaraz po włączeniu komputera.

## Nośniki danych

**Nośnik danych** to taki rodzaj pamięci, w której można zapisać i przechowywać dane przez bardzo długi czas — nawet wtedy, gdy komputer jest wyłączony. Na takich urządzeniach zapisane są programy, które — kiedy zajdzie taka potrzeba — są ładowane do pamięci głównej. Na nośnikach danych są także zapisywane ważne dane, takie jak dokumenty tekstowe, informacje dotyczące wynagrodzeń pracowników czy wykazy zapasów magazynowych.

Najpowszechniejszym przykładem nośnika danych jest dysk twardy. Tradycyjny **dysk twardy** zapisuje dane na nośnikach magnetycznych w postaci talerzy. Natomiast zyskujące obecnie coraz większą popularność **dyski SSD** (ang. *solid state drive*) zapisują dane w pamięciach będących układami scalonymi. W dysku SSD nie ma żadnych ruchomych elementów i działa on znacznie szybciej od tradycyjnego dysku twardego. Większość komputerów jest wyposażona w jakiś rodzaj nośnika danych — czy to tradycyjny dysk twardy, czy dysk SSD. Istnieją także zewnętrzne dyski, które można podłączyć do któregoś z portów komunikacyjnych komputera. Używa się ich najczęściej do przechowywania kopii zapasowych danych lub podczas przenoszenia danych do innego komputera.

Poza dyskami zewnętrznymi istnieje również szereg urządzeń służących do kopiowania lub przenoszenia danych pomiędzy komputerami. Są to **napędy USB** (ang. *universal serial bus drive*), mające postać małego urządzenia podłączanego do portu USB. Po ich podłączeniu system wykrywa je jako kolejny dysk — jednak w ich wnętrzu tak naprawdę nie kryją się żadne dyski. Urządzenia te zapisują dane w specjalnej pamięci zwanej **pamięcią flash**. Napędy USB (określano często jako **pendrive**) są urządzeniami niedrogimi, niezawodnymi i na tyle małymi, że mieszkają się w kieszeni spodni.

Do przechowywania danych wykorzystuje się także nośniki optyczne, takie jak dyski **CD** (ang. *compact disc*) czy **DVD** (ang. *digital versatile disc*). Dane na dyskach optycznych nie są zapisane w sposób magnetyczny, lecz zakodowane za pomocą szeregu rowków wytłoczonych na powierzchni płyty. Napędy CD i DVD odczytują zakodowane dane za pomocą lasera. Na dysku optycznym można zapisać ogromne ilości danych, a ponieważ nie ma obecnie problemu z dostępnością zapisywanych płyt CD lub DVD, stanowią one idealny nośnik do przechowywania kopii zapasowych danych.



**UWAGA:** Obecnie popularność zyskuje przechowywanie danych w **chmurze**. Zapisane w chmurze dane są przechowywane na zdalnych komputerach dostępnych przez internet lub w prywatnej sieci danej firmy. Po zapisaniu danych w chmurze możesz uzyskać do nich dostęp z wielu różnych urządzeń oraz z dowolnego miejsca, pod warunkiem że masz dostęp do sieci. W chmurze można również zapisywać kopie zapasowe ważnych danych znajdujących się na komputerze.

## Urządzenia wejściowe

Dane wejściowe (ang. *input*) to dane, które komputer pobiera od użytkownika lub z innego urządzenia. Urządzenie, które pobiera te dane i przesyła je do komputera, nazywamy **urządzeniem wejściowym** (ang. *input device*). Najpopularniejsze urządzenia wejściowe to myszka, klawiatura, ekran dotykowy, skaner, mikrofon i aparat cyfrowy. Dyski twardy i napędy optyczne także można traktować jako urządzenia wejściowe, ponieważ programy i dane są z nich ładowane do pamięci komputera.

## Urządzenia wyjściowe

Dane wyjściowe (ang. *output*) to dane, które komputer prezentuje użytkownikowi lub przekazuje do innego urządzenia. Może to być raport sprzedaży, lista nazwisk lub plik graficzny. Dane przekazywane są do **urządzenia wyjściowego** (ang. *output device*), które z kolei je formatuje i prezentuje użytkownikowi. Popularnymi przykładami urządzeń wyjściowych są monitor lub drukarka. Dyski twardy i nagrywarki CD także można traktować jako urządzenia wyjściowe, ponieważ komputer wysyła do nich dane, które mają zostać zapisane.



## Punkt kontrolny

- 1.1. Co to jest program komputerowy?
- 1.2. Co to jest sprzęt?
- 1.3. Wymień pięć głównych elementów komputera.
- 1.4. Który z komponentów komputera służy do wykonywania programu?
- 1.5. Który element służy jako przestrzeń robocza komputera, w której na czas działania programu zapisywane są dane i sam program?
- 1.6. Który element komputera służy do przechowywania danych przez dłuższy okres czasu, nawet wtedy, gdy komputer jest wyłączony?
- 1.7. Który element komputera pobiera dane od użytkownika lub z innego urządzenia?
- 1.8. Który element komputera odpowiedzialny jest za formatowanie danych i prezentowanie ich użytkownikowi?

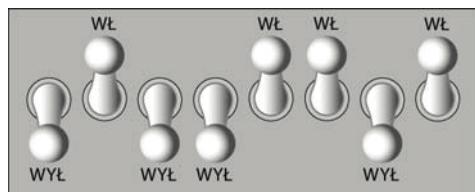
## 1.3

## W jaki sposób komputer przechowuje dane

**WYJAŚNIENIE:** Wszelkie dane, które mają zostać zapisane w komputerze, są konwertowane do postaci sekwencji złożonej z zer (0) i jedynek (1).

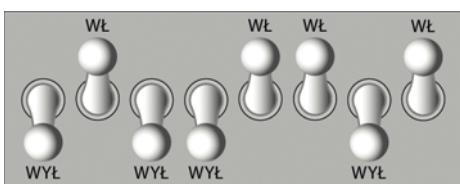
Pamięć komputera jest podzielona na malutkie obszary zwane **bajtami**. Jeden bajt pozwala zapisać jedną literę lub niewielką liczbę. Aby komputer mógł zapisać jakąś bardziej znaczącą informację, musi on być wyposażony w bardzo dużą liczbę takich bajtów. Obecnie większość komputerów jest wyposażona w pamięci o rozmiarze wielu milionów, a nawet miliardów bajtów.

Każdy bajt jest z kolei podzielony na osiem mniejszych części zwanych **bitami**. Słowo „bit” oznacza cyfrę binarną (ang. *binary digit*). Często stosuje się analogię, w której bit przedstawiony jest jako przełącznik — może on być włączony albo wyłączony. Jednak same bity nie są przełącznikami — przynajmniej nie w dosłownym tego słowa znaczeniu. W większości systemów komputerowych bit przyjmuje formę elementu elektronicznego, który charakteryzuje się dodatnim lub ujemnym ładunkiem elektrycznym. Dodany ładunek można sobie wyobrazić jako przełącznik w pozycji **włączonej**, a ładunek ujemny jako przełącznik w pozycji **wyłączonej**. Na rysunku 1.6 przedstawiłem coś, co można sobie wyobrazić jako 1 bajt pamięci: szereg przełączników, z których każdy jest włączony albo wyłączony.

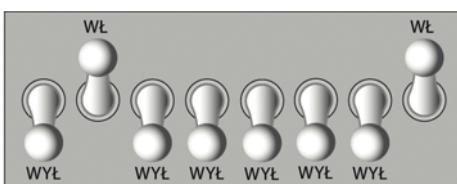


Rysunek 1.6. Wyobraź sobie bajt jako zespół ośmiu przełączników

Kiedy komputer zapisuje w danym bajcie jakąś informację, ustawa on w odpowiedni sposób każdy z tych ośmiu przełączników. Przykładowo na rysunku 1.7 z lewej strony przedstawiłem, jak zostanie zapisana w bajcie liczba 77, natomiast z prawej strony pokazałem, jak zapisana zostanie litera A. Za chwilę dowiesz się, jak można określić, jaką wartość reprezentuje dana kombinacja ustawień przełączników.



Liczba 77 zapisana za pomocą 1 bajta



Litera A zapisana za pomocą 1 bajta

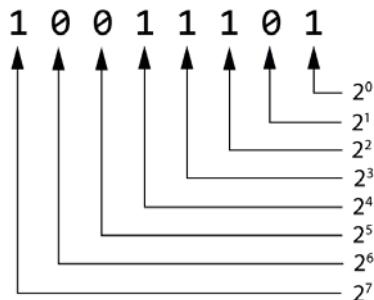
Rysunek 1.7. Kombinacje przełączników przedstawiające liczbę 77 i literę A

## Zapisywanie liczb

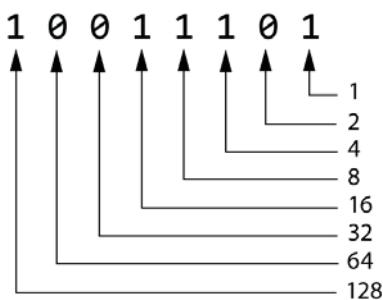
Pojedynczy bit może reprezentować liczbę w bardzo ograniczony sposób. W zależności od tego, czy bit jest ustawiony czy nie, reprezentuje on jedną z dwóch wartości. W przypadku systemów komputerowych bit, który nie jest ustawiony, reprezentuje liczbę 0, a bit ustawiony reprezentuje liczbę 1. Doskonale wpisuje się to w naturę **binarnego systemu liczbowego**. W systemie takim każdą liczbę przedstawia się za pomocą sekwencji zer i jedynek. Oto przykład liczby zapisanej w systemie binarnym:

10011101

Położenie każdej z cyfr wskazuje jednocześnie na wartość, jaką ona reprezentuje. Zaczynając od cyfry położonej najdalej z prawej strony i idąc w lewą stronę, są to kolejno wartości:  $2^0, 2^1, 2^2, 2^3$  itd. — przedstawiłem to na rysunku 1.8. Na rysunku 1.9 widoczna jest ta sama sekwencja, tylko z wyliczonymi już kolejnymi wartościami bitów. Zaczynając od cyfry położonej najdalej z prawej strony i idąc w lewą stronę, są to kolejno wartości: 1, 2, 4, 8 itd.

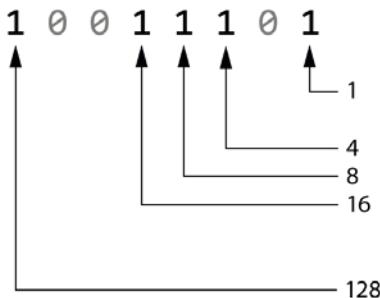


**Rysunek 1.8.** Wartości bitów to kolejne potęgi liczby 2

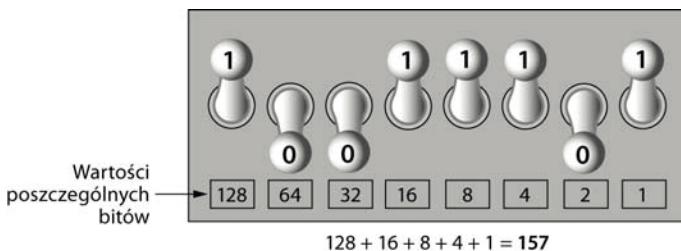


**Rysunek 1.9.** Wartości bitów

Aby określić, jaką wartość reprezentuje dana sekwencja bitów, należy dodać do siebie wartości wszystkich bitów ustawionych na 1. Przykładowo w przypadku liczby binarnej 10011101 jedynki znajdują się na pozycjach reprezentujących wartości 1, 4, 8, 16 i 128. Zilustrowałem to na rysunku 1.10. Suma poszczególnych wartości da nam liczbę 157. Tak więc wartość binarnej liczby 10011101 równa jest 157.

**Rysunek 1.10.** Określanie wartości liczby binarnej 10011101

Na rysunku 1.11 pokazałem, w jaki sposób możesz sobie wyobrazić liczbę 157 zapisaną w 1 bajcie pamięci. Każdą jedynkę reprezentuje bit w pozycji włączonej, a każde zero — bit w pozycji wyłączonej.

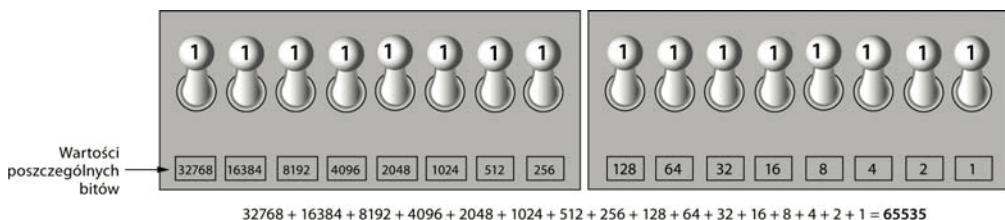
**Rysunek 1.11.** Kombinacja bitów reprezentująca liczbę 157

Kiedy wszystkie bity ustawione są na 0 (wyłączone), wartość bajta jest równa 0. Kiedy wszystkie bity ustawione są na 1 (włączone), wartość bajta jest równa największej wartości, jaką można w nim zapisać. Największa wartość, jaką można zapisać w 1 bajcie to  $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$ . Ograniczenie to wynika z faktu, że bajt składa się tylko z 8 bitów.

A co w sytuacji, gdy chcesz zapisać liczbę większą niż 255? Odpowiedź na to pytanie jest prosta: wykorzystaj więcej niż 1 bajt. Przykładowo wykorzystajmy 2 bajty — da nam to w sumie 16 bitów. W tym przypadku wartości kolejnych bitów będą równe:  $2^0, 2^1, 2^2, 2^3$  itd. — aż do  $2^{15}$ . Na rysunku 1.12 pokazałem, że największa wartość, jaką można zapisać za pomocą 2 bajtów, wynosi 65535. Jeżeli będziemy chcieli zapisać jeszcze większą wartość, będziemy musieli użyć jeszcze większej liczby bajtów.



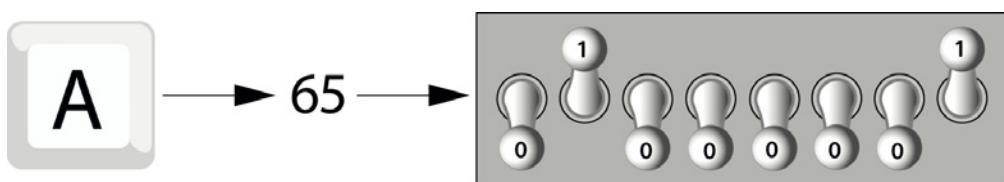
**WSKAZÓWKA:** Jeśli przytaczają Cię te wszystkie informacje, nie przejmuj się! Pisząc programy, nie musimy zamieniać liczb na wartości w systemie binarnym. Jednak wiedza dotycząca tego, w jaki sposób komputer zapisuje informacje, ułatwi Ci naukę i sprawi, że będziesz lepszym programistą.



**Rysunek 1.12.** Większą liczbę można zapisać za pomocą 2 bajtów

## Zapisywanie znaków

Każda informacja w pamięci komputera musi zostać zapisana za pomocą liczby binarnej. Dotyczy to także znaków — takich jak litery czy znaki przestankowe. Kiedy komputer zapisuje w pamięci znak, zamienia go najpierw do postaci kodu numerycznego. Następnie taki kod numeryczny zapisywany jest w pamięci komputera za pomocą liczby binarnej. Na przestrzeni lat rozwinęło się wiele różnych sposobów kodowania znaków. Z historycznego punktu widzenia najważniejszym z nich jest system ASCII, który jest akronimem nazwy *American Standard Code for Information Interchange*. Kodowanie ASCII to zestaw 128 kodów, które reprezentują litery w języku angielskim, znaki przestankowe oraz kilka innych znaków. Przykładowo dużą literę A reprezentuje kod o numerze 65. Jeżeli więc wpiszesz na klawiaturze literę A, w pamięci komputera zostanie zapisana liczba 65 (oczywiście za pomocą systemu binarnego). Przedstawiłem to na rysunku 1.13.



**Rysunek 1.13.** Litera A jest zapisana w pamięci jako liczba 65



**WSKAZÓWKA:** Skrót ASCII wymawia się „aski”.

Jeśli Cię to interesuje, to literę B reprezentuje liczba 66, literę C — liczba 67 itd. W dodatku A znajduje się tabela kodów ASCII oraz wartości, jakim one odpowiadają.

Zestaw znaków ASCII został stworzony na początku lat sześćdziesiątych XX wieku i ostatecznie zyskał aprobatę większości producentów sprzętu komputerowego. Jest to jednak zestaw bardzo ograniczony — można za jego pomocą zakodować tylko 128 znaków. Aby temu zaradzić, na początku lat dziewięćdziesiątych XX wieku stworzono nowy zestaw znaków, o nazwie **Unicode**. Jest to bardzo obszerny zestaw, który jednocześnie jest kompatybilny z systemem ASCII, ale można za jego pomocą kodować znaki z bardzo wielu języków. Obecnie Unicode jest standardowym systemem kodowania znaków w przemyśle komputerowym.

## Zapisywanie bardziej skomplikowanych liczb

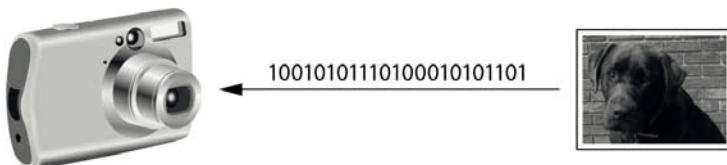
Przed chwilą wyjaśniłem Ci, jak w pamięci przechowywane są wartości liczbowe. Być może zwróciłeś wtedy uwagę, że za pomocą systemu binarnego można wyrażać tylko liczby całkowite — począwszy od 0. Za pomocą opisanej przeze mnie prostej techniki nie da się przedstawić liczb ujemnych i zmiennoprzecinkowych (np. 3,14159).

Komputery jednak potrafią zapisywać w pamięci zarówno liczby ujemne, jak i zmiennoprzecinkowe, ale aby to było możliwe, poza systemem binarnym korzysta się także z różnych metod kodowania. Na przykład liczby ujemne koduje się za pomocą techniki zwanej **uzupełnieniem do 2**, a ułamki — za pomocą **zapisu zmiennoprzecinkowego** (ang. *floating-point notation*). Nie musisz wiedzieć, jak działają te kodowania — wystarczy Ci wiedzieć, że służą one do zapisywania w systemie binarnym liczb ujemnych i ułamkowych.

## Inne typy danych

Komputery są często określane mianem urządzeń cyfrowych. Przymiotnika „cyfrowy” używamy w przypadku rzeczy, które w jakikolwiek sposób korzystają z liczb binarnych. **Cyfrowe dane** (ang. *digital data*) to dane zapisane w systemie binarnym, a **urządzenie cyfrowe** to urządzenie, które wykorzystuje podczas działania dane binarne. W tym podrozdziale omówię, w jaki sposób w systemie binarnym zapisywane są liczby i znaki, jednak komputery operują także na wielu innych typach danych cyfrowych.

Wyobraź sobie zdjęcia, jakie wykonujesz swoim aparatem cyfrowym. Obrazy te składają się z małutkich kolorowych kropek zwanych **pikselami** (słowo to pochodzi od angielskiego wyrażenia *picture element*). Na rysunku 1.14 pokazałem, że każdy piksel obrazu zamieniany jest na liczbę, która odpowiada jego kolorowi. Liczba ta jest zapisywana w pamięci za pomocą systemu binarnego.



**Rysunek 1.14.** Obraz cyfrowy jest zapisywany w systemie binarnym  
(zdjęcie dzięki uprzejmości Gaddis, Tony)

Muzyka, którą odtwarzasz z płyt CD, iPoda czy odtwarzacza MP3, też ma postać cyfrową. Utwór zapisany cyfrowo jest podzielony na małe kawałki zwane **próbkami**. Każda próbka jest zamieniana na wartość binarną, która z kolei może zostać zapisana w pamięci. Z im większej liczby próbek składa się dany utwór muzyczny, tym dokładniej odzwierciedla on oryginalne nagranie. Jedna sekunda utworu o jakości CD składa się z ponad 44000 próbek!



## Punkt kontrolny

- 1.9. Ile pamięci będziesz potrzebować, aby zapisać niewielką liczbę lub jedną literę alfabetu?
- 1.10. Jak inaczej określamy mały „przełącznik”, który można ustawić w pozycji włączonej bądź wyłączonej?
- 1.11. Jak nazywa się system liczbowy, w którym wartości zapisujemy za pomocą szeregu zer i jedynek?
- 1.12. Do czego służy ASCII?
- 1.13. Za pomocą którego systemu kodowania można przedstawić wszystkie znaki wszystkich języków świata?
- 1.14. Co oznaczają wyrażenia „dane cyfrowe” i „urządzenie cyfrowe”?

1.4

## W jaki sposób działa program

**WYJAŚNIENIE:** Procesor komputera potrafi zrozumieć jedynie instrukcje zapisane w języku maszynowym. Ponieważ język ten jest na tyle trudny, że człowiekowi bardzo ciężko byłoby napisać za jego pomocą cały program, stworzono inne języki programowania komputerów.

Wcześniej wspomniałem, że procesor jest najważniejszym elementem komputera, ponieważ to on odpowiada za wykonanie programu. Czasami mówi się, że procesor to „mózg komputera”, jednak jest to tylko popularna metafora, a sam procesor jest po prostu urządzeniem elektronicznym, którego zadaniem jest wykonywanie określonych operacji — w szczególności takich jak:

- odczytywanie danych z pamięci głównej;
- dodawanie do siebie dwóch liczb;
- odejmowanie jednej liczby od drugiej;
- mnożenie dwóch liczb;
- dzielenie jednej liczby przez drugą;
- przenoszenie danych z jednego obszaru pamięci do innego;
- sprawdzanie, czy dana wartość jest równa innej wartości;
- itp.

Zapewne wywnioskowałeś z tej listy, że procesor zajmuje się głównie wykonywaniem na danych prostych operacji. Jednak sam z siebie nie jest on w stanie wykonać żadnej operacji — trzeba mu powiedzieć, co ma rozbić, i właśnie to jest zadaniem programu. Program jest niczym innym jak zbiorem instrukcji, które ma wykonać procesor.

Każda instrukcja w programie to polecenie wykonania określonej operacji wyданie procesorowi. Oto przykład instrukcji, jaka może wystąpić w programie:

10110000

Zarówno dla mnie, jak i dla Ciebie jest to szereg zer i jedynek. Jednak dla procesora jest to polecenie wykonania określonej operacji<sup>1</sup>. Ponieważ procesor rozumie jedynie instrukcje zapisane w języku maszynowym, jest ona zapisana za pomocą zer i jedynek — polecenia w **języku maszynowym** (ang. *machine language*) są zawsze zapisane w systemie binarnym.

Dla każdej operacji, którą potrafi wykonać dany procesor, istnieje osobna instrukcja w języku maszynowym. Przykładowo istnieje osobna instrukcja dla dodawania liczb, osobna dla odejmowania liczb itd. Kompletny zestaw instrukcji, które potrafi wykonać dany procesor, nazywa się **listą rozkazów procesora** (ang. *instruction set*).



**UWAGA:** Obecnie istnieje wielu producentów procesorów komputerowych — najbardziej znany są firmy Intel, AMD i Motorola. Jeżeli spojrzysz na swój komputer, być może znajdziesz na nim naklejkę z logo danego producenta.

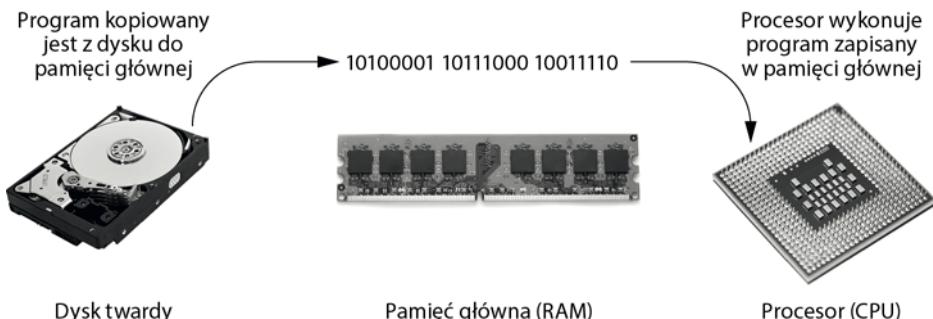
Każda rodzina procesorów charakteryzuje się inną listą rozkazów procesora, którą potrafią zrozumieć jedynie procesory tej rodziny. Przykładowo procesory Intel rozumieją instrukcje pochodzące z ich listy rozkazów, ale nie będą w stanie zrozumieć instrukcji przeznaczonych dla procesorów Motorola.

Instrukcja w języku maszynowym, którą przedstawiłem wcześniej, to tylko jeden z przykładów. Aby komputer mógł wykonać jakąś sensowną czynność, potrzebnych będzie znacznie więcej instrukcji. Ponieważ sam procesor rozumie jedynie bardzo elementarne rozkazy, będzie on musiał wykonać ich bardzo wiele, aby wykonać jakieś znaczące działanie. Przykładowo, jeśli chcemy, aby komputer obliczył wartość odsetek, jakie zostaną naliczone w ciągu roku na rachunku oszczędnościowym, procesor będzie musiał wykonać bardzo dużą liczbę instrukcji — ponadto instrukcje te będą miały być uruchomione w odpowiedniej kolejności. Nie jest niczym niezwykłym program składający się z tysięcy, a nawet wielu milionów instrukcji w języku maszynowym.

Programy są zazwyczaj zapisane na nośnikach danych — na przykład na dysku twardym. Kiedy instalujesz na komputerze program, jest on kopiowany z płyty CD lub z pliku pobranego z sieci na dysk twardy komputera.

Pomimo faktu, że dany program jest już zapisany na dysku twardym, za każdym razem, gdy procesor ma go wykonać, program ten musi najpierw zostać skopiowany do pamięci głównej komputera. Powiedzmy, że na dysku twardym znajduje się edytor tekstowy. Aby go uruchomić, musisz kliknąć myszką odpowiednią ikonkę. Spowoduje to skopiowanie programu z dysku twardego do pamięci głównej. Następnie procesor przystąpi do wykonywania kopii programu zapisanej już w pamięci głównej komputera. Proces ten przedstawiłem na rysunku 1.15.

<sup>1</sup> Przedstawiony przykład to prawdziwa instrukcja dla procesora Intel. Nakazuje ona przeniesienie wartości do procesora.

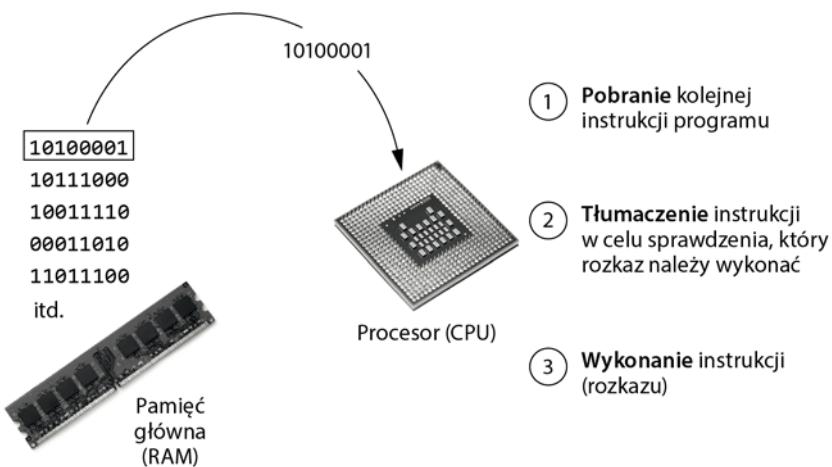


**Rysunek 1.15.** Program najpierw jest kopowany do pamięci głównej, a następnie uruchamiany (dzięki uprzejmości Lefteris Papaulakis/Shutterstock, Garsya/Shutterstock oraz marpan/Shutterstock)

Kiedy procesor przechodzi do wykonywania kolejnych instrukcji zapisanych w programie, dochodzi do tak zwanego **cyklu rozkazowego**. Cykl ten składa się z trzech faz i jest powtarzany dla każdej instrukcji w programie. Fazy te są następujące:

1. **Pobranie.** Program składa się z długiej sekwencji instrukcji w języku maszynowym. Pierwsza faza cyklu rozkazowego ma za zadanie pobrać (odczytać) kolejną instrukcję z pamięci głównej i przekazać ją do procesora.
2. **Tłumaczenie.** Instrukcja zapisana w języku maszynowym to liczba w systemie binarnym reprezentująca określony rozkaz, który ma wykonać procesor. W tej fazie procesor tłumaczy instrukcję pobraną z pamięci i sprawdza, który rozkaz powinien wykonać.
3. **Wykonanie.** W ostatniej fazie cyklu procesor wykonuje dany rozkaz.

Rysunek 1.16 ilustruje te trzy fazy.



**Rysunek 1.16.** Cykl rozkazowy (dzięki uprzejmości Garsya/Shutterstock i marpan/Shutterstock)

## Z języka maszynowego na język asemblera

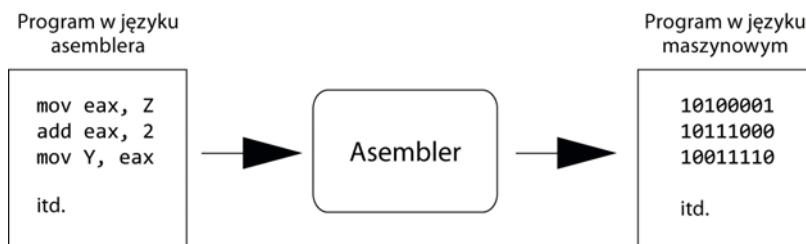
Komputer potrafi wykonywać tylko programy zapisane w języku maszynowym. Jak wspomniałem wcześniej, taki program może składać się z tysięcy, a nawet milionów instrukcji zapisanych w systemie binarnym i jego pisanie byłoby zajęciem bardzo nudnym i czasochłonnym. Byłoby to także bardzo trudne, gdyż postawienie zera lub jedynki w nieodpowiednim miejscu spowodowałoby błąd programu.

Procesor komputera rozumie jedynie instrukcje zapisane w języku maszynowym, jednak dla człowieka pisanie programu w tym języku jest niepraktyczne. Z tego powodu u zarania dziejów komputerów<sup>2</sup> wymyślono język zwany **językiem asemblera**. Zamiast z liczb w systemie binarnym korzysta się w nim z kilkuliterowych skrótów zwanych **mnemonikami**. Przykładowo w języku asemblera mnemonika add oznacza operację dodawania, mnemonika mul oznacza mnożenie, a mnemonika mov oznacza operację przeniesienia wartości do określonego miejsca w pamięci. Kiedy programista pisał program w języku asemblera, posługiwał się takimi skrótnymi mnemonikami zamiast liczbami w systemie binarnym.



**UWAGA:** Istnieje wiele różnych odmian języka asemblera. Wspomniałem wcześniej, że każda rodzina procesorów charakteryzuje się swoją własną listą rozkazów. Analogicznie zazwyczaj każda rodzina procesorów ma swój własny język asemblera.

Procesor komputerowy nie jest jednak w stanie wykonać programu zapisanego w języku asemblera — rozumie on jedynie język maszynowy. Z tego powodu stworzono specjalne programy zwane **asemblerami**, których zadaniem jest przetłumaczenie programu napisanego w języku asemblera na program w języku maszynowym. Proces ten przedstawiłem na rysunku 1.17. Program w języku maszynowym wygenerowany przez asembler może już zostać wykonany przez procesor.



**Rysunek 1.17.** Asembler tłumaczy program zapisany w języku asemblera na program w języku maszynowym

<sup>2</sup> Pierwszy język asemblera został stworzony najprawdopodobniej w latach czterdziestych XX wieku na Uniwersytecie Cambridge i korzystano z niego podczas pracy z komputerem EDSAC.

## Języki wysokiego poziomu

Pomimo faktu, że dzięki językowi asemblera nie ma już konieczności pisania programu w języku maszynowym, nie jest on pozbawiony pewnych wad. Język asemblera pełni głównie rolę zastępczą dla języka maszynowego i podobnie jak w jego przypadku wymaga od programisty dobrej znajomości danego procesora. Ponadto, aby napisać nawet najprostszy program w języku asemblera, trzeba użyć bardzo wielu instrukcji. Ponieważ z natury język asemblera jest bardzo bliski językowi maszynowemu, nazywa się go **językiem niskiego poziomu** (ang. *low-level language*).

W latach pięćdziesiątych XX wieku pojawiła się nowa generacja języków programowania zwanych **językami wysokiego poziomu** (ang. *high-level language*). To właśnie dzięki nim możemy tworzyć potężne i złożone programy, bez konieczności posiadania wiedzy na temat samego procesora, a jednocześnie nie potrzebujemy do tego ogromnej liczby niskopoziomowych instrukcji. Ponadto większość języków wysokiego poziomu zawiera łatwe do zrozumienia słowa. Przykładowo, kiedy programista piszący program w języku COBOL (jeden z pierwszych języków wysokiego poziomu, stworzony w latach pięćdziesiątych XX wieku) chciał wyświetlić na ekranie komputera komunikat „Hello world”, wystarczyło, że użył takiej oto instrukcji:

```
Display "Hello world"
```

Ta sama operacja w języku asemblera wymagałaby użycia wielu instrukcji i wiedzy na temat sposobu komunikowania się procesora z monitorem. Ten przykład pokazuje, że dzięki językom wysokiego poziomu programista może się skoncentrować na tym, jakie zadania powinien wykonać program, a nie na tym, jak procesor będzie wykonywał dany program.

Od tamtego czasu powstały tysiące języków wysokiego poziomu. W tabeli 1.1 zamieściłem listę kilku najbardziej popularnych. Jeśli uczysz się na kierunku związanym z branżą informatyczną, najprawdopodobniej poznasz jeden lub kilka spośród tych języków.

Każdy język wysokiego poziomu charakteryzuje się zestawem słów, które programista musi poznać, aby mógł z niego korzystać. Słowa występujące w danym języku wysokiego poziomu nazywa się **słowami kluczowymi** (ang. *key words*) lub **słowami zarezerwowanymi** (ang. *reserved words*). Każde słowo kluczowe ma określone znaczenie i nie może zostać wykorzystane do żadnego innego celu. Wcześniej pokazałem przykład instrukcji w języku COBOL, w której występuje słowo kluczowe *Display*, służące do wyświetlania komunikatu na ekranie komputera. W języku Python do tego samego celu służy słowo kluczowe *print*.

Poza słowami kluczowymi w językach programowania występują także **operatorы**, które wykonują na danych określone działania. Przykładowo w każdym języku występują operatory matematyczne, które służą do wykonywania działań arytmetycznych. W Javie i w wielu innych językach znak + jest operatorem służącym do dodawania do siebie dwóch liczb. Następujące wyrażenie dodaje do siebie liczby 12 i 75:

```
12 + 75
```

**Tabela 1.1.** Języki programowania

Język	Opis
Ada	Język Ada został stworzony w latach 70. XX w. i używany był głównie przez departament obrony Stanów Zjednoczonych. Język nazwano na cześć hrabiny Ady Lovelace, uważanej za osobę mającą ogromny wpływ na dziedzinę programowania.
BASIC	BASIC jest językiem ogólnego zastosowania, a jego nazwa to akronim od <i>Beginners All-purpose Symbolic Instruction Code</i> . Powstał w latach 60. XX w. jako prosty język do nauki programowania. Obecnie można znaleźć wiele odmian języka BASIC.
FORTRAN	Język FORmula TRANslator był pierwszym językiem programowania wysokiego poziomu. Powstał w latach 50. XX w. i służył do przeprowadzania skomplikowanych obliczeń matematycznych.
COBOL	Język COBOL został stworzony w latach 50. XX w., a jego nazwa jest akronimem od <i>Common Business-Oriented Language</i> . Służył głównie do tworzenia aplikacji biznesowych.
Pascal	Pascal powstał w 1970 r. i pierwotnie służył jako język dydaktyczny do nauki programowania. Język nazwano na cześć słynnego matematyka, fizyka i filozofa, Blaise'a Pascala.
C i C++	C i C++ (ta druga nazwa wymawiana jako „c plus plus”) to języki ogólnego zastosowania o ogromnych możliwościach stworzone przez firmę Bell Laboratories. Język C powstał w 1972 r., a język C++ w 1983 r.
C#	Nazwę C# wymawia się jako „c szarp”. Język ten służy do tworzenia aplikacji dla platformy .NET i powstał około 2000 r. z inicjatywy firmy Microsoft.
Java	Język Java został stworzony we wczesnych latach 90. XX w. przez firmę Sun Microsystems (której właścicielem jest obecnie Oracle). Można dzięki niemu tworzyć zarówno aplikacje działające na komputerze, jak i aplikacje internetowe działające na serwerach WWW.
JavaScript	Język JavaScript powstał w latach 90. XX w. i wykorzystywany jest do przetwarzania stron WWW. Mimo częściowego pokrewieństwa nazwy ma on niewiele wspólnego z językiem Java.
Python	Python to język ogólnego zastosowania stworzony w latach 90. XX w. Stał się popularnym narzędziem do tworzenia aplikacji biznesowych i naukowych.
Ruby	Ruby to język ogólnego zastosowania stworzony w latach 90. XX w. Zyskuje popularność jako język do tworzenia aplikacji działających na serwerach WWW.
Visual Basic	Visual Basic (często nazywany także VB) to język i środowisko programistyczne stworzone przez firmę Microsoft. Dzięki niemu można bardzo szybko tworzyć aplikacje dla systemu Windows. Język VB powstał w latach 90. XX w.

Poza słowami kluczowymi i operatorami każdy język charakteryzuje się także określoną **składnią** (ang. *syntax*), czyli zestawem zasad, których należy bezwzględnie przestrzegać, pisząc program w danym języku. Składnia wskazuje, w jaki sposób należy

używać w programie słów kluczowych, operatorów i znaków przestankowych. Podczas nauki danego języka programowania musisz poznać dokładnie jego składnię.

Poszczególne instrukcje, których będziesz używać w programie pisany w języku wysokiego poziomu, nazywamy **poleceniami** (ang. statements). Polecenie może się składać ze słów kluczowych, operatorów, znaków przestankowych oraz innych dopuszczalnych elementów, ułożonych w odpowiedniej kolejności tak, aby wykonać określoną operację.



**UWAGA:** Języki ludzkie także charakteryzują się składnią. Pamiętasz, jak podczas nauki języka polskiego poznawałeś zasady związane z podmiotami i orzeczeniami oraz innymi częściami zdania? Uczyłeś się wtedy właśnie składni języka polskiego.

Pomimo że ludzie podczas mówienia i pisania bardzo często łamią zasady składni, ich rozmówcy nie mają raczej problemu ze zrozumieniem. Niestety komputery są pozbawione tej cechy. Jeżeli pisząc program złamiesz chociaż jedną zasadę składni, program się nie uruchomi.

## Kompilatory i interprety

Ponieważ procesor komputera rozumie jedynie rozkazy w języku maszynowym, programy napisane w językach wysokiego poziomu muszą zostać najpierw przetłumaczone na język maszynowy. Po napisaniu programu w języku wysokiego poziomu programista musi skorzystać z kompilatora lub interpretera, aby przetłumaczyć program na język maszynowy.

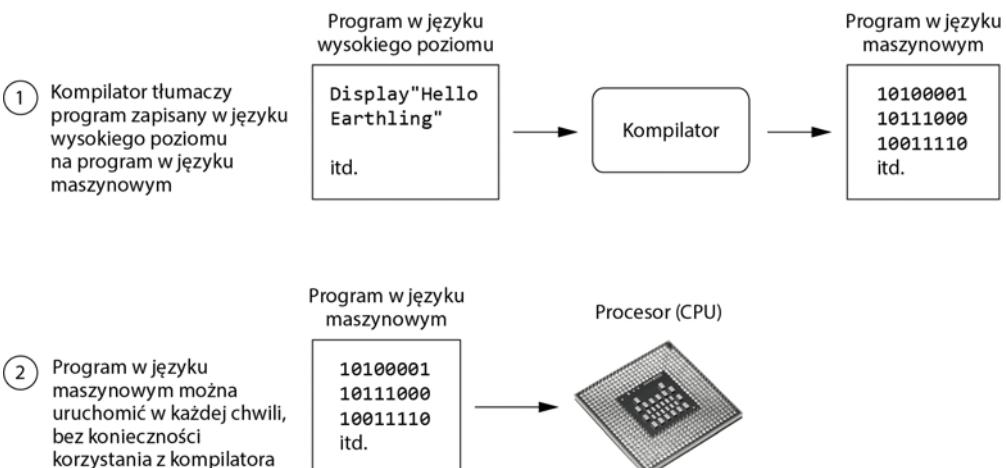
**Kompilator** to program, którego zadaniem jest przetłumaczenie programu napisanego w języku wysokiego poziomu na program w języku maszynowym. Taki program w języku maszynowym może być następnie uruchomiony w dowolnej chwili. Przedstawiłem to na rysunku 1.18. Na rysunku widać, że komplikacja i uruchomienie programu to dwa odrębne procesy.

**Interpreter** to program, który jednocześnie tłumaczy i wykonuje instrukcje w języku wysokiego poziomu. Interpreter odczytuje pojedyncze polecenie w programie, następnie tłumaczy je na język maszynowy i natychmiast wykonuje wynikowy kod maszynowy. Działanie to odbywa się dla każdego polecenia w programie. Proces ten pokazalem na rysunku 1.19. Ponieważ interpreter łączy tłumaczenie kodu i jego wykonanie, nie tworzą zazwyczaj osobnego programu w języku maszynowym.

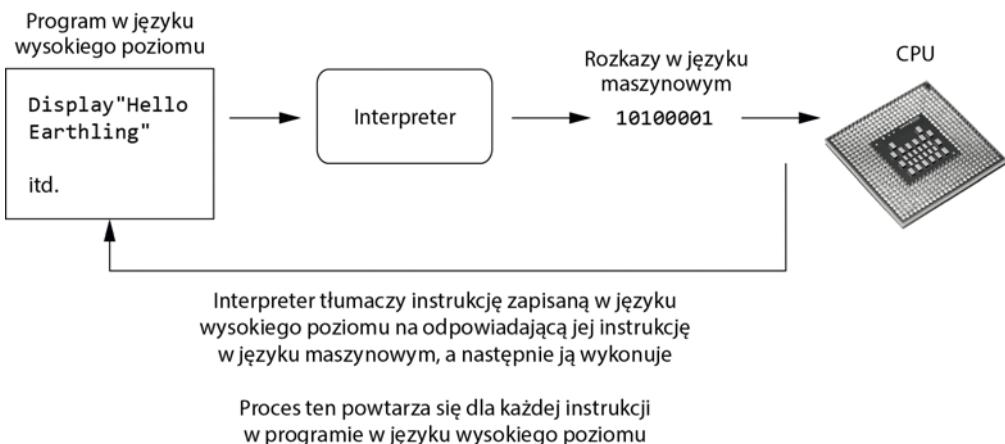


**UWAGA:** Ponieważ skompilowany program w momencie uruchomienia jest już w całości przetłumaczony na język maszynowy, działa on szybciej od programu uruchomionego za pomocą interpretera.

Polecenia w języku wysokiego poziomu, które tworzy programista, nazywamy **kodem źródłowym** lub po prostu **kodem**. Zazwyczaj programista pisze program w pliku tekstowym, po czym zapisuje go na dysku twardym. Następnie musi za pomocą kompilatora



**Rysunek 1.18.** Kompilowanie programu zapisanego w języku wysokiego poziomu i jego uruchomienie (dzięki uprzejmości marpan/Shutterstock)



**Rysunek 1.19.** Wykonywanie programu w języku wysokiego poziomu za pomocą interpretera (dzięki uprzejmości marpan/Shutterstock)

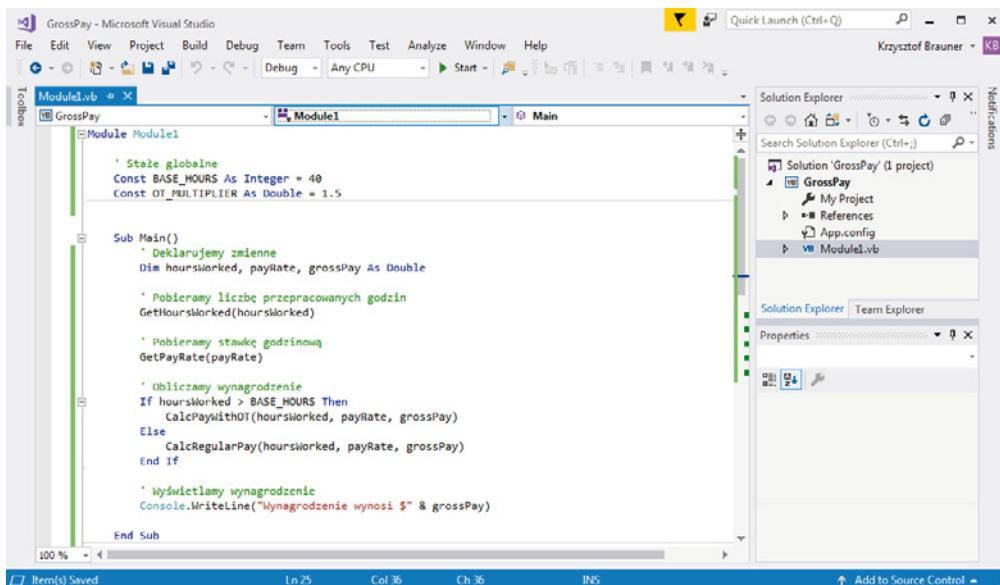
przetłumaczyć kod na program w języku maszynowym lub skorzystać z interpretera, aby przetłumaczyć i wykonać kod programu. Jeśli jednak w programie znajdą się jakieś błędy składniowe, nie będzie on mógł zostać przetłumaczony. **Błąd składniowy** (ang. *syntax error*) to błąd wynikający najczęściej z literówki, brakującego znaku przestankowego lub nieprawidłowego zastosowania operatora. W momencie wystąpienia takiego błędu kompilator lub interpreter wyświetli komunikat informujący o błędzie składniowym w kodzie programu. Programista musi wtedy poprawić błąd i ponownie przystąpić do tłumaczenia programu na język maszynowy.

## Zintegrowane środowisko programistyczne (IDE)

Chociaż do tworzenia programu można wykorzystać edytor tekstu taki jak Notatnik (będący częścią systemu operacyjnego Windows), większość programistów korzysta ze specjalnego oprogramowania, zwanego **zintegrowanym środowiskiem programistycznym** (ang. *integrated development environment* — IDE). Większość takich środowisk składa się z następujących elementów:

- edytor tekstu wyposażony w specjalne narzędzia ułatwiające pisanie poleceń w danym języku wysokiego poziomu;
- kompilator lub interpreter;
- narzędzia do testowania programów i wychwytywania błędów.

Na rysunku 1.20 przedstawiłem ekran programu Microsoft Visual Studio — popularnego środowiska służącego do tworzenia programów w językach C++, Visual Basic i C#. Innymi przykładami zintegrowanych środowisk programistycznych są pakiety Eclipse, NetBeans, Dev-C++ i jGRASP.



**Rysunek 1.20.** Zintegrowane środowisko programistyczne (dzięki uprzejmości Microsoft Corporation)



### Punkt kontrolny

- 1.15. W którym języku muszą być zapisane instrukcje, aby zrozumiał je procesor?
- 1.16. Do której pamięci kopowany jest program, zanim procesor zacznie go wykonywać?
- 1.17. Jaki proces następuje, kiedy procesor przechodzi do wykonywania kolejnych instrukcji zapisanych w programie?

- 1.18. Co to jest język asemblera?
- 1.19. Który rodzaj języków programowania umożliwia tworzenie potężnych i złożonych programów, bez konieczności posiadania wiedzy na temat procesora?
- 1.20. Każdy język programowania charakteryzuje się zestawem zasad, których należy przestrzegać, pisząc program w danym języku. Jak nazywa się ten zestaw zasad?
- 1.21. Jak nazywa się program, którego zadaniem jest przetłumaczenie programu zapisanego w języku wysokiego poziomu na program w języku maszynowym?
- 1.22. Jak nazywa się program, który jednocześnie tłumaczy i wykonuje kolejne instrukcje programu zapisanego w języku wysokiego poziomu?
- 1.23. Jak nazywa się błąd wynikający z literówki w słowie kluczowym, brakującego znaku przestankowego czy niewłaściwego użycia operatora?

## 1.5

## Rodzaje oprogramowania

**WYJAŚNIENIE:** Programy zazwyczaj zaliczają się do jednej z dwóch kategorii: oprogramowania systemowego lub oprogramowania użytkownika. Oprogramowanie systemowe składa się z szeregu programów, które odpowiadają za pracę komputera i powiększają jego możliwości. Oprogramowanie użytkowe sprawia, że komputer staje się przydatnym narzędziem w codziennej pracy.

Aby działać, komputer potrzebuje oprogramowania. Wszystko, co robi komputer — od momentu, gdy go włączysz, aż do chwili jego wyłączenia — odbywa się dzięki oprogramowaniu. Istnieją dwie kategorie oprogramowania: oprogramowanie systemowe i oprogramowanie użytkowe. Większość programów komputerowych można z łatwością zaklasyfikować do jednej z tych kategorii. Przyjrzymy się więc im bliżej.

### Oprogramowanie systemowe

Programy, które odpowiadają za podstawowe operacje komputera, nazywa się **oprogramowaniem systemowym**. Składają się na nie następujące programy:

- **System operacyjny.** System operacyjny to zestaw najważniejszych programów zapewniających pracę komputera. Zadaniem systemu operacyjnego jest sterowanie sprzętem, z jakiego składa się dany komputer, zarządzanie wszystkimi urządzeniami podłączonymi do komputera, umożliwienie odczytu i zapisu danych na nośnikach danych, umożliwienie uruchomienia na komputerze innych programów. Dzisiaj najczęściej używanymi systemami operacyjnymi są Windows, Mac OS, iOS, Android i Linux.

- **Programy narzędziowe.** Programy narzędziowe wykonują specjalistyczne zadania zapewniające bezpieczeństwo danych zapisanych na komputerze lub powiększające możliwości komputera. Przykładem programów narzędziowych są programy antywirusowe, programy do kompresji danych i tworzenia kopii zapasowych.
- **Narzędzia do tworzenia oprogramowania.** Narzędzia do tworzenia oprogramowania to programy, które umożliwiają programistom tworzenie, modyfikowanie i testowanie oprogramowania. Do tej kategorii zaliczają się na przykład asemblerы, kompilatory i interpreterы.

## Oprogramowanie użytkowe

Oprogramowaniem użytkowym nazywamy wszystkie programy, dzięki którym komputer staje się użytecznym narzędziem w codziennej pracy. To programy, przy których użytkownicy komputera spędzają najwięcej czasu. Na rysunku 1.1, znajdującym się na początku tego rozdziału, pokazalem ekranów dwóch bardzo popularnych programów użytkowych — procesora tekstu Microsoft Word oraz programu do tworzenia i wyświetlania prezentacji Microsoft PowerPoint. Innymi przykładami programów użytkowych są arkusze kalkulacyjne, programy do obsługi wiadomości e-mail, przeglądarki internetowe i gry.



### Punkt kontrolny

- 1.24. Jak nazywa się zestaw najważniejszych programów zapewniających pracę komputera?
- 1.25. Jak nazywają się programy, które wykonują specjalistyczne zadania, np. programy antywirusowe, programy do kompresji danych i programy tworzenia kopii zapasowych?
- 1.26. Do której kategorii oprogramowania należą takie programy jak arkusze kalkulacyjne, programy do obsługi wiadomości e-mail, przeglądarki internetowe i gry?

## Pytania kontrolne

### Test jednokrotnego wyboru

1. \_\_\_\_\_ to szereg instrukcji, które musi uruchomić komputer, aby wykonać określone działanie.
  - a) kompilator
  - b) program
  - c) interpreter
  - d) język programowania

2. Fizyczne urządzenia, w jakie wyposażony jest komputer, nazywamy \_\_\_\_\_.
  - a) sprzętem
  - b) oprogramowaniem
  - c) systemem operacyjnym
  - d) narzędziami
3. Elementem komputera odpowiedzialnym za wykonanie programu jest \_\_\_\_\_.
  - a) RAM
  - b) nośnik danych
  - c) pamięć główna
  - d) procesor
4. Dostępne obecnie procesory mają postać małych chipów zwanych \_\_\_\_\_.
  - a) ENIAC
  - b) mikroprocesorami
  - c) chipami pamięci
  - d) systemem operacyjnym
5. W momencie uruchomienia programu zarówno program, jak i dane, na których ten program pracuje, są przechowywane w \_\_\_\_\_.
  - a) nośniku danych
  - b) procesorze
  - c) pamięci głównej
  - d) mikroprocesorze
6. Jak nazywa się pamięć ulotna, która jest wykorzystywana jako tymczasowe miejsce przechowywania danych na czas działania programu?
  - a) RAM
  - b) nośnik danych
  - c) dysk twardy
  - d) napęd USB
7. Pamięć, która potrafi przechowywać dane przez dłuższy okres czasu — nawet po wyłączeniu komputera — nazywa się \_\_\_\_\_.
  - a) RAM
  - b) pamięcią główną
  - c) nośnikiem danych
  - d) pamięcią procesora
8. Urządzenie, które pobiera dane od użytkownika lub z innego urządzenia, a następnie przekazuje je do komputera, nazywa się \_\_\_\_\_.
  - a) urządzeniem wyjściowym
  - b) urządzeniem wejściowym
  - c) nośnikiem danych
  - d) pamięcią główną
9. Monitor komputerowy jest \_\_\_\_\_.
  - a) urządzeniem wyjściowym
  - b) urządzeniem wejściowym
  - c) nośnikiem danych
  - d) pamięcią główną

10. Jeden \_\_\_\_\_ pamięci wystarczy, aby zapisać w niej literę alfabetu angielskiego lub niewielką liczbę.
- bajt
  - bit
  - przełącznik
  - tranzystor
11. Jeden bajt składa się z ośmiu \_\_\_\_\_.  
a) procesorów  
b) instrukcji  
c) zmiennych  
d) bitów
12. W \_\_\_\_\_ systemie liczbowym liczby przedstawia się za pomocą szeregu zer i jedynek.  
a) szesnastkowym  
b) binarnym  
c) ósemkowym  
d) dziesiętnym
13. Wyłączony bit reprezentuje wartość \_\_\_\_\_.  
a) 1  
b) -1  
c) 0  
d) „nie”
14. Zestaw 128 kodów liczbowych reprezentujących litery alfabetu angielskiego, znaki przestankowe i inne znaki nazywa się \_\_\_\_\_.  
a) binarnym systemem liczbowym  
b) ASCII  
c) Unicode  
d) ENIAC
15. Rozbudowany zestaw kodowania znaków reprezentujący wszystkie znaki wielu języków świata nazywa się \_\_\_\_\_.  
a) binarnym systemem liczbowym  
b) ASCII  
c) Unicode  
d) ENIAC
16. Liczby ujemne zapisuje się w systemie binarnym za pomocą techniki zwanej \_\_\_\_\_.  
a) uzupełnieniem do 2  
b) zapisem zmiennoprzecinkowym  
c) ASCII  
d) Unicode
17. Ułamki zapisuje się w systemie binarnym za pomocą techniki zwanej \_\_\_\_\_.  
a) uzupełnieniem do 2  
b) zapisem zmiennoprzecinkowym  
c) ASCII  
d) Unicode

18. Małutkie kropki, z których składają się pliki graficzne, nazywa się \_\_\_\_\_.  
a) bitami  
b) bajtami  
c) zestawami kolorów  
d) pikselami
19. Jeśli przyjrzyisz się programowi napisanemu w języku maszynowym, to zobaczysz \_\_\_\_\_.  
a) kod w języku Java  
b) ciąg liczb binarnych  
c) słowa w języku angielskim  
d) obwód drukowany
20. Podczas fazy \_\_\_\_\_ cyklu rozkazowego procesor sprawdza, którą operację powinien wykonać.  
a) pobierania  
b) tłumaczenia  
c) wykonania  
d) występującej tuż przed wykonaniem polecenia
21. Komputer potrafi wykonywać tylko programy napisane w języku \_\_\_\_\_.  
a) Java  
b) asemblera  
c) maszynowym  
d) C++
22. Zadaniem \_\_\_\_\_ jest przetłumaczenie programu zapisanego w języku asemblera na program w języku maszynowym.  
a) asemblera  
b) kompilatora  
c) tłumacza  
d) interpretera
23. Słowa, z których składa się język programowania wysokiego poziomu, to \_\_\_\_\_.  
a) polecenia binarne  
b) mnemoniki  
c) polecenia  
d) słowa kluczowe
24. Zestaw zasad, których należy przestrzegać, pisząc program w danym języku, nazywa się \_\_\_\_\_.  
a) składnią  
b) interpunkcją  
c) słowami kluczowymi  
d) operatorami
25. \_\_\_\_\_ to program, którego zadaniem jest przetłumaczenie programu zapisanego w języku wysokiego poziomu na program w języku maszynowym.  
a) asembler  
b) kompilator  
c) tłumacz  
d) narzędzie

### **Prawda czy fałsz?**

1. Dostępne obecnie procesory to ogromne urządzenia mechaniczno-elektryczne składające się z lamp elektronowych i przełączników.
2. Pamięć główna to inaczej RAM.
3. Każda informacja, która ma zostać zapisana w pamięci komputera, musi mieć postać liczby w systemie binarnym.
4. Zdjęć wykonanych aparatem cyfrowym nie da się zapisać za pomocą liczb w systemie binarnym.
5. Jedynym językiem, jaki potrafi zrozumieć procesor komputera, jest język maszynowy.
6. Język asemblera jest językiem wysokiego poziomu.
7. Interpreter to program, którego zadaniem jest jednoczesne tłumaczenie i wykonywanie programu zapisanego za pomocą języka wysokiego poziomu.
8. Program, w którym występuje błąd składniowy, można skompilować i uruchomić.
9. Windows, Mac OS, iOS, Android i Linux to przykłady programów użytkowych.
10. Procesory tekstu, arkusze kalkulacyjne i programy do zarządzania wiadomościami e-mail to przykłady programów narzędziowych.

### **Krótką odpowiedź**

1. Dlaczego procesor jest najważniejszym komponentem komputera?
2. Jaką wartość reprezentuje włączony (ustawiony) bit? Jaką wartość reprezentuje wyłączony bit?
3. Jakim przymiotnikiem określisz urządzenie, które w jakiś sposób przetwarza dane w systemie binarnym?
4. Jaką nazwą noszą słowa, z których składa się język programowania wysokiego poziomu?
5. Jak nazywają się krótkie słowa występujące w języku asemblera?
6. Czym się różni kompilator od interpretera?
7. Oprogramowanie którego typu jest odpowiedzialne za sterowanie sprzętem, z jakiego składa się dany komputer?

### **Ćwiczenia**

1. W dodatku D znajdziesz informacje dotyczące zamiany liczb w systemie dziesiętnym na liczby w systemie binarnym. Wykorzystaj technikę przedstawioną w dodatku D i zamień do postaci binarnej następujące liczby:  
11  
65  
100  
255
2. Wykorzystaj wiedzę zdobytą podczas lektury tego rozdziału i zamień na liczby w systemie dziesiętnym następujące liczby binarne:  
1101  
1000  
101011

3. Przyjrzyj się tabeli kodów ASCII zamieszczonej w dodatku A i wskaż numery, jakie odpowiadają literom Twojego imienia.
4. Poszukaj w sieci informacji na temat historii języków programowania BASIC, C++, Java i Python, a następnie odpowiedz na następujące pytania:
  - Kto stworzył dany język?
  - Kiedy powstał dany język?
  - Czy twórcy danego języka stworzyli go w jakimś określonym celu? Jeśli tak, to co było tym celem?

# Dane wejściowe, przetwarzanie i dane wyjściowe

## TEMATYKA

- |  |   |
|--|---|
| 2.1 Projektowanie programu                                     | 2.5 Stałe nazwane                         |
| 2.2 Dane wejściowe, dane wyjściowe i zmienne                   | 2.6 Ręczne śledzenie programu             |
| 2.3 Przypisywanie wartości do zmiennych i wykonywanie obliczeń | 2.7 Dokumentowanie programu               |
| 2.4 Deklarowanie zmiennych i typy danych                       | 2.8 Projektowanie pierwszego programu     |
|  | 2.9 Rzut oka na języki Java, Python i C++ |

### 2.1

## Projektowanie programu

**WYJAŚNIENIE:** Zanim przystąpisz do pisania programu, musisz go dokładnie zaprojektować. Do projektowania programiści używają takich narzędzi jak pseudokod czy schematy blokowe, które przedstawiają model programu.

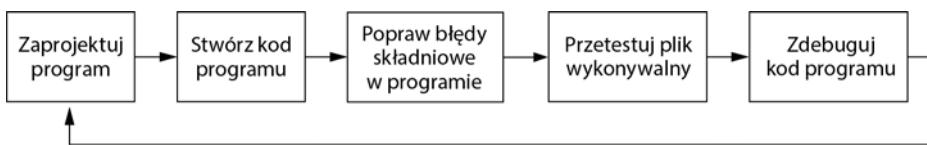
Z rozdziału 1. wiesz, że programiści do pisania programów używają najczęściej języków programowania wysokiego poziomu. Jednak każdy programista powie Ci także, że zanim zabierzesz się za pisanie programu, musisz go najpierw bardzo dokładnie zaprojektować. Gdy programista zaczyna pracę nad nowym projektem, nie od razu pisze kod — najpierw tworzy projekt programu.

Po stworzeniu projektu przychodzi czas na pisanie kodu w którymś z języków wysokiego poziomu. Przypomnij sobie, jak w rozdziale 1. wspomniałem, że każdym językiem rządzą pewne zasady, zwane składnią, i trzeba ich przestrzegać podczas pisania programu. Składnia danego języka programowania mówi o tym, w jaki sposób należy używać dostępnych w nim słów kluczowych, operatorów czy znaków przestankowych. Kiedy programista się pomyli, w programie wystąpi błąd składni.

Jeżeli w programie znajdzie się chociaż jeden, nawet najmniejszy błąd składniowy w postaci literówki w słowie kluczowym, kompilator lub interpreter wyświetli komunikat informujący, jaki to jest błąd. W praktyce niemal każdy nowo stworzony kod programu zawiera jakieś błędy składniowe, więc programista będzie musiał poświęcić trochę czasu na ich naprawienie. Kiedy już wszystkie błędy składniowe zostaną usunięte z programu, można go skompilować i przetłumaczyć na język maszynowy (lub, w zależności od języka, uruchomić za pomocą interpretera).

Gdy program ma już postać pliku wykonywalnego, należy go przetestować na obecność błędów logicznych. **Błąd logiczny** to błąd, który mimo swojej obecności umożliwi uruchomienie programu, ale spowoduje powstanie nieprzewidywalnych wyników. Typowym przykładem takich błędów są błędy matematyczne.

Kiedy programista wykryje obecność błędu logicznego w programie, przystępuje do jego **debugowania**. Zadanie to polega na odszukaniu kłopotliwego miejsca w kodzie programu i naprawieniu błędu. W pewnych sytuacjach programista może podczas debugowania programu stwierdzić, że należy go nieco zmienić. Cały ten proces nazywa się **procesem tworzenia programu**. Proces ten powtarza się aż do momentu, gdy program będzie pozbawiony błędów. Ilustruje to rysunek 2.1.



Rysunek 2.1. Proces tworzenia programu

Niniejsza książka poświęcona jest w całości omówieniu pierwszego z tych etapów, czyli projektowania programu. Faza procesu polegająca na projektowaniu jest chyba najważniejszą z faz. Można ją sobie wyobrazić jako budowanie fundamentów: jeśli wybudujesz dom na kiepskim fundamencie, musisz się liczyć z tym, że po pewnym czasie będziesz poświęcać sporo czasu na różne naprawy! Podobnie sprawa ma się z projektowaniem programu: jeśli kiepsko zaprojektujesz program, będziesz musiał poświęcić później dużo czasu na poprawianie błędów.

## Projektowanie programu

Cały proces projektowania programu można sprowadzić do dwóch kroków:

1. Określ, jakie zadanie ma wykonywać program.
2. Określ kroki, dzięki którym program wykona to zadanie.

Przyjrzymy się bliżej tym krokom.

### Określ, jakie zadanie ma wykonywać program

Określenie tego, jakie zadanie ma wykonywać program, jest sprawą kluczową. Programiści zazwyczaj określają to podczas kontaktu z klientem. Przez słowo „klient” rozumiemy osobę, grupę osób lub przedsiębiorstwo, które zleca stworzenie danego pro-

gramu. Może to być zarówno klient w tradycyjnym tego słowa znaczeniu — czyli ktoś, kto płaci za wykonanie programu — jak i szef lub kierownik któregoś z działów Twojej firmy. Niezależnie od tego, kim jest klient, będzie on musiał wytłumaczyć Ci, jakie ważne zadanie ma wykonywać program.

Aby określić, jakie zadanie ma wykonywać program, programista najczęściej prowadza rozmowę z klientem. Podczas takiej rozmowy klient musi dokładnie wytłumaczyć, co powinien robić program, a programista musi zapytać klienta o jak najwięcej szczegółów. Często dochodzi też do kolejnych rozmów z klientem, gdyż w trakcie pierwszej rozmowy trudno jest ustalić wszystkie szczegóły.

Programista analizuje informacje, które zgromadził w wyniku rozmów z klientem, i tworzy listę wymagań. **Wymaganie** to po prostu pojedyncza funkcja, którą będzie wykonywał program. Kiedy klient zaaprobuje listę wymagań, programista może przystąpić do kolejnego etapu pracy.



**WSKAZÓWKA:** Jeśli masz zamiar zostać zawodowym twórcą oprogramowania, Twoimi klientami będą osoby, które zlecają Ci napisanie danego programu — w ramach Twojej działalności zawodowej. Jednak dopóki jesteś studentem, Twoim klientem jest osoba prowadząca zajęcia! Jest prawie pewne, że na każdych zajęciach z programowania prowadzący przedstawi Ci zadanie programistyczne, które będziesz musiał rozwiązać. Bardzo ważne jest więc, abyś zrozumiał to, o co prosi Cię prowadzący zajęcia, ponieważ dzięki temu nie będziesz mieć problemów podczas pisania programów.

### Określ kroki, dzięki którym program wykona to działanie

Kiedy już określiłeś, jakie zadanie ma wykonywać program, możesz przystąpić do podziału tego zadania na mniejsze operacje. Przypomina to listę czynności, jaką przekażesz innej osobie, chcąc jej wytłumaczyć, w jaki sposób ma postępować, aby wykonać określone zadanie. Przykładowo Twoja siostra pyta Cię, jak zagotować wodę. Zakładając, że jest ona w odpowiednim wieku i możesz pozwolić jej na korzystanie z kuchenki, mógłbyś podzielić proces gotowania wody na następujące czynności:

1. Wlej odpowiednią ilość wody do garnka.
2. Postaw garnek na kuchence.
3. Włącz palnik.
4. Obserwuj wodę, aż zauważysz, że pojawiają się w niej bąbelki. Kiedy tak się stanie, oznacza to, że woda sięgotuje.

Jest to przykład **algorytmu**, czyli listy ściśle określonych operacji, które należy przedsięwziąć, aby wykonać określone zadanie. Zwróć uwagę, że operacje w powyższym algorytmie występują w określonej kolejności. Czyli operację 1. należy wykonać przed operacją 2. itd. Jeżeli Twoja siostra będzie postępowała dokładnie według tego opisu, wykonując kolejne operacje w prawidłowej kolejności, powinno się jej udać zagościć wodę.

W podobny sposób programista dzieli program na mniejsze zadania. W efekcie powstanie algorytmu składającego się z listy operacji, które należy przedsięwziąć, aby wykonać

określone zadanie. Założymy, że ktoś poprosił Cię o stworzenie programu, który będzie obliczał i wyświetlał wynagrodzenie pracownika. Oto kroki, które musi wykonać program:

1. Pobranie liczby przepracowanych godzin.
2. Pobranie stawki godzinowej pracownika.
3. Pomnożenie liczby przepracowanych godzin przez stawkę godzinową.
4. Wyświetlenie wyniku działania z punktu 3.

Oczywiście nie możemy takiego algorytmu przekazać komputerowi — musimy taką listę operacji przełożyć na kod programu. Programiści najczęściej używają do tego celu dwóch narzędzi: pseudokodu i schematów blokowych. Przyjrzymy się więc bliżej tym narzędziom.

## Pseudokod

W rozdziale 1. wyjaśniłem, że każdy język programowania charakteryzuje się zestawem zasad zwanych składnią, których musi przestrzegać programista, pisząc program w danym języku. Jeżeli programista złamie któryś z tych zasad, w programie wystąpi błąd składniowy i nie uda się skompilować ani uruchomić takiego programu. Kiedy tak się stanie, zadaniem programisty jest odnalezienie błędu i poprawienie go.

Ponieważ źródłem błędu składniowego może być nawet najmniejsza literówka, programista musi bardzo uważać podczas pisania programu. Z tego powodu część programistów stosuje technikę polegającą na tym, że przed napisaniem programu w wybranym języku zapisuje go za pomocą pseudokodu.

Słowo „pseudo” oznacza udawanie czegoś, czyli **pseudokod** to kod, który udaje, że jest kodem. To nieformalny język, który nie ma żadnej składni i nie służy do tego, aby go kompilować i uruchamiać. Natomiast programiści używają go do tworzenia modeli lub „makiet” programów. Ponieważ podczas pisania programu za pomocą pseudokodu nie trzeba trzymać się zasad składni, programista może skoncentrować całą uwagę na samym projekcie programu. Kiedy uda się utworzyć satysfakcyjną projekt za pomocą pseudokodu, można przystąpić do tłumaczenia go na język docelowy programu.

Oto przykład — przedstawiony wcześniej program do obliczania wynagrodzenia zapisany za pomocą pseudokodu:

```
Display "Wprowadź liczbę przepracowanych godzin."
Input hours
Display "Wprowadź stawkę godzinową."
Input payRate
Set grossPay = hours * payRate
Display "Wynagrodzenie pracownika wynosi ", grossPay, " zł."
```

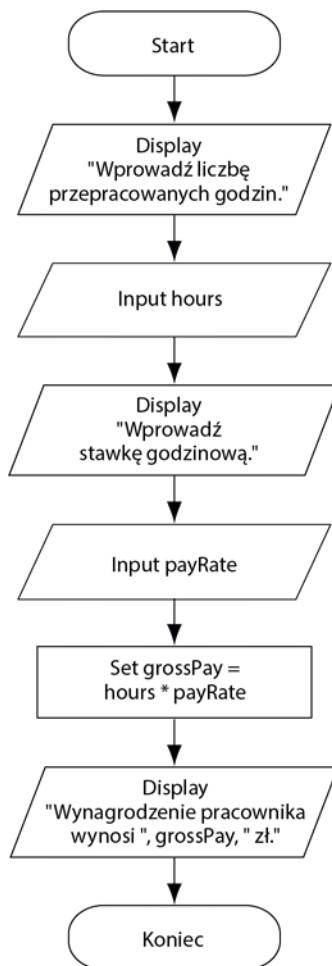
Każde polecenie w pseudokodzie odpowiada operacji, jaka powinna zostać wykonana w języku wysokiego poziomu. Przykładowo każdy język programowania zawiera instrukcje służące do wyświetlania komunikatów na ekranie monitora, odczytywania znaków wprowadzonych z klawiatury czy wykonywania działań matematycznych. Na ten moment nie musisz wiedzieć, o co chodzi w powyższym programie — po lekturze dalszej części rozdziału będziesz wiedzieć, do czego służy każda z występujących w nim instrukcji.



**UWAGA:** Podczas lektury książki pamiętaj, że przykłady, które w niej występują, są zapisane za pomocą pseudokodu, a nie któregoś z języków programowania. Pseudokod to po prostu ogólny sposób, w jaki zapisuje się algorytmy bez potrzeby przestrzegania składni języka. Jeżeliomylkowo wprowadzisz program napisany w pseudokodzie do edytora któregoś z języków programowania, np. Pythona czy Visual Basica, edytor zgłosi błędy.

## Schematy blokowe

Schematy blokowe to drugie narzędzie, za pomocą którego można projektować programy. **Schemat blokowy** to diagram, w którym za pomocą symboli graficznych przedstawia się kolejne operacje, jakie musi wykonać program. Na rysunku 2.2 przedstawiłem schemat blokowy programu obliczającego wynagrodzenie pracownika.



Rysunek 2.2. Schemat blokowy programu do obliczania wynagrodzenia pracownika

Zwróć uwagę, że na schemacie występują trzy różne typy symboli: owale, równolegloboki i prostokąty. Owale, które pojawiają się na górze i na dole schematu, to **bloki graniczne**. Symbol **Start** wskazuje początek programu, a symbol **Koniec** wskazuje koniec programu.

Pomiędzy blokami granicznymi pojawiają się równolegloboki, czyli **bloki wejścia/wyjścia**, i prostokąty, czyli **bloki operacyjne**. Każdy z tych symboli odpowiada jednej operacji w programie. Symbole połączone są strzałkami, które wskazują „przepływ” programu. Aby wykonać operacje we właściwej kolejności, należy prześledzić schemat w kierunku wskazywanym przez strzałki — od bloku *Start* aż do bloku *Koniec*. W dalszej części rozdziału przyjrzymy się bliżej każdemu z tych symboli. W dodatku B znajdziesz omówienie wszystkich symboli schematów blokowych, którymi będę się posługiwał w tej książce.

Schematy blokowe można tworzyć na kilka różnych sposobów. Twój wykładowca najprawdopodobniej poinformuje Cię na zajęciach, który z nich jest preferowany przez niego. Najprostszym i najtańszym sposobem jest na pewno narysowanie schematu blokowego ręcznie za pomocą ołówka i kartki papieru. Jeśli chcesz, aby rysowane przez Ciebie schematy wyglądały bardziej profesjonalnie, odwiedź najbliższy sklep z artykułami biurowymi (lub sklepik na uczelni), gdzie powinieneś znaleźć szablony do schematów blokowych — są to małe plastikowe płytki z wyciętymi symbolami używanymi w schematach blokowych. Dzięki szablonowi możesz bardzo łatwo odrysować symbole na kartce papieru.

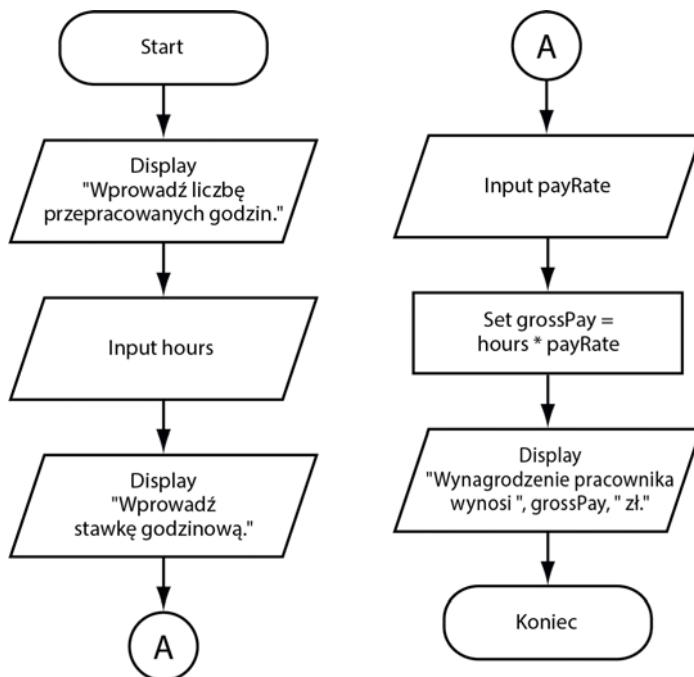
Wadą ręcznego rysowania schematów blokowych jest to, że w przypadku pomyłki trzeba będzie użyć gumki, a często nawet przrysować całą kartkę na nowo. Bardziej wydajnym i profesjonalnym podejściem jest skorzystanie z oprogramowania do tworzenia schematów blokowych. Dostępnych jest wiele różnych pakietów oprogramowania, dzięki którym można tworzyć schematy blokowe.

## Łączniki schematów blokowych

Bardzo często cały schemat blokowy będzie tak długi, że nie zmieści się na jednej stronie. Niekiedy można temu zaradzić, dzieląc schemat na dwa lub większą liczbę mniejszych i umieszczając je obok siebie na jednej stronie. Kiedy skorzystasz z tego rozwiązania, będziesz musiał użyć **symboli łączników wewnętrznych**, dzięki którym połączysz poszczególne fragmenty schematu znajdujące się na jednej stronie. Symbol łącznika wewnętrznego jest reprezentowany przez mały okrąg z wpisaną w nim literą lub liczbą. Na rysunku 2.3 przedstawiłem przykład schematu blokowego z symbolami łączników wewnętrznych.

Na rysunku 2.3 symbol łącznika **(A)** wskazuje miejsce, w którym kończy się pierwszy fragment schematu blokowego, a także miejsce, w którym rozpoczyna się kolejny fragment schematu.

Kiedy schemat blokowy okaże się tak duży, że nie uda się go zmieścić na jednej stronie, można go podzielić na fragmenty i umieścić je na osobnych stronach. Należy wtedy skorzystać z **symboli łączników zewnętrznych** i połączyć za ich pomocą fragmenty



**Rysunek 2.3.** Schemat blokowy z symbolami łączników wewnętrznych

schematu blokowego. Symbol łącznika zewnętrznego wygląda jak połączenie prostokąta i trójkąta, z wpisanym wewnątrz numerem strony. Jeżeli symbol łącznika zewnętrznego znajduje się na końcu fragmentu schematu, wówczas numer wskazuje stronę, na której znajduje się dalszy fragment schematu. Jeżeli symbol łącznika zewnętrznego znajduje się na początku fragmentu schematu, wówczas numer wskazuje stronę, na której umieszczony jest wcześniejszy fragment schematu. Na rysunku 2.4 przedstawiłem przykład. Widać na nim, że schemat blokowy z lewej strony umieszczony jest na stronie o numerze 1. Na dole schematu widoczny jest łącznik zewnętrzny, który wskazuje, że dalsza część schematu znajduje się na stronie o numerze 2. Na schemacie po prawej stronie (umieszczonym na stronie 2) łącznik zewnętrzny widoczny u góry wskazuje, że poprzedni fragment schematu blokowego znajduje się na stronie 1.

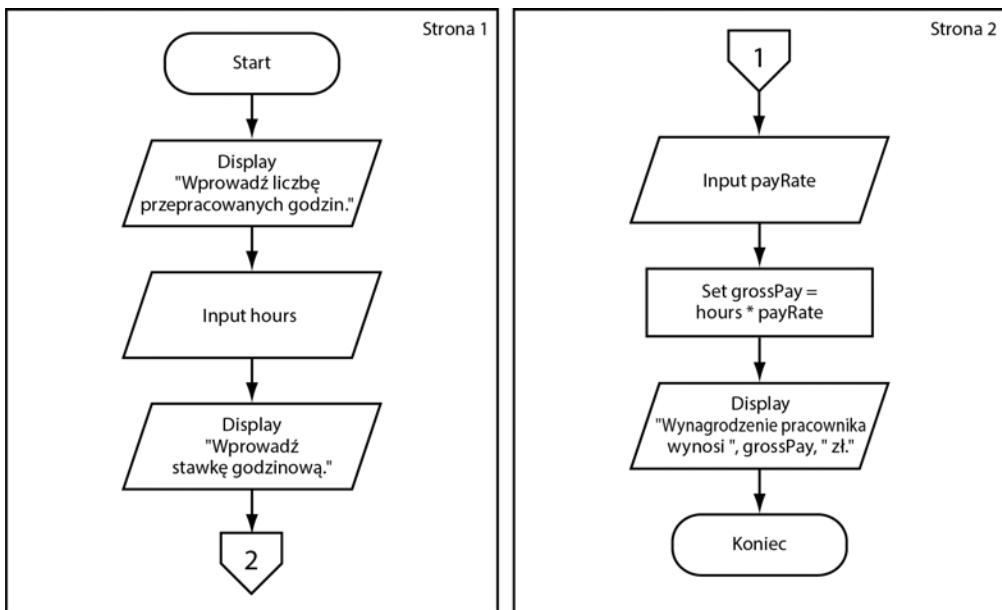


**UWAGA:** W zależności od książki lub pakietu oprogramowania symbole schematów blokowych mogą się między sobą nieco różnić. Jeśli będziesz korzystać z któregoś ze specjalistycznych pakietów do rysowania schematów blokowych, być może zauważysz niewielkie różnice w stosunku do symboli, jakie zaprezentowałem w książce.



## Punkt kontrolny

- 2.1. Kto to jest klient?
- 2.2. Co to są wymagania?



**Rysunek 2.4.** Schemat blokowy z symbolami łączników zewnętrznych

- 2.3. Co to jest algorytm?
- 2.4. Co to jest pseudokod?
- 2.5. Co to jest schemat blokowy?
- 2.6. Co oznaczają w schemacie blokowym następujące symbole?
  - owal
  - równoległobok
  - prostokąt

## 2.2

## Dane wejściowe, dane wyjściowe i zmienne

**WYJAŚNIENIE:** Dane wyjściowe to dane, które program generuje lub wyświetla na ekranie. Dane wejściowe to dane, które program przyjmuje. Kiedy program pobierze dane, zapisuje je w zmiennych, które mają swoją nazwę i położenie w pamięci komputera.

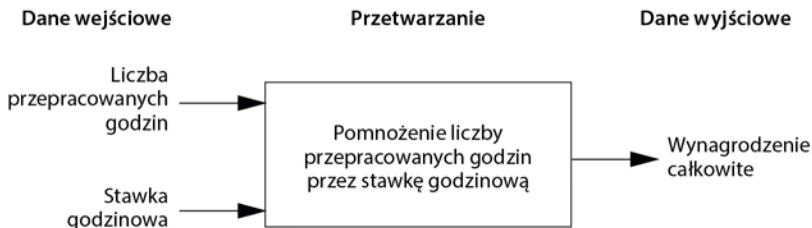
Program komputerowy wykonuje zazwyczaj następujące trzy zadania:

1. Pobiera dane.
2. Wykonuje na danych jakieś operacje.
3. Generuje wynik.

Dane wejściowe to wszelkie dane, które trafiają do programu w czasie jego działania. Jedną z najpopularniejszych form danych wejściowych są dane wprowadzane z klawiatury. Gdy program pobierze dane, najczęściej przystępuje do ich przetwarzania —

na przykład wykonuje na nich jakieś działania matematyczne. Wynik tych operacji jest następnie zwracany przez program w formie danych wyjściowych.

Na rysunku 2.5 przedstawiłem te trzy zadania na przykładzie programu do obliczania wynagrodzenia, który omawiałem już wcześniej. Dane wejściowe w tym przypadku to liczba przepracowanych godzin i stawkę godzinową.



**Rysunek 2.5.** Dane wejściowe, wyjściowe i przetwarzanie danych na przykładzie programu do obliczania wynagrodzenia

Przetwarzanie danych w tym przykładzie sprowadza się do pomnożenia liczby przepracowanych godzin przez stawkę godzinową. Wynik obliczeń jest prezentowany na ekranie jako dane wyjściowe.

## Tabelki IPO

Tabelki IPO to proste, lecz efektywne narzędzie, z którego często korzystają programiści podczas projektowania programów. IPO to skrót od angielskich wyrazów *input*, *processing* i *output* (czyli w języku polskim: wejście, przetwarzanie i wyjście), a tabela IPO prezentuje dane wejściowe i wyjściowe oraz przetwarzanie danych, jakie ma mieć miejsce w projektowanym programie. Te trzy elementy są zazwyczaj umieszczone w kolumnach. Kolumna „wejście” wskazuje w sposób opisowy, jakiego typu danych spodziewa się program na wejściu. Kolumna „przetwarzanie” opisuje operację lub kilka operacji, które program będzie wykonywał. Kolumna „wyjście” służy do opisu danych wyjściowych, które program będzie zwracał. Na rysunku 2.6 przedstawiłem tabelkę IPO dla programu obliczającego wynagrodzenie.

Tabela IPO dla programu obliczającego wynagrodzenie		
Wejście	Przetwarzanie	Wyjście
Liczba przepracowanych godzin Stawka godzinowa	Pomnożenie liczby przepracowanych godzin przez stawkę godzinową. Wynik jest równy wynagrodzeniu całkowitemu.	Wynagrodzenie całkowite

**Rysunek 2.6.** Tabela IPO dla programu obliczającego wynagrodzenie

W dalszej części tego podrozdziału przyjrzymy się kilku prostym programom, które przyjmują i zwracają dane. W kolejnym punkcie omówię w jaki sposób można przetwarzać dane.

## Wyświetlanie wyniku na ekranie

Jednym z najbardziej podstawowych zadań, jakie możesz wykonać w programie, jest wyświetlenie wyniku na ekranie monitora. Jak już wcześniej wspomniałem, wszystkie języki wysokiego poziomu umożliwiają wyświetlanie wyniku. W tej książce będę posługiwał się w tym celu poleceniem w pseudokodzie o nazwie `Display`. Oto przykład:

`Display "Witaj, świecie!"`

Polecenie to wyświetla na ekranie monitora tekst `Witaj, świecie!`. Zwróć uwagę, że zdanie `Witaj, świecie!` występujące zaraz za słowem `Display` umieściłem w cudzysłówach. Jednak program nie wyświetli na ekranie znaków cudzysłowu — wskazują one po prostu, gdzie rozpoczyna się i kończy tekst, który mamy zamiar wyświetlić.

Załóżmy, że prowadzący zajęcia poprosi Cię o napisanie za pomocą pseudokodu programu, który będzie wyświetlał Twoje imię i nazwisko oraz adres zamieszkania. Program taki będzie wyglądał jak ten na listingu 2.1.

### Listing 2.1

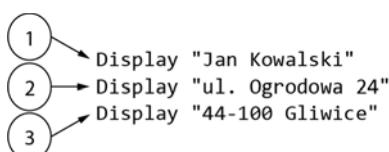


```
Display "Jan Kowalski"
Display "ul. Ogrodowa 24"
Display "44-100 Gliwice"
```

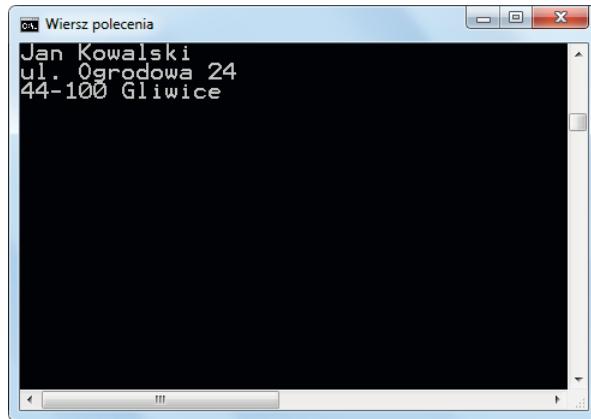
Ważne jest, aby zrozumieć, że instrukcje w powyższym programie zostaną wykonane w takiej kolejności, w jakiej je zapisano — od góry do dołu. Zilustrowałem to na rysunku 2.7. Jeżeli przetłumaczylibyśmy ten program na któryś z języków programowania, a następnie byśmy go uruchomili, najpierw wykonaliby się pierwsza instrukcja, potem druga instrukcja, a na końcu trzecia instrukcja. Jeśli chcesz mieć wyobrażenie, w jaki sposób dane będą prezentowały się na ekranie komputera, spójrz na rysunek 2.8. Każda z instrukcji `Display` wyświetla kolejną linię tekstu.



**UWAGA:** W tej książce jako instrukcji wyświetlającej wynik na ekranie używam słowa `Display`, niemniej inni programiści mogą używać do tego celu innych słów, na przykład `Print` lub `Write`. W pseudokodzie nie ma żadnych zasad nakazujących Ci stosowanie konkretnego słowa.

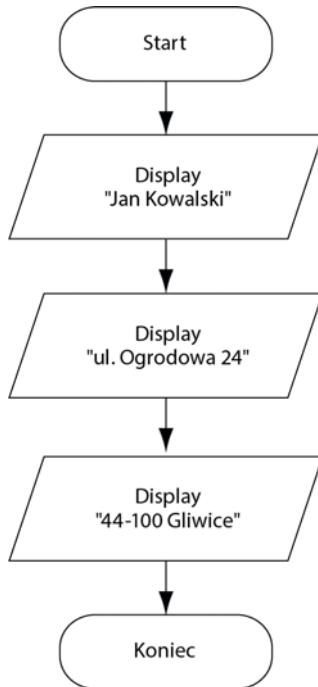


**Rysunek 2.7.** Polecenia wykonują się po kolei (dzięki uprzejmości firmy Microsoft)



**Rysunek 2.8.** Wynik działania programu z listingu 2.1 (dzięki uprzejmości firmy Microsoft)

Na rysunku 2.9 przedstawiłem schemat blokowy omawianego programu. Zwróć uwagę, że pomiędzy symbolami bloków granicznych umieszczone są trzy równoległy boki. Romb może reprezentować zarówno wprowadzanie, jak i wyświetlanie danych. W tym przypadku wszystkie trzy równoległy boki reprezentują wyświetlanie danych — każdy z nich odpowiada jednej instrukcji `Display`.



**Rysunek 2.9.** Schemat blokowy programu z listingu 2.1

## Struktury sekwencyjne

Wspomniałem wcześniej, że instrukcje w programie z listingu 2.1 będą uruchamiane w takiej kolejności, w jakiej zostały zapisane — z góry na dół. **Strukturą sekwencyjną** nazywamy grupę instrukcji, które będą uruchamiane w takiej kolejności, w jakiej zostały zapisane w programie. Tak naprawdę, to wszystkie przykłady, które przedstawię w tym rozdziale, są strukturami sekwencyjnymi.

**Struktura** (lub **struktura sterująca**) to konstrukcja logiczna, której celem jest sterowanie kolejnością, w jakiej będzie uruchamiana grupa instrukcji. W latach sześćdziesiątych XX wieku grupa matematyków udowodniła, że do napisania jakiegokolwiek programu wystarczą tylko trzy struktury. Najprostszą z nich jest struktura sekwencyjna. W dalszej części książki dowiesz się o jeszcze dwóch innych strukturach — warunkowej i cyklicznej.

## Ciągi znaków i ich literaty

Niemal każdy program podczas działania wykonuje jakieś operacje na danych określonego typu. Przykładowo program z listingu 2.1 wykonuje operacje na trzech następujących danych:

```
"Jan Kowalski"  
"ul. Ogrodowa 24"  
"44-100 Gliwice"
```

Dane te mają postać sekwencji znaków. W nomenklaturze programistycznej sekwencję znaków nazywamy **ciągiem znaków**. Kiedy ciąg znaków występuje w programie (lub w pseudokodzie, jak na listingu 2.1), nazywamy go **literałem ciągu znaków**. Literal ciągu znaków oznaczamy zazwyczaj w programie lub pseudokodzie za pomocą cudzysłówów. Jak wspomniałem wcześniej, cudzysłowy wskazują, gdzie dany ciąg znaków się rozpoczyna, a gdzie kończy.

W tej książce do oznaczania literalów ciągów znaków będę używał cudzysłowów (""). W przypadku większości języków programowania jest podobnie, jednak w kilku językach stosuje się apostrofy (').

## Zmienne a dane wejściowe

Bardzo często program musi zapisywać dane w pamięci komputera, aby mógł później na nich wykonać określone operacje. Przykładowo wyobraź sobie, jak wygląda robienie zakupów przez internet: wchodzisz na stronę sklepu i dodajesz do koszyka artykuły, które chcesz kupić. Gdy dodajesz kolejne artykuły do koszyka, komputer zapisuje dane na ich temat w pamięci. Następnie, gdy klikniesz przycisk *Do kasy*, program działający na serwerze sklepu internetowego oblicza wartość wszystkich artykułów znajdujących się w koszyku i dolicza do niej koszty wysyłki. Kiedy program wykona te obliczenia, wynik także zapisuje w pamięci komputera.

W programach informacje przechowywane są w zmiennych. **Zmienna** (ang. *variable*) to nic innego jak miejsce w pamięci komputera, które jest reprezentowane przez określoną nazwę. Przykładowo w przypadku programu obliczającego wartość podatku może to być zmienna o nazwie *tax* i to w niej zostanie zapisana obliczona kwota. W przypadku programu obliczającego odległość dzielącą Ziemię od jakiejś dalekiej gwiazdy wynik można zapisać w zmiennej o nazwie *distance*.

W tym punkcie omówię podstawową operację wejścia, czyli wprowadzanie danych z klawiatury komputera. Kiedy program odczyta dane z klawiatury, zazwyczaj zapisuje je w zmiennej, aby później mógł z nich skorzystać. W przypadku pseudokodu do odczytywania danych z klawiatury posłuży nam instrukcja *Input*. Jako przykład jej wykorzystania przedstawię polecenie, które widziałeś już wcześniej w programie obliczającym wynagrodzenie:

Input hours

Słowo *Input* to instrukcja nakazująca programowi odczytać dane wprowadzone na klawiaturze. W wyniku wykonania tej instrukcji zadziażą się dwie rzeczy:

- Nastąpi chwilowe wstrzymanie pracy programu i będzie on oczekiwany, aż użytkownik wprowadzi jakieś dane na klawiaturze, a następnie naciśnie klawisz *Enter*.
- Po naciśnięciu przez użytkownika klawisza *Enter* wprowadzone przez niego dane zostaną zapisane w zmiennej o nazwie *hours*.

Na listingu 2.2 widoczny jest zapisany pseudokodem prosty program, w którym zadeemonstrowałem wykorzystanie instrukcji *Input*. Zanim przyjrzymy się temu programowi, chciałem wspomnieć o kilku rzeczach. Po pierwsze, jak zapewne zauważysz, poszczególne linie kodu są ponumerowane. Jednak numery linii nie są częścią pseudokodu — będę się jednak do nich odnosić, aby wskazać konkretne miejsca w programie. Po drugie, wynik działania programu przedstawiony jest zaraz pod pseudokodem. Od tej chwili będę przedstawiał w ten sposób wszystkie przykłady zapisane pseudokodem.

### **Listing 2.2**



```

1 Display "Ile masz lat?"
2 Input age
3 Display "Oto wartość, którą wprowadziłeś:"
4 Display age
  
```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Ile masz lat?

**24 [Enter]**

Oto wartość, którą wprowadziłeś:

**24**

Polecenie w linii 1. wyświetla na ekranie ciąg znaków "Ile masz lat?". Następnie w linii 2. pojawia się instrukcja oczekująca, aż użytkownik wprowadzi wartość. Wartość ta jest zapisywana w zmiennej o nazwie *age*. W przykładowym wywołaniu programu użytkownik wprowadził wartość 24. Instrukcja w linii 3. wyświetla ciąg

## 64 Rozdział 2. Dane wejściowe, przetwarzanie i dane wyjściowe

znaków "Oto wartość, którą wprowadziłeś:", a instrukcja w linii 4. wyświetla na ekranie wartość zapisaną w zmiennej age.

Zwróć uwagę, że w linii 4. nie ma żadnych cudzysłówów po obu stronach słowa age. Jeśli ujęlibyśmy słowo age w cudzysłowy, wskazalibyśmy programowi, że chcemy, aby wyświetlił na ekranie słowo age zamiast wartości zapisanej w zmiennej age. Innymi sowy, poniższe polecenie wyświetla wartość zapisaną w zmiennej age:

```
Display age
```

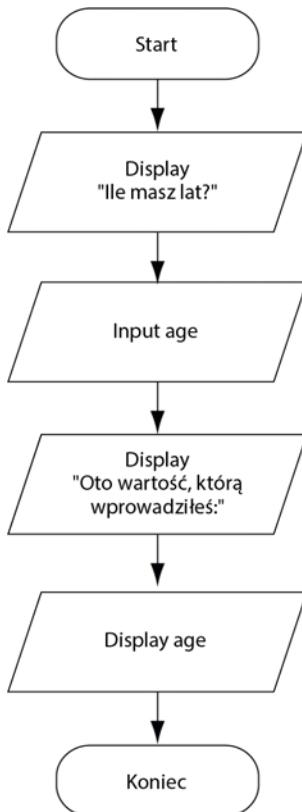
Natomiast poniższe polecenie wyświetla po prostu słowo age:

```
Display "age"
```



**UWAGA:** W punkcie tym wspomniałem o użytkowniku. Przez słowo *użytkownik* rozumiem hipotetyczną osobę, która będzie korzystała z programu i wprowadzała do niego dane. Niekiedy taką osobę nazywa się także **użytkownikiem końcowym**.

Na rysunku 2.10 widoczny jest schemat blokowy programu z listingu 2.2. Zauważ, że instrukcja Input także jest reprezentowana przez równoleglobok.



**Rysunek 2.10.** Schemat blokowy programu z listingu 2.2

## Nazwy zmiennych

Wszystkie języki programowania wysokiego poziomu umożliwiają nadawanie nazw zmiennym. Niemniej nadając nazwę zmiennej, zazwyczaj nie masz pełnej swobody pod tym względem. Każdy z języków charakteryzuje się określonym zbiorem reguł, których trzeba przestrzegać, tworząc nazwy zmiennych.

Mimo że zasady nazewnictwa zmiennych różnią się nieco w zależności od języka, występują w nich pewne wspólne ograniczenia:

- Nazwa zmiennej musi się składać z jednego słowa (nie może zawierać znaków spacji).
- W większości języków w nazwie nie mogą występować znaki przestankowe. Dobrą praktyką jest wykorzystywanie w nazwach zmiennych tylko liter i liczb.
- W większości języków pierwszym znakiem w nazwie zmiennej nie może być cyfra.

Poza wspomnianymi zasadami rządzącymi danym językiem programowania należy trzymać się reguły, aby nadawać zmiennym takie nazwy, które będą wskazywały, do czego dana zmienna służy. Przykładowo zmienna, w której zapisana będzie wartość temperatury, może się nazywać `temperature`, a zmienna, w której będzie zapisana prędkość samochodu, może się nazywać `speed`. Być może kuszące wyda Ci się nadawanie zmiennym nazw takich jak `x` lub `b2`, ale nazwy te absolutnie nie informują, do czego dana zmienna służy.

Ponieważ nazwa zmiennej powinna wskazywać, do czego ona służy, programiści często nadają zmiennym nazwy składające się z kilku wyrazów. Przyjrzyjmy się następującym nazwom zmiennych:

```
grosspay
payrate
hotdogsoldtoday
```

Niestety w tym przypadku poszczególne wyrazy w nazwach zmiennych nie zostały od siebie w żaden sposób oddzielone i ich odczytanie jest dosyć trudne. Ponieważ nie możemy w nazwie zmiennej używać znaków spacji, musimy znaleźć jakiś inny sposób na oddzielenie od siebie poszczególnych wyrazów w nazwie zmiennej — tak aby nazwa stała się bardziej czytelna.

Jednym ze sposobów jest zastąpienie spacji znakiem podkreślenia. Poniższe nazwy zmiennych będzie się odczytywało znacznie łatwiej niż te, które przedstawiłem wcześniej:

```
gross_pay
pay_rate
hot_dogs_sold_today
```

Innym sposobem poprawienia czytelności nazw zmiennych jest stosowanie systemu zapisu o nazwie **camelCase**. Tak zapisane nazwy zmiennych charakteryzują się dwiema cechami:

- nazwa zmiennej rozpoczyna się od małej litery;
- drugie słowo i kolejne słowa w nazwie zmiennej rozpoczynają się od dużej litery.

Oto przykłady nazw zmiennych zapisanych za pomocą systemu camelCase:

```
grossPay  
payRate  
hotDogsSoldToday
```

Ponieważ zapis ten jest bardzo popularny wśród programistów, będę go stosował w dalszej części książki. Tak naprawdę używałem tego zapisu już we wcześniejszych przykładach — program do obliczania wynagrodzenia zamieszczony na początku rozdziału korzystał ze zmiennej o nazwie payRate. W dalszej części rozdziału użyję zmiennych o nazwach originalPrice i salePrice w programie z listingu 2.9 oraz zmiennych futureValue i presentValue w programie z listingu 2.11.



**UWAGA:** Nazwa zapisu camelCase wzięła się stąd, że duże litery w nazwie zmiennej wyglądają jak garby wielbląda.

## Wyświetlanie kilku elementów za pomocą jednej instrukcji Display

Jeśli przyjrzyisz się ponownie programowi z listingu 2.2, zauważysz, że w liniach 3. i 4. użyłem następujących poleceń:

```
Display "Oto wartość, którą wprowadziłeś:"  
Display age
```

Wykorzystałem dwa razy instrukcję Display, ponieważ chciałem wyświetlić dwie informacje. W linii 3. wyświetla się komunikat "Oto wartość, którą wprowadziłeś:", a w linii 4. wyświetla się wartość zmiennej age.

Większość języków programowania umożliwia jednak wyświetlenie kilku informacji za pomocą jednej instrukcji. Ponieważ jest to bardzo powszechna cecha języków programowania, ja także będę często z niej korzystać i będę wyświetlał kilka elementów za pomocą pojedynczej instrukcji Display. W tym celu wystarczy, że oddzielimy od siebie poszczególne elementy znakiem przecinka, tak jak w programie z listingu 2.3.

### **Listing 2.3**

```
1 Display "Ile masz lat?"  
2 Input age  
3 Display "Oto wartość, którą wprowadziłeś: ", age
```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

```
Ile masz lat?  
24 [Enter]  
Oto wartość, którą wprowadziłeś: 24
```

Przyjrzyj się dokładniej linii 3. w programie z listingu 2.3:

```
Display "Oto wartość, którą wprowadziłeś: ", age
          ↑
          Tutaj jest znak spacji
```

Zwróć uwagę, że literal ciągu znaków "Oto wartość, którą wprowadziłeś: " kończy się znakiem spacji. To dlatego, że chcemy, aby w wyniku za znakiem dwukropka pojawiła się spacja, tak jak tutaj:

```
Oto wartość, którą wprowadziłeś: 24
          ↑
          Tutaj jest znak spacji
```

W większości przypadków, gdy będziesz chciał wyświetlić na ekranie kilka elementów, oddzielisz je znakami spacji. Większość języków programowania nie wstawia automatycznie znaków spacji pomiędzy poszczególne elementy. Spójrz na następujący przykład w pseudokodzie:

```
Display "Styczeń", "Luty", "Marzec"
```

Po uruchomieniu takiej instrukcji większość języków wygeneruje taki wynik:

```
StyczeńLutyMarzec
```

Aby oddzielić od siebie poszczególne ciągi znaków znakiem spacji, należy instrukcję zmodyfikować do takiej postaci:

```
Display "Styczeń ", "Luty ", "Marzec"
```

## Dane wejściowe w postaci ciągu znaków

W programach z listingu 2.2 i 2.3 odczytywaliśmy liczby wprowadzone za pomocą klawiatury i zapisywaliśmy je w zmiennych za pomocą polecenia Input. W programie można także odczytywać ciągi znaków. Przykładowo w pseudokodzie z listingu 2.4 używam dwóch instrukcji Input: pierwsza odczytuje ciąg znaków, a druga — liczbę.

### **Listing 2.4**



```
1 Display "Wprowadź swoje imię."
2 Input name
3 Display "Wprowadź swój wiek."
4 Input age
5 Display "Witaj, ", name
6 Display "Masz ", age, " lat/a."
```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje imię.

**Magda [Enter]**

Wprowadź swój wiek.

**24 [Enter]**

Witaj, Magda

Masz 24 lat/a.

Instrukcja `Input` w linii 2. odczytuje dane wprowadzone za pomocą klawiatury i zapisuje je w zmiennej `name`. W przykładowym wywołaniu programu użytkownik wprowadził imię **Magda**. Instrukcja `Input` w linii 4. odczytuje dane wprowadzone za pomocą klawiatury i zapisuje je w zmiennej `age`. W przykładowym wywołaniu programu użytkownik wprowadził liczbę **24**.

## Komunikat dla użytkownika

Aby pobrać od użytkownika dane, należy najczęściej wykonać dwie operacje:

1. Wyświetlić na ekranie komunikat.
2. Odczytać wprowadzoną na klawiaturze wartość.

**Komunikat** to wiadomość, która wskazuje użytkownikowi, że musi wprowadzić pewną wartość. Przykładowo w programie z listingu 2.3 za pomocą następujących instrukcji prosimy użytkownika, aby wprowadził swój wiek:

```
Display "Wprowadź swój wiek."
Input age
```

W większości języków programowania instrukcja, która pobiera od użytkownika dane, nie wyświetla na ekranie żadnego komunikatu. Jej działanie polega po prostu na wstrzymaniu pracy programu i oczekiwaniu, aż użytkownik wprowadzi dane za pomocą klawiatury. Dlatego przed instrukcją odczytującą dane z klawiatury należy umieścić instrukcję, która wskaże użytkownikowi, o jakie informacje go prosisz. W przeciwnym razie użytkownik nie będzie wiedział, co ma zrobić. Usuńmy dla przykładu linię 1. z listingu 2.3:

```
Input age
Display "Oto wartość, którą wprowadziłeś: ", age
```

Czy widzisz, co by się stało, gdyby tak wyglądał prawdziwy program? Po uruchomieniu programu pojawiłby się pusty ekran, a instrukcja `Input` spowodowałaby, że program oczekiwalby, aż użytkownik wprowadzi na klawiaturze jakieś dane. Można by dojść do wniosku, że coś jest nie tak z tym programem.

W branży komputerowej o programach łatwych w obsłudze mówi się, że są „przyjazne dla użytkownika”. Korzystanie z programu, który nie prezentuje użytkownikowi żadnych komunikatów, jest frustrujące i z całą pewnością nie można go nazwać przyjaznym dla użytkownika. Jedną z najprostszych rzeczy, dzięki którym program będzie bardziej przyjazny, jest wyświetlanie jasnych i zrozumiałych komunikatów przed pobraniem od użytkownika danych.



**WSKAZÓWKA:** Wykładowcy często tłumaczą studentom w żartobliwy sposób, że programy należy pisać w taki sposób, aby mogli z nich korzystać „wujek Janusz” i „ciocia Grażyna”. Oczywiście nie są to prawdziwe osoby, ale potencjalni użytkownicy, którzy mogą popełnić błąd, jeśli w sposób klarowny nie podpowie się im, jak mają postępować. Projektując program, miej na uwadze, że z programu będzie korzystał ktoś, kto nie ma pojęcia o tym, jak działa on od środka.



## Punkt kontrolny

- 2.7. Wymień trzy zadania, które wykonuje zazwyczaj program komputerowy.
- 2.8. Do czego służą tabelki IPO?
- 2.9. Co to jest struktura sekwencyjna?
- 2.10. Co to jest ciąg znaków? Co to jest literał ciągu znaków?
- 2.11. Jakimi znakami jest zazwyczaj otoczony literał ciągu znaków?
- 2.12. Co to jest zmienna?
- 2.13. Wymień trzy podstawowe zasady dotyczące nazewnictwa zmiennych.
- 2.14. Którym systemem zapisu zmiennych posługuję się w tej książce?
- 2.15. Spójrz na następujące polecenie pseudokodu:  

```
Input temperature
```

Co się stanie po wykonaniu tego polecenia?
- 2.16. Kto to jest użytkownik?
- 2.17. Do czego służy komunikat dla użytkownika?
- 2.18. Wymień dwie operacje, które trzeba wykonać, aby pobrać od użytkownika dane.
- 2.19. Co oznacza termin „przyjazny dla użytkownika”?

## 2.3

# Przypisywanie wartości do zmiennych i wykonywanie obliczeń

**WYJAŚNIENIE:** Wartość w zmiennej zapisuje się za pomocą instrukcji przypisania. Wartość ta może być wynikiem jakiegoś działania z użyciem operatorów matematycznych.

## Przypisywanie wartości do zmiennych

Z poprzedniego podrozdziału wiesz, że instrukcja Input pobiera wartość wprowadzoną za pomocą klawiatury i zapisuje ją w zmiennej. Możesz także używać poleceń, które zapiszą w zmiennej określoną wartość. Oto przykład zapisany za pomocą pseudokodu:

```
Set price = 20
```

Jest to przykład instrukcji przypisania. **Instrukcja przypisania** zapisuje w zmiennej określoną wartość. W tym przypadku w zmiennej price program zapisał wartość 20. Do przypisywania wartości do zmiennej będziemy w pseudokodzie używali słowa Set, następnie nazwy zmiennej, znaku równości (=) i na końcu wartości, jaką mamy zamiar przypisać. Na listingu 2.5 przedstawiłem kolejny przykład.

**Listing 2.5**

```
1 Set dollars = 2.75
2 Display "Mam na koncie ", dollars, " dol."
```

**Wynik działania programu**

Mam na koncie 2.75 dol.

W linii 1. przypisuję do zmiennej `dollars` wartość równą 2.75. W linii 2. wyświetlam komunikat *Mam na koncie 2.75 dol.* Przyjrzymy się bliżej instrukcji w linii 2., abyś zrozumiał, jak działa polecenie `Display`. Po słowie `Display` pojawiają się trzy elementy — czyli na ekranie wyświetla się trzy informacje. Pierwszą z nich jest literal ciągu znaków "Mam na koncie ". Następnie pojawia się wartość przypisana do zmiennej `dollars`, czyli w tym przypadku 2.75. Na końcu wyświetla się kolejny literal ciągu znaków: " dol.".

Zmienne nazywane są zmiennymi dlatego, że podczas działania programu można do nich przypisywać różne wartości. Kiedy przypiszesz do zmiennej wartość, pozostanie ona przypisana do niej aż do momentu, gdy zdecydujesz się przypisać do niej inną wartość. Spójrz na pseudokod na listingu 2.6:

**Listing 2.6**

```
1 Set dollars = 2.75
2 Display "Mam na koncie ", dollars, " dol."
3 Set dollars = 99.95
4 Display "A teraz mam na koncie ", dollars, " dol.!"
```

**Wynik działania programu**

Mam na koncie 2.75 dol.
A teraz mam na koncie 99.95 dol.!

W linii 1. ustawiam wartość zmiennej `dollars` na 2.75, następnie po wykonaniu instrukcji w linii 2. wyświetli się komunikat *Mam na koncie 2.75 dol.* Polecenie w linii 3. przypisuje do zmiennej `dollars` wartość 99.95. W wyniku tego wartość 2.75 uprzednio zapisana w zmiennej zostanie zastąpiona wartością 99.95. Po wykonaniu instrukcji w linii 4. wyświetli się komunikat *A teraz mam na koncie 99.95 dol.!*. Program ten ilustruje dwie ważne cechy zmiennych:

- w zmiennej zapisana jest w danym momencie tylko jedna wartość;
- kiedy przypiszesz do zmiennej wartość, zastąpi ona wartość, która była wcześniej przypisana do tej zmiennej.



**UWAGA:** We wszystkich językach programowania obowiązuje zasada, że przyując wartość do zmiennej, nazwę zmiennej należy umieścić po lewej stronie operatora `=`. Przykładowo poniższa instrukcja jest błędna:

Set 99.95 = dollars ← **Tutaj jest błąd!**

Takie polecenie będzie traktowane jako błąd składni.

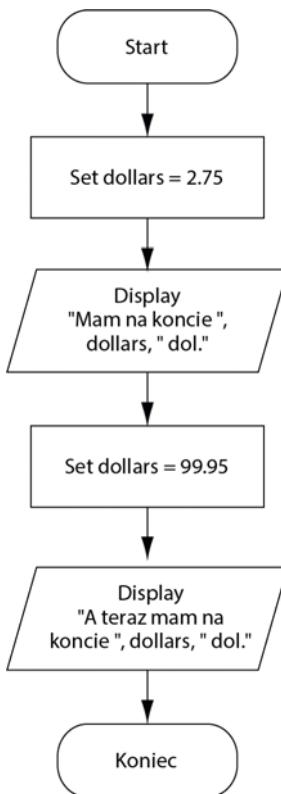
Na schematach blokowych operację przypisania wartości oznaczamy za pomocą bloku operacyjnego, czyli prostokąta. Na rysunku 2.11 zaprezentowałem schemat blokowy programu z listingu 2.6.



**UWAGA:** W tej książce do przypisywania wartości do zmiennej zdecydowałem się korzystać ze słowa Set, ponieważ w najbardziej klarowny sposób opisuje ono tę operację<sup>1</sup>. W większości języków programowania operacji przypisania wartości nie towarzyszy jednak żadne słowo — zazwyczaj wygląda to następująco:

```
dollars = 99.95
```

Jeśli Twój wykładowca się zgodzi, możesz w pseudokodzie przypisywać wartość do zmiennej z pominięciem słowa Set. Pamiętaj jednak, że nazwę zmiennej, do której chcesz przypisać wartość, musisz umieścić po lewej stronie znaku równości.



Rysunek 2.11. Schemat blokowy programu z listingu 2.6

<sup>1</sup> Słowo set oznacza w języku angielskim ustawianie czegoś — *przyp. tłum.*

## Wykonywanie obliczeń

Większość algorytmów z prawdziwego zdarzenia musi dokonywać pewnych obliczeń. Programista używa do wykonywania tych obliczeń **operatorów matematycznych**. Języki programowania udostępniają zazwyczaj operatory przedstawione w tabeli 2.1.

**Tabela 2.1.** Powszechnie używane operatory matematyczne<sup>2</sup>

Symbol	Operator	Opis
+	Dodawanie	Dodaje do siebie dwie liczby
-	Odejmowanie	Odejmuje jedną liczbę od drugiej
*	Mnożenie	Mnoży przez siebie dwie liczby
/	Dzielenie	Dzieli jedną liczbę przez drugą i zwraca iloraz
MOD	Modulo	Dzieli jedną liczbę przez drugą i zwraca resztę z dzielenia
^	Potęgowanie	Podnosi liczbę do potęgi

Programiści dzięki operatorom przedstawionym w tabeli 2.1 mogą tworzyć **wyrażenia matematyczne**. Wyrażenie matematyczne dokonuje obliczeń i zwraca wartość. Oto przykład prostego wyrażenia matematycznego:

12 + 2

Wartości po lewej i po prawej stronie operatora nazywamy **operandami** — są to wartości, które operator + doda do siebie. Wartość zwracana przez to wyrażenie jest równa 14.

W wyrażeniu matematycznym można także użyć zmiennych. Założmy, że mamy dwie zmienne o nazwach `hours` i `payRate`. Poniższe wyrażenie matematyczne mnoży za pomocą operatora \* wartość przypisaną do zmiennej `hours` przez wartość przypisaną do zmiennej `payRate`:

`hours * payRate`

Zazwyczaj, gdy dokonujemy obliczeń za pomocą wyrażeń matematycznych, wynik tych obliczeń także chcemy zapisać w pamięci, abyśmy mogli z niego skorzystać w dalszej części programu. Możemy to zrobić za pomocą instrukcji przypisania. Na listingu 2.7 przedstawiłem przykład.

W linii 1. ustawiam wartość zmiennej `price` na 100, a w linii 2. ustawiam wartość zmiennej `discount` na 20. W linii 3. do zmiennej `sale` przypisuję wartość zwracaną przez wyrażenie `price - discount`. Jak możesz zauważyć w wyniku, w zmiennej `sale` została zapisana wartość 80.

<sup>2</sup> W niektórych językach programowania operatorem modulo jest symbol %, natomiast operatorem potęgowania są znaki \*\*.

**Listing 2.7**

```

1 Set price = 100
2 Set discount = 20
3 Set sale = price - discount
4 Display "Całkowity koszt to ", sale, " zł."

```

**Wynik działania programu**

Całkowity koszt to 80 zł.

**W centrum uwagi****Obliczanie opłat  
za dodatkowe minuty**

Załóżmy, że Twój pakiet u operatora telefonicznego pozwala Ci na 700 darmowych minut rozmów miesięcznie. Kiedy przekroczyś ten limit, operator naliczy Ci dodatkową opłatę w wysokości 35 groszy na każdą przekrozoną minutę. Telefon wyświetla liczbę dodatkowych minut w danym miesiącu, ale nie informuje Cię o wysokości naliczonej opłaty dodatkowej. Do tej pory musiałeś obliczać tę wartość w tradycyjny sposób (za pomocą ołówka i kartki papieru lub kalkulatora), ale wpadłeś na pomysł, by zaprojektować program, który ułatwi te obliczenia. Chcesz więc, aby program umożliwiał wprowadzenie liczby dodatkowych minut i obliczał wysokość opłaty dodatkowej.

Na samym początku musisz się upewnić, że dokładnie zrozumiałeś, jakie operacje musi przeprowadzić taki program. Łatwiej będzie to zrozumieć, gdy bliżej przyjrzysz się temu, w jaki sposób dotychczas rozwiązywałeś to zadanie (za pomocą ołówka i kartki lub kalkulatora):

**Algorytm ręczny (korzystamy z ołówka, kartki papieru i kalkulatora)**

1. Sprawdzasz, ile dodatkowych minut wykorzystałeś w danym miesiącu.
2. Mnożysz liczbę minut przez 0,35.
3. Wynikiem mnożenia jest wartość opłaty dodatkowej.

Wypadałoby zadać sobie kilka pytań dotyczących tego algorytmu:

**Pytanie:** Jakich danych wejściowych będę potrzebował?

**Odpowiedź:** Będzie mi potrzebna liczba dodatkowych minut.

**Pytanie:** Jaką operację trzeba wykonać na danych wejściowych?

**Odpowiedź:** Trzeba pomnożyć dane wejściowe (czyli liczbę dodatkowych minut) przez 0,35. W wyniku otrzymam wysokość opłaty dodatkowej.

**Pytanie:** Czym będą dane wyjściowe?

**Odpowiedź:** Wartością opłaty dodatkowej.

Kiedy już zdefiniujesz, czym są dane wejściowe, jak wygląda przetwarzanie i jak wyglądają dane wyjściowe, możesz przystąpić do stworzenia ogólnego szkicu algorytmu komputerowego.

### Algorytm komputerowy

1. Pobierz od użytkownika liczbę dodatkowych minut.
2. Oblicz opłatę dodatkową, mnożąc liczbę dodatkowych minut przez 0,35.
3. Wyświetl na ekranie opłatę dodatkową.

W pierwszym kroku algorytmu komputerowego program pobiera od użytkownika liczbę dodatkowych minut. Za każdym razem, gdy chcemy, aby użytkownik wprowadził jakąś wartość, program musi wykonać dwie rzeczy: (1) wyświetlić komunikat z prośbą o wprowadzenie danych oraz (2) odczytać wprowadzone na klawiaturze dane i zapisać je w zmiennej. Pierwszy krok algorytmu będzie więc wyglądał w pseudokodzie tak:

```
Display "Wprowadź liczbę dodatkowych minut."
Input excessMinutes
```

Zwróci uwagę, że polecenie Input zapisuje wprowadzoną przez użytkownika liczbę w zmiennej o nazwie excessMinutes.

W drugim kroku algorytmu program oblicza opłatę dodatkową, mnożąc liczbę dodatkowych minut przez 0,35. Poniższa linia pseudokodu wykonuje te obliczenia i zapisuje wynik w zmiennej overageFee:

```
Set overageFee = excessMinutes * 0.35
```

W trzecim kroku algorytmu program wyświetla opłatę dodatkową. Ponieważ wartość ta jest zapisana w zmiennej overageFee, program wyświetla odpowiedni komunikat i wartość zmiennej overageFee. Polecenie będzie wyglądało następująco:

```
Display "Opłata dodatkowa wynosi obecnie ", overageFee, " zł."
```

Na listingu 2.8 zaprezentowałem cały program w pseudokodzie i przykładowe wywołanie programu. Na rysunku 2.12 widoczny jest schemat blokowy tego programu.

### **Listing 2.8.**

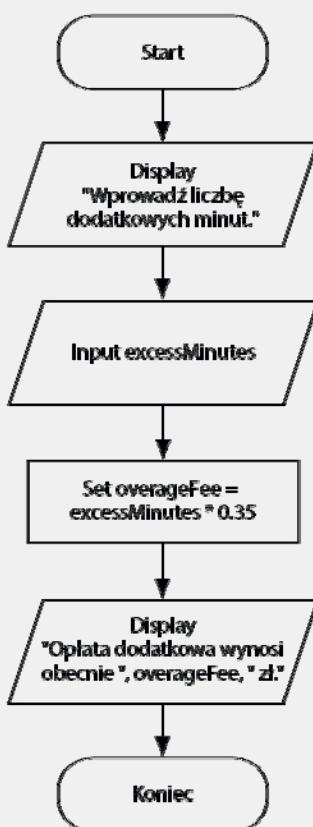
```
1 Display "Wprowadź liczbę dodatkowych minut."
2 Input excessMinutes
3 Set overageFee = excessMinutes * 0.35
4 Display "Opłata dodatkowa wynosi obecnie ", overageFee, " zł."
```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę dodatkowych minut.

**100 [Enter]**

Opłata dodatkowa wynosi obecnie 35 zł.



**Rysunek 2.12.** Schemat blokowy programu z listingu 2.8

## W centrum uwagi

### Obliczanie procentów

Obliczanie procentów jest bardzo często wykonywaną operacją podczas programowania komputerów. W matematyce do oznaczania procentów wykorzystuje się symbol %, jednak w większości języków programowania tak nie jest. Pisząc program, musisz zamienić liczbę procentów na wartość ułamkową. Przykładowo: 50% można zapisać jako 0.5, a 2% jako 0.02.

Przyjrzymy się temu, w jaki sposób stworzylibyśmy program do obliczania procentu określonej wartości. Założmy, że mamy do czynienia z akcją promocyjną w sklepie, gdzie na wszystkie produkty przysługuje rabat w wysokości 20%. Poproszono nas o napisanie programu, który będzie obliczał cenę danego produktu po zastosowaniu rabatu promocyjnego. Oto algorytm takiego programu:



1. Pobierz cenę detaliczną produktu.
2. Oblicz wartość równą 20% ceny detalicznej — będzie to wartość rabatu.
3. Odejmij wartość rabatu od ceny detalicznej — wartość ta będzie równa cenie sprzedaży.
4. Wyświetl cenę sprzedaży.

W kroku 1. pobieramy cenę detaliczną artykułu — prosimy użytkownika o wprowadzenie tej ceny. Przypominam, że zadanie to składa się z dwóch operacji: (1) wyświetlenia komunikatu, w którym prosimy użytkownika o wprowadzenie danych, i (2) odczytania danych z klawiatury. Wykorzystamy do tego dwie poniższe instrukcje. Zwróć uwagę, że wartość wprowadzona przez użytkownika zostanie zapisana w zmiennej `originalPrice`.

```
Display "Wprowadź cenę detaliczną artykułu."  
Input originalPrice
```

W kroku 2. obliczamy wartość rabatu. Aby to zrobić, musimy obliczyć wartość równą 20% ceny detalicznej. Poniższe polecenie wykonuje te obliczenia i zapisuje wynik w zmiennej `discount`:

```
Set discount = originalPrice * 0.2
```

W kroku 3. odejmujemy wartość rabatu od ceny detalicznej. Poniższe polecenie wykonuje te obliczenia i zapisuje wynik w zmiennej `salePrice`:

```
Set salePrice = originalPrice - discount
```

Na samym końcu, w kroku 4., wyświetlimy cenę sprzedaży:

```
Display "Cena sprzedaży wynosi ", salePrice, " zł."
```

Na listingu 2.9 znajduje się pełny pseudokod i przykładowe wywołanie programu. Na rysunku 2.13 przedstawiłem schemat blokowy tego programu.

### **Listing 2.9**



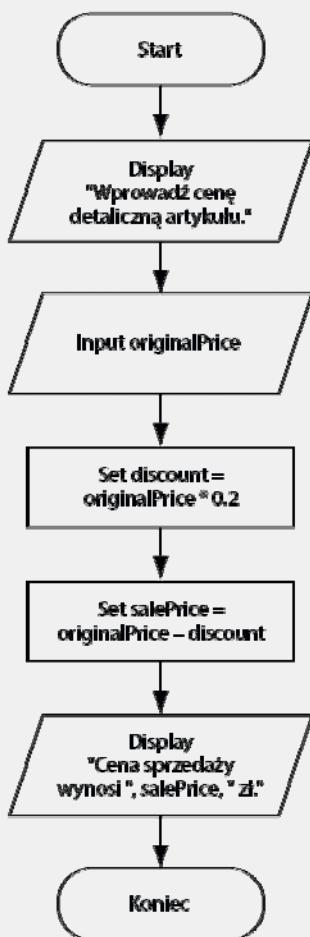
```
1 Display "Wprowadź cenę detaliczną artykułu."  
2 Input originalPrice  
3 Set discount = originalPrice * 0.2  
4 Set salePrice = originalPrice - discount  
5 Display "Cena sprzedaży wynosi ", salePrice, " zł."
```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę detaliczną artykułu.

**100 [Enter]**

Cena sprzedaży wynosi 80 zł.



**Rysunek 2.13.** Schemat blokowy programu z listingu 2.9

## Kolejność działań

Wyrażenie matematyczne może się składać z kilku działań. Poniższe polecenie oblicza sumę następujących składników: liczby 17, wartości zmiennej  $x$ , liczby 21 i wartości zmiennej  $y$ , po czym zapisuje ją w zmiennej `answer`:

```
Set answer = 17 + x + 21 + y
```

Jednak niektóre obliczenia nie będą takie oczywiste. Przyjrzyjmy się następującemu przykładowi:

```
Set outcome = 12 + 6 / 3
```

Jaka wartość znajdzie się w zmiennej `outcome`? Liczba 6 jest operandem zarówno dodawania, jak i dzielenia. W zależności od tego, kiedy wykonamy operację dzielenia,

## 78 Rozdział 2. Dane wejściowe, przetwarzanie i dane wyjściowe

w zmiennej `outcome` znajdzie się liczba 6 lub 14. Odpowiedź prawidłowa brzmi 14, ponieważ zasada kolejności wykonywania działań wskazuje, że operacja dzielenia ma pierwszeństwo względem operacji dodawania.

W większości języków programowania kolejność wykonywania działań można podsumować następująco:

1. Wykonaj wszystkie operacje umieszczone w nawiasach.
2. Wykonaj wszystkie operacje podnoszenia do potęgi.
3. Wykonaj wszystkie operacje mnożenia, dzielenia i modulo, od lewej do prawej strony.
4. Wykonaj wszystkie operacje dodawania i odejmowania, od lewej do prawej strony.

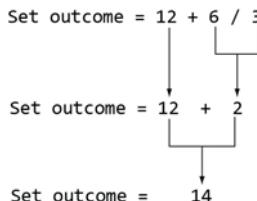
Wyrażenia matematyczne oblicza się od lewej do prawej strony. Jeżeli dwie różne operacje mają wspólny operand, kolejność obliczeń wskazuje zasada kolejności wykonywania działań. Mnożenie i dzielenie wykonuje się przed dodawaniem i odejmowaniem, więc polecenie:

```
Set outcome = 12 + 6 / 3
```

zostanie wykonane następująco:

- 1) 6 podziel przez 3, w wyniku otrzymasz 2;
- 2) 12 dodaj do 2, w wyniku otrzymasz 14

Można to przedstawić za pomocą diagramu jak na rysunku 2.14.



**Rysunek 2.14.** Zasada kolejności wykonywania działań

W tabeli 2.2 przedstawiłem kilka przykładowych wyrażeń i ich wartości.

**Tabela 2.2.** Kilka przykładowych wyrażeń i ich wartości

Wyrażenie	Wartość
$5 + 2 * 4$	13
$10 / 2 - 3$	2
$8 + 12 * 2 - 4$	28
$6 - 3 * 2 + 7 - 1$	6

## Grupowanie za pomocą nawiasów

Jeśli chcemy wymusić na programie, aby pewne działania wykonał przed innymi, możemy fragment wyrażenia matematycznego ująć w nawiasy. W poniższej instrukcji najpierw dodane zostaną do siebie zmienne  $a$  i  $b$ , a następnie ich suma zostanie podzielona przez 4:

```
Set result = (a + b) / 4
```

Gdybyśmy nie użyły nawiasów, zmienność  $b$  zostałaby podzielona przez 4, a iloraz zostałby dodany do zmiennej  $a$ .

W tabeli 2.3 przedstawiłem kilka innych wyrażeń i ich wartości.

**Tabela 2.3.** Kilka innych wyrażeń i ich wartości

Wyrażenie	Wartość
$(5 + 2) * 4$	28
$10 / (5 - 3)$	5
$8 + 12 * (6 - 2)$	56
$(6 - 3) * (2 + 7) / 3$	9



**UWAGA:** Dzięki nawiasom możesz poprawić przejrzystość operacji matematycznych, nawet wtedy, gdy ich użycie nie jest konieczne. Spójrz na przykład na to polecenie:

```
Set fahrenheit = celsius * 1.8 + 32
```

Mimo że nie jest to konieczne, możemy użyć nawiasów, aby wyraźnie zaznaczyć, iż operacja mnożenia następuje przed operacją dodawania:

```
Set fahrenheit = (celsius * 1.8) + 32
```

## W centrum uwagi

### Obliczanie średniej



Obliczanie średniej kilku liczb jest bardzo proste: wystarczy dodać do siebie wszystkie liczby i podzielić sumę przez liczbę tych liczb. Choć jest to proste zadanie, podczas pisania programu można łatwo popełnić błąd. Założymy, że chcemy obliczyć średnią trzech liczb zapisanych w zmiennej  $a$ ,  $b$  i  $c$ . Gdybyśmy postąpili pochopnie, moglibyśmy wykonać w programie obliczenia za pomocą następującej instrukcji:

```
Set average = a + b + c / 3
```

Czy widzisz tutaj błąd? Podczas wykonywania tej instrukcji najpierw zostanie wykonana operacja dzielenia. Wartość przypisana do zmiennej  $c$  zostanie podzielona przez 3,

a następnie iloraz zostanie dodany do sumy  $a + b$ . W ten sposób nie obliczymy prawidłowo średniej. Aby poprawić ten błąd, musimy ująć  $a + b + c$  w nawiasy, tak jak tutaj:

```
Set average = (a + b + c) / 3
```

Przyjrzyjmy się teraz etapom, z których będzie się składał program do obliczania średniej. Założymy, że na zajęciach z informatyki dostałeś oceny z trzech sprawdzianów i chcesz obliczyć średnią wyników z tych sprawdzianów. Oto jak będzie wyglądał algorytm:

1. Pobierz wynik z pierwszego sprawdzianu.
2. Pobierz wynik z drugiego sprawdzianu.
3. Pobierz wynik z trzeciego sprawdzianu.
4. Oblicz średnią, dodając do siebie wyniki z trzech sprawdzianów i dzieląc otrzymaną sumę przez 3.
5. Wyświetl średnią na ekranie.

W krokach 1., 2. i 3. poprosimy użytkownika o wprowadzenie wyników trzech sprawdzianów. Wyniki zapiszemy w zmiennych `test1`, `test2` i `test3`. W kroku 4. obliczymy średnią trzech wyników — posłużymy się w tym celu następującym poleceniem:

```
Set average = (test1 + test2 + test3) / 3
```

W ostatnim kroku wyświetlimy wynik na ekranie. Na listingu 2.10 przedstawiłem pseudokod tego programu, a na rysunku 2.15 jego schemat blokowy.

### **Listing 2.10.**

```
1 Display "Wprowadź wynik z pierwszego sprawdzianu."
2 Input test1
3 Display "Wprowadź wynik z drugiego sprawdzianu."
4 Input test2
5 Display "Wprowadź wynik z trzeciego sprawdzianu."
6 Input test3
7 Set average = (test1 + test2 + test3) / 3
8 Display "Twój średni wynik wynosi ", average
```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik z pierwszego sprawdzianu.

**90 [Enter]**

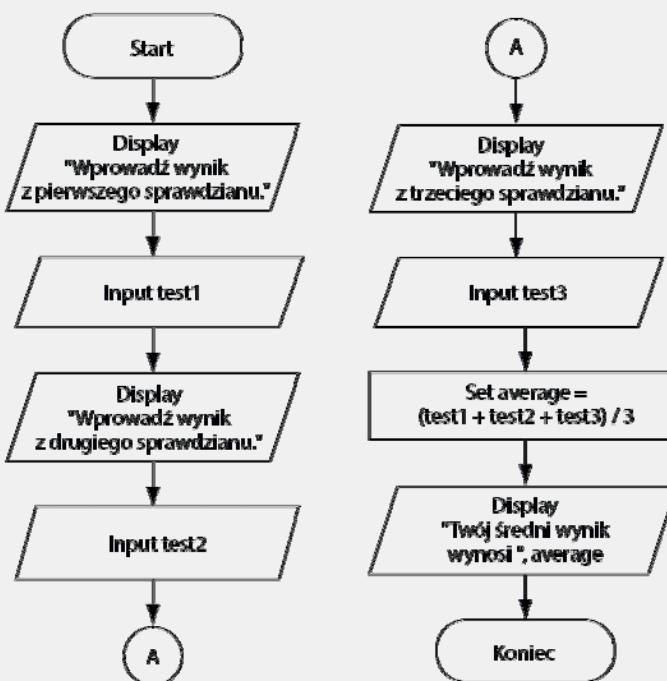
Wprowadź wynik z drugiego sprawdzianu.

**80 [Enter]**

Wprowadź wynik z trzeciego sprawdzianu.

**100 [Enter]**

Twój średni wynik wynosi 90



**Rysunek 2.15.** Schemat blokowy programu z listingu 2.10

## Zaawansowane operatory matematyczne: potęgowanie i modulo

Wiele języków programowania, poza prostymi operatorami matematycznymi takimi jak dodawanie, odejmowanie, mnożenie i dzielenie, udostępnia także operatory, których przeznaczeniem jest podnoszenie do potęgi i obliczanie reszty z dzielenia. Jako operator potęgowania służy często symbol  $\wedge$ , a jego działanie polega na podniesieniu określonej liczby do wskazanej potęgi. Przykładowo poniższe polecenie podnosi wartość przypisaną do zmiennej `length` do drugiej potęgi i zapisuje wynik w zmiennej `area`:

```
Set area = length2
```

Do obliczania reszty z dzielenia (modulo) służy często słowo MOD (w pewnych językach jest to operator `%`). Operator modulo dokonuje operacji dzielenia, ale zamiast ilorazu zwraca resztę z dzielenia. Poniższa instrukcja przypisuje do zmiennej `leftover` wartość 2:

```
Set leftover = 17 MOD 3
```

Do zmiennej `leftover` zostanie przypisana wartość 2, ponieważ w wyniku dzielenia liczby 17 przez 3 otrzymamy resztę równą 2. Prawdopodobnie nie będziesz korzystać z tego operatora zbyt często, ale w pewnych sytuacjach jest on bardzo pomocny. Często używa się go, aby na przykład sprawdzić, czy dana liczba jest parzysta czy nieparzysta, określić dzień tygodnia, zmierzyć upływ czasu, a także w wielu innych przypadkach.

## Zamienianie wzorów matematycznych na polecenia programu

Zapewne pamiętasz z zajęć z algebry, że wyrażenie  $2xy$  należy rozumieć jako 2 razy  $x$  razy  $y$ . Na zajęciach z matematyki nie zawsze zapisuje się symbol mnożenia, jednak wszystkie języki programowania wymagają umieszczenia operatora przy każdej operacji matematycznej. W tabeli 2.4 przedstawiłem kilka wyrażeń algebraicznych, w których występuje mnożenie, i zamieściłem odpowiadające im wyrażenia w programie komputerowym.

**Tabela 2.4.** Wyrażenie algebraiczne

Wyrażenie algebraiczne	Wykonywane działanie	Wyrażenie w programie
$6B$	6 razy B	$6 * B$
$(3)(12)$	3 razy 12	$3 * 12$
$4xy$	4 razy $x$ razy $y$	$4 * x * y$

W przypadku pewnych wyrażeń algebraicznych podczas zamiany na wyrażenie w programie trzeba umieścić nawiasy, które nie występują w wyrażeniu algebraicznym. Spójrz na następujący wzór:

$$x = \frac{a + b}{c}$$

Aby zamienić ten wzór na wyrażenie w programie, będziemy musieli umieścić  $a + b$  w nawiasach:

Set  $x = (a + b) / c$

W tabeli 2.5 przedstawiłem kilka kolejnych wyrażeń algebraicznych i odpowiadający im pseudokod.

**Tabela 2.5.** Wyrażenie algebraiczne i polecenie w pseudokodzie

Wyrażenie algebraiczne	Polecenie w pseudokodzie
$y = 3\frac{x}{2}$	Set $y = x / 2 * 3$
$z = 3bc + 4$	Set $z = 3 * b * c + 4$
$a = \frac{x + 2}{a - 1}$	Set $a = (x + 2) / (a - 1)$



## W centrum uwagi

### Zamiana wzoru matematycznego na wyrażenie

Załóżmy, że zamierzasz przelać na rachunek oszczędnościowy pewną kwotę pieniężny i pozostawić ją tam przez okres 10 lat. Chcesz, aby po 10. roku na rachunku tym była zgromadzona kwota 10000 złotych. Jaką kwotę musisz w takim przypadku przelać na konto? Możesz posłużyć się następującym wzorem:

$$P = \frac{F}{(1 + r)^n}$$

Gdzie:

- $P$  oznacza kwotę, jaką musisz przelać na rachunek;
- $F$  oznacza kwotę, jaką w przyszłości chcesz uzbiereć na rachunku (w naszym przypadku  $F$  będzie równe 10000);
- $r$  oznacza roczne oprocentowanie rachunku;
- $n$  oznacza liczbę lat, jaką pieniądze mają być ulokowane na rachunku.

Fajnie byłoby stworzyć program komputerowy, który dokona stosowych obliczeń — dzięki temu będziemy mogli eksperymentować z różnymi kwotami. Oto algorytm, z którego skorzystamy:

1. Pobierz kwotę końcową, którą chcesz uzyskać na rachunku.
2. Pobierz roczne oprocentowanie rachunku.
3. Pobierz liczbę lat, jaką pieniądze mają być ulokowane na rachunku.
4. Oblicz kwotę, którą trzeba przelać na rachunek.
5. Wyświetl na ekranie wynik otrzymany w kroku 4.

W krokach 1., 2. i 3. poprosimy użytkownika o wprowadzenie odpowiednich wartości. Wartość kwoty końcowej zapiszemy w zmiennej `futureValue`, roczne oprocentowanie w zmiennej `rate`, a liczbę lat w zmiennej `years`.

W kroku 4. obliczamy kwotę, którą trzeba będzie przelać na rachunek. W tym celu zamieniamy przedstawiony wcześniej wzór na poniższe wyrażenie w pseudokodzie. Wynik obliczeń zapiszemy w zmiennej `presentValue`.

```
Set presentValue = futureValue / (1 + rate)^years
```

W kroku 5. wyświetlimy na ekranie wartość przypisaną do zmiennej `presentValue`. Na listingu 2.11 przedstawiłem cały program w pseudokodzie, a na rysunku 2.16 jego schemat blokowy.

#### **Listing 2.11.**

- 1 Display "Wprowadź kwotę, jaką chcesz uzyskać na rachunku."
- 2 Input `futureValue`
- 3 Display "Wprowadź stopę oprocentowania rachunku."
- 4 Input `rate`
- 5 Display "Przez ile lat chcesz trzymać pieniądze na rachunku?"

```

6 Input years
7 Set presentValue = futureValue / (1 + rate)^years
8 Display "Musisz przelać na rachunek ", presentValue, " zł."

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź kwotę, jaką chcesz uzyskać na rachunku.

**10000 [Enter]**

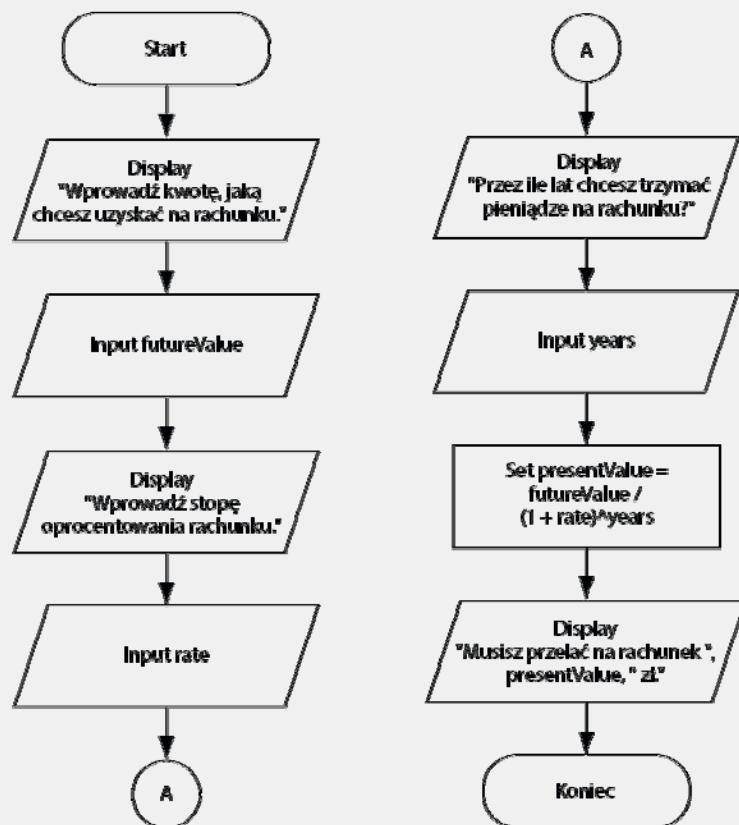
Wprowadź stopę oprocentowania rachunku.

**0.05 [Enter]**

Przez ile lat chcesz trzymać pieniądze na rachunku?

**10 [Enter]**

Musisz przelać na rachunek 6139 zł.



Rysunek 2.16. Schemat blokowy programu z listingu 2.11



## Punkt kontrolny

- 2.20. Do czego służy polecenie przypisania?
- 2.21. Co się stanie z wartością przypisaną do tej pory do zmiennej, gdy przypiszesz do niej inną wartość?

- 2.22. Wymień kolejność, w jakiej wykonywane są działania w większości języków programowania.
- 2.23. Do czego służy operator potęgowania?
- 2.24. Do czego służy operator modulo?

## 2.4

# Deklarowanie zmiennych i typy danych

**WYJAŚNIENIE:** Wiele języków programowania wymaga zadeklarowania zmiennej w programie przed jej użyciem. Podczas deklarowania zmiennej można ją także (ale nie trzeba) zainicjalizować jakąś wartością. Częstym źródłem błędów w programie jest właśnie używanie niezainicjalizowanych zmiennych.

Wiele języków programowania wymaga deklarowania wszystkich zmiennych, które zamierzamy wykorzystać w programie. Deklaracja zmiennej to instrukcja, która wskazuje zazwyczaj dwie rzeczy:

- nazwę zmiennej;
- typ danych zmiennej.

Typ danych (ang. *data type*) to po prostu rodzaj wartości, jakie dana zmienna ma przechowywać. Kiedy już zadeklarujesz zmienną, będzie ona mogła przechowywać tylko wartości wskazanego typu. Próba zapisania w zmiennej wartości innego typu niż zadeklarowana spowoduje w większości języków programowania wystąpienie błędu.

Typy danych, z jakich możesz korzystać, zależą od konkretnego języka programowania. Przykładowo Java udostępnia cztery typy danych odpowiadające liczbom całkowitym, dwa typy danych odpowiadające liczbom rzeczywistym, jeden typ danych dla ciągów znaków, a także wiele innych typów.

Dotychczas w naszych programach zapisanych za pomocą pseudokodu nie deklarowaliśmy zmiennych — po prostu używaliśmy zmiennych bez ich deklarowania. Takie rozwiązanie jest dopuszczalne, gdy program jest niewielki, jednak kiedy programy staną się znacznie dłuższe i bardziej skomplikowane, deklarowanie zmiennych okaże się jak najbardziej sensowne. Jeśli będziesz deklarować zmienne w programach zapisanych pseudokodem, ich późniejsze tłumaczenie na język docelowy będzie znacznie łatwiejsze.

W większości programów zawartych w tej książce będziemy korzystać z trzech typów danych: `Integer`, `Real` i `String`. Oto ich opisy:

- W zmiennej typu `Integer` można zapisywać liczby całkowite. Przykładowo można w niej zapisać wartości takie jak `42`, `0` lub `-99`. W zmiennej typu `Integer` nie można zapisać liczby z częścią ułamkową, np. `22,1` czy `-4,9`.
- W zmiennej typu `Real` można zapisywać zarówno liczby całkowite, jak i liczby rzeczywiste. Przykładowo można w niej zapisać wartości takie jak `3,5`, `-87,96` lub `3,0`.

- W zmiennej typu `String` można zapisać jakikolwiek ciąg znaków, np. imię i nazwisko, adres, hasło itp.

W tej książce deklarację zmiennej będziemy rozpoczynali od słowa `Declare`, a następnie wskażemy typ danych i nazwę zmiennej. Oto przykład:

```
Declare Integer length
```

W tym poleceniu zadeklarowaliśmy zmienną typu `Integer` o nazwie `length`. Oto inny przykład:

```
Declare Real grossPay
```

W tym poleceniu zadeklarowaliśmy zmienną typu `Real` o nazwie `grossPay`. I jeszcze jeden przykład:

```
Declare String name
```

W tym poleceniu zadeklarowaliśmy zmienną typu `String` o nazwie `name`.

Jeżeli chcemy zadeklarować więcej niż jedną zmienną tego samego typu, możemy to zrobić za pomocą jednego polecenia. Założymy, że chcemy zadeklarować trzy zmienne typu `Integer` o nazwach `length`, `width` i `height`. Możemy to zrobić za pomocą jednego polecenia:

```
Declare Integer length, width, height
```



**UWAGA:** W wielu językach programowania poza typem `String` dostępny jest także typ znakowy `Character`. Różnica między nimi polega na tym, że w zmiennej typu `String` można zapisać sekwencję znaków dowolnej długości, natomiast w zmiennej typu `Character` można zapisać tylko jeden znak. W tej książce postawiłem jednak na prostotę i do zapisywania jakichkolwiek danych znakowych będę używał tylko zmiennych typu `String`.

## Deklarowanie zmiennych przed ich użyciem

Celem deklarowania zmiennej jest poinformowanie kompilatora lub interpretera, że w dalszej części programu będziesz z niej korzystać. Zazwyczaj instrukcja deklarująca zmienną spowoduje utworzenie jej w pamięci komputera. Z tego powodu musisz zadeklarować zmienną, *zanim* wykonasz jakiekolwiek polecenie, które będzie się do niej odwoływało. Jest to jak najbardziej sensowne: przecież nie da się zapisać wartości w zmiennej, która jeszcze nie istnieje w pamięci komputera.

Przyjrzymy się przykładowemu pseudokodowi. Gdybyśmy przetłumaczyli ten kod na prawdziwy język programowania, taki jak Java czy C++, pojawiłby się błąd, ponieważ w instrukcji `Input` odwołujemy się do zmiennej `age`, której jeszcze nie zadeklarowaliśmy.

<pre>Display "Ile masz lat?" Input age Declare Integer age</pre>	} <b>W tym pseudokodzie występuje błąd!</b>
--	---

Na listingu 2.12 przedstawiłem prawidłowy sposób deklarowania zmiennej. Zwróć uwagę, że polecenie, w którym deklaruję zmienną, występuje przed innym poleceniem odwołującym się do zmiennej age.

### **Listing 2.12.**

```
1 Declare Integer age
2 Display "Ile masz lat?"
3 Input age
4 Display "Oto wartość, którą wprowadziłeś:"
5 Display age
```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

```
Ile masz lat?
24 [Enter]
Oto wartość, którą wprowadziłeś:
24
```

Na listingu 2.13 przedstawiłem kolejny przykład. Deklaruję w nim cztery zmienne: trzy z nich służą do przechowywania wyników, a w czwartej zapisywana jest średnia wartość wyników.

### **Listing 2.13.**

```
1 Declare Real test1
2 Declare Real test2
3 Declare Real test3
4 Declare Real average
5
6 Set test1 = 88.0
7 Set test2 = 92.5
8 Set test3 = 97.0
9 Set average = (test1 + test2 + test3) / 3
10 Display "Twój średni wynik wynosi ", average
```

#### **Wynik działania programu**

```
Twój średni wynik wynosi 92.5
```

W programie tym posłużyłem się często spotykanym sposobem deklarowania zmiennych: wszystkie zmienne zadeklarowałem na samym początku programu, przed jakąkolwiek inną instrukcją. To jeden ze sposobów, dzięki którym można mieć pewność, że zmienne zostaną zadeklarowane przed ich użyciem.

Zwróć uwagę, że linia 5. jest pusta. Nie ma to żadnego wpływu na działanie programu, ponieważ większość kompilatorów i interpreterów ignoruje puste linie w kodzie. Jednak dzięki temu oddzieliłem wizualnie sekcję zawierającą deklarację zmiennych od reszty programu. Sprawia to, że program jest bardziej czytelny i łatwiej go zrozumieć.

Programiści bardzo często korzystają z pustych linii i wcięć w kodzie, aby uporządkować wizualnie program. Przypomina to sposób, w jaki pisarze układają tekst na stronach książek: zamiast pisać całą książkę za pomocą szeregu następujących po sobie zdań, dzielą tekst na poszczególne akapity. Nie zmienia to treści samej książki, ale poprawia jej czytelność.

Mimo że możesz wstawiać puste linie i wcięcia w dowolnym miejscu programu, nie rób tego w sposób chaotyczny. Programiści często kierują się w takim przypadku pewną konwencją. Przed chwilą dowiedziałeś się, że jedną z tych konwencji jest oddzielanie sekcji zawierającej deklarację zmiennych od reszty programu. Takie konwencje wpływają na **styl programowania**. W miarę jak będziesz się zagłębiać w tę książkę, poznasz jeszcze kilka innych konwencji programistycznych.

## Iinicjalizowanie zmiennych

Podczas deklarowania zmiennej możesz także przypisać do niej wartość. Nazywamy to **inicjalizacją** zmiennej. W poniższym przykładzie deklaruję zmienną o nazwie `price` i przypisuję do niej wartość równą 49.95:

```
Declare Real price = 49.95
```

Możemy powiedzieć, że zainicjalizowaliśmy zmienną `price` wartością 49.95. Oto inny przykład:

```
Declare Integer length = 2, width = 4, height = 8
```

W poleceniu tym deklaruję trzy zmienne. Zmienna `length` została zainicjalizowana wartością 2, zmienna `width` — wartością 4, a zmienna `height` — wartością 8.

## Zmienne niezainicjalizowane

**Zmienna niezainicjalizowana** to taka zmienna, która została zadeklarowana, ale nie została do niej przypisana żadna wartość. Niezainicjalizowane zmienne są źródłem wielu błędów logicznych w programie. Spójrzmy na następujący przykład:

```
Declare Real dollars
Display "Mam na rachunku ", dollars, " dol."
```

W powyższym programie zadeklarowaliśmy zmienną `dollars`, ale jej nie zainicjalizowaliśmy. Mimo to użyliśmy jej w poleceniu `Display`.

Zapewne interesuje Cię, co w takim przypadku wyświetli program. Należy tutaj uczciwie odpowiedzieć: nie wiadomo. A to dlatego, że każdy język programowania może inaczej postępować z niezainicjalizowanymi zmiennymi. Niektóre przypisują do niezainicjalizowanych zmiennych domyślną wartość równą 0. W innych może to być inna, zupełnie nieprzewidywalna wartość, ponieważ rezerwują one miejsce w pamięci komputera na daną zmienną, ale nie ustawiają w komórkach pamięci żadnej wartości. W rezultacie do niezainicjalizowanej zmiennej będzie przypisana taka wartość, jaka była wcześniej zapisana w określonych komórkach pamięci. Programiści często nazywają takie nieprzewidywalne wartości danymi „śmieciowymi”.

Niezainicjalizowane zmienne mogą być przyczyną trudnych do wychwycenia błędów logicznych w programie — szczególnie gdy takiej zmiennej będziemy używali do jakichś obliczeń. Spójrz na przykładowy pseudokod — jest on zmodyfikowaną wersją programu z listingu 2.13. Czy widzisz błąd?

```

1 Declare Real test1
2 Declare Real test2
3 Declare Real test3
4 Declare Real average
5
6 Set test1 = 88.0
7 Set test2 = 92.5
8 Set average = (test1 + test2 + test3) / 3
9 Display "Twój średni wynik wynosi ", average

```

**W tym pseudokodzie występuje błąd!**

Program ten nie zadziała prawidłowo, ponieważ zmienna `test3` nie została zainicjalizowana żadną wartością. Podczas wykonywania obliczeń w linii 8. zmienna `test3` może zawierać śmieciowe dane, co oznacza, że wynikiem obliczeń może być nieprzewidywalna wartość, która następnie zostanie zapisana w zmiennej `average`. Początkujący programista może mieć problem z wychwyceniem tego błędu, gdyż w pierwszej kolejności założy, że coś jest nie tak z działaniem matematycznym w linii 8.

W jednym z kolejnych podrozdziałów omówię technikę debugowania, która ułatwi Ci wychwytywanie takich błędów jak ten w zmodyfikowanej wersji programu z listingu 2.13. Musisz jednak kierować się zasadą, aby przed użyciem danej zmiennej zainicjalizować ją w momencie deklaracji lub przypisać do niej poprawną wartość za pomocą instrukcji przypisania lub polecenia `Input`.

## Literały liczbowe i kompatybilność typów danych

W wielu programach, które stworzyliśmy do tej pory, wartości liczbowe zapisywaliśmy bezpośrednio w pseudokodzie. Przykładowo w programie z listingu 2.6 zapisaliśmy liczbę 2,75:

```
Set dollars = 2.75
```

A w tym poleceniu z listingu 2.7 zapisaliśmy liczbę 100 w następujący sposób:

```
Set price = 100
```

Liczbę, która występuje w kodzie programu, nazywamy **literałem liczbowym** (ang. *numeric literal*). W przypadku większości języków programowania literały liczbowe, które zapiszemy z separatorem dziesiętnym, na przykład 2,75, zostaną zapisane w pamięci jako liczby rzeczywiste typu `Real` i jako takie będą też traktowane, gdy uruchomimy program. Jeżeli zapiszemy liczbę bez separatora dziesiętnego, na przykład 100, taki literał liczbowy zostanie umieszczony w pamięci jako liczba całkowita `Integer`.

Ważne, aby o tym pamiętać podczas pisania programu i przypisywania wartości lub inicjalizowania zmiennych. Wiele języków zgłosi po prostu błąd, gdy spróbujesz przypisać do zmiennej określonego typu wartość wskazującą na inny typ danych. Spójrz na następujący przykład:

```
Declare Integer i
Set i = 3.7 ← Tutaj jest błąd!
```

Taka operacja przypisania wartości spowoduje wystąpienie błędu, ponieważ próbujemy przypisać do zmiennej typu całkowitego, wartość w postaci liczby rzeczywistej 3,7. Poniższy kod także jest błędny:

```
Declare Integer i
Set i = 3.0 ← Tutaj jest błąd!
```

Mimo tego, że literal liczbowy 3,0 nie ma części ułamkowej (z matematycznego punktu widzenia jest to taka sama liczba jak 3), przez komputer jest ona traktowana jako liczba rzeczywista, ponieważ została zapisana z separatorem dziesiętnym.



**UWAGA:** W większości języków programowania przypisanie liczby rzeczywistej do zmiennej typu całkowitego jest niedopuszczalne, ponieważ zmienna typu Integer nie zapisuje w pamięci części ułamkowej. Niemniej można w takich językach do zmiennej typu Real przypisać wartość w postaci liczby całkowitej i nie spowoduje to wystąpienia błędu. Oto przykład:

```
Declare Real r
Set r = 77
```

Mimo że 77 jest literałem liczbowym reprezentującym liczbę całkowitą Integer, można ją bez żadnej utraty informacji zapisać w zmiennej typu Real.

## Dzielenie całkowite

Zwróc szczególną uwagę podczas dzielenia jednej liczby całkowitej przez inną liczbę całkowitą. W przypadku wielu języków programowania wynikiem takiej operacji będzie także liczba całkowita. Takie zachowanie nazywamy **dzieleniem całkowitym**. Oto przykład takiej operacji w pseudokodzie:

```
Set number = 3 / 2
```

W poleceniu dzielimy liczbę 3 przez liczbę 2, a wynik zapisujemy w zmiennej number. Ale co tak naprawdę znajdzie się w zmiennej number? Spodziewasz się zapewne wartości 1,5, bo kalkulator pokaże taką wartość, gdy podzielisz 3 przez 2. Jednak w przypadku wielu języków programowania stanie się coś innego. Ponieważ liczby 3 i 2 są traktowane jako liczby całkowite, w wyniku operacji dzielenia może zostać pominięta część ułamkowa. Takie odrzucenie części ułamkowej liczby rzeczywistej nazywamy **przyjęciem**. Właśnie dlatego w zmiennej number zostanie zapisana wartość 1, a nie 1,5.

Jeśli będziesz pisać program w którymś z języków zachowujących się w ten sposób, pamiętaj, że aby otrzymać wynik dzielenia w postaci liczby rzeczywistej, co najmniej jeden z operandów także musi być liczbą rzeczywistą.



**UWAGA:** W językach Java, C++ i C operator / odrzuca część ułamkową, gdy oba operandy są liczbami całkowitymi. W językach tych wynik wyrażenia 3/2 będzie równy 1. W przypadku języka Visual Basic operator / nie odrzuca części ułamkowej, a więc wynik wyrażenia 3/2 będzie wynosił 1.5.



## Punkt kontrolny

- 2.25. Jakie dwa elementy musisz zazwyczaj wskazać, deklarując zmienną?
- 2.26. Czy ma znaczenie, w którym miejscu programu umieścisz deklarację zmiennej?
- 2.27. Co to jest inicjalizacja zmiennej?
- 2.28. Czy niezainicjalizowane zmienne stwarzają jakieś zagrożenie w programie?
- 2.29. Co to jest niezainicjalizowana zmienna?

## 2.5

## Stałe nazwane

**WYJAŚNIENIE:** Stała nazwana to nazwa reprezentująca pewną wartość, której nie będzie się dało zmienić podczas działania programu.

Załóżmy, że w programie bankowym występuje takie typowe w przypadku udzielania kredytów polecenie:

```
Set amount = balance * 0.069
```

W programie takim pojawią się dwa potencjalne problemy. Pierwszy polega na tym, że tylko programista, który pisał program, będzie wiedział, co oznacza liczba 0,069. Wygląda na to, że jest to oprocentowanie kredytu, ale czasami dochodzą także inne opłaty związane z kredytem. W jaki sposób dowiemy się więc, co oznacza ta liczba, bez konieczności czasochłonnego sprawdzania pozostały części kodu?

Drugi problem pojawi się, gdy będziemy musieli wykorzystać tę liczbę w innym miejscu w programie, ale będzie ona ulegała zmianie co jakiś czas. Załóżmy, że liczbą tą jest wartość oprocentowania — co się stanie, kiedy oprocentowanie zmieni się z 6,9% na 7,2%? Programista w takim przypadku będzie musiał prześledzić cały kod programu i znaleźć miejsca, w których występuje liczba 0,069. W takiej sytuacji nie można skorzystać z funkcji „znajdź i zamień” edytora tekstowego i zamienić wszystkich wystąpień liczby 0,069, gdyż możemy zmienić wartość, która nie powinna zostać zmieniona. Spowoduje to powstanie w programie błędu logicznego.

Obu tym problemom można zaradzić, korzystając ze stałych nazwanych. **Stała nazwana** to wartość, której nie da się zmienić w trakcie działania programu, a która reprezentowana jest przez określoną nazwę. Oto przykład deklaracji stałej nazwanej w pseudokodzie:

```
Constant Real INTEREST_RATE = 0.069
```

Dzięki temu utworzyliśmy stałą o nazwie `INTEREST_RATE`. Stała ta przechowuje wartość typu `Real` równą 0,069. Zwrót uwagę, że deklaracja ta wygląda podobnie jak deklaracja zmiennej, z tą różnicą, że zamiast słowa `Declare` występuje słowo `Constant`. Zauważ także, że nazwa stałej jest zapisana dużymi literami. Jest to bardzo często stosowana technika w wielu językach programowania, gdyż dzięki niej bardzo łatwo można odróżnić w kodzie stałe od zmiennych. Podczas deklarowania stałej nazwanej trzeba też do niej przypisać wartość.

Argumentem przemawiającym na korzyść stałych jest to, że dzięki nim program staje się bardziej oczywisty. Polecenie:

```
Set amount = balance * 0.069
```

możemy zamienić na bardziej opisowe:

```
Set amount = balance * INTEREST_RATE
```

Nawet programista, który zajrzy do takiego kodu po raz pierwszy, będzie wiedział, do czego służy to polecenie. Wynika z niego jasno, że saldo (zmienna `balance`) mnożymy przez oprocentowanie (stała `INTEREST_RATE`). Kolejną zaletą takiego rozwiązania jest to, że bardzo łatwo można wprowadzić zmiany w całym programie. Powiedzmy, że wartość oprocentowania pojawia się w wielu miejscach programu. Kiedy oprocentowanie się zmieni, wystarczy, że zmodyfikujesz tylko wartość, którą zainicjalizowałeś zmienną nazwaną. Jeśli oprocentowanie zmieni się na 7,2%, wystarczy zmienić deklarację stałej na:

```
Constant Real INTEREST_RATE = 0.072
```

Nowa wartość oprocentowania będzie od teraz automatycznie stosowana w każdej instrukcji, w której odwołaliśmy się do stałej `INTEREST_RATE`.



**UWAGA:** Do stałej nazwanej nie da się przypisać wartości za pomocą instrukcji `Set`. Jeżeli w programie pojawi się polecenie, które będzie próbowało to zrobić, pojawi się błąd.

## 2.6

## Ręczne śledzenie programu

**WYJAŚNIENIE:** Ręczne śledzenie programu to prosta technika debugowania, dzięki której można wychwycić w programie trudne do wykrycia błędy.

Ręczne śledzenie (ang. *hand tracing*) programu polega na postawieniu się w roli komputera wykonującego dany program. Należy prześledzić po kolejnej instrukcji programu i zapisać wartości, jakie będą zapisane w zmiennych po wykonaniu danej instrukcji. Technika ta okazuje się szczególnie pomocna w wychwytywaniu błędów matematycznych i logicznych.

Aby prześledzić ręcznie program, musisz narysować tabelkę, w której kolumny będą odpowiadały kolejnym zmiennym, a wiersze — kolejnym instrukcjom. Na rysunku 2.17 przedstawiłem taką tabelkę dla przykładowego programu z podrozdziału 2.4. Każda kolumna tabelki odpowiada kolejno zmiennym: `test1`, `test2`, `test3` i `average`. Tabelka ma też dziewięć wierszy — każdy odpowiada jednej instrukcji programu.

Aby prześledzić program, sprawdź, co robi dana instrukcja, i zapisz w tabelce wartości, jakie przyjmą zmienne po wykonaniu danej instrukcji. Kiedy skończysz pracę, tabelka powinna wyglądać jak na rysunku 2.18. Znaki zapytania widoczne w tabelce oznaczają, że dana zmienna nie została zainicjalizowana.

```

1 Declare Real test1
2 Declare Real test2
3 Declare Real test3
4 Declare Real average
5
6 Set test1 = 88.0
7 Set test2 = 92.5
8 Set average = (test1 + test2 + test3) / 3
9 Display "Twój średni wynik wynosi ", average

```

	test1	test2	test3	average
1				
2				
3				
4				
5				
6				
7				
8				
9				

Rysunek 2.17. Program i tabelka do ręcznego śledzenia programu

```

1 Declare Real test1
2 Declare Real test2
3 Declare Real test3
4 Declare Real average
5
6 Set test1 = 88.0
7 Set test2 = 92.5
8 Set average = (test1 + test2 + test3) / 3
9 Display "Twój średni wynik wynosi ", average

```

	test1	test2	test3	average
1	?	?	?	?
2	?	?	?	?
3	?	?	?	?
4	?	?	?	?
5	?	?	?	?
6	88	?	?	?
7	88	92.5	?	?
8	88	92.5	?	Brak danych
9	88	92.5	?	Brak danych

Rysunek 2.18. Program i uzupełniona tabelka do śledzenia programu

Kiedy dojdziemy do linii 8., musimy wykonać działanie matematyczne — trzeba więc przyjrzeć się, jakie wartości mają zmienne wykorzystane w tym wyrażeniu. W tym miejscu odkryjemy, że zmienna `test3` nie została zainicjalizowana. Ponieważ zmienna jest niezainicjalizowana, nie jesteśmy w stanie określić, jaka wartość jest w niej zapisana. W rezultacie nie da się także określić wyniku obliczeń. Gdy odkryjemy ten fakt, będziemy mogli poprawić program, dodając do niego linię, w której zainicjalizujemy zmienną `test3`.

Ręczne śledzenie programu to prosta technika, dzięki której możesz przyjrzeć się każdej instrukcji i znaleźć nieoczywiste błędy.

## 2.7

## Dokumentowanie programu

**WYJAŚNIENIE:** Zewnętrzna dokumentacja programu służy do opisania aspektów działania programu istotnych dla użytkownika końcowego. Dokumentacja wewnętrzna jest przeznaczona dla programisty i zawiera opis sposobu działania poszczególnych fragmentów kodu.

W dokumentacji programu znajduje się opis, który wskazuje, jak z niego korzystać. Zazwyczaj mamy do czynienia z dwoma rodzajami dokumentacji: zewnętrzną i wewnętrzną. Dokumentacja zewnętrzna jest zazwyczaj przeznaczona dla użytkownika

końcowego. Składają się na nią takie kwestie jak opis funkcji programu czy samouczki służące do zaznajomienia użytkownika z programem.

Niekiedy to obowiązkiem programisty będzie stworzenie pełnej lub częściowej dokumentacji zewnętrznej programu. Często sytuacja taka ma miejsce w małych firmach lub w firmach z niewielką liczbą programistów. W niektórych przedsiębiorstwach, szczególnie w bardzo dużych, zatrudnia się pracowników, których zadaniem jest tylko i wyłącznie tworzenie dokumentacji zewnętrznych. Dokumentacja taka może mieć formę wydrukowanych instrukcji obsługi lub być zapisana w plikach, które można przeglądać na komputerze. Obecnie bardzo popularne stało się dostarczanie przez producentów oprogramowania dokumentacji w formie plików PDF (Printable Document Format).

**Dokumentacja wewnętrzna** ma formę **komentarzy** w kodzie programu. Komentarze to krótkie informacje tekstowe, umieszczone w różnych miejscach kodu, informujące o tym, jak dany fragment kodu działa. Komentarze, pomimo ogromnego znaczenia, są jednak pomijane przez kompilator czy interpreter. Są więc przeznaczone dla programisty, a nie dla komputera.

Do oznaczania w kodzie komentarzy służą w wielu językach specjalne znaki lub słowa. W niektórych językach, między innymi w Javie, C i C++, komentarz rozpoczyna się od podwójnych ukośników (//). Wszystko, co umieścisz w danej linii za tym symbolem, zostanie zignorowane przez kompilator. Oto przykład takiego komentarza:

```
// Pobieramy liczbę przepracowanych godzin
```

W niektórych językach do oznaczania początku komentarza korzysta się z innych symboli. Przykładowo w języku Visual Basic stosuje się do tego celu znak apostrofu ('), a w Pythonie — symbol #. W tej książce będę jednak korzystał z podwójnych ukośników (//)

## Komentarze blokowe i liniowe

Programiści umieszczają zazwyczaj w kodzie dwa rodzaje komentarzy: blokowe i liniowe. **Komentarze blokowe** zajmują wiele linii kodu i służą do szczegółowego opisania fragmentu kodu. Przykładowo na początku programu pojawia się przeważnie komentarz blokowy opisujący, do czego służy program, kto jest jego autorem, jaka jest data ostatniej modyfikacji i wiele innych pozytywnych informacji. Oto przykład komentarza blokowego:

```
// Program służy do obliczania wynagrodzenia pracownika.  
// Autorem jest Matt Hoyle.  
// Data ostatniej modyfikacji: 2018-12-14
```



**UWAGA:** W niektórych językach programowania do oznaczania początku i końca komentarza blokowego stosuje się specjalne symbole.

Komentarze liniowe zajmują tylko jedną linię i służą do objaśnienia krótkiego fragmentu kodu. Oto przykład takich komentarzy:

```
// Obliczanie odsetek
Set interest = balance * interest_Rate
// Dodanie odsetek do salda
Set balance = balance + interest
```

Komentarz liniowy nie musi zajmować całej linii. Kompilator ignoruje w danej linii wszystko, co występuje po symbolu //, więc możemy umieścić komentarz także w linii, w której pojawia się inna instrukcja. Oto przykład:

```
Input age      // Pobieramy wiek
```

Jako początkujący programista być może będziesz podchodzić sceptycznie do umieszczania w kodzie komentarzy — przecież najfajniejsze jest pisanie kodu, który faktycznie coś robi! Niemniej poświęć trochę czasu i komentuj kod programu. Komentarze z całą pewnością zaoszczędzą Ci w przyszłości sporo czasu, kiedy przyjdzie Ci zmodyfikować program albo usunąć z niego błędy. Nawet bardzo duże i skomplikowane programy da się łatwo zrozumieć, gdy są w nich umieszczone w odpowiedni sposób komentarze.

## W centrum uwagi

### Korzystanie ze stałych nazwanych, konwencje zapisu i komentarze



Powiedzmy, że mamy do czynienia z następującym zadaniem: naukowcy określili, że poziom wody w oceanach podnosi się w ciągu roku o 1,5 milimetra. Musimy napisać program, który będzie wyświetlał następujące informacje:

- o ile milimetrów podniesie się poziom wody w oceanach w ciągu 5 lat;
- o ile milimetrów podniesie się poziom wody w oceanach w ciągu 7 lat;
- o ile milimetrów podniesie się poziom wody w oceanach w ciągu 10 lat.

Oto algorytm:

1. Obliczyć, o ile milimetrów podniesie się poziom wody w oceanach w ciągu 5 lat.
2. Wyświetlić wynik obliczeń z kroku 1.
3. Obliczyć, o ile milimetrów podniesie się poziom wody w oceanach w ciągu 7 lat.
4. Wyświetlić wynik obliczeń z kroku 3.
5. Obliczyć, o ile milimetrów podniesie się poziom wody w oceanach w ciągu 10 lat.
6. Wyświetlić wynik obliczeń z kroku 5.

Program jest bardzo prosty: oblicza trzy wartości i wyświetla je na ekranie. W wyniku obliczeń otrzymamy wartości, o jakie podniesie się poziom wody w oceanie w ciągu 3, 7 i 10 lat. Każdą z tych wartości można obliczyć za pomocą wzoru:

$$\text{roczny wzrost poziomu wody} \cdot \text{liczba lat}$$

Rocznego wzrostu poziomu wody będzie taki sam w przypadku wszystkich obliczeń, więc stworzymy stałą, która będzie reprezentowała tę wartość. Na listingu 2.14 przedstawiłem pełny pseudokod takiego programu.

**Listing 2.14**

```

1 // Deklaracja zmiennych
2 Declare Real fiveYears
3 Declare Real sevenYears
4 Declare Real tenYears
5
6 // Tworzymy stałą, w której zapisany będzie roczny wzrost poziomu wody
7 Constant Real YEARLY_RISE = 1.5
8
9 // Wyświetlamy, o ile podniesie się poziom wody po 5 latach
10 Set fiveYears = YEARLY_RISE * 5
11 Display "Poziom wody w oceanach podniesie się po pięciu latach o ", fiveYears,
12     " mm."
13
14 // Wyświetlamy, o ile podniesie się poziom wody po siedmiu latach
15 Set sevenYears = YEARLY_RISE * 7
16 Display "Poziom wody w oceanach podniesie się po siedmiu latach o ", sevenYears,
17     " mm."
18
19 // Wyświetlamy, o ile podniesie się poziom wody po dziesięciu latach
20 Set tenYears = YEARLY_RISE * 10
21 Display "Poziom wody w oceanach podniesie się po dziesięciu latach o ", tenYears,
22     " mm."

```

**Wynik działania programu**

Poziom wody w oceanach podniesie się po pięciu latach o 7.5 mm.

Poziom wody w oceanach podniesie się po siedmiu latach o 10.5 mm.

Poziom wody w oceanach podniesie się po dziesięciu latach o 15 mm.

W liniach od 2. do 4. zadeklarowałem trzy zmienne: `fiveYears`, `sevenYears` i `tenYears`.

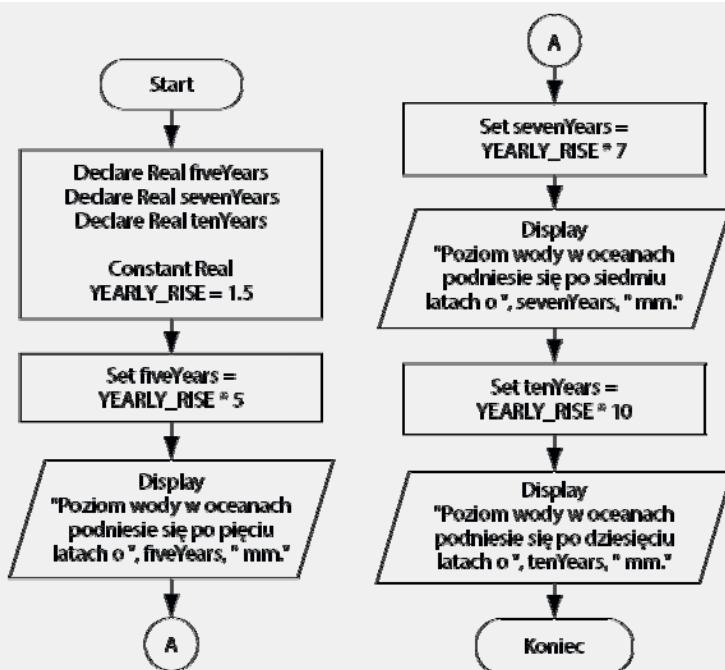
W zmiennych tych będę zapisywał wartość, o jaką podniesie się poziom wody w oceanach w ciągu 5, 7 i 10 lat.

W linii 7. zadeklarowałem stałą o nazwie `YEARLY_RISE` i przypisałem do niej wartość 1,5. Informuje ona, o ile milimetrów podnosi się rocznie poziom wody w oceanach.

W liniach od 10. do 12. obliczam i wyświetlам, o ile milimetrów podniesie się poziom wody w oceanach w ciągu 5 lat. Podobnie robię w przypadku okresu 7- i 10-letniego — odpowiednio w liniach od 15. do 17. i od 20. do 22.

W programie tym zaprezentowałem następujące konwencje zapisu:

- Zostawiłem kilka pustych linii kodu (linie 5., 8., 13. i 18.). Nie mają one wpływu na to, jak działa program, ale poprawiają czytelność kodu.
- W kilku miejscach wstawiłem komentarze informujące, co w danych fragmentach kodu się dzieje.
- Każda z instrukcji `Display` występujących w programie nie mieści się w jednej linii kodu (linie 11., 12., 16., 17., 21. i 22.). Większość języków programowania umożliwia zapisywanie długich instrukcji w kilku liniach kodu. W przypadku pseudokodu każdą kolejną linię będę wcinał w kodzie względem pierwszej linii, w której zaczyna się dana instrukcja. Dzięki temu będzie bardziej czytelne, że instrukcja znajduje się w kilku liniach. Na rysunku 2.19 widoczny jest schemat blokowy programu.



**Rysunek 2.19.** Schemat blokowy programu z listingu 2.14



## Punkt kontrolny

- 2.30. Co to jest dokumentacja zewnętrzna?
- 2.31. Co to jest dokumentacja wewnętrzna?
- 2.32. Wymień dwa rodzaje komentarzy, jakimi mogą posługiwać się programiści w kodzie programu. Opisz każdy z nich.

## 2.8

# Projektowanie pierwszego programu

W pewnych sytuacjach początkujący programista może mieć problem z rozpoczęciem pracy nad programem. W tym podrozdziale przedstawię proste zadanie i przeanalizuję wymagania programu, zaprojektuję za pomocą pseudokodu algorytm i zaprezentuję schemat blokowy programu. Oto nasze przykładowe zadanie:

## Skuteczność odbijania piłki

W baseballu skuteczność odbijania piłki informuje o tym, jak dobrym pałkarzem jest dany zawodnik. Aby obliczyć skuteczność odbijania piłki, posłużymy się następującym wzorem:

$$\text{skuteczność} = \text{odbicia} : \text{próby}$$

We wzorze tym *odbicia* to liczba udanych prób odbicia piłki przez gracza, a *próby* to liczba wszystkich prób odbicia piłki. Przykładowo, jeżeli zawodnik w czasie sezonu próbował odbić piłkę 500 razy, a odbił ją tylko 150 razy, to jego skuteczność wynosi 0,3. Zaprojektuj program, który będzie obliczał skuteczność odbijania dla dowolnego zawodnika.

Przypomnij sobie, jak w podrozdziale 2.2 wspomniałem, że program zazwyczaj można podzielić na trzy fazy:

1. Pobranie danych wejściowych.
2. Przetwarzanie danych wejściowych (np. dokonywanie obliczeń).
3. Zwrócenie danych wyjściowych.

Twoim pierwszym zadaniem jest określenie, czego będziesz potrzebować w każdej z tych faz. Zazwyczaj nie znajdziesz tych informacji w samym opisie zadania. Na przykład we wspomnianym przed chwilą zadaniu polegającym na obliczaniu skuteczności odbijania piłek jest jedynie wyjaśnienie, czym jest ta skuteczność, oraz informacja, że program powinien ją obliczać. Teraz musisz nieco pogłówkować i określić, czego będziesz potrzebować w poszczególnych fazach. Przyjrzyjmy się bliżej każdej z faz w kontekście naszego programu do obliczania skuteczności odbijania piłek.

### 1. Pobieramy dane wejściowe.

Aby wskazać, jakich danych wejściowych będziemy potrzebować, musimy określić, jakie informacje będą nam potrzebne, aby program wykonał powierzone mu zadanie. Kiedy przyjrzyz się wzorowi na skuteczność odbijania piłki, zauważysz, że pojawiają się w nim dwie wartości, które będą nam potrzebne do obliczeń:

- liczba udanych odbić;
- liczba wszystkich prób odbicia piłki.

Ponieważ nie znamy tych wartości, będziemy je musieli uzyskać od użytkownika naszego programu.

Każda z tych informacji będzie zapisana w zmiennej. Podczas projektowania programu musisz zadeklarować te zmienne, więc już w tym momencie pomocne będzie wymyślenie nazw zmiennych i ich typów. W naszym programie stworzymy zmienną `hits`, w której zapiszemy liczbę udanych odbić, i zmienną o nazwie `atBat`, w której zapiszemy liczbę wszystkich prób obicia piłki. Obie te wartości są liczbami całkowitymi, więc zadeklarujemy je jako `Integer`.

### 2. Przetwarzamy dane wejściowe (np. dokonujemy obliczeń).

Kiedy już pobierzemy od użytkownika dane, możemy przystąpić do wykonywania na nich pewnych obliczeń lub innych operacji. W naszym programie podzielimy liczbę udanych odbić przez liczbę wszystkich prób odbicia. Wynikiem tych obliczeń będzie wartość informująca o skuteczności odbijania piłek przez danego zawodnika.

Pamiętaj, że zazwyczaj będziesz chciał zapisać wyniki obliczeń w zmiennych — musisz więc pomyśleć na tym etapie projektu, jakiego typu powinny być to zmienne i jak je nazwać. W naszym przypadku do zapisania wyniku obliczeń

użyjemy zmiennej o nazwie `battingAverage`. Ponieważ wynikiem obliczeń będzie liczba rzeczywista, wybierzemy dla zmiennej typ `Real`.

### 3. Zwracamy dane wyjściowe.

Dane wyjściowe to zazwyczaj po prostu wynik obliczeń, których dokonaliśmy wcześniej. Danymi wyjściowymi w naszym przykładzie będzie zatem wynik obliczeń zapisany w zmiennej `battingAverage`. Program wyświetli tę wartość razem z komunikatem informującym, co dana wartość reprezentuje.

Kiedy już określmy, czym są w naszym przykładzie dane wejściowe, jak wygląda ich przetwarzanie i czym są dane wyjściowe, możemy przystąpić do pisania pseudokodu i/lub schematu blokowego. Zacznijmy od zadeklarowania zmiennych:

```
Declare Integer hits
Declare Integer atBat
Declare Real battingAverage
```

Następnie napiszemy pseudokod, za pomocą którego pobierzemy dane od użytkownika. Przypominam, że zazwyczaj będzie to proces składający się z dwóch etapów: (1) wyświetlenia komunikatu, który poinformuje użytkownika, jakich informacji od niego oczekujemy, oraz (2) pobrania wprowadzonych na klawiaturze danych i zapisania ich w zmiennej. Oto pseudokod odpowiedzialny za odczytanie dwóch informacji, których będziemy potrzebować w naszym programie:

```
Display "Wprowadź liczbę udanych odbić zawodnika."
Input hits

Display "Wprowadź liczbę wszystkich prób odbicia piłki przez zawodnika."
Input atBat
```

Następnie napiszemy kod odpowiedzialny za obliczenia:

```
Set battingAverage = hits / atBat
```

Na końcu napiszemy kod, dzięki któremu wynik wyświetli się na ekranie:

```
Display "Skuteczność odbijania piłki przez zawodnika wynosi ", battingAverage
```

Teraz możemy połączyć te fragmenty i utworzyć kompletny program. Na listingu 2.15 przedstawiłem cały program, razem z komentarzami, a na rysunku 2.20 zamieściłem jego schemat blokowy.

#### **Listing 2.15**

```
1 // Deklaracja zmiennych
2 Declare Integer hits
3 Declare Integer atBat
4 Declare Real battingAverage
5
6 // Pobieramy liczbę udanych odbić
7 Display "Wprowadź liczbę udanych odbić zawodnika."
8 Input hits
9
10 // Pobieramy liczbę wszystkich prób odbicia piłki
11 Display "Wprowadź liczbę wszystkich prób odbicia piłki przez zawodnika."
12 Input atBat
13
```

```

14 // Obliczamy skuteczność odbijania piłki
15 Set battingAverage = hits / atBat
16
17 // Wyświetlamy wynik
18 Display "Skuteczność odbijania piłki przez zawodnika wynosi ", battingAverage

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

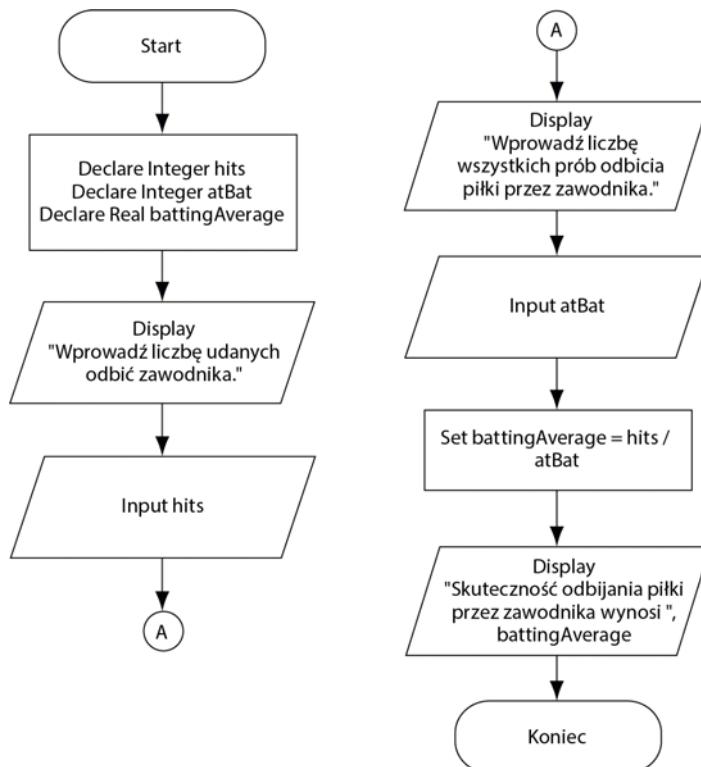
Wprowadź liczbę udanych odbić zawodnika.

**150 [Enter]**

Wprowadź liczbę wszystkich prób odbicia piłki przez zawodnika.

**500 [Enter]**

Skuteczność odbijania piłki przez zawodnika wynosi 0.3



**Rysunek 2.20.** Schemat blokowy programu z listingu 2.15

### Podsumowanie

Jako początkujący programista, ilekroć będziesz mieć problem z rozpoczęciem projektu programu, pomyśl o następujących elementach:

1. **Dane wejściowe.** Dokładnie przemyśl zadanie i wskaż, które dane program będzie musiał pobrać od użytkownika, a następnie określ typy tych danych i nazwy zmiennych, do których je przypiszesz.
2. **Przetwarzanie.** Co program zrobi z danymi, które wprowadził użytkownik? Określ, jakie obliczenia lub inne operacje trzeba wykonać na tych danych.

W tym momencie wymyśl nazwy i typy zmiennych, w których zapiszesz wyniki obliczeń.

3. **Dane wyjściowe.** Jakie dane musi zwracać program? W większości przypadków będzie to wynik obliczeń lub innych operacji na danych wejściowych.

Kiedy już określisz wszystkie te elementy, będziesz wiedzieć, jak musi działać dany program. Będziesz też mieć listę potrzebnych zmiennych i ich typy. Następnym etapem jest napisanie algorytmu za pomocą pseudokodu lub narysowanie schematu blokowego programu.

## 2.9

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

### Java

Wejście, przetwarzanie i wyjście w języku Java

#### Przygotowywanie programu w języku Java

Zaczynając tworzyć nowy program w Javie, musisz najpierw zapisać deklarację klasy przechowującej kod programu. Na razie możesz traktować deklarację klasy jak kontener przechowujący cały kod w języku Java. Oto przykład takiej deklaracji:

```
public class Simple
{
}
```

Pierwszy wiersz deklaracji klasy nazywany jest jej **nagłówkiem klasy**. W podanym tu przykładzie nagłówek klasy wygląda następująco:

```
public class Simple
```

Słowa `public` i `class` są słowami kluczowymi języka Java, a `Simple` jest nazwą deklarowanej klasy. Zauważ, że oba słowa — `public` i `class` — zapisane są wyłącznie małymi literami. Jeżeli przez pomyłkę umieścisz dużą literę w słowie kluczowym, to pojawi się błąd, a programu nie uda się skompilować.

Nazwa klasy, która w tym przypadku brzmi `Simple`, nie musi być zapisywana wyłącznie małymi literami, ponieważ nie jest ona słowem kluczowym Javy. Jest zwyczajnym słowem, którego programista używa jako nazwy swojej klasy. Zauważ, że pierwszy

znak nazwy klasy zapisany jest dużą literą. Nie jest to wymagane, ale w Javie przyjęło się zapisywać nazwy klas z pierwszą dużą literą. Programiści stosują tę konwencję, aby łatwiej odróżniać nazwy klas od nazw innych elementów w swoich programach.

Kolejną ważną rzeczą do zapamiętania na temat nazw klas jest to, że nazwa klasy musi być taka sama jak nazwa pliku, w którym znajduje się ta klasa. Na przykład jeżeli przygotujesz klasę o nazwie `Simple` (której pokazałem wcześniej), to jej deklaracja musi znaleźć się w pliku o nazwie `Simple.java`. (Wszystkie pliki z kodem źródłowym języka Java mają rozszerzenie `.java`.)

Zauważ, że za nagłówkiem klasy znajdują się nawiasy klamrowe. Ich zadaniem jest grupowanie elementów, a w tym przypadku będą one gromadzić kod zapisany wewnętrz klas. Oznacza to, że w następnym kroku musimy zapisać w tych nawiasach jakiś kod.

W nawiasach klamrowych należy napisać definicję metody o nazwie `main`. Metoda jest kolejnym rodzajem kontenera przechowującego kod. W momencie uruchomienia programu napisanego w Javie automatycznie uruchamiany jest kod znajdujący się w metodzie `main`. Oto klasa `Simple` po dopisaniu do niej deklaracji metody `main`:

```
Public class Simple
{
    Public static void main(String[] args)
    {

    }
}
```

Pierwszy wiersz definicji metody nazywany jest **nagłówkiem metody** i zaczyna się od słów `public static void main` itd. W tym momencie nie musisz przejmować się dokładnym znaczeniem tych słów. Pamiętaj jedynie, że musisz zapisać nagłówek tej metody *dokładnie* tak jak pokazałem. Zauważ, że za nagłówkiem metody znów pojawiają się nawiasy klamrowe. Całość kodu zisanego dla tej metody musi się znaleźć w tych nawiasach.

### Wyświetlanie tekstu na ekranie w języku Java

Aby w Javie wyświetlić na ekranie jakiś tekst, należy użyć następujących polecień:

- `System.out.println()`
- `System.out.print()`

Najpierw przyjrzymy się poleceniu `System.out.println()`. Jego zadaniem jest wyświetlenie jednego wiersza tekstu. Zauważ, że polecenie to kończy się nawiasami okrągłymi. Tekst, który chcesz wyświetlić, powinien zostać zapisany jako ciąg znaków umieszczony w tych nawiasach. Przykładem będzie tutaj listing 2.16. (Pamiętaj, że numery wierszy NIE są elementem programu! Nie wpisuj ich podczas wprowadzania kodu programu. Numery wierszy podaję tutaj tylko po to, żeby móc się do nich odwoływać w tekście).

Polecenia pojawiające się w wierszach od 5. do 7. zakończone są średnikiem. Podobnie jak kropka kończy zdanie w języku polskim, średnik oznacza zakończenie polecenia w języku Java. Zauważysz jednak, że nie wszystkie wiersze kodu w programie będą kończyły się średnikiem.

**Listing 2.16. (Output.java)**

```

1 public class Output
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Zajmuję się informatyką.");
6         System.out.println("Chcę zostać programistą.");
7         System.out.println("Programowanie to świetna zabawa!");
8     }
9 }
```

**Wynik działania programu**

Zajmuję się informatyką.  
Chcę zostać programistą.  
Programowanie to świetna zabawa!

Na przykład nagłówki klas lub metod nie są zakończone średnikiem, ponieważ nie są uznawane za instrukcje. Oprócz tego średnik nie pojawia się też za nawiasami klamrowymi, bo one również nie są uznawane za instrukcje. (Jeżeli nic z tego nie rozumiesz, nie przejmuj się! Ćwicząc pisanie programów w języku Java, z czasem zaczniesz intuicyjnie odróżniać instrukcje i wiersze kodu niebędące instrukcjami).

Zauważ, że teksty wypisywane przez instrukcję `System.out.println()` pojawiają się w osobnych wierszach. Gdy instrukcja `System.out.println()` wyświetla jakiś tekst, to przy okazji przenosi też kurSOR (czyli pozycję, w której pojawi się następny wyświetlany element) do następnego wiersza. Oznacza to, że instrukcja `System.out.println()` wyświetla przekazany jej ciąg znaków, a kolejne wyświetlane teksty pojawią się już w następnym wierszu.

Instrukcja `System.out.print()` wyświetla podany jej w parametrze tekst, ale nie przekazuje kursora do następnego wiersza. Przykład jej działania prezentuję na listingu 2.17.

**Listing 2.17. (Output2.java)**

```

1 public class Output2
2 {
3     public static void main(String[] args)
4     {
5         System.out.print("Programowanie");
6         System.out.print("jest");
7         System.out.print("fajne.");
8     }
9 }
```

**Wynik działania programu**

Programowaniejestfajne.

Zauważ, że w wyniku działania programu poszczególne słowa sklejone są ze sobą i tworzą jeden ciąg znaków. Jeżeli chcesz, żeby pomiędzy słowami znajdowały się odstępy, to musisz je jawnie wpisać do wyświetlonego tekstu. Na listingu 2.18 pokazuję, jak

**Listing 2.18. (Output3.java)**

```

1 public class Output3
2 {
3     public static void main(String[] args)
4     {
5         System.out.print("Programowanie ");
6         System.out.print("jest ");
7         System.out.print("fajne. ");
8     }
9 }
```

**Wynik działania programu**

Programowanie jest fajne.

należy wstawać spacje do wyświetlanego ciągu znaków, aby oddzielić od siebie słowa widoczne na ekranie. Zauważ, że w wierszu 5. spacja pojawiła się w ciągu znaków, zaraz za literą e, natomiast w wierszu 6. spację dopisałem za literą t.

**Zmienne w języku Java**

W Javie zmienne muszą zostać zadeklarowane przed ich pierwszym użyciem w programie. Instrukcja deklarująca zmienną zapisywana jest z zastosowaniem poniższego formatu:

*TypDanychNazwaZmiennej;*

W tym ogólnym formacie *TypDanych* jest nazwą typu danych języka Java, natomiast *NazwaZmiennej* to nazwa deklarowanej zmiennej. Instrukcja deklaracji musi być zakończona średnikiem. Na przykład słowo kluczowe int oznacza w Javie typ danych liczby całkowitej, dlatego poniższa instrukcja deklaruje zmienną o nazwie number:

int number;

W tabeli 2.6 prezentuję listę typów danych Javy. Podaję w niej też dokładniejszy opis poszczególnych typów danych z zakresem wartości, jakie mogą one przechowywać. Zaznaczam, że w tej książce będziemy przede wszystkich korzystać z typów danych int, double i String<sup>3</sup>.

Oto inne przykłady deklaracji zmiennych w języku Java:

```

int speed;
double distance;
String name;
```

W ramach jednej instrukcji można zadeklarować kilka zmiennych o takim samym typie danych. Na przykład w poniżej instrukcji deklarowane są trzy zmienne typu int o nazwach: width, height i length:

int width, height, length;

---

<sup>3</sup> Zauważ, że nazwa typu String zapisywana jest z pierwszą wielką literą. Wynika to stąd, że String nie jest typem danych Javy, lecz klasą. Mimo to będziemy jej używać jak zwykłego typu danych.

**Tabela 2.6.** Typy danych języka Java

Typ danych	Co może przechowywać?
byte	Liczby całkowite od -128 do +127
short	Liczby całkowite od -32 768 do +32 767
int	Liczby całkowite od -2 147 483 648 do +2 147 483 647
long	Liczby całkowite od -9 223 372 036 854 775 808 do +9 223 372 036 854 775 807
float	Liczby zmiennoprzecinkowe od $\pm 3,4 \times 10^{-38}$ do $\pm 3,4 \times 10^{38}$ , z dokładnością do 7 cyfr
double	Liczby zmiennoprzecinkowe od $\pm 1,7 \times 10^{-308}$ do $\pm 1,7 \times 10^{308}$ , z dokładnością do 15 cyfr
String	Ciągi znaków

Podczas deklarowania zmiennych można też zainicjalizować je, przypisując im początkową wartość. W poniższej instrukcji deklarowana jest zmienna typu `int` o nazwie `hours` i przypisywana jest jej początkowa wartość 40:

```
int hours = 40;
```

### Nazwy zmiennych w języku Java

W Javie możesz samodzielnie wybierać nazwy zmiennych, pod warunkiem że nie użyjesz żadnego słowa kluczowego tego języka. Słowa kluczowe stanowią trzon całego języka, a każde z nich ma określone zadanie. W tabeli 2.2 dodatku *Java Language Companion* znajdziesz pełną listę słów kluczowych języka Java.

Oto kilka reguł, których trzeba przestrzegać przy nadawaniu nazw zmiennym w języku Java:

- Pierwszy znak nazwy zmiennej musi być małą lub dużą literą (a – z, A – Z), znakiem podkreślenia (\_) albo symbolem dolara (\$).
- Za pierwszym znakiem można używać małych lub dużych liter (a – z, A – Z), cyfr (0 – 9), znaku podkreślenia (\_) oraz symbolu dolara (\$).
- Rozróżniane są wielkości liter użytych w nazwach. Oznacza to, że nazwy `itemsOrdered` i `itemsordered` nie są jednakowe.
- W nazwach zmiennych nie można stosować spacji.

Na listingu 2.19 można zobaczyć trzy deklaracje zmiennych. W wierszu 5. deklarowana jest zmienna typu `String` o nazwie `name` i przypisywana jest jej wartość początkowa „Jan Kowalski”. W wierszu 6. deklarowana jest zmienna typu `int` o nazwie `hours`, której przypisywana jest wartość początkowa 40. W wierszu 7. deklarowana jest zmienna typu `double` o nazwie `pay` razem z wartością początkową 852,99. Zwróć uwagę, że w wierszach od 9. do 11. korzystamy z instrukcji `System.out.println()`, aby wyświetlić zawartość każdej z tych zmiennych.

**Listing 2.19. (VariableDemo.java)**

```

1 public class VariableDemo
2 {
3     public static void main(String[] args)
4     {
5         String name = "Jan Kowalski";
6         int hours = 40;
7         double pay = 852,99;
8
9         System.out.println(name);
10        System.out.println(hours);
11        System.out.println(pay);
12    }
13 }
```

**Wynik działania programu**

Jan Kowalski

40

852.99

**Pobieranie danych z klawiatury w języku Java**

Aby w Javie odczytać dane z klawiatury, musisz utworzyć w pamięci obiekt o nazwie Scanner. Za pomocą obiektu Scanner możesz odczytywać dane z klawiatury i przypisywać je jako wartości do zmiennych. Na listingu 2.20 przedstawiam przykład realizacji tego zadania. (Jest to wersja pseudokodu z listingu 2.2 zapisana w języku Java). Przyjrzymy się dokładniej poszczególnym elementom tego kodu:

- Wiersz 1. zawiera polecenie `import java.util.Scanner;`. Jest ono niezbędne, żeby poinformować kompilator Javy, że mamy zamiar w programie skorzystać z obiektu Scanner.
- W wierszu 7. tworzymy obiekt Scanner i nadajemy mu nazwę keyboard.
- W wierszu 8. deklarujemy zmienną o nazwie age.
- W wierszu 10. wyświetlany jest tekst „Ile masz lat?“.
- W wierszu 11. z klawiatury odczytywana jest wartość liczby całkowitej, a następnie jest ona przypisywana do zmiennej age.
- W wierszu 12. wyświetlany jest tekst „Oto wartość, którą wprowadziłeś:“.
- W wierszu 13. wyświetlana jest wartość ze zmiennej age.

**Listing 2.20. (GetAge.java)**

```

1 import java.util.Scanner;
2
3 public class GetAge
4 {
5     public static void main(String[] args)
6     {
7         Scanner keyboard = new Scanner(System.in);
8         int age;
9
10        System.out.println("Ile masz lat?");
11        age = keyboard.nextInt();
12        System.out.println("Oto wartość, którą wprowadziłeś:");
```

```

13     System.out.println(age);
14 }
15 }
```

### **Wynik działania programu (pogrubieniem oznaczono wartości podane przez użytkownika)**

Ile masz lat?

**24 [Enter]**

Oto wartość, którą wprowadziłeś:

24

Zauważ, że w wierszu 11. użyliśmy wyrażenia `keyboard.nextInt()`, aby odczytać z klawiatury wartość liczby całkowitej. Jeżeli chcielibyśmy z klawiatury uzyskać liczbę typu `double`, to należałoby użyć wyrażenia `keyboard.nextDouble()`. Poza tym, chcąc z klawiatury odczytać ciąg znaków, należałoby skorzystać z wyrażenia `keyboard.nextLine()`.

Na listingu 2.21 prezentuję sposób użycia obiektu `Scanner` do odczytywania nie tylko liczb całkowitych, ale też liczb zmiennoprzecinkowych i ciągów znaków:

- W wierszu 7. tworzony jest obiekt `Scanner` i nadawana jest mu nazwa `keyboard`.
- W wierszu 8. deklarowana jest zmienna typu `String` o nazwie `name`, w wierszu 9. deklarowana jest zmienna typu `double` o nazwie `payRate`, natomiast w wierszu 10. deklarowana jest zmienna typu `int` o nazwie `hours`.
- W wierszu 13. używane jest wyrażenie `keyboard.nextLine()` do odczytania ciągu znaków z klawiatury i przypisania go do zmiennej `name`.
- W wierszu 16. wyrażenie `keyboard.nextDouble()` zostało użyte do odczytania z klawiatury wartości zmiennoprzecinkowej i zapisania jej w zmiennej `payRate`.
- W wierszu 19. przy pomocy wyrażenia `keybaord.nextInt()` odczytywana jest z klawiatury wartość liczby całkowitej i przypisywana jest do zmiennej `hours`.

### **Listing 2.21. (GetInput.java)**

```

1 import java.util.Scanner;
2
3 public class GetInput
4 {
5     public static void main(String[] args)
6     {
7         Scanner keyboard = new Scanner(System.in);
8         String name;
9         double payRate;
10        int hours;
11
12        System.out.print("Podaj swoje nazwisko: ");
13        name = keyboard.nextLine();
14
15        System.out.print("Podaj swoją stawkę za godzinę: ");
16        payRate = keyboard.nextDouble();
17
18        System.out.print("Podaj liczbę przepracowanych godzin: ");
19        hours = keyboard.nextInt();
20
21        System.out.println("Oto wartość, którą wprowadziłeś:");
22        System.out.println(name);
23        System.out.println(payRate);
```

```

24     System.out.println(hours);
25 }
26 }
```

### **Wynik działania programu (pogrubieniem oznaczono wartości podane przez użytkownika)**

Podaj swoje nazwisko: Jan Kowalski [Enter]

Podaj swoją stawkę za godzinę: 55.25 [Enter]

Podaj liczbę przepracowanych godzin: 40 [Enter]

Oto wartość, którą wprowadziłeś:

Jan Kowalski

55.25

40

### **Wyświetlanie wielu elementów za pomocą operatora +**

Jeżeli operator dodawania (+) zostanie użyty na ciągach znaków, to staje się tak zwanym **operatorem konkatenacji** tych ciągów. Konkatenacja jest operacją łączenia, a zatem operator konkatenacji ciągów znaków łączy dwa ciągi znaków. Spójrz na poniższe polecenie:

```
System.out.println("To jest " + "ciąg znaków.");
```

Polecenie to spowoduje wyświetlenie takiego tekstu:

To jest ciąg znaków.

Operator dodawania tworzy zatem nowy ciąg znaków powstały w wyniku łączenia dwóch ciągów podanych jako parametry operatora. Operator dodawania można stosować też w celu konkatenacji zawartości zmiennej i ciągu znaków. Spójrz na poniższy przykład:

```
number = 5;
System.out.println("Wartość wynosi " + number);
```

W drugim wierszu korzystam z operatora dodawania, aby połączyć zawartość zmiennej number z ciągiem znaków "Wartość wynosi ". Mimo że zmienna number nie jest ciągiem znaków, operator dodawania zmienia ją najpierw w ciąg znaków, a następnie uzyskaną wartość łączy z pierwszym ciągiem. W wyniku działania takiego programu zostanie wypisany następujący tekst:

Wartość wynosi 5

### **Wykonywanie obliczeń w języku Java**

Operatory arytmetyczne Javy są niemal identyczne jak prezentowane wcześniej w tym rozdziale, w tabeli 2.1.

---

+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Dzielenie modulo

---

Oto kilka przykładów instrukcji Javy korzystających z operatora arytmetycznego w celu wyliczenia pewnej wartości i przypisania jej do wybranej zmiennej:

```
total = price + tax;
sale = price - discount;
population = population * 2;
half = number / 2;
leftOver = 17 % 3;
```

W Javie nie ma operatora potęgowania, ale dostępna jest metoda `Math.pow`, która wykonuje tę operację. Oto przykład wykorzystania metody `Math.pow` w programie:

```
result = Math.pow(4.0, 2.0);
```

Ta metoda pobiera dwa argumenty (liczby zapisane w nawiasach) typu `double`, a następnie podnosi wartość pierwszego argumentu do potęgi z drugiego argumentu i zwraca uzyskaną liczbę jako wartość typu `double`. W tym przykładzie liczba `4.0` podnoszona jest do potęgi `2.0`. Instrukcja ta jest zatem równoważna z poniższym wyrażeniem algebraicznym:

```
result = 42
```

### **Stałe nazywane w języku Java**

W Javie stałe nazywane tworzone są za pomocą słowa kluczowego `final` umieszczonego w deklaracji zmiennej. Słowo to musi się znaleźć tuż przed definicją typu danych. A oto przykład:

```
final double INTEREST_RATE = 0.069;
```

Ta instrukcja wygląda jak zwykła deklaracja zmiennej, z tą różnicą, że słowo `final` pojawia się przed deklaracją typu danych, natomiast nazwa zmiennej jest zapisana wielkimi literami.

Nie istnieje żaden wymóg, aby nazwy zmiennych były zapisywane wielkimi literami, ale wielu programistów zapisuje je w ten sposób, żeby można je było łatwiej odróżnić od nazw zwykłych zmiennych.

### **Tworzenie dokumentacji programu w języku Java za pomocą komentarzy**

Aby napisać komentarz wierszowy w Javie, wystarczy umieścić dwa ukośniki (`//`) w miejscu, w którym ma się on rozpocząć. Kompilator będzie ignorował wszystko od tego miejsca do końca wiersza. Oto przykład:

```
// Ten program oblicza wynagrodzenie brutto pracownika
```

Komentarze wielowierszowe lub komentarze blokowe zaczynają się od znaków `/*` (ukośnik, a następnie gwiazdka) i kończą się znakami `*/` (gwiazdka z następującym po niej ukośnikiem). Wszystko pomiędzy tymi znacznikami będzie ignorowane. Oto przykład:

```
/*
Ten program ten oblicza wynagrodzenie brutto pracownika,
w tym wynagrodzenie za przepracowane nadgodziny
*/
```

## Python

Wejście, przetwarzanie i wyjście w języku Python

### Wyświetlanie ekranu wynikowego w języku Python

W Pythonie do wyświetlenia danych wyjściowych na ekranie komputera należy użyć funkcji `print`. Oto przykład instrukcji, która wykorzystuje funkcję `print`, aby wyświetlić komunikat  `Witaj, świecie!`:

```
print('Witaj, świecie!')
```

Chcąc użyć funkcji `print`, należy wpisać słowo `print`, a następnie nawiasy okrągłe. W nawiasach wpisujemy **argument**, czyli dane, które chcemy wyświetlić na ekranie. W tym przykładzie argumentem są słowa `'Witaj, świecie!'`. Gdy instrukcja zostanie wykonana, znaki cudzysłowu nie będą wyświetlane, ponieważ wyznaczają one tylko początek i koniec tekstu, który chcemy wyświetlić. Listing 2.22 pokazuje przykład pełnego programu wyświetlającego dane wyjściowe na ekranie. (Pamiętaj, że numery wierszy NIE są częścią programu! Nie wpisuj tych numerów podczas wprowadzania kodu programu. Numery wierszy są tu pokazane wyłącznie w celach informacyjnych).

#### **Listing 2.22. (output.py)**

```
1 print('Zajmuję się informatyką.')
2 print('Chcę zostać programistą.')
3 print('Programowanie to świetna zabawa!')
```

#### **Wynik działania programu**

```
Zajmuję się informatyką.
Chcę zostać programistą.
Programowanie to świetna zabawa!
```

W Pythonie można umieścić ciągi znaków w zestawie apostrofów ('') lub cudzysłówów (""). Ciągi znaków przedstawione na listingu 2.22 są ujęte w znaki apostrofu, ale program może być również napisany tak:

```
print("Zajmuję się informatyką.")
print("Chcę zostać programistą.")
print("Programowanie to świetna zabawa!")
```

### Zmienne w języku Python

W Pythonie nie deklaruje się zmiennych. Zamiast tego, aby utworzyć zmienną, należy skorzystać z instrukcji przypisania. Oto przykład takiej instrukcji:

```
age = 25
```

Po wykonaniu tej instrukcji zostanie utworzona zmienna o nazwie `age` i zostanie jej przypisana wartość całkowita równa 25. A oto inny przykład:

```
title = 'Wiceprezydent'
```

Po wykonaniu tego polecenia zostanie utworzona zmienna o nazwie `title`, do której zostanie przypisany ciąg znaków `'Wiceprezydent'`.

## Nazwy zmiennych w języku Python

W Pythonie możesz wybrać własne nazwy zmiennych, o ile nie użyjesz żadnego ze słów kluczowych tego języka. (Pełna lista słów kluczowych języka Python jest zawarta w tabeli 2.1 w *Python Language Companion*.) Podczas nazywania zmiennych w Pythonie trzeba dodatkowo przestrzegać następujących reguł:

- Nazwa zmiennej nie może zawierać spacji.
- Pierwszy znak nazwy zmiennej musi być małą lub dużą literą(a – z, A – Z) albo znakiem podkreślenia (\_).
- Po pierwszym znaku można używać małych lub dużych liter (a – z, A – Z), cyfr (0 – 9) albo znaku podkreślenia.
- Rozróżniane są wielkie i małe litery. Oznacza to, że nazwa zmiennej `ItemsOrdered` nie jest taka sama jak nazwa `itemsordered`.

## Wyświetlanie wielu elementów za pomocą funkcji print w języku Python

Do funkcji `print` można przekazać wiele argumentów, a Python wypisze na ekranie wartości wszystkich argumentów, oddzielając je spacją. Na listingu 2.23 przedstawiam przykład takiego użycia funkcji `print`.

### **Listing 2.23. (print\_multiple.py)**

```
1 room = 503
2 print('Mieszkam w pokoju numer' , room)
```

### **Wynik działania programu**

```
Mieszkam w pokoju numer 503
```

Instrukcja w wierszu 1. tworzy zmienną o nazwie `room` i przypisuje jej wartość całkowitą 503. Natomiast instrukcja w wierszu 2. wyświetla dwie pozycje: literal ciągu znaków, po którym następuje wartość zmiennej `room`. Zauważ, że Python自动ycznie umieścił spację między tymi dwoma elementami.

## Odczyt danych wejściowych z klawiatury w języku Python

W Pythonie istnieje wbudowana funkcja `input`, która służy do odczytu danych wejściowych z klawiatury. Funkcja ta odczytuje fragment danych wprowadzony z klawiatury i zwraca go z powrotem do programu w postaci ciągu znaków. Zwykle używamy funkcji `input` w instrukcji przypisania, jak w poniższym ogólnym formacie:

```
zmienna = input(zachęta)
```

W tym ogólnym formacie słowo `zachęta` jest ciągiem znaków wyświetlonym na ekranie. Celem tego ciągu znaków jest poinstruowanie użytkownika, żeby wprowadził jakąś wartość. Słowo `zmienna` to nazwa zmiennej, która będzie przechowywać dane wprowadzone z klawiatury. Oto przykład instrukcji, która wykorzystuje funkcję `input` do odczytu danych z klawiatury:

```
name = input('Jak się nazywasz? ')
```

Oto co dzieje się w ramach tego polecenia:

- Na ekranie wyświetlany jest ciąg znaków: 'Jak się nazywasz?'.
- Następnie program zatrzymuje się i oczekuje, aż użytkownik wpisze coś na klawiaturze, po czym naciśnie klawisz *Enter*.
- Po naciśnięciu klawisza *Enter* dane, które zostały wpisane, są zwracane jako ciąg znaków i przypisywane do zmiennej *name*.

Na listingu 2.24 został pokazany kompletny program, który korzysta z funkcji *input* w celu odczytania dwóch ciągów znaków jako danych wejściowych z klawiatury.

#### **Listing 2.24. (string\_input.py)**

```
1 first_name = input('Wprowadź swoje imię: ')
2 last_name = input('Wprowadź swoje nazwisko: ')
3 print(Witaj, first_name, last_name)
```

#### **Wynik działania programu (wpisywane dane z klawiatury zaznaczono pogrubioną czcionką)**

Wprowadź swoje imię: **Jaś** [Enter]  
 Wprowadź swoje nazwisko: **Fasola** [Enter]  
 Witaj, Jaś Fasola

#### **Odczytywanie liczb za pomocą funkcji *input* w języku Python**

Funkcja *input* zawsze zwraca dane wejściowe użytkownika jako ciąg znaków, nawet jeśli wprowadzi on dane numeryczne. Założmy, że wywołujesz funkcję *input*, a użytkownik wpisuje liczbę 72 i naciska klawisz *Enter*. Wartością zwracaną przez funkcję wejściową jest ciąg znaków '72'. Może to stanowić problem w momencie, gdy chcesz użyć wartości w operacji matematycznej. Operacje matematyczne można wykonywać tylko na wartościach numerycznych, a nie na ciągach znaków.

Na szczęście Python ma wbudowane funkcje, których można użyć do przekonwertowania ciągu znaków na typ numeryczny. W tabeli 2.7 przedstawiam dwie z tych funkcji.

**Tabela 2.7.** Funkcje konwersji danych w języku Python

Funkcja	Opis
<i>int(element)</i>	Funkcja <i>int()</i> zwraca wartość argumentu skonwertowaną na liczbę całkowitą.
<i>float(element)</i>	Funkcja <i>float()</i> zwraca wartość argumentu skonwertowaną na liczbę zmiennoprzecinkową.

Na przykład ten kod prosi użytkownika o wpisanie liczby całkowitej i liczby zmiennej oprzecinkowej:

```
hours = int(input('Ile godzin pracowałeś?'))
pay_rate = float(input('Jaka jest twoja stawka godzinowa?'))
```

## Wykonywanie obliczeń w języku Python

Operatory arytmetyczne Pythona są prawie takie same jak te przedstawione wcześniej w tabeli 2.1 w tym rozdziale:

---

+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Dzielenie modulo
**	Potęgowanie

---

Oto kilka przykładów instrukcji Pythona, które używają operatora arytmetycznego do obliczenia wartości i przypisują tę wartość do zmiennej:

```
total = price + tax
sale = price - discount
population = population * 2
half = number / 2
leftOver = 17 % 3
result = 4**2
```

Na listingu 2.25 pokazano przykładowy program, który wykonuje obliczenia matematyczne. (Ten program jest wersją pseudokodu z listingu 2.8 napisaną dla języka Python).

### **Listing 2.25. (sale\_price.py)**

```
1 original_price = float(input("Wprowadź podstawową cenę przedmiotu: "))
2 discount = original_price * 0.2
3 sale_price = original_price - discount
4 print('Cena sprzedaży wynosi', sale_price)
```

### **Wynik działania programu (wpisywane dane z klawiatury zaznaczono pogrubioną czcionką)**

Wprowadź podstawową cenę przedmiotu: **100.00** [Enter]

Cena sprzedaży wynosi **80.0**

W Pythonie kolejność wykonywania operacji i sposób użycia nawiasów jako symboli grupowania są takie same jak opisane wcześniej w tym rozdziale.

## Komentarze w języku Python

Aby napisać komentarz wierszowy w Pythonie, wystarczy umieścić symbol # w miejscu, w którym ma się on zacząć. Interpreter Pythona zignoruje wszystko od tego momentu aż do końca wiersza. A oto przykład:

```
# Ten program oblicza wynagrodzenie brutto pracownika
```

## C++

### Dane wejściowe, przetwarzanie i dane wyjściowe w języku C++

#### Części programu w języku C++

Typowy program napisany w C++ zawiera następujący kod:

```
#include <iostream>
using namespace std;

int main()
{
    return 0;
}
```

Powyższy przykład można uznać za podstawową formę programu. I rzeczywiście ten program nie robi absolutnie nic. Można jednak dodać do niego dodatkowy kod, aby mógł wykonać jakieś operacje. Podczas pisania swoich pierwszych programów w języku C++ zauważysz, że dodatkowe instrukcje trzeba umieszczać w nawiasach klamrowych, tak jak pokazano na rysunku 2.21.

```
#include <iostream>
using namespace std;

int main()
{
    ← Dodatkowe instrukcje C++ umieszczaj w tym obszarze
    return 0;
}
```

**Rysunek 2.21.** Podstawowy program w języku C++

#### Średniki w języku C++

W języku C++ kompletna instrukcja kończy się średnikiem. Tak jak w języku polskim zdanie kończy się kropką, tak w języku C++ średnik oznacza koniec instrukcji. Zauważ jednak, że niektóre wiersze kodu w naszych przykładowych programach wcale nie kończą się średnikiem. To dlatego, że nie każdy wiersz kodu jest instrukcją. Jeśli to jest mylące, nie rozpaczaj! Tworząc kolejne programy w C++, intuicyjne zaczniesz zauważać różnicę między instrukcjami a wierszami kodu niebędącymi instrukcjami.

#### Wyświetlanie danych na ekranie w języku C++

Aby wyświetlić na ekranie dane wyjściowe w C++, należy napisać instrukcję cout (wymawianasiazut). Instrukcja cout rozpoczyna się od słowa cout, po którym następuje operator <<, a następnie wprowadzane są dane, które mają zostać wyświetlone. Instrukcja kończy się średnikiem. Na listingu 2.26 prezentuję przykład takiej instrukcji. (Pamiętaj, że numery wierszy NIE są częścią programu! Pokazane są tylko w celach informacyjnych. Nie wpisuj numerów wierszy podczas wprowadzania kodu programu).

**Listing 2.26.** (DisplayOutput.cpp)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Witaj, świecie!";
7     return 0;
8 }
```

**Wynik działania programu**

Witaj, świecie!

Operator `<<` to tak zwany **operator wstawiania do strumienia**. Zawsze pojawia się on po lewej stronie elementu danych, który chcemy wyświetlić. Zauważ, że w wierszu 6. operator `<<` pojawia się po lewej stronie ciągu znaków "Witaj, świecie!". Po uruchomieniu programu wyświetlany jest napis: *Witaj, świecie!*.

Z pomocą pojedynczej instrukcji `cout` można wyświetlić wiele elementów, pod warunkiem że operator `<<` pojawi się po lewej stronie każdego z nich. Na listingu 2.27 przedstawiam przykład takiego użycia operatora. W wierszu 6. wyświetlane są trzy elementy danych: ciąg znaków "Programowanie", ciąg znaków "jest" oraz ciąg znaków "zabawne". Zauważ, że operator `<<` pojawia się po lewej stronie każdego z tych elementów.

**Listing 2.27.** (DisplayMultipleItems.cpp)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Programowanie " << "jest " << "zabawne.";
7     return 0;
8 }
```

**Wynik działania programu**

Programowanie jest zabawne.

Gdy wyświetlasz dane za pomocą instrukcji `cout`, musisz pamiętać, że wszystkie elementy wyświetlane będą jako ciągły blok tekstu, taki jak na przykład na listing 2.28. Mimo że program ma trzy instrukcje `cout`, jego wynik pojawia się tylko w jednym wierszu.

**Listing 2.28.** (DisplayOneLine.cpp)

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
```

```

6   cout << "Programowanie ";
7   cout << "jest ";
8   cout << "zabawne.";
9   return 0;
10 }

```

**Wynik działania programu**

Programowanie jest zabawne.

Wyjściowy tekst jest wyświetlany jako jeden długi wiersz, ponieważ instrukcja cout nie przechodzi do nowego wiersza, chyba że otrzyma takie polecenie. Aby nakazać funkcji cout rozpoczęcie nowego wiersza, musisz użyć manipulatora endl. Na listingu 2.29 prezentuję przykład jego zastosowania.

**Listing 2.29. (EndlManipulator.cpp)**

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Programowanie" << endl;
7     cout << "jest" << endl;
8     cout << "zabawne." << endl;
9     return 0;
10 }

```

**Wynik działania programu**

Programowanie  
jest  
zabawne.

**Zmienne w języku C++**

W C++ zmienne muszą zostać zadeklarowane, zanim będą mogły być wykorzystane w programie. Deklaracja zmiennej jest zapisana w następującym ogólnym formacie:

*TypDanychNazwaZmiennej;*

W zapisie tym element *TypDanych* reprezentuje nazwę typu danych, natomiast element *NazwaZmiennej* stanowi nazwę zmiennej, którą deklarujemy. Instrukcja deklaracji kończy się średnikiem. Na przykład słowo kluczowe *int* jest nazwą typu danych liczb całkowitych, więc poniższa instrukcja deklaruje zmienną o nazwie *number*:

*int number;*

Tabela 2.8 zawiera listę niektórych typów danych języka C++, podaje ich wielkość w bajtach i opisuje rodzaj przechowywanych przez nie danych. Zauważ, że w niniejszej książce będziemy przede wszystkim używać typów danych *int*, *double* i *string*<sup>4</sup>.

<sup>4</sup> Aby użyć typu danych *string*, musisz na początku programu wpisać dyrektywę `#include <string>`. Gwoli ściśleści, w języku C++ *string* nie jest typem danych, lecz klasą. Używamy go jednak jako typu danych.

**Tabela 2.8.** Typy danych w języku C++

Typ danych	Co może przechowywać?
short	Liczby całkowite od -32 768 do +32 767
int	Liczby całkowite od -2 147 483 648 do +2 147 483 647
long	Liczby całkowite od -2 147 483 648 do +2 147 483 647
float	Liczby zmiennoprzecinkowe od $\pm 3,4 \times 10^{-38}$ do $\pm 3,4 \times 10^{38}$ , z dokładnością do 7 cyfr po przecinku
double	Liczby zmiennoprzecinkowe od $\pm 1,7 \times 10^{-308}$ do $\pm 1,7 \times 10^{308}$ , z dokładnością do 15 cyfr po przecinku
char	Może przechowywać liczby całkowite od -128 do +127, ale zazwyczaj używany do przechowywania znaków
string	Ciąg znaków
bool	Wartości true lub false

Oto kilka innych przykładów deklaracji zmiennych:

```
int speed;
double distance;
String name;
```

Kilka zmiennych tego samego typu danych można zadeklarować w ramach jednej instrukcji deklaracji. Na przykład ta instrukcja deklaruje trzy zmienne typu int o nazwie width, height i length:

```
int width, height, length;
```

Podczas deklarowania zmiennych można też inicjalizować je różnymi wartościami początkowymi. Ta instrukcja deklaruje zmienną int o nazwie hours i inicjalizuje ją wartością początkową równą 40:

```
int hours = 40;
```

### Nazwy zmiennych w języku C++

W C++ możesz wybrać dowolne nazwy dla zmiennych, o ile nie użyjesz żadnego ze słów kluczowych tego języka. (Aby uzyskać pełną listę słów kluczowych w C++, spójrz na tabelę 1.1 w *C++ Language Companion*). Słowa kluczowe tworzą rdzeń języka i każde z nich ma określone zadanie. Oto kilka dodatkowych reguł, których należy przestrzegać przy nazywaniu zmiennych:

- Pierwszy znak musi być małą lub dużą literą (a – z, A – Z) albo znakiem podkreślenia (\_).
- Po pierwszym znaku można używać małych lub dużych liter (a – z, A – Z), cyfr (0 – 9) albo znaku podkreślenia (\_).
- Rozróżniane są duże i małe litery. Oznacza to, że nazwa itemsOrdered nie jest taka sama jak nazwa itemsordered.
- Nazwy zmiennych nie mogą zawierać spacji.

Na listingu 2.30 prezentuję przykład z trzema deklaracjami zmiennych. Zauważ, że ze względu na to, iż używamy zmiennej typu `string`, w wierszu 2. umieszczona została dyrektywa `#include <string>`. Natomiast w wierszu 7. deklarowana jest zmienna typu `string` o nazwie `name` zainicjalizowana ciągiem znaków „Jaś Fasola”. Dodatkowo w wierszu 8. deklarowana jest zmienna typu `int` o nazwie `hours` zainicjalizowana wartością 40. Z kolei w wierszu 9. deklarowana jest zmienna typu `double` o nazwie `pay` zainicjalizowana wartością 852,99. Zauważ, że w wierszach od 11. do 13. używamy funkcji `cout` do wyświetlania zawartości każdej ze zmiennych.

### **Listing 2.30. (Variables.cpp)**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main()
6 {
7     string name = "Jaś Fasola";
8     int hours = 40;
9     double pay = 852.99;
10
11    cout << name << endl;
12    cout << hours << endl;
13    cout << pay << endl;
14
15 }
```

### **Wynik działania programu**

```

Jaś Fasola
40
852.99
```

### **Odczytywanie danych wejściowych z klawiatury w języku C++**

W C++, aby odczytać dane wprowadzane z klawiatury, należy wprowadzić instrukcję `cin` (wymawianą *cin*). Instrukcja `cin` zaczyna się od słowa kluczowego `cin`, po którym następuje operator `>>`, a po nim nazwa zmiennej. Instrukcja kończy się średnikiem. Po wykonaniu tej instrukcji program zaczeka, aż użytkownik wprowadzi dane wejściowe na klawiaturze i naciśnie klawisz *Enter*. W momencie, gdy użytkownik naciśnie ten klawisz, dane wejściowe zostaną przypisane do zmiennej, która jest wyświetlana za operatorem `>>`. (Operator `>>` znany jest jako operator ekstrakcji ze strumienia). Na listingu 2.31 prezentuję przykład użycia tego operatora.

### **Listing 2.31. (KeyboardInput1.cpp)**

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int age;
7
8     cout << "Ile masz lat?" << endl;
```

```

9     cin >> age;
10    cout << "Oto wprowadzona wartość:" << endl;
11    cout << age;
12    return 0;
13 }

```

**Wynik działania programu (wpisywane dane z klawiatury zaznaczono pogrubioną czcionką)**

Ile masz lat?

**24 [Enter]**

Oto wprowadzona wartość:

**24**

Program pokazany na listingu 2.32 wykorzystuje instrukcję `cin` do odczytywania ciągu znaków, liczby całkowitej i liczby zmiennoprzecinkowej.

**Listing 2.32. (KeyboardInput2.cpp)**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main()
6 {
7     string name;
8     double payRate;
9     int hours;
10
11    cout << "Wprowadź swoje imię." << endl;
12    cin >> name;
13    cout << "Wprowadź swoją stawkę godzinową." << endl;
14    cin >> payRate;
15    cout << "Wprowadź liczbę przepracowanych godzin." << endl;
16    cin >> hours;
17
18    cout << "Oto podane wartości:" << endl;
19    cout << name << endl;
20    cout << payRate << endl;
21    cout << hours << endl;
22
23    return 0;
24 }

```

**Wynik działania programu (wpisywane dane z klawiatury zaznaczono pogrubioną czcionką)**

Wprowadź swoje imię.

**Janina [Enter]**

Wprowadź swoją stawkę godzinową.

**55.25 [Enter]**

Wprowadź liczbę przepracowanych godzin.

**40 [Enter]**

Oto podane wartości:

Janina

55.25

40

### Przypadek specjalny: odczytywanie ciągów znaków zawierających spacje

Instrukcja `cin` może odczytywać ciąg znaków składający się z pojedynczego słowa, jak pokazano wcześniej na listingu 2.32. Jednak gdy dane wejściowe użytkownika są ciągiem zawierającym wiele słów oddzielonych spacjami, to nie zachowuje się już tak, jak można byłoby się spodziewać. Jeśli chcesz wprowadzić ciąg znaków zawierający wiele słów, musisz użyć funkcji `getline`. Funkcja ta odczytuje cały wiersz danych wejściowych, razem z wszystkimi spacjami, i zapisuje je w zmiennej typu `string`. Używa się jej tak, jak pokazałem na poniższym przykładzie, w którym `inputLine` jest nazwą zmiennej typu `string`, odbierającej dane wejściowe:

```
getline(cin, inputLine);
```

Na listingu 2.33 przedstawiam przykład wykorzystania funkcji `getline`.

#### **Listing 2.33. (KeyboardInput3.cpp)**

```
1 // Ten program demonstruje użycie funkcji getline
2 // do odczytu danych wejściowych do zmiennej string
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string name;
10    string city;
11
12    cout << "Jak się nazywasz?" << endl;
13    getline(cin, name);
14
15    cout << "Wprowadź miasto, w którym mieszkasz." << endl;
16    getline(cin, city);
17
18    cout << "Witaj, " << name << endl;
19    cout << "Mieszkasz w miejscowości " << city << endl;
20    return 0;
21 }
```

#### **Wynik działania programu (wpisywane dane z klawiatury zaznaczono pogrubioną czcionką)**

Jak się nazywasz?

**Madzia Kowalska** [Enter]

Wprowadź miasto, w którym mieszkasz.

**Łódź** [Enter]

Witaj, Madzia Kowalska

Mieszkasz w miejscowości **Łódź**

### Wykonywanie obliczeń w języku C++

Oto kilka przykładów instrukcji języka C++, które używają operatora arytmetycznego do obliczenia wartości i przypisują tę wartość do zmiennej:

```
total = price + tax;
sale = price - discount;
population = population * 2;
half = number / 2;
leftOver = 17 % 3;
```

Operatory arytmetyczne w C++ są prawie takie same jak te przedstawione w tabeli 2.1.

---

+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Dzielenie modulo

---

W C++ nie ma operatora potęgowania, ale dostępna jest funkcja o nazwie `pow`. Oto przykład użycia tej funkcji:

```
result = pow(4.0, 2.0);
```

Funkcja przyjmuje dwa argumenty typu `double` (są to liczby podane w nawiasach). Podnosi wartość pierwszego argumentu do potęgi wartości drugiego argumentu i zwraca wynik jako wartość typu `double`. W tym przykładzie wartość 4.0 jest podnoszona do potęgi 2.0. Powyższa instrukcja jest zatem równoważna z następującym wyrażeniem algebraicznym:

$$\text{wynik} = 4^2$$

### Stałe nazwane w języku C++

W C++ można tworzyć stałe nazwane za pomocą słowa kluczowego `const` umieszczonego w deklaracji zmiennej. Słowo to jest umieszczane tuż przed typem danych. Oto przykład:

```
const double INTEREST_RATE = 0.069;
```

Nie ma wymogu, aby nazwy zmiennej były zapisywane wielkimi literami, ale wielu programistów zapisuje je w ten sposób, aby można było łatwiej odróżnić nazwy stałych od nazw zwykłych zmiennych.

Podczas deklarowania zmiennej za pomocą modyfikatora `const` musi być podana wartość inicjalizująca, w przeciwnym wypadku wystąpi błąd podczas komplikacji programu. Błąd kompilatora pojawia się również wtedy, gdy program zawiera instrukcje, które próbują zmienić wartość takiej stałej.

### Tworzenie dokumentacji programu za pomocą komentarzy w języku C++

Aby napisać komentarz wierszowy w C++, wystarczy umieścić dwa ukośniki (//) w miejscu, w którym ma się on rozpoczęć. Kompilator zignoruje wszystko od tego miejsca aż do końca wiersza. Oto przykład:

```
// Ten program oblicza wynagrodzenie brutto pracownika
```

Komentarze wielowierszowe lub komentarze blokowe rozpoczynają się od znaków /\* (ukośnik, a następnie gwiazdka) i kończą znakami \*/ (gwiazdka z następującym po niej ukośnikiem). Wszystko pomiędzy tymi znacznikami będzie ignorowane przez kompilator. Oto przykład:

```
/*
Ten program oblicza wynagrodzenie brutto pracownika,
w tym wynagrodzenie za wszystkie przepracowane nadgodziny
*/
```

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Błąd \_\_\_\_\_ umożliwia uruchomienie programu, ale może spowodować, że program będzie zwracał nieprzewidywalne wyniki.
  - a) składni
  - b) sprzętu
  - c) logiczny
  - d) ostateczny
2. \_\_\_\_\_ to pojedyncza funkcja, jaką musi wykonywać program, aby spełnić oczekiwania klienta.
  - a) zadanie
  - b) wymaganie
  - c) warunek wstępny
  - d) predykat
4. \_\_\_\_\_ składa się ze ścisłe zdefiniowanych kroków, które należy wykonać, aby osiągnąć określony cel.
  - a) logarytm
  - b) plan działania
  - c) plan logiczny
  - d) algorytm
3. Nieformalny język, który nie ma własnej składni i którego nie da się skompilować ani uruchomić, nazywamy \_\_\_\_\_.
  - a) sztucznym kodem
  - b) pseudokodem
  - c) Javą
  - d) schematem blokowym
4. \_\_\_\_\_ to diagram, w którym w sposób graficzny przedstawia się kolejne kroki działania programu.
  - a) schemat blokowy
  - b) wykres krokowy
  - c) graf kodu
  - d) graf programu
5. \_\_\_\_\_ to zbiór instrukcji wykonywanych przez program w takiej kolejności, w jakiej zostały zapisane w programie.
  - a) program szeregowy
  - b) posortowany kod
  - c) struktura sekwencyjna
  - d) struktura uszeregowana

6. \_\_\_\_\_ to dane w formie sekwencji znaków.  
a) struktura sekwencyjna  
b) zbiór znaków  
c) ciąg znaków  
d) blok tekstu
7. \_\_\_\_\_ to miejsce w pamięci komputera, do którego można się odwołać za pomocą nazwy.  
a) zmienna  
b) rejestr  
c) gniazdo RAM  
d) bajt
8. \_\_\_\_\_ to potencjalna osoba, która będzie korzystała z programu i wprowadzała do niego dane.  
a) projektant  
b) użytkownik  
c) świnia morska  
d) przedmiot testu
9. \_\_\_\_\_ służy do poinformowania użytkownika, że program oczekuje na wprowadzenie danych.  
a) zapytanie  
b) instrukcja wejściowa  
c) dyrektywa  
d) komunikat dla użytkownika
10. \_\_\_\_\_ zapisuje w zmiennej określona wartość.  
a) deklaracja zmiennej  
b) instrukcja przypisania  
c) wyrażenie matematyczne  
d) literal ciągu znaków
11. W wyrażeniu  $12 + 7$  wartości, które pojawiają się po lewej i po prawej stronie symbolu  $+$ , nazywamy \_\_\_\_\_.  
a) operandami  
b) operatorami  
c) argumentami  
d) wyrażeniem matematycznym
12. Operator \_\_\_\_\_ służy do podnoszenia liczby do potęgi.  
a) modulo  
b) mnożenia  
c) potęgowania  
d) operand
13. Operator \_\_\_\_\_ wykonuje dzielenie, ale zamiast ilorazu zwraca resztę z dzielenia.  
a) modulo  
b) mnożenia  
c) potęgowania  
d) operand

14. Podczas \_\_\_\_\_ wskazujemy nazwę zmiennej i typ danych.
  - a) przypisania
  - b)
  - c) specyfikacji zmiennej certyfikacji zmiennej
  - d) deklaracji zmiennej
15. Przypisanie wartości do zmiennej podczas jej deklarowania nazywamy \_\_\_\_\_.
  - a) alokacją
  - b) inicjalizacją
  - c) certyfikacją
  - d) stylem programowania
16. Zmienna \_\_\_\_\_ to taka zmienna, która została zadeklarowana, ale nie została do niej jeszcze przypisana wartość.
  - a) niezdefiniowana
  - b) niezainicjalizowana
  - c) pusta
  - d) domyślna
17. \_\_\_\_\_ to rodzaj zmiennej, do której została przypisana wartość, której nie można zmienić podczas działania programu.
  - a) zmienna statyczna
  - b) zmienna niezainicjalizowana
  - c) stała nazwana
  - d) zmienna zablokowana
18. Technika wykrywania błędów w programie, w której musisz się postawić w roli komputera wykonującego kolejne instrukcje w programie, nazywa się \_\_\_\_\_.
  - a) obliczeniami wyobrażonymi
  - b) grą fabularną
  - c) simulacją umysłu
  - d) ręcznym śledzeniem programu
19. Krótki opis umieszczony w różnych miejscach programu wskazujący, jak działa dany fragment kodu, nazywamy \_\_\_\_\_.
  - a) komentarzem
  - b) instrukcją użytkownika
  - c) samouczkiem
  - d) dokumentacją zewnętrzną

### **Prawda czy fałsz?**

1. Podczas pisania programu w pseudokodzie programista musi uważać, aby nie popełnić błędu składni.
2. Operacja mnożenia i dzielenia w wyrażeniu matematycznym ma miejsce przed dodawaniem i odejmowaniem.
3. W nazwie zmiennej można umieścić znak spacji.

4. W przypadku większości języków programowania pierwszym znakiem w nazwie zmiennej nie może być liczba.
5. Nazwa zmiennej `gross_Pay` to przykład zapisu camelCase.
6. W językach programowania wymagających deklarowania zmiennych deklaracja musi mieć miejsce, zanim dana zmienna zostanie wykorzystana w innym poleceniu.
7. Niezainicjalizowane zmienne są powodem licznych błędów.
8. Nie można zmienić nazwy stałej nazwanej.
9. Ręczne śledzenie kodu polega na ręcznym przetłumaczeniu programu na język maszynowy.
10. Dokumentacja wewnętrzna odnosi się do podręczników i instrukcji obsługi wyjaśniających działanie programu i jest używana tylko w dziale IT przedsiębiorstwa.

### Krótką odpowiedź

1. Od czego powinien zacząć pracę zawodowy programista, przystępując do nowego zadania?
2. Co to jest pseudokod?
3. Jakie trzy operacje wykonuje zazwyczaj program komputerowy?
4. Co oznacza określenie „przyjazny dla użytkownika”?
5. Wymień dwie rzeczy, które musisz wskazać, deklarując zmienną.
6. Jaka wartość jest zapisana w niezainicjalizowanej zmiennej?

### Warsztat projektanta algorytmów

1. Zaprojektuj algorytm, który będzie prosił użytkownika o wprowadzenie wzrostu i zapisze tę informację w zmiennej `height`.
2. Zaprojektuj algorytm, który będzie prosił użytkownika o wprowadzenie ulubionego koloru i zapisze tę informację w zmiennej `color`.
3. Napisz instrukcję przypisania z wykorzystaniem zmiennych `a` i `b`, która będzie wykonywała następujące czynności:
  - a) doda 2 do zmiennej `a` i zapisze wynik w zmiennej `b`;
  - b) pomnoży `b` przez 4 i zapisze wynik w zmiennej `a`;
  - c) podzieli `a` przez 3,14 i zapisze wynik w zmiennej `b`;
  - d) odejmie 8 od `b` i zapisze wynik w zmiennej `a`.
4. Założmy, że zmienne `result`, `x`, `y` i `z` są typu całkowitego oraz że  $x = 4$ ,  $y = 8$ , a  $z = 2$ . Jaka wartość zostanie przypisana do zmiennej `result` po wykonaniu poniższych poleceń?
  - a) Set `result` = `x` + `y`
  - b) Set `result` = `z` \* 2
  - c) Set `result` = `y` / `x`
  - d) Set `result` = `y` - `z`

5. Napisz w pseudokodzie instrukcję, która zadeklaruje zmienną o nazwie
6. cost i będzie liczbą rzeczywistą. Napisz w pseudokodzie instrukcję, która zadeklaruje zmienną o nazwie total i będzie liczbą całkowitą. Zainicjalizuj zmienną wartością równą 0.
7. Napisz w pseudokodzie instrukcję, która przypisze do zmiennej count wartość równą 27.
8. Napisz w pseudokodzie instrukcję, która przypisze do zmiennej total wartość równą sumie liczb 10 i 14.
9. Napisz w pseudokodzie instrukcję, która odejmie od zmiennej total wartość przypisaną do zmiennej downPayment i przypisze wynik do zmiennej due.
10. Napisz w pseudokodzie instrukcję, która pomnoży zmienną subtotal przez 0,15 i przypisze wynik do zmiennej totalFee.
11. Gdyby poniższy pseudokod był prawdziwym programem, jaki wynik wyświetliłby się na ekranie?

```

Declare Integer a = 5
Declare Integer b = 2
Declare Integer c = 3
Declare Integer result
Set result = a + b + c
Display result

```

12. Gdyby poniższy pseudokod był prawdziwym programem, jaki wynik wyświetliłby się na ekranie?

```

Declare Integer num = 99
Set num = 5
Display num

```

## Ćwiczenia z wykrywania błędów

1. Gdyby poniższy pseudokod był prawdziwym programem, dlaczego program nie zadziałałby zgodnie z oczekiwaniami programisty?

```

Declare String favoriteFood

Display "Jaka jest Twoja ulubiona potrawa?"
Input favoriteFood

Display "Twoja ulubiona potrawa to "
Display "favoriteFood"

```

2. Po przetłumaczeniu poniższego pseudokodu na prawdziwy język programowania w programie pojawi się najprawdopodobniej błąd składni. Czy wiesz dlaczego?

```

Declare String 1stPrize

Display "Wprowadź nagrodę za pierwsze miejsce."
Input 1stPrize

Display "Zwycięzca otrzyma jako nagrodę ", 1stPrize

```

3. Poniższy kod nie wyświetli na ekranie takiego wyniku, jakiego spodziewa się programista. Czy wiesz, dlaczego?

```
Declare Real lowest, highest, average
```

```
Display "Wprowadź najniższy wynik."
```

```
Input lowest
```

```
Display "Wprowadź najwyższy wynik."
```

```
Input highest
```

```
Set average = low + high / 2
```

```
Display "Średni wynik to: " , average, ". "
```

4. Znajdź błąd w następującym pseudokodzie:

```
Display "Wprowadź długość pokoju."
```

```
Input length
```

```
Declare Integer length
```

5. Znajdź błąd w następującym pseudokodzie:

```
Declare Integer value1, value2, value3, sum
```

```
Set sum = value1 + value2 + value3
```

```
Display "Wprowadź pierwszą wartość."
```

```
Input value1
```

```
Display "Wprowadź drugą wartość."
```

```
Input value2
```

```
Display "Wprowadź trzecią wartość."
```

```
Input value3
```

```
Display "Suma jest równa ", sum
```

6. Znajdź błąd w następującym pseudokodzie:

```
Declare Real pi
```

```
Set 3.14159265 = pi
```

```
Display "Pi jest równe ", pi
```

7. Znajdź błąd w następującym pseudokodzie:

```
Constant Real GRAVITY = 9.81
```

```
Display "Wartości przyspieszenia spadających obiektów wynoszą:"
```

```
Display "Na Ziemi: ", GRAVITY, " m/s2."
```

```
Set GRAVITY = 1.63
```

```
Display "Na Księżyku: ", GRAVITY, " m/s2."
```

## Ćwiczenia programistyczne

1. Dane osobowe

Zaprojektuj program, który będzie wyświetlał następujące informacje:

- Twoje imię i nazwisko
- Twój adres
- Twój numer telefonu
- Twój kierunek studiów

## 2. Prognozowanie sprzedaży

Pewna firma stwierdziła, że jej roczny zysk ze sprzedaży wynosi 23% od wartości sprzedaży. Zaprojektuj program, który umożliwi użytkownikowi wpisanie przewidywanej wartości sprzedaży i wyświetli zysk z tej sprzedaży.

*Podpowiedź:* Aby zapisać w programie 23%, użyj liczby 0,23.

## 3. Obliczanie powierzchni pola

Jeden akr to 43530 stóp kwadratowych. Zaprojektuj program, który pozwoli użytkownikowi wprowadzić powierzchnię pola w stopach kwadratowych i obliczy powierzchnię tego pola w akrach.

*Podpowiedź:* Aby uzyskać liczbę akrów, należy podzielić wprowadzoną przez użytkownika liczbę przez 43560.

## 4. Suma wartości zakupów

Klient kupił w sklepie pięć artykułów. Zaprojektuj program, który poprosi użytkownika o wprowadzenie ceny netto każdego artykułu, a następnie wyświetli wartość netto, wartość podatku i wartość brutto zakupów. Założymy, że podatek wynosi 6%.

## 5. Przebyta odległość

Odległość, jaką przejedzie samochód na autostradzie (zakładając, że nie będzie po drodze żadnych wypadków i opóźnień), można obliczyć według następującego wzoru:

$$\text{odległość} = \text{prędkość} \cdot \text{czas}$$

Samochód jedzie z prędkością 130 km/h. Zaprojektuj program, który będzie wyświetlał następujące informacje:

- odległość, jaką przebędzie samochód w ciągu 5 godzin;
- odległość, jaką przebędzie samochód w ciągu 8 godzin;
- odległość, jaką przebędzie samochód w ciągu 12 godzin;

## 6. Podatek

Zaprojektuj program, który poprosi użytkownika o wprowadzenie wartości zakupów. W Stanach Zjednoczonych podatek naliczany jest zarówno w stanie, jaki i w hrabstwie. Projektowany program powinien obliczyć wartość obu tych podatków. Założymy, że podatek w stanie wynosi 4%, a podatek w hrabstwie wynosi 2%. Program powinien wyświetlić wartość zakupów, wartość podatku w stanie, wartość podatku w hrabstwie, sumę obu tych podatków i sumę wszystkich tych wartości.

*Podpowiedź:* Aby zapisać w programie 2%, użyj liczby 0,02, a aby zapisać 4%, użyj liczby 0,04.

## 7. Zużycie paliwa

Zużycie paliwa przez samochód można obliczyć za pomocą następującego wzoru:

$$KNL = \text{przejechane kilometry} : \text{litry zużytego paliwa}$$

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby przejechanych kilometrów i liczby litrów zużytego paliwa. Program powinien obliczyć zużycie paliwa samochodu i wyświetlić wynik na ekranie.

## 8. Napiwek, podatek i suma

Zaprojektuj program, który będzie obliczał całkowity koszt obiadu w restauracji. Program powinien zapytać użytkownika o koszt posiłku, a następnie powinien obliczyć wysokość napiwku w wysokości 15% tego kosztu i wartość podatku w wysokości 7%. Wyświetl na ekranie wartości napiwku, podatku i sumy.

## 9. Utrata wagi

Jeśli umiarkowanie aktywna osoba zmniejsza ilość przyjmowanych kalorii o 500 dziennie, może stracić około 2 kilogramów miesięcznie. Napisz program, w którym użytkownik ma wprowadzić swoją początkową wagę, a następnie program wyświetli tabelę pokazującą, jaka powinna być oczekiwania waga na koniec każdego miesiąca przez następne 6 miesięcy, o ile użytkownik pozostanie na tej diecie.

## 10. Płatności

Pewna osoba płaci co miesiąc określona kwotę jako opłatę za samochód. Zaprojektuj program, który poprosi użytkownika o podanie comiesięcznej kwoty i liczby miesięcy, w których była dokonywana wpłata. Program powinien wyświetlić całkowitą kwotę, którą zapłacił użytkownik.

## 11. Resztki pizzy

Urządzasz przyjęcie z pizzą i planujesz poczęstować każdego gościa trzema kawałkami. Zaprojektuj program wyświetlający liczbę kawałków, które pozostaną. Program powinien poprosić o następujące dane wejściowe:

- liczba posiadanych sztuk pizzy;
- liczba kawałków, z których składa się każda pizza;
- liczba gości.

Program powinien wyświetlać liczbę kawałków, które pozostaną po przyjęciu.

## 12. Zamiana stopni Celsjusza na stopnie Fahrenheita

Zaprojektuj program, który będzie zamieniał temperaturę podaną w stopniach Celsjusza na stopnie Fahrenheita. Wzór jest następujący:

$$F = \frac{9}{5}C + 32$$

Program powinien prosić użytkownika o wprowadzenie temperatury w stopniach Celsjusza, a następnie wyświetlić temperaturę w stopniach Fahrenheita.

## 13. Transakcje giełdowe

W zeszłym miesiącu Janek zakupił akcje spółki Acme Software Inc. Oto szczegóły tej transakcji:

- Janek kupił 10000 akcji;
- w momencie kupna zapłacił za każdą akcję 32,87 złotego;
- zapłacił brokerowi prowizję w wysokości 2% wartości zakupionych akcji.

Dwa tygodnie później Janek sprzedał akcje. Oto szczegóły tej transakcji:

- Janek sprzedał 1000 akcji;
- każdą akcję sprzedał za 33,92 złotego;

- zapłacił brokerowi kolejną prowizję w wysokości 2% wartości sprzedanych akcji.

Zaprojektuj program, który będzie wyświetlał następujące informacje:

- Ile Janek zapłacił za akcje?
- Jaką prowizję zapłacił Janek brokerowi w momencie kupna akcji?
- Za ile Janek sprzedał akcje?
- Jaką prowizję zapłacił Janek brokerowi w momencie sprzedaży akcji?
- Czy Janek zyskał, czy stracił na tej transakcji? Wyświetl kwotę, jaką zyskał lub stracił Janek po tym, jak sprzedał akcje i zapłacił obie prowizje.

#### **14. Ciasteczka i kalorie**

W jednym opakowaniu jest 40 ciasteczek. Na opakowaniu jest informacja, że w środku znajduje się 10 porcji, a jedna porcja to 300 kalorii. Zaprojektuj program, który będzie pytał użytkownika, ile ciasteczek zjadł, i wyświetli liczbę spożytych kalorii.

#### **15. Procentowy udział kobiet i mężczyzn**

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby kobiet i mężczyzn uczęszczających na zajęcia. Program powinien następnie wyświetlić, jaki jest procent kobiet i mężczyzn na zajęciach.

Podpowiedź: Założymy, że na zajęcia uczęszcza 8 mężczyzn i 12 kobiet. Razem jest więc 20 studentów. Procentowy udział mężczyzn możemy obliczyć jako  $8/20 = 0,4$ , czyli 40%. Procentowy udział kobiet możemy obliczyć jako  $12/20 = 0,6$ , czyli 60%.

#### **16. Dozownik składników**

Przepis na ciasteczka jest następujący:

- 1,5 szklanki cukru
- 1 szklanka masła
- 2,75 szklanki mąki

Dzięki temu przepisowi można upiec 48 ciasteczek. Zaprojektuj program, który będzie pytał użytkownika, ile ciasteczek chce upiec, a następnie wyświetli, ile szklanek każdego ze składników będzie potrzebował użytkownik, aby upiec wskazaną liczbę ciasteczek.

**TEMATYKA**

- |  |   |
|--|---|
| 3.1 Moduły — informacje wstępne        | 3.4 Przekazywanie argumentów do modułów   |
| 3.2 Definiowanie i wywoływanie modułów | 3.5 Zmienne globalne i stałe globalne     |
| 3.3 Zmienne lokalne                    | 3.6 Rzut oka na języki Java, Python i C++ |

**3.1****Moduły — informacje wstępne**

**WYJAŚNIENIE:** Moduł to zbiór instrukcji, który służy do wykonania określonego zadania w programie.

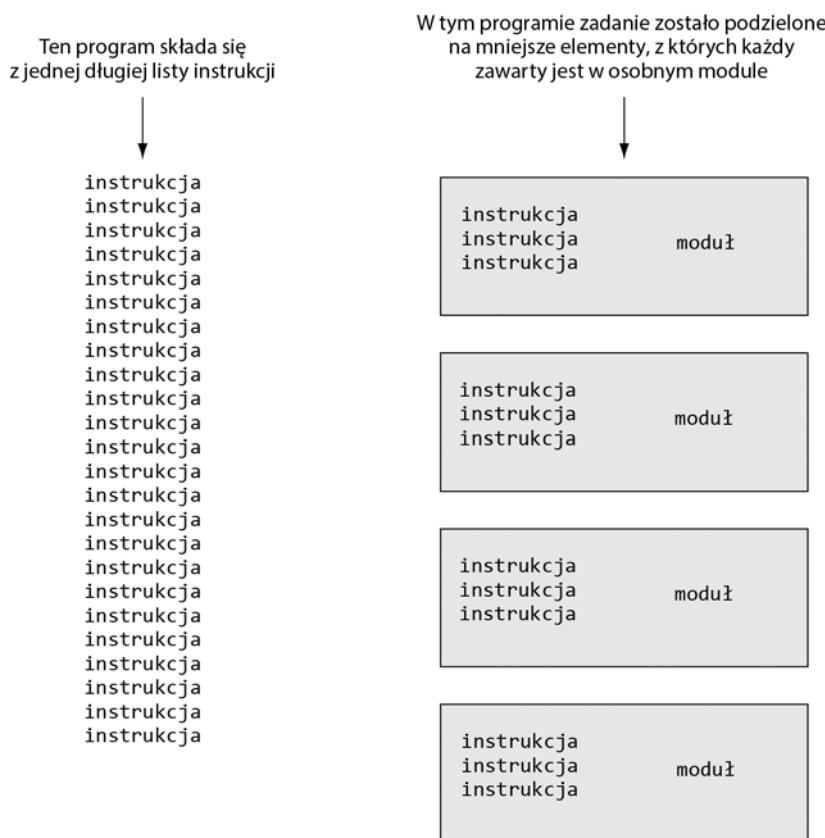
Z rozdziału 1. wiesz, że program to szereg instrukcji, które musi wykonać komputer, aby zrealizować określone zadanie. Następnie w rozdziale 2. przedstawiłem przykład programu, którego zadaniem było obliczenie wynagrodzenia pracownika. Dla przyponnienia: program mnożył liczbę przepracowanych godzin przez kwotę stawki godzinowej. Jednak prawdziwy program do obliczania wynagrodzenia powinien być znacznie bardziej rozbudowany. W rzeczywistej aplikacji obliczenie wynagrodzenia powinno się składać z kilku podzadań:

- pobranie stawki godzinowej;
- pobranie liczby przepracowanych godzin;
- obliczenie wynagrodzenia zasadniczego;
- obliczenie wynagrodzenia za nadgodziny;
- obliczenie potrąconych podatków i składek;
- obliczenie wynagrodzenia netto;
- wydrukowanie paska wynagrodzenia.

Większość programów wykonuje na tyle skomplikowane zadania, że dzieli się je na mniejsze podzadania. Z tego też powodu programiści decydują się podzielić program na moduły. **Moduł** to zbiór instrukcji, który służy do wykonania określonego zadania

w programie. Zamiast pisać jeden wielki program składający się z długiej listy instrukcji można podzielić go na kilka mniejszych modułów, z których każdy będzie odpowiedzialny za wykonanie jednego z fragmentów większego zadania. Następnie będzie można uruchomić poszczególne moduły w odpowiedniej kolejności tak, aby wykonały całe zadanie.

Takie podejście określa się często mianem „**dziel i zwyciężaj**”, ponieważ skomplikowane zadanie dzieli się na mniejsze i znacznie łatwiejsze do wykonania podzadania. Zilustrowałem tę koncepcję na rysunku 3.1, gdzie porównałem dwa programy: jeden, który składa się z długiej listy instrukcji, oraz drugi, w którym zadanie zostało podzielone na mniejsze elementy, umieszczone w osobnych modułach.



**Rysunek 3.1.** Wykorzystanie modułów w metodzie „dziel i zwyciężaj”

Korzystając z modułów, w każdym z nich umieszcza się zazwyczaj operacje wykonujące inne zadanie. Przykładowo prawdziwy program do obliczania wynagrodzenia pracownika będzie prawdopodobnie zawierał następujące moduły:

- moduł do pobierania stawki godzinowej;
  - moduł do pobierania liczby przepracowanych godzin;
  - moduł do obliczania wynagrodzenia zasadniczego;
  - moduł do obliczania liczby przepracowanych nadgodzin;

- moduł do obliczania potrąconych podatków i składek;
- moduł do obliczania wynagrodzenia netto;
- moduł do drukowania paska wynagrodzenia;

Niemal każdy współcześnie używany język programowania umożliwia tworzenie modułów, ale nie zawsze występują one w danym języku pod taką nazwą. Bardzo często nazywa się moduły **procedurami**, **podprocedurami**, **podprogramami**, **metodami** czy **funkcjami**. Funkcja to specjalny typ modułu, który omówię szerzej w rozdziale 6.

## Korzyści wynikające ze stosowania modułów

Podzielenie programu na moduły niesie pewne korzyści:

### Prostszy kod

Kod programu w module jest zazwyczaj prostszy i bardziej zrozumiały. Znacznie łatwiej jest odczytać kod w kilku małych modułach niż odczytać bardzo długą sekwencję instrukcji.

### Wielokrotne wykorzystanie kodu

Moduły ograniczają powielanie tego samego kodu w programie. Jeżeli określona operacja jest wykonywana w kilku miejscach programu, można ją ująć w osobnym module i uruchamiać moduł wtedy, gdy będzie taka konieczność. Takie podejście nazywa się **wielokrotnym wykorzystaniem kodu**, ponieważ kod tworzy się tylko raz, a można z niego korzystać wielokrotnie.

### Lepsze testowanie

Kiedy każde podzadanie zostanie umieszczone w osobnym module, znacznie łatwiej będzie testować i debugować program. Programista może przetestować każdy moduł z osobna, aby sprawdzić, czy działa on prawidłowo. Dzięki temu dużo łatwiejsze jest lokalizowanie i naprawianie błędów.

### Łatwiejsza konserwacja

Większość programów wymaga co pewien czas poprawienia wykrytych w nich błędów, polepszenia wydajności lub ułatwienia pracy użytkownikowi. Takie okresowe modyfikacje nazywa się **konserwacją oprogramowania**. Ponieważ program podzielony na moduły jest znacznie prostszy, mniejszy i bardziej zrozumiały, jego konserwacja także przebiega sprawniej niż w przypadku programu nie podzielnego na moduły.

### Szybsze tworzenie oprogramowania

Założmy, że programista lub zespół złożony z wielu programistów tworzy kilka różnych programów. Okazuje się, że niektóre z programów wykonują te same operacje — na przykład pytają użytkownika o login i hasło, wyświetlają aktualny czas itp. W takiej

sytuacji nie ma sensu wielokrotne tworzenie tego samego kodu w każdym programie. Można natomiast stworzyć moduły wykonujące te wspólne operacje i wykorzystać te moduły w programach, które ich wymagają.

### Łatwiejsza praca w zespołach

Moduły sprawiają także, że programistom jest dużo łatwiej pracować w zespołach. W sytuacji, gdy tworzony program jest podzielony na moduły wykonujące poszczególne zadania, do pracy nad każdym z modułów można oddelegować innego programistę.



### Punkt kontrolny

- 3.1. Co to jest moduł?
- 3.2. O co chodzi w metodzie „dziel i zwyciężaj”?
- 3.3. W jaki sposób moduły umożliwiają wielokrotne wykorzystanie kodu?
- 3.4. Wyjaśnij, dlaczego dzięki wykorzystaniu modułów praca nad kilkoma programami jednocześnie przebiega szybciej.
- 3.5. Wyjaśnij, dlaczego dzięki wykorzystaniu modułów praca w zespole programistów jest znacznie łatwiejsza.

## 3.2

### Definiowanie i wywoływanie modułów

**WYJAŚNIENIE:** Kod programu umieszczony w module nazywa się definicją modułu. Aby wykonać moduł, należy skorzystać z polecenia wywołującego danego moduł.

### Nazwy modułów

Zanim przejdziemy do omawiania tego, w jaki sposób się tworzy moduły i z nich korzysta, muszę wspomnieć o kilku zagadnieniach dotyczących ich nazw. Podobnie jak w przypadku zmiennych, modułom także należy nadać nazwy. Nazwa powinna być na tyle jasna, aby każdy, kto będzie korzystał z danego modułu, mógł się domyślić, do czego on służy.

Ponieważ zadaniem modułów jest wykonywanie pewnych operacji, wielu programistów nadaje im nazwy zawierające jakiś czasownik. Przykładowo moduł, który oblicza wynagrodzenie brutto, może się nazywać `calculateGrossPay`. Taka nazwa jednoznacznie podpowie osobie mającej zamiar skorzystać z tego modułu, że służy on do obliczania czegoś. Ale do obliczania czego? Oczywiście wynagrodzenia brutto. Innymi przykładami nazw modułów mogą być: `getHours`, `getPayrate`, `calculateOvertime`, `printCheck` itp. Nazwa wskazuje więc, co dany moduł robi.

W przypadku większości języków programowania nazwy modułów muszą spełniać te same zasady co nazwy zmiennych. Oznacza to, że nazwa modułu nie może zawierać znaków spacji, zazwyczaj nie może zawierać znaków przestankowych i najczęściej nie może zaczynać się od cyfry. Są to jednak tylko ogólne zasady. W przypadku konkretnego języka zasady nazewnictwa modułów mogą się nieco różnić — przypomnij sobie, jak omawiałem zasady nazewnictwa zmiennych w rozdziale 2.

## Definiowanie i wywoływanie modułu

Aby stworzyć moduł, należy napisać jego **definicję**. W większości języków programowania definicja modułu składa się z dwóch elementów: nagłówka i ciała. **Nagłówek** określa punkt początkowy modułu, a **ciało** to lista instrukcji wchodzących w skład modułu. Oto ogólny schemat modułu, zapisany za pomocą pseudokodu:

```
Module name()
    instrukcja
    instrukcja
    itd.
}
End Module
```

**Te instrukcje to ciało modułu.**

W pierwszej linii umieszczony jest nagłówek. W pseudokodzie składa się on ze słowa **Module**, następnie nazwy modułu i pary nawiasów. Umieszczanie nawiasów za nazwą modułu jest bardzo częstą praktyką w wielu językach programowania. Z dalszej części rozdziału dowiesz się, do czego służą nawiasy, jednak teraz wystarczy, że zapamiętasz, iż należy je umieścić zaraz za nazwą modułu.

Po linii zawierającej nagłówek umieszcza się jedną lub kilka instrukcji — stanowią one ciało modułu i zostaną wykonane po każdym wywołaniu modułu. W ostatniej linii definicji, po ciele modułu, pojawia się tekst **End Module**. Linia ta wskazuje miejsce, w którym kończy się definicja modułu.

Spójrzmy na przykład (pamiętaj, że nie jest to kompletny program, cały kod programu zaprezentuję za chwilę):

```
Module showMessage()
    Display "Witaj, świecie!"
End Module
```

Zdefiniowałem tutaj za pomocą pseudokodu moduł o nazwie **showMessage**. Jak wskazuje nazwa, moduł ten służy do wyświetlania komunikatu na ekranie. Ciało modułu składa się tylko z jednej instrukcji: **Display** — jej zadaniem jest wyświetlenie na ekranie tekstu **Witaj, świecie!**.

Zwróć uwagę, że w zaprezentowanym przykładzie instrukcja stanowiąca ciało modułu jest wcięta. Zazwyczaj nie trzeba tego robić w prawdziwym programie<sup>1</sup>, jednak dzięki temu program staje się znacznie bardziej czytelny. Wcięcia linii wewnętrz modułu powodują, że odznaczają się ona graficznie w kodzie. Sprawia to, że już na pierwszy rzut oka widać, które linie należą do ciała modułu. Techniki tej używają praktycznie wszyscy programiści.

---

<sup>1</sup> Język Python wymaga, aby instrukcje będące ciałem modułu były wcięte.

## Wywoływanie modułu

Definicja modułu wskazuje jedynie, co dany moduł robi, nie powoduje jednak wykonania zawartych w nim instrukcji. Aby wykonać instrukcje zawarte w module, należy wywołać moduł. W pseudokodzie służy do tego instrukcja `Call`. Oto przykład wywoływania modułu o nazwie `showMessage`:

```
Call showMessage()
```

W momencie wywoływania modułu komputer przechodzi do niego i wykonuje instrukcje zawarte w ciele modułu. Gdy komputer dotrze do końca modułu, wraca do miejsca, w którym moduł został wywołany, i kontynuuje pracę.

Aby w pełni zilustrować, jak działa wywoływanie modułu, posłużę się przykładem z listingu 3.1.

### Listing 3.1



```
1 Module main()
2   Display "Mam dla Ciebie wiadomość."
3   Call showMessage()
4   Display "To wszystko!"
5 End Module
6
7 Module showMessage()
8   Display "Witaj, świecie!"
9 End Module
```

### Wynik działania programu

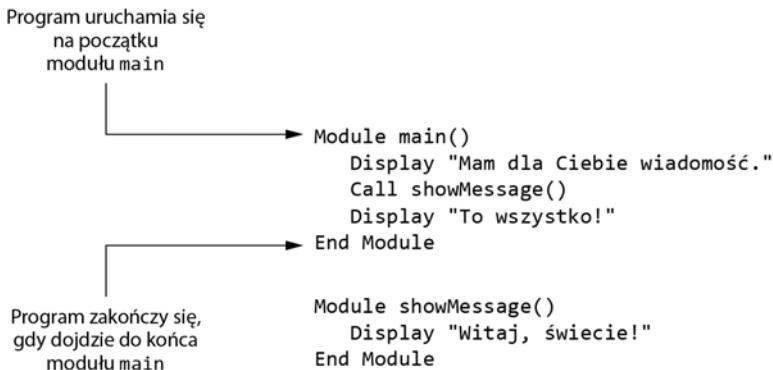
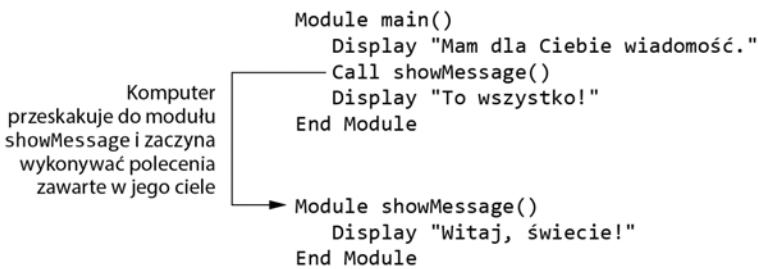
```
Mam dla Ciebie wiadomość.
Witaj, świecie!
To wszystko!
```

Zwróć uwagę, że program na listingu 3.1 zawiera dwa moduły: moduł o nazwie `main` znajduje się w liniach od 1. do 5., a moduł o nazwie `showMessage` znajduje się w liniach od 7. do 9. Wiele języków programowania wymaga umieszczenia w programie **modułu głównego**. Moduł główny to punkt początkowy programu, wewnątrz którego wywoływane są zazwyczaj inne moduły. Program zakończy się, gdy dojdzie do końca modułu głównego. Kiedy zobaczyś w książce przykład zapisany za pomocą pseudokodu zatrzymujący moduł o nazwie `main`, oznacza on po prostu początek programu. Analogicznie program zakończy działanie, kiedy dojdzie do końca modułu `main`. Przedstawiłem to na rysunku 3.2.

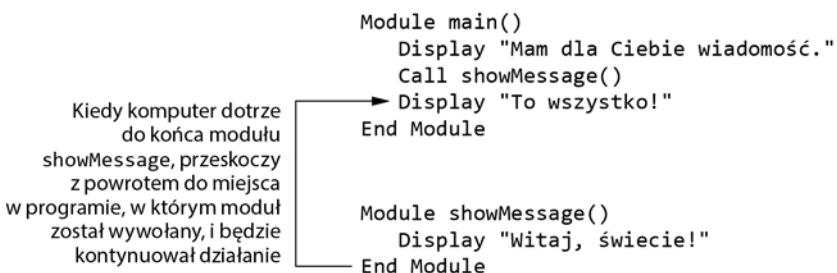


**UWAGA:** W wielu językach programowania (np. w Javie, C i C++), wymaga się umieszczenia w programie modułu głównego o nazwie `main` — tak jak przedstawiłem to na listingu 3.1.

Prześledźmy ten program. Kiedy program zacznie działać, uruchomi się moduł `main` i wyświetli komunikat *Mam dla Ciebie wiadomość*. Potem w linii 3. następuje wywołanie modułu `showMessage`. Jak widać na rysunku 3.3, komputer przeskakuje do modułu

**Rysunek 3.2.** Moduł main**Rysunek 3.3.** Wywołanie modułu showMessage

showMessage i wykonuje polecenia zawarte w jego ciele. Zawiera ono tylko jedną instrukcję: `Display` w linii 8. Instrukcja ta wyświetla na ekranie komunikat `Witaj, świecie!` i moduł się kończy. Jak widać na rysunku 3.4, komputer wraca do miejsca w programie, w którym nastąpiło wywołanie modułu `showMesage`. i przystępuje do wykonania kolejnych instrukcji. W tym przypadku jest to linia 4., w której wyświetlany jest komunikat `To wszystko!`. W linii 5. następuje koniec modułu `main` i program kończy działanie.

**Rysunek 3.4.** Powrót z modułu showMessage

Kiedy komputer trafi na wywołanie modułu, na przykład takie jak w linii 3. na listingu 3.1, musi „po kryjomu” wykonać kilka operacji, tak aby „wiedział”, gdzie ma wrócić, gdy wykona ostatnią operację w wywoływany module. Komputer najpierw zapamiętuje adres w pamięci, do którego powinien wrócić — jest to zazwyczaj

adres kolejnej instrukcji występującej zaraz po wywołaniu modułu. Adres ten nazywa się adresem powrotnym. Następnie komputer przeskakuje do miejsca, w którym zaczyna się moduł, i wykonuje kolejne instrukcje w nim zawarte. Kiedy moduł się skończy, komputer przeskakuje pod zapisany wcześniej adres powrotny i kontynuuje wykonywanie programu.



**UWAGA:** Kiedy następuje wywołanie modułu, mówi się, że moduł przejmuje sterowanie nad programem. Oznacza to, że teraz moduł jest odpowiedzialny za działanie programu.

## Tworzenie schematu blokowego programu za pomocą modułów

W przypadku schematów blokowych wywołanie modułu oznacza się za pomocą prostokąta z dodatkowymi pionowymi kreskami po bokach — tak jak przedstawiłem to na rysunku 3.5. W środku symbolu umieszczona jest także nazwa modułu, który ma zostać wywołany. W przykładzie przedstawionym na rysunku 3.5 pokazałem, jak wygląda wywołanie modułu o nazwie `showMessage`.



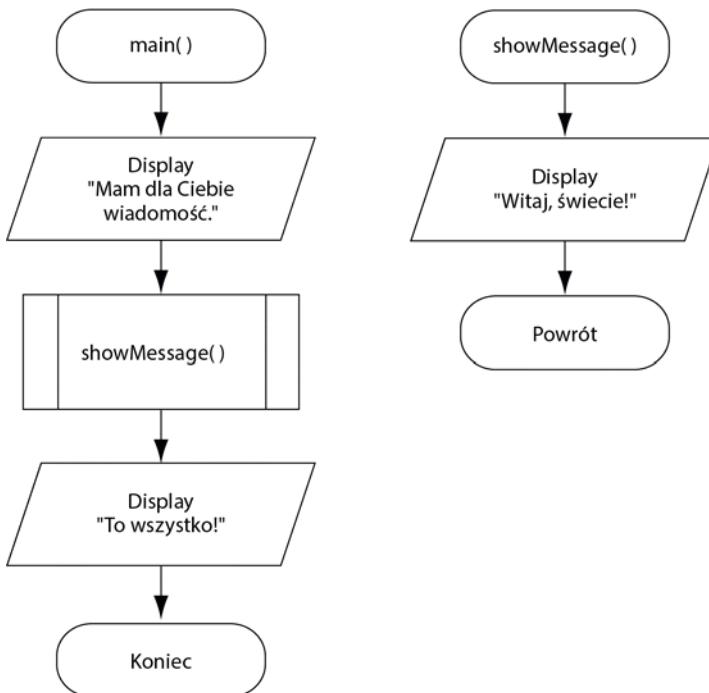
Rysunek 3.5. Symbol wywołania modułu

Programiści najczęściej rysują osobny schemat blokowy dla każdego modułu w programie. Na rysunku 3.6 pokazałem, jak wygląda schemat blokowy programu z listingu 3.1. Zwróć uwagę, że na rysunku znajdują się dwa schematy blokowe: pierwszy reprezentuje moduł `main`, a drugi — moduł `showMessage`.

W schemacie blokowym modułu pierwszy symbol graniczny zawiera przeważnie nazwę tego modułu. W przypadku modułu `main` w dolnym symbolu granicznym znajduje się napis `Koniec`, ponieważ oznacza to koniec programu, natomiast w przypadku pozostałych modułów w dolnym symbolu granicznym widnieje napis `Powrót`, gdyż komputer w tym miejscu powróci do miejsca, w którym dany moduł został wywołany.

## Projektowanie „od ogółu do szczegółu”

W tym podrozdziale wyjaśniam działanie modułów. Dowiedziałeś się, jak komputer przechodzi do modułu w momencie jego wywołania i jak po zakończeniu modułu powraca do miejsca w kodzie, w którym moduł został wywołany. Zrozumienie tej mechaniki jest bardzo ważne.



**Rysunek 3.6.** Schemat blokowy programu z listingu 3.1

Równie ważne jest jednak zrozumienie, w jaki sposób projektuje się taki podzielony na moduły program. Aby podzielić algorytm na moduły, programiści często używają techniki zwanej projektowaniem „od ogółu do szczegółu”. Technika ta wygląda następująco:

- Zadanie główne, jakie ma wykonać program, dzielimy na mniejsze podzadania.
- Każde z podzadań analizujemy i sprawdzamy, czy nie da się go podzielić na jeszcze mniejsze podzadania. Powtarzamy ten krok dopóty, dopóki jesteśmy w stanie dzielić zadanie na kolejne podzadania.
- Kiedy wszystkie podzadania zostały określone, przystępujemy do pisania kodu każdego z nich.

Technika ta nazywa się „od ogółu do szczegółu”, ponieważ programista zaczyna od najbliższego obrazu zadania, a następnie dzieli je na bardziej szczegółowe podzadania.

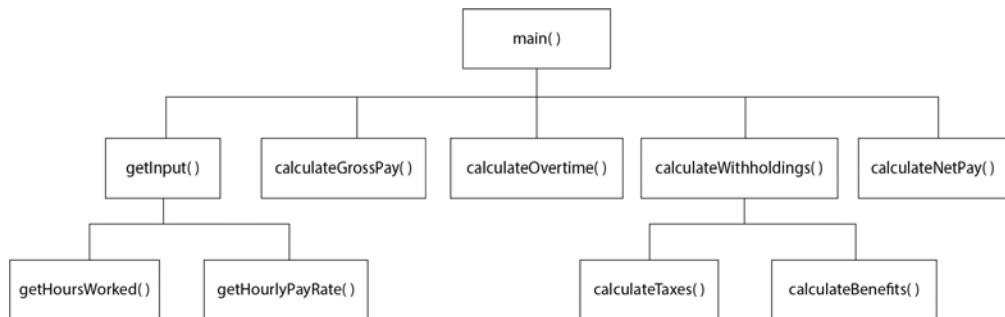


**UWAGA:** Technika „od ogółu do szczegółu” nazywana jest także techniką **uściślania stopniowego**.

## Schematy hierarchiczne

Schematy blokowe to bardzo pozyteczne narzędzia służące do graficznego opisu działania programu wewnątrz modułu, ale nie są one w stanie zaprezentować tego, jakie relacje łączą poszczególne moduły. Programiści często używają w takich

przypadkach **schematów hierarchicznych**. Schemat hierarchiczny, zwany też niekiedy **schematem strukturalnym**, przedstawia za pomocą prostokątów poszczególne moduły programu. Prostokąty są ze sobą połączone w sposób, który odzwierciedla relacje łączące moduły. Na rysunku 3.7 pokazałem schemat hierarchiczny programu do obliczania wynagrodzenia.



**Rysunek 3.7.** Schemat hierarchiczny

Na schemacie przedstawionym na rysunku 3.7 widać, że na najwyższym poziomie hierarchii znajduje się moduł `main`. Wywołuje on pięć innych modułów: `getInput`, `calculateGrossPay`, `calculateOvertime`, `calculateWithholdings` i `calculateNetPay`. Moduł `getInput` z kolei wywołuje moduły `getHoursWorked` i `getHourlyPayRate`. Moduł `calculateWithholdings` także wywołuje dwa moduły: `calculateTaxes` i `calculateBenefits`.

Zwróć uwagę, że schemat hierarchiczny w żaden sposób nie informuje, jakie operacje wykonuje dany moduł. Ponieważ schematy hierarchiczne nie zawierają żadnych informacji o tym, jak działa dany moduł, nie są one w stanie zastąpić schematów blokowych czy pseudokodu.

## W centrum uwagi

### Definiowanie i wywoływanie modułów

Firma Professional Appliance Service Inc. świadczy usługi w zakresie konserwacji i naprawy sprzętu AGD. Właściciel firmy chce, aby każdy z zatrudnionych u niego serwisantów posiadał mały komputerek, na którym będą się wyświetlały szczegółowe instrukcje serwisowe wielu urządzeń AGD. Aby sprawdzić, jak to będzie działało, właściciel firmy poprosił Cię o stworzenie programu, który będzie wyświetlał następujące instrukcje demontażu suszarki na pranie ACME:

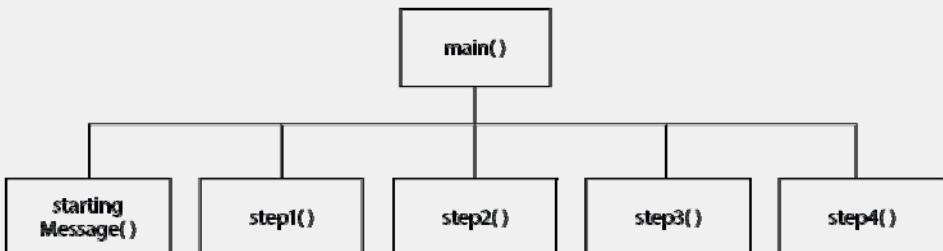
- Krok 1. Odłącz suszarkę i odsuń ją od ściany.
- Krok 2. Odkręć sześć śrub z tyłu suszarki.
- Krok 3. Zdejmij tylny panel suszarki.
- Krok 4. Pociągnij do góry pokrywę suszarki.



Podczas rozmowy z właścicielem firmy dowiedziałeś się, że program powinien wyświetlać w danym momencie tylko jeden komunikat. Zdecydowałeś, że po wyświetleniu komunikatu dla danego kroku użytkownik będzie mógł nacisnąć klawisz, aby przejść do kolejnego kroku. Oto algorytm takiego programu:

1. Wyświetl komunikat początkowy, informujący serwisanta o tym, jak działa program.
2. Poproś użytkownika, żeby nacisnął klawisz, jeśli chce przejść do kroku 1.
3. Wyświetl instrukcje dla kroku 1.
4. Poproś użytkownika, żeby nacisnął klawisz, jeśli chce przejść do kolejnego kroku.
5. Wyświetl instrukcje dla kroku 2.
6. Poproś użytkownika, żeby nacisnął klawisz, jeśli chce przejść do kolejnego kroku.
7. Wyświetl instrukcje dla kroku 3.
8. Poproś użytkownika, żeby nacisnął klawisz, jeśli chce przejść do kolejnego kroku.
9. Wyświetl instrukcje dla kroku 4.

Algorytm ten przedstawia najbardziej ogólne zadania, jakie musi wykonać program, i będzie on bazą dla modułu `main` w naszym programie. Na rysunku 3.8 przedstawiono strukturę programu za pomocą schematu hierarchicznego.

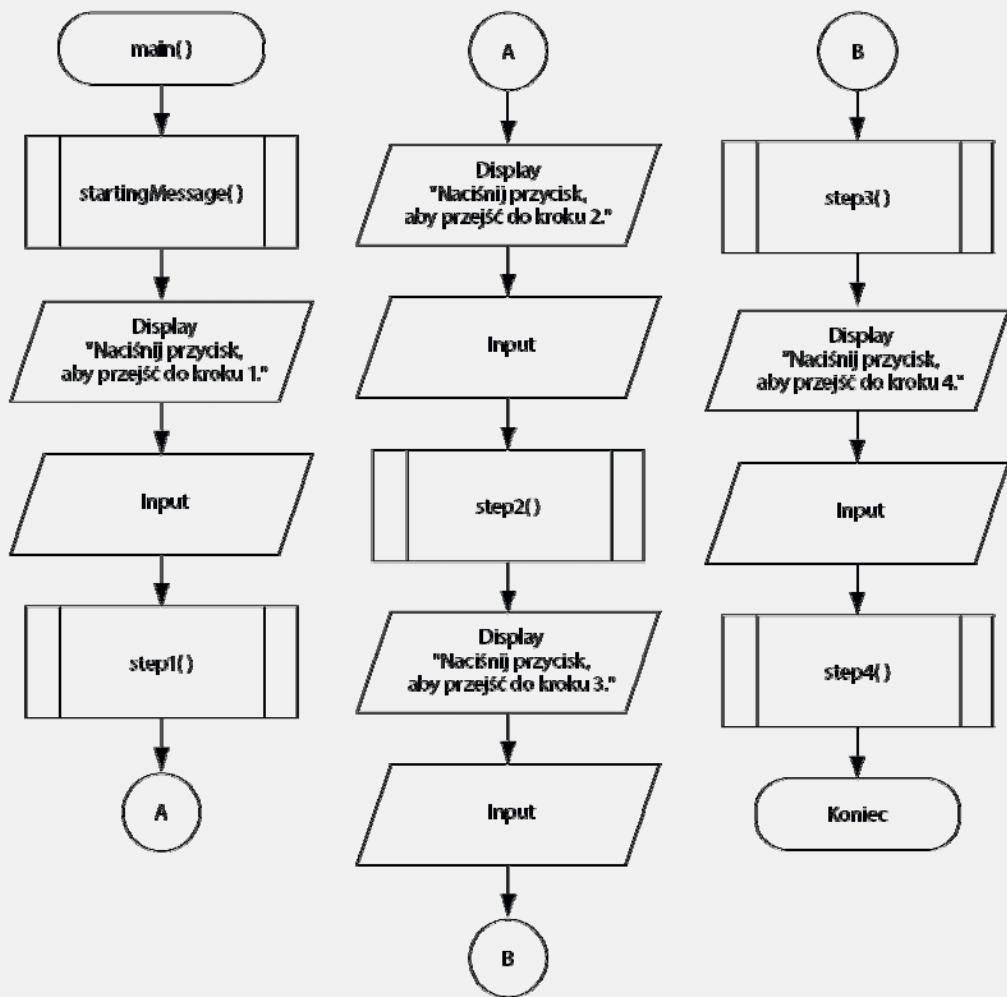


**Rysunek 3.8.** Schemat hierarchiczny programu

Jak widać na schemacie hierarchicznym, moduł `main` będzie wywoływał kilka innych modułów. Oto podsumowanie tych modułów:

- `startingMessage` — będzie wyświetlał komunikat początkowy, informujący serwisanta o tym, jak działa program;
- `step1` — będzie wyświetlał instrukcje dla kroku 1.;
- `step2` — będzie wyświetlał instrukcje dla kroku 2.;
- `step3` — będzie wyświetlał instrukcje dla kroku 3.;
- `step4` — będzie wyświetlał instrukcje dla kroku 4..

Pomiędzy wywoywaniem kolejnych modułów moduł `main` będzie informował użytkownika, że aby przejść do kolejnego kroku, należy nacisnąć klawisz. Na listingu 3.2 przedstawiłem pseudokod takiego programu. Na rysunku 3.9 widoczny jest schemat blokowy modułu `main`, a na rysunku 3.10 — schematy blokowe modułów `starting Message`, `step1`, `step2`, `step3` i `step4`.



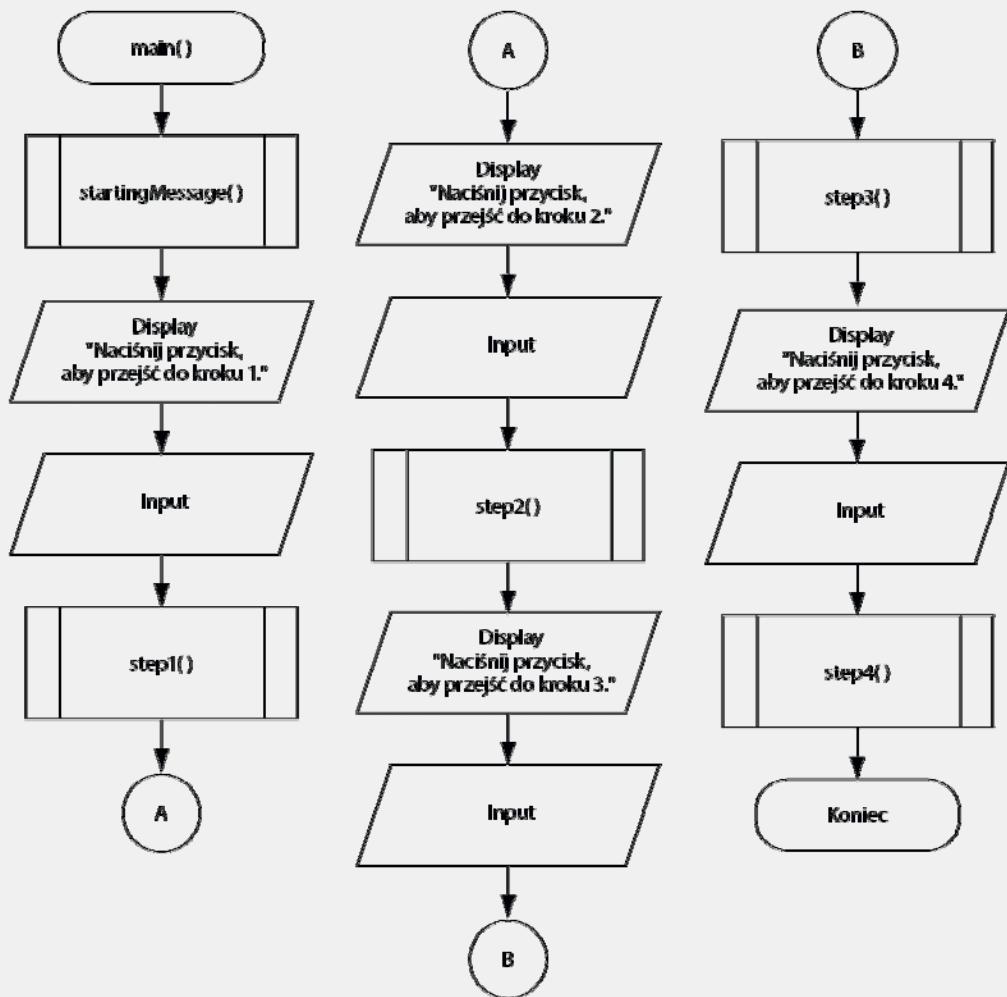
Rysunek 3.9. Schemat blokowy modułu main w programie z listingu 3.2

### Listing 3.2

```

1 Module main()
2 // Wyświetlenie komunikatu początkowego
3 Call startingMessage()
4 Display "Naciśnij przycisk, aby przejść do kroku 1."
5 Input
6
7 // Wyświetl krok 1.
8 Call step1()
9 Display "Naciśnij przycisk, aby przejść do kroku 2."
10 Input
11
12 // Wyświetl krok 2.
13 Call step2()

```



**Rysunek 3.10.** Schemat blokowy pozostałych modułów w programie z listingu 3.2

```

14     Display "Naciśnij przycisk, aby przejść do kroku 3."
15     Input
16
17     // Wyświetl krok 3.
18     Call step3()
19     Display "Naciśnij przycisk, aby przejść do kroku 4."
20     Input
21
22     // Wyświetl krok 4.
23     Call step4()
24 End Module
25
26 // Moduł startingMessage wyświetla
27 // komunikat początkowy
28 Module startingMessage()
29     Display "Ten program poinstruuje Cię,"

```

```

30     Display "w jaki sposób zdemontować suszarkę ACME."
31     Display "Proces składa się z czterech kroków."
32 End Module
33
34 // Moduł step1 wyświetla instrukcje
35 // dla kroku 1.
36 Module step1()
37     Display "Krok 1.: Odłącz suszarkę"
38     Display "i odsuń ją od ściany."
39 End Module
40
41 // Moduł step2 wyświetla instrukcje
42 // dla kroku 2.
43 Module step2()
44     Display "Krok 2.: Odkręć sześć śrub"
45     Display "z tyłu suszarki."
46 End Module
47
48 // Moduł step3 wyświetla instrukcje
49 // dla kroku 3.
50 Module step3()
51     Display "Krok 3.: Zdejmij tylny"
52     Display "panel suszarki."
53 End Module
54
55 // Moduł step4 wyświetla instrukcje
56 // dla kroku 4.
57 Module step4()
58     Display "Krok 4.: Pociągnij do góry"
59     Display "pokrywę suszarki."
60 End Module

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Ten program poinstruuje Cię,  
w jaki sposób zdemontować suszarkę ACME.  
Proces składa się z czterech kroków.  
Naciśnij przycisk, aby przejść do kroku 1.

**[Enter]**

Krok 1.: Odłącz suszarkę  
i odsuń ją od ściany.  
Naciśnij przycisk, aby przejść do kroku 2.

**[Enter]**

Krok 2.: Odkręć sześć śrub  
z tyłu suszarki.  
Naciśnij przycisk, aby przejść do kroku 3.

**[Enter]**

Krok 3.: Zdejmij tylny  
panel suszarki.  
Naciśnij przycisk, aby przejść do kroku 4.

**[Enter]**

Krok 4.: Pociągnij do góry  
pokrywę suszarki.



**UWAGA:** W liniach 5., 10., 15. i 20. pojawia się instrukcja Input, ale nie ma przy niej żadnej zmiennej. W ten sposób możemy w pseudokodzie odczytać naciśnięty klawisz, bez konieczności zapisywania w zmiennej, jaki klawisz nacisnął użytkownik. Taką operację umożliwia wiele języków programowania.



## Punkt kontrolny

- 3.6. Z jakich dwóch elementów składa się deklaracja modułu w większości języków programowania?
- 3.7. Co oznacza określenie „wywołanie modułu”?
- 3.8. Co się stanie, gdy podczas wykonywania kodu w module program dojdzie do jego końca?
- 3.9. Wyjaśnij, na czym polega technika projektowania „od ogółu do szczegółu”.

### 3.3

## Zmienne lokalne

**WYJAŚNIENIE:** Zmienną lokalną deklaruje się wewnątrz modułu i nie można się do niej odwoływać poza modułem. Ponieważ jeden moduł nie może się odwoływać do zmiennych lokalnych zadeklarowanych w innym module, każdy z modułów może zawierać zmienne lokalne o takich samych nazwach.

W przypadku większości języków programowania zmienna zadeklarowana wewnątrz modułu nazywa się **zmienną lokalną**. Zmienna lokalna należy tylko i wyłącznie do modułu, w którym została zadeklarowana, i mogą z niej korzystać tylko instrukcje wewnątrz tego modułu. Okreście „lokalna” wskazuje, że z danej zmiennej można korzystać lokalnie, tylko w obrębie modułu, w którym została ona zadeklarowana.

W przypadku, gdy instrukcja w jednym z modułów będzie się chciała odwołać do zmiennej lokalnej należącej do innego modułu, pojawi się błąd. Przyjrzyj się programowi zapisanemu za pomocą pseudokodu na listingu 3.3.

### Listing 3.3

```

1 Module main()
2   Call getName()
3   Display "Cześć ", name ←———— W tej linii jest błąd!
4 End Module
5
6 Module getName()
7   Declare String name ←———— To jest zmienna lokalna
8   Display "Wprowadź swoje imię."
9   Input name
10 End Module

```

W linii 2. modułu `getName` jest wywoływany przez moduł `main`. Następnie w linii 3. instrukcja `Display` próbuje wyświetlić wartość zapisaną w zmiennej `name`. Będzie to skutkowało pojawiением się błędu, ponieważ zmienna `name` jest zmienną lokalną należącą do modułu `getName`, więc instrukcje w module `main` nie mają do niej dostępu.

## Zasięg widoczności zmiennej i zmienne lokalne

Aby określić tę część programu, w której można się odwoływać do danej zmiennej, programiści często używają pojęcia **zasięgu widoczności zmiennej**. Do danej zmiennej można się odwoływać tylko w instrukcjach znajdujących się w jej zasięgu widoczności.

Zasięg widoczności zmiennej lokalnej zaczyna się zazwyczaj w miejscu, w którym została ona zadeklarowana, a kończy na samym końcu modułu, w którym została zadeklarowana. Poza tym obszarem nie można się odwoływać do tej zmiennej. Oznacza to, że zarówno kod umieszczony na zewnątrz modułu, jak i kod wewnętrz modułu, ale przed miejscem deklaracji zmiennej, nie może korzystać z danej zmiennej. Przyjrzyj się poniższemu kodowi. Zawiera on błąd, ponieważ instrukcja `Input` znajduje się poza zasięgiem widoczności zmiennej `name`, a mimo to próbuje zapisać w niej wartość. Przesunięcie linii, w której deklarujemy zmienną, przed instrukcję `Input` rozwiąże ten problem.

```
Module getName()
    Display "Wprowadź swoje imię."
    Input name ← Ta instrukcja spowoduje wystąpienie błędu, ponieważ zmieniła name nie została jeszcze zadeklarowana.
    Declare String name
End Module
```

## Zdublowane nazwy zmiennych

W przypadku większości języków programowania nie można nadać takiej samej nazwy dwóm zmiennym występującym w tym samym zasięgu widoczności. Spójrz na następujący moduł:

```
Module getTwoAges()
    Declare Integer age
    Display "Podaj swój wiek."
    Input age
    Declare Integer age ← Tu jest błąd!
    Display "Podaj wiek Twojego zwierzaka."
    Input age Zmienią o nazwie age została już zadeklarowana.
End Module
```

W module zadeklarowane są dwie zmienne o nazwie `age`. Podczas wykonania instrukcji, w której następuje ponowna deklaracja zmiennej `age`, wystąpi błąd, ponieważ zmienna o takiej nazwie została już wcześniej zadeklarowana w tym module. Problem ten można rozwiązać, modyfikując nazwę drugiej zmiennej.



**WSKAZÓWKA:** Nie możesz umieścić w jednym module dwóch zmiennych o takich samych nazwach, ponieważ kompilator lub interpreter nie wiedziałby, której z nich ma użyć, gdy nastąpi odwołanie do jednej z nich. Wszystkie zmienne umieszczone w danym module muszą więc mieć unikatowe nazwy.

W jednym module nie można zadeklarować dwóch zmiennych o takich samych nazwach, jednak zazwyczaj nie stanowi problemu nadanie w jednym module zmiennej lokalnej takiej samej nazwy, jaką ma zmienna lokalna w innym module. Spójrz na przykładowy program na listingu 3.4.

W programie z listingu 3.4 występują trzy moduły: main, showSquare i showHalf. Zwróć uwagę, że w module showSquare znajduje się zmienna lokalna o nazwie number (zadeklarowana w linii 7.), a w module showHalf znajduje się także zmienna lokalna o nazwie number (zadeklarowana w linii 17.). W tym przypadku jak najbardziej prawidłowe jest użycie dwóch zmiennych o takich samych nazwach, ponieważ nie występują one w tym samym zasięgu widoczności. Na rysunku 3.11 przedstawiłem zasięg widoczności obu zmiennych number.

### **Listing 3.4**

```

1 Module main()
2   Call showSquare()
3   Call showHalf()
4 End Module
5
6 Module showSquare()
7   Declare Real number
8   Declare Real square
9
10  Display "Wprowadź liczbę."
11  Input number
12  Set square = number^2
13  Display "Druga potęga tej liczby jest równa ", square
14 End Module
15
16 Module showHalf()
17   Declare Real number
18   Declare Real half
19
20  Display "Wrowadź liczbę."
21  Input number
22  Set half = number / 2
23  Display "Połowa tej liczby jest równa ", half
24 End Module

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę.

**5 [Enter]**

Druga potęga tej liczby jest równa 25

Wprowadź liczbę.

**20 [Enter]**

Połowa tej liczby jest równa 10



### **Punkt kontrolny**

3.10. Co to są zmienne lokalne? W jakim obszarze można odwoływać się do zmiennej lokalnej?

3.11. Co to jest zasięg widoczności zmiennej?

```

Module main()
  Call showSquare()
  Call showHalf()
End Module

Module showSquare()
  Declare Real number
  Declare Real square

  Display "Wprowadź liczbę."
  Input number
  Set square = number^2
  Display "Druga potęga tej liczby jest równa ", square
End Module

Module showHalf()
  Declare Real number
  Declare Real half

  Display "Wprowadź liczbę."
  Input number
  Set half = number / 2
  Display "Połowa tej liczby jest równa ", half
End Module

```

← Zasięg widoczności zmiennej  
number w module showSquare

← Zasięg widoczności zmiennej  
number w module showHalf

Rysunek 3.11. Zasięg widoczności dwóch zmiennych liczbowych

- 3.12. Czy można zadeklarować dwie zmienne o takich samych nazwach w jednym zasięgu widoczności? Wyjaśnij, dlaczego można lub nie można tak postąpić.
- 3.13. Czy można nadać zmiennej lokalnej w jednym module taką samą nazwę, jaką ma zmienna lokalna w innym module?

### 3.4

## Przekazywanie argumentów do modułów

**WYJAŚNIENIE:** Argument to dane, które przekazuje się do modułu w momencie jego wywołania. Parametr to zmienna, w której zapisany jest argument przekazany do modułu.

W pewnych sytuacjach pomocne okazuje się nie tylko wywołanie modułu, ale także przekazanie do niego jednej lub kilku danych. Dane przekazywane do modułu nazywamy **argumentami**. Moduł może następnie wykorzystać przekazane przez argumenty dane do obliczeń lub innych operacji.

Jeśli chcesz, aby moduł był w stanie odebrać przekazane do niego argumenty, musisz wyposażyć go w jeden lub kilka parametrów. **Parametr** to specjalna zmienna, która odbiera informację przekazaną jako argument podczas wywołania modułu. Oto przykład modułu z parametrem zapisany za pomocą pseudokodu:

```

Module doubleNumber(Integer value)
  Declare Integer result
  Set result = value * 2
  Display result
End Module

```

Powyższy moduł nazywa się `doubleNumber`. Jego zadaniem jest pobranie liczby przekazanej jako argument i wyświetlenie wartości równej dwukrotności tej liczby. Przyjrzyj się nagłówkowi modułu i zwróć uwagę na słowa `Integer value`, które pojawiają się w nawiasach — stanowią one deklarację parametru. Parametr to zmienna typu `Integer` o nazwie `value`. Zadaniem tej zmiennej jest odebranie wartości typu `Integer` przekazanej jako argument podczas wywołania modułu. Na listingu 3.5 zaprezentowałem wykorzystanie tego modułu w kompletnym programie.

### **Listing 3.5**

```

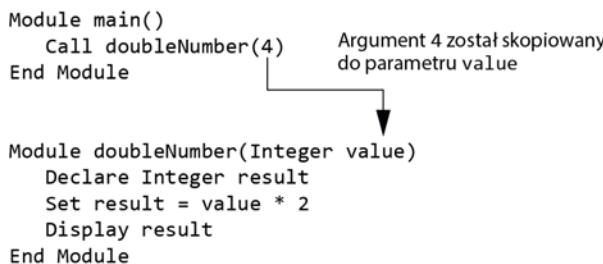
1 Module main()
2   Call doubleNumber(4)
3 End Module
4
5 Module doubleNumber(Integer value)
6   Declare Integer result
7   Set result = value * 2
8   Display result
9 End Module

```

### **Wynik działania programu**

8

Po uruchomieniu programu komputer zacznie wykonywać kod w module `main`. Instrukcja w linii 2. wywołuje moduł `doubleNumber` — zwróć uwagę, że wewnątrz nawiasów znajduje się liczba 4. Jest to argument przekazany do modułu `doubleNumber`. Po uruchomieniu tej instrukcji wywołany zostanie moduł `doubleNumber` i zostanie w nim przypisana wartość 4 do parametru `value`. Ilustruje to rysunek 3.12.



**Rysunek 3.12.** Argument 4 jest kopiowany do parametru `value`

Prześledźmy teraz kod w module `doubleNumber`. Gdy będziemy to robili, pamiętaj, że w zmiennej `value` będzie zapisany argument, jaki przekazaliśmy do modułu. W naszym przypadku będzie to liczba 4.

W linii 6. deklarujemy zmienną lokalną typu `Integer` o nazwie `result`. Następnie w linii 7. przypisujemy do niej wartość zwracaną przez wyrażenie `value * 2`. Ponieważ w zmiennej `value` jest zapisana wartość 4, w linii 8. zostanie przypisana do zmiennej `result` wartość 8. Moduł kończy się w linii 9.

Gdybyśmy przykładowo wywołały moduł w sposób następujący:

```
Call doubleNumber(5)
```

to moduł wyświetliłby wartość 10.

Jako argument możemy także przekazać wartość przypisaną do zmiennej. Spójrz na program z listingu 3.6. W module main w linii 2. zadeklarowałem zmienną typu Integer o nazwie number. W liniach 3. i 4. proszę użytkownika, aby wprowadził dowolną liczbę, a w linii 5. pobieram i zapisuję w zmiennej number wprowadzoną wartość. Zwróć uwagę, że w linii 6. przekazuję zmienną number jako argument modułu doubleNumber, co spowoduje, że wartość zapisana w zmiennej number zostanie skopiowana do parametru value w module. Pokazałem to na rysunku 3.13.

### **Listing 3.6**



```

1 Module main()
2   Declare Integer number
3   Display "Wprowadź dowolną liczbę, a ja wyświetrzę"
4   Display "liczbę dwa razy większą."
5   Input number
6   Call doubleNumber(number)
7 End Module
8
9 Module doubleNumber(Integer value)
10  Declare Integer result
11  Set result = value * 2
12  Display result
13 End Module

```

#### **Wynik działania programu (po grubione linie to dane wprowadzone przez użytkownika)**

Wprowadź dowolną liczbę, a ja wyświetrzę  
liczbę dwa razy większą.

20 [Enter]

40

```

Module main()
  Declare Integer number
  Display "Wprowadź dowolną liczbę, a ja wyświetrzę"
  Display "liczbę dwa razy większą."
  Input number
  Call doubleNumber(number)
End Module

```

20

Wartość przypisana do zmiennej  
number została skopiowana  
do parametru value

```

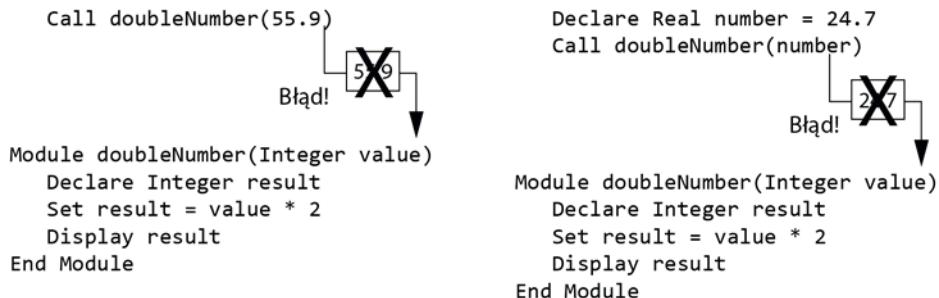
Module doubleNumber(Integer value)
  Declare Integer result
  Set result = value * 2
  Display result
End Module

```

**Rysunek 3.13.** Wartość przypisana do zmiennej number przekazana jako argument

## Dopasowanie typów danych parametru i argumentu

Większość języków programowania wymaga, aby zarówno argument, jak i parametr były takiego samego typu. Jeśli spróbujesz przekazać do modułu argument innego typu niż typ parametru, zazwyczaj program zgłosi błąd. Na rysunku 3.14 przedstawiłem przykład, z którego wynika, że nie można przekazać liczby rzeczywistej lub zmiennej typu Real do modułu, który oczekuje wartości typu Integer.



Rysunek 3.14. Argument i parametr muszą być tego samego typu



**UWAGA:** W niektórych językach programowania można przekazywać argument, który jest niezgodny z parametrem, ale tylko wtedy, gdy nie nastąpi w wyniku takiej operacji utrata danych. Przykładowo można przekazać argument w postaci liczby całkowitej do parametru typu rzeczywistego, ponieważ w zmiennych typu rzeczywistego można także zapisywać liczby całkowite. Gdybyś jednak chciał przekazać liczbę rzeczywistą, np. 24,7, do parametru typu całkowitego, część ułamkowa zostałaby utracona.

## Zasięg widoczności parametru

Wcześniej w tym rozdziale wyjaśniłem, że zasięg widoczności zmiennej określa tę część programu, w której można się do danej zmiennej odwoływać. Zmienna jest dostępna tylko dla polecień umieszczonych w jej zasięgu widoczności. Zasięg widoczności parametru obejmuje zazwyczaj cały moduł, w którym parametr został zdefiniowany. Nie ma więc do niego dostępu żadne polecenie znajdujące się na zewnątrz modułu.

## Przekazywanie kilku argumentów

W większości języków programowania można tworzyć moduły, które przyjmują wiele argumentów. Na listingu 3.7 przedstawiłem moduł `showSum`, który przyjmuje dwa argumenty typu Integer. Zadanie modułu polega na dodaniu do siebie tych dwóch argumentów i wyświetleniu ich sumy.

**Listing 3.7**

```

1 Module main()
2   Display "Suma liczb 12 i 45 wynosi:"
3   Call showSum(12, 45)
4 End Module
5
6 Module showSum(Integer num1, Integer num2)
7   Declare Integer result
8   Set result = num1 + num2
9   Display result
10 End Module

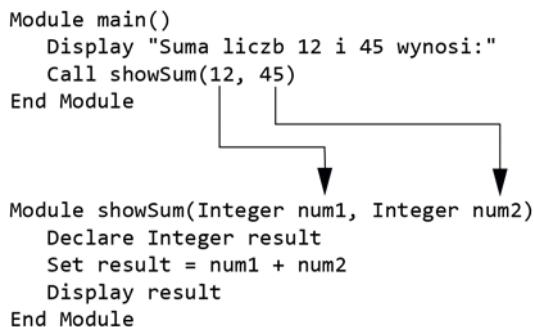
```

**Wynik działania programu**

Suma liczb 12 i 45 wynosi:  
57

Zwróć uwagę, że tym razem w nagłówku modułu w nawiasach umieściłem dwa parametry — num1 i num2. Nazywa się to często **listą parametrów**. Zauważ także, że parametry oddzielone są znakiem przecinka.

W linii 3. następuje wywołanie modułu showSum i przekazanie do niego dwóch argumentów: liczb 12 i 45. Argumenty przekazywane są do parametrów w takiej kolejności, w jakiej zostały zapisane przy wywołaniu modułu. Innymi słowy: pierwszy argument zostanie przekazany do pierwszego parametru, a drugi argument zostanie przekazany do drugiego parametru. Tak więc w naszym przykładzie do parametru num1 trafi liczba 12, a do parametru num2 trafi liczba 45, tak jak przedstawiłem to na rysunku 3.15.



**Rysunek 3.15.** Dwa argumenty przekazane do dwóch parametrów

Załóżmy, że odwrócilibyśmy kolejność, w jakiej występują argumenty podczas wywołania modułu:

Call showSum(45,12)

Spowodowałoby to, że do parametru num1 trafiłaby wartość 45, a do parametru num2 — wartość 12. Oto jeszcze jeden przykład — tym razem przekazuję argumenty w formie zmiennych:

```

Declare Integer value1 = 2
Declare Integer value2 = 3
Call showSum(value1, value2)

```

Podczas wywołania modułu do parametru `num1` trafi wartość 2, a do parametru `num2` trafi wartość 3.



## W centrum uwagi

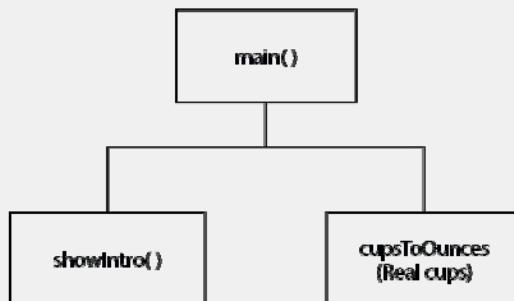
### Przekazywanie argumentu do modułu

Twój przyjaciel Michał prowadzi firmę kateringową. W przepisach niektórych serwówanych przez niego posiłków znajdują się składniki odmierzane za pomocą szklanek. Kiedy jednak kupuje składniki w sklepie spożywczym, są one dostępne wyłącznie w opakowaniach mierzonych w uncjach. Dlatego poprosił Cię o napisanie prostego programu, dzięki któremu będzie mógł zamienić szklanki na uncje.

Wymyśliłeś więc następujący algorytm:

1. Wyświetl ekran powitalny, informujący o tym, jak działa program.
2. Pobierz liczbę szklanek danego składnika.
3. Zamień liczbę szklanek na uncje i wyświetl wynik na ekranie.

Algorytm ten przedstawia najbardziej ogólne zadania, jakie musi wykonać program, i będzie on bazą dla modułu `main` w naszym programie. Na rysunku 3.16 przedstawiłem strukturę programu za pomocą schematu hierarchicznego.



**Rysunek 3.16.** Schemat hierarchiczny programu

Jak widać na schemacie, moduł `main` wywołuje dwa inne moduły.

Oto krótkie podsumowanie tych modułów:

- `showIntro` — będzie wyświetlał komunikat informujący, do czego służy program.
- `cupsToOunces` — będzie przyjmował jako argument liczbę szklanek danego składnika oraz będzie obliczał i wyświetlał odpowiadającą tej wartością liczbę uncji.

Poza wywołaniem tych modułów moduł `main` będzie też prosił użytkownika o wprowadzenie liczby szklanek danego składnika. Wartość ta zostanie następnie przekazana do modułu `cupsToOunces`. Na listingu 3.8 przedstawiłem pseudokod tego programu, a na rysunku 3.17 widoczny jest schemat blokowy programu.

**Listing 3.8**

```

1 Module main()
2   // Deklaracja zmiennej, w której zapisana
3   // będzie liczba szklanek danego składnika
4   Declare Real cupsNeeded
5
6   // Wyświetlenie komunikatu powitalnego
7   Call showIntro()
8
9   // Pobranie liczby szklanek
10  Display "Wprowadź liczbę szklanek składnika."
11  Input cupsNeeded
12
13  // Zamiana szklanek na uncje
14  Call cupsToOunces(cupsNeeded)
15 End Module
16
17 // Moduł showIntro wyświetla
18 // komunikat powitalny
19 Module showIntro()
20   Display "Program służy do zamiany szklanek"
21   Display "na uncje. Dla Twojej wiadomości"
22   Display "przelicznik wygląda następująco:"
23   Display "    1 szklanka = 8 uncji."
24 End Module
25
26 // Moduł cupsToOunces przyjmuje liczbę szklanek
27 // i wyświetla odpowiadającą jej
28 // liczbę uncji
29 Module cupsToOunces(Real cups)
30   // Deklaracja zmiennej
31   Declare Real ounces
32
33   // Zamiana szklanek na uncje
34   Set ounces = cups * 8
35
36   // Wyświetlenie wyniku
37   Display "Ta ilość odpowiada ",
38   oounces, " uncjom."
39 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

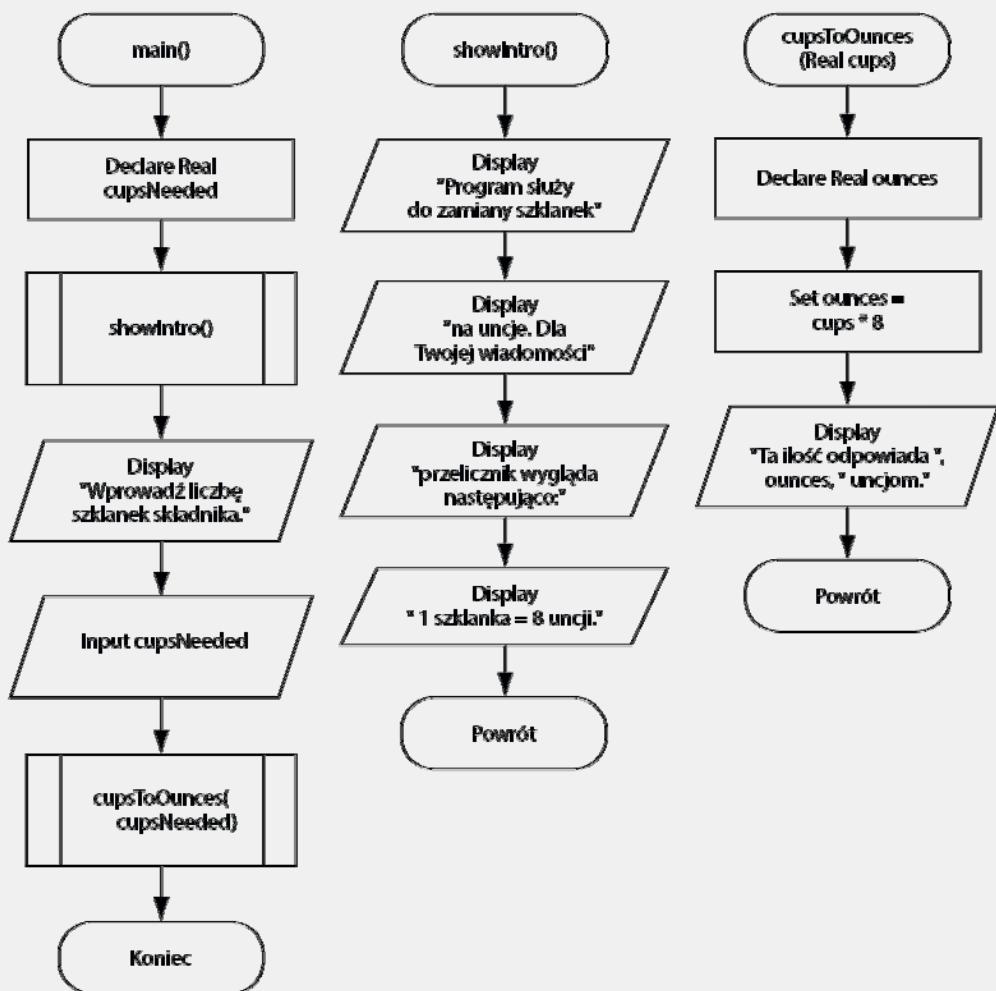
Program służy do zamiany szklanek  
na uncje. Dla Twojej wiadomości  
przelicznik wygląda następująco:

1 szklanka = 8 uncji.

Wprowadź liczbę szklanek składnika.

**2 [Enter]**

Ta ilość odpowiada 16 uncjom.



Rysunek 3.17. Schemat blokowy programu z listingu 3.8

## Przekazywanie argumentów przez wartość i przez referencję

W wielu językach programowania można przekazywać argumenty na dwa sposoby: przez wartość i przez referencję. Zanim przejdę do bardziej szczegółowego omawiania tego zagadnienia, chciałem wspomnieć, że w zależności od języka oba sposoby przekazywania argumentu mogą się nieco różnić. W tej książce przedstawię podstawowe informacje na ten temat i pokażę, jak robi się to za pomocą pseudokodu. Kiedy będziesz już pisać program w którymś z prawdziwych języków, będziesz musiał się dowiedzieć, jak to wygląda w tym konkretnym języku.

## Przekazywanie argumentu przez wartość

We wszystkich przedstawionych do tej pory przykładach przekazywaliśmy zmienne przez wartość. Argument i parametr w tym przypadku to dwa osobne elementy umieszczone w pamięci komputera. Przekazywanie argumentu **przez wartość** oznacza, że do parametru trafi kopia przekazanej wartości. Jeśli wewnątrz modułu zmienisz wartość zapisaną w parametrze, nie będzie to miało wpływu na wartość argumentu. Spójrz na przykład z listingu 3.9.

### **Listing 3.9**

```

1 Module main()
2 Declare Integer number = 99
3
4 // Wyświetlamy wartość zapisaną w zmiennej number
5 Display "Liczba jest równa ", number
6
7 // Wywołujemy moduł changeMe
8 // i przekazujemy jako argument zmiennej number
9 Call changeMe(number)
10
11 // Ponownie wyświetlamy wartość zapisaną w zmiennej number
12 Display "Liczba jest równa ", number
13 End Module
14
15 Module changeMe(Integer myValue)
16   Display "Zmieniam wartość parametru."
17
18 // Przypisujemy do parametru myValue
19 // wartość równą 0
20 Set myValue = 0
21
22 // Wyświetlamy wartość przypisaną do parametru myValue
23 Display "Teraz liczba jest równa ", myValue
24 End Module

```

### **Wynik działania programu**

```

Liczba jest równa 99
Zmieniam wartość parametru.
Teraz Liczba jest równa 0
Liczba jest równa 99

```

W linii 2. w module `main` deklaruję zmienną lokalną o nazwie `number` i inicjalizuję ją wartością 99. W rezultacie instrukcja `Display` w linii 5. wyświetli komunikat „Liczba jest równa 99”. Następnie w linii 9. przekazuję do modułu `changeMe` wartość zapisaną w zmiennej `number`. Oznacza to, że wartość 99 zostanie skopiowana w module `changeMe` do parametru `myValue`.

W linii 20. wewnątrz modułu `changeMe` ustawiam wartość parametru `myValue` na 0. W rezultacie instrukcja `Display` w linii 23. wyświetli komunikat „Teraz liczba jest równa 0”. W tym momencie kończy się moduł i program powraca do modułu `main`.

Następnie zostanie wykonane polecenie `Display` w linii 12. Wyświetli ono tekst „Liczba jest równa 99”. Pomiędzy wewnątrz modułu `changeMe` zmieniła się wartość parametru `myValue`, sam argument (zmienna `number` w module `main`) nie został zmodyfikowany.

Dzięki przekazywaniu argumentu jeden moduł może się komunikować z innym modelem. Kiedy przekazujemy argument przez wartość, komunikacja ta ma charakter jednokierunkowy: tylko moduł wywołujący może się komunikować z modelem wywoływanym, moduł wywoływany nie może się komunikować z modelem wywołującym.

### Przekazywanie argumentu przez referencję

Przekazywanie argumentu **przez referencję** oznacza, że argument zostanie przekazany do specjalnego typu parametru zwanego **zmienną typu referencyjnego**. Kiedy użyjemy jako parametru zmiennej typu referencyjnego, moduł będzie mógł modyfikować wartość argumentu zdefiniowanego na zewnątrz tego modułu.

Zmienna typu referencyjnego działa jak alias zmiennej przekazanej jako argument. Słowo „referencyjny” oznacza w tym przypadku, że zmienna ta stanowi referencję (odwołanie) do innej zmiennej. Wszystkie operacje, jakie wykonasz na referencji, zostaną przeniesione na zmienną, na którą ta referencja wskazuje.

Dzięki zmiennym typu referencyjnego możemy między modułami zapewnić komunikację w obie strony. Kiedy jeden moduł wywoła inny moduł i przekaże do niego argument przez referencję, komunikacja między nimi może przebiegać na dwa sposoby:

- moduł wywołujący może komunikować się z modelem wywoływanym, przesyłając do niego argument;
- moduł wywoływany może komunikować się z modelem wywołującym, zmieniając wartość zmiennej przekazanej do modułu przez referencję.

W pseudokodzie będziemy oznaczały, że parametr jest typu referencyjnego, poprzez umieszczenie w nagłówku modułu słowa Ref przed nazwą parametru. Przyjrzymy się przykładowemu modułowi w pseudokodzie:

```
Module setToZero(Integer Ref value)
    Set value = 0
End Module
```

Słowo Ref oznacza tutaj, że parametr value jest typu referencyjnego. Wewnątrz modułu przypisujemy do parametru value wartość 0. Ponieważ parametr value jest typu referencyjnego, operacja ta zmodyfikuje tak naprawdę zmienną przekazaną do modułu jako argument. Na listingu 3.10 pokazałem przykład z wykorzystaniem tego modułu.

#### **Listing 3.10**

```
1 Module main()
2     //Deklarujemy i inicjalizujemy zmienne
3     Declare Integer x = 99
4     Declare Integer y = 100
5     Declare Integer z = 101
6
7     //Wyświetlamy wartości przypisane do zmiennych
8     Display "x jest równe ", x
9     Display "y jest równe ", y
10    Display "z jest równe ", z
11
12    //Przekazujemy każdą ze zmiennych do modułu setToZero
```

```

13 Call setToZero(x)
14 Call setToZero(y)
15 Call setToZero(z)
16
17 // Ponownie wyświetlamy wartości przypisane do zmiennych
18 Display "-----"
19 Display "x jest równe ", x
20 Display "y jest równe ", y
21 Display "z jest równe ", z
22 End Module
23
24 Module setToZero(Integer Ref value)
25   Set value = 0
26 End Module

```

**Wynik działania programu**

x jest równe 99  
 y jest równe 100  
 z jest równe 101

-----  
 x jest równe 0  
 y jest równe 0  
 z jest równe 0

W module `main` zmienną `x` inicjalizujemy wartością 99, zmienną `y` inicjalizujemy wartością 100, a zmienną `z` inicjalizujemy wartością 101. Następnie w liniach od 13. do 15. przekazujemy te zmienne jako argumenty do modułu `setToZero`. Po każdym wywołaniu modułu `setToZero` wartość przekazana do niego jako argument jest ustawiana na 0. Widać to w liniach od 19. do 21., kiedy to wyświetlamy wartości przypisane do tych zmiennych.



**UWAGA:** W prawdziwym programie nie należy używać zmiennych o nazwach takich jak `x`, `y` czy `z`. W tym przypadku jednak program posłużył nam tylko do celów demonstracyjnych i takie nazwy były wystarczające.



**UWAGA:** Zazwyczaj przez referencję można przekazywać tylko zmienne. Gdy spróbujesz przekazać jako argument coś innego niż zmienna, pojawi się błąd. W naszym przykładowym module `setToZero` następująca operacja spowodowałaby błąd:

```
// To jest błąd!
setToZero(5)
```



**OSTRZEŻENIE!** Przekazując argumenty poprzez zmienną typu referencyjnego, zachowaj szczególną ostrożność. Wewnątrz modułu każda modyfikacja zmiennej zadeklarowanej na zewnątrz modułu może być powodem problemów podczas debugowania ewentualnych błędów. Zmiennych typu referencyjnego należy używać tylko wtedy, gdy rzeczywiście jest to konieczne.



## W centrum uwagi

### Przekazywanie argumentu przez referencję

W poprzedniej sekcji „W centrum uwagi” stworzyliśmy program, z którego może korzystać w swojej firmie kateringuowej Michał. Program robi to, czego oczekuje od niego Michał: zamienia liczbę szklanek na liczbę uncji. Jednak gdy przyjrzałeś się bliżej temu programowi, okazało się, że można go jeszcze ulepszyć. Jak pokazałem poniżej, w module `main` znajduje się kod służący do odczytywania danych wprowadzonych przez użytkownika. Ten fragment kodu można jednak potraktować jako osobne podzadanie i umieścić go w osobnym module. Kiedy zmienimy program, będzie można go przedstawić za pomocą nowego schematu hierarchicznego, widocznego na rysunku 3.18.

```

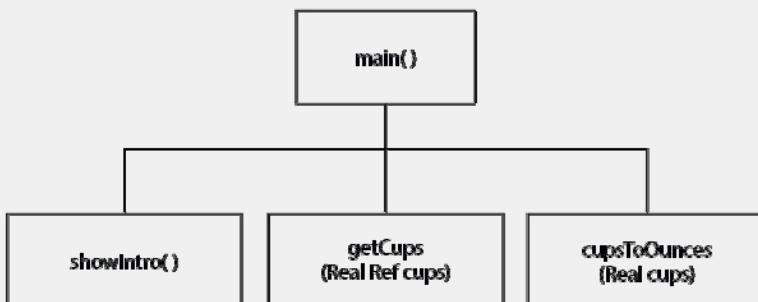
Module main()
    // Deklaracja zmiennej, w której zapisana
    // będzie liczba szklanek danego składnika
    Declare Real cupsNeeded

    // Wyświetlenie komunikatu powitalnego
    Call showIntro()

    // Pobranie liczby szklanek
    Display "Wprowadź liczbę szklanek składnika."
    Input cupsNeeded } Ten fragment kodu można  
umieścić w osobnym module.

    // Zamiana szklanek na uncje
    Call cupsToOunces(cupsNeeded)
End Module

```



**Rysunek 3.18.** Zmodyfikowany schemat hierarchiczny

W poprawionej wersji schematu hierarchicznego można zauważyć nowy moduł: `getCups`. Oto kod deklaracji modułu `getCups`:

```

Module getCups(Real Ref cups)
    Display "Wprowadź liczbę szklanek składnika."
    Input cups
End Module

```

Moduł `getCups` ma jeden parametr, o nazwie `cups`, który jest zmienną typu referencyjnego. W module prosimy użytkownika, aby wprowadził liczbę szklanek danego składnika, i wynik zapisujemy w zmiennej `cups`. Kiedy w module `main` wywołujemy

moduł getCups, przekazujemy do niego jako argument zmienną lokalną cupsNeeded. Ponieważ przekazujemy ten argument przez referencję, po powrocie z modułu będzie w nim zapisana wartość wprowadzona przez użytkownika. Na listingu 3.11 przedstawiłem poprawioną wersję programu, a na rysunku 3.19 widoczny jest schemat blokowy.



**UWAGA:** W tym przykładzie zmodyfikowaliśmy istniejący już program, ale nie zmieniliśmy jego funkcjonalności. Taki proces „uporządkowywania” programu programiści nazywają **refaktoryzacją**.

### Listing 3.11

```

1 Module main()
2   // Deklaracja zmiennej, w której zapisana
3   // będzie liczba szklanek danego składnika
4   Declare Real cupsNeeded
5
6   // Wyświetlenie komunikatu powitalnego
7   Call showIntro()
8
9   // Pobranie liczby szklanek
10  Call getCups(cupsNeeded)
11
12  // Zamiana szklanek na uncje
13  Call cupsToOunces(cupsNeeded)
14 End Module
15
16 // Moduł showIntro wyświetla
17 // komunikat powitalny
18 Module showIntro()
19   Display "Program służy do zamiany szklanek"
20   Display "na uncje. Dla Twojej wiadomości"
21   Display "przelicznik wygląda następująco:"
22   Display "    1 szklanka = 8 uncji."
23 End Module
24
25 // Moduł getCups pobiera od użytkownika liczbę szklanek
26 // i zapisuje ją w zmiennej typu referencyjnego cups
27 Module getCups(Real Ref cups)
28   Display "Wprowadź liczbę szklanek składnika."
29   Input cups
30 End Module
31
32 // Moduł cupsToOunces przyjmuje liczbę szklanek
33 // i wyświetla odpowiadającą jej
34 // liczbę uncji
35 Module cupsToOunces(Real cups)
36   // Deklaracja zmiennej
37   Declare Real ounces
38
39   // Zamiana szklanek na uncje
40   Set ounces = cups * 8
41
42   // Wyświetlenie wyniku.
43   Display "Ta ilość odpowiada ",
44   oounces, " uncjom."
45 End Module

```

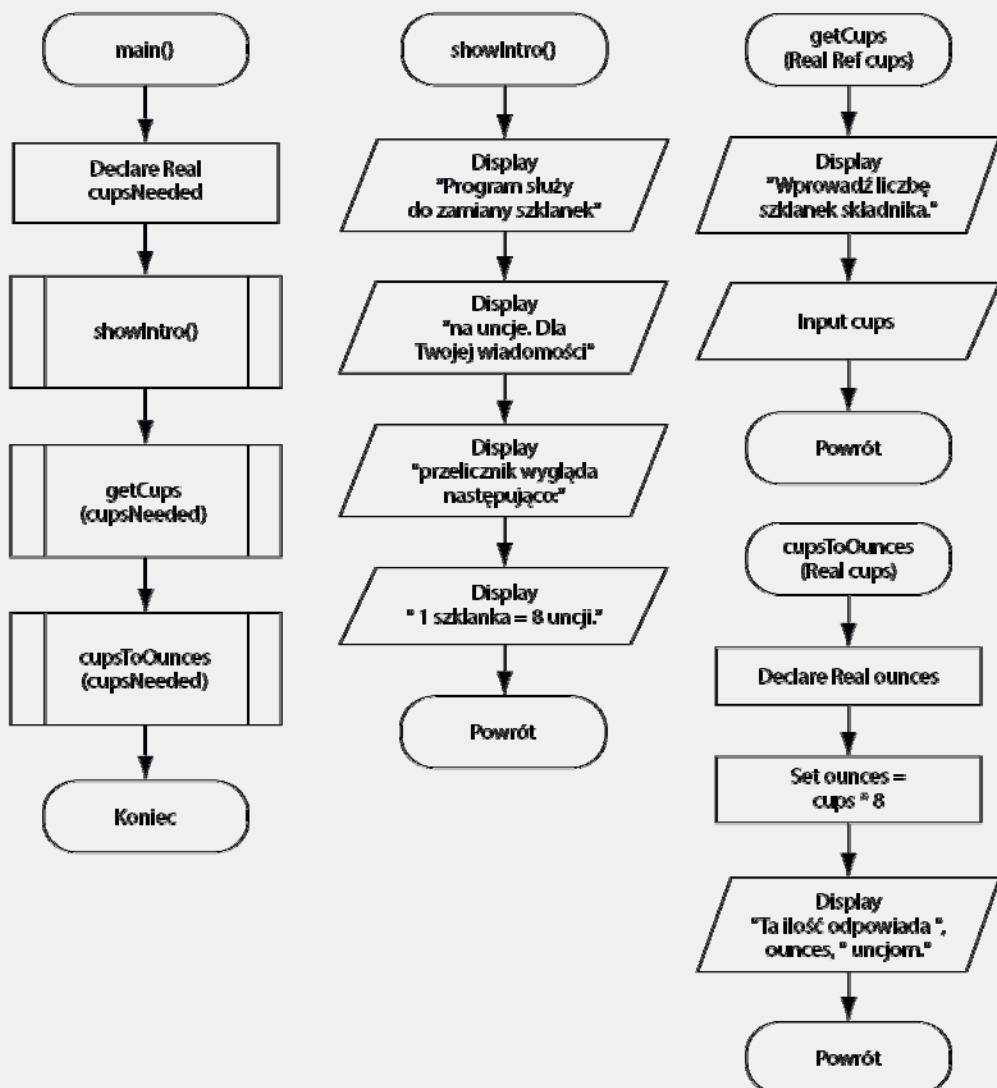
**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Program służy do zamiany szklanek na uncje. Dla Twojej wiadomości przelicznik wygląda następująco:  
1 szklanka = 8 uncji.

Wprowadź liczbę szklanek składnika.

**2 [Enter]**

Ta ilość odpowiada 16 uncjom.



Rysunek 3.19. Schemat blokowy programu z listingu 3.11



## Punkt kontrolny

- 3.14. Jak nazywają się dane, które przekazuje się do modułu?
- 3.15. Jak nazywają się zmienne, do których trafiają dane po wywołaniu modułu?
- 3.16. Czy argument i parametr modułu muszą być tego samego typu?
- 3.17. Jaki obszar programu jest zazwyczaj objęty zasięgiem widoczności parametru?
- 3.18. Wyjaśnij różnicę między przekazywaniem argumentu przez wartość i przez referencję.

3.5

## Zmienne globalne i stałe globalne

**WYJAŚNIENIE:** Zmiennej globalnej można używać we wszystkich modułach w programie.

### Zmienne globalne

Zmienna **globalna** to zmienna, która jest widoczna wewnętrz wstępnych modułów w programie. Zasięg widoczności zmiennej globalnej obejmuje cały program, więc może się do niej odwoływać każdy moduł. W przypadku większości języków programowania zmienną globalną tworzy się poprzez zadeklarowanie jej na zewnątrz wszystkich modułów. Na listingu 3.12 przedstawiłem sposób, w jaki można zadeklarować zmienną globalną w pseudokodzie.

#### Listing 3.12

```

1 // Tutaj znajduje się deklaracja zmiennej globalnej typu Integer
2 Declare Integer number
3
4 // Moduł main
5 Module main()
6     // Pobieramy od użytkownika liczbę
7     // i zapisujemy ją w zmiennej globalnej
8     Display "Wprowadź liczbę."
9     Input number
10
11    // Wywołanie modułu showNumber
12    Call showNumber()
13 End Module
14
15 // Moduł showNumber wyświetla wartość
16 // przypisaną do zmiennej globalnej number
17 Module showNumber()
18     Display "Wprowadzona przez Ciebie liczba to ", number
19 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę.

**22 [Enter]**

Wprowadzona przez Ciebie liczba to 22

W linii 2. deklaruję zmienną typu `Integer` o nazwie `number`. Ponieważ nie zadeklarowałem jej w żadnym konkretnym module, jest ona zmienną globalną. Mają do niej dostęp wszystkie zdefiniowane w programie moduły. Po wykonaniu instrukcji `Input` w linii 9. (w module `main`) wartość wprowadzona przez użytkownika zostanie przypisana do zmiennej globalnej `number`. W linii 18. pojawia się polecenie `Display` (w module `showNumber`) i po jego wywołaniu wyświetli się wartość przypisana do tej samej zmiennej globalnej.

Większość programistów jest jednak zgodna co do tego, że ze zmiennych globalnych należy korzystać z umiarem, a nawet nie korzystać z nich w ogóle. A oto kilka powodów takiego stanu rzeczy:

- Zmienne globalne utrudniają debugowanie programu. Wartość przypisaną do zmiennej globalnej może zmienić każde polecenie w programie. Jeśli okaże się, że w zmiennej globalnej jest nie taka wartość, jakiej się spodziewaliśmy, będziemy musieli prześledzić każdą instrukcję, która może modyfikować tę wartość, aby znaleźć miejsce, w którym jest błąd. Może to być kłopotliwe, zwłaszcza jeżeli program składa się z kilku tysięcy linii kodu.
- Moduły, które korzystają ze zmiennych globalnych, są od nich zależne. Jeśli zechcesz taki moduł wykorzystać w innym programie, najprawdopodobniej będziesz go musiał przeprojektować, aby nie odwoływał się do tej zmiennej globalnej.
- Zmienne globalne sprawiają, że program jest mniej zrozumiały. Zmienną globalną może zmodyfikować każde polecenie w programie. Jeśli będziesz chciał zrozumieć fragment kodu, w którym pojawia się odwołanie do zmiennej globalnej, będziesz też musiał sprawdzić inne miejsca w programie, w których następuje odwołanie do tej zmiennej.

W znaczącej większości przypadków powinieneś deklarować zmienne jako lokalne i przekazywać je do modułów, w których będziesz chciał się do nich odwołać.

## Stałe globalne

Należy unikać stosowania w programie zmiennych globalnych, jednak używanie stałych globalnych jest jak najbardziej wskazane. **Stała globalna** to taka stała nazwana, do której ma dostęp każdy moduł w programie. Ponieważ nie da się zmienić w programie wartości przypisanej do stałej, nie musisz się martwić zagrożeniami wynikającymi z korzystania ze zmiennych globalnych.

Stałych globalnych używa się najczęściej, aby zdefiniować pewne niezmienne wartości, z których będziemy korzystać w całym programie. Przykładowo możesz sobie wyobrazić, że w programie bankowym będzie potrzebna stała nazwana, w której zapisana będzie wartość oprocentowania. Jeśli z wartości tej będziemy korzystać w wielu modułach, łatwiej będzie nam utworzyć stałą globalną niż lokalną stałą w każdym module. Dzięki temu łatwiej będzie modyfikować program — jeżeli wartość oprocentowania ulegnie zmianie, wystarczy, że zmienimy tylko wartość przypisaną do zmiennej globalnej.



## W centrum uwagi

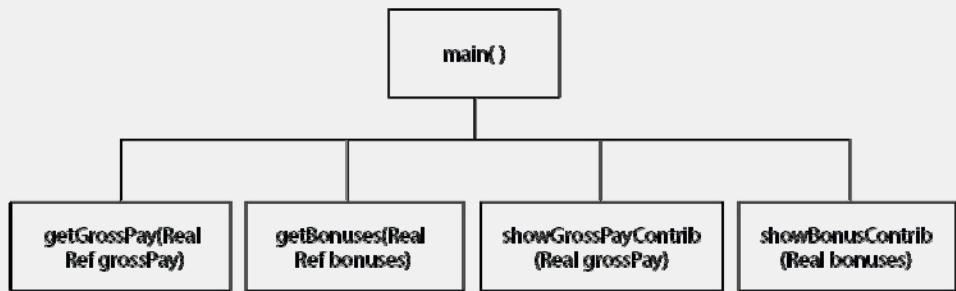
### Korzystanie ze stałych globalnych

Marysia pracuje w Integrated Systems Inc. Jest to przedsiębiorstwo znane ze swoich rozbudowanych świadczeń socjalnych. Jednym ze świadczeń jest premia kwartalna wyplacana wszystkim pracownikom. Innym świadczeniem są składki wpłacane na fundusz emerytalny. Firma przekazuje na fundusz emerytalny 5% kwoty wynagrodzenia i premii. Marysia chce zaprojektować program, który będzie obliczał wartość składek na fundusz emerytalny wpłaconych w ciągu roku. Chce także, aby program wyświetlał osobno wartość składki naliczoną od wynagrodzenia i od premii.

Oto algorytm tego programu:

1. Pobierz roczne wynagrodzenie pracownika.
2. Pobierz wartość premii wpłaconych pracownikowi.
3. Oblicz i wyświetl wartość składki naliczonej od wynagrodzenia.
4. Oblicz i wyświetl wartość składki naliczonej od premii.

Na rysunku 3.20 przedstawiłem schemat hierarchiczny programu. Pseudokod programu znajduje się na listingu 3.13, a schematy blokowe widoczne są na rysunku 3.21.



Rysunek 3.20. Schemat hierarchiczny

#### **Listing 3.13**

```

1 // Stała globalna zawierająca procentową wysokość składki
2 Constant Real CONTRIBUTION_RATE = 0.05
3
4 // Moduł main
5 Module main()
6   // Zmienne lokalne
7   Declare Real annualGrossPay
8   Declare Real totalBonuses
9
10  // Pobranie wynagrodzenia pracownika
11  Call getGrossPay(annualGrossPay)
12
13  // Pobranie wartości wpłaconych premii
14  Call getBonuses(totalBonuses)
15
16  // Wyświetlenie składki naliczonej
17  // od wynagrodzenia
18  Call showGrossPayContrib(annualGrossPay)
  
```

```

19 // Wyświetlenie składki naliczonej
20 // od premii
21 Call showBonusContrib(totalBonuses)
22 End Module
23
24
25 // Moduł getGrossPay pobiera od użytkownika
26 // wartość wynagrodzenia i zapisuje ją
27 // w zmiennej typu referencyjnego grossPay
28 Module getGrossPay(Real Ref grossPay)
29   Display "Wprowadź wartość wynagrodzenia."
30   Input grossPay
31 End Module
32
33 // Moduł getBonuses pobiera od użytkownika
34 // wartość wypłaconych premii i zapisuje ją
35 // w zmiennej typu referencyjnego bonuses
36 Module getBonuses(Real Ref bonuses)
37   Display "Wprowadź wartość wypłaconych premii."
38   Input bonuses
39 End Module
40
41 // Moduł showGrossPayContrib
42 // przyjmuje jako argument wartość wynagrodzenia
43 // i wyświetla wartość składki naliczonej
44 // od wynagrodzenia
45 Module showGrossPayContrib(Real grossPay)
46   Declare Real contrib
47   Set contrib = grossPay * CONTRIBUTION_RATE
48   Display "Wartość składki naliczona od wynagrodzenia"
49   Display "wynosi ", contrib, " zł."
50 End Module
51
52 // Moduł showBonusContrib
53 // przyjmuje jako argument wartość wypłaconych premii
54 // i wyświetla wartość składki naliczonej
55 // od premii.
56 Module showBonusContrib(Real bonuses)
57   Declare Real contrib
58   Set contrib = bonuses * CONTRIBUTION_RATE
59   Display "Wartość składki naliczona od premii"
60   Display "wynosi ", contrib, " zł."
61 End Module

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wartość wynagrodzenia.

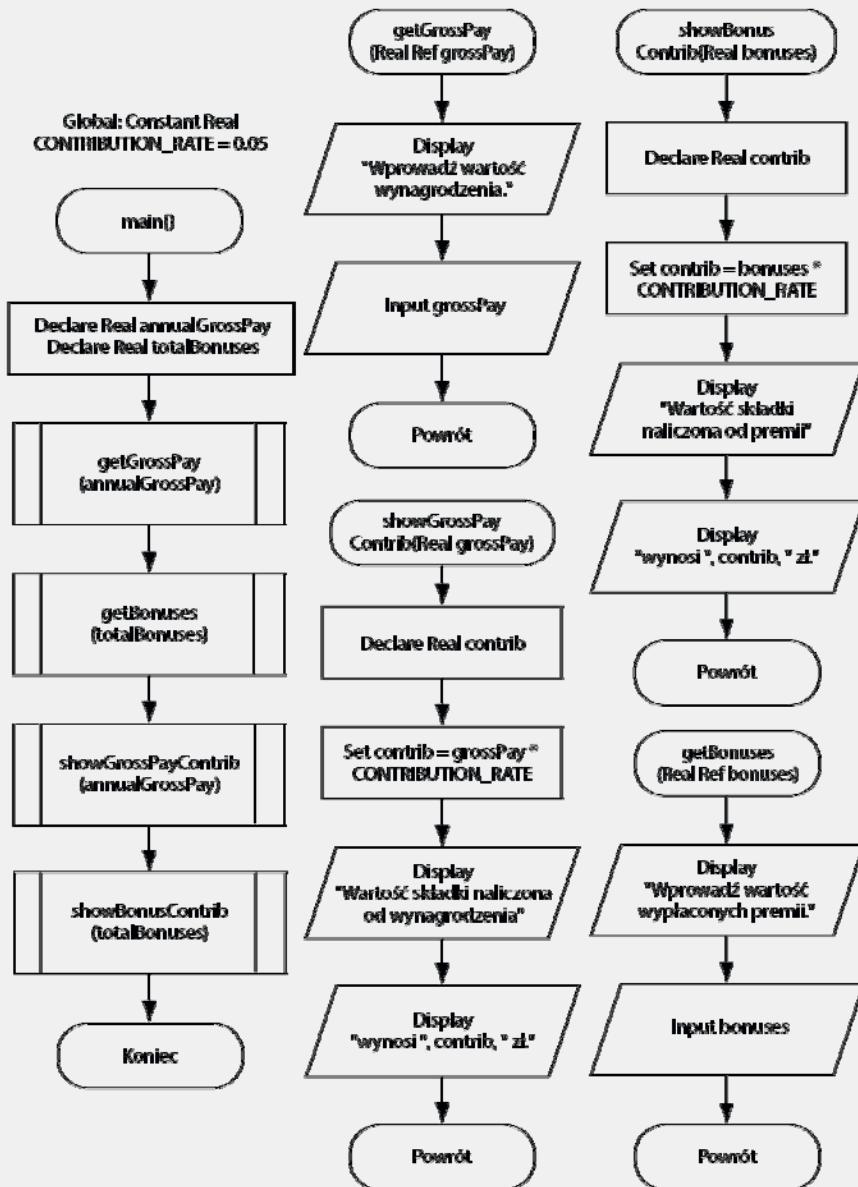
**80000.00 [Enter]**

Wprowadź wartość wypłaconych premii.

**20000.00 [Enter]**

Wartość składki naliczona od wynagrodzenia  
wynosi 4000 zł.

Wartość składki naliczona od premii  
wynosi 1000 zł.



Rysunek 3.21. Schemat blokowy programu z listingu 3.13

W linii 2. zadeklarowałem stałą globalną o nazwie `CONTRIBUTION_RATE` i zainicjalizowałem ją wartością 0.05. Stała ta jest wykorzystywana w obliczeniach widocznych w liniach 47. (w module `showGrossPayContrib`) i 58. (w module `showBonusContrib`). Marysia postanowiła skorzystać ze stałej globalnej z dwóch powodów:

- Kod jest bardziej czytelny. Kiedy spojrzysz na obliczenia w liniach 47. i 58., od razu wiadomo, jaką wartość się w tych miejscach oblicza.

- Co pewien czas będzie się zmieniała procentowa wartość składki na fundusz emerytalny. Kiedy tak się stanie, modyfikacja programu będzie bardzo prosta — wystarczy zmienić wartość w linii 2.



## Punkt kontrolny

- 3.19. Jakim zasięgiem widoczności charakteryzują się zmienne globalne?
- 3.20. Wymień jeden powód, dla którego nie warto używać w programie zmiennych globalnych.
- 3.21. Co to jest stała globalna? Czy w programie powinno się używać stałych globalnych?

**3.6**

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ wielu zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

### Java

#### Modularyzowanie programów w języku Java

##### Programy z metodami

W Javie moduły są nazywane **metodami**. W celu utworzenia metody musisz napisać jej **definicję**, która składa się z dwóch części ogólnych: nagłówka i ciała. **Nagłówek metody** jest wierszem pojawiającym się na początku definicji metody. W wierszu tym podawanych jest kilka informacji na temat metody, w tym jej nazwa. **Ciało metody** jest zbiorem instrukcji, które są przetwarzane podczas jej wykonywania. Te instrukcje są zawarte w nawiasach klamrowych.

Jak wiesz, każdy kompletny program w języku Java musi mieć metodę **main**. Programy napisane w Javie mogą zawierać również inne metody. Oto przykład prostej metody wyświetlającej komunikat na ekranie:

```
public static void showMessage()
{
    System.out.println("Witaj, świecie!");
}
```



Na razie nagłówki wszystkich metod w Javie, które napiszesz, będą się zaczynać od słów kluczowych `public static void`. Następnie napisz nazwę metody i nawiasy okrągłe. Pamiętaj jednak, że nagłówek metody nigdy nie kończy się średnikiem!

### **Wywoływanie metody w języku Java**

W momencie wywoływania metody program przechodzi do niej i wykonuje instrukcje zapisane w jej ciele. Oto przykład instrukcji wywołującej metodę `showMessage`, której już wcześniej się przyglądaliśmy:

```
showMessage();
```

Instrukcja ta jest po prostu nazwą metody, po której następują nawiasy okrągłe. Jest to już cała instrukcja, dlatego kończy się średnikiem.

### **Zmienne lokalne**

Zmienne zadeklarowane wewnętrz metody nazywane są zmiennymi lokalnymi. „Lokalne”, ponieważ są dostępne lokalnie w ramach metody, w której zostały zadeklarowane. Instrukcje znajdujące się poza tą metodą nie mogą uzyskać dostępu do jej zmiennych lokalnych.

### **Przekazywanie argumentów do metod**

W celu przekazania argumentu do metody w jej nagłówku trzeba zadeklarować zmienną parametru. Zmienna parametru otrzyma argument, który jest jej przekazywany przy wywoływaniu metody. Oto definicja metody, która korzysta z parametru:

```
public static void displayValue(int num)
{
    System.out.println("Wartość wynosi " + num);
```

Zwróć uwagę na deklarację zmiennej typu `int`, która pojawia się w nawiasach (`int num`). Jest to deklaracja zmiennej parametru, która pozwala metodzie `displayValue` przyjmować wartość całkowitą jako argument. Oto przykład wywołania metody `displayValue` przekazujący jej jako argument liczbę 5:

```
displayValue(5);
```

Ta instrukcja powoduje wykonanie metody `displayValue`. Argument zawarty w nawiasach jest przypisywany do zmiennej parametru metody — `num`.

### **Przekazywanie wielu argumentów**

Często przydatne jest przekazanie więcej niż jednego argumentu do metody. Podczas definiowania metody należy zadeklarować zmienną parametru dla każdego argumentu, który ma zostać do niej przekazany. Deklaracje parametrów są oddzielone przecinkami. Oto przykład:

```
public static void showSum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    System.out.println(result);
}
```

Oto jak możemy wywołać tę metodę:

```
showSum(12, 45);
```

Ta instrukcja wywołuje metodę `showSum` i przekazuje jej dwa argumenty: 12 i 45. Te argumenty są przekazywane **przez pozycję** do odpowiednich zmiennych parametrów funkcji. Innymi słowy, pierwszy argument jest przekazywany do pierwszej zmiennej parametru, a drugi argument jest przekazywany do drugiej zmiennej parametru. Tak więc ta instrukcja powoduje, że wartość 12 zostanie przypisana do parametru `num1`, a wartość 45 do parametru `num2`.

### **W języku Java argumenty przekazywane są przez wartość**

W Javie wszystkie argumenty wbudowanych typów danych są przekazywane **przez wartość**, co oznacza, że jedynie kopia wartości argumentu jest przekazywana do zmiennej parametru. Zmienne parametrów metody są oddzielne i niezależne od argumentów wymienionych w nawiasach wywołania metody. Jeśli zmienią parametru zostanie zmieniona wewnętrz metody, nie ma to wpływu na pierwotny argument.

### **Stałe globalne**

W Javie zmienne i stałe nie mogą być zadeklarowane poza klasą. Jeśli zadeklarujesz zmienną lub stałą w klasie, ale poza wszystkimi metodami klasy, ta zmienna lub stała jest znana jako **pole klasy** i będzie dostępna dla wszystkich metod w klasie. Na listingu 3.14 przedstawiam sposób, w jaki można zadeklarować taką stałą. Zauważ, że wierszu 3. zadeklarowaliśmy stałą o nazwie `VALUE`. Deklaracja znajduje się wewnątrz klasy, ale nie znajduje się w żadnej z tych metod. W rezultacie stała jest dostępna dla wszystkich metod w klasie. Zauważ też, że deklaracja zaczyna się od słów kluczowych `public static`. Na tym etapie już nie musisz się zastanawiać, dlaczego użyto tutaj tych słów kluczowych. Ważne jest tylko to, że dzięki temu stała dostępna jest dla wszystkich metod oznaczonych słowami kluczowymi `public static`.

#### **Listing 3.14. (FieldTest.java)**

```

1 public class FieldTest
2 {
3     public static final int VALUE = 10;
4
5     public static void main(String[] args)
6     {
7         // Znajdujące się tutaj instrukcje mają
8         // dostęp do stałej VALUE
9     }
10
11    public static void method2()
12    {
13        // Znajdujące się tutaj instrukcje mają
14        // dostęp do stałej VALUE
15    }
16
17    public static void method3()
18    {
19        // Znajdujące się tutaj instrukcje mają

```

```

20      // dostęp do stałej VALUE
21  }
22 }
```

## Python

### Modularyzowanie programów w języku Python za pomocą funkcji

W tym rozdziale omówilem moduły, czyli nazwane grupy instrukcji wykonujące określone zadania w programie. Takie moduły wykorzystujemy w celu rozbicia programu na mniejsze, łatwe w zarządzaniu części. W Pythonie w tym celu używamy **funkcji**. (W Pythonie termin „moduł” ma nieco inne znaczenie. Moduł Pythona to plik zawierający zestaw powiązanych ze sobą elementów programu, takich jak funkcje).

#### Definiowanie i wywoływanie funkcji w języku Python

W celu utworzenia funkcji zapisujemy jej **definicję**. Oto ogólny format definicji funkcji w Pythonie:

```

def nazwa_funkcji():
    instrukcja
    instrukcja
    itd.
```

Pierwszy wiersz jest **nagłówkiem funkcji**. Oznacza początek definicji funkcji. Nagłówek funkcji rozpoczyna się od słowa kluczowego `def`, po którym następuje nazwa funkcji, a po niej nawiasy i dwukropki.

Od następnego wiersza zaczyna się zbiór instrukcji zwany blokiem. **Blok** to po prostu zestaw instrukcji, które należą do jednej grupy. Takie instrukcje są wykonywane za każdym razem, gdy funkcja jest wywoływana. Zwróć uwagę na ogólną zasadę, że wszystkie instrukcje w bloku są wcięte. Te wcięcia są wymagane, ponieważ interpreter Pythona używa ich do wskazania, gdzie zaczyna się, a gdzie kończy blok. Oto przykład definicji funkcji:

```

def message():
    print('Jestem Artur,')
    print('król Brytanii.')
```

Ten kod definiuje funkcję o nazwie `message`, która zawiera blok z dwiema instrukcjami. Wywołanie tej funkcji spowoduje wykonanie obu instrukcji.

#### Wywołanie funkcji w języku Python

Definicja funkcji określa, co zawiera ta funkcja, ale nie powoduje jej wykonania. Chcąc uruchomić tę funkcję, trzeba ją wywołać. Funkcję `message` można wywołać w ten sposób:

```
message()
```

Podczas wywołania funkcji interpreter przechodzi do niej i wykonuje instrukcje w jej bloku. Następnie, po osiągnięciu końca bloku, interpreter wraca do części programu, która wywołała funkcję, i wznowia działanie w tym miejscu. Taka operacja nazywana jest **powrotem z funkcji**. Przykład takiego działania prezentuję na listingu 3.15.

### **Listing 3.15. (function\_demo.py)**

```
1 def message():
2     print('Jestem Artur,')
3     print('król Brytanii.')
4
5 message()
```

#### **Wynik działania programu**

```
Jestem Artur,
król Brytanii
```

W momencie, gdy interpreter Pythona odczytuje instrukcję `def` znajdującą się wierszu 1., w pamięci tworzona jest funkcja o nazwie `message`, która zawiera blok instrukcji w wierszach 2. i 3. (Definicja funkcji tworzy funkcję, ale nie powoduje jej wykonania). Następnie interpreter wykonuje instrukcję w wierszu 5., która jest wywołaniem funkcji. Powoduje to wykonanie funkcji `message`, która wypisuje na ekranie dwa wiersze tekstu.

### **Wcięcia w języku Python**

W Pythonie każdy wiersz w bloku musi mieć wcięcie. Jak pokazałem na rysunku 3.22, ostatni wiersz z wcięciem za nagłówkiem funkcji jest ostatnim wierszem w bloku funkcji.

```
def greeting():
    print('Dzień dobry!')
    print('Dzisiaj będziemy poznawać funkcje.')
print('Wywołam funkcję greetings.')
greeting()
```

Ostatni wiersz z wcięciem  
jest ostatnim wierszem w bloku

Te instrukcje nie znajdują się już w bloku

**Rysunek 3.22.** Wszystkie instrukcje w bloku Pythona są wcięte

Podczas tworzenia wcięć w bloku należy się upewnić, że każdy wiersz zaczyna się od tej samej liczby spacji. W przeciwnym razie wystąpi błąd.

### **Zmienne lokalne w języku Python**

Za każdym razem w momencie przypisania wartości do zmiennej wewnętrz funkcji tworzymy **zmienną lokalną**. Zmienna lokalna należy do funkcji, w której jest tworzona, i tylko instrukcje wewnątrz tej funkcji mogą uzyskać do niej dostęp. Zakresem zmiennej lokalnej jest funkcja, w której tworzona jest ta zmienna.

### Przekazywanie argumentów do funkcji w języku Python

Jeśli chcesz, aby funkcja w Pythonie otrzymywała argumenty przy wywołaniu, musisz wyposażyć ją w co najmniej jedną zmienną parametru. Oto przykład funkcji, która ma zmienną parametru:

```
def double_number(value):
    result = value * 2
    print(result)
```

Spójrz na nagłówek funkcji i zauważ, że słowo `value` pojawia się w nawiasach. Jest to nazwa zmiennej parametru. Ta zmienna zostanie przypisana do wartości argumentu, gdy funkcja zostanie wywołana. Na listingu 3.16 przedstawiam działanie funkcji w programie.

#### **Listing 3.16. (pass\_integer.py)**

```
1 # Definicja funkcji main
2 def main():
3     double_number(4)
4
5 # Definicja funkcji double_number
6 def double_number(value):
7     result = value * 2
8     print(result)
9
10 # Wywołanie funkcji main
11 main()
```

#### **Wynik działania programu**

```
8
```

Po uruchomieniu programu w wierszu 11. wywoływana jest funkcja `main`. Wewnątrz tej funkcji, w wierszu 3., wywoywana jest funkcja `double_number`, razem z wartością 4 jako argumentem.

Funkcja `double_number` jest zdefiniowana w wierszach od 6. do 8. Funkcja ta ma zmienną parametru o nazwie `value`. W wierszu 7. lokalnej zmiennej o nazwie `result` przypisywane jest wyrażenie matematyczne `value * 2`. Natomiast w wierszu 8. wyświetlana jest wartość zmiennej `result`.

### Przekazywanie wielu argumentów w języku Python

Często przydatne jest przekazanie więcej niż jednego argumentu do funkcji. W momencie definiowania funkcji musisz mieć zmienną parametru dla każdego argumentu, który chcesz do niej przekazać. Oto przykład:

```
def show_sum(num1, num2):
    result = num1 + num2
    print(result)
```

Zauważ, że dwie nazwy zmiennych parametrów, `num1` i `num2`, pojawiają się w nawiasach w nagłówku funkcji i są oddzielone przecinkiem. Oto jak możemy wywołać taką funkcję:

```
show_sum(12, 45)
```

Taka instrukcja wywołuje funkcję `show_sum` i przekazuje dwa argumenty: 12 i 45. Te argumenty są przekazywane **przez pozycję** do odpowiednich zmiennych parametrów w funkcji. Innymi słowy, pierwszy argument jest przekazywany do pierwszej zmiennej parametru, a drugi argument jest przekazywany do drugiej zmiennej parametru. Tak więc ta instrukcja powoduje, że wartość 12 zostanie przypisana do parametru `num1`, a wartość 45 do parametru `num2`.

### **Wprowadzanie zmian w parametrach w języku Python**

Podczas przekazywania argumentu do funkcji w Pythonie zmiennej parametru funkcji przypisywana jest wartość argumentu. Jednak wszelkie zmiany wprowadzone w zmiennej parametru nie będą miały wpływu na argument. Spójrz na listing 3.17.

#### **Listing 3.17. (change\_me.py)**

```

1 def main():
2     value = 99
3     print('Wartość wynosi ', value)
4     change_me(value)
5     print('Znowu w funkcji main wartość wynosi ', value)
6
7 def change_me(arg):
8     print('Zmieniam wartość!')
9     arg = 0
10    print('Teraz wartość wynosi', arg)
11
12 main()

```

#### **Wynik działania programu**

```

Wartość wynosi 99
Zmieniam wartość!
Teraz wartość wynosi 0
Znowu w funkcji main wartość wynosi 99

```

W wierszu 2. funkcja `main` tworzy lokalną zmienną o nazwie `value`, której przypisana jest wartość 99. W wierszu 3. wyświetlana jest **wartość** zmiennej `value`, czyli liczba 99. Zmienna `value` jest następnie przekazywana jako argument do funkcji `change_me` w wierszu 4. Oznacza to, że w funkcji `change_me` parametrowi `arg` również zostanie przypisana wartość 99. Po wykonaniu instrukcji z wiersza 9. parametrowi `arg` zostanie przypisana nowa wartość 0. W tym punkcie wykonania programu w zmiennej `value` nadal zapisana będzie wartość 99, natomiast parametrowi `arg` zostanie przypisana wartość 0.

### **Zmienne globalne w języku Python**

W Pythonie zmienną nazywamy *globalną*, gdy jest tworzona za pomocą instrukcji przypisania i jednocześnie jest zapisana poza wszystkimi funkcjami w pliku programu. Do zmiennej globalnej może uzyskać dostęp dowolna instrukcja w pliku programu, w tym instrukcja dowolnej funkcji. Przykład znajduje się na listingu 3.18.

**Listing 3.18.** (global1.py)

```

1 # Tworzenie zmiennej globalnej
2 my_value = 10
3
4 # Funkcja show_value wypisuje
5 # wartość zmiennej globalnej
6 def show_value():
7     print(my_value)
8
9 # Wywołanie funkcji show_value
10 show_value()

```

**Wynik działania programu**

10

Instrukcja przypisania w wierszu 2. tworzy zmienną o nazwie `my_value`. Ta definicja znajduje się poza jakkolwiek funkcją, dzięki czemu jest ona zmienną globalną. Podczas wykonywania funkcji `show_value` instrukcja z wiersza 7. wypisuje wartość, do której odwołuje się zmienna `my_value`.

Jeśli chcesz, aby instrukcja w funkcji przypisywała wartość do zmiennej globalnej, musisz zdefiniować dodatkowy krok. W funkcji należy zadeklarować zmienną globalną, tak jak pokazano na listingu 3.19.

**Listing 3.19.** (global2.py)

```

1 # Tworzenie zmiennej globalnej
2 number = 0
3
4 def main():
5     global number
6     number = int(input('Wprowadź liczbę: '))
7     show_number()
8
9 def show_number():
10    print('Podano liczbę ', number)
11
12 # Wywołanie funkcji main
13 main()

```

**Wynik działania programu**

Wprowadź liczbę: 22 [Enter]

Podano liczbę 22

Instrukcja przypisania w wierszu 2. tworzy globalną zmienną o nazwie `number`. Zwróć uwagę, że w wierszu 5. w funkcji `main` używane jest słowo kluczowe `global` w celu zadeklarowania zmiennej `number`. Ta instrukcja informuje interpreter, że funkcja `main` ma zamiar przypisać wartość globalnej zmiennej `number`. Samo przypisanie wykonywane jest w wierszu 6. Wartość wprowadzona przez użytkownika jest przypisana do zmiennej `number`.

## Stałe globalne w języku Python

Python nie pozwala na tworzenie prawdziwych stałych globalnych, ale można je symułować za pomocą zmiennych globalnych. Jeżeli w funkcji nie zadeklarujesz zmiennej globalnej za pomocą słowa kluczowego `global`, to zmiana wartości tej zmiennej nie będzie możliwa.

## C++

### Modularyzowanie funkcji i programów w języku C++

W niniejszym rozdziale omówilem moduły jako nazywane grupy instrukcji wykonujących określone zadania w programie. Modułów używa się w celu podzielenia programu na małe, łatwe w zarządzaniu jednostki. W języku C++ w tym celu wykorzystywane są **funkcje**.

#### Definiowanie i wywoływanie funkcji w języku C++

Aby utworzyć funkcję w C++, należy napisać jej **definicję**. Oto ogólny format definicji funkcji:

```
void showMessage()
{
    cout << "Witaj, świecie!" << endl;
```

Na razie nagłówki wszystkich funkcji języka C++, które tutaj napiszesz, będą zaczynać się od słowa kluczowego `void`. Następnie należy wprowadzić nazwę funkcji i zestaw nawiasów. Pamiętaj, że nagłówek funkcji nigdy nie kończy się średnikiem!

#### Wywołanie funkcji

Funkcja jest wykonywana po jej wywołaniu. Funkcja `main` jest wywoływana automatycznie po uruchomieniu programu, ale inne funkcje są wykonywane jedynie przez instrukcje wywołania funkcji. Kiedy funkcja jest wywoływana, program przechodzi do niej i wykonuje instrukcje w jej wnętrzu. Oto przykład instrukcji wywołania funkcji, która wywołuje funkcję `showMessage`:

```
showMessage();
```

Instrukcja jest po prostu nazwą funkcji, po której następują nawiasy okrągłe. Kończy się ona średnikiem, ponieważ jest to już pełna instrukcja. Na listingu 3.20 przedstawiam program w języku C++ zawierający funkcję `showMessage`.

#### **Listing 3.20. (FunctionDemo.cpp)**

```
1 #include <iostream>
2 using namespace std;
3
4 void showMessage();
```

```

5
6 int main()
7 {
8     cout << "Mam dla Ciebie wiadomość." << endl
9     showMessage();
10    cout << "To wszystko!" << endl;
11    return 0;
12 }
13
14 void showMessage()
15 {
16     cout << "Witaj, świecie!" << endl;
17 }
```

### **Wynik działania programu**

Mam dla Ciebie wiadomość.  
Witaj, świecie!  
To wszystko!

Program składa się z dwóch funkcji: `main` i `showMessage`. Funkcja `main` pojawia się wierszach od 6. do 12., a funkcja `showMessage` zapisana jest wierszach od 14. do 17. W momencie uruchomienia programu wykonywana jest funkcja `main`. Instrukcja wierszu 8. wyświetli komunikat *Mam dla Ciebie wiadomość*. Następnie instrukcja z wiersza 9. wywoła funkcję `showMessage`. Spowoduje to, że program przejdzie do funkcji `showMessage` i wykona instrukcję, która pojawia się wierszu 16. Na ekranie pojawi się wtedy tekst *Witaj, świecie!*. Następnie program wraca do funkcji `main` i wznowia wykonywanie od instrukcji wierszu 10. Znajdująca się tu instrukcja wyświetla zdanie *To wszystko!*.

Zwróć uwagę na instrukcję, który pojawia się wierszu 4.:

```
void showMessage();
```

Ten wiersz kodu jest prototypem funkcji. **Prototyp funkcji** to instrukcja, która deklaruje istnienie funkcji, ale jej nie definiuje. Jest to jedynie sposób poinformowania kompilatora, że dana funkcja istnieje w programie, a jej definicja pojawia się później. Bez tej instrukcji wystąpiłby błąd podczas komplikacji programu.

### **Zmienne lokalne w języku C++**

Zmienne zadeklarowane w funkcji są znane jako zmienne lokalne. „Lokalne”, ponieważ są dostępne lokalnie w funkcji, w której zostały zadeklarowane. Wyrażenia poza funkcją nie mogą uzyskać dostępu do zmiennych lokalnych tej funkcji. Dzięki temu, że zmienne lokalne funkcji są ukryte przed innymi funkcjami, pozostałe funkcje mogą mieć własne zmienne lokalne o tej samej nazwie.

### **Przekazywanie argumentów do funkcji w języku C++**

Jeśli chcesz przekazać argument do funkcji w języku C++, musisz zadeklarować zmienną parametru w nagłówku tej funkcji. Zmienna parametru otrzyma argument, który zostanie jej przekazany, gdy funkcja będzie wywoływana. Oto definicja funkcji, która wykorzystuje taki parametr:

```
void displayValue(int num)
{
    cout << "Wartość wynosi " << num << endl;
}
```

Zwróć uwagę na deklarację zmiennej typu int, która pojawia się w nawiasach: (int num). Jest to deklaracja zmiennej parametru, która umożliwia funkcji `displayValue` przyjęcie wartości całkowitej jako argumentu. Oto przykład wywołania funkcji `displayValue` przekazującego wartość 5 jako argument:

```
displayValue(5);
```

Ta instrukcja wywołuje funkcję `displayValue`. Argument podany w nawiasach jest kopiowany do zmiennej parametru o nazwie num.

### **Przekazywanie wielu argumentów w języku C++**

Często przydatne jest przekazanie więcej niż jednego argumentu do funkcji. Podczas definiowania funkcji należy zadeklarować zmienną parametru dla każdego argumentu, który ma zostać przekazany do metody. Deklaracje parametrów są oddzielone przecinkami. Oto przykład:

```
void showSum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    cout << result << endl;
}
```

A oto przykład tego, jak możemy wywołać tę metodę:

```
showSum(12, 45);
```

Ta instrukcja wywołuje metodę `showSum` i przekazuje dwa argumenty: 12 i 45. Te argumenty są przekazywane **przez pozycję** do odpowiednich zmiennych parametrów w funkcji. Innymi słowy, pierwszy argument jest przekazywany do pierwszej zmiennej parametru, a drugi argument jest przekazywany do drugiej zmiennej parametru. A zatem ta instrukcja powoduje, że argument 12 zostanie przypisany do parametru `num1`, a argument 45 do parametru `num2`.

### **Przekazywanie argumentów przez referencję w języku C++**

Gdy argument jest przekazywany przez referencję, oznacza to, że funkcja ma dostęp do tego argumentu i może wprowadzać w nim zmiany. Język C++ udostępnia specjalny typ zmiennej zwany **zmienną referencyjną**, która po użyciu jako parametr funkcji umożliwia dostęp do pierwotnego argumentu.

Zmienna referencyjna jest aliasem dla innej zmiennej. Wszelkie zmiany dokonane w zmiennej referencyjnej są faktycznie wykonywane na zmiennej, dla której jest to alias. Dzięki zastosowaniu zmiennej referencyjnej jako parametru funkcja może zmodyfikować zmienną zdefiniowaną w innej funkcji.

Zmienne referencyjne są deklarowane tak jak zwykłe zmienne, ale przed ich nazwą umieszcza się ampersand (&). Spójrz na przykład przedstawiający funkcję `setToZero`:

```

void setToZero(int &num)
{
    num = 0;
}

```

Funkcja `setToZero` ustawia zmienną parametru na wartość 0, co również ustawia oryginalną zmienną, która została przekazana jako argument do wartości 0.

### Zmienne globalne i stałe globalne w języku C++

W celu zadeklarowania zmiennej globalnej lub stałej globalnej w C++ należy zapisać deklarację poza wszystkimi funkcjami i nad definicjami funkcji. W rezultacie wszystkie funkcje programu będą miały dostęp do zmiennej lub stałej.

WCześniej w tym rozdziale ostrzegałem przed użyciem zmiennych globalnych, ponieważ utrudniają one debugowanie programów. Stałe globalne są jednak dopuszczalne, gdyż instrukcje programu nie mogą zmienić ich wartości. Na listingu 3.21 przedstawiam, jak należy zadeklarować taką stałą. Zauważ, że w wierszu 9. zadeklarowana została stała o nazwie `INTEREST_RATE`. Deklaracja nie znajduje się w żadnej z funkcji, ale jest zapisana ponad wszystkimi funkcjami. W rezultacie stała jest dostępna dla wszystkich funkcji w programie.

#### **Listing 3.21. (GlobalConstants.cpp)**

```

1 #include <iostream>
2 using namespace std;
3
4 // Prototypy funkcji
5 void function2();
6 void function3();
7
8 // Stałe globalne
9 const double INTEREST_RATE = 0.05;
10
11 int main()
12 {
13     // Znajdujące się tutaj instrukcje
14     // mają dostęp do stałej INTEREST_RATE
15     return 0;
16 }
17
18 void function2()
19 {
20     // Znajdujące się tutaj instrukcje
21     // mają dostęp do stałej INTEREST_RATE
22 }
23
24 void function3()
25 {
26     // Znajdujące się tutaj instrukcje
27     // mają dostęp do stałej INTEREST_RATE
28 }

```

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Zbiór instrukcji, który służy do wykonania określonego zadania w programie, nazywamy \_\_\_\_\_.
  - a) blokiem
  - b) parametrem
  - c) modułem
  - d) wyrażeniem
2. Zaletę korzystania z modułów polegającą na tym, że unikamy duplikowania kodu w programie, nazywamy \_\_\_\_\_.
  - a) wielokrotnym wykorzystaniem kodu
  - b) dziel i zwyciężaj
  - c) debugowaniem
  - d) nagłówkiem
3. Pierwsza linia definicji modułu nazywana jest \_\_\_\_\_.
  - a) ciałem
  - b) wprowadzeniem
  - c) inicjalizacją
  - d) nagłówkiem
4. Aby uruchomić kod w module, należy go \_\_\_\_\_.
  - a) zdefiniować
  - b) wywołać
  - c) zaimportować
  - d) wyeksportować
5. Miejsce w pamięci komputera, do którego powraca program po wykonaniu kodu w module, nazywa się \_\_\_\_\_.
  - a) zakończeniem
  - b) definicją modułu
  - c) adresem powrotnym
  - d) referencją
6. Technika projektowania polegająca na podzieleniu algorytmu na mniejsze elementy nazywa się \_\_\_\_\_.
  - a) „od ogółu do szczegółu”
  - b) „upraszczaniem kodu”
  - c) „refaktoryzacją kodu”
  - d) „hierarchicznym zadaniowaniem”
7. \_\_\_\_\_ to schemat, na którym w sposób graficzny przedstawione są relacje występujące między modułami w danym programie.
  - a) schemat blokowy
  - b) schemat powiązań modułów
  - c) schemat symboliczny
  - d)

8. Schemat hierarchiczny \_\_\_\_\_ to zmienna zadeklarowana wewnątrz modułu.
  - a) zmienna globalna
  - b) zmienna lokalna
  - c) zmienna ukryta
  - d) żadne z powyższych — nie można zadeklarować zmiennej wewnątrz modułu.
9. \_\_\_\_\_ to część programu, w której można się odwoływać do danej zmiennej.
  - a) strefa deklaracji
  - b) pole widoczności
  - c) zasięg widoczności
  - d) tryb
10. \_\_\_\_\_ to dane, które przekazuje się do modułu.
  - a) argument
  - b) parametr
  - c) nagłówek
  - d) pakiet
11. \_\_\_\_\_ to specjalna zmienna, do której są przekazywane dane podczas wywołania modułu.
  - a) argument
  - b) parametr
  - c) nagłówek
  - d) pakiet
12. Podczas \_\_\_\_\_ do parametru trafia tylko kopia wartości przekazanej jako argument.
  - a) przekazywania argumentu przez referencję
  - b) przekazywania argumentu przez nazwę
  - c) przekazywania argumentu przez wartość
  - d) przekazywania argumentu przez typ danych
13. Po \_\_\_\_\_ moduł będzie mógł modyfikować wartość argumentu zadeklarowanego na zewnątrz modułu.
  - a) przekazywaniu argumentu przez referencję
  - b) przekazywaniu argumentu przez nazwę
  - c) przekazywaniu argumentu przez wartość
  - d) przekazywaniu argumentu przez typ danych
14. Zmienna widoczna w każdym module programu to \_\_\_\_\_.
  - a) zmienna lokalna
  - b) zmienna uniwersalna
  - c) zmienna programowa
  - d) zmienna globalna
15. Jeśli jest to możliwe, należy unikać stosowania w programie zmiennych \_\_\_\_\_.
  - a) lokalnych
  - b) globalnych
  - c) referencyjnych
  - d) parametrycznych

### Prawda czy fałsz?

1. Określenie „dziel i zwycięzaj” oznacza, że programiści pracujący w zespole powinni się podzielić i pracować osobno.
2. Dzięki modułom praca w zespołach jest łatwiejsza.
3. Nazwa modułu powinna być jak najkrótsza.
4. Wywołanie modułu i definiowanie modułu to to samo.
5. Schemat blokowy służy do przedstawienia relacji występujących między modułami w programie.
6. Schemat hierarchiczny nie przedstawia operacji, jakie mają miejsce w programie.
7. Polecenie umieszczone w jednym module może się odwoływać do zmiennej lokalnej zdefiniowanej w innym module.
8. W przypadku większości języków programowania nie można w jednym zasięgu widoczności umieścić dwóch zmiennych o takich samych nazwach.
9. Większość języków programowania wymaga, aby argument przekazywany do modułu był tego samego typu co parametr.
10. W większości języków programowania nie można tworzyć modułów przyjmujących więcej niż jeden argument.
11. Kiedy przekazujemy argument przez referencję, moduł może zmodyfikować wartość argumentu i zmiana ta będzie zarejestrowana na zewnątrz modułu.
12. Przekazanie argumentu przez wartość zapewnia dwukierunkową komunikację między modułami.

### Krótką odpowiedź

1. W jaki sposób moduły przyczyniają się do wielokrotnego wykorzystania kodu?
2. Wymień i opisz dwa elementy, z których składa się definicja modułu w przypadku większości języków programowania.
3. Co się stanie, gdy program zakończy wykonywanie kodu umieszczonego w module?
4. Co to jest zmienna lokalna? Które instrukcje mogą się do niej odwoływać?
5. Gdzie się zaczyna, a gdzie kończy zasięg widoczności zmiennej lokalnej w większości języków programowania?
6. Jaka jest różnica między przekazywaniem argumentu przez wartość i przez referencję?
7. Dlaczego zmienne globalne sprawiają, że program jest znacznie trudniej debugować?

### Warsztat projektanta algorytmów

1. Zaprojektuj moduł o nazwie `timesTen`. Powinien on przyjmować argument typu `Integer`. Po wywołaniu powinien wyświetlić iloczyn wartości przekazanej jako argument i liczby 10.
2. Spójrz na poniższy nagłówek modułu i napisz polecenie, w którym wywołasz ten moduł i przekażesz do niego jako argument wartość 12.

`Module showValue(Integer quantity)`

3. Spójrz na następujący nagłówek modułu:

`Module myModule(Integer a, Integer b, Integer c)`

Teraz spójrz na następujące wywołanie modułu:

```
Call myModule(3, 2, 1)
```

Jakie wartości znajdą się w zmiennych a, b i c po wywołaniu modułu?

4. Założmy, że program zawiera następujący moduł:

```
Module display(Integer arg1, Real arg2, String arg3)
    Display "Oto wartości:"
    Display arg1, " ", arg2, " ", arg3
End Module
```

Założmy także, że ten sam program ma moduł main z następującymi deklaracjami zmiennych:

```
Declare Integer age
Declare Real income
Declare String name
```

Napisz polecenie, w którym wywołasz moduł display i przekażesz do niego te zmienne.

5. Zaprojektuj moduł getNumber, który za pomocą zmiennej typu referencyjnego będzie przyjmował argument typu Integer. Moduł powinien poprosić użytkownika o wprowadzenie liczby, a następnie powinien przypisać tę wartość do zmiennej typu referencyjnego.
6. Jaki wynik wyświetli się po wykonaniu poniższego pseudokodu?

```
Module main()
    Declare Integer x = 1
    Declare Real y = 3.4
    Display x, " ", y
    Call changeUs(x, y)
    Display x, " ", y
End Module

Module changeUs(Integer a, Real b)
    Set a = 0
    Set b = 0
    Display a, " ", b
End Module
```

7. Jaki wynik wyświetli się po wykonaniu poniższego pseudokodu?

```
Module main()
    Declare Integer x = 1
    Declare Real y = 3.4
    Display x, " ", y
    Call changeUs(x, y)
    Display x, " ", y
End Module

Module changeUs(Integer Ref a, Real Ref b)
    Set a = 0
    Set b = 0.0
    Display a, " ", b
End Module
```

## Ćwiczenia z wykrywania błędów

1. Znajdź błąd w następującym pseudokodzie:

```
Module main()
    Declare Real mileage
    Call getMileage()
    Display "Przejechałeś łącznie ", mileage, " mil."
End Module
Module getMileage()
    Display "Wprowadź przebieg pojazdu."
    Input mileage
End Module
```

2. Znajdź błąd w następującym pseudokodzie:

```
Module main()
    Call getCalories()
End Module

Module getCalories()
    Declare Real calories
    Display "Ile kalorii miał pierwszy posiłek?"
    Input calories
    Declare Real calories
    Display "Ile kalorii miał drugi posiłek?"
    Input calories
End Module
```

3. Znajdź potencjalny błąd w następującym pseudokodzie:

```
Module main()
    Call squareNumber(5)
End Module

Module squareNumber(Integer Ref number)
    Set number = number^2
    Display number
End Module
```

4. Znajdź błąd w następującym pseudokodzie:

```
Module main()
    Call raiseToPower(2, 1.5)
End Module

Module raiseToPower(Real value, Integer power)
    Declare Real result
    Set result = value^power
    Display result
End Module
```

## Ćwiczenia programistyczne

1. Zamiana kilometrów

Zaprojektuj program składający się z modułów, który będzie prosił użytkownika o wprowadzenie liczby kilometrów, a następnie będzie zamieniał tę wartość na mile. Wzór do przeliczania kilometrów na mile wygląda tak:

$$\text{mile} = \text{kilometry} \cdot 0,6214$$

## 2. Podatek — modyfikacja

Ćwiczenie programistyczne 6. w rozdziale 2. dotyczyło obliczania podatków obowiązujących w Stanach Zjednoczonych — czyli podatku stanowego i podatku w hrabstwie. W ćwiczeniu tym poprosiłem Cię o zaprojektowanie programu, który będzie służył do obliczania i wyświetlania wartości obu tych podatków na liczących od wartości zakupów. Jeśli już zaprojektowałeś ten program, zmień go tak, aby zawrzeć poszczególne podzadania w modułach. Jeśli jeszcze go nie zaprojektowałeś, zrób to teraz, tylko z podziałem na moduły.

## 3. Jaka kwota ubezpieczenia?

Wielu doradców finansowych radzi, aby właściciele nieruchomości ubezpieczali je do wartości nie mniejszej niż 80% wartości danej nieruchomości. Zaprojektuj program podzielony na moduły, który będzie prosił użytkownika o wprowadzenie wartości nieruchomości, a następnie wyświetli minimalną wartość ubezpieczenia dla danej nieruchomości.

## 4. Wydatki związane z samochodem

Zaprojektuj program składający się z modułów, który będzie prosił użytkownika o wprowadzenie miesięcznych kosztów związanych z posiadaniem samochodu, takich jak rata kredytu, składka ubezpieczeniowa, paliwo, olej, opony i serwis. Program powinien wyświetlać sumę tych wydatków, a także koszt roczny.

## 5. Podatek od nieruchomości

Hrabstwo pobiera podatek od nieruchomości na liczbę jej szacowanej wartości, wynoszącej 60% faktycznej wartości. Przykładowo jeżeli akr ziemi kosztuje 10000 dolarów, jego szacowana wartość wyniesie 6000 dolarów. Podatek od nieruchomości wynosi 64 centy od każdych 100 dolarów wartości szacowanej. Podatek od akra nieruchomości o wartości szacowanej równej 6000 dolarów będzie więc wynosił 38,40 dolara. Zaprojektuj program składający się z modułów, który będzie prosił użytkownika o wprowadzenie rzeczywistej wartości nieruchomości, a następnie będzie wyświetlał wartość szacowaną i kwotę podatku od nieruchomości.

## 6. Wskaźnik masy ciała

Zaprojektuj program składający się z modułów, który będzie obliczał i wyświetlał wskaźnik masy ciała (BMI) użytkownika. Wskaźnik BMI jest często wykorzystywany, aby określić, czy dana osoba ma niedowagę czy nadwagę. Wzór na wskaźnik BMI wygląda następująco:

$$\text{BMI} = \frac{\text{masa}}{\text{wzrost}^2}$$

## 7. Kalorie pochodzące z tłuszczy i z węglowodanów

Dietetyczka pracująca w klubie fitness pomaga klientom, obliczając kaloryczność spożytych przez nich posiłków. W związku z tym prosi ona klientów, aby podali, ile gramów tłuszczy i węglowodanów spożyli w ciągu dnia. Potem na podstawie następującego wzoru oblicza liczbę kalorii zawartych w tłuszczy:

$$\text{kalorie w tłuszczy} = \text{gramy tłuszczy} \cdot 9$$

A liczbę kalorii zawartą w węglowodanach oblicza według takiego wzoru:

$$\text{kalorie w węglowodanach} = \text{gramy węglowodanów} \cdot 4$$

Dietetyczka poprosiła Cię o zaprojektowanie programu składającego się z modułów, który będzie dokonywał tych obliczeń.

#### 8. Strefy na stadionie

Stadion jest podzielony na trzy strefy. Bilety w strefie A kosztują 70 złotych, w strefie B kosztują 55 złotych, a w strefie C — 32 złotych. Zaprojektuj program składający się z modułów, który będzie prosił użytkownika o wprowadzenie liczby sprzedanych biletów w każdej ze stref, a następnie wyświetli całkowity wpływ ze sprzedaży biletów.

#### 9. Kalkulator prac malarskich

Firma remontowa świadczy usługi malarskie i stwierdziła, że do pomalowania  $12 \text{ m}^2$  powierzchni trzeba zużyć 4 litry farby i poświęcić 8 godzin na pracę. Firma wyceniła pracę na 80 złotych za godzinę. Zaprojektuj program składający się z modułów, który będzie prosił użytkownika o wprowadzenie powierzchni, jaką trzeba pomalować, oraz kosztu 4 literów farby. Program powinien wyświetlać następujące dane:

- ilość farby (w litrach);
- czas malowania (w godzinach);
- koszt farby;
- koszt pracy;
- całkowity koszt.

#### 10. Miesięczny raport sprzedaży

Firma zarejestrowana w Stanach Zjednoczonych zajmująca się sprzedażą detaliczną musi raz w miesiącu sporządzić raport sprzedaży zawierający wartość sprzedaży, wartość podatku stanowego i wartość podatku w hrabstwie. Stawka podatku stanowego wynosi 4%, a podatku w hrabstwie — 2%. Zaprojektuj program składający się z modułów, który będzie prosił użytkownika o wprowadzenie wartości miesięcznej sprzedaży. Na jej podstawie program powinien obliczyć i wyświetlić następujące wartości:

- wartość podatku naliczonego w hrabstwie;
- wartość podatku stanowego;
- całkowita wartość podatku (suma obu podatków).

#### 11. Kalkulator hotdogowy

Założymy, że parówki do hot dogów kupuje się w opakowaniach po 10 sztuk, a bułki do hot dogów w opakowaniach po 8 sztuk. Zaprojektuj program składający się z modułów, który będzie obliczał minimalną liczbę opakowań z parówkami i bułkami, jaka będzie potrzebna, by zaspokoić apetyt gości przybyłych na imprezę. Program powinien poprosić użytkownika o wprowadzenie liczby gości oraz liczby hot dogów, jaką chce podać każdemu z gości. Program powinien wyświetlać następujące wartości:

- minimalna liczba opakowań z parówkami, jaka będzie potrzebna;
- minimalna liczba opakowań z bułkami, jaka będzie potrzebna;
- liczba niewykorzystanych parówek;
- liczba niewykorzystanych bułek.



## TEMATYKA

- |  |   |
|--|---|
| 4.1 Struktury warunkowe — informacje wstępne | 4.5 Struktura decyzyjna                   |
| 4.2 Struktury warunkowe podwójnego wyboru    | 4.6 Operatory logiczne                    |
| 4.3 Porównywanie ciągów znaków               | 4.7 Zmienne boolowskie                    |
| 4.4 Zagnieżdżone struktury warunkowe         | 4.8 Rzut oka na języki Java, Python i C++ |

## 4.1

Struktury warunkowe  
— informacje wstępne

**WYJAŚNIENIE:** Struktura warunkowa (zwana także strukturą wyboru) umożliwia programowi wykonywanie wybranego fragmentu kodu tylko wtedy, gdy spełniony jest określony warunek.

Struktura sterująca to konstrukcja logiczna, której celem jest sterowanie kolejnością, w jakiej będzie uruchamiana grupa instrukcji. Dotychczas używaliśmy tylko jednego typu takiej struktury — struktury sekwencyjnej. W rozdziale 2. wyjaśniłem, że struktura sekwencyjna to zbiór instrukcji, które uruchamiane będą w takiej kolejności, w jakiej zostały zapisane w programie. Oto przykład struktury sekwencyjnej zapisanej za pomocą pseudokodu (polecenia zostaną wykonane w kolejności od góry do dołu):

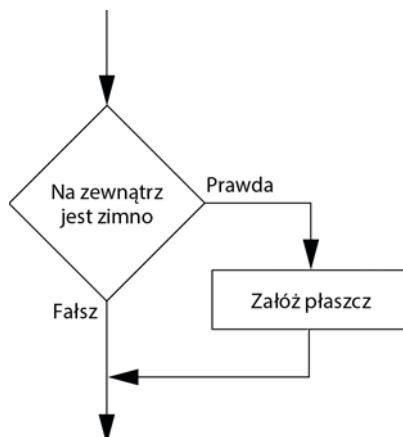
```
Declare Integer age
Display "Ile masz lat?"
Input age
Display "Oto wartość, którą wprowadziłeś:"
Display age
```

Nawet w rozdziale 3., w którym omawiałem moduły, każdy moduł także został zapisany jako struktura sekwencyjna. Przykładowo przedstawiony poniżej moduł jest strukturą sekwencyjną, ponieważ polecenia w nim zapisane zostaną wykonane w takiej kolejności, w jakiej pojawiają się w kodzie — od początku modułu aż do jego końca:

```
Module doubleNumber(Integer value)
    Declare Integer result
    Set result = value * 2
    Display result
End Module
```

Pomimo tego, że struktury sekwencyjne są bardzo często wykorzystywane w programach komputerowych, nie są one w stanie wykonać wszystkich typów zadań. Niektórych zadań nie da się rozwiązać, wykonując po prostu instrukcje, jedna za drugą. Przykładowo zastanówmy się nad programem do obliczania wynagrodzenia, który będzie sprawdzał, czy danemu pracownikowi należy doliczyć wynagrodzenie za nadgodziny. Jeżeli pracownik przepracował więcej niż 40 godzin, powinien otrzymać dodatkowe wynagrodzenie za każdą godzinę powyżej tego progu. W przeciwnym wypadku program nie powinien naliczać wynagrodzenia za nadgodziny. W programach tego typu będziemy potrzebować struktur innego rodzaju — takiej, która uruchomi pewne instrukcje tylko wtedy, gdy spełniony będzie określony warunek. Do tego typu zadań służą **struktury warunkowe** (ang. *decision structure*), zwane także **strukturami wyboru** (ang. *selection structures*).

W najprostszej postaci struktura warunkowa wykona pewne instrukcje tylko wtedy, gdy spełniony zostanie określony warunek. Jeżeli warunek nie zostanie spełniony, instrukcje się nie wykonają. Na schemacie blokowym przedstawionym na rysunku 4.1 pokazałem, w jaki sposób można wyobrazić sobie strukturę warunkową na przykładzie z życia wziętym. Za pomocą rombu oznaczony jest warunek, jaki będziemy sprawdzać. Jeżeli warunek zostanie spełniony (czyli wynikiem będzie prawda), będziemy się kierować wskazaną ścieżką, która poprowadzi nas do bloku operacyjnego. Jeżeli warunek nie zostanie spełniony (wynikiem będzie fałsz), będziemy kierować się inną ścieżką i blok operacyjny zostanie pominięty.



Rysunek 4.1. Prosta struktura warunkowa

Symbol rombu na schemacie blokowym oznacza, że w tym miejscu będziemy musieli sprawdzić pewien warunek. W tym przypadku sprawdzamy, czy *Na zewnątrz jest zimno* jest stwierdzeniem prawdziwym czy fałszywym. Jeśli jest to stwierdzenie prawdziwe, przejdziemy do wykonywania operacji *Załóż płaszcz*. Jeżeli jest to stwierdzenie fałszywe, operacja nie zostanie wykonana. Operacja zostanie więc **wykonana warunkowo**, ponieważ wykona się tylko wtedy, gdy spełniony zostanie określony warunek.

Programiści nazywają taką strukturę przedstawioną na rysunku 4.1 **strukturą warunkową pojedynczego wyboru**. To dlatego, że w takim przypadku mamy do wyboru tylko jedną ścieżkę alternatywną — jeśli warunek umieszczony w rombie jest spełniony, zostaniemy skierowani na ścieżkę alternatywną, w innym przypadku wyjdziemy po prostu ze struktury.

## Łączenie struktur

Z pomocą tylko i wyłącznie struktur warunkowych nie da się stworzyć całego programu. Za ich pomocą możemy w pewnym miejscu programu określić, czy spełniony jest pewien warunek, i warunkowo wykonać pewne instrukcje. W dalszej części programu będziemy musieli korzystać z innych struktur. Przykładowo na rysunku 4.2 przedstawiłem pełny schemat blokowy, w którym połączylem strukturę warunkową z dwiema strukturami sekwencyjnymi (schemat ten nie dotyczy programu komputerowego, lecz działań człowieka).

Schemat blokowy rozpoczyna się od struktury sekwencyjnej. Zakładając, że masz przy oknie termometr, pierwszymi krokami będą: *Podejdź do okna i Odczytaj wskazanie termometru*. Następnie pojawia się struktura warunkowa, która sprawdza warunek *Na zewnątrz jest zimno*. Jeżeli jest to prawda, wykonana zostanie operacja *Załóż płaszcz*. Następnie pojawia się kolejna struktura sekwencyjna i wykonane zostaną kroki *Otwórz drzwi*, a potem *Wyjdź na zewnątrz*.

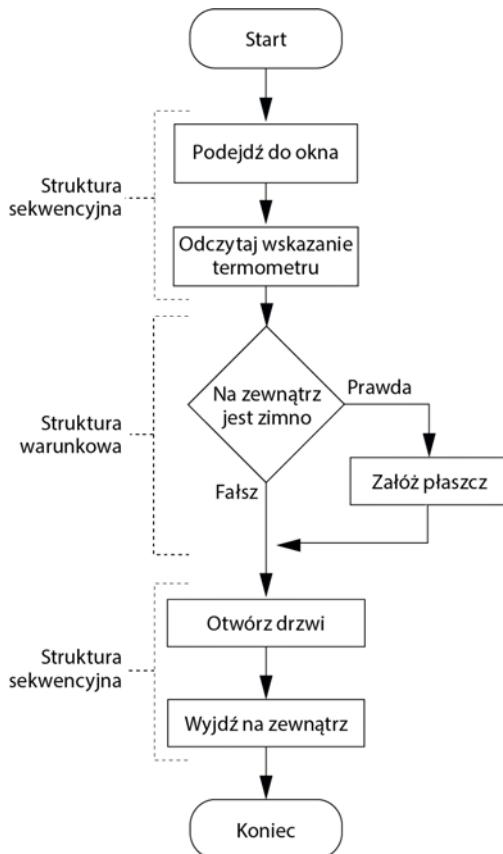
Bardzo często będziesz musiał też zagnieźdzać jedną strukturę w innej. Spójrz na fragment schematu blokowego z rysunku 4.3. Widać na nim strukturę warunkową, a w niej strukturę sekwencyjną (ponownie schemat ilustruje działania człowieka, a nie programu komputerowego). W strukturze warunkowej sprawdzany jest warunek *Na zewnątrz jest zimno*. Jeżeli warunek jest spełniony, zostaną wykonane operacje umieszczone w strukturze sekwencyjnej.

## Tworzenie struktury warunkowej w pseudokodzie

Do tworzenia struktury warunkowej pojedynczego wyboru w pseudokodzie będziemy używali instrukcji If-Then. Oto ogólna postać instrukcji If-Then:

```
If warunek Then
    instrukcja
    instrukcja
    itd.
}
End If
```

**Te instrukcje zostaną wykonane warunkowo.**



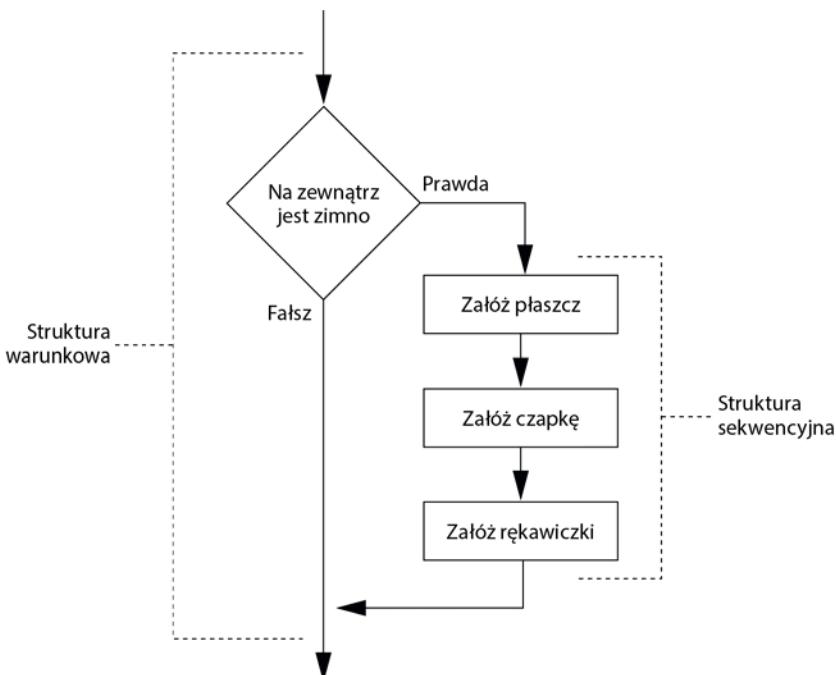
**Rysunek 4.2.** Połączenie struktur sekwencyjnych ze strukturą warunkową

Aby uniknąć nieporozumień, odwołując się do linii ze słowem If, będę się posługiwał w tekście określeniem *klauzula If*, a odwołując się do linii ze słowami *End If*, będę się posługiwał określeniem *klauzula End If*. W takiej ogólnej postaci *warunek* oznacza dowolne wyrażenie, które zwraca wartość typu prawda lub fałsz. Kiedy uruchomi się instrukcja *If-Then*, zostanie sprawdzony wskazany warunek. Jeżeli jest on spełniony, uruchomią się *instrukcje* zawarte pomiędzy klauzulami *If* i *End If*. Klauzula *End If* wskazuje koniec instrukcji *If-Then*.

### Wyrażenia boolowskie i operatory relacji

We wszystkich językach programowania można tworzyć wyrażenia, które zwracają wynik typu prawda lub fałsz. Nazywamy je *wyrażeniami boolowskimi* (ang. *Boolean expressions*) od nazwiska angielskiego matematyka George'a Boole'a. W XIX wieku Boole wprowadził do matematyki system, który umożliwia dokonywanie obliczeń na podstawie abstrakcyjnych pojęć: prawda i fałsz. Warunek, który będziemy sprawdzali za pomocą instrukcji *If-Then*, musi być wyrażeniem boolowskim.

Zazwyczaj wyrażenie boolowskie, które będziemy weryfikowali za pomocą instrukcji *If-Then*, będzie zawierało operatory relacji. **Operator relacji** służy do sprawdzania,



**Rysunek 4.3.** Struktura sekwencyjna zagnieżdzona w strukturze warunkowej

czy między dwiema wartościami zachodzi określona relacja. Przykładowo operator *większe niż* ( $>$ ) sprawdza, czy jedna wartość jest większa od drugiej, a operator *równe* ( $==$ ) sprawdza, czy dwie wartości są sobie równe. W tabeli 4.1 zamieściłem listę operatorów relacji powszechnie stosowanych w większości języków programowania.

**Tabela 4.1.** Operatory relacji

Operator	Znaczenie
$>$	Większe niż
$<$	Mniejsze niż
$\geq$	Większe niż lub równe
$\leq$	Mniejsze niż lub równe
$==$	Równe
$\neq$	Różne od

Oto przykład wyrażenia, w którym za pomocą operatora *większe niż* porównuję ze sobą wartości zapisane w dwóch zmiennych, `length` i `width`:

`length > width`

Wyrażanie to sprawdza, czy wartość zapisana w zmiennej `length` jest większa od wartości zapisanej w zmiennej `width`. Jeśli `length` jest większe od `width`, wyrażenie zwróci prawdę. W przeciwnym wypadku zwróci fałsz. Ponieważ wyrażenie to zwraca

prawdę albo fałsz, jest wyrażeniem boolowskim. Poniższe wyrażenie sprawdza za pomocą operatora *mniejsze niż*, czy `length` jest mniejsze od `width`:

```
length < width
```

W tabeli 4.2 przedstawiłem kilka przykładów wyrażeń boolowskich porównujących wartości zapisane w zmiennych `x` i `y`.

**Tabela 4.2.** Wyrażenia boolowskie z operatorami relacji

Wyrażenie	Znaczenie
<code>x &gt; y</code>	Czy <code>x</code> jest większe niż <code>y</code> ?
<code>x &lt; y</code>	Czy <code>x</code> jest mniejsze niż <code>y</code> ?
<code>x &gt;= y</code>	Czy <code>x</code> jest większe niż lub równe <code>y</code> ?
<code>x &lt;= y</code>	Czy <code>x</code> jest mniejsze niż lub równe <code>y</code> ?
<code>x == y</code>	Czy <code>x</code> jest równe <code>y</code> ?
<code>x != y</code>	Czy <code>x</code> jest różne od <code>y</code> ?

### Operatory `>=` i `<=`

Operatory `>=` i `<=` sprawdzają dwie rzeczy. Operator `>=` sprawdza, czy operand znajdujący się po jego lewej stronie jest większy lub równy operandowi po prawej stronie. Założymy, że do zmiennej `a` jest przypisana wartość 4, do zmiennej `b` — wartość 6, a do zmiennej `c` — wartość 4. W takim przypadku wyrażenia `b >= a` i `a >= c` będą zwracały prawdę, a wyrażenie `a >= 5` będzie zwracało fałsz.

Operator `<=` sprawdza, czy operand znajdujący się po jego lewej stronie jest mniejszy lub równy operandowi po prawej stronie. Ponownie, jeśli założymy, że do zmiennej `a` jest przypisana wartość 4, do zmiennej `b` — wartość 6, a do zmiennej `c` — wartość 4, zarówno wyrażenie `a <= c`, jak i wyrażenie `b <= 10` będą zwracały prawdę, a wyrażenie `b <= a` będzie zwracało fałsz

### Operator `==`

Operator `==` sprawdza, czy operand znajdujący się po jego lewej stronie jest równy operandowi po prawej stronie. Jeśli oba operandy mają taką samą wartość, wyrażenie będzie zwracało prawdę. Przy założeniu, że w zmiennej `a` jest zapisana wartość 4, wyrażenie `a == 4` będzie zwracało prawdę, a wyrażenie `a == 2` będzie zwracało fałsz.

Aby uniknąć zamieszania, w niniejszej książce do oznaczania operatora *równie* będę używał dwóch znaków równości — w ten sposób nie będzie się on mylił z operatorem przypisania, oznaczanym jednym znakiem równości. W kilku językach programowania jest identycznie — łącznie z Javą, Pythonem, C i C++.



**OSTRZEŻENIE!** Jeżeli będziesz pisać program w jednym z języków, w których do oznaczania operatora *równe* używa się podwójnych znaków równości, zachowaj ostrożność i nie pomył go z operatorem przypisania. W językach takich jak Java, Python, C czy C++ operator == sprawdza, czy jedna wartość jest równa innej wartości, a operator = służy do przypisywania wartości do zmiennej.

### Operator !=

Operator != oznacza *różne od*. Sprawdza on, czy operand znajdujący się po jego lewej stronie jest różny od operandu po prawej stronie — czyli jest on przeciwieństwem operatora ==. Założymy tak jak przed chwilą, że do zmiennej a jest przypisana wartość 4, do zmiennej b — wartość 6, a do zmiennej c — wartość 4. W takim przypadku wyrażenia a != b i b != c będą zwracały prawdę — ponieważ a jest różne od b, natomiast b jest różne od c. Z kolei wyrażenie a != c zwróci fałsz, gdyż a nie jest różne od c.

Symbol != jest wykorzystywany jako operator *różne od* w wielu językach programowania — łącznie z Javą, C i C++. Jednak w niektórych językach, na przykład w Visual Basic, rolę tego operatora pełni symbol <>.

## Zbieramy wszystkie informacje razem

Przyjrzyjmy się teraz następującemu pseudokodowi:

```
If sales > 50000 Then
    Set bonus = 500.0
End If
```

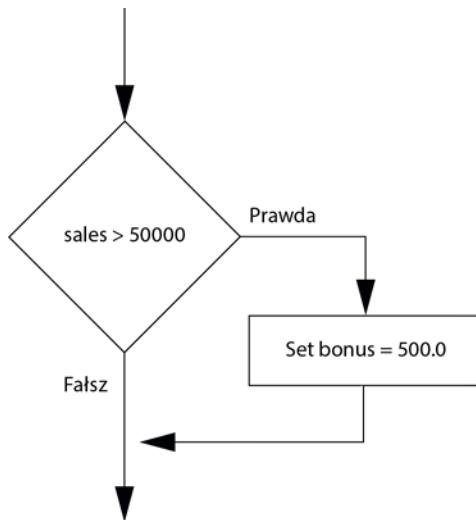
W poleceniu tym operator > sprawdza, czy zmienna sales jest większa niż 50000. Jeśli wyrażenie sales > 50000 zwróci prawdę, do zmiennej bonus zostanie przypisana wartość 500.0. W przeciwnym wypadku instrukcja przypisania wartości zostanie pominięta przez program. Na rysunku 4.4 przedstawiłem schemat blokowy tego fragmentu kodu.

W poniższym przykładzie sekwencja instrukcji zostanie wykonana warunkowo. Na rysunku 4.5 widoczny jest schemat blokowy dla tego kodu.

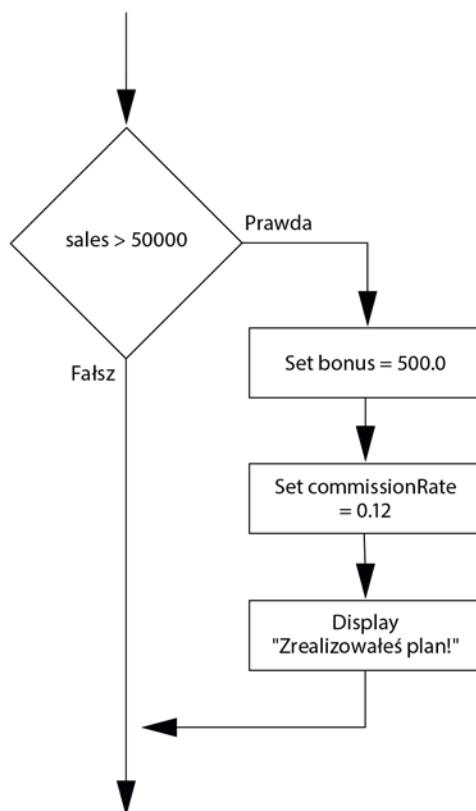
```
If sales > 50000 Then
    Set bonus = 500.0
    Set commissionRate = 0.12
    Display "Zrealizowałeś plan!"
End If
```

W poniższym pseudokodzie za pomocą operatora == sprawdzam, czy dwie wartości są sobie *równe*. Wyrażenie balance == 0 zwróci prawdę wtedy, gdy zmienna balance będzie równa 0. W przeciwnym wypadku wyrażenie zwróci fałsz.

```
If balance == 0 Then
    // Widoczne tutaj instrukcje
    // uruchomią się wtedy, gdy zmienna balance
    // będzie równa 0
End If
```



Rysunek 4.4. Przykładowa struktura warunkowa



Rysunek 4.5. Przykładowa struktura warunkowa

W poniższym pseudokodzie za pomocą operatora `!=` sprawdzam, czy dwie wartości są od siebie różne. Wrażenie `choice != 5` zwróci prawdę wtedy, gdy zmieniona `choice` będzie różna od 5. W przeciwnym wypadku wyrażenie zwróci fałsz.

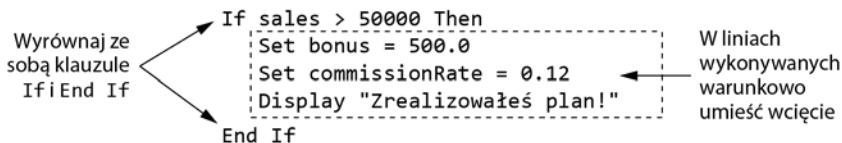
```
If choice != 5 Then
    // Widoczne tutaj instrukcje
    // uruchomią się wtedy, gdy zmieniona choice
    // będzie różna od 5
End If
```

## Styl programowania a instrukcja If-Then

Na rysunku 4.6 pokazałem konwencję, jakiej należy się trzymać, umieszczając w programie instrukcję If-Then:

- upewnij się, że klauzule `If` i `End If` są wyrównane do lewej strony;
- w liniach zawierających instrukcje, które mają zostać wykonane warunkowo, umieść wcięcie względem klauzul `If` i `End If`.

Umieszczając wcięcia przy poleceńach, które mają zostać wykonane warunkowo, sprawisz, że fragment ten będzie się odróżniał od pozostałej części kodu. Dzięki temu Twój program będzie czytelniejszy i łatwiejszy w debugowaniu. Większość programistów stosuje się do tej konwencji, zarówno pisząc programy w pseudokodzie, jak i w prawdziwych językach programowania.



Rysunek 4.6. Konwencja zapisu instrukcji If-Then

## W centrum uwagi Korzystanie z instrukcji If-Then

Kasia uczy w szkole biologii. Uczniowie muszą w czasie semestru napisać trzy sprawdziany. Kasia chce stworzyć program, z którego będą mogli korzystać uczniowie do obliczania średniego wyniku ze sprawdzianów. Chce także, aby program pogratulował uczniowi, jeżeli jego średni wynik ze sprawdzianów będzie wyższy niż 95. Oto algorytm programu:

1. Pobierz wynik z pierwszego sprawdzianu.
2. Pobierz wynik z drugiego sprawdzianu.
3. Pobierz wynik z trzeciego sprawdzianu.
4. Oblicz średnią wyników.
5. Wyświetl średnią.
6. Jeżeli średnia jest wyższa niż 95, złoż uczniowi gratulacje.



Na listingu 4.1 przedstawiłem pseudokod programu, a na rysunku 4.7 znajduje się jego schemat blokowy.

### **Listing 4.1**



```

1 // Deklaracja zmiennych
2 Declare Real test1, test2, test3, average
3
4 // Pobieramy wynik z pierwszego sprawdzianu
5 Display "Wprowadź wynik ze sprawdzianu nr 1."
6 Input test1
7
8 // Pobieramy wynik z drugiego sprawdzianu
9 Display "Wprowadź wynik ze sprawdzianu nr 2."
10 Input test2
11
12 // Pobieramy wynik z trzeciego sprawdzianu
13 Display "Wprowadź wynik ze sprawdzianu nr 3."
14 Input test3
15
16 // Obliczamy średni wynik
17 Set average = (test1 + test2 + test3) / 3
18
19 // Wyświetlamy średni wynik
20 Display "Średni wynik wynosi ", average
21
22 // Jeżeli średni wynik jest większy niż 95,
23 // złożyć uczniowi gratulacje
24 If average > 95 Then
25   Display "Gratuluję! Świetny wynik!"
26 End If

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik ze sprawdzianu nr 1.

**82 [Enter]**

Wprowadź wynik ze sprawdzianu nr 2.

**76 [Enter]**

Wprowadź wynik ze sprawdzianu nr 3.

**91 [Enter]**

Średni wynik wynosi 83

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik ze sprawdzianu nr 1.

**93 [Enter]**

Wprowadź wynik ze sprawdzianu nr 2.

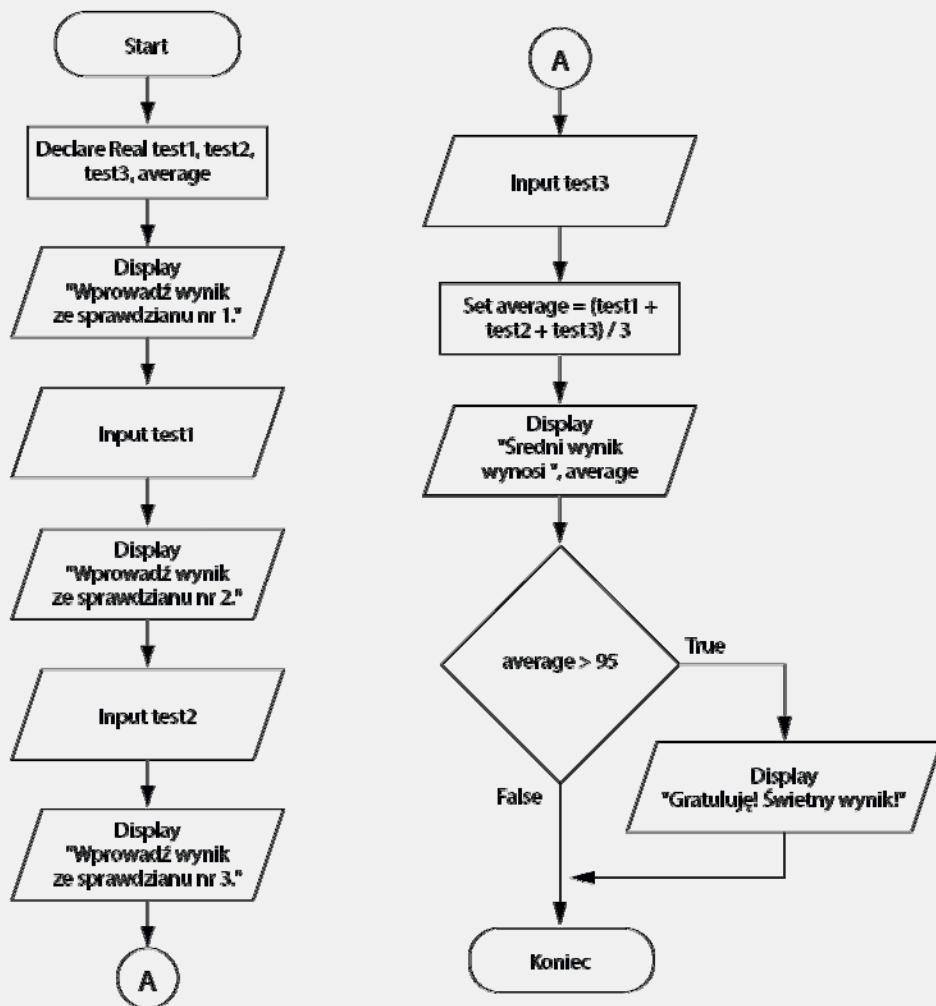
**99 [Enter]**

Wprowadź wynik ze sprawdzianu nr 3.

**96 [Enter]**

Średni wynik wynosi 96

Gratuluję! Świetny wynik!



**Rysunek 4.7.** Schemat blokowy programu z listingu 4.1



## Punkt kontrolny

- 4.1. Co to jest struktura sterująca?
- 4.2. Co to jest struktura warunkowa?
- 4.3. Co to jest struktura warunkowa pojedynczego wyboru?
- 4.4. Co to jest wyrażenie boolowskie?
- 4.5. Jakiego rodzaju relacje między dwiema wartościami można sprawdzać za pomocą operatorów relacji?
- 4.6. Napisz za pomocą pseudokodu instrukcję If-Then, która będzie przypisywała do zmiennej x wartość 0, jeżeli y jest równe 20.

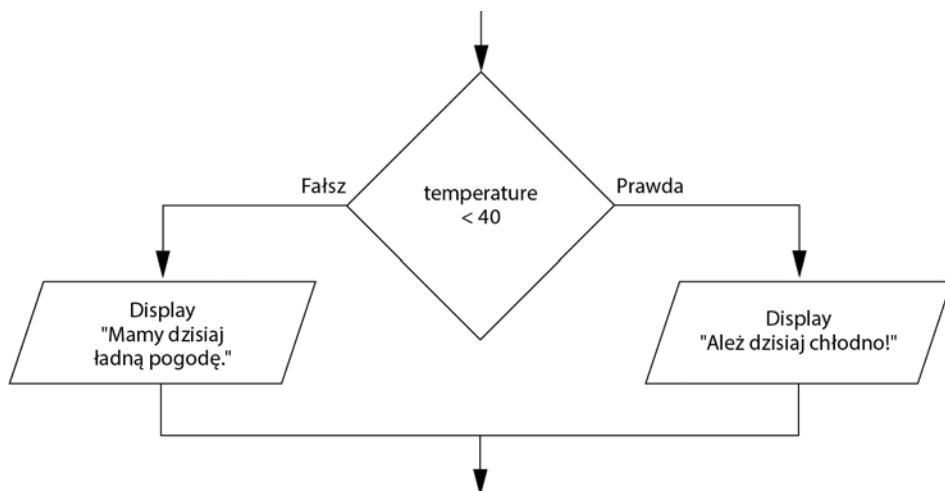
- 4.7. Napisz za pomocą pseudokodu instrukcję If-Then, która będzie przypisywała do zmiennej commission wartość 0.2, jeżeli zmienna sales jest większa lub równa 10000.

## 4.2

# Struktury warunkowe podwójnego wyboru

**WYJAŚNIENIE:** Struktura warunkowa podwójnego wyboru można wykonać jeden zbiór instrukcji, jeżeli określone wyrażenie boolowskie zwróci prawdę, lub inny zbiór instrukcji, gdy wyrażenie zwróci fałsz.

W strukturze warunkowej podwójnego wyboru istnieją dwie ścieżki, którymi może podążyć program. Pierwsza z nich dotyczy sytuacji, gdy warunek zostanie spełniony, a druga — gdy warunek nie zostanie spełniony. Na rysunku 4.8 widoczny jest schemat blokowy struktury warunkowej podwójnego wyboru.



Rysunek 4.8. Struktura warunkowa podwójnego wyboru

Struktura warunkowa przedstawiona na schemacie blokowym sprawdza, czy spełniony jest warunek `temperature < 40`. Jeżeli warunek ten jest spełniony, wykonane zostanie polecenie `Display "Mamy dzisiaj ładną pogodę."`. Jeżeli warunek nie jest spełniony, uruchomi się polecenie `Display "Ależ dzisiaj chłodno!"`.

W pseudokodzie będziemy zapisywali taką strukturę za pomocą instrukcji If-Then-Else. Oto ogólna postać instrukcji If-Then-Else:

```

If warunek Then
    instrukcja
    instrukcja
    itd.
} Te instrukcje wykonają się, gdy warunek zostanie spełniony

```

```

Else
    instrukcja
    instrukcja
    itd.
}
End If

```

} Te instrukcje wykonają się, gdy warunek nie zostanie spełniony

Warunek oznacza tutaj dowolne wyrażenie boolowskie. Jeśli wyrażenie to zwróci prawdę, wykonają się instrukcje zawarte między tym wyrażeniem a słowem Else. Jeśli wyrażenie zwróci fałsz, wykonają się instrukcje zawarte między słowami Else i End If. Linia, w której pojawiają się słowa End If, oznacza koniec instrukcji If-Then-Else.

Poniższy pseudokod pokazuje przykładową instrukcję If-Then-Else. Odpowiada ona schematowi blokowemu z rysunku 4.8.

```

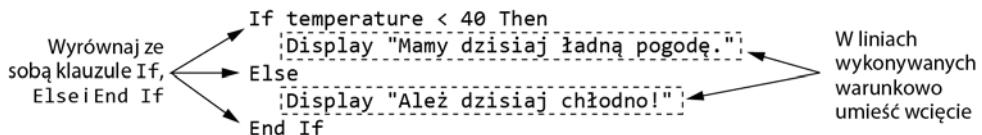
If temperature < 40 Then
    Display "Mamy dzisiaj ładną pogodę."
Else
    Display "Ależ dzisiaj chłodno!"
End If

```

Odwołując się do linii ze słowem Else, będę się posługiwał w tekście określeniem *klauzula Else*. Tworząc w programie instrukcję If-Then-Else, stosuj się do następującej konwencji:

- upewnij się, że klauzule If, Else i End If są wyrównane do lewej strony;
- w liniach między klauzulami If i Else oraz Else i End If, zawierających instrukcje, które mają zostać wykonane warunkowo, umieść wcięcie.

Przedstawiłem to na rysunku 4.9.



**Rysunek 4.9.** Konwencja zapisu instrukcji If-Then-Else



## W centrum uwagi

### Korzystanie z instrukcji If-Then-Else

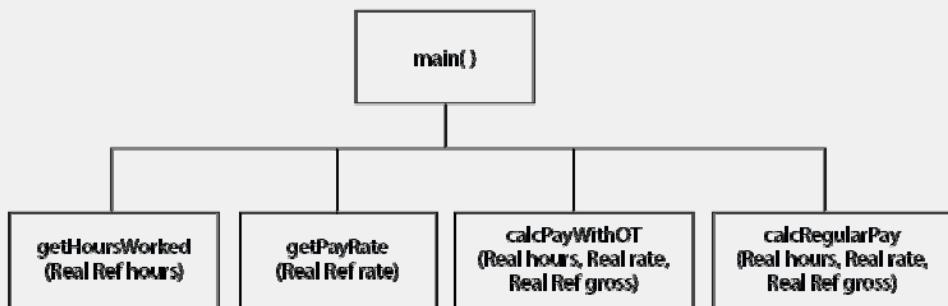
Krzysztof jest właścicielem warsztatu samochodowego i zatrudnia kilku pracowników. Jeżeli dany pracownik przepracuje w ciągu tygodnia więcej niż 40 godzin, Krzysztof wyplaca mu za każdą nadgodzinę kwotę 1,5 raza większą od normalnej stawki godzinowej. Krzysztof poprosił Cię o zaprojektowanie prostego programu, który będzie obliczał wynagrodzenie pracownika, uwzględniając stawkę za nadgodziny. Zaprojektujesz więc następujący algorytm:

1. Pobierz liczbę przepracowanych godzin.
2. Pobierz stawkę godzinową.

3. Jeżeli pracownik przepracował więcej niż 40 godzin, oblicz wynagrodzenie, uwzględniając nadgodziny. W przeciwnym razie oblicz wynagrodzenie standardowo.
4. Wyświetl wynagrodzenie.

Stosując technikę projektowania „od ogółu do szczegółu”, stworzyłeś schemat hierarchiczny widoczny na rysunku 4.10. Widać na nim moduł `main`, który będzie wywoływał cztery inne moduły. Oto podsumowanie informacji na temat tych modułów:

- `getHoursWorked` — będzie prosił użytkownika o wprowadzenie liczby przepracowanych godzin;
- `getPayRate` — będzie prosił użytkownika o wprowadzenie stawki godzinowej;
- `calcPayWithOT` — będzie obliczał wynagrodzenie pracownika za przepracowane nadgodziny;
- `calcRegularPay` — będzie obliczał wynagrodzenie pracownika bez uwzględnienia nadgodzin.



**Rysunek 4.10.** Schemat hierarchiczny

W module `main` będziemy wywoływać te moduły i wyświetliemy obliczone wynagrodzenie. Pseudokod programu przedstawiony jest na listingu 4.2, a na rysunkach 4.11 i 4.12 widoczne są schematy blokowe poszczególnych modułów.

### Listing 4.2



```

1 // Stale globalne
2 Constant Integer BASE_HOURS = 40
3 Constant Real OT_MULTIPLIER = 1.5
4
5 Module main()
6   // Zmienne lokalne
7   Declare Real hoursWorked, payRate, grossPay
8
9   // Pobieramy liczbę przepracowanych godzin
10  Call getHoursWorked(hoursWorked)
11
12  // Pobieramy stawkę godzinową
13  Call getPayRate(payRate)
14
15  // Obliczamy wynagrodzenie
16  If hoursWorked > BASE_HOURS Then
17    Call calcPayWithOT(hoursWorked, payRate,
18                      grossPay)
  
```

```

19     Else
20         Call calcRegularPay(hoursWorked, payRate,
21                             grossPay)
22     End If
23
24     // Wyświetlamy wynagrodzenie
25     Display "Wynagrodzenie wynosi ", grossPay, " zł."
26 End Module
27
28 // Moduł getHoursWorked pobiera
29 // liczbę przepracowanych godzin i zapisuje ją
30 // w parametrze hours
31 Module getHoursWorked(Real Ref hours)
32     Display "Wprowadź liczbę przepracowanych godzin."
33     Input hours
34 End Module
35
36 // Moduł getPayRate pobiera
37 // stawkę godzinową i zapisuje ją
38 // w parametrze rate
39 Module getPayRate(Real Ref rate)
40     Display "Wprowadź stawkę godzinową."
41     Input rate
42 End Module
43
44 // Moduł calcPayWithOT oblicza wynagrodzenie
45 // bez nadgodzin i zapisuje je
46 // w parametrze gross
47 Module calcPayWithOT(Real hours, Real rate,
48                      Real Ref gross)
49     // Zmienne lokalne
50     Declare Real overtimeHours, overtimePay
51
52     // Obliczamy liczbę nadgodzin
53     Set overtimeHours = hours - BASE_HOURS
54
55     // Obliczamy wynagrodzenie za nadgodziny
56     Set overtimePay = overtimeHours * rate *
57                     OT_MULTIPLIER
58
59     // Obliczamy całkowite wynagrodzenie
60     Set gross = BASE_HOURS * rate + overtimePay
61 End Module
62
63 // Moduł calcRegularPay oblicza
64 // wynagrodzenie bez nadgodzin i zapisuje je
65 // w parametrze gross
66 Module calcRegularPay(Real hours, Real rate,
67                       Real Ref gross)
68     Set gross = hours * rate
69 End Module

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę przepracowanych godzin.

**40 [Enter]**

Wprowadź stawkę godzinową.

**20 [Enter]**

Wynagrodzenie wynosi 800 zł.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

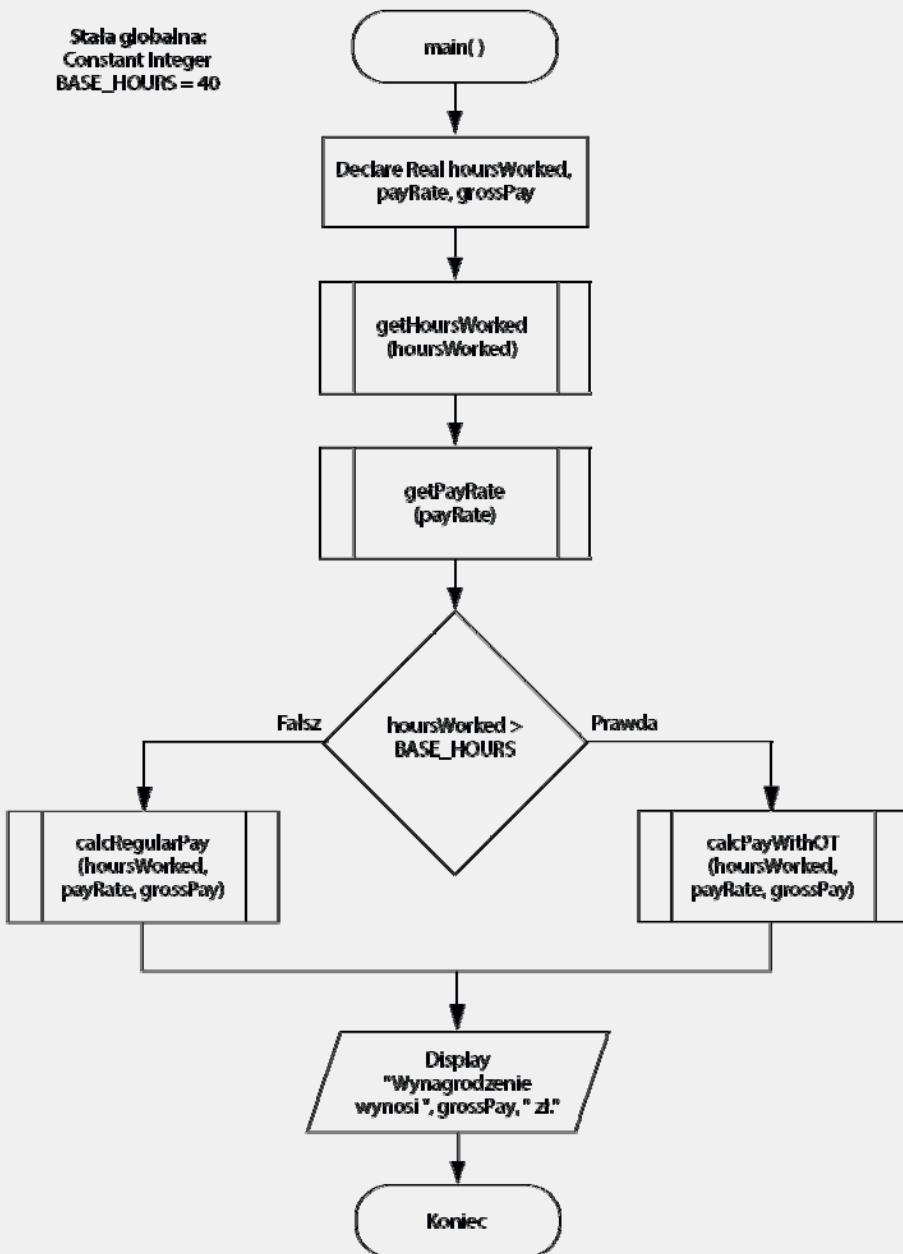
Wprowadź liczbę przepracowanych godzin.

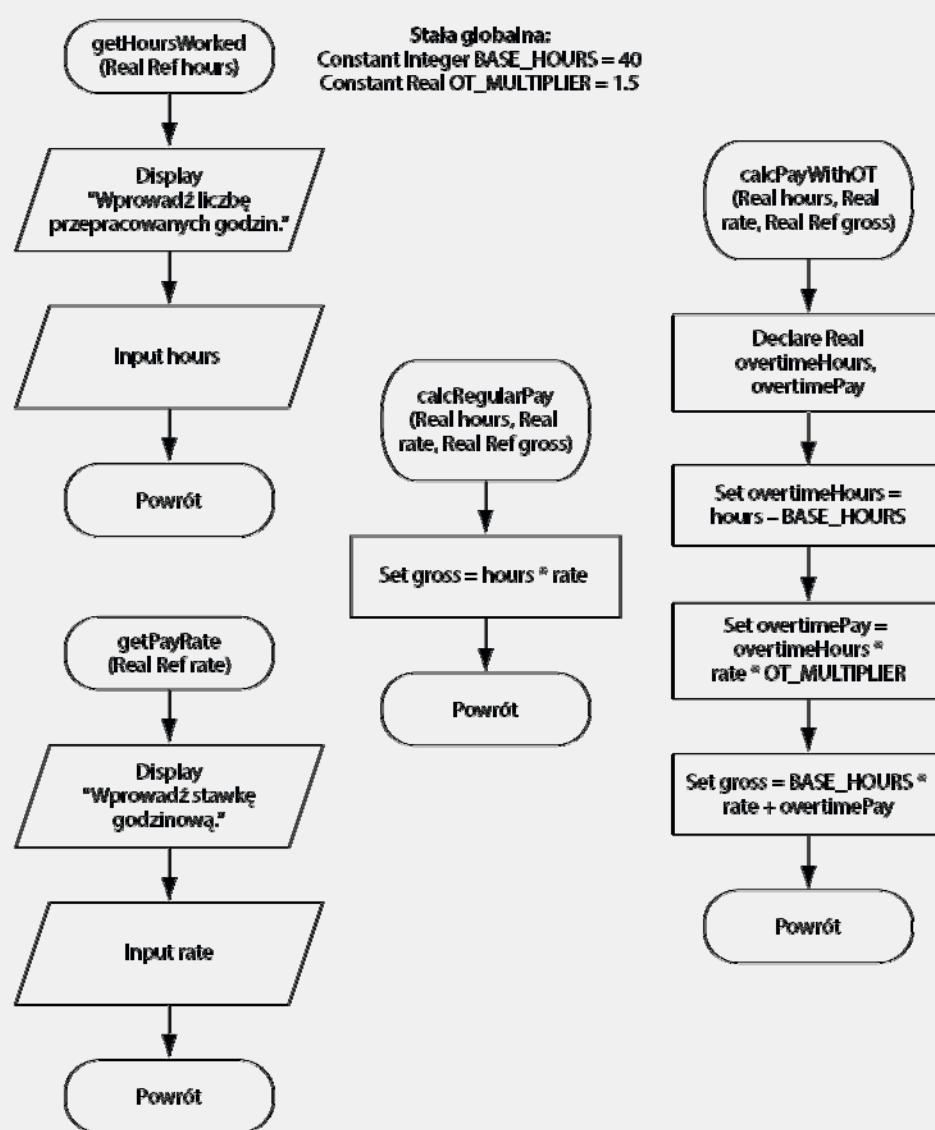
50 [Enter]

Wprowadź stawkę godzinową.

20 [Enter]

Wynagrodzenie wynosi 1100 zł.

**Rysunek 4.11.** Schemat blokowy modułu main



Rysunek 4.12. Schematy blokowe pozostałych modułów

Zwróc uwagę, że w liniach 2. i 3. zadeklarowałem stałe globalne. Do stałej BASE\_HOURS jest przypisana wartość 40, która oznacza liczbę godzin, do której nie wypłaca się dodatku za nadgodziny. Do stałej OT\_MULTIPLIER jest przypisana wartość 1.5, przez którą mnożona jest stawka godzinowa za każdą nadgodzinę.



## Punkt kontrolny

- 4.8. W jaki sposób działa struktura warunkowa podwójnego wyboru?
- 4.9. Za pomocą której instrukcji zapisujemy w pseudokodzie strukturę warunkową podwójnego wyboru?
- 4.10. Kiedy wykonają się instrukcje zawarte między klauzulami Else i End If w przypadku instrukcji If-Then-Else?

### 4.3

## Porównywanie ciągów znaków

**WYJAŚNIENIE:** W większości języków programowania można porównywać ze sobą ciągi znaków. Dzięki temu można tworzyć w programie struktury warunkowe, które sprawdzają wartość przypisaną do ciągu znaków.

Na przedstawionych wcześniej przykładach pokazalem, w jaki sposób można porównywać ze sobą wartości liczbowe. W przypadku większości języków programowania można jednak także porównywać ze sobą ciągi znaków. Spójrz na następujący przykład:

```
Declare String name1 = "Mary"
Declare String name2 = "Mark"
If name1 == name2 Then
    Display "Imiona są takie same."
Else
    Display "Imiona NIE są takie same."
End If
```

Operator == sprawdza, czy wartości zapisane w zmiennych name1 i name2 są sobie równe. Ponieważ ciągi znaków "Mark" i "Mary" nie są sobie równe, klauzula Else wyświetli komunikat "Imiona NIE są takie same.".

Mogą także porównywać ciąg znaków z literałem ciągu znaków. Założymy, że zmienna month jest typu String. Poniższy przykładowy pseudokod sprawdza za pomocą operatora !=, czy zmienna month jest różna od "Październik":

```
If month != "Październik" Then
    instrukcja
End If
```

Pseudokod zamieszczony na listingu 4.3, jak można porównać dwa ciągi znaków. Program prosi użytkownika o wprowadzenie hasła, a następnie sprawdza, czy wprowadzony ciąg znaków jest równy "prospero".



**UWAGA:** W większości języków programowania podczas porównywania ciągów znaków uwzględniana jest wielkość liter. Przykładowo ciągi znaków "sobota" i "Sobota" nie są sobie równe, ponieważ w pierwszym z nich litera „s” jest mała, a w drugim — duża.

**Listing 4.3**

```

1 // Zmienna, w której zapiszemy hasło
2 Declare String password
3
4 // Prosimy użytkownika o wprowadzenie hasła
5 Display "Wprowadź hasło."
6 Input password
7
8 // Sprawdzamy, czy użytkownik wprowadził
9 // prawidłowe hasło
10 If password == "prospero" Then
11   Display "Hasło jest poprawne."
12 Else
13   Display "Wprowadzone hasło jest niepoprawne."
14 End If

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź hasło.

**ferdinand [Enter]**

Wprowadzone hasło jest niepoprawne.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź hasło.

**prospero [Enter]**

Hasło jest poprawne.

## Inne przykłady porównywania ciągów znaków

Poza prostym porównaniem, czy oba ciągi znaków są sobie równe, wiele języków umożliwia sprawdzenie, czy jeden z ciągów jest większy lub mniejszy od drugiego. To bardzo pożyteczna cecha, ponieważ programiści często muszą projektować programy sortujące listę wyrazów.

W rozdziale 1. wspomniałem, że komputer tak naprawdę nie przechowuje w pamięci znaków takich jak A, B, C itd. Zamiast tego w pamięci komputera umieszczone są kody tych znaków. W rozdziale 1. wspomniałem także, że najbardziej popularnym systemem kodowania znaków jest ASCII (*American Standard Code for Information Interchange*). W dodatku A znajdziesz kompletny zestaw znaków ASCII, ale tutaj przedstawię kilka podstawowych informacji na ich temat:

- Dużym literom od „A” do „Z” odpowiadają kody liczbowe od 65 do 90.
- Małym literom od „a” do „z” odpowiadają kody liczbowe od 97 do 122.
- Cyfrą od „0” do „9” zapisanym jako znaki odpowiadają kody liczbowe od 48 do 57. Przykładowo ciągi znaków "abc123" odpowiadają kolejno kody 97, 98, 99, 49, 50 i 51.
- Znakowi spacji odpowiada kod o numerze 32.

Poza tym, że kodowanie ASCII nadaje każdemu znakowi odpowiedni kod liczbowy, ustawa także te znaki w odpowiedniej kolejności. Znak „A” znajduje się przed znakiem „B”, a ten znajduje się przez znakiem „C” itd.

Kiedy program porównuje znaki, to tak naprawdę porównuje kody liczbowe odpowiadające tym znakom. Spójrz na następujący przykład:

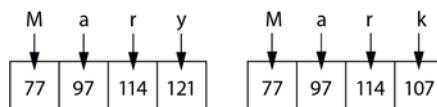
```
If "a" < "b" Then
    Display "Litera a jest mniejsza niż litera b."
End If
```

Instrukcja If sprawdza, czy kod liczbowy odpowiadający znakowi „a” jest mniejszy niż kod liczbowy odpowiadający znakowi „b”. Wyrażenie „a” < „b” zwraca prawdę, ponieważ kod liczbowy odpowiadający znakowi „a” jest mniejszy od kodu liczbowego odpowiadającego znakowi „b”. Tak więc w tym przypadku program wyświetliłby komunikat „Litera a jest mniejsza niż litera b.”.

Spójrzmy teraz, jak są porównywane ciągi znaków składające się z większej liczby znaków. Założymy, że mamy zapisane w pamięci ciągi znaków „Mary” i „Mark”:

```
Declare String name1 = "Mary"
Declare String name2 = "Mark"
```

Na rysunku 4.13 pokazałem, w jaki sposób ciągi znaków „Mary” i „Mark” będą zapisane w pamięci za pomocą kodowania ASCII.

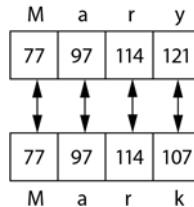


**Rysunek 4.13.** Kody liczbowe znaków w ciągach znaków „Mary” i „Mark”

Jeżeli do porównania ciągów znaków użyjemy operatorów relacji, zostaną one porównane znak po znaku. Spójrz na taki przykładowy pseudokod:

```
Declare String name1 = "Mary"
Declare String name2 = "Mark"
If name1 > name2 Then
    Display "Mary jest większe niż Mark."
Else
    Display "Mary nie jest większe niż Mark."
End If
```

Operator > porównuje ze sobą po kolejne każdy znak w ciągach znaków „Mary” i „Mark”, zaczynając od pierwszego. Pokazałem to na rysunku 4.14.



**Rysunek 4.14.** Porównanie poszczególnych znaków w ciągach znaków

Oto jak wygląda to porównanie:

1. Litera „M” w ciągu „Mary” jest porównywana z literą „M” w ciągu „Mark”. Ponieważ litery są identyczne, porównane zostaną kolejne znaki.

2. Litera „a” w ciągu "Mary" jest porównywana z literą „a” w ciągu "Mark". Ponieważ litery są identyczne, porównane zostaną kolejne znaki.
3. Litera „r” w ciągu "Mary" jest porównywana z literą „r” w ciągu "Mark". Ponieważ litery są identyczne, porównane zostaną kolejne znaki.
4. Litera „y” w ciągu "Mary" jest porównywana z literą „k” w ciągu "Mark". Ponieważ litery nie są takie same, ciągi znaków też nie są sobie równe. Znak „y” w systemie ASCII ma wyższy kod liczbowy (121) niż znak „k” (107), więc okazuje się, że ciąg "Mary" jest większy od ciągu "Mark".

Jeżeli jeden ciąg znaków będzie krótszy od drugiego, w wielu językach programowania zostaną porównane tylko odpowiadające sobie znaki. Jeżeli odpowiadające sobie znaki są takie same, wtedy krótszy ciąg będzie mniejszy od dłuższego. Założymy, że porównujemy ciągi znaków "Helikopter" i "Hel". W wyniku porównania obu ciągów znaków to ciąg "Hel" będzie mniejszy od ciągu "Helikopter", ponieważ jest krótszy.

Na listingu 4.4 pokazałem, jak można za pomocą operatora < porównać ze sobą dwa ciągi znaków. Użytkownik wprowadza dwa imiona i nazwiska, a program wyświetla je w kolejności alfabetycznej.

#### **Listing 4.4**



```

1 // Deklaracja zmiennych dla dwóch imion i nazwisk
2 Declare String name1
3 Declare String name2
4
5 // Prosimy użytkownika o wprowadzenie dwóch imion i nazwisk
6 Display "Wprowadź pierwsze nazwisko i imię."
7 Input name1
8 Display "Wprowadź drugie nazwisko i imię."
9 Input name2
10
11 // Wyświetlamy oba imiona i nazwiska w kolejności alfabetycznej
12 Display "Oto nazwiska i imiona w kolejności alfabetycznej:"
13 If name1 < name2 Then
14   Display name1
15   Display name2
16 Else
17   Display name2
18   Display name1
19 End If

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź pierwsze nazwisko i imię.

**Kowalski, Jan [Enter]**

Wprowadź drugie nazwisko i imię.

**Bartoszewska, Anna [Enter]**

Oto nazwiska i imiona w kolejności alfabetycznej:

Bartoszewska, Anna

Kowalski, Jan



## Punkt kontrolny

- 4.11. Co wyświetliłoby się na ekranie, gdyby poniższy pseudokod był prawdziwym programem?

```
If "z" < "a" Then
    Display "z jest mniejsze niż a."
Else
    Display "z nie jest mniejsze a."
End If
```

- 4.12. Co wyświetliłoby się na ekranie, gdyby poniższy pseudokod był prawdziwym programem?

```
Declare String s1 = "Nowy Jork"
Declare String s2 = "Boston"
If s1 > s2 Then
    Display s2
    Display s1
Else
    Display s1
    Display s2
End If
```

### 4.4

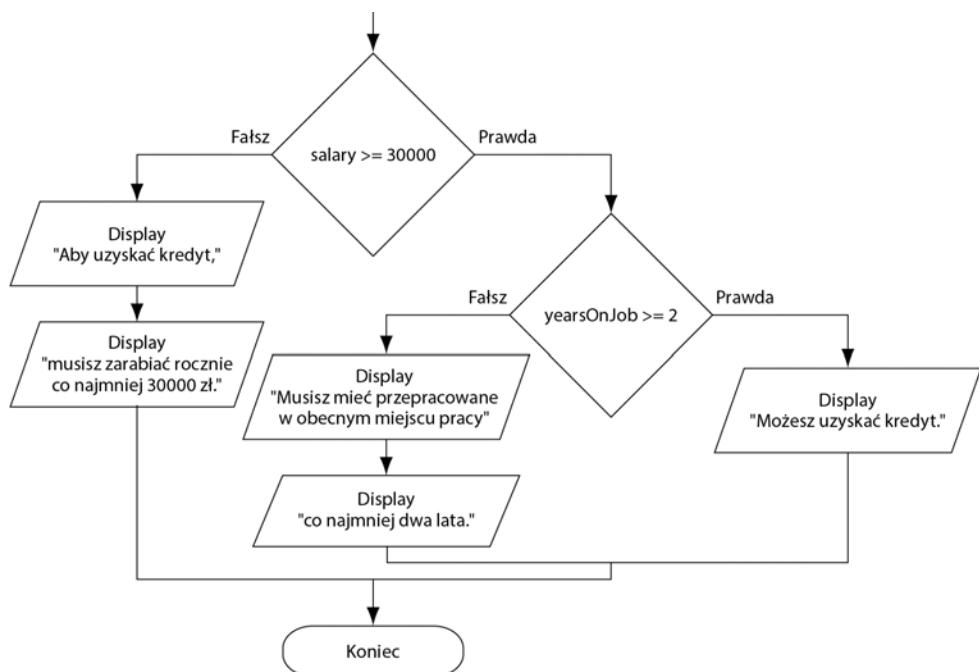
## Zagnieżdżone struktury warunkowe

**WYJAŚNIENIE:** Aby sprawdzić więcej niż jeden warunek, struktury warunkowe można zagnieżdzać.

W podrozdziale 4.1. wspomniałem, że programy projektuje się zazwyczaj jako kombinację różnych typów struktur sterujących. Przedstawiłem tam przykład struktury sekwencyjnej umieszczonej w strukturze warunkowej (rysunek 4.3). Można także zagnieżdzić jedną strukturę warunkową wewnętrz innej struktury warunkowej. Jest to wręcz wymagane, aby zaprojektować program, który będzie jednocześnie sprawdzał dwa warunki.

Przyjrzymy się przykładowemu programowi do sprawdzania zdolności kredytowej klienta. Aby otrzymać kredyt, trzeba spełnić łącznie dwa warunki: (1) należy zarabiać rocznie co najmniej 30000 zł i (2) należy być zatrudnionym w obecnym miejscu pracy przez okres co najmniej dwóch lat. Na rysunku 4.15 pokazałem schemat blokowy algorytmu takiego programu. Założmy, że w zmiennej `salary` jest zapisane roczne wynagrodzenie, a w zmiennej `yearsOnJob` — liczba lat przepracowanych przez potencjalnego kredytobiorcę w aktualnym miejscu pracy.

Jeśli prześledzimy ten schemat, to zobaczymy, że w pewnym miejscu sprawdzany jest warunek `salary >= 30000`. Jeżeli warunek ten nie jest spełniony, nie ma sensu sprawdzanie kolejnego warunku, ponieważ wiemy, że klient nie kwalifikuje się do uzyskania kredytu. Jeżeli jednak warunek jest spełniony, musimy jeszcze sprawdzić drugi warunek. Ma to miejsce w zagnieżdżonej strukturze warunkowej, gdzie sprawdzany jest warunek `yearsOnJob >= 2`. Jeżeli ten warunek jest także spełniony,



**Rysunek 4.15.** Zagnieżdżona struktura warunkowa

to klient kwalifikuje się do uzyskania kredytu. Jeżeli warunek nie jest spełniony, klient nie kwalifikuje się do uzyskania kredytu. Na listingu 4.5 przedstawiony jest pseudokod programu.

#### **Listing 4.5**

```

1 // Deklaracja zmiennych
2 Declare Real salary, yearsOnJob
3
4 // Pobieramy roczne wynagrodzenie
5 Display "Wprowadź swoje roczne wynagrodzenie."
6 Input salary
7
8 // Pobieramy liczbę lat przepracowanych w obecnym miejscu pracy
9 Display "Wprowadź liczbę lat przepracowanych"
10 Display "w obecnym miejscu pracy."
11 Input yearsOnJob
12
13 // Sprawdzamy, czy klient kwalifikuje się do uzyskania kredytu
14 If salary >= 30000 Then
15   If yearsOnJob >= 2 Then
16     Display "Możesz uzyskać kredyt."
17   Else
18     Display "Musisz mieć przepracowane w obecnym miejscu pracy"
19     Display "co najmniej dwa lata."
20   End If
21 Else
22   Display "Aby uzyskać kredyt,"
23   Display "musisz zarabiać rocznie co najmniej 30000 zł."
24 End If
  
```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**35000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**1 [Enter]**

Musisz mieć przepracowane w obecnym miejscu pracy co najmniej dwa lata.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**25000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**5 [Enter]**

Aby uzyskać kredyt, musisz zarabiać rocznie co najmniej 30000 zł.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**35000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**5 [Enter]**

Możesz uzyskać kredyt.

Przyjrzyj się instrukcji If-Then-Else, która zaczyna się w linii 14. Sprawdza ona warunek salary >= 30000. Jeżeli warunek ten zostanie spełniony, instrukcja If-Then-Else przejdzie do wykonywania instrukcji umieszczonej w linii 15. W przeciwnym wypadku program przeskoczy do klauzuli Else w linii 21. i wykona dwa polecenia Display w liniach 22. i 23. Następnie program opuści strukturę warunkową i zakończy działanie.

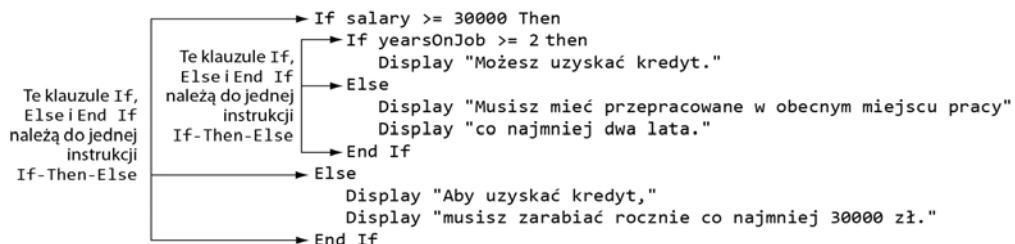
## Styl programowania a zagnieżdżone struktury warunkowe

Bardzo ważne jest, aby zapisując w kodzie zagnieżdżone struktury warunkowe, stosować odpowiednie wyrównanie i wcięcia tekstu. Dzięki temu łatwiej będzie zauważać w kodzie, które operacje są wykonywane w danej strukturze warunkowej. W większości języków programowania poniższy kod zachowa się dokładnie tak samo jak kod w liniach od 14. do 24. na listingu 4.5. Pomimo tego, że w kodzie nie ma żadnych błędów, będzie go bardzo trudno debugować w przypadku wystąpienia błędu, ponieważ linie kodu nie zostały odpowiednio wcięte.

```
If salary >= 30000 Then
  If yearsOnJob >= 2 Then
    Display "Możesz uzyskać kredyt."
  Else
    Display "Musisz mieć przepracowane w obecnym miejscu pracy"
    Display "co najmniej dwa lata."
  End If
  Else
    Display "Aby uzyskać kredyt,"
    Display "musisz zarabiać rocznie co najmniej 30000 zł."
  End If
```

**Nie zapisuj pseudokodu w taki sposób!**

Prawidłowe wyrównanie tekstu i wcięcia w kodzie powodują także, że łatwiej jest śledzić poszczególne klauzule If, Else i End If, co pokazałem na rysunku 4.16.



**Rysunek 4.16.** Rozmieszczenie klauzul If, Else i End If

## Sprawdzanie kilku warunków

W poprzednim przykładzie przedstawiłem program, w którym do sprawdzenia kilku warunków wykorzystałem zagnieżdżone struktury warunkowe. Sprawdzanie kilku warunków i w zależności od otrzymanego wyniku wykonywanie określonych operacji jest bardzo częstą praktyką podczas programowania. Jednym z rozwiązań tego problemu jest użycie wielu zagnieżdżonych struktur warunkowych. Zilustruję to przykładem, który przedstawiłem w sekcji „W centrum uwagi”.

### W centrum uwagi

Wielokrotne zagnieżdżenie struktur warunkowych



Doktor Sobotka uczy w szkole języka polskiego i do oceniania sprawdzianów napisanych przez uczniów wykorzystuje następujący system:

Liczba punktów	Ocena
90 i więcej	5
80 – 89	4
70 – 79	3
60 – 69	2
Poniżej 60	1

Doktor Sobotka poprosił Cię o napisanie programu, który pozwoli uczniom wprowadzić liczbę punktów ze sprawdzianu i wyświetli otrzymaną ocenę. Oto algorytm, z którego skorzystasz:

1. Poproś użytkownika, aby wprowadził liczbę punktów ze sprawdzianu.
2. Sprawdź ocenę w następujący sposób:

Jeżeli liczba punktów jest mniejsza niż 60, ocena to 1.

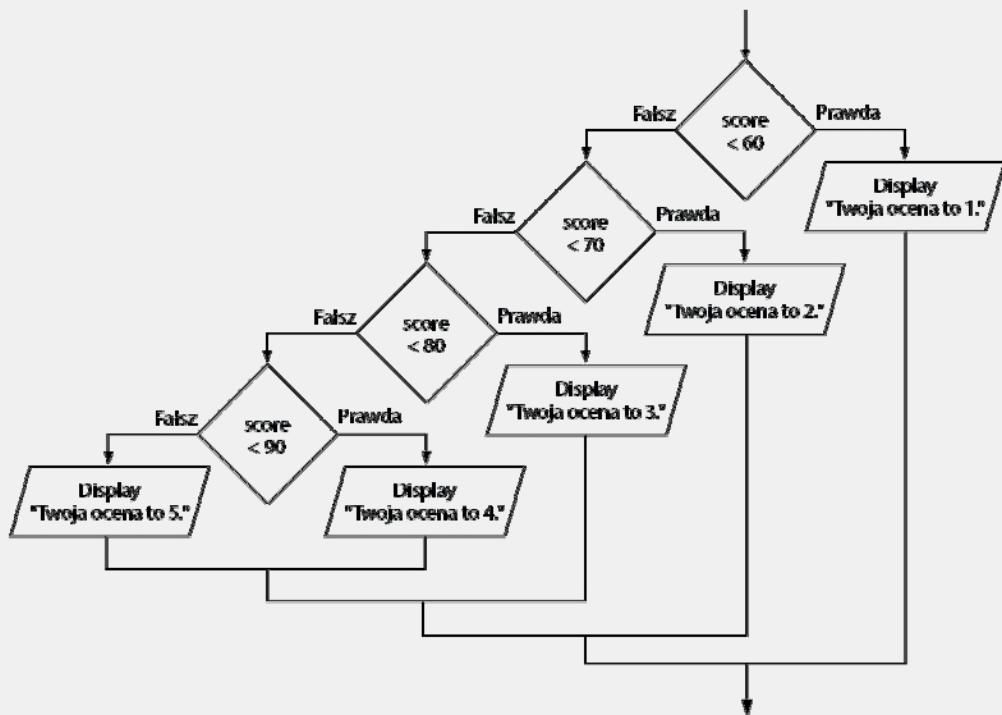
W przeciwnym wypadku, jeżeli liczba punktów jest mniejsza niż 70, ocena to 2.

W przeciwnym wypadku, jeżeli liczba punktów jest mniejsza niż 80, ocena to 3.

W przeciwnym wypadku, jeżeli liczba punktów jest mniejsza niż 90, ocena to 4.

W przeciwnym wypadku ocena to 5.

Doszedłeś do wniosku, że żeby ustalić ocenę, musisz skorzystać z wielu zagnieżdzonych struktur warunkowych, tak jak przedstawia to rysunek 4.17. Na listingu 4.6 znajduje się pseudokod programu. Kod obejmujący zagnieżdżone struktury warunkowe zawiera się w liniach od 9. do 25.



Rysunek 4.17. Określanie oceny za pomocą zagnieżdzonych struktur warunkowych

#### Listing 4.6



```

1 // Zmienna, w której zapisana jest liczba punktów
2 Declare Real score
3
4 // Pobieramy liczbę punktów
5 Display "Wprowadź liczbę punktów ze sprawdzianu."
6 Input score
7
8 // Określamy ocenę
9 If score < 60 Then
10   Display "Twoja ocena to 1."
11 Else
12   If score < 70 Then
13     Display "Twoja ocena to 2."
14   Else
15     If score < 80 Then

```

```

16     Display "Twoja ocena to 3."
17 Else
18     If score < 90 Then
19         Display "Twoja ocena to 4."
20     Else
21         Display "Twoja ocena to 5."
22     End If
23 End If
24 End If
25 End If

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę punktów ze sprawdzianu.

**78 [Enter]**

Twoja ocena to 3.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę punktów ze sprawdzianu.

**84 [Enter]**

Twoja ocena to 4.

## Instrukcja If-Then-Else If

Mimo że program z listingu 4.6 wygląda na prosty, logika wynikająca z wykorzystania wielu zagnieżdżonych struktur warunkowych jest całkiem złożona. W większości języków programowania dostępna jest jednak struktura warunkowa zwana If-Then-Else If, dzięki której zapisanie tego typu logiki w kodzie jest znacznie prostsze. W przypadku pseudokodu instrukcję If-Then-Else If będziemy zapisywać w następujący sposób:

```

If warunek_1 Then
    instrukcja
    instrukcja
    itd. } Jeżeli będzie spełniony warunek_1, wykonają się te instrukcje,
          a reszta struktury zostanie pominięta.

    Else If warunek_2 Then
        instrukcja
        instrukcja
        itd. } Jeżeli będzie spełniony warunek_2, wykonają się te instrukcje,
              a reszta struktury zostanie pominięta.

        tutaj możesz wstawić tyle klauzul Else If, ile będziesz potrzebować
        Else
            instrukcja
            instrukcja
            itd. } Te instrukcje wykonają się wtedy, gdy żaden z powyższych
                  warunków nie będzie spełniony.

    End If

```

Kiedy uruchomisz to polecenie, najpierw program sprawdzi *warunek\_1*. Jeżeli *warunek\_1* jest spełniony, program wykona instrukcje umieszczone bezpośrednio za tym warunkiem, aż do klauzuli *Else If*. Pozostała część struktury zostanie pominięta. Jeżeli *warunek\_1* nie jest spełniony, program przejdzie do następnej klauzuli *Else If* i sprawdzi *warunek\_2*. Jeżeli jest on spełniony, program wykona instrukcje umieszczone bezpośrednio za tym warunkiem, aż do kolejnej klauzuli *Else If*. Pozostała część struktury zostanie pominięta. Proces ten będzie się powtarzał aż do

momentu, gdy spełniony będzie któryś z warunków lub program dojdzie do klauzuli `Else`. Jeżeli żaden warunek nie jest spełniony, program uruchomi instrukcje umieszczone za klauzulą `Else`.

Na listingu 4.7 przedstawiłem przykładowy pseudokod z instrukcją `If-Then-Else If`. Działa on dokładnie tak samo jak program z listingu 4.6. Jednak zamiast zagnieżdżonych struktur warunkowych użyliśmy tutaj w liniach od 9. do 19. instrukcji `If-Then-Else If`.

### **Listing 4.7**

```

1 // Zmienna, w której zapisana jest liczba punktów
2 Declare Real score
3
4 // Pobieramy wynik
5 Display "Wprowadź liczbę punktów ze sprawdzianu."
6 Input score
7
8 // Określamy ocenę
9 If score < 60 Then
10   Display "Twoja ocena to 1."
11 Else If score < 70 Then
12   Display "Twoja ocena to 2."
13 Else If score < 80 Then
14   Display "Twoja ocena to 3."
15 Else If score < 90 Then
16   Display "Twoja ocena to 4."
17 Else
18   Display "Twoja ocena to 5."
19 End If

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę punktów ze sprawdzianu.

**78 [Enter]**

Twoja ocena to 3.

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę punktów ze sprawdzianu.

**84 [Enter]**

Twoja ocena to 4.

Zwróć uwagę na to, w jaki sposób jest teraz ułożony i wcięty kod instrukcji `If-Then-Else If`: klauzule `If`, `Else If`, `Else` i `End If` są wyrównane, a instrukcje, które będą wykonywane warunkowo, są wcięte.

Nie musisz korzystać z instrukcji `If-Then-Else If`, ponieważ to samo można osiągnąć, korzystając z zagnieżdżonych instrukcji `If-Then-Else`. Jednak używanie dużej liczby zagnieżdżonych instrukcji `If-Then-Else` ma kilka wad:

- Kod programu stanie się bardzo złożony i ciężko go będzie zrozumieć.
- Ponieważ w przypadku zagnieżdżonych instrukcji bardzo ważne jest odpowiednie wcinanie linii, to w przypadku bardzo dużej liczby takich instrukcji kod programu stanie się bardzo rozległy i linia kodu stanie się tak szeroka, że aby ją wyświetlić,

trzeba będzie przewijać okno edytora na boki. Ponadto długie linie kodu mają tendencję do zawijania się podczas drukowania, co czyni kod jeszcze mniej czytelnym.

Logikę kryjącą się za instrukcją If-Then-Else If jest znacznie łatwiej śledzić w programie niż zagnieżdżone instrukcje If-Then-Else. Ponadto, ponieważ wszystkie klauzule w instrukcji If-Then-Else If są wyrównane, długość linii kodu w takim przypadku jest krótsza.



## Punkt kontrolny

- 4.13. W jaki sposób działa struktura warunkowa podwójnego wyboru?
- 4.14. Za pomocą której instrukcji można w pseudokodzie utworzyć strukturę warunkową podwójnego wyboru?
- 4.15. Kiedy wykonają się polecenia zawarte między klauzulami Else i End If w przypadku instrukcji If-Then-Else?
- 4.16. Zamień następujący kod na instrukcję If-Then-Else If:

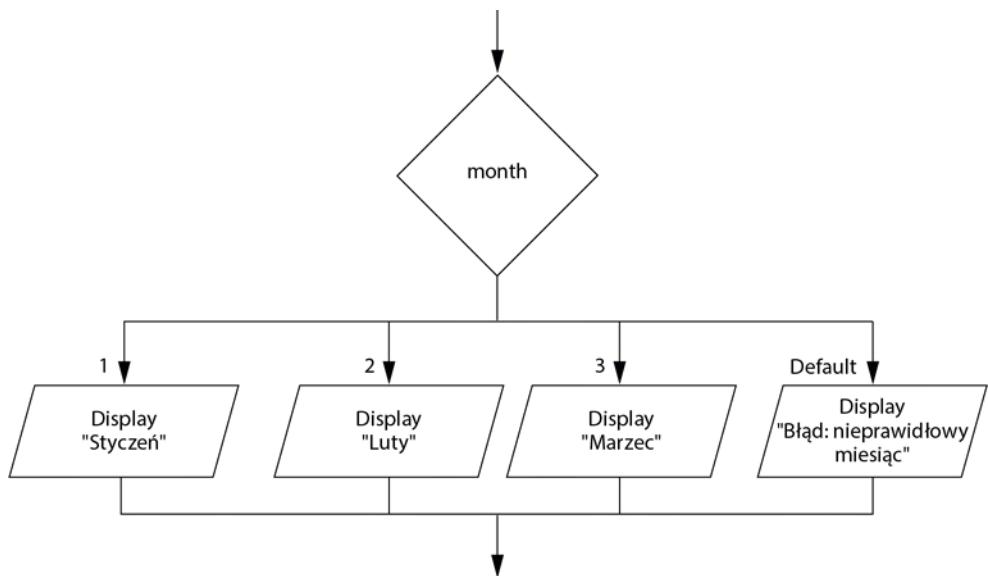
```
If number == 1 Then
    Display "Jeden"
Else
    If number == 2 Then
        Display "Dwa"
    Else
        If number == 3 Then
            Display "Trzy"
        Else
            Display "Brak danych"
        End If
    End If
End If
```

### 4.5

## Struktura decyzyjna

**WYJAŚNIENIE:** Dzięki strukturze decyzyjnej można na podstawie wyrażenia lub wartości zapisanej w zmiennej decydować o tym, w jaki sposób będzie przebiegało wykonanie programu.

Struktura decyzyjna (ang. *case structure*) to przykład struktury warunkowej wielokrotnego wyboru (ang. *multiple alternative decision structure*). Dzięki niej można sprawdzić wartość wyrażenia lub zmiennej i w zależności od tej wartości wykonać w programie określone instrukcje. W wielu przypadkach struktura decyzyjna zachowuje się tak samo jak zagnieżdżone instrukcje If-Else, ale tworzy znacznie prostszy i bardziej przejrzysty kod. Na rysunku 4.18 przedstawiłem na schemacie decyzyjnym, jak wygląda struktura decyzyjna.

**Rysunek 4.18.** Struktura decyzyjna

Na schemacie można zauważyć, że w rombie jest umieszczona nazwa zmiennej. Jeżeli do zmiennej jest przypisana wartość 1, uruchomi się instrukcja `Display "Styczeń"`. Jeżeli do zmiennej jest przypisana wartość 2, uruchomi się instrukcja `Display "Luty"`. Jeżeli do zmiennej jest przypisana wartość 3, uruchomi się instrukcja `Display "Marzec"`. Jeżeli do zmiennej nie jest przypisana żadna z wymienionych wartości, uruchomi się kod oznaczony etykietą `Default`. W tym przypadku będzie to instrukcja `Display "Błąd: nieprawidłowy miesiąc"`.

Do zapisania struktury decyzyjnej w pseudokodzie będziemy używali instrukcji `Select Case`. Oto ogólna postać takiej instrukcji:

```

Select wyrażenie ← Tutaj jest wyrażenie lub zmienna.
Case wartość_1:
  instrukcja
  instrukcja
  itd. } Te instrukcje wykonają się wtedy, gdy wyrażenie będzie
        równe wartość_1.

Case wartość_2:
  instrukcja
  instrukcja
  itd. } Te instrukcje wykonają się wtedy, gdy wyrażenie będzie
        równe wartość_2.

tutaj możesz wstawić tyle klauzul Case, ile będziesz potrzebować

Case wartość_N:
  instrukcja
  instrukcja
  itd. } Te instrukcje wykonają się wtedy, gdy wyrażenie będzie
        równe wartość_N.

Default:
  instrukcja
  instrukcja
  itd. } Te instrukcje wykonają się wtedy, gdy wyrażenie nie będzie
        równe żadnej z wartości wymienionych wcześniej.

End Select ← Tutaj instrukcja się kończy.
  
```

Pierwsza linia struktury zaczyna się od słowa `Select`, a następnie pojawia się *wyrażenie*. *Wyrażenie* to zazwyczaj zmienna, ale w wielu językach programowania może to być wszystko to, co zwraca jakąkolwiek wartość (np. wyrażenie matematyczne). Wewnątrz struktury znajduje się jeden lub więcej bloków instrukcji zaczynających się od słowa `Case`. Zwróć uwagę, że za słowem `Case` umieszczona jest wartość.

Kiedy uruchomisz się instrukcję `Select Case`, *wyrażenie* zostanie porównane z wartością znajdująjącą się przy instrukcji `Case` — z góry na dół. Jeśli okaże się, że jedna z tych wartości jest taka sama jak wartość zwracana przez *wyrażenie*, program przejdzie do tej instrukcji, zacznie uruchamiać instrukcje umieszczone bezpośrednio pod nią, a następnie opuści strukturę. Jeżeli okaże się, że żadna z wartości nie jest równa wartości zwracanej przez *wyrażenie*, program przejdzie do instrukcji `Default` i zacznie wykonywać instrukcje znajdujące się pod nią.

Poniższy przykładowy pseudokod wykonuje te same operacje, które są zamieszczone na schemacie blokowym na rysunku 4.18.

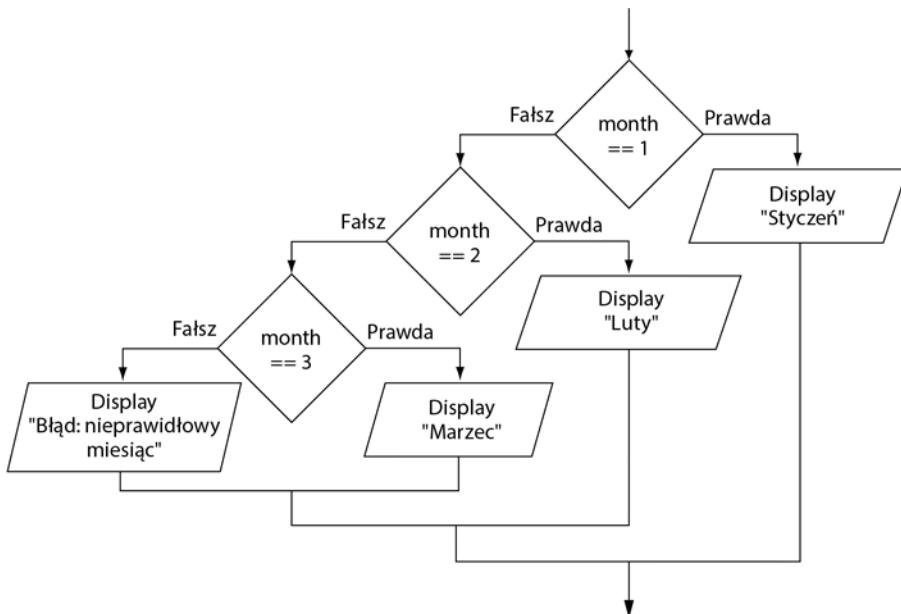
```
Select month
    Case 1:
        Display "Styczeń"
    Case 2:
        Display "Luty"
    Case 3:
        Display "Marzec"
    Default:
        Display "Błąd: nieprawidłowy miesiąc"
End Select
```

W tym przypadku *wyrażenie* to zmienna `month`. Jeśli zmienna `month` jest równa 1, program przeskoczy do instrukcji `Case 1:` i wykona umieszczone pod nią polecenie `Display "Styczeń"`. Jeśli zmienna `month` jest równa 2, program przeskoczy do instrukcji `Case 2:` i wykona umieszczone pod nią polecenie `Display "Luty"`. Jeśli zmienna `month` jest równa 3, program przeskoczy do instrukcji `Case 3:` i wykona umieszczone pod nią polecenie `Display "Marzec"`. Jeżeli zmienna `month` nie jest równa 1, 2 ani 3, program przeskoczy do instrukcji `Default:` i wykona umieszczone pod nią polecenie `Display "Błąd: nieprawidłowy miesiąc"`.



**UWAGA:** W wielu językach programowania struktura decyzyjna jest reprezentowana przez instrukcję `switch`.

Nie musisz korzystać ze struktur decyzyjnych, ponieważ taki sam efekt można uzyskać za pomocą zagnieżdzonych struktur warunkowych. Na rysunku 4.19 przedstawiłem zagnieżdżone struktury warunkowe odpowiadające strukturze decyzyjnej z rysunku 4.18. Struktury decyzyjne wyglądają jednak w kodzie znacznie prościej.



**Rysunek 4.19.** Zagnieżdżone struktury warunkowe

## W centrum uwagi

### Korzystanie ze struktury decyzyjnej

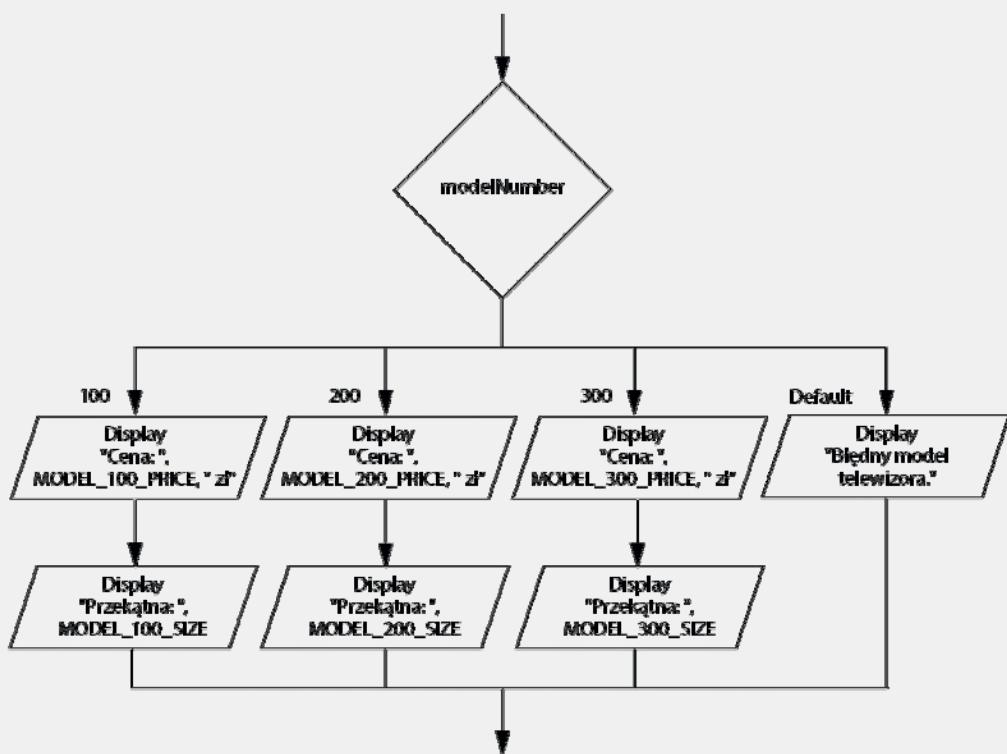
Leszek jest właścicielem sklepu Audio & Video i poprosił Cię o napisanie programu, dzięki któremu klient będzie mógł wybrać jeden z trzech modeli telewizorów, a program wyświetli jego cenę i przekątną ekranu. Oto algorytm:

1. Pobierz model telewizora.
2. Jeżeli model telewizora to 100, wyświetl informacje o tym modelu.

W przeciwnym wypadku, jeżeli model telewizora to 200, wyświetl informacje o tym modelu.

W przeciwnym wypadku, jeżeli model telewizora to 300, wyświetl informacje o tym modelu.

Na początku do określenia modelu telewizora i wyświetlenia odpowiednich informacji chciałeś wykorzystać zagnieżdżone struktury warunkowe. Doszczętnie jednak do wniosku, że równie dobrze sprawdzi się tutaj struktura decyzyjna, ponieważ na podstawie jednej wartości (modelu telewizora) będziesz mógł wybrać operacje, jakie wykona program. Model telewizora możemy zapisać w zmiennej i będziemy ją sprawdzać w strukturze decyzyjnej. Założymy, że model telewizora zapiszemy w zmiennej o nazwie `modelNumber`. Na rysunku 4.20 przedstawiłem schemat blokowy struktury decyzyjnej. Na listingu 4.8 widoczny jest pseudokod programu.



Rysunek 4.20. Schemat blokowy struktury decyzyjnej

### Listing 4.8



```

1 // Stałe z ceną wybranych modeli telewizorów
2 Constant Real MODEL_100_PRICE = 1999.99
3 Constant Real MODEL_200_PRICE = 2699.99
4 Constant Real MODEL_300_PRICE = 3499.99
5
6 // Stałe z przekątną ekranu wybranych modeli telewizorów
7 Constant Integer MODEL_100_SIZE = 24
8 Constant Integer MODEL_200_SIZE = 27
9 Constant Integer MODEL_300_SIZE = 32
10
11 // Zmienna z modelem telewizora
12 Declare Integer modelNumber
13
14 // Pobieramy model telewizora
15 Display "Który telewizor Ci się podoba?"
16 Display "Model 100, 200 czy 300?"
17 Input modelNumber
18
19 // Wyświetlamy cenę i przekątną ekranu
20 Select modelNumber
21 Case 100:
22     Display "Cena: ", MODEL_100_PRICE, " zł"
23     Display "Przekątna: ", MODEL_100_SIZE
24 Case 200:

```

```

25     Display "Cena: ", MODEL_200_PRICE, " zł"
26     Display "Przekątna: ", MODEL_200_SIZE
27 Case 300:
28     Display "Cena: ", MODEL_300_PRICE, " zł"
29     Display "Przekątna: ", MODEL_300_SIZE
30 Default:
31     Display "Błędny model telewizora."
32 End Select

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Który telewizor Ci się podoba?

Model 100, 200 czy 300?

**100 [Enter]**

Cena: 1999.99 zł

Przekątna: 24

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Który telewizor Ci się podoba?

Model 100, 200 czy 300?

**200 [Enter]**

Cena: 2699.99 zł

Przekątna: 27

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Który telewizor Ci się podoba?

Model 100, 200 czy 300?

**300 [Enter]**

Cena: 3499.99 zł

Przekątna: 32

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Który telewizor Ci się podoba?

Model 100, 200 czy 300?

**500 [Enter]**

Błędny model telewizora.



**UWAGA:** Struktury decyzyjne w wielu językach programowania się od siebie różnią. Ponieważ w każdym języku obowiązują inne reguły związane z zapisem struktury decyzyjnej, może się okazać, że dla pewnych struktur warunkowych wielokrotnego wyboru nie da się stworzyć instrukcji w danym języku. Możesz w takiej sytuacji skorzystać z instrukcji If-Then-Else If lub z zagnieżdżonych struktur warunkowych.

**Punkt kontrolny**

- 4.17. Co to jest struktura warunkowa wielokrotnego wyboru?
- 4.18. W jaki sposób zapisujemy w pseudokodzie strukturę warunkową wielokrotnego wyboru?
- 4.19. Co sprawdza struktura decyzyjna, aby określić, które instrukcje powinny się uruchomić?

- 4.20. Musisz stworzyć w programie strukturę warunkową wielokrotnego wyboru, ale w języku, z którego korzystasz, nie da się sprawdzić wartości wyrażenia, które masz zamiar umieścić w instrukcji `Select Case`. W jaki inny sposób możesz wykonać to samo zadanie?

## 4.6

# Operatory logiczne

**WYJAŚNIENIE:** Dzięki operatorom logicznym **AND** i **OR** można tworzyć rozbudowane wyrażenia boolowskie poprzez połączenie ze sobą kilku innych wyrażeń. Operator **NOT** neguje wartość zwracaną przez wyrażenie boolowskie.

Języki programowania wyposażone są także w **operatorы logiczne** (ang. *logical operators*), dzięki którym można tworzyć bardziej rozbudowane wyrażenia boolowskie. W tabeli 4.3 przedstawiłem te operatory.

**Tabela 4.3.** Operatory logiczne

Operator	Znaczenie
AND	Operator AND łączy dwa wyrażenia boolowskie w jedno złożone wyrażenie. Aby takie złożone wyrażenie zwracało prawdę, oba podwyrażenia także muszą zwracać prawdę.
OR	Operator OR łączy dwa wyrażenia boolowskie w jedno złożone wyrażenie. Aby takie złożone wyrażenie zwracało prawdę, jedno z dwóch podwyrażeń musi zwracać prawdę. Wystarczy, że tylko jedno, którekolwiek z wyrażeń będzie zwracało prawdę.
NOT	Operator NOT jest operatorem jednoargumentowym, co znaczy, że wymaga tylko jednego operandu. Operand musi być wyrażeniem boolowskim. Operator NOT neguje wartość wyrażenia — jeżeli wyrażenie zwraca prawdę, operator NOT zwróci fałsz, a jeżeli wyrażenie zwraca fałsz, operator NOT zwróci prawdę.

W tabeli 4.4 przedstawiłem kilka złożonych wyrażeń boolowskich.

**Tabela 4.4.** Złożone wyrażenia boolowskie zawierające operatory logiczne

Wyrażenie	Znaczenie
$x > y \text{ AND } a < b$	Czy $x$ jest większe niż $y$ i $a$ jest mniejsze niż $b$ ?
$x == y \text{ OR } x == z$	Czy $x$ jest równe $y$ lub $x$ jest równe $z$ ?
$\text{NOT } (x > y)$	Czy wyrażenie $x > y$ zwraca fałsz?



**UWAGA:** W wielu językach programowania, w tym w C, C++ i Java, operator AND zapisuje się jako `&&`, operator OR jako `||`, a operator NOT jako `!`.

## Operator AND

Operator AND przyjmuje jako operandy dwa wyrażenia boolowskie i tworzy z nich złożone wyrażenie boolowskie, które będzie zwracało prawdę tylko w przypadku, gdy oba podwyrażenia także zwracają prawdę. Oto przykład instrukcji If-Then, w której wykorzystuję operator AND:

```
If temperature < 20 AND minutes > 12 Then
    Display "Uwaga! Produkt może być niezdatny do spożycia."
End If
```

W instrukcji tej połączylem dwa wyrażenia boolowskie, `temperature < 20` i `minutes > 12`, i stworzyłem jedno wyrażenie złożone. Polecenie `Display` wykona się tylko wtedy, gdy zarówno wartość zapisana w zmiennej `temperature` będzie mniejsza niż 20, jak i wartość zapisana w zmiennej `minutes` będzie większa niż 12. Jeżeli którekolwiek z tych wyrażeń zwróci fałsz, wyrażenie złożone także zwróci fałsz i komunikat się nie wyświetli.

W tabelce 4.5 zamieściłem tablicę prawdy operatora AND. W tablicy prawdy znajdują się wszystkie możliwe kombinacje wartości typu prawda i fałsz operandów dla operatora AND. Widoczna jest także wartość zwracana przez takie wyrażenie.

**Tabela 4.5.** Tablica prawdy operatora AND

Wyrażenie	Wartość wyrażenia
prawda AND fałsz	fałsz
fałsz AND prawda	fałsz
fałsz AND fałsz	fałsz
prawda AND prawda	prawda

Jak widać, aby wyrażenie z operatorem AND zwróciło prawdę, oba operandy także muszą zwracać prawdę.

## Operator OR

Operator OR przyjmuje jako operandy dwa wyrażenia boolowskie i tworzy z nich złożone wyrażenie boolowskie, które będzie zwracało prawdę tylko w przypadku, gdy którekolwiek z podwyrażeń także zwraca prawdę. Oto przykład instrukcji If-Then, w której wykorzystuję operator OR:

```
If temperature < 20 OR minutes > 100 Then
    Display "Uwaga! Produkt może być niezdatny do spożycia."
End If
```

Polecenie `Display` uruchomi się tylko wtedy, gdy wyrażenie `temperature < 20` zwróci prawdę lub wyrażenie `minutes > 100` zwróci prawdę. Wystarczy więc, aby tylko jedno z wyrażeń zwracało prawdę. W tabelce 4.6 zamieściłem tablicę prawdy operatora OR.

**Tabela 4.6.** Tablica prawdy operatora OR

Wyrażenie	Wartość wyrażenia
prawda OR fałsz	prawda
fałsz OR prawda	prawda
fałsz OR fałsz	fałsz
prawda OR prawda	prawda

Aby wyrażenie z operatorem OR zwróciło prawdę, wystarczy, że tylko jeden z operandów będzie zwracał prawdę — nie ma znaczenia który.

### Optymalizacja określania wartości wyrażenia

Wiele języków programowania używa do określania wartości wyrażenia z operatorem AND lub OR techniki optymalizacyjnej zwanej po angielsku **short-circuit evaluation**. Oto jak działa ona w przypadku operatora AND: jeżeli wyrażenie po lewej stronie operatora AND zwraca fałsz, wtedy wyrażenie po prawej stronie operatora AND nie zostanie w ogóle sprawdzone. Ponieważ w przypadku operatora AND wyrażenie złożone zwraca fałsz, gdy którykolwiek z operandów zwraca fałsz, obliczanie wartości drugiego podwyrażenia byłoby marnowaniem czasu procesora. Dlatego w sytuacji, gdy przy operatorze AND z lewej strony pojawi się wyrażenie zwracające fałsz, pominięte zostanie drugie podwyrażenie.

A oto jak działa optymalizacja w przypadku operatora OR: jeżeli wyrażenie po lewej stronie operatora OR zwraca prawdę, wtedy wyrażenie po prawej stronie operatora OR nie zostanie w ogóle sprawdzone. Ponieważ w przypadku operatora OR wystarczy, aby tylko jedno z podwyrażeń zwracało prawdę, obliczanie wartości drugiego podwyrażenia także byłoby marnowaniem czasu procesora.

## Operator NOT

Operator NOT jest operatorem jednoargumentowym. Przyjmuje on wyrażenie boolowskie i zwraca jego zanegowaną wartość logiczną. Innymi słowy: jeżeli wyrażenie zwraca prawdę, operator NOT zwróci fałsz, a jeżeli wyrażenie zwraca fałsz, operator NOT zwróci prawdę. Oto przykład instrukcji If-Then z wykorzystaniem operatora NOT:

```
If NOT(temperature > 100) Then
    Display "Temperatura jest niższa od maksymalnej."
End If
```



**WSKAZÓWKA:** Jeżeli będziesz pisać program w którymś z języków wykorzystujących opisaną technikę optymalizacji, przemyśl kolejność, w jakiej umieścisz podwyrażenia boolowskie w wyrażeniu złożonym. Jeżeli jesteś w stanie określić, które z podwyrażeń ma większe szanse zwracać prawdę, może to nieco poprawić wydajności programu.

Przykładowo możesz nieco poprawić wydajność programu, jeśli w instrukcji z operatorem OR warunek, który ma większe szanse zwracać prawdę, umieścisz po jego lewej stronie. W takiej sytuacji w wielu przypadkach podwyrażenie po prawej stronie operatora OR nie będzie w ogóle sprawdzane i program będzie działał szybciej.

Natomiast w przypadku instrukcji z operatorem AND możesz poprawić nieco wydajność programu, jeżeli warunek, który ma mniejsze szanse zwracać prawdę, umieścisz po jego lewej stronie. W takiej sytuacji w wielu przypadkach podwyrażenie po prawej stronie operatora AND nie będzie w ogóle sprawdzane i program będzie działał szybciej.

Najpierw sprawdzana jest wartość zwracana przez wyrażenie `temperature > 100` — będzie to prawda albo fałsz. Następnie wartość ta przekazywana jest do operatora NOT. Jeżeli wyrażenie `temperature > 100` zwraca prawdę, operator NOT zwróci fałsz. Jeżeli wyrażenie `temperature > 100` zwraca fałsz, operator NOT zwróci prawdę. Warunek ten można porównać do pytania: „Czy temperatura NIE jest większa niż 100”?



**UWAGA:** W przykładzie tym wyrażenie `temperature > 100` umieściłem w nawiasach. Zrobilem to dlatego, że w wielu językach programowania operator NOT ma pierwszeństwo przed operatorami relacji. Założymy, że zapisalibyśmy to wyrażenie w sposób następujący:

`NOT temperature > 100`

W przypadku wielu języków wyrażenie to nie zadziała prawidłowo, ponieważ program zastosuje operator NOT do zmiennej `temperature`, a nie do wyrażenia `temperature > 100`. Aby mieć pewność, że operator zostanie zastosowany do wyrażenia, ująłem je w nawias.

W tabeli 4.7 zamieściłem tablicę prawdy operatora NOT.

**Tabela 4.7.** Tablica prawdy operatora NOT

Wyrażenie	Wartość wyrażenia
NOT prawda	fałsz
NOT fałsz	prawda

## Poprawiony program do sprawdzania zdolności kredytowej

W pewnych sytuacjach dzięki operatorowi AND możemy nieco uprosić zagnieżdżone struktury warunkowe. Przypomnij sobie program do sprawdzania zdolności kredytowej klienta zamieszczony na listingu 4.5. Użyliśmy tam zagnieżdżonych instrukcji If-Then-Else:

```
If salary >= 30000 Then
    If yearsOnJob >= 2 Then
        Display "Możesz uzyskać kredyt."
    Else
        Display "Musisz mieć przepracowane w obecnym miejscu pracy"
        Display "co najmniej dwa lata."
    End If
Else
    Display "Aby uzyskać kredyt,"
    Display "musisz zarabiać rocznie co najmniej 30000 zł."
End If
```

Zadaniem tej struktury warunkowej było sprawdzenie, czy dana osoba zarabia rocznie co najmniej 30000 zł i czy przepracowała w obecnym miejscu pracy co najmniej dwa lata. Na listingu 4.9 pokazałem, jak można osiągnąć ten sam efekt za pomocą znacznie prostszego kodu.

### Listing 4.9



```
1 // Deklaracja zmiennych
2 Declare Real salary, yearsOnJob
3
4 // Pobieramy roczne wynagrodzenie
5 Display "Wprowadź swoje roczne wynagrodzenie."
6 Input salary
7
8 // Pobieramy liczbę lat przepracowanych w obecnym miejscu pracy
9 Display "Wprowadź liczbę lat przepracowanych",
10      "w obecnym miejscu pracy."
11 Input yearsOnJob
12
13 // Sprawdzamy, czy klient kwalifikuje się do uzyskania kredytu
14 If salary >= 30000 AND yearsOnJob >= 2 Then
15     Display "Możesz uzyskać kredyt."
16 Else
17     Display "Nie możesz uzyskać kredytu."
18 End If
```

### Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)

Wprowadź swoje roczne wynagrodzenie.

**35000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**1 [Enter]**

Nie możesz uzyskać kredytu.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**25000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**5 [Enter]**

Nie możesz uzyskać kredytu.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**35000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**5 [Enter]**

Możesz uzyskać kredyt.

W liniach od 14. do 18. instrukcja If-Then-Else sprawdza warunek złożony salary  $\geq 30000$  AND yearsOnJob  $\geq 2$ . Jeżeli oba podwyrażenia zwrócią prawdę, wyrażenie złożone także zwróci prawdę i wyświetli się komunikat *Możesz uzyskać kredyt*. Jeżeli jednak któryś z podwyrażeń zwróci fałsz, wyrażenie złożone także zwróci fałsz i wyświetli się komunikat *Nie możesz uzyskać kredytu*.



**UWAGA:** Wnikliwy Czytelnik zauważa zapewne, że programy z listingów 4.9 i 4.5 są bardzo podobne, ale nie identyczne. W przypadku programu z listingu 4.9, jeżeli klient nie kwalifikuje się do uzyskania kredytu, program wyświetla komunikat *Nie możesz uzyskać kredytu*, a program z listingu 4.5 w takiej sytuacji wyświetli jeden z dwóch komunikatów informujących klienta, z jakiego powodu nie kwalifikuje się do uzyskania kredytu.

## Kolejny program do sprawdzania zdolności kredytowej

Załóżmy, że pewien bank traci klientów na rzecz innego, konkurencyjnego banku, który nie jest aż tak restrykcyjny, jeżeli chodzi o udzielanie kredytów. Bank postanawia więc zmienić nieco warunki decydujące o zdolności kredytowej klienta. Od tej pory klient musi spełniać tylko jeden z postawionych warunków, a nie oba. Na listingu 4.10 przedstawiłem zmodyfikowaną wersję programu. W linii 14., w wyrażeniu złożonym sprawdzanym w instrukcji If-Then-Else, używam teraz operatora OR.

### **Listing 4.10**



```

1 // Deklaracja zmiennych
2 Declare Real salary, yearsOnJob
3
4 // Pobieramy roczne wynagrodzenie
5 Display "Wprowadź swoje roczne wynagrodzenie."
6 Input salary
7
8 // Pobieramy liczbę lat przepracowanych w obecnym miejscu pracy
9 Display "Wprowadź liczbę lat przepracowanych"
10 Display "w obecnym miejscu pracy."
11 Input yearsOnJob

```

```

12
13 // Sprawdzamy, czy klient kwalifikuje się do uzyskania kredytu
14 If salary >= 30000 OR yearsOnJob >= 2 Then
15   Display "Możesz uzyskać kredyt."
16 Else
17   Display "Nie możesz uzyskać kredytu."
18 End If

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**35000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**1 [Enter]**

Możesz uzyskać kredyt.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**25000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**5 [Enter]**

Możesz uzyskać kredyt.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź swoje roczne wynagrodzenie.

**12000 [Enter]**

Wprowadź liczbę lat przepracowanych w obecnym miejscu pracy.

**1 [Enter]**

Nie możesz uzyskać kredytu.

## Sprawdzanie przedziału liczbowego za pomocą operatorów logicznych

Niekiedy trzeba zaprojektować algorytm, który sprawdza, czy dana wartość mieści się w określonym przedziale liczbowym. W takiej sytuacji najlepiej użyć operatora AND. Przykładowo poniższa instrukcja If-Then sprawdza, czy wartość przypisana do zmiennej x mieści się w przedziale 20 – 40:

```

If x >= 20 AND x <= 40 Then
  Display "Wartość mieści się w przedziale."
End If

```

Złożone wyrażenie boolowskie, które sprawdzamy w tym przykładzie, zwróci prawdę tylko wtedy, gdy wartość przypisana do zmiennej x jest zarówno większa lub równa 20, jak i mniejsza lub równa 40. Aby wyrażenie złożone zwróciło prawdę, wartość zmiennej x musi więc mieścić się w przedziale 20 – 40.

Jeśli chcemy sprawdzić, czy wartość jest spoza określonego przedziału liczbowego, możemy użyć operatora OR. Poniższa instrukcja sprawdza, czy wartość przypisana do zmiennej x jest spoza przedziału 20 – 40:

```

If x < 20 OR x > 40 Then
  Display "Wartość nie mieści się w przedziale."
End If

```

Ważne tutaj jest, aby tworząc wyrażenie złożone, nie pomylić operatorów OR i AND. Przykładowo wyrażenie złożone w tej instrukcji nigdy nie zwróci prawdy:

```
// Tutaj jest błąd!
If x < 20 AND x > 40 Then
    Display "Wartość nie mieści się w przedziale."
End If
```

Jest oczywiste, że zmienna x nie może być jednocześnie mniejsza niż 20 i większa niż 40.



## Punkt kontrolny

- 4.21. Co to jest złożone wyrażenie boolowskie?
- 4.22. W poniższej tablicy prawdy zamieścilem różne kombinacje wartości prawda i fałsz dla różnych operatorów logicznych. Zaznacz w tabeli P, jeżeli wynikiem danej kombinacji jest prawda, lub F, jeżeli wynikiem jest fałsz.

Wyrażenie logiczne		Wynik
prawda AND fałsz	P	F
prawda AND prawda	P	F
fałsz AND prawda	P	F
fałsz AND fałsz	P	F
prawda OR fałsz	P	F
prawda OR prawda	P	F
fałsz OR prawda	P	F
fałsz OR fałsz	P	F
NOT prawda	P	F
NOT fałsz	P	F

- 4.23. Założymy, że  $a = 2$ ,  $b = 4$ ,  $a c = 6$ . Zaznacz P, jeżeli dane wyrażenie zwraca prawdę, lub F, jeżeli dane wyrażenie zwraca fałsz.

$a == 4 \text{ OR } b > 2$	P	F
$6 <= c \text{ AND } a > 3$	P	F
$1 != b \text{ AND } c != 3$	P	F
$a >= -1 \text{ OR } a <= b$	P	F
NOT ( $a > 2$ )	P	F

- 4.24. Wyjaśnij, jak działa technika optymalizacyjna *short-circuit evaluation* dla operatorów AND i OR.
- 4.25. Napisz instrukcję If-Then, która będzie wyświetlała komunikat *Liczba jest poprawna*, gdy zmienna speed będzie się mieściła w przedziale 0 – 200.
- 4.26. Napisz instrukcję If-Then, która będzie wyświetlała komunikat *Liczba nie jest poprawna*, gdy zmienna speed nie będzie się mieściła w przedziale 0 – 200.

**4.7**

## Zmienne boolowskie

**WYJAŚNIENIE:** W zmiennej boolowskiej można zapisać tylko jedną z dwóch wartości: prawdę albo fałsz. Zmienne boolowskie są bardzo często używane do ustawiania flag wskazujących, czy spełniony jest określony warunek.

Dotychczas korzystaliśmy tylko ze zmiennych typu Integer, Real i String. Języki programowania, poza tymi liczbowymi i tekstowymi typami danych, udostępniają jeszcze zmienne typu Boolean. W zmiennej typu Boolean można zapisać jedną z dwóch możliwych wartości: True (prawda) i False (fałsz). Oto przykład, na którym pokazalem, w jaki sposób będziemy deklarować zmienne typu Boolean w książce:

```
Declare Boolean isHungry
```

W większości języków programowania do zmiennych typu Boolean można przypisać określone słowa kluczowe, np. True albo False. Oto przykład przypisania wartości do zmiennej typu Boolean:

```
Set isHungry = True
Set isHungry = False
```

Zmienne boolowskie są często wykorzystywane jako flagi. **Flaga** to zmienna, która informuje, czy spełniony jest określony warunek. Jeżeli flaga jest ustawiona na wartość False, oznacza to, że warunek nie jest spełniony. Kiedy flaga jest ustawiona na wartość True, oznacza to, że warunek jest spełniony.

Spójrzmy na przykład. Założymy, że każdy sprzedawca musi zrealizować miesięczny plan w wysokości 50000 zł. Wartość sprzedanych przez niego do tej pory produktów jest zapisana w zmiennej sales. Na poniższym pseudokodzie przedstawiłem, jak możemy sprawdzić, czy sprzedawca zrealizował plan:

```
If sales >= 50000 Then
    Set salesQuotaMet = True
Else
    Set salesQuotaMet = False
End If
```

Po wykonaniu tego kodu możemy wykorzystać zmienną salesQuotaMet jako flagę wskazującą, czy sprzedawca zrealizował plan. W dalszej części programu możemy użyć tej flagi w następujący sposób:

```
If salesQuotaMet Then
    Display "Zrealizowałeś plan!"
End If
```

Kod ten wyświetli komunikat *Zrealizowałeś plan!* tylko wtedy, gdy zmienna typu Boolean będzie ustawiona na True. Zwróć uwagę, że w tym przypadku nie użyliśmy operatora == do porównania ze sobą wartości salesQuotaMet i True. Kod ten jest równoważny temu kodowi:

```
If salesQuotaMet == True Then
    Display "Nie zrealizowałeś jeszcze planu!"
End If
```



## Punkt kontrolny

- 4.27. Jaka wartość jest zapisywana w zmiennej typu Boolean?
- 4.28. Do czego służy flaga?

4.8

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ wielu zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: [ftp://ftp.helion.pl/przykłady/pkpro5.zip](ftp://ftp.helion.pl/przykladы/pkpro5.zip).

### Java

Struktury warunkowe  
i logika boolowska w języku Java

#### Operatory relacji w języku Java

W tabeli 4.8 przedstawiam operatory relacji używane w Javie. Są one takie same jak te, których używaliśmy wcześniej w tym rozdziale, w ramach różnych pseudokodów.

**Tabela 4.8.** Operatory relacji

Operator	Znaczenie
>	Większe niż
<	Mniejsze niż
>=	Większe niż lub równe
<=	Mniejsze niż lub równe
==	Równe
!=	Nierówne

#### Instrukcje if w języku Java

Oto ogólny format instrukcji if w Javie:

```
if (WyrażenieLogiczne)
{
    instrukcja;
    instrukcja;
    itd;
}
```

Wykonanie instrukcji `if` powoduje sprawdzenie wartości wyrażenia logicznego. Jeżeli to wyrażenie jest prawdziwe, to wykonywane są instrukcje znajdujące się w nawiasach klamrowych. Jeżeli jednak wyrażenie logiczne okaże się fałszywe, to instrukcje w nawiasach klamrowych są pomijane.

Jeśli napiszesz instrukcję `if`, która będzie zawierała tylko jedną instrukcję wykonywaną warunkowo, to instrukcji tej nie musisz umieszczać w nawiasach klamrowych. Taką instrukcję `if` można zapisać w następującym ogólnym formacie:

```
if (WyrażenieLogiczne)
    instrukcja;
```

Podczas wykonywania instrukcji `if` zapisanej w tym formacie również sprawdzane jest wyrażenie logiczne. Jeżeli jest ono prawdziwe, to wykonywana jest pojedyncza instrukcja znajdująca się w następnym wierszu. Jeżeli jednak wyrażenie logiczne będzie nieprawdziwe, to ta instrukcja zostanie pominięta.

### Instrukcja if-else w języku Java

W Javie instrukcji `if-else` używa się do tworzenia struktury decyzyjnej o dwóch wariantach wykonania kodu. Oto ogólny format instrukcji `if-else`:

```
if (WyrażenieLogiczne)
{
    instrukcja;
    instrukcja;
    itd;
}
else
{
    instrukcja;
    instrukcja;
    itd;
}
```

Wykonanie instrukcji `if-else` powoduje sprawdzenie wartości wyrażenia logicznego. Jeżeli to wyrażenie jest prawdziwe, to wykonywane są instrukcje znajdujące się w nawiasach klamrowych zaraz za instrukcją `if`. Jeżeli jednak wyrażenie logiczne okaże się fałszywe, to wykonywane są instrukcje znajdujące się w nawiasach klamrowych zaraz za instrukcją `else`. Na zakończenie program przechodzi do instrukcji znajdujących się bezpośrednio za instrukcją `if-else`.

Jeśli napiszesz instrukcję `if-else`, która będzie zawierała tylko jedną instrukcję w klauzuli `if` i jedną w klauzuli `else`, to instrukcji tych nie musisz umieszczać w nawiasach klamrowych. Taką instrukcję `if-else` można zapisać w następującym ogólnym formacie:

```
if (WyrażenieLogiczne)
    instrukcja;
else
    instrukcja;
```

### Porównywanie ciągów znaków w języku Java

W Javie ciągów znaków nie porównuje się za pomocą operatorów relacji. Chcąc sprawdzić, czy ciąg znaków *string1* jest identyczny z ciągiem znaków *string2*, należy zastosować poniższą instrukcję:

```
string1.equals(string2)
```

Wyrażenie to zwraca wartość `true`, jeżeli ciąg znaków *string1* jest identyczny z ciągiem znaków *string2*. Jeżeli ciągi będą się od siebie różniły, to wyrażenie zwróci wartość `false`. Oto przykład jego działania:

```
String password = "Prospero";
if (password.equals("prospero"))
    System.out.println("Hasło jest poprawne");
else
    System.out.println("Wprowadzone hasło jest niepoprawne.");
```

Możesz też sprawdzić, czy ciąg znaków *string1* jest równy ciągowi znaków *string2* albo od niego większy lub mniejszy. W tym celu należy użyć poniższego wyrażenia:

```
string1.compareTo(string2)
```

Wyrażenie to zwraca liczbę całkowitą, którą można wykorzystać zgodnie z następującym schematem:

- Jeżeli wynik wyrażenia jest ujemny, znaczy to, że ciąg znaków *string1* jest mniejszy niż *string2*.
- Jeżeli wynik wyrażenia jest zerem, znaczy to, że ciąg znaków *string1* jest taki sam jak *string2*.
- Jeżeli wynik wyrażenia jest dodatni, znaczy to, że ciąg znaków *string1* jest większy niż *string2*.

Załóżmy, że mamy w programie zmienne typu `String` o nazwach `name1` i `name2`. Poniżej polecenie `if` pozwoli nam porównać ze sobą te dwa ciągi znaków:

```
if (name1.compareTo(name2) == 0)
    System.out.println("Imiona są takie same.");
```

Dodatkowo poniższa instrukcja porównuje ciąg znaków ze zmiennej `name1` i ciąg znaków literału "Jan":

```
if (name1.compareTo("Jan") == 0)
    System.out.println("Imiona są takie same.");
```

### Polecenie `switch` (struktura przypadków) w języku Java

W Javie struktury przypadków zapisywane są za pomocą instrukcji `switch`. Oto ogólny format tej instrukcji:

```
switch (Wyrażenie) ←———— To jest zmienna lub wyrażenie
{
    case wartość_1:
        instrukcja
        instrukcja
        itd.
    break;
```

} Te instrukcje są wykonywane,  
jeżeli wyrażenie ma wartość *wartość\_1*

```

case wartość_2:
    instrukcja } Te instrukcje są wykonywane,
    instrukcja } jeżeli wyrażenie ma wartość wartość_2
    itd.
    break;
Wstaw tutaj tyle sekcji case, ile potrzeba.
case wartość_N:
    instrukcja } Te instrukcje są wykonywane,
    instrukcja } jeżeli wyrażenie ma wartość wartość_N
    itd.
    break;
default:
    instrukcja } Te instrukcje wykonywane są w przypadku, gdy wyrażenie
    instrukcja } nie ma żadnej z wartości wymienionych w sekcjach case
    itd.
}   ← To jest koniec instrukcji switch

```

Na przykład te instrukcje wykonują tę samą operację co w schemacie blokowym przedstawionym na rysunku 4.18:

```

switch (month)
{
    case 1:
        System.out.println("Styczeń");
        break;
    case 2:
        System.out.println("Luty");
        break;
    case 3:
        System.out.println("Marzec");
        break;
    default:
        System.out.println("Błąd: nieprawidłowy miesiąc");
}

```

W tym przykładzie wyrażeniem testowym jest zmienna month. Jeśli wartość w zmiennej month wynosi 1, program przejdzie do sekcji case 1: i wykona znajdującą się w niej instrukcję System.out.println("Styczeń"). Natomiast jeśli zmienna month będzie miała wartość 2, to program przejdzie do sekcji case 2: i wykona znajdującą się w niej instrukcję System.out.println("Luty"). W przypadku, gdy wartość w zmiennej month wyniesie 3, program przejdzie do sekcji case 3: i wykona znajdującą się w niej instrukcję System.out.println("Marzec"). Jeśli zmienna month będzie miała wartość różną od 1, 2 lub 3, to program przejdzie do sekcji default: i wykona znajdującą się w niej instrukcję System.out.println("Błąd: nieprawidłowy miesiąc").

Oto kilka ważnych właściwości instrukcji switch dostępnej w Javie:

- Wyrażenie musi być wartością lub wyrażeniem o następującym typie danych: char, byte, short, int lub String.
- Wartość po instrukcji case musi być literałem lub stałą nazwaną o następującym typie danych: char, byte, short, int lub String.
- Instrukcja break, która pojawia się na końcu sekcji case, jest opcjonalna, ale w większości sytuacji będzie nam potrzebna. Jeśli program wykona sekcję case, która nie będzie zakończona instrukcją break, program zacznie wykonywać kod znajdujący się w następnej sekcji.

- Sekcja `default` jest opcjonalna, ale w większości sytuacji powinna się ona pojawiać. Sekcja `default` jest wykonywana wtedy, gdy *wyrażenie* nie jest równe żadnej z wartościami podanymi w poszczególnych sekcjach `case`.
- Jeśli sekcja `default` pojawi się na końcu instrukcji `switch`, nie potrzeba już instrukcji `break`.

### **Operatory logiczne w języku Java**

Operatory logiczne Javy wyglądają nieco inaczej niż operatory logiczne wykorzystywane w pseudokodach tego rozdziału, ale działają w ten sam sposób. W tabeli 4.9 przedstawiam operatory logiczne Javy.

**Tabela 4.9.** Operatory logiczne w języku Java

Operator	Znaczenie
<code>&amp;&amp;</code>	Logiczne AND
<code>  </code>	Logiczne OR
<code>!</code>	Logiczne NOT

Na przykład ta instrukcja `if` sprawdza wartość w zmiennej `x`, aby określić, czy znajduje się ona w przedziale od 20 do 40:

```
if (x >= 20 && x <= 40)
    System.out.println(x + " mieści się w przedziale.");
```

Wyrażenie logiczne zawarte w instrukcji `if` będzie prawdziwe tylko wtedy, gdy wartość `x` jest większa lub równa 20 i mniejsza lub równa 40. Wartość `x` musi zawierać się w przedziale od 20 do 40, aby wyrażenie było prawdziwe. Ta instrukcja sprawdza, czy wartość zmiennej `x` znajduje się poza przedziałem od 20 do 40:

```
if (x < 20 || x > 40)
    System.out.println(x + " nie mieści się w przedziale.");
```

Oto instrukcja `if` korzystająca z operatora `!`:

```
if (!(temperature > 100))
    System.out.println("Temperatura jest niższa od maksymalnej.");
```

W pierwszej kolejności testowane jest wyrażenie (`temperature > 100`), a wynikiem testu jest wartość `true` lub `false`. Następnie wobec uzyskanej wartości stosowany jest operator `!`. Jeśli wyrażenie (`temperature > 100`) jest prawdziwe, to operator `!` zwraca wartość `false`, natomiast kiedy wyrażenie (`temperature > 100`) jest fałszywe, to operator `!` zwraca wartość `true`. Poprzedni kod programu odpowiada na pytanie: „Czy temperatura nie jest większa niż 100?”.

### **Zmienna typu Boolean w języku Java**

W Javie typ danych `boolean` służy do tworzenia zmiennych logicznych. Zmienna `boolean` może zawierać jedną z dwóch możliwych wartości: `true` lub `false`. Oto przykład deklaracji zmiennej typu `boolean`:

```
boolean highScore;
```

Taka zmienna może być użyta do zasygnalizowania, że został osiągnięty wysoki wynik. Oto przykład:

```
if (average > 95)
    highScore = true;
```

Na późniejszym etapie ten sam program może wykorzystać kod podobny do poniższego, aby za pomocą zmiennej `highScore` sprawdzić, czy został osiągnięty wysoki wynik:

```
if (highScore)
    System.out.println("To wysoki wynik!");
```

## Python

Struktury decyzyjne  
i wyrażenia logiczne w języku Python

### Operatory relacji w języku Python

Operatory relacji w Pythonie, przedstawione w tabeli 4.10, są takie same jak te, których używaliśmy wcześniej w tym rozdziale, w programach zapisanych pseudokodem.

**Tabela 4.10.** Operatory relacji

Operator	Znaczenie
>	Większe niż
<	Mniejsze niż
>=	Większe niż lub równe
<=	Mniejsze niż lub równe
==	Równe
!=	Nierówne

### Instrukcja if w języku Python

Oto ogólny format instrukcji `if` w Pythonie:

```
if WyrażenieLogiczne:
    instrukcja
    instrukcja
    itd.
```

Dla uproszczenia pierwszy wiersz nazywać będziemy *klauzulą if*. Klauzula `if` zaczyna się od słowa `if`, po którym następuje wyrażenie logiczne. Za wyrażeniem logicznym pojawia się dwukropka, a od nowego wiersza zaczyna się blok instrukcji. Wszystkie instrukcje zawarte w bloku muszą konsekwentnie mieć jednakowe wcięcie. Jest ono wymagane, ponieważ interpreter Pythona za jego pomocą określa, gdzie zaczyna się, a gdzie kończy blok.

Podczas wykonywania instrukcji `if` sprawdzane jest wyrażenie logiczne. Jeśli to wyrażenie przyjmie wartość `true`, to zostaną wykonane instrukcje znajdujące się w bloku za klauzulą `if`. Jeśli wyrażenie logiczne przyjmie wartość `false`, to instrukcje z tego bloku zostaną pominięte.

### Instrukcja if-else w języku Python

Instrukcję `if-else` wykorzystuje się w celu utworzenia struktury decyzyjnej o dwóch wariantach wykonania kodu. Oto format instrukcji `if-else`:

```
if WyrażenieLogiczne:
    instrukcja
    instrukcja
    itd.

else:
    instrukcja
    instrukcja
    itd.
```

Wykonanie tej instrukcji powoduje sprawdzenie wartości wyrażenia logicznego. Jeśli wyrażenie jest prawdziwe, to wykonywany jest blok instrukcji z wcięciem następujących po klauzuli `if`, a następnie program przejdzie do instrukcji po instrukcji `if-else`. Jeśli wyrażenie logiczne ma wartość `false`, to wykonany zostanie blok instrukcji z wcięciem następujący po klauzuli `else`, a następnie program przejdzie do instrukcji po instrukcji `if-else`.

### Porównywanie ciągów znaków w języku Python

Operatory relacji mogą być używane do porównywania ciągów znaków:

```
name1 = 'Marysia'
name2 = 'Marek'
if name1 == name2:
    print('Imiona są takie same.')
else:
    print('Imiona NIE są takie same.')
```

Operator `==` porównuje zmienne `name1` i `name2`, aby ustalić, czy są równe. Ciągi '`Marysia`' i '`Marek`' nie są równe, dlatego klauzula `else` wyświetli komunikat '`Imiona NIE są takie same.`'.

Spójrzmy na inny przykład. Założymy, że zmiennej `month` przypisany został ciąg znaków. Poniższy kod wykorzystuje operator `!=` do sprawdzenia, czy wartość w zmiennej `month` nie jest równa ciągowi znaków '`Październik`':

```
if month != 'Październik':
    print('To nie jest czas na Oktoberfest!')
```



**UWAGA:** W języku Python nie występuje struktura `case`, więc nie możemy jej tutaj opisać.

### Operatory logiczne w języku Python

W tabeli 4.11 przedstawiam operatory logiczne dostępne w Pythonie, które działają podobnie do omówionych wcześniej w tym rozdziale.

**Tabela 4.11.** Operatory logiczne w języku Python

Operator	Znaczenie
and	Logiczne AND
or	Logiczne OR
not	Logiczne NOT

Na przykład ta instrukcja `if` sprawdza, czy wartość w zmiennej `x` znajduje się w przedziale od 20 do 40:

```
if x >= 20 and x <= 40:
    print(x, mieści się w przedziale.');
```

Wyrażenie logiczne zawarte w instrukcji `if` będzie prawdziwe tylko wtedy, gdy wartość `x` jest większa lub równa 20 i mniejsza lub równa 40. Wartość `x` musi zawierać się w przedziale od 20 do 40, aby wyrażenie było prawdziwe. Ta instrukcja określa, czy wartość zmiennej `x` znajduje się poza przedziałem od 20 do 40:

```
if x < 20 or x > 40
    print(x, nie mieści się w przedziale.');
```

Oto instrukcja `if` korzystająca z operatora `not`:

```
If not (temperature > 100)
    print('Temperatura jest niższa od maksymalnej.');
```

W pierwszej kolejności sprawdzane jest wyrażenie `(temperature > 100)`, którego wynikiem jest wartość `true` lub `false`. Następnie wobec uzyskanej wartości zostanie użyty operator `not`. Jeśli wyrażenie `(temperature > 100)` jest prawdziwe, to operator `not` zwraca wartość `false`, natomiast kiedy wyrażenie `(temperature > 100)` jest fałszywe, to operator `not` zwraca wartość `true`. Powyższy kod programu odpowiada na pytanie: „Czy temperatura nie jest większa niż 100?”.

### Zmienne logiczne w języku Python

W Pythonie typ danych `Bool` służy do tworzenia zmiennych, które mogą przyjmować jedną z dwóch wartości: `True` lub `False`. Oto przykład deklaracji zmiennej typu `bool`:

```
hungry = True
sleepy = False
```

Zmienne logiczne najczęściej używane są jako znaczniki informujące o zaistnieniu jakiegoś warunku w programie. Gdy zmienna znacznika ma wartość `False`, znaczy to, że warunek nie jest spełniony. Natomiast w przeciwnym wypadku, gdy zmienna znacznika ma wartość `True`, znaczy to, że warunek jest spełniony.

Załóżmy, że sprzedawca ma osiągnąć sprzedaż na poziomie 50 000 PLN. Jeżeli zmienna `sales` będzie przechowywała kwotę sprzedaży uzyskaną przez sprzedawcę, to poniższy kod powie nam, czy limit sprzedaży został osiągnięty:

```
if sales >= 50000.0:
    sales_quota_met = True
else:
    sales_quota_met = False
```

## C++

### Struktury decyzyjne i operatory logiczne w języku C++

#### Operatory relacji w języku C++

Operatory relacji w C++, przedstawione w tabeli 4.12, są takie same jak te, których używaliśmy wcześniej w tym rozdziale, w programach zapisanych pseudokodem.

**Tabela 4.12.** Operatory relacji

Operator	Znaczenie
>	Większe niż
<	Mniejsze niż
>=	Większe niż lub równe
<=	Mniejsze niż lub równe
==	Równe
!=	Nierówne

#### Instrukcja if w języku C++

Oto ogólny format instrukcji `if` w C++:

```
if (WyrażenieLogiczne)
{
    Instrukcja;
    Instrukcja;
    itd.;
}
```

W ramach wykonywania instrukcji `if` obliczane jest wyrażenie logiczne. Jeżeli wyrażenie jest prawdziwe, to wykonywane są instrukcje zawarte w nawiasach klamrowych. Jeżeli wyrażenie logiczne przyjmie wartość `false`, to instrukcje w nawiasach klamrowych zostaną pominięte.

Jeżeli zapisujesz instrukcję `if`, która ma tylko jedną warunkowo wykonywaną instrukcję, to nie musisz umieszczać jej w nawiasach klamrowych. Taką instrukcję `if` można zapisać w następującym ogólnym formacie:

```
if (WyrażenieLogiczne)
    instrukcja;
```

Podczas wykonywania instrukcji `if` zapisanej w takim formacie sprawdzana jest wartość wyrażenia logicznego. Jeśli wyrażenie jest prawdziwe, to zostanie wykonana jedna instrukcja, znajdująca się w następnym wierszu. Jeśli jednak wyrażenie logiczne przyjmie wartość `false`, to instrukcja ta zostanie pominięta.

### Instrukcja if-else w języku C++

Instrukcję `if-else` wykorzystuje się w celu utworzenia struktury decyzyjnej o dwóch wariantach wykonania kodu. Oto format instrukcji `if-else`:

```
if (WyrażenieLogiczne)
{
    instrukcja;
    instrukcja;
    itd.;

}
else
{
    instrukcja;
    instrukcja;
    itd.;

}
```

Wykonanie tej instrukcji powoduje sprawdzenie wartości wyrażenia logicznego. Jeśli wyrażenie jest prawdziwe, to wykonywany jest blok instrukcji z wcięciem następujących po klauzuli `if`, a następnie program przejdzie do instrukcji po instrukcji `if-else`. Jeśli wyrażenie logiczne ma wartość `false`, to wykonany zostanie blok instrukcji z wcięciem następujący po klauzuli `else`, a następnie program przejdzie do instrukcji po instrukcji `if-else`.

Jeśli którykolwiek z bloków instrukcji wykonanych warunkowo zawiera tylko jedną instrukcję, to nawiasy klamrowe nie są wymagane. Na przykład poniższy schemat instrukcji pokazuje tylko jedną instrukcję następującą po klauzuli `if` i tylko jedną instrukcję następującą po klauzuli `else`:

```
if (WyrażenieLogiczne)
    instrukcja;
else
    instrukcja;
```

### Instrukcja switch (struktura przypadków) w języku C++

W C++ struktury przypadków są zapisywane jako instrukcje `switch`. Oto ogólny format instrukcji `switch`:

```
switch (Wyrażenie) ←———— To jest zmienna całkowita lub wyrażenie
{
    case wartość_1:
        instrukcja
        instrukcja
        itd.
        break;
    }

} Te instrukcje są wykonywane,
jeśli wyrażenie jest równe wartość_1
```

```

case wartość_2:
    instrukcja
    instrukcja
    itd.
    break;
}

```

} Te instrukcje są wykonywane,  
jeśli wyrażenie jest równe wartość\_2

Wstaw tutaj tyle sekcji case, ile potrzeba:

```

case wartość_N:
    instrukcja
    instrukcja
    itd.
    break;
default:
    instrukcja
    instrukcja
    itd.
}

```

} Te instrukcje są wykonywane, jeśli wyrażenie nie jest równe  
żadnej z wartości wymienionych po instrukcjach case

} ← To jest koniec instrukcji switch

Podczas wykonywania instrukcji switch następuje porównanie wartości *wyrażenia* z wartościami, które są zawarte w każdej instrukcji case (od góry do dołu). Po znalezieniu odpowiedniej wartości przypadku case, która będzie równa wartości *wyrażenia*, program przeskoczy do instrukcji case. Instrukcje następujące po instrukcji case są wykonywane, dopóki nie zostanie napotkana instrukcja break. W tym momencie program wychodzi z pętli switch. Jeśli wartość *wyrażenia* nie pasuje do żadnej z wartości w przypadkach case, program przeskakuje do instrukcji default i wykonuje instrukcję znajdującą się bezpośrednio po niej.

Na przykład ten kod wykonuje tę samą operację co schemat blokowy przedstawiony na rysunku 4.18:

```

switch (miesiac)
{
    case 1:
        cout << "Styczeń" << endl;
        break;
    case 2:
        cout << "Luty" << endl;
        break;
    case 3:
        cout << "Marzec" << endl;
        break;
    default:
        cout << "Błąd: nieprawidłowy miesiąc" << endl;
}

```

Oto kilka ważnych punktów do zapamiętania odnoszących się do instrukcji switch w języku C++:

- Wyrażenie musi przyjmować wartość lub wyrażenie jednego z całkowitych typów danych (w tym także typu char).
- Wartość następująca po instrukcji case musi zawierać literal lub stałą nazwaną jednego z całkowitych typów danych (w tym także typu char).

- Instrukcja `break`, która pojawia się na końcu sekcji `case`, jest opcjonalna, ale w większości sytuacji będzie nam potrzebna. Jeśli program wykona sekcję `case`, która nie kończy się instrukcją `break`, będzie kontynuował wykonywanie kodu do następnej sekcji `case`.
- Sekcja `default` jest opcjonalna, ale w większości sytuacji powinno się ją mieć. Jest ona wykonywana, gdy *wyrażenie* nie pasuje do żadnej wartości znajdującej się w sekcji `case`.
- Jeśli w sekcji `default` na końcu pojawia się instrukcja `switch`, instrukcja `break` nie będzie już potrzebna.

### Operatory logiczne w języku C++

Operatory logiczne w C++ wyglądają nieco inaczej niż te, których używaliśmy w pseudokodach niniejszego rozdziału, ale ich działanie jest takie samo. Tabela 4.13 pokazuje operatory logiczne w C++.

**Tabela 4.13.** Operatory logiczne w języku C++

Operator	Znaczenie
<code>&amp;&amp;</code>	Logiczne AND
<code>  </code>	Logiczne OR
<code>!</code>	Logiczne NOT

Na przykład ta instrukcja `if` sprawdza wartość `x`, aby określić, czy znajduje się ona w przedziale od 20 do 40:

```
if (x >= 20 && x <= 40)
    cout << x << " mieści się w przedziale." << endl;
```

Wyrażenie logiczne Boolean zawarte w instrukcji `if` będzie prawdziwe tylko wtedy, gdy wartość `x` jest większa lub równa 20, a mniejsza lub równa 40. Wartość `x` musi zawierać się w przedziale od 20 do 40, aby wyrażenie było prawdziwe. Ta instrukcja określa, czy wartość `x` znajduje się poza przedziałem od 20 do 40:

```
if (x < 20 || x > 40)
    cout << x << " nie mieści się w przedziale." << endl;
```

Oto instrukcja `if` korzystająca z operatora `!`:

```
if (!(temperatura > 100))
    cout << "Temperatura jest niższa od maksymalnej." << endl;
```

W pierwszej kolejności testowane jest wyrażenie `(temperatura > 100)`, a wynikiem testu jest wartość `true` lub `false`. Następnie operator `!` zostanie zastosowany do tej wartości. Jeśli wyrażenie `(temperatura > 100)` jest prawdziwe, to operator `!` zwraca wartość `false`, natomiast kiedy wyrażenie `(temperatura > 100)` jest fałszywe, to operator `!` zwraca wartość `true`. Poprzedni kod programu odpowiada na pytanie: „Czy temperatura nie jest większa niż 100?”.

### Zmienne typu bool w języku C++

W C++ typ danych `bool` służy do tworzenia zmiennych typu logicznego. Zmienna typu `bool` może zawierać jedną z dwóch możliwych wartości: `true` lub `false`. Oto przykład deklaracji zmiennej typu `bool`:

```
boolean highScore;
```

Zmienne typu `bool` są powszechnie używane jako flagi, które sygnalizują, gdy w programie istnieje jakiś warunek. Jeśli zmienna flagi ma wartość `false`, oznacza to, że warunek jeszcze nie istnieje, natomiast jeśli zmienna ta jest ustawiona na wartość `true`, oznacza to, że warunek istnieje.

Załóżmy na przykład, że testowy program oceny ma zmienną `bool` o nazwie `highScore`. Zmienna może być użyta do zasygnalizowania, że wysoki wynik został osiągnięty:

```
if (average > 95)
    highScore = true;
```

## Pytania kontrolne

### Test jednokrotnego wyboru

- Struktura \_\_\_\_\_ wykonuje określone operacje tylko wtedy, gdy spełniony zostanie określony warunek.
  - sekwencyjna
  - okolicznościowa
  - warunkowa
  - boolowska
- \_\_\_\_\_ umożliwia wybranie tylko jednej alternatywnej ścieżki.
  - struktura sekwencyjna
  - struktura warunkowa pojedynczego wyboru
  - struktura jednej ścieżki alternatywnej
  - struktura warunkowa pojedynczego wykonania
- Instrukcja If-Then w pseudokodzie jest przykładem \_\_\_\_\_.
  - struktury sekwencyjnej
  - struktury warunkowej
  - struktury ścieżkowej
  - struktury klasowej
- Wyrażenie \_\_\_\_\_ zwraca albo prawdę, albo fałsz.
  - binarne
  - warunkowe
  - bezwarunkowe
  - boolowskie
- Symboly `>`, `<` i `==` są operatorami \_\_\_\_\_.
  - relacji
  - logicznymi

- c) warunkowymi  
d) potrójnymi
6. W strukturze \_\_\_\_\_ sprawdzany jest określony warunek i jeżeli jest on spełniony, program może podążyć jedną ścieżkę, a w przeciwnym wypadku może podążyć inną ścieżką.  
a) typu If-Then  
b) warunkowej pojedynczego wyboru  
c) warunkowej podwójnego wyboru  
d) sekwencyjnej
7. Za pomocą instrukcji \_\_\_\_\_ możemy stworzyć w pseudokodzie strukturę warunkową pojedynczego wyboru.  
a) Test-Jump  
b) If-Then  
c) If-Then-Else  
d) If-Call
8. Za pomocą instrukcji \_\_\_\_\_ możemy stworzyć w pseudokodzie strukturę warunkową podwójnego wyboru.  
a) Test-Jump  
b) If-Then  
c) If-Then-Else  
d) If-Call
9. Struktura \_\_\_\_\_ sprawdza wartość przypisaną do zmiennej (lub wartość zwracaną przez wyrażenie) i w zależności od tej wartości wykonuje w programie określone instrukcje.  
a) warunkowa zmiennej testowej  
b) warunkowa pojedynczego wyboru  
c) warunkowa podwójnego wyboru  
d) warunkowa wielokrotnego wyboru
10. Jeśli w instrukcji Select Case wartość wyrażenia nie zostanie dopasowana do żadnej ze wskazanych wartości, program przeskoczy do polecenia \_\_\_\_\_.  
a) Else  
b) Default  
c) Case  
d) Otherwise
11. AND, OR i NOT to operatory \_\_\_\_\_.  
a) relacji  
b) logiczne  
c) warunkowe  
d) potrójne
12. Wyrażenie złożone zbudowane za pomocą operatora \_\_\_\_\_ zwraca prawdę tylko wtedy, gdy oba podwyrażenia zwracają prawdę.  
a) AND  
b) OR  
c) NOT  
d) EITHER

13. Wyrażenie złożone zbudowane za pomocą operatora \_\_\_\_\_ zwraca prawdę wtedy, gdy którykolwiek z podwyrażeń zwraca prawdę.
  - a) AND
  - b) OR
  - c) NOT
  - d) EITHER
14. Operator \_\_\_\_\_ przyjmuje wartość typu Boolean i zwraca jej zanegowaną wartość logiczną.
  - a) AND
  - b) OR
  - c) NOT
  - d) EITHER
15. \_\_\_\_\_ to zmienna typu Boolean, która informuje, czy spełniony jest określony warunek.
  - a) flaga
  - b) sygnał
  - c) wartownik
  - d) syrena

### Prawda czy fałsz?

1. Każdy program można napisać, korzystając tylko ze struktury sekwencyjnej.
2. W programie można wykorzystać tylko jeden typ struktury. Struktur nie można ze sobą łączyć.
3. Struktura warunkowa pojedynczego wyboru sprawdza określony warunek i w zależności od tego, czy warunek jest spełniony, może wykonać pewne instrukcje, a w przeciwnym przypadku może wykonać inne instrukcje.
4. Jedną strukturę warunkową można zagnieździć wewnątrz innej struktury warunkowej.
5. Złożone wyrażenie boolowskie składające się z operatora AND zwraca prawdę tylko wtedy, gdy oba podwyrażenia zwracają prawdę.

### Krótką odpowiedź

1. Wyjaśnij, co oznacza określenie, że instrukcje zostaną wykonane warunkowo.
2. Musisz sprawdzić określony warunek i w zależności od tego, czy jest on spełniony, czy nie, wykonać jeden lub drugi szereg instrukcji. Z której struktury skorzystasz?
3. Z której struktury skorzystasz, aby sprawdzić wartość przypisaną do zmiennej i w zależności od tej wartości wykonać instrukcję lub grupę instrukcji?
4. Opisz krótko, jak działa operator AND.
5. Opisz krótko, jak działa operator OR.
6. Jakiego operatora logicznego najlepiej użyć, aby sprawdzić, czy liczba mieści się w określonym przedziale?
7. Co to jest flaga i do czego służy?

## Warsztat projektanta algorytmów

- Zaprojektuj instrukcję If-Then (lub narysuj jej schemat blokowy za pomocą struktury warunkowej pojedynczego wyboru), która w przypadku, gdy zmienna *x* jest równa 100, przypisze do zmiennej *y* wartość 20, a do zmiennej *z* wartość 40.
- Zaprojektuj instrukcję If-Then (lub narysuj jej schemat blokowy za pomocą struktury warunkowej pojedynczego wyboru), która w przypadku, gdy zmienna *a* jest mniejsza niż 10, przypisze do zmiennej *b* wartość 0, a do zmiennej *b* wartość 1.
- Zaprojektuj instrukcję If-Then-Else (lub narysuj jej schemat blokowy za pomocą struktury warunkowej podwójnego wyboru), która w przypadku, gdy zmienna *a* jest mniejsza niż 10, przypisze do zmiennej *a* wartość 0, a w przeciwnym wypadku przypisze do zmiennej *b* wartość 99.
- Poniższy pseudokod zawiera kilka zagnieżdzonych instrukcji If-Then-Else. Niestety kod jest pozbawiony wcięć. Popraw kod zgodnie z konwencją zapisu instrukcji If-Then-Else, wyrównując odpowiednio linie i wstawiając wcięcia.

```
If score < 60 Then
    Display "Twoja ocena to 1."
Else
    If score < 70 Then
        Display "Twoja ocena to 2."
    Else
        If score < 80 Then
            Display "Twoja ocena to 3."
        Else
            If score < 90 Then
                Display "Twoja ocena to 4."
            Else
                Display "Twoja ocena to 5."
            End If
        End If
    End If
End If
```

- Zaprojektuj zagnieżdżone struktury warunkowe, aby wykonywały następujące zadanie: jeżeli zmienna *amount1* jest większa niż 10, a zmienna *amount2* jest mniejsza niż 100, wyświetl na ekranie większą z tych wartości.
- Zamień poniższą instrukcję If-Then-Else na instrukcję Select Case.

```
If selection == 1 Then
    Display "Wybrałeś A."
Else If selection == 2 Then
    Display "Wybrałeś 2."
Else If selection == 3 Then
    Display "Wybrałeś 3."
Else If selection == 4 Then
    Display "Wybrałeś 4."
Else
    Display "Masz chyba problem z matką, co nie?"
End If
```

- Zaprojektuj instrukcję If-Then-Else (lub narysuj jej schemat blokowy za pomocą struktury warunkowej podwójnego wyboru), która w przypadku, gdy zmienna *speed* mieści się w przedziale 24 – 56, wyświetli komunikat *Przekkość jest prawidłowa*. Jeśli wartość przypisana do zmiennej *speed* nie mieści się w przedziale, instrukcja powinna wyświetlić komunikat *Przekkość jest nieprawidłowa*.

7. Zaprojektuj instrukcję If-Then-Else (lub narysuj jej schemat blokowy za pomocą struktury warunkowej podwójnego wyboru), która sprawdza, czy wartość przypisana do zmiennej `points` mieści się w przedziale 9 – 51. Jeśli wartość jest spoza przedziału, wyświetl komunikat *Liczba punktów jest nieprawidłowa*. W przeciwnym razie wyświetl komunikat *Liczba punktów jest prawidłowa*.
8. Zaprojektuj strukturę decyzyjną, która w zależności od wartości przypisanej do zmiennej `month` będzie wykonywała następujące operacje:
  - Jeśli zmienna `month` jest równa 1, wyświetl tekst *Styczeń ma 31 dni*.
  - Jeśli zmienna `month` jest równa 2, wyświetl tekst *Luty ma 28 dni*.
  - Jeśli zmienna `month` jest równa 3, wyświetl tekst *Marzec ma 31 dni*.
  - Jeśli do zmiennej `month` jest przypisana inna wartość, wyświetl tekst *Nieprawidłowa wartość*.
9. Napisz instrukcję If-Then, która w przypadku, gdy ustawiona jest flaga `minimum`, przypisze do zmiennej `hours` wartość 10.

## Ćwiczenia z wykrywania błędów

1. Poniższy pseudokod nie zadziała prawidłowo w językach takich jak Java, Python, C i C++. Wskaź błąd. W jaki sposób poprawiłybyś błąd, gdybyś chciał przetłumaczyć ten program na jeden z wymienionych języków programowania?

```
Module checkEquality(Integer num1, Integer num2)
  If num1 = num2 Then
    Display "Wartości są równe."
  Else
    Display "Wartości NIE są równe."
  End If
End Module
```

2. Poniższy moduł miał za zadanie zapisywać w parametrze `temp` wartość 32.0, gdy wartość tego parametru jest różna od 32.0. Jednak program nie zadziała tak, jak oczekuje tego programisty. Znajdź błąd.

```
Module resetTemperature(Real Ref temp)
  If NOT temp == 32.0 Then
    Set temp = 32.0
  End If
End Module
```

3. Poniższy moduł miał za zadanie sprawdzać, czy wartość parametru `value` mieści się w określonym przedziale. Jednak program nie zadziała prawidłowo. Znajdź błąd.

```
Module checkRange(Integer value, Integer lower, Integer upper)
  If value < lower AND value > upper Then
    Display "Wartość nie mieści się w przedziale."
  Else
    Display "Wartość mieści się w przedziale."
  End If
End Module
```

# Ćwiczenia programistyczne

## 1. Liczby rzymskie

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby z przedziału 1 – 10. Program powinien następnie wyświetlić tę liczbę zapisaną cyframi rzymskimi. Jeżeli wprowadzona przez użytkownika liczba jest spoza przedziału, program powinien wyświetlić komunikat o błędzie.

## 2. Pole powierzchni prostokątów

Pole powierzchni prostokąta jest równe iloczynowi długości jego boków. Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie długości boków dwóch prostokątów. Program powinien poinformować użytkownika, który z prostokątów ma większe pole powierzchni lub czy pola powierzchni są takie same.

## 3. Masa i ciężar

Naukowcy wyrażają masę obiektu w kilogramach, a jego ciężar w newtonach. Jeżeli znasz masę obiektu, możesz obliczyć jego ciężar, stosując następujący wzór:

$$\text{ciężar} = \text{masa} \cdot 9,8$$

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie masy obiektu, a następnie obliczał jego ciężar. Jeżeli ciężar obiektu jest większy niż 1000 N, wyświetl komunikat, że obiekt jest za ciężki. Jeżeli ciężar obiektu jest mniejszy niż 10 N, wyświetl komunikat, że obiekt jest za lekki.

## 4. Magiczne daty

Data 10 czerwca 1960 jest wyjątkowa, ponieważ można ją zapisać w takiej formie, że w wyniku pomnożenia dnia i miesiąca otrzymamy rok:

10/6/60

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie dnia, miesiąca (jako liczby) i roku (w postaci dwucyfrowej). Program powinien sprawdzić, czy iloczyn dnia i miesiąca będzie równy rokowi. Jeśli tak, powinien się wyświetlić komunikat informujący, że data jest magiczna. W przeciwnym razie powinien się wyświetlić komunikat, że data nie jestmagiczna.

## 5. Mikser kolorów

Kolory czerwony, niebieski i żółty nazywane się kolorami podstawowymi, ponieważ nie da się ich uzyskać, mieszając ze sobą inne kolory. Mieszając kolory podstawowe, uzyskamy kolory pochodne w następujący sposób:

- w wyniku wymieszania kolorów czerwonego i niebieskiego otrzymamy kolor fioletowy;
- w wyniku wymieszania kolorów czerwonego i żółtego otrzymamy kolor pomarańczowy;
- w wyniku wymieszania kolorów niebieskiego i żółtego otrzymamy kolor zielony.

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie nazw dwóch kolorów podstawowych. Jeżeli użytkownik wprowadzi inny tekst niż

„czerwony”, „niebieski” lub „żółty”, program powinien wyświetlić komunikat błędu. W przeciwnym razie program powinien wyświetlić nazwę koloru powstałą w wyniku wymieszania dwóch wprowadzonych kolorów.

#### 6. Program lojalnościowy

Księgarnia Serendipity Booksellers prowadzi program lojalnościowy, który w zależności od liczby książek zakupionych przez klienta w danym miesiącu nagradza go odpowiednią liczbą punktów. Punkty przyznawane są w następujący sposób:

- jeżeli klient kupił 0 książek, otrzymuje 0 punktów;
- jeżeli klient kupił 1 książkę, otrzymuje 5 punktów;
- jeżeli klient kupił 2 książki, otrzymuje 15 punktów;
- jeżeli klient kupił 3 książki, otrzymuje 30 punktów;
- jeżeli klient kupił 4 lub więcej książek, otrzymuje 60 punktów.

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby książek zakupionych przez niego w danym miesiącu, a następnie wyświetli liczbę przyznanych punktów.

#### 7. Sprzedaż oprogramowania

Sklep z oprogramowaniem komputerowym sprzedaje produkt o wartości detalicznej 99 zł. W zależności od liczby zakupionych sztuk udziela następującego rabatu:

Liczba sztuk	Rabat
10 – 19	20%
20 – 49	30%
50 – 99	40%
100 i więcej	50%

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby sztuk, które ma zamiar zakupić. Program powinien wyświetlić wartość rabatu (jeśli rabat się należy) i całkowitą wartość zakupów z uwzględnieniem rabatu.

#### 8. Grosze na złotówkę

Stwórz grę, która będzie zliczała monety 1-, 2-, 5-, 10- i 50-groszowe i sprawdzała, czy ich suma daje jedną złotówkę. Program powinien poprosić użytkownika, aby wprowadził po kolej liczbe monet 1-groszowych, 2-groszowych, 5-groszowych, 10-groszowych i 50-groszowych. Jeżeli suma monet będzie równa jednej złotówce, program powinien pogratulować użytkownikowi wygranej. W przeciwnym razie program powinien wyświetlić komunikat informujący, czy suma była mniejsza czy większa od złotówki.

#### 9. Koszty wysyłki

Firma Fast Freight Shipping Company ma następujący cennik kosztów wysyłki:

Waga przesyłki	Cena za kg
2 kg lub mniej	5,10 zł
powyżej 2 kg, ale mniej niż 6 kg	6,20 zł
powyżej 6 kg, ale mniej niż 10 kg	7,70 zł
powyżej 10 kg	8,10 zł

Zaprojektuj program, który będzie prosił użytkownika o podanie wagi przesyłki, a następnie wyświetli odpowiedni koszt.

## 10. Wskaźnik masy ciała — wersja rozbudowana

W ćwiczeniu programistycznym 6. w rozdziale 3. poprosiłem Cię o zaprojektowanie programu obliczającego wskaźnik masy ciała (BMI). Wskaźnik masy ciała może służyć do oceny, czy dana osoba ma nadwagę czy niedowagę. Wzór na wskaźnik masy ciała wygląda następująco:

$$\text{BMI} = \text{waga} / \text{wzrost}^2$$

We wzorze tym wagę wyrażamy w kilogramach, a wzrost w metrach. Rozwiń ten program w taki sposób, aby wyświetlał, czy osoba ma właściwą wagę albo czy ma nadwagę lub niedowagę. Dla osoby dorosłej waga jest prawidłowa, gdy BMI zawiera się w przedziale 18,5 – 25. Jeśli BMI jest mniejsze niż 18,5, osoba ma niedowagę. Jeśli BMI jest większe niż 25, osoba ma nadwagę.

## 11. Kalkulator czasu

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby sekund i będzie działał następująco:

- Jedna minuta zawiera 60 sekund. Jeżeli liczba sekund, którą wprowadził użytkownik, jest większa niż 60, program powinien wyświetlić liczbę minut odpowiadającą danej liczbie sekund.
- Jedna godzina zawiera 3600 sekund. Jeżeli liczba sekund, którą wprowadził użytkownik, jest większa niż 3600, program powinien wyświetlić liczbę godzin odpowiadającą danej liczbie sekund.
- Jeden dzień to 86400 sekund. Jeżeli liczba sekund, którą wprowadził użytkownik, jest większa niż 86400, program powinien wyświetlić liczbę dni odpowiadającą danej liczbie sekund.

## 12. Wykrywanie roku przestępnnego

Zaprojektuj program, który prosi użytkownika o wpisanie roku, a następnie wyświetla komunikat wskazujący, czy dany rok jest rokiem przestępnnym, czy nie. Użyj następującej logiki, aby rozwinąć swój algorytm:

- Jeśli rok można podzielić bez reszty przez 100 i jest dodatkowo podzielny bez reszty przez 400, to jest to rok przestępny. Na przykład rok 2000 jest rokiem przestępnnym, ale 2010 nie jest.
- Jeśli rok nie jest podzielny bez reszty przez 100, ale jest podzielny bez reszty przez 4, to jest rokiem przestępnnym. Na przykład rok 2008 jest rokiem przestępnnym, ale 2009 nie jest.



5

# Struktury cykliczne

TEMATYKA

- |  |   |
|--|---|
| 5.1 Struktury cykliczne — wprowadzenie             | 5.4 Obliczanie sumy bieżącej              |
| 5.2 Pętle warunkowe: While, Do-While<br>i Do-Until | 5.5 Wartownik                             |
| 5.3 Pętle licznikowe i instrukcja For              | 5.6 Pętle zagnieździone                   |
|  | 5.7 Rzut oka na języki Java, Python i C++ |

5.1

## **Struktury cykliczne — wprowadzenie**

**WYJAŚNIENIE:** Zadaniem struktury cyklicznej jest wielokrotne wykonywanie wybranego fragmentu kodu.

Programiści często muszą napisać taki kod, który będzie wykonywał określone operacje raz za razem. Założmy przykładowo, że poproszono Cię o napisanie programu, który oblicza dla każdego sprzedawcy premię w wysokości 10% wartości sprzedaży. Nie jest to dobre rozwiązanie, ale możesz napisać fragment kodu obliczający premię dla jednego ze sprzedawców, a następnie skopiować go dla kolejnych sprzedawców. Przyjrzyjmy się następującemu przykładowi:

```
// Zmienne wartości sprzedaży i premii
Declare Real sales, commission

// Stała z wysokością premii
Constant Real COMMISSION_RATE = 0.10

// Wprowadzenie wartości sprzedaży
Display "Wprowadź wartość sprzedaży."
Input sales

// Obliczenie premii
Set commission = sales * COMMISSION_RATE

// Wyświetlenie premii
Display "Premia wynosi ", commission, "
```

**Tutaj obliczamy premię dla pierwszego sprzedawcy.**

```

// Wprowadzenie wartości sprzedaży
Display "Wprowadź wartość sprzedaży."
Input sales

// Obliczenie premii
Set commission = sales * COMMISSION_RATE

// Wyświetlenie premii
Display "Premia wynosi ", commission, " zł."

```

**Dalszy kod programu wygląda identycznie.**

Tutaj obliczamy premię dla drugiego sprzedawcy.

Jak widać, jest to jedna dłuża struktura sekwencyjna, która zawiera bardzo wiele powielonych instrukcji. Takie podejście ma bardzo dużo wad, łącznie z następującymi:

- powielone instrukcje sprawiają, że program jest dłuższy;
- pisanie długiej sekwencji poleceń zajmuje dużo czasu;
- jeżeli zajdzie potrzeba poprawienia lub zmodyfikowania kodu, zmian trzeba będzie dokonać w wielu miejscach programu.

Zamiast wielokrotnie pisać tą samą sekwencję instrukcji znacznie lepszym rozwiązaniem jest zapisanie jej raz, a następnie umieszczenie jej wewnątrz struktury, która będzie ją wykonywała określoną liczbę razy. Do tego służą właśnie **struktury cykliczne** (ang. *repetition structure*), znane szerzej jako **pętle** (ang. *loops*).

## Pętle warunkowe i pętle licznikowe

W tym rozdziale przyjrzymy się dwóm rodzajom pętli: warunkowym i licznikowym. W **pętlach warunkowych** do kontrolowania tego, ile razy wykonają się określone instrukcje, używa się warunku typu prawda – fałsz. W **pętlach licznikowych** instrukcje wykonują się po prostu określoną liczbę razy. Omówię także, w jaki sposób konstruuje się tego typu pętle w większości języków programowania.



### Punkt kontrolny

- 5.1. Do czego służą struktury cykliczne?
- 5.2. Co to jest pętla warunkowa?
- 5.3. Co to jest pętla licznikowa?

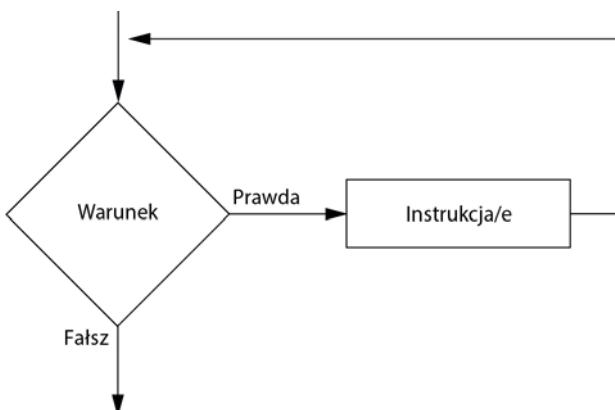
**5.2**

## Pętle warunkowe: While, Do-While i Do-Until

**WYJAŚNIENIE:** Zarówno pętla While, jak i pętla Do-While wykonuje cyklicznie instrukcję lub zbiór instrukcji dopóty, dopóki spełniony jest określony warunek. Pętla Do-Until wykonuje cyklicznie instrukcję lub zbiór instrukcji do momentu, aż spełniony zostanie określony warunek.

### Pętla While

Pętla While działa dokładnie tak, jak wskazuje jej nazwa: *While a condition is true, do some task* (dopóki warunek jest spełniony, wykonuj określone operacje). Pętla składa się z dwóch elementów: (1) warunku logicznego, który jest sprawdzany, i (2) jednej lub kilku instrukcji, które są wykonywane cyklicznie dopóty, dopóki warunek jest spełniony. Na rysunku 5.1 pokazałem, jak działa pętla While.



Rysunek 5.1. Zasada działania pętli While

Romb reprezentuje sprawdzany warunek. Zwróć uwagę, co się dzieje, gdy warunek jest spełniony: zostanie wykonana instrukcja lub grupa instrukcji, a następnie program powróci do miejsca nad symbolem rombu. Warunek zostanie więc sprawdzony po raz kolejny i jeżeli nadal będzie spełniony, cały proces się powtórzy, a jeżeli warunek nie zostanie spełniony, program wyjdzie z pętli. Łatwo można rozpoznać pętlę na schemacie blokowym, ponieważ strzałka oznaczająca przepływ programu jest w takim przypadku skierowana do poprzedniego fragmentu schematu.

### Tworzenie pętli While za pomocą pseudokodu

W pseudokodzie pętlę While będziemy zapisywać za pomocą instrukcji While. Oto ogólna postać instrukcji While:

```

While warunek
    instrukcja
    instrukcja
    itd.
} Te instrukcje stanowią ciało pętli. Będą one wykonywane dopóki,
    spełniony będzie warunek.
End While

```

Warunek oznacza dowolne wyrażenie boolowskie, a instrukcje zawarte między klawiszami While i End While nazywamy **ciałem pętli**. Kiedy uruchomisz się pętla, sprawdzony zostanie warunek. Jeżeli jest on spełniony, zostaną uruchomione instrukcje składające się na ciało pętli, a następnie pętla uruchomisz się ponownie. Jeżeli warunek nie jest spełniony, program opuści pętlę.

Jak widzisz, do zapisu takiej ogólnej postaci pętli While należy trzymać się pewnej konwencji:

- klauzule While i End While powinny być wyrównane do lewej strony;
- instrukcje składające się na ciało pętli powinny być wcięte.

Dzięki wcięciom linie stanowiące ciało pętli będą się wizualnie odróżniały od reszty kodu — program będzie czytelniejszy i łatwiej go będzie debugować. Ponadto taką konwencją zapisu pętli posługuje się większość programistów.

Na listingu 5.1 pokazałem, w jaki sposób możemy użyć pętli While w programie służącym do obliczania premii, który opisałem na początku rozdziału.

### Listing 5.1

```

1 // Deklaracja zmiennych
2 Declare Real sales, commission
3 Declare String keepGoing = "t"
4
5 // Stała zawierająca wysokość premii
6 Constant Real COMMISSION_RATE = 0.10
7
8 While keepGoing == "t"
9     // Pobieramy wartość sprzedaży
10    Display "Wprowadź wartość sprzedaży."
11    Input sales
12
13    // Obliczamy premię
14    Set commission = sales * COMMISSION_RATE
15
16    // Wyświetlamy premię
17    Display "Premia wynosi ", commission, " zł."
18
19    Display "Czy chcesz obliczyć kolejną?"
20    Display "premii? (Jeśli tak, wpisz t)"
21    Input keepGoing
22 End While

```

#### Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)

Wprowadź wartość sprzedaży.

**10000.00 [Enter]**

Premia wynosi 1000 zł.

Czy chcesz obliczyć kolejną premię? (Jeśli tak, wpisz t)

**t [Enter]**

Wprowadź wartość sprzedaży.

```

5000.00 [Enter]
Premia wynosi 500 zł.
Czy chcesz obliczyć kolejną
premię? (Jeśli tak, wpisz t)
t [Enter]
Wprowadź wartość sprzedaży.
12000.00 [Enter]
Premia wynosi 1200 zł.
Czy chcesz obliczyć kolejną
premię? (Jeśli tak, wpisz t)
n [Enter]

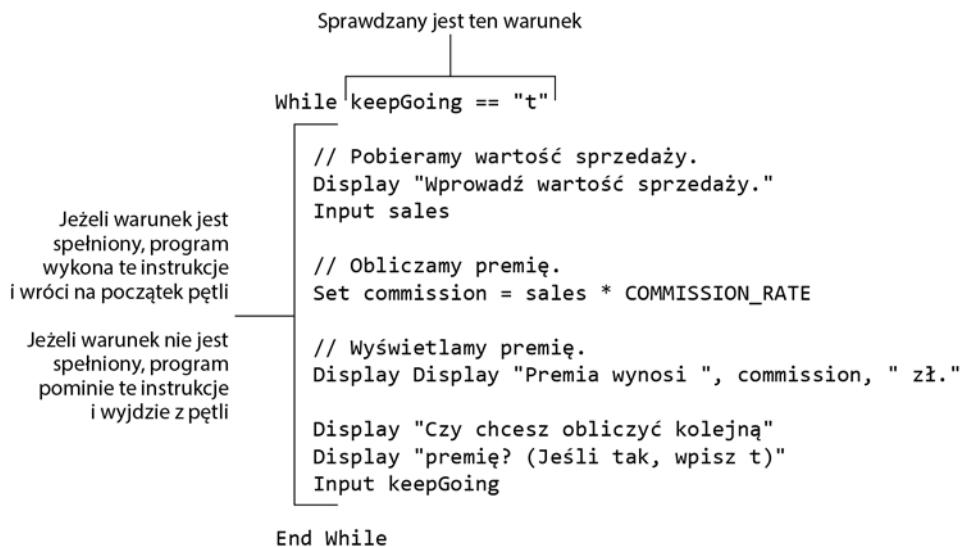
```

W linii 2. deklaruję zmienną `sales`, w której będę zapisywać wartość sprzedaży, i zmienną `commission`, w której będę zapisywać wartość premii. Następnie w linii 3. deklaruję ciąg znaków `String` o nazwie `keepGoing`. Zwróć uwagę, że zainicjalizowałem go wartością `"t"`. Jest to bardzo istotne — za chwilę się dowiesz dlaczego. W linii 6. deklaruję stałą `COMMISSION_RATE` i inicjalizuję ją wartością `0.10`. Jest to wysokość premii, której użyję w obliczeniach.

Pętla `While` rozpoczyna się w linii 8.:

```
While keepGoing == "t"
```

Zwróć uwagę na warunek, który będzie sprawdzany: `keepGoing == "t"`. Pętla będzie sprawdzała ten warunek i jeśli będzie on spełniony, uruchomią się instrukcje składające się na ciało pętli (w liniach od 9. do 21.). Następnie pętla powraca do linii 8. i kolejny raz sprawdza warunek `keepGoing == "t"`. Jeżeli jest on spełniony, znowu wykonają się instrukcje w ciele pętli. Ten proces będzie się powtarzał aż do momentu, gdy warunek `keepGoing == "t"` w linii 8. nie zostanie spełniony. Kiedy tak się stanie, program opuści pętlę. Ilustruje to rysunek 5.2.



Rysunek 5.2. Pętla While

Aby pętla zakończyła działanie, musi nastąpić wewnątrz niej coś, co spowoduje, że warunek `keepGoing == "t"` przestanie być prawdziwy. Służą do tego instrukcje w liniach od 19. do 21. W liniach 19. i 20. wyświetla się pytanie: Czy chcesz obliczyć kolejną premię? (Jeśli tak, wpisz t). Następnie instrukcja `Input` w linii 21. odczytuje, jaki klawisz nacisnął użytkownik, i zapisuje go w zmiennej `keepGoing`. Jeżeli użytkownik wprowadził „t” (musi to być mała litera „t”), wtedy wyrażenie `keepGoing == "t"` nadal będzie spełnione i pętla po raz kolejny wykona umieszczone w niej instrukcje. Jednak jeżeli użytkownik wprowadzi jakiś inny znak niż mała litera „t”, warunek nie zostanie spełniony i program wyjdzie z pętli.

Kiedy już zrozumiesz pseudokod, przyjrzyj się przykładowym wynikom zwracanym przez program. Najpierw program poprosił użytkownika o wprowadzenie wartości sprzedaży. Użytkownik wprowadził `10000.00`, a następnie program wyświetlił wartość premii równą `1000.00 zł`. Program zapytał następnie użytkownika: Czy chcesz obliczyć kolejną premię? (Jeśli tak, wpisz t). Użytkownik wprowadził „t” i pętla uruchomiła się ponownie. W tym przypadku użytkownik postąpił w ten sposób trzykrotnie. Każdorazowe uruchomienie instrukcji zawartych w ciele pętli nazywamy **iteracją**. Tak więc w tym przypadku pętla wykonała trzy iteracje.

Na rysunku 5.3 przedstawiony jest schemat blokowy programu z listingu 5.1. Na schemacie można zauważyć, że wewnątrz struktury cyklicznej (pętli `While`) zagnieżdziona jest struktura sekwencyjna (ciało pętli). Zachowana jest jednak podstawowa konstrukcja pętli `While` — sprawdzany jest określony warunek i jeżeli warunek ten jest spełniony, uruchamiają się pewne instrukcje, a następnie program zwraca do miejsca, gdzie sprawdzany jest warunek.

## W pętli While warunek jest sprawdzany na początku

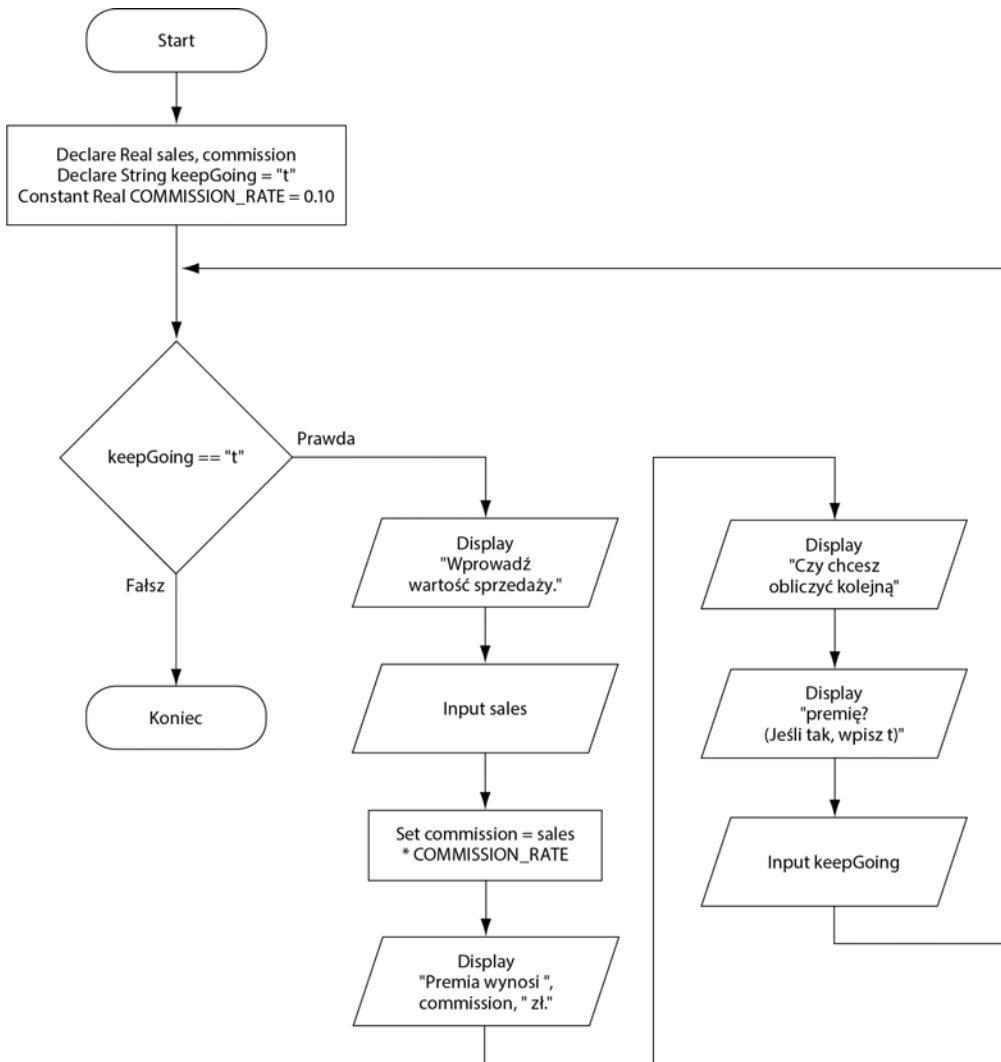
Pętla `While` jest rodzajem pętli, w której warunek jest sprawdzany na samym początku — czyli zanim pętla wykona pierwszą iterację. Ponieważ warunek jest sprawdzany na początku, oznacza to, że przed pętlą musimy wykonać kroki, dzięki którym będziemy mieli pewność, że pętla wykona się przynajmniej jeden raz. W naszym przykładzie z listingu 5.1 warunek wygląda tak:

```
While keepGoing == "t"
```

Pętla wykona więc iterację tylko wtedy, gdy wyrażenie `keepGoing == "t"` zwróci prawdę. Aby mieć pewność, że tak właśnie będzie w pierwszej iteracji, w linii 3. zadeklarowałem i zainicjalizowałem zmienną `keepGoing` w ten sposób:

```
Declare String keepGoing = "t"
```

Jeśli zainicjalizowałbym zmienną `keepGoing` jakąś inną wartością (lub nie zrobiłbym tego w ogóle), pętla by się nie uruchomiła. To bardzo istotna cecha pętli `While` — nie uruchomi się ona, gdy warunek nie będzie spełniony. W niektórych programach ta cecha okaże się dokładnie tym, czego potrzebujemy. W sekcji „W centrum uwagi” przedstawiłem przykład.



**Rysunek 5.3.** Schemat blokowy programu z listingu 5.1

## W centrum uwagi

### Projektowanie pętli While

Firma Chemical Labs realizuje projekt, który wymaga, aby substancja znajdująca się w kadzi była stale podgrzewana. Pracownik, który nadzoruje ten proces, musi sprawdzać temperaturę substancji regularnie co 15 minut. Jeżeli temperatura nie przekracza 102,5°C, pracownik nie musi nic robić. Jednak gdy temperatura przekroczy 102,5°C, pracownik musi wyłączyć termostat, odczekać 5 minut i sprawdzić ponownie temperaturę. Pracownik powtarza te kroki aż do momentu, gdy temperatura nie przekracza 102,5. Dyrektor działu technicznego poprosił Cię o zaprojektowanie programu, który ułatwi pracę pracownikowi nadzorującemu proces podgrzewania substancji.



Oto algorytm:

1. Pobierz temperaturę substancji.
2. Powtarzaj poniższe kroki dopóty, dopóki temperatura jest wyższa niż 102,5°C:
  - a) Proś pracownika, aby wyłączył termostat, oczekaj 5 minut i sprawdź ponownie temperaturę.
  - b) Pobierz temperaturę substancji.
3. Kiedy program wyjdzie z pętli, poinformuj pracownika, że temperatura jest prawidłowa, i poproś go, aby sprawdził ją ponownie za 15 minut.

Kiedy przyjrzysz się temu algorytmowi, zauważysz, że gdy warunek (temperatura wyższa niż 102,5°C) na początku nie jest spełniony, nie należy wykonywać kroków 2a) i 2b). Możemy więc w tym przypadku wykorzystać pętlę `While`, ponieważ jeżeli określony warunek nie jest spełniony, nie wykona się ona ani razu. Na listingu 5.2 przedstawiłem pseudokod programu, a na rysunku 5.4 jego schemat blokowy.

### **Listing 5.2**



```

1 // Zmienne, w której zapiszemy wartość temperatury
2 Declare Real temperature
3
4 // Stała zawierająca maksymalną temperaturę
5 Constant Real MAX_TEMP = 102.5
6
7 // Pobieramy temperaturę substancji
8 Display "Wprowadź temperaturę substancji."
9 Input temperature
10
11 // Jeżeli trzeba, wyłącz termostat
12 While temperature > MAX_TEMP
13   Display "Temperatura jest za wysoka."
14   Display "Wyłącz termostat i oczekaj"
15   Display "pięć minut. Zmierz ponownie temperaturę"
16   Display "i ją wprowadź."
17   Input temperature
18 End While
19
20 // Przypominamy użytkownikowi, aby ponownie sprawdził
21 // temperaturę za 15 minut
22 Display "Temperatura jest prawidłowa."
23 Display "Sprawdź ją ponownie za 15 minut."

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź temperaturę substancji.

**104.7 [Enter]**

Temperatura jest za wysoka.

Wyłącz termostat i oczekaj

pięć minut. Zmierz ponownie temperaturę  
i ją wprowadź.

**103.2 [Enter]**

Temperatura jest za wysoka.

Wyłącz termostat i oczekaj

pięć minut. Zmierz ponownie temperaturę  
i ją wprowadź.

**102.1 [Enter]**

Temperatura jest prawidłowa.  
Sprawdź ją ponownie za 15 minut.

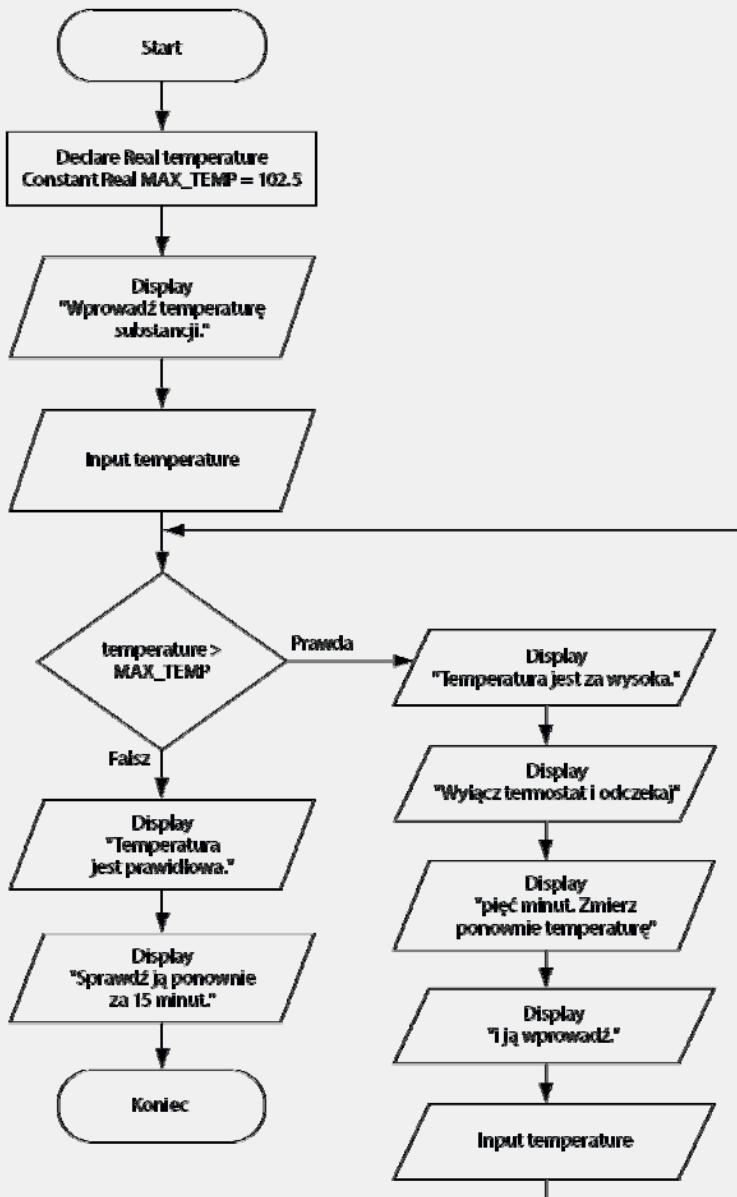
**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź temperaturę substancji.

**102.1 [Enter]**

Temperatura jest prawidłowa.

**Sprawdź ją ponownie za 15 minut.**



Rysunek 5.4. Schemat blokowy programu z listingu 5.2

## Pętle nieskończono

W niemal wszystkich przypadkach w pętli musi się znaleźć operacja, która pozwoli opuścić pętlę — czyli coś, co sprawi, że warunek w pewnym momencie przestanie być spełniony. Pętla w programie z listingu 5.1 zakończy się, gdy wyrażenie `keepGoing == "t"` zwróci fałsz. Jeśli w pętli nie ma takiej operacji, będzie ona pętlą nieskończoną. **Pętla nieskończona** będzie się wykonywała aż do momentu, gdy zamkniesz program. Pętle nieskończono pojawiają się w programie zazwyczaj wtedy, kiedy programista zapomni umieścić w ciele pętli instrukcję, która będzie powodowała, że warunek przestanie być prawdziwy. W znaczącej większości przypadków należy unikać pętli nieskończonych.

Na listingu 5.3 przedstawiłem przykład pętli nieskończonej. Jest to zmodyfikowana wersja programu do obliczania premii. W tym przypadku usunąłem z ciała pętli instrukcję, która przypisywała wartość do zmiennej `keepGoing`. Podczas sprawdzania warunku `keepGoing == "t"` w linii 9. do zmiennej `keepGoing` zawsze będzie przypisana wartość "t". W wyniku tego pętla nigdy się nie skończy.

### **Listing 5.3**

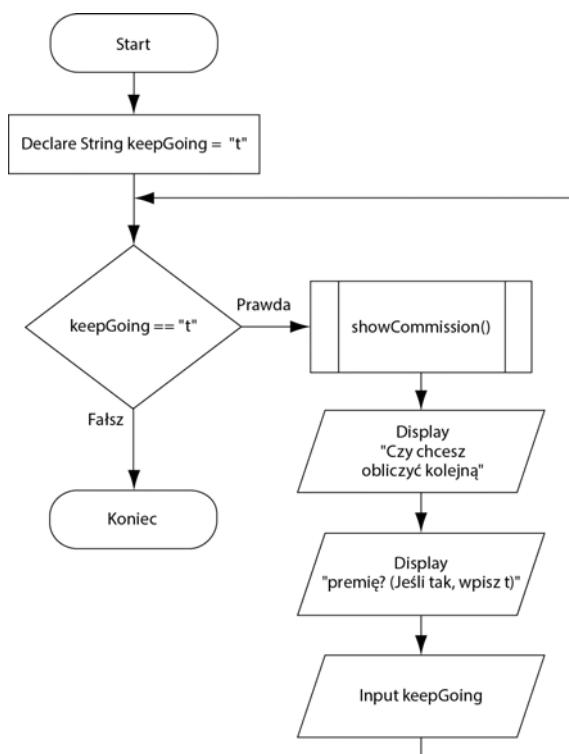
```

1 // Deklaracja zmiennych
2 Declare Real sales, commission
3 Declare String keepGoing = "t"
4
5 // Stała zawierająca wysokość premii
6 Constant Real COMMISSION_RATE = 0.10
7
8 // Uwaga! Pętla nieskończona!
9 While keepGoing == "t"
10    // Pobieramy wartość sprzedaży
11    Display "Wprowadź wartość sprzedaży."
12    Input sales
13
14    // Obliczamy premię
15    Set commission = sales * COMMISSION_RATE
16
17    // Wyświetlamy premię
18    Display "Premia wynosi ", commission, " zł."
19 End While

```

## Modularyzacja kodu w ciele pętli

Wewnątrz ciała pętli można także wywoływać moduły. Dzięki temu możemy otrzymać lepszy projekt. Na przykładzie z listingu 5.1 moglibyśmy umieścić w module instrukcje, które służą do pobrania od użytkownika wartości sprzedaży oraz obliczenia i wyświetlenia premii. Moduł taki moglibyśmy następnie wywołać wewnątrz pętli. Na listingu 5.4 pokazałem, jak można to zrobić. Program składa się z modułu `main`, który zostanie wywołany po uruchomieniu programu, i z modułu `showCommission`, którego zadaniem jest wykonanie wszystkich operacji związanych z obliczaniem i wyświetlaniem premii. Na rysunku 5.5 widoczny jest schemat blokowy modułu `main`, a na rysunku 5.6 schemat blokowy modułu `showCommission`.

**Rysunek 5.5.** Moduł main programu z listingu 5.4**Rysunek 5.6.** Moduł showCommission**Listing 5.4**

```

1 Module main()
2   // Zmienna lokalna
3   Declare String keepGoing = "t"
4
5   // Obliczamy tyle razy premię,
6   // ile będzie trzeba
7   While keepGoing == "t"
8     // Wyświetlamy premię dla danego sprzedawcy
9     Call showCommission()
10
11    // Jeszcze raz?
12    Display "Czy chcesz obliczyć kolejną"
13    Display "premię? (Jeśli tak, wpisz t)"
14    Input keepGoing
15  End While
16 End Module
17
18 // Moduł showCommission pobiera
19 // wartość sprzedaży i wyświetla
20 // premię
21 Module show Commission()
22   // Zmienne lokalne
23   Declare Real sales, commission
  
```

```

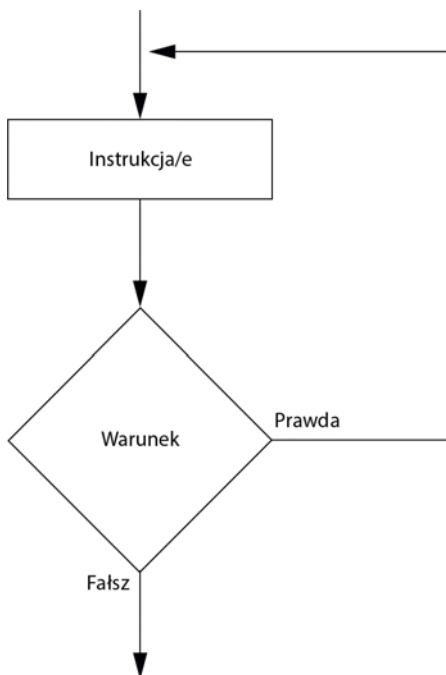
24
25 // Stała zawierająca wysokość premii
26 Constant Real COMMISSION_RATE = 0.10
27
28 // Pobieramy wartość sprzedaży
29 Display "Wprowadź wartość sprzedaży."
30 Input sales
31
32 // Obliczamy premię
33 Set commission = sales * COMMISSION_RATE
34
35 // Wyświetlamy premię
36 Display "Premia wynosi ", commission, " zł."
37 End Module

```

**Wynik działania tego programu jest taki sam jak na listingu 5.1**

## Pętla Do-While

Wiesz już, że w przypadku pętli While warunek jest sprawdzany jeszcze przed pierwszą iteracją. W przypadku pętli Do-While jest inaczej: warunek jest sprawdzany dopiero po wykonaniu iteracji. Dlatego pętla Do-While zawsze wykona co najmniej jedną iterację — nawet jeśli warunek już na samym początku nie jest spełniony. Zasadę działania pętli Do-While pokazałem na rysunku 5.7.



Rysunek 5.7. Zasada działania pętli Do-While

Jak widać na schemacie blokowym, najpierw wykonują się określone instrukcje, a następnie sprawdzany jest warunek. Jeżeli warunek jest spełniony, program powraca na początek pętli i proces zaczyna się jeszcze raz. Jeżeli warunek nie jest spełniony, program wychodzi z pętli.

## Tworzenie pętli Do-While w pseudokodzie

W pseudokodzie pętlę Do-While będziemy tworzyć za pomocą instrukcji Do-While. Oto jej ogólna postać:

```

Do
  instrukcja
  instrukcja
  itd.
While warunek
  } Te instrukcje stanowią ciało pętli. W każdym przypadku
      zostaną wykonane co najmniej jeden raz, a jeżeli warunek
      jest spełniony, zostaną wykonane ponownie.

```

Instrukcje znajdujące się pomiędzy klauzulami Do i While stanowią ciało pętli. Warunek, który pojawia się po klauzuli While, jest wyrażeniem boolowskim. Po uruchomieniu pętli program wykona instrukcje zawarte w ciele pętli, a następnie sprawdzi warunek. Jeżeli warunek jest spełniony, pętla uruchomi się ponownie i program ponownie wykona instrukcje umieszczone w ciele pętli. Jeżeli warunek nie jest spełniony, program wyjdzie z pętli.

Jak widzisz, podczas zapisywania pętli Do-While należy trzymać się następującej konwencji:

- klauzule Do i While powinny być wyrównane do lewej strony;
- instrukcje składające się na ciało pętli powinny być wcięte.

Jak widać na listingu 5.5, program do obliczania premii można bardzo łatwo zmodyfikować, aby działał w oparciu o pętlę Do-While. Zwróć uwagę, że w tej wersji programu w linii 3. nie inicjalizuję już zmiennej keepGoing wartością "t". Nie muszę tego robić, ponieważ pętla umieszczona w liniach od 7. do 15. uruchomi się co najmniej jeden raz. Oznacza to, że uruchomi się także polecenie Input w linii 14., za pomocą którego pobieram od użytkownika wartość, którą następnie zapisuję do zmiennej keepGoing — jeszcze zanim warunek zostanie po raz pierwszy sprawdzony w linii 15.

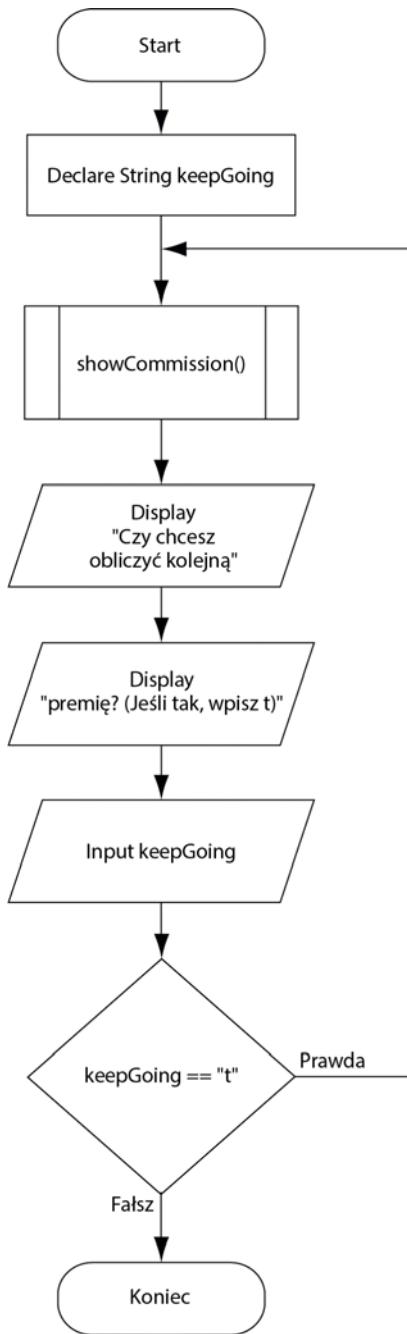
Na rysunku 5.8 widoczny jest schemat blokowy modułu main.

### Listing 5.5

```

1 Module main()
2   // Zmienna lokalna
3   Declare String keepGoing
4
5   // Obliczamy tyle razy premię,
6   // ile będzie trzeba
7   Do
8     // Wyświetlamy premię dla danego sprzedawcy
9     Call showCommission()
10

```



Rysunek 5.8. Schemat blokowy modułu main programu z listingu 5.5

```

11 // Jeszcze raz?
12 Display "Czy chcesz obliczyć kolejną"
13 Display "premię? (Jeśli tak, wpisz t)"
14 Input keepGoing
15 While keepGoing == "t"
16 End Module
17
18 // Moduł showCommission pobiera
19 // wartość sprzedaży i wyświetla
20 // premię
21 Module showCommission()
22   // Zmienne lokalne
23   Declare Real sales, commission
24
25   // Stała zawierająca wysokość premii
26   Constant Real COMMISSION_RATE = 0.10
27
28   // Pobieramy wartość sprzedaży
29   Display "Wprowadź wartość sprzedaży."
30   Input sales
31
32   // Obliczamy premię
33   Set commission = sales * COMMISSION_RATE
34
35   // Wyświetlamy premię
36   Display "Premia wynosi ", commission, " zł."
37 End Module

```

### **Wynik działania tego programu jest taki sam jak na listingu 5.1**

Mimo że pętle Do-While są w pewnych sytuacjach bardzo pomocne, nie są one niezastąpione. Każdą pętlę Do-While można także zapisać jako pętlę While. Jak wspomniałem wcześniej, aby mieć pewność, że pętla While uruchomi się co najmniej raz, najczęściej trzeba zainicjalizować pewną zmienną.

## **W centrum uwagi**

### Projektowanie pętli Do-While

Agata prowadzi sklep i ceny detaliczne produktów oblicza w następujący sposób:

$$\text{cena detaliczna} = \text{cena hurtowa} \cdot 2,5$$

Agata poprosiła Cię, abyś zaprojektował program, który będzie wykonywał to działanie dla każdego produktu, który przyjdzie w dostawie. Dowiedziałeś się, że w każdej dostawie może znajdować się dowolna liczba produktów, więc postanowileś, że to za pomocą pętli będziesz prosić użytkownika o wprowadzenie ceny każdego z produktów, a następnie zapytasz go, czy chce przejść do kolejnego produktu. Pętla będzie wykonywała kolejne iteracje tak długo, jak użytkownik będzie wskazywał, że chce wprowadzić cenę kolejnego produktu. Na listingu 5.6 przedstawiłem pseudokod programu, a na rysunku 5.9 znajduje się schemat blokowy.



**Listing 5.6**

```

1 Module main()
2   //Zmienna lokalna
3   Declare String doAnother
4
5   Do
6     //Obliczamy i wyświetlamy cenę detaliczną
7     Call showRetail()
8
9     //Jeszcze raz?
10    Display "Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)"
11    Input doAnother
12    While doAnother == "t" OR doAnother == "T"
13  End Module
14
15 //Moduł showRetail pobiera od użytkownika cenę hurtową
16 //i wyświetla cenę detaliczną
17 Module showRetail()
18   //Zmienne lokalne
19   Declare Real wholesale, retail
20
21   //Stała zawierająca marżę
22   Constant Real MARKUP = 2.50
23
24   //Pobieramy cenę hurtową
25   Display "Wprowadź cenę hurtową produktu."
26   Input wholesale
27
28   //Obliczamy cenę detaliczną
29   Set retail = wholesale * MARKUP
30
31   //Wyświetlamy cenę detaliczną
32   Display "Cena detaliczna wynosi ", retail, " zł."
33 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę hurtową produktu.

**10.00 [Enter]**

Cena detaliczna wynosi 25 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)

**t [Enter]**

Wprowadź cenę hurtową produktu.

**15.00 [Enter]**

Cena detaliczna wynosi 37.50 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)

**t [Enter]**

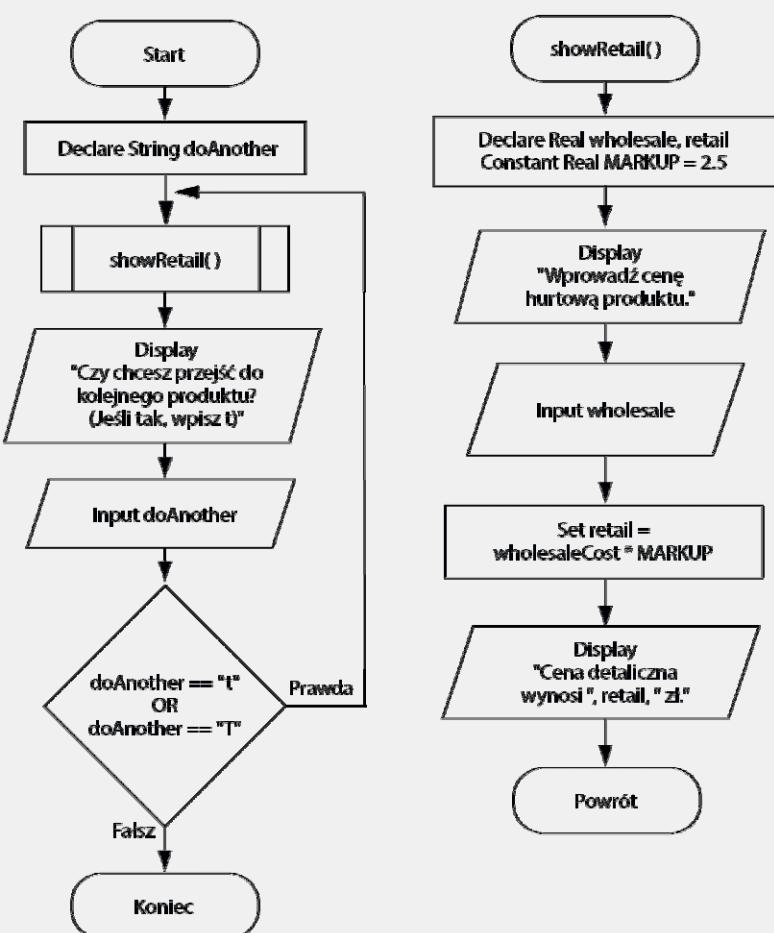
Wprowadź cenę hurtową produktu.

**12.50 [Enter]**

Cena detaliczna wynosi 31.25 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)

**n [Enter]**



Rysunek 5.9. Schemat blokowy programu z listingu 5.6

Program składa się z dwóch modułów: `main`, który zostanie uruchomiony razem z programem, i `showRetail`, który oblicza i wyświetla cenę detaliczną produktu. W module `main`, w liniach od 5. do 12., pojawia się pętla `Do-While`. W linii 7. wywołuję w pętli moduł `showRetail`. Następnie, w linii 10., pytam użytkownika: `Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)`. W linii 11. pobieram wartość, którą wprowadził użytkownik, i przypisuję ją do zmiennej `doAgain`. Pętlę kończę następujące polecenie w linii 12.:

```
While doAnother == "t" OR doAnother == "T"
```

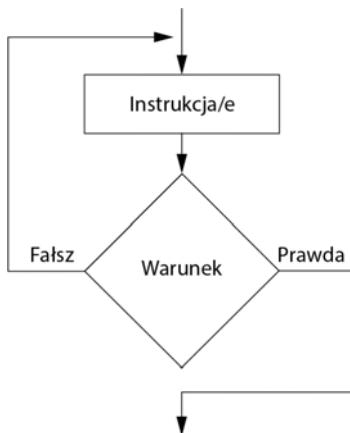
Zwróć uwagę, że w tym miejscu zastosowałem złożone wyrażenie boolowskie z operatorem logicznym `OR`. Wyrażenie po lewej stronie operatora `OR` zwróci prawdę, gdy użytkownik wprowadzi na klawiaturze małą literę `"t"`. Wyrażenie po prawej stronie operatora `OR` zwróci prawdę, gdy użytkownik wprowadzi dużą literę `"T"`. Jeśli którekolwiek z tych podwyrażeń zwróci prawdę, pętla wykona się jeszcze raz. W ten prosty sposób mamy pewność, że niezależnie od tego, czy użytkownik wprowadzi małą czy dużą literę, warunek będzie spełniony.

## Pętla Do-Until

Zarówno pętla `While`, jak i pętla `Do-While` wykonują się dopóty, dopóki spełniony jest warunek. W pewnych sytuacjach jednak bardziej poręczne będzie, jeżeli pętla będzie się wykonywała do momentu, gdy warunek zostanie spełniony. Oznacza to, że pętla będzie się wykonywała dopóty, dopóki warunek nie jest spełniony, a przestanie działać, gdy warunek zostanie spełniony.

Wyobraź sobie przykład linii lakierniczej, na której przesuwają się samochody. Kiedy na linii nie będzie już żadnego samochodu, linia powinna się zatrzymać. Gdybyś musiał zaprogramować taką linię, wybrałbyś zapewne pętlę, która będzie przesuwała linię aż do momentu, gdy nie będzie już samochodów do polakierowania.

Pętlę, która wykonuje iteracje aż do momentu, gdy warunek zostanie spełniony, nazywamy pętlą `Do-Until`. Na rysunku 5.10 przedstawiłem zasadę działania pętli `Do-Until`.



**Rysunek 5.10.** Zasada działania pętli Do-Until

Zwróć uwagę, że w pętli `Do-Until` warunek jest sprawdzany na końcu. Najpierw więc wykona się jedna lub więcej instrukcji, a dopiero potem sprawdzany jest warunek. Jeżeli warunek nie jest spełniony, pętla wraca na początek i cały proces się powtarza. Jeżeli warunek jest spełniony, program wychodzi z pętli.



**UWAGA:** Ponieważ w pętli `Do-Until` warunek jest sprawdzany na końcu, w każdym przypadku wykona ona co najmniej jedną iterację.

## Tworzenie pętli Do-Until w pseudokodzie

W pseudokodzie pętlę `Do-Until` będziemy tworzyć za pomocą instrukcji `Do-Until`. Oto jej ogólna postać:

Do  
*instrukcja*  
*instrukcja*  
*itd.*  
} Te instrukcje stanowią ciało pętli. W każdym przypadku zostaną wykonane co najmniej jeden raz i będą wykonywane powtórnie, aż do momentu, gdy warunek zostanie spełniony.  
Until warunek

Instrukcje znajdujące się pomiędzy klauzulami Do i Until stanowią ciało pętli. Warunek, który pojawia się po klauzuli Until, jest wyrażeniem boolowskim. Po uruchomieniu pętli program wykona instrukcje zawarte w ciele pętli, a następnie sprawdzi warunek. Jeżeli warunek jest spełniony, program wyjdzie z pętli. Jeżeli warunek nie jest spełniony, pętla uruchomi się ponownie i program ponownie wykona instrukcje umieszczone w ciele pętli.

Jak widzisz, podczas zapisywania w kodzie pętli Do-Until należy trzymać się następującej konwencji:

- klauzule Do i Until powinny być wyrównane do lewej strony;
- instrukcje składające się na ciało pętli powinny być wcięte.

Na listingu 5.7 przedstawiłem przykład, w którym wykorzystałem pętlę Do-Until. Pętla w liniach od 6. do 16. prosi użytkownika o wprowadzenie hasła aż do momentu, gdy wprowadzi on ciąg znaków "prospero". Na rysunku 5.11 widoczny jest schemat blokowy programu.

### Listing 5.7

```

1 // Deklarujemy zmienną, w której zapiszemy hasło
2 Declare String password
3
4 // Prosimy użytkownika o wprowadzenie hasła aż do momentu,
5 // gdy hasło będzie prawidłowe
6 Do
7     // Prosimy użytkownika o wprowadzenie hasła
8     Display "Wprowadź hasło."
9     Input password
10
11    // Jeżeli hasło jest błędne,
12    // wyświetlamy komunikat
13    If password != "prospero" Then
14        Display "Przykro mi, spróbuj jeszcze raz."
15    End If
16 Until password == "prospero"
17
18 // Informujemy użytkownika, że hasło jest poprawne
19 Display "Hasło jest poprawne."
```

### Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)

Wprowadź hasło.

**ariel [Enter]**

Przykro mi, spróbuj jeszcze raz.

Wprowadź hasło.

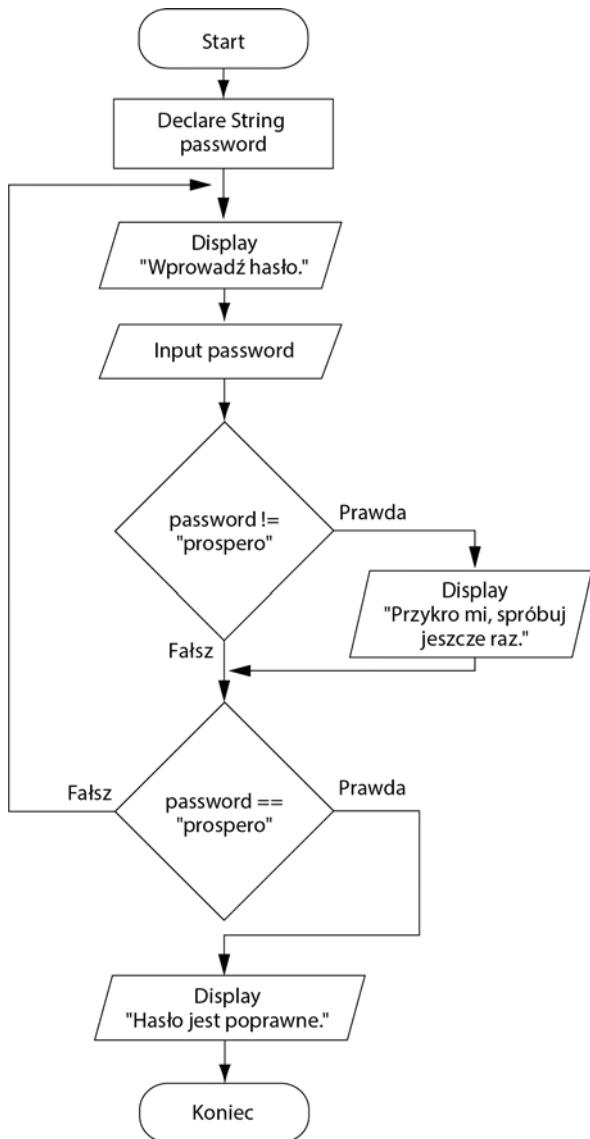
**caliban [Enter]**

Przykro mi, spróbuj jeszcze raz.

Wprowadź hasło.

**prospero [Enter]**

Hasło jest poprawne.



Rysunek 5.11. Schemat blokowy programu z listingu 5.7



**UWAGA:** Pętla Do-Until występuje nie we wszystkich językach programowania, ponieważ można ją w prosty sposób zastąpić pętlą Do-While.

## Wybór odpowiedniego rodzaju pętli

W tym podrozdziale przedstawiłem trzy rodzaje pętli warunkowych: While, Do-While i Do-Until. Kiedy przystąpisz do pisania programu, który będzie wymagał wykorzystania pętli warunkowej, będziesz musiał dokonać wyboru — która z ich będzie odpowiednia.

Pętla `While` przydaje się, gdy trzeba powtarzać pewne operacje tak długo, jak określony warunek jest spełniony. W pętli `While` warunek jest sprawdzany na początku, więc idealnie nadaje się do zadań, w których warunek od samego początku może być nieprawdziwy, i pętla w takiej sytuacji nie powinna wykonać ani jednej iteracji. Dobrym przykładem jest pseudokod, który pokazałem na listingu 5.2.

Pętla `Do-While` także przyda się w zadaniach, w których operacje trzeba wykonywać dopóty, dopóki spełniony jest warunek. W tej pętli warunek jest sprawdzany na końcu, więc użyj jej, gdy chcesz, aby pętla wykonała co najmniej jedną iterację — niezależnie od tego, czy warunek jest spełniony, czy nie.

W pętli `Do-Until` warunek także sprawdzany jest na końcu, więc w tym przypadku pętla również wykona co najmniej jedną iterację. Wybierz ją w sytuacji, gdy chcesz, aby określone operacje były wykonywane *do momentu*, aż warunek zostanie spełniony. Pętla `Do-Until` działa dopóty, dopóki warunek nie jest spełniony. Kiedy warunek zostanie spełniony, pętla zakończy działanie.



## Punkt kontrolny

- 5.4. Co to jest iteracja pętli?
- 5.5. Czym różni się pętla, w której warunek jest sprawdzany na początku, od pętli, w której warunek jest sprawdzany na końcu?
- 5.6. Czy w przypadku pętli `While` warunek jest sprawdzany przed wykonaniem pierwszej iteracji czy po?
- 5.7. Czy w przypadku pętli `Do-While` warunek jest sprawdzany przed wykonaniem pierwszej iteracji czy po?
- 5.8. Co to jest pętla nieskończona?
- 5.9. Czym różnią się pętle `Do-While` i `Do-Until`?

**5.3**

## Pętle licznikowe i instrukcja For

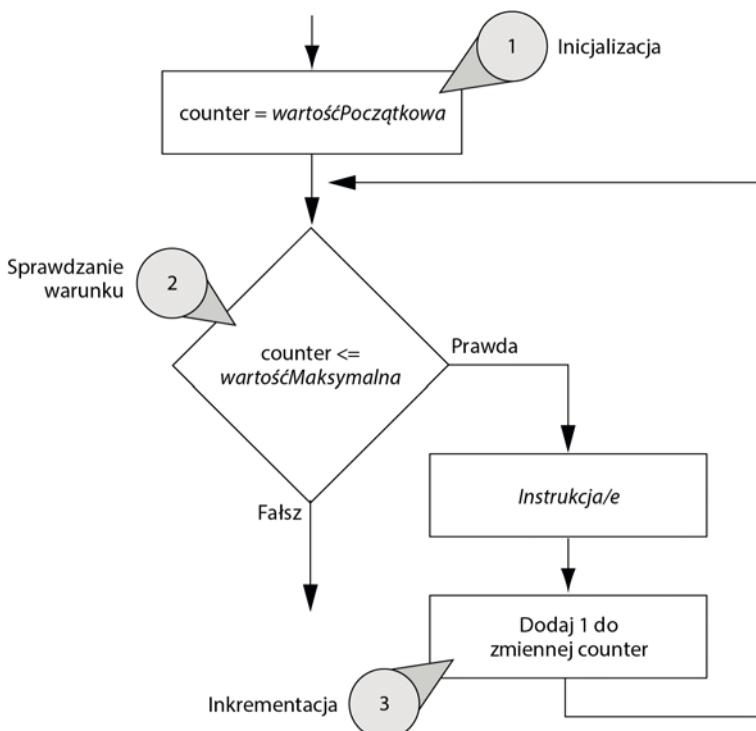
**WYJAŚNIENIE:** Pętle licznikowe wykonują ścisłe określona liczbę iteracji. Mimo faktu, że pętlę warunkową można zaprojektować w taki sposób, aby wykonywała określona liczbę iteracji, w większości języków programowania dostępna jest pętla `For`, będąca zaprojektowaną specjalnie do tego celu pętlą licznikową.

Jak wspomniałem na początku rozdziału, pętle licznikowe wykonują ścisłe określona liczbę iteracji. Są one bardzo często wykorzystywane w programach. Założymy przykładowo, że sklep jest otwarty przez sześć dni w tygodniu i masz zamiar napisać program, który będzie obliczał sumaryczną wartość sprzedaży w całym tygodniu. Będziesz w takim przypadku potrzebować pętli, która wykona dokładnie sześć iteracji. Podczas każdej iteracji poprosisz użytkownika o wprowadzenie wartości sprzedaży w danym dniu.

Zasada działania pętli licznikowej jest bardzo prosta: pętla zlicza, ile wykonała już iteracji, i gdy liczba ta będzie równa określonej wartości, zakończy działanie. Do zliczania iteracji w pętli używa się zmiennej zwanej **zmienią licznikową** (ang. *counter variable*) lub po prostu **licznikiem**. Pętla wykonuje na zmiennej licznikowej zazwyczaj trzy operacje: **inicjalizuje ją**, **sprawdza warunek** i **inkrementuje**:

1. **Inicjalizacja** — zanim pętla się uruchomi, zmenna licznikowa inicjalizowana jest określoną wartością początkową. Wartość ta będzie różna w zależności od zadania.
2. **Sprawdzanie warunku** — pętla porównuje zmenną licznikową do pewnej wartości maksymalnej. Jeżeli zmenna licznikowa jest mniejsza od tej wartości lub jej równa, pętla wykonuje kolejną iterację. Jeżeli zmenna licznikowa jest większa od wartości maksymalnej, program opuszcza pętlę.
3. **Inkrementacja** — inkrementacja zmiennej oznacza zwiększenie jej wartości. Podczas każdej iteracji pętla inkrementuje wartość zmiennej licznikowej, dodając do niej 1.

Na rysunku 5.12 przedstawiłem zasadę działania pętli licznikowej. Inicjalizację, sprawdzanie warunku i inkrementację oznaczylem znacznikami ①, ② i ③,

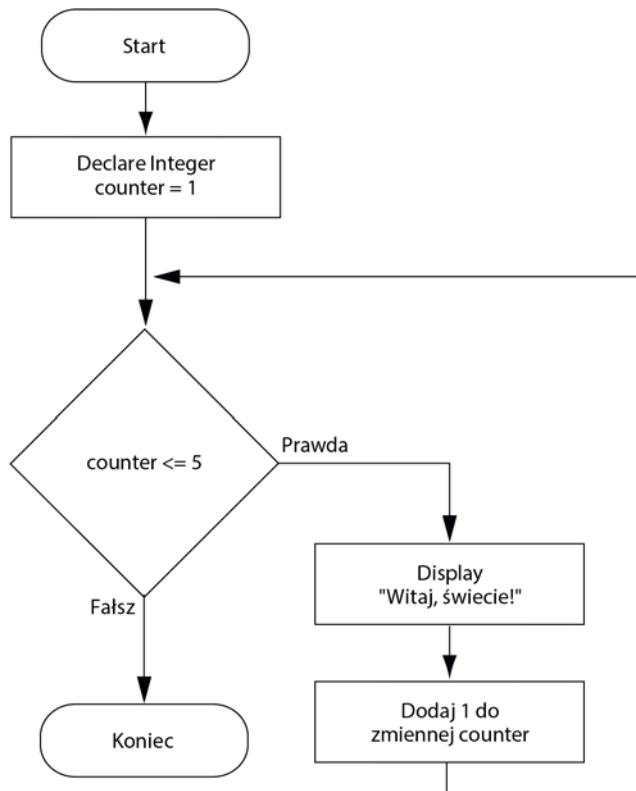


**Rysunek 5.12.** Zasada działania pętli licznikowej

Na schemacie blokowym założyłem, że zmienią counter jest typu Integer. Pierwszym krokiem jest przypisanie do zmiennej counter wartości początkowej. Następnie program sprawdza, czy zmienią counter jest mniejsza lub równa wartości maksy-

malnej. Jeżeli tak, wykonane zostaną instrukcje umieszczone w ciele pętli. W przeciwnym razie program wyjdzie z pętli. Zwróć uwagę, że w ciele pętli wykonuje się jedna lub więcej instrukcji, a następnie do zmiennej counter dodawana jest wartość równa 1.

Spójrz na przykładowy schemat blokowy widoczny na rysunku 5.13. Najpierw deklaruję zmienną typu Integer o nazwie counter i inicjalizuję ją wartością początkową równą 1. Następnie program sprawdza warunek counter  $\leq 5$ . Jeśli wyrażenie to zwraca prawdę, wyświetli się komunikat *Witaj, świecie!*, a następnie do zmiennej counter dodawana jest wartość 1. W przeciwnym przypadku program wychodzi z pętli. Jeśli prześledzisz działanie tego programu, zauważysz, że pętla wykona pięć iteracji.



Rysunek 5.13. Pętla licznikowa

## Instrukcja For

Pętle licznikowe są tak często wykorzystywane w programach, że większość języków programowania ma specjalną instrukcję przeznaczoną do tego celu. Zazwyczaj nazywa się ona For. Instrukcja ta została zaprojektowana tak, aby można było za jej pomocą inicjalizować zmienną licznikową, sprawdzać warunek i inkrementować zmienną licznikową. Oto ogólna postać instrukcji For, z której będziemy korzystali w pseudokodzie:

```

For zmiennaLiczniowa = wartośćPoczątkowa To wartośćMaksymalna
    instrukcja
    instrukcja
    instrukcja
    itd.
End For
  
```

**Te instrukcje stanowią ciało pętli.**

*ZmiennaLiczniowa* to nazwa zmiennej, która będzie pełniła funkcję zmiennej licznikowej, *wartośćPoczątkowa* to wartość, jaką zmienna ta zostanie zainicjalizowana, a *wartośćMaksymalna* to maksymalna wartość, jaką może przyjąć zmienna licznikowa. Kiedy uruchomimy się pętla, program wykona następujące operacje:

1. Do zmiennej *zmiennaLiczniowa* zostanie przypisana *wartośćPoczątkowa*.
2. Wartość przypisana do zmiennej *zmiennaLiczniowa* zostanie porównana z wartością *wartośćMaksymalna*. Jeżeli *zmiennaLiczniowa* jest większa niż *wartośćMaksymalna*, program opuści pętlę. W przeciwnym wypadku:
  - a) wykonane zostaną instrukcje umieszczone w ciele pętli;
  - b) *zmiennaLiczniowa* zostanie zwiększeniowa;
  - c) pętla powróci do kroku 2.

Znacznie łatwiej będzie zrozumieć pętlę *For* na przykładzie. Na listingu 5.8 przedstawiłem pseudokod, który wyświetla pięć razy komunikat „Witaj, świecie!”. Schemat blokowy na rysunku 5.14 ilustruje działanie programu.

**Listing 5.8**



```

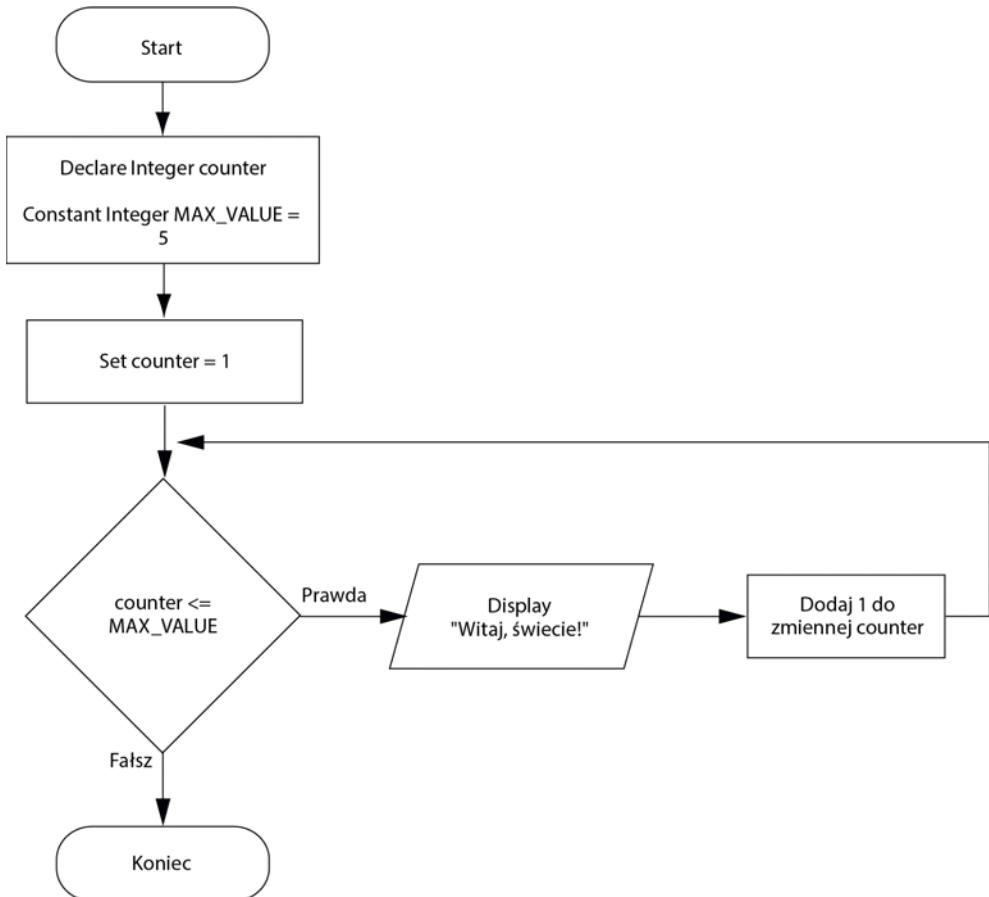
1 Declare Integer counter
2 Constant Integer MAX_VALUE = 5
3
4 For counter = 1 To MAX_VALUE
5   Display "Witaj, świecie!"
6 End For
  
```

#### **Wynik działania programu**

```

Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
  
```

W linii 1. deklaruję zmienną typu *Integer*, która będzie pełniła rolę zmiennej licznikowej. Nie musisz nadawać tej zmiennej nazwy *counter* (może to być dowolna nazwa), ale często taka nazwa wydaje się najbardziej odpowiednia. W linii 2. deklaruję stałą o nazwie *MAX\_VALUE* i zapisuję w niej maksymalną wartość licznika. Pętla *For* rozpoczyna się w linii 4. — jest to instrukcja *For counter = 1 To MAX\_VALUE*. Określam za jej pomocą, że wartość początkowa zmiennej licznikowej *counter* jest równa 1 i że pętla powinna zakończyć zliczanie, gdy licznik dojdzie do liczby 5. Aby pętla wykonała się pięć razy, na końcu każdej iteracji wartość zmiennej *counter* inkrementowana jest o 1. Podczas każdej iteracji wyświetli się komunikat „Witaj, świecie!”.



**Rysunek 5.14.** Schemat blokowy programu z listingu 5.8



**WSKAZÓWKA:** Na listingu 5.8 maksymalną wartość licznika zapisałem w stałej MAX\_VALUE. Jednak pierwszą linię pętli można także zapisać w taki sposób:

```
For counter = 1 To 5
```

Korzystanie ze stałej w tak prostym programie nie było konieczne, ale ogólnie rzecz biorąc, zapisywanie ważnych wartości w stałych jest dobrym nawykiem. Przypomnij sobie, że w rozdziale 2. wyjaśniłem, że stałe nazwane poprawiają czytelność i ułatwiają ewentualne modyfikacje programu.

Zauważ, że w ciele pętli nie ma żadnej instrukcji, która inkrementuje wartość zmiennej counter — w pętli For dzieje się to automatycznie, na końcu każdej iteracji. Dlatego ważne jest, aby nie umieszczać w ciele pętli żadnej instrukcji, która będzie modyfikowała wartość zmiennej counter. Operacja taka powoduje najczęściej nieprawidłowe działanie pętli.

## Wykorzystanie zmiennej licznikowej w ciele pętli

Głównym zadaniem zmiennej licznikowej w pętli jest zapisywanie liczby wykonanych przez pętlę iteracji. Niekiedy jednak przyda nam się ona do wykonywania obliczeń lub innych operacji w ciele pętli. Założymy, że chcesz stworzyć program, który w formie tabelki będzie wyświetlał liczby od 1 do 10 oraz drugą potęgę tych liczb, na przykład w taki sposób:

Liczba	Potęga
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Taki efekt można osiągnąć za pomocą pętli licznikowej, która wykona 10 iteracji. W pierwszej iteracji w zmiennej licznikowej będzie wartość równa 1, w drugiej iteracji — wartość 2 itd. Ponieważ podczas działania pętli zmienna licznikowa będzie przyjmowała po kolej wartości od 1 do 10, możemy jej użyć do wykonania działania.

Na rysunku 5.15 przestawiłem schemat blokowy omawianego programu. Zwróć uwagę na ciało pętli — zmienna `counter` jest w nim wykorzystywana w następującym działaniu:

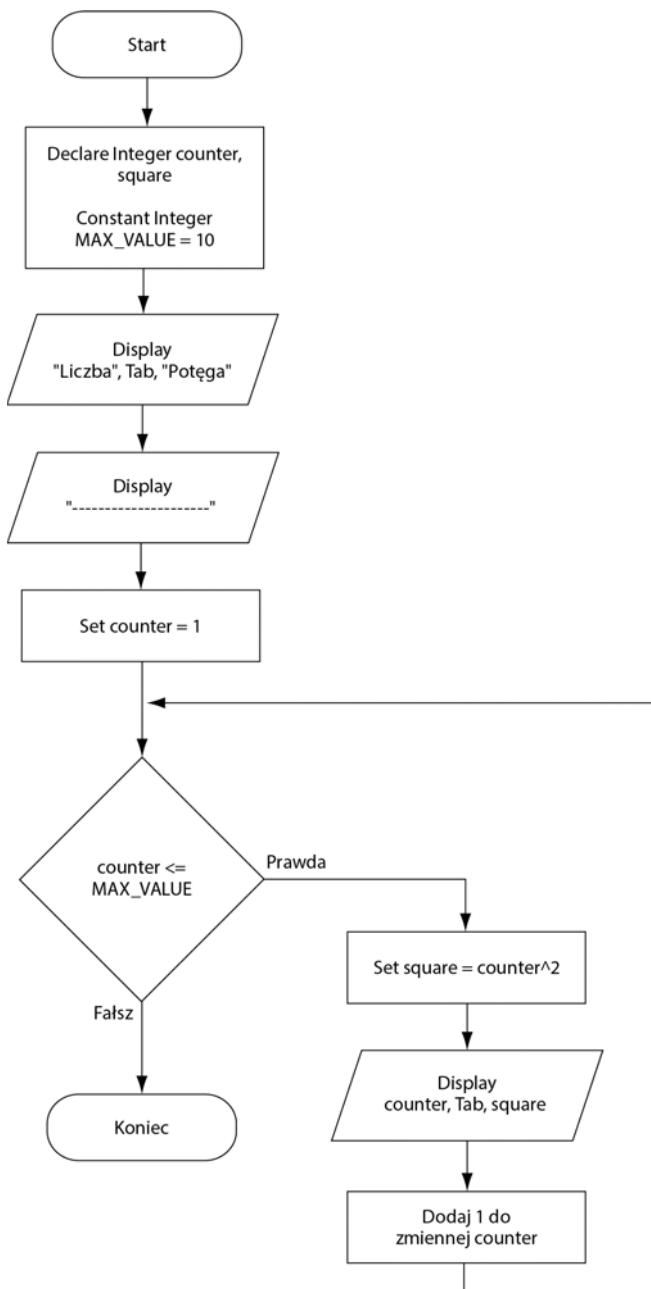
```
Set square = counter^2
```

Do zmiennej `square` przypisujemy wartość równą `counter^2`. Po wykonaniu tej operacji wyświetlamy na ekranie wartości zmiennych `counter` i `square`. Następnie do zmiennej `counter` zostanie dodana wartość 1 i pętla przejdzie do kolejnej iteracji.

Na listingu 5.9 przedstawiłem pseudokod programu. Zauważ, że w liniach 8. i 18., w poleceniu `Display`, użyłem słowa `Tab`. W ten prosty sposób zaznaczamy w pseudokodzie, że w danym miejscu należy wstawić znak tabulacji. Spójrz na linię 18.:

```
Display counter, Tab, square
```

Na ekranie wyświetli się wartość zmiennej `counter`, następnie znak tabulacji i na końcu wartość zmiennej `square`. W wyniku tego wartości na ekranie będą wyrównane w dwóch kolumnach. Ze znaków tabulacji można korzystać w wielu językach programowania.



**Rysunek 5.15.** Wyświetlanie liczb z zakresu od 1 do 10 i ich potęg

### Listing 5.9



```

1 // Zmienne
2 Declare Integer counter, square
3
4 // Stala równa maksymalnej wartości licznika
5 Constant Integer MAX_VALUE = 10
  
```

```

6
7 // Wyświetlamy nazwy kolumn
8 Display "Liczba", Tab, "Potęga"
9 Display "-----"
10
11 // Wyświetlamy liczby od 1 do 10
12 // i ich drugą potęgę
13 For counter = 1 To MAX_VALUE
14   // Obliczamy drugą potęgę liczby
15   Set square = counter^2
16
17 // Wyświetlamy liczbę i jej drugą potęgę
18 Display counter, Tab, square
19 End For

```

**Wynik działania programu**

Liczba	Potęga
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

**Inkrementowanie o wartości innie niż 1**

Liczbę, o której inkrementowana jest zmienna licznikowa w pętli For, nazywamy **krokiem**. Domyślny krok jest równy 1. W większości języków można jednak ten krok zmienić — dzięki temu mamy możliwość inkrementacji licznika o dowolną wartość.

Do zmiany kroku pętli For w pseudokodzie będziemy używali opcjonalnej klauzuli Step. Oto przykładowy pseudokod:

```

For counter = 0 To 100 Step 10
  Display counter
End For

```

W powyższej pętli ustawiam wartość początkową zmiennej counter na 0, a wartość maksymalną na 100. Klauzula Step określa krok równy 10, co oznacza, że do zmiennej counter w każdej iteracji zostanie dodana wartość 10. Podczas pierwszej iteracji zmienna counter będzie równa 0, podczas drugiej — 10, podczas trzeciej — 20 itd.

Na listingu 5.10 znajduje się kolejny przykład. Program wyświetla wszystkie liczby nieparzyste od 1 do 11.

**Listing 5.10**

```

1 // Deklaracja zmiennej licznikowej
2 Declare Integer counter
3
4 // Stała równa maksymalnej wartości licznika

```

```

5 Constant Integer MAX_VALUE = 11
6
7 // Wyświetlamy liczby nieparzyste od 1 do 11
8 For counter = 1 To MAX_VALUE Step 2
9   Display counter
10 End For

```

### Wynik działania programu

```

1
3
5
7
9
11

```

## W centrum uwagi

### Projektowanie pętli licznikowej za pomocą instrukcji For



Twoja przyjaciółka Amanda otrzymała w spadku po wujku europejski samochód sportowy. Amanda jest Amerykanką, a ponieważ prędkościomierz samochodu jest wyskalowany w km/h, obawia się, że otrzyma mandat za przekroczenie dozwolonej prędkości. Amanda poprosiła Cię o stworzenie programu, który wyświetli tabelkę prędkości zawierającą prędkość w kilometrach na godzinę i odpowiadającą jej wartość w milach na godzinę. Wzór do zamiany km/h na mph jest następujący:

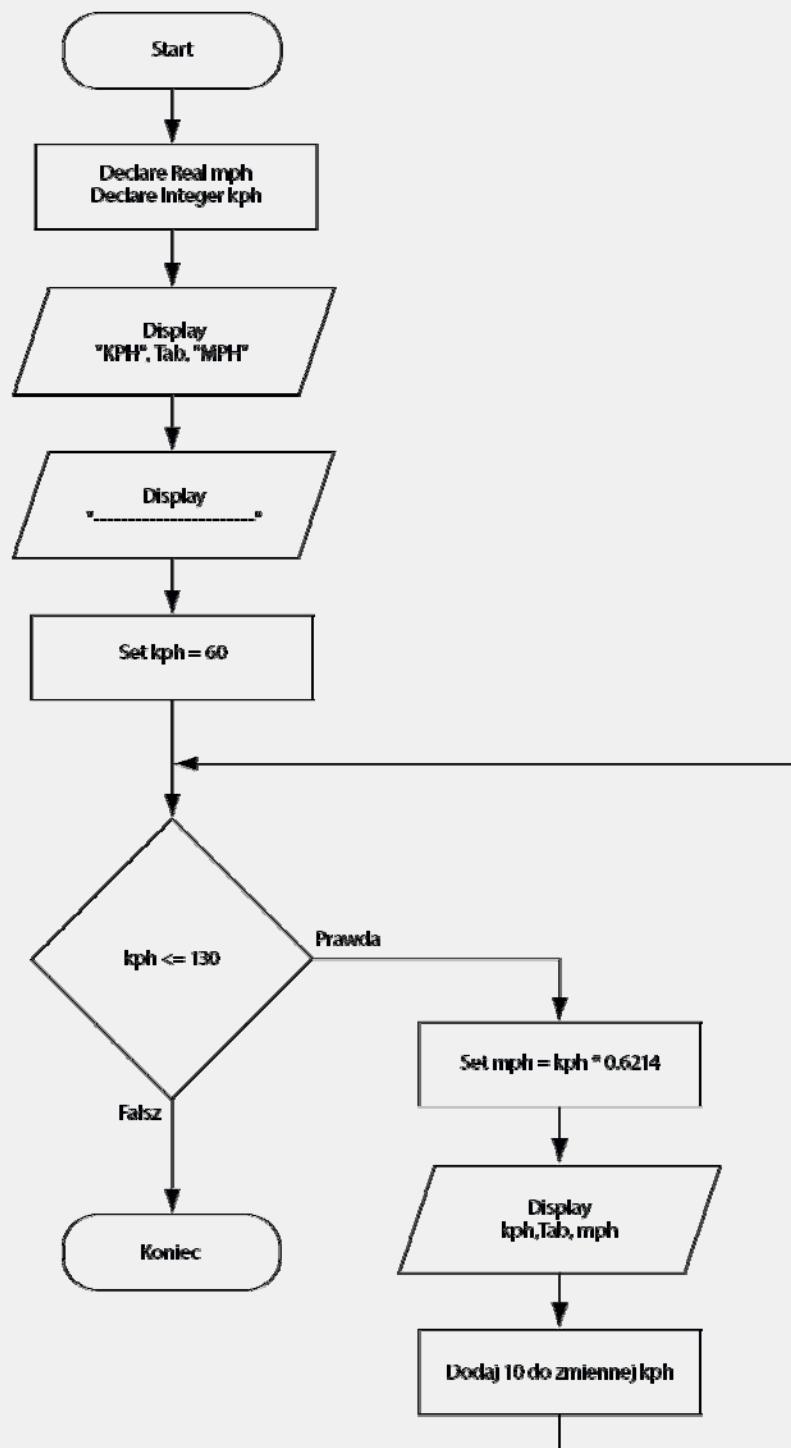
$$\text{MPH} = \text{KPH} \cdot 0,6214$$

We wzorze MPH oznacza prędkość w milach na godzinę, a KPH to prędkość w kilometrach na godzinę.

Program powinien wyświetlać prędkości, począwszy od 60 km/h do 130 km/h, w odsłonach co 10 km/h, oraz odpowiadającą im prędkość w milach na godzinę. Tabelka powinna wyglądać tak:

KPH	MPH
60	37.284
70	43.498
80	49.712
itd.	
130	80.782

Po przemyśleniu zdecydowałeś się skorzystać z pętli For, w której prędkość w km/h będzie zapisana w zmiennej licznikowej. Początkową wartością zmiennej licznikowej będzie 60, maksymalną — 130, a krok pętli będzie równy 10. Wewnątrz pętli wykorzystasz zmienną licznikową, aby obliczyć prędkość w milach na godzinę. Na listingu 5.11 przedstawiłem pseudokod programu, a na rysunku 5.16 widoczny jest jego schemat blokowy.



Rysunek 5.16. Schemat blokowy programu z listingu 5.11

**Listing 5.11**

```

1 // Deklaracja zmiennych dla km/h i mph
2 Declare Real mph
3 Declare Integer kph
4
5 // Wyświetlamy nagłówki tabelki
6 Display "KPH", Tab, "MPH"
7 Display "-----"
8
9 // Wyświetlamy prędkości
10 For kph = 60 To 130 Step 10
11    // Obliczamy prędkość w milach na godzinę
12    Set mph = kph * 0.6214
13
14    // Wyświetlamy km/h i mph
15    Display kph, Tab, mph
16 End For

```

**Wynik działania programu**

KPH	MPH
60	37.284
70	43.498
80	49.712
90	55.926
100	62.14
110	68.354
120	74.568
130	80.782

Zwróć uwagę, że tym razem zmienną licznikową nazwałem kph. Dotychczas zmienną licznikową nazywałem counter. Jednak w przypadku tego programu bardziej odpowiednia wydaje się nazwa kph, ponieważ zapisana jest w niej prędkość w kilometrach na godzinę.

## Zliczanie do tytułu poprzez dekrementację zmiennej licznikowej

Mimo że zmienna licznikowa jest zazwyczaj w pętli inkrementowana, nic nie stoi na przeszkodzie, aby ją dekrementować. **Dekrementowanie** oznacza zmniejszanie wartości. Aby dekrementować w pętli For wartość zmiennej licznikowej, użyjemy po prostu ujemnej wartości skoku. Spójrz na następujący przykład:

```

For counter = 10 To 1 Step -1
  Display counter
End For

```

W pętli tej wartość początkowa licznika jest równa 10, a wartość końcowa to 1. Krok jest równy -1, co oznacza, że w każdej iteracji od wartości licznika counter zostanie odjęta liczba 1. W pierwszej iteracji zmienna counter będzie równa 10, w drugiej iteracji — 9 itd. Gdyby powyższy kod był prawdziwym programem, w wyniku pojawiłyby się liczby 10, 9, 8 itd. — aż do 1.

## Sterowanie liczbą iteracji przez użytkownika

W wielu przypadkach to programista określa, ile iteracji wykona pętla. Przykładowo program z listingu 5.9 wyświetlał tabelkę liczb od 1 do 10 i ich drugą potęgę. Podczas pisania programu programista założył, że pętla wykona dokładnie 10 iteracji. Stała `MAX_VALUE` została więc zainicjalizowana wartością 10, a w pętli odnieśliśmy się do tej wartości:

```
For counter = 1 To MAX_VALUE
```

W wyniku tego pętla wykonała 10 iteracji. W pewnych sytuacjach programista będzie musiał jednak pozwolić, by to użytkownik programu decydował o tym, ile iteracji wykona pętla. Czyż program z listingu 5.9 nie byłby bardziej uniwersalny, gdyby to użytkownik decydował, jak dобра ma być tabelka? Na listingu 5.12 pokazalem, w jaki sposób można to zrobić w pseudokodzie.

### **Listing 5.12**

```

1 // Zmienna
2 Declare Integer counter, square, upperLimit
3
4 // Pobieramy górną próg
5 Display "Program wyświetla liczby (poczynając od 1)"
6 Display "i ich drugą potęgę. Ile liczb wyświetlić?"
7 Input upperLimit
8
9 // Wyświetlamy nazwy kolumn
10 Display "Liczba", Tab, "Potęga"
11 Display "-----"
12
13 // Wyświetlamy liczby i ich drugą potęgę
14 For counter = 1 To upperLimit
15   // Obliczamy drugą potęgę liczby
16   Set square = counter^2
17
18   // Wyświetlamy liczbę i jej drugą potęgę
19   Display counter, Tab, square
20 End For

```

### **Wynik działania programu**

Program wyświetla liczby (poczynając od 1)  
i ich drugą potęgę. Ile liczb wyświetlić?

5 [Enter]

Liczba	Potęga
1	1
2	4
3	9
4	16
5	25

W liniach 5. i 6. proszę użytkownika o wprowadzenie ilości liczb, które mają się wyświetlić w tabelce, a następnie w linii 7. zapisuję wprowadzoną wartość w zmiennej `upperLimit`. Następnie w pętli `For` odwołuję się do tej wartości jako do wartości maksymalnej licznika:

```
For counter = 1 To upperLimit
```

W wyniku tego pętla zacznie zliczanie od 1 do wartości przypisanej do zmiennej `upperLimit`. W ten sposób można także wskazać wartość początkową licznika. Na listingu 5.13 przedstawiłem przykład. W programie tym użytkownik wprowadza zarówno początkową, jak i maksymalną liczbę, która zostanie zamieszczona w tabelce. Zwróć uwagę, że w pętli For w linii 20. zmienne reprezentują zarówno wartość początkową, jak i końcową licznika.

### **Listing 5.13**

```

1 // Zmienne
2 Declare Integer counter, square,
3           lowerLimit, upperLimit
4
5 // Pobieramy dolny próg
6 Display "Program wyświetla liczby"
7 Display "i ich drugą potęgę. Od której liczby"
8 Display "mam rozpocząć?"
9 Input lowerLimit
10
11 // Pobieramy górny próg
12 Display "Na której liczbie mam skończyć?"
13 Input upperLimit
14
15 // Wyświetlamy nazwy kolumn
16 Display "Liczba", Tab, "Potęga"
17 Display "-----"
18
19 // Wyświetlamy liczby i ich drugą potęgę
20 For counter = lowerLimit To upperLimit
21   // Obliczamy drugą potęgę liczby
22   Set square = counter^2
23
24   // Wyświetlamy liczbę i jej drugą potęgę
25   Display counter, Tab, square
26 End For

```

#### **Wynik działania programu**

Program wyświetla liczby  
i ich drugą potęgę. Od której liczby  
mam rozpocząć?

3 [Enter]

Na której liczbie mam skończyć?

7 [Enter]

Liczba	Potęga
3	9
4	16
5	25
6	36
7	49

## **Projektowanie pętli licznikowej za pomocą pętli While**

W większości przypadków pętle licznikowe będziesz tworzyć za pomocą instrukcji For. W wielu językach programowania można osiągnąć taki sam efekt, korzystając z innego rodzaju pętli. Przykładowo pętlę licznikową można zaprojektować za pomocą pętli

**While, Do-While czy Do-Until.** Niezależnie od typu pętli, który wybierzesz, pętla licznikowa musi wykonywać na zmiennej licznikowej opisane wcześniej operacje: inicjalizację, porównanie wartości i inkrementację.

Aby utworzyć w pseudokodzie pętlę licznikową za pomocą pętli **While**, można się posłużyć następującym wzorcem:

- ① Declare Integer count = *wartośćPoczątkowa* ← Inicjalizujemy zmienną licznikową wartością początkową.
- ② While counter <= *wartośćMaksymalna* ← Porównujemy wartość licznika z wartością maksymalną.
  - instrukcja
  - instrukcja
  - instrukcja
  - itd.
- ③ Set counter = counter + 1 ← W każdej iteracji dodajemy do licznika 1.

Inicjalizację, porównywanie i inkrementowanie zmiennej licznikowej zaznaczylem znacznikami ①, ② i ③.

- ① to miejsce, w którym deklarujemy zmienną typu **Integer**, która będzie służyła jako zmienna licznikowa. Zmienną inicjalizujemy wartością początkową.
- ② to miejsce, w którym sprawdzamy warunek **counter <= wartośćMaksymalna**, gdzie *wartośćMaksymalna* oznacza maksymalną wartość, jaką może przyjąć zmienna licznikowa.
- ③ to miejsce, w którym do zmiennej **counter** dodajemy 1. W przypadku pętli **While** nie następuje automatyczna inkrementacja zmiennej licznikowej. Musimy jawnie umieścić w kodzie instrukcję, która wykona tę operację. Bardzo ważne jest zrozumienie, w jaki sposób działa ta instrukcja. Przyjrzyjmy się jej bliżej:

```
Set counter = counter + 1
```

Komputer wykona to polecenie w następujący sposób: najpierw obliczy wartość wyrażenia po prawej stronie operatora **=**, czyli **counter + 1**, następnie wartość tę przypisze do zmiennej **counter**. W wyniku wykonania tego polecenie do zmiennej **counter** zostanie dodane 1.



**OSTRZEŻENIE!** Jeżeli zapomnisz umieścić na końcu pętli **While** instrukcję inkrementującą wartość licznika, pętla będzie wykonywała iteracje w nieskończoność.

Na listingu 5.14 przedstawiłem przykładową pętlę licznikową z wykorzystaniem pętli **While**. Zasada działania tej pętli jest taka sama jak na listingu 5.13, jednak tym razem program wyświetli pięciokrotnie tekst **Witaj, świecie!**. Rysunek 5.17 ilustruje, w których miejscach programu ma miejsce inicjalizacja, porównywanie i inkrementacja zmiennej licznikowej.

Na listingu 5.15 przedstawiłem kolejny przykład. Program ten działa identycznie jak program z listingu 5.9: wyświetla liczby od 1 do 10 i ich drugą potęgę. Schemat blokowy tego programu jest taki sam jak ten z rysunku 5.15.

**Listing 5.14**

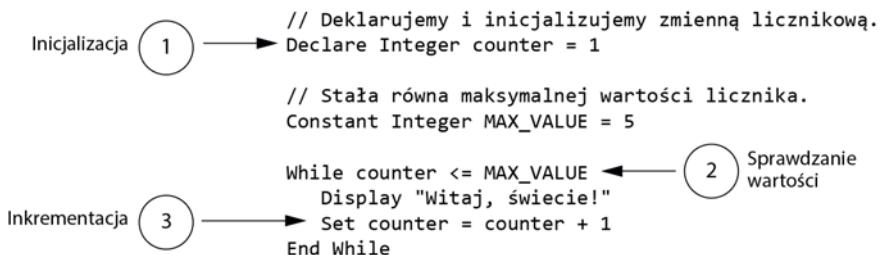
```

1 // Deklarujemy i inicjalizujemy zmienną licznikową
2 Declare Integer counter = 1
3
4 // Stała równa maksymalnej wartości licznika
5 Constant Integer MAX_VALUE = 5
6
7 While counter <= MAX_VALUE
8   Display "Witaj, Świecie!"
9   Set counter = counter + 1
10 End While

```

**Wynik działania programu**

Witaj, Świecie!  
 Witaj, Świecie!  
 Witaj, Świecie!  
 Witaj, Świecie!  
 Witaj, Świecie!

**Rysunek 5.17.** Inicjalizacja, sprawdzanie i inkrementowanie zmiennej licznikowej**Listing 5.15**

```

1 // Zmienne
2 Declare Integer counter = 1
3 Declare Integer square
4
5 // Stała równa maksymalnej wartości licznika
6 Constant Integer MAX_VALUE = 10
7
8 // Wyświetlamy nazwy kolumn
9 Display "Liczba", Tab, "Potęga"
10 Display "-----"
11
12 // Wyświetlamy liczby od 1 do 10
13 // i ich drugą potęgę
14 While counter <= MAX_VALUE
15   // Obliczamy drugą potęgę liczby
16   Set square = counter^2
17
18   // Wyświetlamy liczbę i jej drugą potęgę
19   Display counter, Tab, square
20
21   // Inkrementujemy licznik
22   Set counter = counter + 1
23 End While

```

**Wynik działania programu**

Liczba	Potęga
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

**Inkrementowanie o inne wartości niż 1**

W programach na listingach 5.14 i 5.15 zmienna licznikowa była w każdej iteracji inkrementowana o 1:

```
Set counter = counter + 1
```

Polecenie to jednak można łatwo zmodyfikować, aby licznik był inkrementowany o inne wartości niż 1. Przykładowo w następujący sposób można sprawić, że licznik będzie w każdej iteracji zwiększany o 2:

```
Set counter = counter + 2
```

Na listingu 5.16 zademonstrowałem, jak można wykorzystać to polecenie w pętli licznikowej utworzonej za pomocą instrukcji `While`.

**Listing 5.16**

```

1 // Deklarujemy zmienną licznikową
2 Declare Integer counter = 1
3
4 // Stała równa maksymalnej wartości licznika
5 Constant Integer MAX_VALUE = 11
6
7 // Wyświetlamy liczby nieparzyste
8 // od 1 do 11
9 While counter <= MAX_VALUE
10   Display counter
11   Set counter = counter + 2
12 End While

```

**Wynik działania programu**

```
1
3
5
7
9
11
```

**Zliczanie do tytułu poprzez dekrementację**

Wcześniej pokazałem, że używając ujemnej wartości kroku w pętli `For`, można sprawić, że licznik będzie dekrementowany. W przypadku pętli licznikowej z wykorzystaniem instrukcji `While` wartość licznika możemy dekrementować w następujący sposób:

```
Set counter = counter - 1
```

Polecenie to odejmuje 1 od wartości zmiennej licznikowej. Jeżeli w zmiennej counter była zapisana wartość 5, po wykonaniu tego polecenia zmieni się ona na 4. Na listingu 5.17 zademonstrowałem, w jaki sposób można wykorzystać to polecenie w pętli licznikowej utworzonej za pomocą instrukcji `While`. Program zlicza do tyłu liczby od 10 do 1.

### **Listing 5.17**

```

1 // Deklarujemy zmenną licznikową
2 Declare Integer counter = 10
3
4 // Stała równa minimalnej wartości licznika
5 Constant Integer MIN_VALUE = 1
6
7 // Wyświetlamy odliczanie
8 Display "Zaczynam odliczanie..."
9 While counter >= MIN_VALUE
10   Display counter
11   Set counter = counter - 1
12 End While
13 Display "Start!"
```

### **Wynik działania programu**

Zaczynam odliczanie...

```
10
9
8
7
6
5
4
3
2
1
```

Start!

Przyjrzyjmy się bliżej temu programowi. Zwróć uwagę na linię 11., gdzie od wartości zmiennej counter odejmuję 1. Ponieważ tym razem program zlicza do tyłu, musiałem nieco zmienić działanie pętli. Przykładowo w linii 2. inicjalizuję zmenną counter wartością 10, a nie 1 — ponieważ jest to w tym przypadku wartość początkowa licznika. Ponadto w linii 5. tworzę stałą, w której zapisuję minimalną wartość licznika (czyli 1) zamiast maksymalnej — ponieważ chcę, aby program zliczając do tyłu, zatrzymał się na liczbie 1. Ostatnia modyfikacja ma miejsce w linii 9., gdzie w warunku wykorzystałem tym razem operator relacji `>=` — pętla powinna działać dopóty, dopóki zmieniona licznikowa jest większa lub równa 1. Kiedy licznik osiągnie wartość mniejszą niż 1, pętla zakończy działanie.



### **Punkt kontrolny**

- 5.10. Co to jest zmenna licznikowa?
- 5.11. Jakie trzy operacje wykonuje zazwyczaj pętla licznikowa na zmiennej licznikowej?

5.12. Co to jest inkrementacja wartości? Co to jest dekrementacja wartości?

5.13. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer number = 5
Set number = number + 1
Display number
```

5.14. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer counter
For counter = 1 To 5
    Display counter
End For
```

5.15. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer counter
For counter = 0 To 500 Step 100
    Display counter
End For
```

5.16. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer counter = 1
Constant Integer MAX = 8
While counter <= MAX
    Display counter
    Set counter = counter + 1
End While
```

5.17. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer counter = 1
Constant Integer MAX = 7
While counter <= MAX
    Display counter
    Set counter = counter + 2
End While
```

5.18. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer counter
Constant Integer MIN = 1
For counter = 5 To MIN Step -1
    Display counter
End For
```

## 5.4

## Obliczanie sumy bieżącej

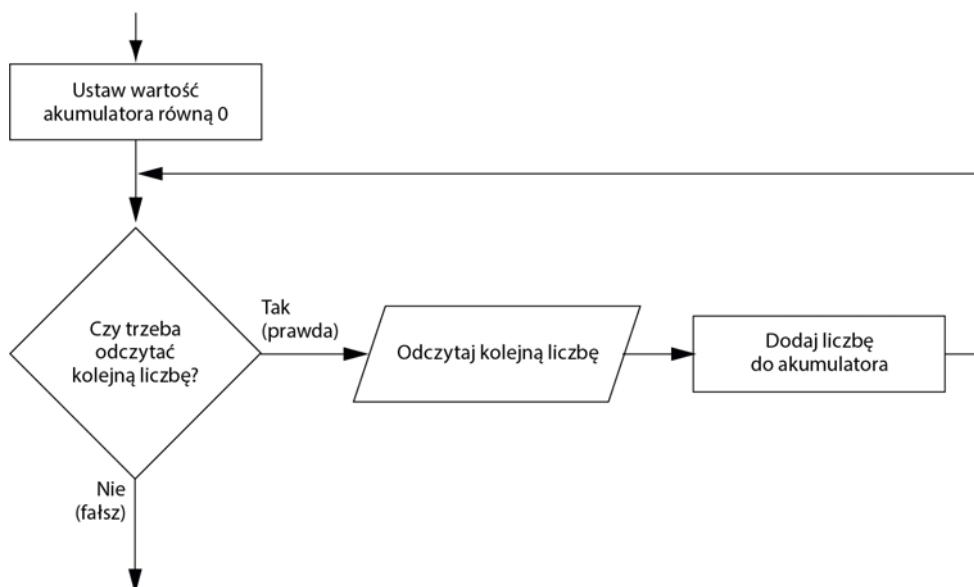
**WYJAŚNIENIE:** Suma bieżąca to suma, do której w każdej iteracji jest dodawana kolejna wartość. Zmienna, w której zapisywana jest suma bieżąca, nazywa się akumulatorem.

Sporo zadań programistycznych wiąże się z obliczaniem sumy wielu liczb. Założymy, że chcesz napisać program, który będzie obliczał wartość sprzedaży z całego tygodnia. Program taki będzie pobierał wartości sprzedaży w kolejnych dniach tygodnia i obliczał wartość sumaryczną.

Program obliczający sumę szeregu wartości wykorzystuje zazwyczaj dwa elementy:

- pętlę, za pomocą której będzie odczytywał kolejne wartości;
- zmienną, w której sumowane będą odczytane wartości.

Zmienna, w której sumowane są wartości, nazywa się **akumulatorem**. Często mówi się, że w pętli zapisywana jest **suma bieżąca** — ponieważ z iteracji na iterację ulega ona zmianie. Na rysunku 5.18 przedstawiłem zasadę działania pętli, która oblicza sumę bieżącą.



Rysunek 5.18. Ogólna zasada obliczania sumy bieżącej

Kiedy pętla zakończy działanie, w akumulatorze znajdzie się suma wszystkich liczb odczytanych w pętli. Zwrót uwagę, że pierwszym krokiem na schemacie blokowym jest ustalenie w akumulatorze wartościowej 0. To bardzo ważny krok. Po każdym odczytaniu liczby zostanie ona dodana do akumulatora. Jeżeli w akumulatorze znajdzie się na samym początku wartość inna niż 0, błędna będzie też suma wartości obliczona w pętli.

Spójrzmy teraz, jak wygląda program do obliczania sumy bieżącej. Program z listingu 5.18 prosi użytkownika o wprowadzenie pięciu liczb, a następnie wyświetla ich sumę.

**Listing 5.18**

```

1 // Deklarujemy zmienną, w której będziemy zapisywać
2 // liczbę wprowadzoną przez użytkownika
3 Declare Integer number
4
5 // Deklarujemy zmienną akumulatora
6 // i inicjalizujemy ją wartością 0
7 Declare Integer total = 0
8
9 // Deklarujemy zmienną licznikową
10 Declare Integer counter
11
12 // Wyjaśniamy, do czego służy program
13 Display "Program oblicza"
14 Display "sumę pięciu liczb."
15
16 // Pobieramy pięć liczb i je sumujemy
17 For counter = 1 To 5
18     Display "Wprowadź liczbę."
19     Input number
20     Set total = total + number
21 End For
22
23 // Wyświetlamy sumę liczb
24 Display "Suma jest równa ", total

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

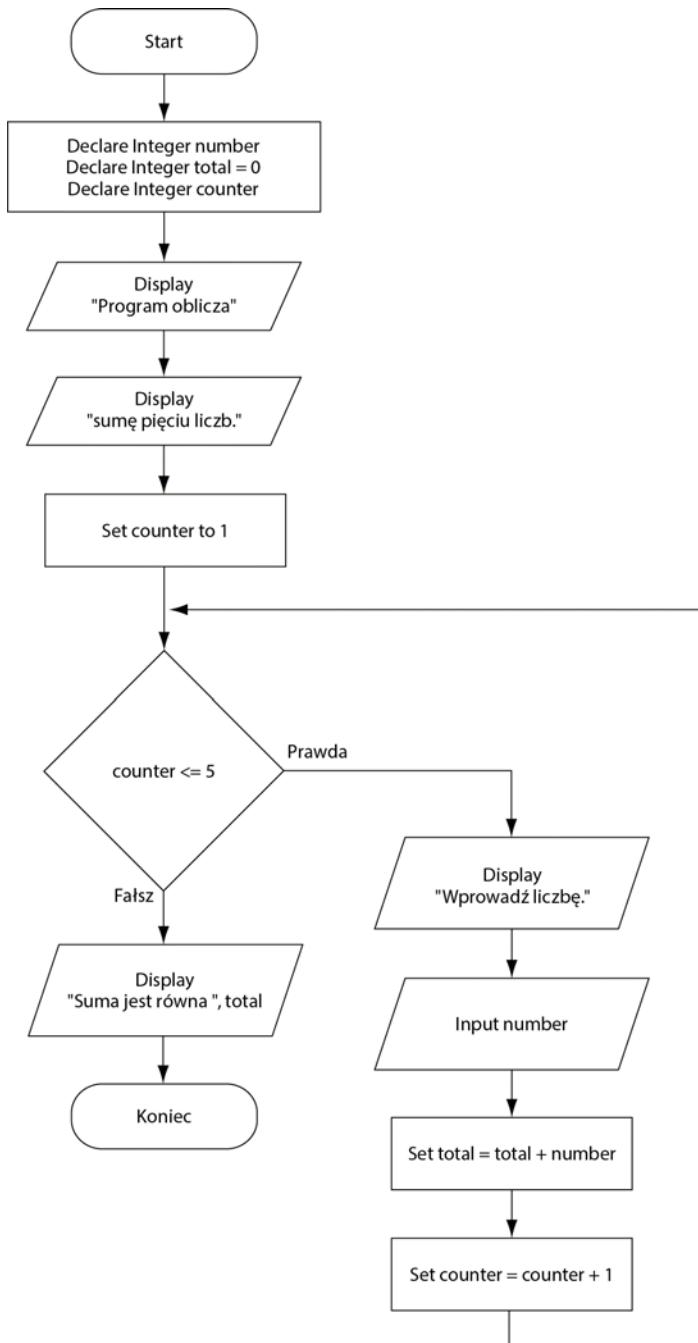
Program oblicza  
sumę pięciu liczb.  
Wprowadź liczbę.  
**2 [Enter]**  
Wprowadź liczbę.  
**4 [Enter]**  
Wprowadź liczbę.  
**6 [Enter]**  
Wprowadź liczbę.  
**8 [Enter]**  
Wprowadź liczbę.  
**10 [Enter]**  
Suma jest równa 30

Spójrz na deklaracje zmiennych. W linii 3. deklaruję zmienną `number`, w której będę zapisywać liczbę wprowadzoną przez użytkownika. W linii 7. deklaruję zmienną `total`, która jest akumulatorem — zauważ, że inicjalizuję ją wartością równą 0. W linii 10. deklaruję zmienną `counter`, która jest zmienną licznikową.

W liniach od 17. do 21. pojawia się pętla `For`, której zadaniem jest pobranie danych i obliczenie sumy. W linii 18. proszę użytkownika o wprowadzenie liczby, a w linii 19. zapisuję wprowadzoną liczbę w zmiennej `number`. Następnie, w linii 20., obliczam sumę w następujący sposób:

`Set total = total + number`

Po wykonaniu tego polecenia do wartości zapisanej w zmiennej `total` zostanie dodana liczba zapisana w zmiennej `number`. Po wyjściu z pętli w zmiennej `total` będzie się znajdowała suma wszystkich liczb. Wartość tę wyświetlам w linii 24. Na rysunku 5.19 widoczny jest schemat blokowy programu z listingu 5.18.



**Rysunek 5.19.** Schemat blokowy programu z listingu 5.18



## Punkt kontrolny

- 5.19. Z jakich elementów składa się zazwyczaj program obliczający sumę szeregu wartości?
- 5.20. Co to jest akumulator?
- 5.21. Czy akumulator trzeba inicjalizować jakąś wartością? Wyjaśnij, dlaczego trzeba lub nie trzeba tego robić.
- 5.22. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer number1 = 10, number2 = 5
Set number1 = number1 + number2
Display number1
Display number2
```

- 5.23. Spójrz na poniższy pseudokod. Co wyświetliłoby się na ekranie, gdyby ten pseudokod był prawdziwym programem?

```
Declare Integer counter, total = 0
For counter = 1 To 5
    Set total = total + counter
End For
Display total
```

### 5.5

## Wartownik

**WYJAŚNIENIE:** Wartownik to specjalna wartość, która wskazuje koniec listy wartości.

Projektujesz program, który będzie przetwarzał w pętli długą listę wartości. W momencie projektowania programu nie jesteś w stanie wskazać, jak długa będzie ta lista. Tak naprawdę może ona zawierać różną liczbę wartości przy każdym uruchomieniu programu. W jaki sposób zaprojektować pętlę w takiej sytuacji? Oto techniki, które poznaleś w tym rozdziale, i wady, jakie wykazują w przypadku przetwarzania długiej listy wartości:

- Na koniec każdej iteracji zapytamy użytkownika, czy chce pobrać kolejną wartość z listy. Jednak gdy lista będzie dłuża, taki sposób będzie dla użytkownika bardzo uciążliwy.
- Na samym początku programu zapytamy użytkownika, ile wartości zawiera lista. To rozwiązanie także może być nieporęczne dla użytkownika — jeżeli lista jest dłuża, a użytkownik nie zna dokładnej liczby wartości, będzie on je musiał policzyć.

W przypadku przetwarzania w pętli długiej listy wartości znacznie lepszą techniką jest skorzystanie z wartownika. **Wartownik** (ang. sentinel) to specjalna wartość, która wskazuje koniec listy elementów. Kiedy program odczyta tę wartość, będzie wiedział, że dotarł do końca listy i powinien wyjść z pętli. Założmy, że pewien lekarz potrzebuje programu, za pomocą którego będzie obliczał średnią wagę swoich pacjentów.

Program taki mógłby wyglądać następująco: pętla prosi użytkownika o wprowadzenie wagi kolejnego pacjenta lub o wprowadzenie wartości 0, kiedy program ma wyjść z pętli. Gdy program odczyta wartość równą 0, będzie wiedział, że nie ma już kolejnych wartości do odczytania. Pętla się zakończy i program wyświetli średnią wagę.

Wartość wartownika musi więc być na tyle unikatowa, aby nie mogła być pomyłona z innymi wartościami wprowadzonymi przez użytkownika. W powyższym przykładzie, aby zasygnalizować koniec szeregu wartości, lekarz wprowadza 0. W tym przypadku liczba ta sprawdzi się jako wartownik, ponieważ żaden pacjent nie będzie miał wagi równej 0.

## **W centrum uwagi**

### Korzystanie z wartownika

Urząd skarbowy oblicza wartość podatku od nieruchomości na postawie następującego wzoru:

$$\text{podatek od nieruchomości} = \text{wartość nieruchomości} \cdot 0,0065$$

Urzędnik otrzymuje każdego dnia listę nieruchomości i musi dla każdej z nich obliczyć podatek. Poproszono Cię o zaprojektowanie programu, z którego będzie mógł korzystać urzędnik, aby obliczać wartość podatku.

Podczas rozmów z urzędnikiem dowiedziałeś się, że do każdej nieruchomości przypisany jest numer działki oraz że numer ten jest zawsze większy lub równy 1. Postanowileś więc, że skorzystasz z pętli, w której jako wartownika użyjesz liczby 0. W każdej iteracji program będzie prosił urzędnika o wprowadzenie kolejnego numeru działki lub wprowadzenie 0, aby zakończyć program. Na listingu 5.19 przedstawiony jest pseudokod programu, a na rysunku 5.20 jego schemat blokowy.

#### **Listing 5.19**

```

1 Module main()
2   // Zmienna lokalna z numerem działki
3   Declare Integer lotNumber
4
5   // Pobieramy pierwszy numer działki
6   Display "Wprowadź numer działki"
7   Display "(lub 0, jeśli chcesz zakończyć)."
8   Input lotNumber
9
10  // Przetwarzamy dane, dopóki
11  // użytkownik nie wprowadzi liczby 0
12  While lotNumber != 0
13    // Wyświetlamy wartość podatku od nieruchomości
14    Call showTax()
15
16    // Pobieramy kolejny numer działki
17    Display "Wprowadź kolejny numer działki"
18    Display "(lub 0, jeśli chcesz zakończyć)."
19    Input lotNumber

```



```

20    End While
21 End Module
22
23 // Moduł showTax pobiera wartość nieruchomości
24 // i wyświetla wartość podatku
25 Module showTax()
26   // Zmienne lokalne
27   Declare Real PropertyValue, tax
28
29   // Stała z wysokością podatku
30   Constant Real TAX_FACTOR = 0.0065
31
32   // Pobieramy wartość nieruchomości
33   Display "Wprowadź wartość nieruchomości."
34   Input PropertyValue
35
36   // Obliczamy podatek
37   Set tax = PropertyValue * TAX_FACTOR
38
39   // Wyświetlamy podatek
40   Display "Podatek od nieruchomości wynosi ", tax, " zł."
41 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź numer działki  
(lub 0, jeśli chcesz zakończyć).

**417 [Enter]**

Wprowadź wartość nieruchomości.

**100000 [Enter]**

Podatek od nieruchomości wynosi 650 zł.

Wprowadź kolejny numer działki  
(lub 0, jeśli chcesz zakończyć).

**692 [Enter]**

Wprowadź wartość nieruchomości.

**60000 [Enter]**

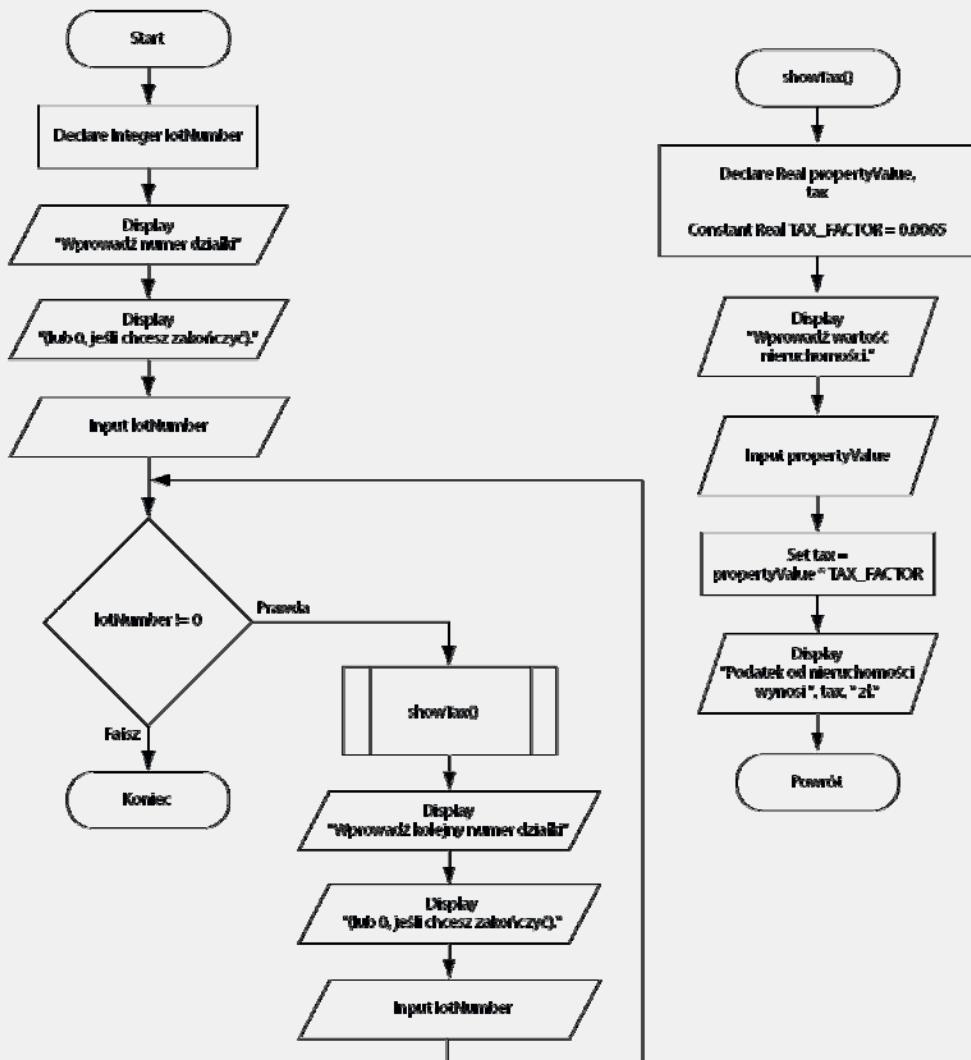
Podatek od nieruchomości wynosi 390 zł.

Wprowadź kolejny numer działki  
(lub 0, jeśli chcesz zakończyć).

**0 [Enter]**

**Punkt kontrolny**

- 5.24. Co to jest wartownik?
- 5.25. Dlaczego należy zachować ostrożność, dobierając unikatową wartość dla wartownika?



Rysunek 5.20. Schemat blokowy programu z listingu 5.19

## 5.6

## Pętle zagnieżdżone

**WYJAŚNIENIE:** Pętlę umieszczoną wewnętrz w innej pętli nazywamy pętlą zagnieżdżoną.

Pętla zagnieżdzona to pętla umieszczona wewnętrz innej pętli. Dobrym przykładem czegoś, co działa jak pętla zagnieżdzona, jest zegar. Wskazówki sekund, minut i godzin kręcą się na tarczy zegara. Wskazówka godzinowa wykonuje jednak jeden obrót na 12 obrotów wskazówka minutowej. Wskazówka sekundowa wykonuje natomiast 60

obrotów na jeden obrót wskazówki minutowej. Oznacza to, że w czasie, gdy wskazówka godzinowa wykona jeden pełny obrót, wskazówka sekundowa wykona 720 obrotów. Oto przykład pseudokodu symulującego działanie zegara cyfrowego. Na ekranie będą się wyświetlały sekundy — od 0 do 59:

```
Declare Integer seconds
For seconds = 0 To 59
    Display seconds
End For
```

Możemy teraz dodać zmienną `minutes` i zagnieździć w powyższej pętli jeszcze jedną pętlę, która będzie zliczała 60 minut:

```
Declare Integer minutes, seconds
For minutes = 0 To 59
    For seconds = 0 To 59
        Display minutes, ":", seconds
    End For
End For
```

Aby zegar był kompletny, dodamy jeszcze jedną zmienną, która będzie zliczała godziny:

```
Declare Integer hours, minutes, seconds
For hours = 0 To 23
    For minutes = 0 To 59
        For seconds = 0 To 59
            Display hours, ":", minutes, ":", seconds
        End For
    End For
End For
```

Gdyby to był prawdziwy program, w wyniku otrzymalibyśmy coś takiego:

```
0:0:0
0:0:1
0:0:2
```

Program zliczałby każdą sekundę, aż doszedłby do całych 24 godzin:

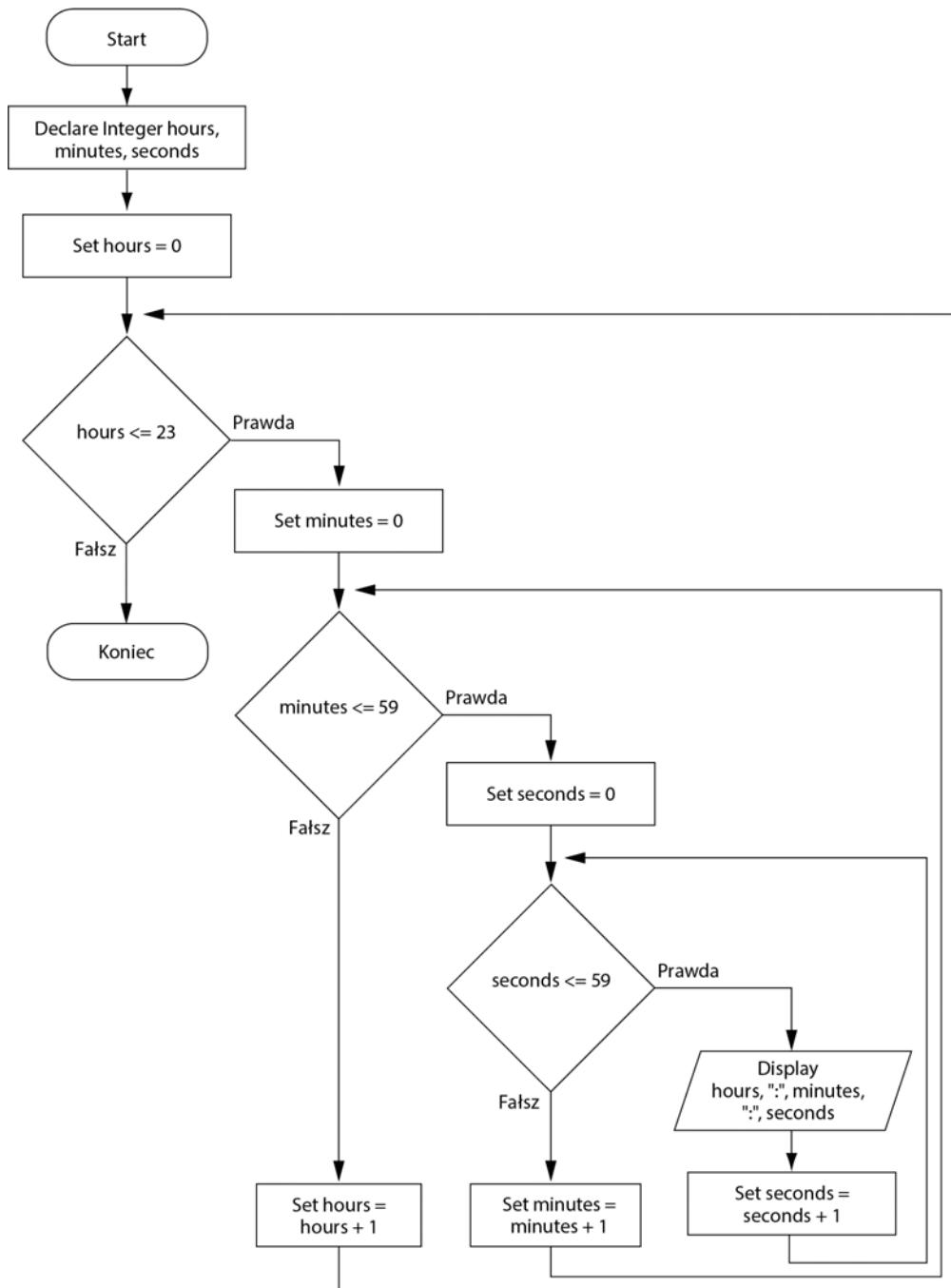
23:59:59

Pętla znajdująca się najbardziej wewnątrz struktury programu wykona 60 iteracji na każdą iterację pętli środkowej. Pętla środkowa wykona 60 iteracji na każdą iterację pętli znajdującej się najbardziej na zewnątrz struktury programu. Zewnętrzna pętla wykona 24 iteracje, pętla środkowa 1440 iteracji, a pętla wewnętrzna aż 86400 iteracji! Na rysunku 5.21 przedstawiony jest schemat blokowy programu symulującego zegar.

Na przykładzie programu symulującego zegar można zauważyc kilka rzeczy:

- pętla wewnętrzna przechodzi przez wszystkie iteracje dla każdej jednej iteracji pętli zewnętrznej;
- pętla wewnętrzna wykonuje iteracje częściej niż pętla zewnętrzna;
- aby obliczyć sumaryczną liczbę iteracji pętli zagnieżdzonej, należy pomnożyć liczbę iteracji w każdej pętli.

Na listingu 5.20 przedstawiłem kolejny przykład. Z programu tego może skorzystać nauczyciel, aby obliczyć średni wynik z kilku sprawdzianów dla każdego ucznia.



**Rysunek 5.21.** Schemat blokowy symulacji zegara

W linii 13. pobieramy liczbę uczniów, a w linii 17. pobieramy liczbę sprawdzianów. W linii 20. rozpoczyna się pętla For, która wykona tyle iteracji, ilu jest uczniów. Kolejna, zagnieżdzona pętla znajduje się w liniach od 27. do 31. i wykona tyle iteracji, ile było sprawdzianów.

**Listing 5.20**

```

1 // Program oblicza średni wynik z kilku sprawdzianów; program prosi o wprowadzenie
2 // liczbę uczniów i liczby sprawdzianów
3 Declare Integer numStudents
4 Declare Integer numTestScores
5 Declare Integer total
6 Declare Integer student
7 Declare Integer testNum
8 Declare Real score
9 Declare Real average
10
11 // Pobieramy liczbę uczniów
12 Display "Ilu jest uczniów?"
13 Input numStudents
14
15 // Pobieramy liczbę sprawdzianów
16 Display "Ile sprawdzianów napisał każdy z uczniów?"
17 Input numTestScores
18
19 // Obliczamy średni wynik dla każdego ucznia
20 For student = 1 To numStudents
21     // Inicjalizujemy akumulator, w którym zapiszemy sumę wyników
22     Set total = 0
23
24     // Pobieramy wyniki sprawdzianów dla danego ucznia
25     Display "Uczeń numer ", student
26     Display "-----"
27     For testNum = 1 To numTestScores
28         Display "Wprowadź wynik ze sprawdzianu numer ", testNum, ":"
29         Input score
30         Set total = total + score
31     End For
32
33     // Obliczamy średni wynik dla danego ucznia
34     Set average = total / numTestScores
35
36     // Wyświetlamy średni wynik
37     Display "Średni wynik ucznia ", student, " jest równy ", average
38     Display
39 End For

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Ilu jest uczniów?  
**3 [Enter]**  
 Ile sprawdzianów napisał każdy z uczniów?  
**3 [Enter]**  
 Uczeń numer 1  
 -----  
 Wprowadź wynik ze sprawdzianu numer 1:  
**100 [Enter]**  
 Wprowadź wynik ze sprawdzianu numer 2:  
**95 [Enter]**  
 Wprowadź wynik ze sprawdzianu numer 3:  
**90 [Enter]**  
 Średni wynik ucznia 1 jest równy 95.0  
 Uczeń numer 2  
 -----  
 Wprowadź wynik ze sprawdzianu numer 1:  
**80 [Enter]**  
 Wprowadź wynik ze sprawdzianu numer 2:

```

81 [Enter]
Wprowadź wynik ze sprawdzianu numer 3:
82 [Enter]
Średni wynik ucznia 2 jest równy 81.0
Uczeń numer 3
-----
Wprowadź wynik ze sprawdzianu numer 1:
75 [Enter]
Wprowadź wynik ze sprawdzianu numer 2:
85 [Enter]
Wprowadź wynik ze sprawdzianu numer 3:
80 [Enter]
Średni wynik ucznia 3 jest równy 80.0

```

## 5.7

# Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ wielu zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

## Java

### Struktury powtórzeń w języku Java

#### Inkrementacja i dekrementacja zmiennych w języku Java

Inkrementacja zmiennej jest operacją zwiększenia jej wartości, a dekrementacja zmiennej jest operacją zmniejszenia jej wartości. Inkrementacja i dekrementacja są tak często wykonywane w programach, że w Javie dostępny jest zestaw prostych jednoargumentowych operatorów wykonujących te operacje. Operatorem inkrementacji jest symbol `++`, a operatorem dekrementacji jest symbol `--`. Poniższa instrukcja używa operatora `++`, aby dodać 1 do zmiennej `num`:

```
num++;
```

Po wykonaniu tej instrukcji wartość w zmiennej `num` zostanie zwiększona o 1. Ta instrukcja używa operatora `--`, aby odjąć 1 od zmiennej `num`:

```
num--;
```

W tych przykładach umieściliśmy operatory `++` i `-` po ich operandach (czyli po prawej stronie operandów). Nazywa się to **trybem postfiksowym**. Operatory mogą być również zapisane przed operandami (czyli po lewej stronie), co nazywa się **trybem prefiksowym**. Oto przykłady:

```
++num;  
--num;
```

Kiedy piszesz prostą instrukcję inkrementacji lub dekrementacji zmiennej, na przykład taką jak pokazana powyżej, nie ma znaczenia, czy używasz trybu prefiksowego czy postfiksowego — operatory robią to samo w obu trybach. Różnica w sposobie działania tych dwóch trybów nastąpi, jeśli wypiszesz instrukcje, które połączą te operatory z innymi operatorami lub z innymi operacjami. Tak złożony kod może być trudny do zrozumienia i debugowania. W tej książce używam operatorów inkrementacji i dekrementacji tylko w sposób prosty i łatwy do zrozumienia, jak w przedstawionych wcześniej przykładach zastosowania. Wprowadzam te operatory w tym momencie, ponieważ są one powszechnie używane w niektórych typach pętli.

### Pętla while w języku Java

W Javie pętla `while` jest zapisywana w następującym ogólnym formacie:

```
while (WyrażenieLogiczne)
{
    instrukcja;
    instrukcja;
    itd;
}
```

Podczas wykonywania pętli `while` sprawdzana jest wartość wyrażenia logicznego. Jeśli wyrażenie to przyjmuje wartość `true`, to zostaną wykonane instrukcje zawarte w nawiasach klamrowych, a następnie pętla rozpoczęcie kolejny obieg. Jeśli wyrażenie logiczne przyjmie wartość `false`, to pętla zakończy się i program wznowi wykonywanie instrukcji znajdującej się bezpośrednio po niej.

Jeśli piszesz pętlę `while` zawierającą w ciele tylko jedną instrukcję, to nie musisz umieszczać jej w nawiasach klamrowych. Taką pętlę można zapisać w następującym ogólnym formacie:

```
while (WyrażenieLogiczne)
    instrukcja;
```

Kiedy wykonywana jest pętla `while` napisana w tym formacie, to sprawdzana jest wartość wyrażenia logicznego. Jeśli wyrażenie ma wartość `true`, to zostanie wykonana jedna instrukcja, która znajduje się w następnym wierszu, a potem pętla rozpoczęcie kolejny obieg. Jeśli jednak wyrażenie logiczne ma wartość `false`, to pętla się zakończy.

Oto przykład pętli `while`, która pięć razy wyświetla słowo „Cześć”:

```
int count = 0;
while (count < 5)
{
    System.out.println("Cześć");
    count++;
}
```

### Pętla do-while w języku Java

Oto ogólny format pętli `do-while`:

```
do
{
    instrukcja;
    instrukcja;
    itd;
} while (WyrażenieLogiczne);
```

Podobnie jak w pętli `while`, jeśli w ciele pętli znajduje się tylko jedna instrukcja, nawiasy klamrowe są opcjonalne. Oto ogólny format pętli `do-while` z tylko jedną warunkowo wykonaną instrukcją:

```
do
    instrukcja;
    while (WyrażenieLogiczne);
```

Zauważ, że na samym końcu instrukcji `do-while` pojawia się średnik. Jest on tutaj niezbędny, a pominięcie go stanowi częsty błąd.

Oto przykład pętli `do-while`, która pięć razy wypisuje słowo „Cześć”:

```
int count = 0;
do
{
    System.out.println("Cześć");
    count++;
} while (count < 5);
```



**UWAGA:** W języku Java nie ma pętli działającej tak jak pętla `Do-Until`, która była omawiana we wcześniejszej części tego rozdziału.

## Pętla `for` w języku Java

Konstrukcja pętli `for` składa się z części inicjalizowania, testowania i inkrementowania licznika zmiennej. Oto ogólny format pętli `for`:

```
for (WyrażenieInicjalizujące; WyrażenieTestujące; WyrażenieInkrementujące)
{
    instrukcja;
    instrukcja;
    itd.
}
```

Instrukcje wyświetlane w nawiasach klamrowych są wykonywane przy każdym obiegu pętli. Jeśli treść pętli zawiera tylko jedną instrukcję, to nawiasy klamrowe są opcjonalne, jak w poniższym ogólnym przykładzie:

```
for (WyrażenieInicjalizujące; WyrażenieTestujące; WyrażenieInkrementujące)
    instrukcja;
```

Pierwszy wiersz pętli `for` jest tak zwanym **nagłówkiem pętli**. Po słowie kluczowym `for` w nawiasie znajdują się trzy wyrażenia oddzielone od siebie średnikami. (Zauważ, że po trzecim wyrażeniu nie ma średnika).

Pierwsze wyrażenie jest **wyrażeniem inicjalizującym**. Zwykle używa się go do zapisania wartości początkowej w zmiennej licznika. Jest to pierwsza operacja wykonywana przez pętlę i jest ona wykonywana tylko raz. Drugie wyrażenie to **wyrażenie**

**testujące.** Jest to wyrażenie logiczne, które steruje wykonywaniem pętli. Dopóki to wyrażenie jest prawdziwe, będzie powtarzana cała zawartość pętli `for`. Pętla `for` jest pętlą wstępna testującą, więc ocenia wyrażenie testowe przed każdą iteracją. Trzecie wyrażenie to **wyrażenie inkrementujące**, które jest wykonywane na końcu każdej iteracji. Zazwyczaj jest to instrukcja, która inkrementuje zmienną licznika pętli.

Oto przykład prostej pętli `for`, która pięć razy wypisuje słowo „Cześć”:

```
for (count = 1; count <= 5; count++)
{
    System.out.println("Cześć");
}
```

W tej pętli wyrażeniem inicjalizującym jest `count = 1`, wyrażeniem testującym jest `count <= 5`, natomiast wyrażenie inkrementujące to `count++`. Ciało pętli składa się z jednej instrukcji, która wywołuje funkcję `System.out.println`. Oto podsumowanie tego, co się dzieje w trakcie wykonywania tej pętli:

1. Wykonywane jest wyrażenie inicjalizujące — `count = 1`. Przypisuje ono wartość 1 do zmiennej `count`.
2. Następnie sprawdzane jest wyrażenie `count <= 5`. Jeśli jest prawdziwe, to przechodzimy do kroku 3. W przeciwnym wypadku pętla zostanie zakończona.
3. Wykonywana jest instrukcja `System.out.println("Cześć");`.
4. Następnie wykonywane jest wyrażenie inkrementujące — `count++`. Dodaje ono 1 do zmiennej `count`.
5. Następuje powrót do kroku 2.

## Python

### Struktury powtórzeń w języku Python

#### Pętla `while` w języku Python

W Pythonie pętla `while` jest zapisana w następującym ogólnym formacie:

```
while warunek:
    instrukcja
    instrukcja
    itd.
```

Klauzula `while` rozpoczyna się od słowa `while`, po którym następuje `warunek` typu Boolean, który musi zwracać wartość prawdy lub fałszu. Po `warunku` pojawia się dwukropki. W kolejnym wierszu rozpoczyna się blok instrukcji. Wszystkie instrukcje w bloku muszą mieć jednakowe wcięcia. Te wcięcia są wymagane, ponieważ interpreter Pythona na ich podstawie ustala, gdzie zaczyna się i kończy blok instrukcji.

Podczas wykonywania pętli `while` sprawdzana jest wartość `warunku`. Jeśli `warunek` jest spełniony, to wykonywane są instrukcje znajdujące się w bloku instrukcji następującym po klauzuli `while`, a potem pętla rozpoczyna kolejny obieg. Jeśli `warunek` jest fałszywy, to program wychodzi z pętli. Oto przykład pętli `while`, która pięć razy wypisuje słowo „Cześć”:

```

count = 0
while count < 5:
    print('Cześć')
    count = count + 1

```



**UWAGA:** Python nie ma pętli działającej tak jak Do-While lub Do-Until, które zostały omówione we wcześniejszej części tego rozdziału.

### Pętla for w języku Python

W Pythonie instrukcja `for` jest przygotowana do pracy z sekwencjami elementów danych. Po wykonaniu tej instrukcji iteruje ona po wszystkich elementach w sekwencji. Instrukcji `for` używa się w następującym ogólnym formacie:

```

for zmienna in [wartość1, wartość2, itd.]:
    instrukcja
    instrukcja
    itd.

```

W klauzuli `for zmienna` jest nazwa zmiennej, natomiast w nawiasach kwadratowych pojawia się ciąg wartości rozdzielanych przecinkami. (W Pythonie rozzielona przecinkami sekwencja elementów danych, które są zawarte w zestawie nawiasów, jest nazywana *listą*). W następnym wierszu rozpoczyna się blok instrukcji, które wykonywane są przy każdej iteracji pętli.

Instrukcja `for` wykonywana jest w następujący sposób: *zmiennej* przypisywana jest pierwsza wartość z listy, a następnie wykonywane są instrukcje znajdujące się w bloku, w kolejnym kroku *zmiennej* przypisywana jest następna wartość z listy i jeszcze raz wykonywane są instrukcje z bloku. Pętla jest powtarzana dopóty, dopóki *zmiennej* nie zostanie przypisana ostatnia wartość z listy. Oto przykład, w którym pętla `for` wyświetla liczby od 1 do 5:

```

for num in [1, 2, 3, 4, 5]:
    print(num)

```

### Wykorzystywanie funkcji range wraz z pętlą for w języku Python

Python udostępnia wbudowaną funkcję o nazwie `range`, która upraszcza proces zapisu pętli `for` sterowanej sekwencją liczb. Oto przykład pętli `for` wykorzystującej funkcję `range`:

```

for num in range(5):
    print(num)

```

Zauważ, że zamiast używać listy wartości wywołujemy funkcję `range`, przekazując jej wartość 5 jako argument. W tym programie funkcja `range` wygeneruje listę liczb całkowitych od 0 do 5 (włącznie). Program działa tak jak przedstawiony poniżej przykład:

```

for num in [0, 1, 2, 3, 4]:
    print(num)

```

Jak można zauważyc, lista zawiera pięć liczb, więc pętla będzie się powtarzać pięć razy. Poniższy kod wykorzystuje funkcję `range` wraz z pętlą `for` do wyświetlenia pięciu słów `Witaj, świecie!`:

```
for x in range(5):
    print('Witaj, świecie!')
```

Jeśli przekażesz funkcji `range` jeden argument, tak jak pokazałem wcześniej, to zostanie on użyty jako wartość ograniczająca listę. Jeśli przekażesz funkcji `range` dwa argumenty, to pierwszy argument zostanie użyty jako wartość początkowa listy, natomiast drugi jako limit końcowy. Oto przykład:

```
for num in range(1, 5):
    print(num)
```

Ten kod wyświetli:

```
1
2
3
4
```

Domyślnie funkcja `range` tworzy listę liczb, które dla każdej kolejnej liczby na liście zwiększą się o 1. Jeśli przekażesz funkcji `range` trzeci argument, to zostanie on użyty jako **wartość kroku**. Zamiast o 1 każda kolejna liczba na liście będzie się zwiększać o wartość tego kroku:

```
for num in range(1, 12, 2):
    print(num)
```

Ten kod wyświetli:

```
1
3
5
7
9
11
```

## C++

### Struktury powtórzeń w języku C++

#### Inkrementacja i dekrementacja zmiennych w języku C++

Inkrementacja zmiennej jest operacją zwiększenia jej wartości, a dekrementacja zmiennej jest operacją zmniejszenia jej wartości. Inkrementacja i dekrementacja są tak często wykonywane w programach, że w C++ dostępny jest zestaw prostych jednoargumentowych operatorów wykonujących te operacje. Operatorem inkrementacji jest symbol `++`, a operatorem dekrementacji jest symbol `--`. Poniższa instrukcja używa operatora `++`, aby dodać 1 do zmiennej `num`:

```
num++;
```

Po wykonaniu tej instrukcji wartość zmiennej `num` zostanie zwiększona o 1. Poniższa instrukcja używa operatora `--`, aby odjąć 1 od zmiennej `num`:

```
num--;
```

W tych przykładach umieściliśmy operatory ++ i - po ich operandach (czyli po prawej stronie operandów). Nazywa się to **trybem postfiksowym**. Operatory mogą być również zapisane przed operandami (czyli po lewej stronie), co nazywa się **trybem prefiksowym**. Oto przykłady:

```
++num;  
--num;
```

Kiedy piszesz prostą instrukcję inkrementacji lub dekrementacji zmiennej, na przykład taką jak pokazana powyżej, nie ma znaczenia, czy używasz trybu prefiksowego czy postfiksowego — operatory robią to samo w obu trybach. Różnica w sposobie działania tych dwóch trybów nastąpi, jeśli wypiszesz instrukcje, które połączą te operatory z innymi operatorami lub z innymi operacjami. Tak złożony kod może być trudny do zrozumienia i debugowania. W tej książce używam operatorów inkrementacji i dekrementacji tylko w sposób prosty i łatwy do zrozumienia, jak w przedstawionych wcześniej przykładach zastosowania. Wprowadzam te operatory w tym momencie, ponieważ są one powszechnie używane w niektórych typach pętli.

### Pętla while w języku C++

W C++ pętla `while` jest zapisywana w następującym ogólnym formacie:

```
while (WyrażenieLogiczne)  
{  
    instrukcja;  
    instrukcja;  
    itd;  
}
```

Podczas wykonywania pętli `while` sprawdzana jest wartość wyrażenia logicznego. Jeśli wyrażenie to przyjmuje wartość `true`, to zostaną wykonane instrukcje zawarte w nawiasach klamrowych, a następnie pętla rozpoczęcie kolejny obieg. Jeśli wyrażenie logiczne przyjmie wartość `false`, to pętla zakończy się i program wznowi wykonywanie instrukcji znajdującej się bezpośrednio po niej.

Jeśli piszesz pętlę `while` zawierającą w ciele tylko jedną instrukcję, to nie musisz umieszczać jej w nawiasach klamrowych. Taką pętlę można zapisać w następującym ogólnym formacie:

```
while (WyrażenieLogiczne)  
    instrukcja;
```

Kiedy wykonywana jest pętla `while` napisana w tym formacie, to sprawdzana jest wartość wyrażenia logicznego. Jeśli wyrażenie ma wartość `true`, to zostanie wykonana jedna instrukcja, która znajduje się w następnym wierszu, a potem pętla rozpoczęcie kolejny obieg. Jeśli jednak wyrażenie logiczne ma wartość `false`, to pętla się zakończy.

Oto przykład pętli `while`, która pięć razy wyświetla słowo „Cześć”:

```
int count = 0;  
while (count < 5)  
{
```

```

        cout << "Cześć" << endl;
        count++;
    }
}

```

### Pętla do-while w języku C++

W C++ pętla do-while jest zapisana w następującym ogólnym formacie:

```

do
{
    instrukcja;
    instrukcja;
    itd;
} while (WyrażenieLogiczne);

```

Podobnie jak w pętli while, nawiasy klamrowe są opcjonalne, jeżeli w ciele pętli znajduje się tylko jedna instrukcja. Oto ogólny format pętli do-while z tylko jedną warunkowo wykonaną instrukcją:

```

do
    instrukcja;
while (WyrażenieLogiczne);

```

Zauważ jednak, że na samym końcu instrukcji do-while pojawia się średnik. Jest on elementem wymaganym, a pominięcie go jest częstym błędem.

Oto przykład pętli do-while, która pięć razy wypisuje słowo „Cześć”:

```

int count = 0;
do
{
    cout << "Cześć" << endl;
    count++;
} while (count < 5);

```



**UWAGA:** Język C++ nie definiuje pętli, która byłaby odpowiednikiem pętli Do-Until, omówionej wcześniej w tym rozdziale.

### Pętla for w języku C++

Konstrukcja pętli for składa się z części inicjalizowania, testowania i inkrementowania licznika zmiennej. Oto ogólny format pętli for:

```

for (WyrażenieInicjalizujące; WyrażenieTestujące; WyrażenieInkrementujące)
{
    instrukcja;
    instrukcja;
    itd.
}

```

Instrukcje umieszczone w nawiasach klamrowych są wykonywane przy każdym obiegu pętli. Jeśli treść pętli zawiera tylko jedną instrukcję, to nawiasy klamrowe są opcjonalne, jak w poniższym ogólnym przykładzie:

```

for (WyrażenieInicjalizujące; WyrażenieTestujące; WyrażenieInkrementujące)
    instrukcja;

```

Pierwszy wiersz pętli `for` jest tak zwany **nagłówkiem pętli**. Po słowie kluczowym `for` w nawiasie znajdują się trzy wyrażenia oddzielone od siebie średnikami. (Zauważ, że po trzecim wyrażeniu nie ma średnika).

Pierwsze wyrażenie jest **wyrażeniem inicjalizującym**. Zwykle używa się go do zapisania wartości początkowej w zmiennej licznika. Jest to pierwsza operacja wykonywana przez pętlę i jest wykonywana tylko raz. Drugie wyrażenie to **wyrażenie testujące**. Jest to wyrażenie logiczne, które steruje wykonywaniem pętli. Dopóki to wyrażenie jest prawdziwe, będzie powtarzana cała zawartość pętli `for`. Pętla `for` jest pętlą wstępna testującą, więc ocenia ona wyrażenie testowe przed każdą iteracją. Trzecie wyrażenie to **wyrażenie inkrementujące**, które jest wykonywane na końcu każdej iteracji. Zazwyczaj jest to instrukcja, która inkrementuje zmienną licznika pętli.

Oto przykład prostej pętli `for`, która pięć razy wypisuje słowo „Cześć”:

```
for (count = 1; count <= 5; count++)
{
    cout << "Cześć" << endl;
}
```

W tej pętli wyrażeniem inicjalizującym jest `count = 1`, wyrażeniem testującym jest `count <= 5`, natomiast wyrażenie inkrementujące to `count++`. Ciało pętli składa się z tylko jednej instrukcji. Oto podsumowanie tego, co się dzieje w trakcie wykonywania tej pętli:

1. Wykonywane jest wyrażenie inicjalizujące — `count = 1`. Przypisuje ono wartość 1 do zmiennej `count`.
2. Następnie sprawdzane jest wyrażenie `count <= 5`. Jeśli jest prawdziwe, to przechodzimy do kroku 3. W przeciwnym wypadku pętla zostanie zakończona.
3. Wykonywana jest instrukcja `cout << "Cześć" << endl;`.
4. Następnie wykonywane jest wyrażenie inkrementujące — `count++`. Dodaje ono 1 do zmiennej `count`.
5. Następuje powrót do kroku 2.

## Pytania kontrolne

### Test jednokrotnego wyboru

1. W pętli \_\_\_\_\_ liczba iteracji uzależniona jest od tego, czy spełniony jest warunek typu prawda – fałsz.
  - a) boolowskiej
  - b) warunkowej
  - c) decyzyjnej
  - d) licznikowej
2. Pętla \_\_\_\_\_ wykonuje określoną liczbę iteracji.
  - a) boolowska
  - b) warunkowa
  - c) decyzyjna
  - d) licznikowa

3. \_\_\_\_\_ to jednokrotne wykonanie instrukcji zawartych w pętli.
  - a) cykl
  - b) obrót
  - c) orbita
  - d) iteracja
4. W pętli `While` warunek sprawdzany jest \_\_\_\_\_ iteracji.
  - a) na początku
  - b) na końcu
  - c) w czasie trwania
  - d) nigdy
5. W pętli `Do-While` warunek sprawdzany jest \_\_\_\_\_ iteracji.
  - a) na początku
  - b) na końcu
  - c) w czasie trwania
  - d) nigdy
6. W pętli `For` warunek sprawdzany jest \_\_\_\_\_ iteracji.
  - a) na początku
  - b) na końcu
  - c) w czasie trwania
  - d) nigdy
7. Pętla \_\_\_\_\_ nigdy nie zakończy działania, chyba że program zostanie zamknięty.
  - a) nieokreślona
  - b) ciągła
  - c) nieskończona
  - d) ponadczasowa
8. Pętla, w której warunek jest sprawdzany \_\_\_\_\_, wykona co najmniej jedną iterację.
  - a) na
  - b) na końcu
  - c) zawsze
  - d) wewnątrz
9. \_\_\_\_\_ to zmienna, w której zapisywana jest suma bieżąca.
  - a) wartownik
  - b) suma
  - c) razem
  - d) akumulator
10. \_\_\_\_\_ to specjalna wartość, która wskazuje koniec listy wartości.  
Wartość ta nie może być taka sama jak inne elementy listy.
  - a) wartownik
  - b) flaga
  - c) sygnał
  - d) akumulator

### **Prawda czy fałsz?**

1. Pętla warunkowa wykonuje określoną liczbę iteracji.
2. W pętli `While` warunek jest sprawdzany na początku.
3. W pętli `Do-While` warunek jest sprawdzany na początku.
4. W ciele pętli `For` nie należy umieszczać instrukcji, które modyfikują wartość zmiennej licznikowej.
5. W ciele pętli nie można wyświetlić wartości zmiennej licznikowej.
6. Wartość zmiennej licznikowej można inkrementować tylko o 1.
7. Następująca instrukcja dekrementuje wartość zmiennej `x`: `Set x = x - 1`.
8. Akumulatora nie trzeba inicjalizować żadną wartością.
9. W przypadku pętli zagnieżdżonej w każdej iteracji pętli zewnętrznej pętla wewnętrzna wykonuje wszystkie swoje iteracje.
10. Aby obliczyć całkowitą liczbę iteracji, które wykona pętla zagnieżdżona, należy dodać do siebie liczbę iteracji wszystkich pętli.

### **Krótką odpowiedź**

1. Dlaczego w kodzie powinno się wcinać linie składające się na ciało pętli?
2. Wyjaśnij różnicę między pętlą, w której warunek jest sprawdzany na początku, a pętlą, w której warunek sprawdzany jest na końcu.
3. Jak działa pętla warunkowa?
4. Jak działa pętla licznikowa?
5. Jakie trzy operacje wykonuje zazwyczaj pętla licznikowa?
6. Co to jest pętla nieskończona? Stwórz taką pętlę.
7. Któża pętlę przypomina pętla `For` na schemacie blokowym?
8. Dlaczego tak ważne jest odpowiednie zainicjalizowanie akumulatora?
9. Jaką zaletę ma korzystanie z wartownika?
10. Dlaczego wartość wartownika trzeba dobierać w sposób przemyślany?

### **Warsztat projektanta algorytmów**

1. Zaprojektuj pętlę `While`, która umożliwi użytkownikowi wprowadzenie liczby. Liczba ta powinna zostać pomnożona przez 10 i zapisana w zmiennej `product`. Pętla powinna działać dopóty, dopóki w zmiennej `product` będzie wartość mniejsza niż 100.
2. Zaprojektuj pętlę `Do-While`, która poprosi użytkownika o wprowadzenie dwóch liczb. Liczby te należy zsumować i wyświetlić ich sumę. Pętla powinna pytać użytkownika, czy chce wykonać działanie ponownie. Jeżeli tak, cała procedura powinna zostać powtórzona. W przeciwnym wypadku pętla powinna zakończyć działanie.
3. Zaprojektuj pętlę `For`, która będzie wyświetlała następujące liczby:  
0, 10, 20, 30, 40, 50 ... 1000
4. Zaprojektuj pętlę, która poprosi użytkownika o wprowadzenie liczby. Pętla powinna wykonać 10 iteracji i zapisywać sumę bieżącą wprowadzonych przez użytkownika liczb.

5. Zaprojektuj pętlę, która obliczy sumę następującego szeregu:

$$\frac{1}{30} + \frac{2}{29} + \frac{3}{28} + \dots + \frac{30}{1}$$

6. Zaprojektuj pętlę, która wyświetli w 10 wierszach znaki #. W każdym wierszu powinno się znajdować 15 znaków #.

7. Zamień poniższą pętlę While na pętlę Do-While:

```
Declare Integer x = 1
While x > 0
    Display "Wprowadź liczbę."
    Input x
End While
```

8. Zamień poniższą pętlę Do-While na pętlę While:

```
Declare String sure
Do
    Display "Czy chcesz już zakończyć?"
    Input sure
    While sure != "T" AND sure != "t"
```

9. Zamień poniższą pętlę While na pętlę For:

```
Declare Integer count =
While count < 50
    Display "Licznik jest równy ", count
    Set count = count + 1
End While
```

10. Zamień poniższą pętlę For na pętlę While:

```
Declare Integer count
For count = 1 To 50
    Display count
End For
```

## Ćwiczenia z wykrywania błędów

1. Znajdź błąd w poniższym pseudokodzie:

```
Declare Boolean finished = False
Declare Integer value, cube

While NOT finished
    Display "Wprowadź liczbę, którą chcesz podnieść do sześciennu."
    Input value;
    Set cube = value^3
    Display value, " podniesione do sześciennu wynosi ", cube
End While
```

2. Programista, pisząc poniższy, kod chciał, aby na ekranie wyświetliły się liczby od 1 do 60, a następnie komunikat Czas minął!. Program jednak nie zadziała prawidłowo. Znajdź błąd.

```
Declare Integer counter = 1
Const Integer TIME_LIMIT = 60

While counter < TIME_LIMIT
    Display counter
```

- ```

        Set counter = counter + 1
End While
Display "Czas minął!"
3. Programista chciał, aby poniższy program pobierał od użytkownika pięć par
liczb, obliczał sumę każdej z par, a następnie sumę wszystkich liczb. Program
nie zadziała jednak tak, jak chciał programista. Znajdź błąd.

```

```

// Program oblicza sumę pięciu par liczb
Declare Integer number, sum, total
Declare Integer sets, numbers

Constant Integer MAX_SETS = 5
Constant Integer MAX_NUMBERS = 2

Set sum = 0;
Set total = 0;

For sets = 1 To MAX_NUMBERS
    For numbers = 1 To MAX_SETS
        Display "Wprowadź liczbę", numbers, " pary numer ", sets, "."
        Input number;
        Set sum = sum + number
    End For
    Display "Suma pary numer ", sets, " jest równa ", sum, "."
    Set total = total + sum
    Set sum = 0
End For

Display "Suma wszystkich liczb jest równa ", total,

```

## Ćwiczenia programistyczne

### 1. Kolekcjoner owadów

Kolekcjoner codziennie poluje na owady. Zaprojektuj program, który będzie obliczał sumę owadów zebranych w ciągu siedmiu dni. Program powinien w pętli pytać użytkownika, ile owadów zebrał każdego dnia tygodnia, a po zakończeniu pętli wyświetlić sumę zebranych owadów.

### 2. Spalone kalorie

Biegając na bieżni, można w ciągu minuty spalić 3,9 kalorii. Zaprojektuj program, który w pętli wyświetli liczbę spalonych kalorii po 10, 15, 20, 25 i 30 minutach biegania.

### 3. Budżet

Zaprojektuj program, który poprosi użytkownika o wprowadzenie budżetu miesięcznego. Następnie w pętli poproś użytkownika o wprowadzenie poszczególnych wydatków w danym miesiącu i zapisz sumę bieżącą wydatków. Po wyjściu z pętli program powinien wyświetlić wartość, o którą użytkownik przekroczył budżet lub która pozostała we wskazanym budżecie.

#### 4. Suma liczb

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie szeregu liczb dodatnich. Kiedy użytkownik wprowadzi ujemną liczbę, będzie to oznaczało, że chce zakończyć wprowadzanie liczb. Po wprowadzeniu liczb program powinien wyświetlić ich sumę.

#### 5. Podwyżka czesnego

Na pewnej uczelni czesne za rok nauki wynosi 6000 złotych. Uczelnia zapowiedziała, że przez kolejne pięć lat będzie co roku podnosiła czesne o 2%. Zaprojektuj program, który wyświetli wysokość czesnego w ciągu pięciu najbliższych lat.

#### 6. Przebyta odległość

Odległość, którą przebędzie pojazd, można obliczyć za pomocą następującego wzoru:

$$\text{odległość} = \text{prędkość} \cdot \text{czas}$$

Przykładowo pociąg jadący przez trzy godziny z prędkością 80 km/h przebędzie 240 km. Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie prędkości pojazdu (w km/h) i czasu trwania podróży. Następnie program za pomocą pętli powinien wyświetlić odległość przebytą przez pojazd w ciągu każdej godziny jazdy. Oto przykład działania programu:

| Z jaką prędkością jechał pojazd?   |                    |
|------------------------------------|--------------------|
| 80                                 |                    |
| Jak długo trwała podróż? 3 [Enter] |                    |
| Godzina                            | Przebyta odległość |
| 1                                  | 80                 |
| 2                                  | 160                |
| 3                                  | 240                |

#### 7. Średnie opady deszczu

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie danych i obliczał średnie opady deszczu w danym roku. Na samym początku program powinien poprosić o wprowadzenie liczby lat — pętla zewnętrzna będzie wykonywała iteracje dla każdego roku. Pętla wewnętrzna będzie wykonywała 12 iteracji — dla każdego z miesięcy. W każdej iteracji pętli wewnętrznej program powinien poprosić użytkownika o sumę opadów w danym miesiącu. Po wykonaniu wszystkich iteracji program powinien wyświetlić liczbę miesięcy, sumę opadów i średni miesięczny opad deszczu we wskazanym okresie.

#### 8. Tabela konwersji stopni Celsjusza na stopnie Fahrenheita

Zaprojektuj program, który będzie wyświetlał tabelkę zawierającą temperaturę w stopniach Celsjusza od 0 do 20 i odpowiadającą jej temperaturę w stopniach Fahrenheita. Wzór służący do zamiany stopni Celsjusza na stopnie Fahrenheita wygląda następująco:

$$F = \frac{9}{5}C + 32$$

We wzorze  $F$  to temperatura w stopniach Fahrenheita, a  $C$  to temperatura w stopniach Celsjusza. Aby wyświetlić tabelkę, skorzystaj z pętli.

#### 9. Groszowe wynagrodzenie

Zaprojektuj program, który będzie wyświetlał sumę zarobionych przez pewien okres pieniędzy, przy założeniu, że wynagrodzenie w pierwszym dniu wynosi 1 grosz, w drugim dniu 2 grosze, a każdego kolejnego dnia wynagrodzenie jest dwa razy większe. Program powinien zapytać użytkownika o liczbę dni, a następnie wyświetlić tabelkę, w której znajdzie się wartość wynagrodzenia w każdym dniu. Na końcu program powinien wyświetlić sumę wynagrodzenia w danym okresie. Wyświetlone liczby powinny być wyrażone w złotówkach, a nie groszach.

#### 10. Największa i najmniejsza

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie serii liczb. Kiedy użytkownik wprowadzi wartość -99, oznacza to koniec serii. Po wprowadzeniu wszystkich liczb program powinien wyświetlić najmniejszą i największą z nich.

#### 11. Pierwsze i ostatnie

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie serii imion (bez określonej kolejności). Po wprowadzeniu ostatniego imienia program powinien wyświetlić pierwsze i ostatnie imię w kolejności alfabetycznej. Przykładowo, jeżeli użytkownik wprowadzi imiona Krystyna, Janusz, Adam, Barbara, Zbigniew i Krzysztof, program powinien wyświetlić imiona Adam i Zbigniew.

#### 12. Silnia

Zapis  $n!$  oznacza w matematyce silnię liczby nieujemnej  $n$ . Silnia liczby  $n$  jest równa iloczynowi wszystkich liczb całkowitych od 1 do  $n$ . Przykładowo:

$$7! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 = 5040$$

natomiast:

$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie nieujemnej liczby całkowitej, a następnie wyświetlał jej silnię.

#### 13. Tabliczka mnożenia

Zaprojektuj program, który będzie używał zagnieżdzonych pętli do wyświetlania tabliczki mnożenia dla liczb od 1 do 12. Wynik programu powinien wyglądać tak:

```
1 * 0 = 0
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
```

i tak dalej...

```
12 * 9 = 108
12 * 10 = 120
12 * 11 = 132
12 * 12 = 144
```



**TEMATYKA**

6.1 Wprowadzenie do funkcji:  
generowanie liczb losowych  
6.2 Tworzenie własnych funkcji

6.3 Inne funkcje biblioteczne  
6.4 Rzut oka na języki Java, Python i C++

**6.1**

## Wprowadzenie do funkcji: generowanie liczb losowych

**WYJAŚNIENIE:** Funkcja to moduł, który zwraca wartość. W większości języków programowania dostępna jest gotowa biblioteka zawierająca wiele funkcji służących do wykonywania różnych operacji. W bibliotece tej znajduje się bardzo często funkcja przeznaczona do generowania liczb losowych.

W rozdziale 3. wyjaśniłem, że moduł składa się ze zbioru instrukcji, które służą do wykonania określonego zadania w programie. Kiedy chcesz, aby moduł wykonał to zadanie, musisz go wywołać — dzięki temu program wykona umieszczone w module instrukcje.

Funkcja to specjalny rodzaj modułu. Jest ona pod pewnymi względami do niego podobna:

- Funkcja składa się ze zbioru instrukcji, które służą do wykonania określonego zadania.
- Aby uruchomić funkcję, trzeba ją wywołać.

Kiedy jednak funkcja zakończy działanie, zwraca także do miejsca w programie, w którym została wywołana, pewną wartość. Wartość ta może zostać użyta w taki sam sposób jak jakakolwiek inna wartość: można ją przypisać do zmiennej, wyświetlić na ekranie, wykorzystać w wyrażeniu matematycznym (jeśli jest to liczba) itp.



**UWAGA:** Warto wspomnieć, że funkcje są wywoływanie nieco inaczej niż moduły. Aby wywołać moduł w pseudokodzie, używaliśmy polecenia Call. Jednak nie robimy tak w przypadku funkcji — funkcję wywołuje się najczęściej jako część innej instrukcji, która w jakiś sposób korzysta z wartości zwracanej przez tę funkcję. Przykładowo wywołanie funkcji można umieścić wewnątrz polecenia Set, które przypisze zwróconą wartość do zmiennej. Aby wyświetlić wartość zwracaną przez funkcję, wywołanie funkcji można umieścić wewnątrz polecenia Display. W tym rozdziale zobaczymy bardzo wiele takich przykładów.

## Funkcje biblioteczne

W większości języków programowania dostępna jest biblioteka, która zawiera gotowe już funkcje. Nazywamy je **funkcjami bibliotecznymi** i są one wbudowane w język programowania, więc można je wywołać w dowolnym momencie. Funkcje te bardzo ułatwiają życie programisty, ponieważ wykonują bardzo często używane operacje. Jak będziesz mieć okazję zobaczyć w tym rozdziale, są to funkcje operujące na liczbach, wykonujące różne operacje matematyczne, konwertujące dane jednego typu na inny, operujące na ciągach znaków i wiele innych.

Kod takich funkcji bibliotecznych w danym języku zapisany jest zazwyczaj w specjalnych plikach. Pliki te kopiowane są na komputer w momencie instalacji kompilatora lub interpretera. Kiedy wywołasz taką funkcję biblioteczną w swoim programie, kompilator lub interpreter automatycznie ją wykona — nie musisz się więc martwić umieszczeniem kodu funkcji w swoim programie. Dzięki temu nie musisz znać kodu danej funkcji, a wystarczy jedynie wiedza, jak dana funkcja działa, jakie argumenty należy do niej przekazać i jakiego typu dane zwraca.

Ponieważ nie widać, jakie operacje mają miejsce wewnątrz funkcji bibliotecznych, wielu programistów nazywa je **czarnymi skrzynkami** (ang. *black boxes*). Termin ten służy tutaj do opisu mechanizmu, w którym przekazuje się pewne dane, następnie mechanizm ten wykonuje na tych danych jakieś operacje i zwraca dane wyjściowe. Ideę tę przedstawiłem na rysunku 6.1.



Rysunek 6.1. Funkcja biblioteczna jako czarna skrzynka

W tym podrozdziale, na przykładzie funkcji bibliotecznej służącej do generowania liczb losowych, wyjaśnię jak działają funkcje. Funkcja ta jest dostępna w wielu językach programowania, a my przyjrzymy się kilku ciekawym programom, które może za jej pomocą stworzyć. W kolejnym podrozdziale nauczymy się tworzyć swoje własne funkcje. W ostatnim podrozdziale wróćmy do tematu funkcji bibliotecznych i zaprezentujemy jeszcze kilka innych przydatnych funkcji występujących często w językach programowania.

## Korzystanie z funkcji random

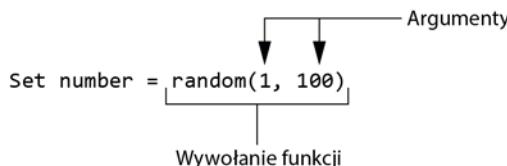
W większości języków programowania dostępna jest funkcja do generowania liczb losowych. W pseudokodzie posłuży nam do tego funkcja `random`. Liczby losowe mają zastosowanie w wielu zadaniach programistycznych. Oto kilka przykładów:

- Liczby losowe są często wykorzystywane w grach. Przykładowo w grach komputerowych, w których gracz rzuca kością, liczbę oczek reprezentuje liczba losowa. W programach, w których ciągnie się karty ze stosu, wartość karty także jest reprezentowana przez liczbę losową.
- Liczby losowe są bardzo pomocne podczas różnych symulacji. W pewnych symulacjach komputer musi zdecydować w sposób losowy, jak zachowa się osoba, zwierzę, owad lub inny organizm. Można stworzyć wzory, w których ta liczba losowa będzie decydowała o zdarzeniach i operacjach mających miejsce w programie.
- Liczby losowe są pomocne w programach do analizy statystycznej, w których trzeba analizować losowe dane.
- Liczby losowe często są wykorzystywane do szyfrowania wrażliwych danych.

Na poniższym przykładzie pokazałem, w jaki sposób można wywołać funkcję `random` w pseudokodzie. Założymy, że `number` jest zmienną typu `Integer`:

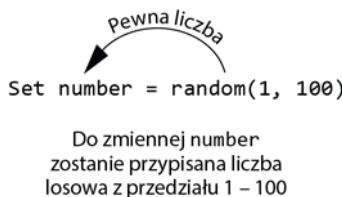
```
Set number = random(1, 100)
```

Fragment `random(1, 100)` to właśnie wywołanie funkcji `random`. Zwróć uwagę, że w nawiasach znajdują się dwa argumenty: 1 i 100. Poprzez te argumenty informujemy funkcję `random`, że chcemytrzymać w wyniku liczbę losową z przedziału 1 – 100. Na rysunku 6.2 przedstawiłem ten fragment polecenia.



**Rysunek 6.2.** Instrukcja wywołująca funkcję `random`

Zauważ, że wywołanie funkcji `random` ma miejsce po prawej stronie operatora `=`. Gdy wywołamy funkcję, wygeneruje ona liczbę losową z przedziału 1 – 100 i ją **zwróci**. Liczba, którą funkcja zwróci, zostanie następnie przypisana do zmiennej `number`, tak jak pokazałem to na rysunku 6.3.



**Rysunek 6.3.** Funkcja `random` zwraca wartość

Na listingu 6.1 przedstawiłem pseudokod programu, w którym wykorzystałem funkcję `random`. W linii 2. generuję liczbę losową z przedziału 1 – 10 i przypisuję ją do zmiennej `number`. W zaprezentowanym wyniku działania programu jest to liczba 7, ale równie dobrze może to być jakaś inna liczba miesząca się w przedziale 1 – 10.

### **Listing 6.1**

```
1 Declare Integer number
2 Set number = random(1, 10)
3 Display number
```

### **Wynik działania programu**

7



**UWAGA:** Sposób konfiguracji programu, aby można było w nim korzystać z funkcji bibliotecznych, różni się w zależności od języka. W niektórych językach programowania nie trzeba robić w takim przypadku nic — takie podejście przyjęłem w pseudokodzie. W innych językach należy na samej górze programu wprowadzić instrukcję informującą o tym, z których funkcji bibliotecznych masz zamiar korzystać w programie.

Na listingu 6.2 zaprezentowałem inny przykład. W programie tym pętla `For` wykonuje pięć iteracji. Wewnątrz pętli, w instrukcji zawartej w linii 9., aby wygenerować liczbę losową z przedziału 1 – 100, wywołuję funkcję `random`.

### **Listing 6.2**



```
1 // Deklaracja zmiennych
2 Declare Integer number, counter
3
4 // Poniższa pętla generuje
5 // pięć losowych liczb
6 For counter = 1 To 5
7   // Generujemy liczbę losową z przedziału
8   // od 1 do 100 i przypisujemy ją do zmiennej number
9   Set number = random(1, 100)
10
11  // Wyświetlamy liczbę
12  Display number
13 End For
```

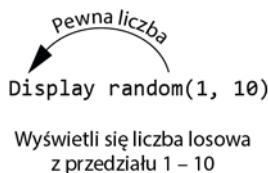
### **Wynik działania programu**

89  
7  
16  
41  
12

W przypadku obu programów, z listingów 6.1 i 6.2, za pomocą funkcji `random` przypisujemy zwrconą wartość do zmiennej `number`. Jeśli jednak chcesz tylko wyświetlić liczbę losową na ekranie, nie ma potrzeby przypisywania jej do zmiennej — możesz po prostu przekazać wartość zwracaną przez funkcję `random` do polecenia `Display`, tak jak tutaj:

```
Display random(1, 10)
```

W momencie uruchomienia tego polecenia wywołana zostanie funkcja `random`. Funkcja ta wygeneruje liczbę losową z przedziału 1 – 10. Wartość zwrócona przez liczbę zostanie następnie przekazana do polecenia `Display`. W wyniku tego na ekranie wyświetli się liczba losowa z przedziału 1 – 10. Ilustruje to rysunek 6.4.



**Rysunek 6.4.** Wyświetlanie liczb losowych

Na listingu 6.3 pokazałem, w jaki sposób można uprościć program z listingu 6.2. W programie tym także wyświetlamy pięć losowych liczb, ale nie korzystamy ze zmiennej. Wartość zwracana przez funkcję `random` jest natychmiast przekazywana do polecenia `Display` w linii 4.

### **Listing 6.3**

```
1 // Zmienna licznikowa
2 Declare Integer counter
3
4 // W pętli wyświetla się pięć losowych liczb
5 For counter = 1 To 5
6   Display random(1, 100)
7 End For
```

### **Wynik działania programu**

```
32
79
6
12
98
```



## W centrum uwagi

### Korzystanie z liczb losowych

Doktor Kozioł prowadzi zajęcia z podstaw statystyki i poprosił Cię o napisanie programu do symulacji rzutu kośmi, którego będzie mógł użyć podczas zajęć. Program powinien generować dwie liczby losowe z przedziału 1 – 6, a następnie je wyświetlać. Z rozmowy z doktorem Koziolem wiesz, że chciałby wykorzystać ten program, aby zasymulować kilka rzutów kośmi — raz za razem. Zdecydowałeś się więc skorzystać z pętli, w której będziesz pytać użytkownika, czy chce zasymulować kolejny rzut kośmi. Pętla będzie się powtarzała tak długo, jak użytkownik będzie wprowadzał „t”. Na lisingu 6.4 przedstawiłem pseudokod takiego programu, a na rysunku 6.5 jego schemat blokowy.

#### **Listing 6.4**

```

1 // Deklaracja zmiennej służącej
2 // do sterowania pętlą
3 Declare String again
4
5 Do
6     // Wykonujemy rzut kośmi
7     Display "Rzucam kośmi..."
8     Display "Oto wynik:"
9     Display random(1, 6)
10    Display random(1, 6)
11
12    // Rzucić jeszcze raz?
13    Display "Chcesz wykonać kolejny rzut? (t = tak)"
14    Input again
15 While again == "t" OR again == "T"

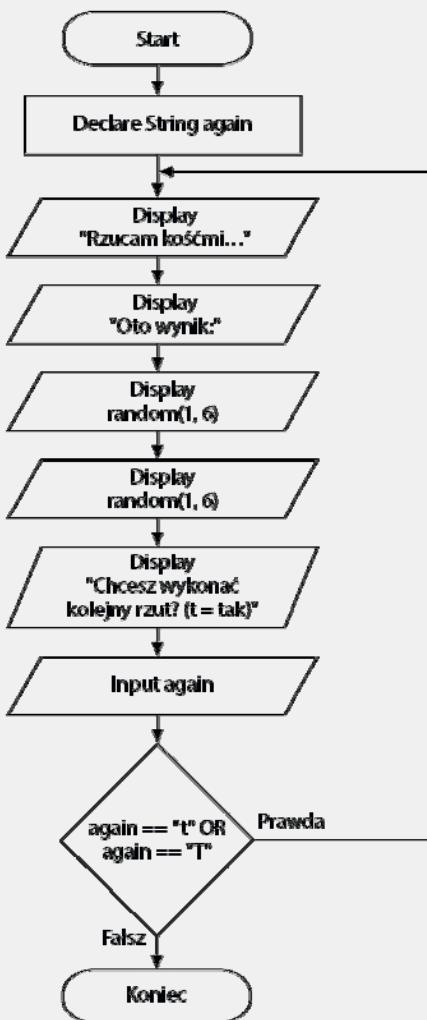
```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

```

Rzucam kośmi...
Oto wynik:
2
6
Chcesz wykonać kolejny rzut? (t = tak)
t [Enter]
Rzucam kośmi...
Oto wynik:
4
1
Chcesz wykonać kolejny rzut? (t = tak)
t [Enter]
Rzucam kośmi...
Oto wynik:
3
3
Chcesz wykonać kolejny rzut? (t = tak)
n [Enter]

```



**Rysunek 6.5.** Schemat blokowy programu z listingu 6.4

Funkcja `random` zwraca liczbę całkowitą, więc możesz zapisać jej wywołanie wszędzie tam, gdzie umieściłbyś jakąkolwiek liczbę całkowitą. Widziałeś już przykłady, w których wynik zwrócony przez funkcję przypisywałem do zmiennej lub przekazywałem do polecenia `Display`. Aby jeszcze lepiej zilustrować, o co tutaj chodzi, przedstawię wyrażenie matematyczne zawierające funkcję `random`:

```
Set x = random(1, 10) * 2
```

W poleceniu tym generuję liczbę losową i mnożę ją przez 2. Wynik działania przypisuję do zmiennej `x`. Możesz także wartość zwracaną przez funkcję sprawdzać w poleceniu `If-Then`, co zaprezentowałem z kolejnej sekcji „W centrum uwagi”.



## W centrum uwagi

### Wykorzystanie liczb losowych do reprezentowania innych wartości

Doktor Kozioł jest bardzo zadowolony z programu symulującego rzut kością, który dla niego napisałś, i poprosił Cię o stworzenie jeszcze jednego programu. Chciałby, aby program symulował serię rzutów monetą. Po każdym rzucie program powinien wyświetlić komunikat „orzeł” albo „reszka”.

Zdecydowałeś, że do symulacji rzutu monetą użyjesz liczby losowej z przedziału liczb losowych 1 lub 2. Stworzysz strukturę warunkową, która będzie wyświetlała tekst „orzeł”, jeżeli liczba losowa jest równa 1, lub tekst „reszka” w przeciwnym wypadku.

Na listingu 6.5 przedstawiłem pseudokod takiego programu, a na rysunku 6.6 jego schemat blokowy.

#### **Listing 6.5**

```

1 // Deklaracja zmiennej licznikowej
2 Declare Integer counter
3
4 // Stała określająca liczbę rzutów monetą
5 Constant Integer NUM_FLIPS = 10
6
7 For counter = 1 To NUM_FLIPS
8     // Symulujemy rzut monetą
9     If random(1, 2) == 1 Then
10        Display "Orzeł"
11     Else
12        Display "Reszka"
13    End If
14 End For

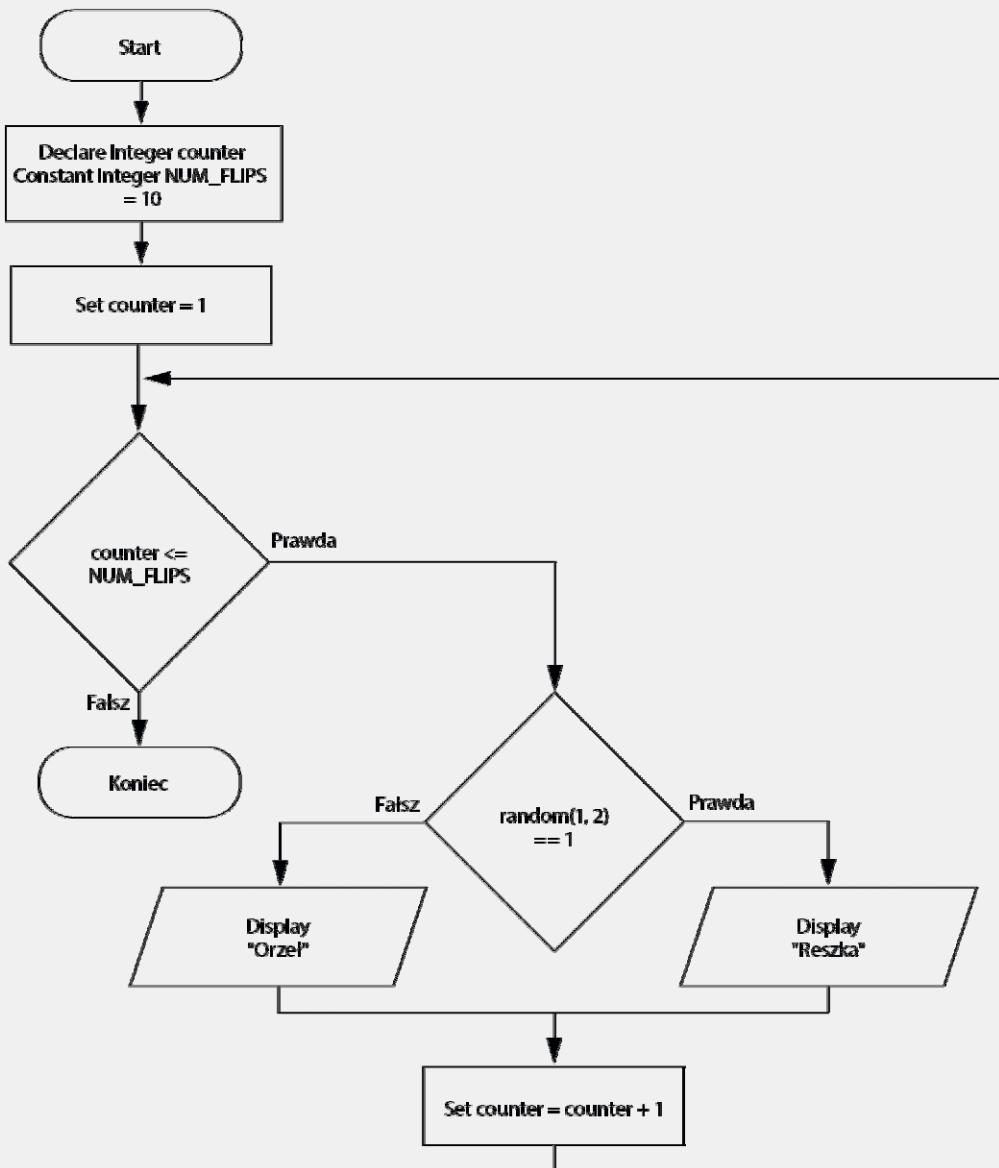
```

#### **Wynik działania programu**

```

Reszka
Reszka
Orzeł
Reszka
Orzeł
Orzeł
Orzeł
Reszka
Orzeł
Reszka

```



**Rysunek 6.6.** Schemat blokowy programu z listingu 6.5



## Punkt kontrolny

- 6.1. Czym różni się funkcja od modułu?
- 6.2. Co to jest funkcja biblioteczna?
- 6.3. Dlaczego funkcje biblioteczne określane są mianem „czarnych skrzynek”?

6.4. Jak działa poniższe polecenie w pseudokodzie?

Set *x* = random(1, 100)

6.5. Jak działa poniższe polecenie w pseudokodzie?

Display random(1, 20)

## 6.2

# Tworzenie własnych funkcji

**WYJAŚNIENIE:** W większości języków programowania można tworzyć własne funkcje. Tworząc funkcję, tworzysz tak naprawdę moduł, który może zwrócić wartość do miejsca, w którym został wywołany.

W rozdziale 3. wyjaśniłem, że moduł tworzymy, definiując go w kodzie. Funkcje tworzymy w podobny sposób. Oto kilka cech, którymi charakteryzuje się definicja funkcji:

- Pierwsza linia definicji funkcji nazywa się **nagłówkiem funkcji** i wskazuje się w niej, jakiego typu dane zwraca funkcja, jaką ma nazwę i jakie argumenty przyjmuje.
- Po nagłówku pojawia się **ciało funkcji**, składające się z jednej lub wielu instrukcji, które zostaną wykonane po wywołaniu funkcji.
- Jednym z poleceń składających się na ciało funkcji musi być **Return**. Polecenie **Return** służy do wskazania wartości, jaką zwróci funkcja, gdy zakończy działanie.

Oto ogólny format, za pomocą którego będziemy zapisywać funkcję w pseudokodzie:

```
Function TypDanych NazwaFunkcji(ListaParametrów)
    instrukcja
    instrukcja
    itd.
    Return wartość
End Function
```

**W funkcji musi się znajdować polecenie Return. Dzięki niemu wskazana wartość zostanie przesłana do tej części programu, w której nastąpiło wywołanie funkcji.**

W pierwszej linii znajduje się nagłówek funkcji składający się ze słowa **Function**, a następnie pojawiają się następujące elementy:

- **TypDanych** wskazuje, jakiego typu wartości zwraca funkcja. Przykładowo, jeżeli funkcja zwraca liczbę całkowitą, będzie to typ **Integer**. Jeżeli funkcja zwraca liczbę rzeczywistą, będzie to typ **Real**. W przypadku ciągu znaków będzie to typ **String**.
- **NazwaFunkcji** wskazuje, jaką nazwę ma funkcja. Podobnie jak w przypadku modułów, funkcjom także należy nadawać takie nazwy, które wskazują, do czego one służą. W większości języków programowania reguły dotyczące nazewnictwa funkcji są identyczne jak w przypadku nazw zmiennych czy nazw modułów.
- Opcjonalnie możesz wskazać w nawiasach listę parametrów. Jeżeli funkcja nie przyjmuje żadnych argumentów, pozostaw puste nawiasy.

Po linii z nagłówkiem funkcji pojawiają się jedna lub wiele instrukcji. Instrukcje te składają się na ciało funkcji i zostaną wykonane po wywołaniu funkcji. Jedną z tych instrukcji musi być **Return**, która wygląda następująco:

### Return wartość

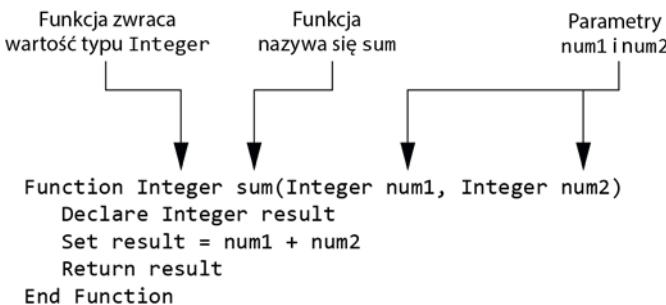
Wartość oznacza wartość, którą zwróci funkcja w miejscu jej wywołania. Może to być dowolna wartość: zmienią lub wyrażenie, które zwraca wartość (np. wyrażenie matematyczne). Wartość ta musi być tego samego typu co wartość wskazana w nagłówku funkcji. W przeciwnym wypadku pojawi się błąd.

W ostatniej linii deklaracji funkcji pojawiają się słowa `End Function`. Linia ta oznacza koniec definicji funkcji.

Oto przykład funkcji zapisanej w pseudokodzie:

```
Function Integer sum(Integer num1, Integer num2)
    Declare Integer result
    Set result = num1 + num2
    Return result
End Function
```

Na rysunku 6.7 przedstawiłem elementy nagłówka funkcji. Zwróć uwagę, że przykładowa funkcja zwraca wartość typu `Integer`, nazywa się `sum` i przyjmuje dwa argumenty typu `Integer` o nazwach `num1` i `num2`.



**Rysunek 6.7.** Elementy nagłówka funkcji

Funkcja ta przyjmuje jako argumenty dwie liczby całkowite i zwraca ich sumę. Przyjrzymy się teraz ciału funkcji i zobaczymy, jak ona działa. W pierwszej linii znajduje się deklaracja zmiennej typu `Integer` o nazwie `result`. W kolejnym poleceniu wartość zwracana przez wyrażenie `num1 + num2` jest przypisywana do zmiennej `result`. Następnie uruchamia się polecenie `Return`, które kończy działanie funkcji i zwraca wartość zapisaną w zmiennej `result` do tego miejsca w programie, w którym została ona wywołana.

Na listingu 6.6 przedstawiłem kod programu, w którym wywouję funkcję `sum`.

### Listing 6.6



```
1 Module main()
2     //Zmienne lokalne
3     Declare Integer firstAge, secondAge, total
4
5     //Pobieramy wiek użytkownika
6     //i wiek jego kolegi
7     Display "Ile masz lat?"
8     Input firstAge
```

```

9   Display "Ile lat ma Twój kolega?"
10  Input secondAge
11
12  // Obliczamy sumę wieku
13  Set total = sum(firstAge, secondAge)
14
15  // Wyświetlamy sumę wieku
16  Display "Razem macie ", total, " lat."
17 End Module
18
19 // Funkcja sum przyjmuje dwa argumenty
20 // i zwraca sumę tych argumentów jako liczbę całkowitą
21 Function Integer sum(Integer num1, Integer num2)
22   Declare Integer result
23   Set result = num1 + num2
24   Return result
25 End Function

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Ile masz lat?

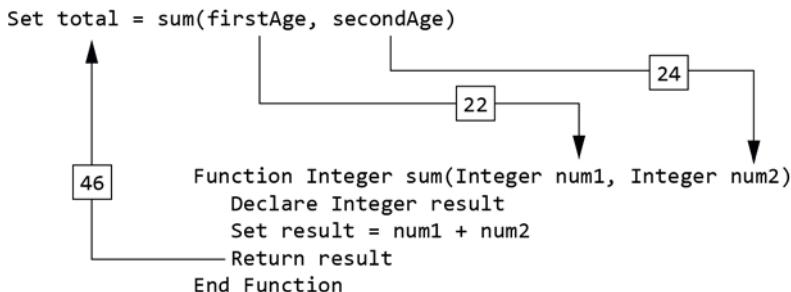
**22 [Enter]**

Ile lat ma Twój kolega?

**24 [Enter]**

Razem macie 46 lat.

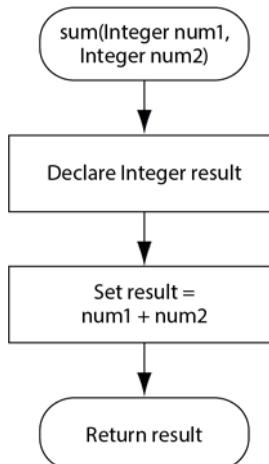
W module main pobieramy od użytkownika dwie liczby i zapisujemy je w zmiennych firstAge i secondAge. W linii 13. wywołujemy funkcję sum i przekazujemy do niej jako argumenty zmienne firstAge i secondAge. Wartość zwróconą przez funkcję zapisujemy w zmiennej total. W tym przykładzie funkcja zwróciła wartość 46. Na rysunku 6.8 pokazałem, w jaki sposób argumenty są przekazywane do funkcji i w jaki sposób funkcja zwraca wartość.



**Rysunek 6.8.** Przekazujemy argumenty do funkcji sum, a ona zwraca wartość

**Schemat blokowy funkcji**

Kiedy będziesz tworzyć schemat blokowy programu, w którym wywołujesz pewne funkcje, powinieneś także zamieścić schemat blokowy każdej z tych funkcji. Schemat blokowy funkcji rozpoczyna się od symbolu granicznego, w którym zazwyczaj wpisana jest nazwa funkcji i lista parametrów. W końcowym symbolu granicznym wpisany jest tekst *Return* oraz wartość lub wyrażenie, które funkcja zwraca. Na rysunku 6.9 pokazałem schemat blokowy funkcji sum z listingu 6.6.



**Rysunek 6.9.** Schemat blokowy funkcji sum

## Wykorzystanie instrukcji Return

Spójrz jeszcze raz na funkcję sum z listingu 6.6:

```

Function Integer sum(Integer num1, Integer num2)
    Declare Integer result
    Set result = num1 + num2
    Return result
End Function
  
```

Zwróć uwagę, że wewnętrz funkcji znajdują się trzy rzeczy: (1) zadeklarowana jest zmienna `result`, (2) do zmiennej `result` jest przypisywana wartość zwracana przez wyrażenie `num1 + num2` i (3) funkcja zwraca wartość przypisaną do zmiennej `result`.

Mimo że funkcja ta działa poprawnie, można ją nieco uprościć. Ponieważ polecenie `Return` może zwracać także wartość wyrażenia, można pominąć zmienną `result` i zmodyfikować kod w następujący sposób:

```

Function Integer sum(Integer num1, Integer num2)
    Return num1 + num2
End Function
  
```

W tej wersji programu nie zapisujemy już wartości wyrażenia `num1 + num2` w zmiennej `result`. Zamiast tego korzystamy z faktu, że polecenie `Return` może zwracać także wartość wyrażenia. Program robi więc dokładnie to samo co poprzedni, ale za pomocą tylko jednej operacji.



**UWAGA:** W większości języków programowania możesz przekazywać do funkcji dowolną liczbę argumentów, ale może ona zwrócić tylko jedną wartość.

## W jaki sposób korzystać z funkcji?

W większości języków programowania można tworzyć zarówno moduły, jak i funkcje. Funkcje mają wiele zalet, którymi charakteryzują się moduły: upraszczają kod, redukują powielanie kodu, ułatwiają testowanie, przyspieszają tworzenie i ułatwiają pracę w zespołach.

Ponieważ funkcje zwracają wartość, są bardzo przydatne w pewnych sytuacjach. Przykładowo możesz napisać funkcję, która będzie prosiła użytkownika o wprowadzenie danych, a następnie zwracała dane, które użytkownik wprowadził. Założymy, że poproszono Cię o zaprojektowanie programu, który będzie obliczał cenę sprzedaży danego produktu. Musisz w takim przypadku pobrać od użytkownika cenę detaliczną produktu. Oto funkcja, którą możesz zdefiniować:

```
Function Real getRegularPrice()
    // Zmienna lokalna
    Declare Real price

    // Pobieramy cenę detaliczną
    Display "Wprowadź cenę detaliczną produktu."
    Input price

    // Zwracamy cenę detaliczną
    Return price
End Function
```

Następnie w innej części kodu możesz tę funkcję wywołać w taki sposób:

```
// Pobieramy cenę detaliczną produktu
Set regularPrice = getRegularPrice()
```

Po uruchomieniu tej instrukcji wywołana zostanie funkcja `getRegularPrice`, która pobierze cenę od użytkownika i ją zwróci. Wartość tę następnie przypisujemy do zmiennej `regularPrice`.

Za pomocą funkcji możesz także upraszczać wyrażenia matematyczne. Przykładowo obliczanie ceny sprzedaży danego produktu jest bardzo proste: wystarczy obliczyć wartość rabatu i odjąć go od ceny detalicznej. W programie jednak operacja ta nie będzie już wyglądała tak prosto, co przedstawiłem poniżej (założymy, że stała `DISCOUNT_PERCENTAGE` jest stałą globalną i zawiera procentową wysokość rabatu):

```
Set salePrice = regularPrice - (regularPrice * DISCOUNT_PERCENTAGE)
```

Polecenie to ciężko zrozumieć, ponieważ wykonuje kilka operacji: oblicza wartość rabatu, odejmuje ją od wartości zapisanej w zmiennej `regularPrice` i przypisuje wynik do zmiennej `salePrice`. Można to wyrażenie uprościć, umieszczając jego część w funkcji. Oto przykład funkcji `discount`, która przyjmuje jako argument cenę detaliczną i zwraca wartość rabatu:

```
Function Real discount(Real price)
    Return price * DISCOUNT_PERCENTAGE
End Function
```

Następnie możemy wywołać tę funkcję w wyrażeniu:

```
Set salePrice = regularPrice - discount(regularPrice)
```

Taka instrukcja jest bardziej czytelna niż poprzednia i łatwo można z niej odczytać, że wartość rabatu jest odejmowana od ceny detalicznej. Na listingu 6.7 przedstawiam pełny pseudokod programu do obliczania ceny sprzedaży, w który korzystam z omówionej przed chwilą funkcji.

### **Listing 6.7**

```

1 // Stała globalna zawierająca wysokość rabatu
2 Constant Real DISCOUNT_PERCENTAGE = 0.20
3
4 // Moduł main jest punktem początkowym programu
5 Module main()
6     // Zmienne lokalne dla ceny detalicznej i ceny sprzedaży
7     Declare Real regularPrice, salePrice
8
9     // Pobieramy cenę detaliczną produktu
10    Set regularPrice = getRegularPrice()
11
12    // Obliczamy cenę sprzedaży
13    Set salePrice = regularPrice - discount(regularPrice)
14
15    // Wyświetlamy cenę sprzedaży
16    Display "Cena sprzedaży jest równa ", salePrice, " zł."
17 End Module
18
19 // Funkcja getRegularPrice prosi
20 // użytkownika o wprowadzenie ceny detalicznej produktu
21 // i zwraca wartość typu Real
22 Function Real getRegularPrice()
23     // Zmienna lokalna z ceną sprzedaży
24     Declare Real price
25
26     // Pobieramy cenę detaliczną produktu
27     Display "Wprowadź cenę detaliczną produktu."
28     Input price
29
30     // Zwracamy cenę detaliczną
31     Return price
32 End Function
33
34 // Funkcja discount przyjmuje jako argument
35 // cenę detaliczną produktu i zwraca wartość rabatu
36 // obliczoną na podstawie stałej DISCOUNT_PERCENTAGE
37 Function Real discount(Real price)
38     Return price * DISCOUNT_PERCENTAGE
39 End Function

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę detaliczną produktu.

**100.00 [Enter]**

Cena sprzedaży jest równa 80 zł.

## Tabelki IPO w przypadku funkcji

W rozdziale 2. wyjaśniłem, że tabelki IPO są prostym, ale efektywnym narzędziem do projektowania programu. Przypominam, że IPO oznacza: **dane wejściowe, przetwarzanie i dane wyjściowe**. Tabelka IPO służy do opisywania danych wejściowych, rodzaju przetwarzania i danych wyjściowych. Można ją także stworzyć dla poszczególnych funkcji w programie. Kolumna *Wejście* służy do określenia danych, które są przekazywane do funkcji, kolumna *Przetwarzanie* opisuje działanie, jakie wykonuje funkcja, a kolumna *Wyjście* określa dane, które funkcja zwraca. W przykładowych tabelkach IPO zamieszczonych na rysunku 6.10 przedstawiłem funkcje `getRegularPrice` i `discount` z programu z listingu 6.7.

| Tabelka IPO dla funkcji <code>getRegularPrice</code> |                                                            |                                                         |
|------------------------------------------------------|------------------------------------------------------------|---------------------------------------------------------|
| Wejście                                              | Przetwarzanie                                              | Wyjście                                                 |
| Brak                                                 | Prosi użytkownika o wprowadzenie ceny detalicznej produktu | Cena detaliczna produktu jako wartość <code>Real</code> |

| Tabelka IPO dla funkcji <code>discount</code> |                                                                                                               |                                                                   |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| Wejście                                       | Przetwarzanie                                                                                                 | Wyjście                                                           |
| Cena detaliczna produktu                      | Oblicza wartość rabatu, mnożąc cenę detaliczną produktu przez stałą globalną <code>DISCOUNT_PERCENTAGE</code> | Wartość rabatu dla danego produktu jako wartość <code>Real</code> |

Rysunek 6.10. Tabelki IPO dla funkcji `getRegularPrice` i `discount`

Zwróć uwagę, że tabelka IPO w zwięzły sposób opisuje tylko dane wejściowe funkcji, przetwarzanie danych i dane wyjściowe i nie pokazuje żadnych szczegółów dotyczących tego, w jaki sposób funkcja działa. W wielu przypadkach będą to jednak wystarczające informacje i można użyć takiej tabelki zamiast schematu blokowego. Decyzja dotycząca tego, czy korzystać z tabelki IPO czy ze schematu blokowego, czy też z obu tych rozwiązań, należy do programisty.

### W centrum uwagi

Modularyzacja kodu  
z wykorzystaniem funkcji

Kamil jest właścicielem firmy Make Your Own Music, która zajmuje się sprzedażą gitar, perkusji, banjo, syntezatorów i wielu innych instrumentów muzycznych. Personel zatrudniony przez Kamila otrzymuje wynagrodzenie w formie premii od sprzedawcy. Na koniec każdego miesiąca dla każdego sprzedawcy obliczana jest premia według tabeli 6.1.



**Tabela 6.1.** Wysokość premii dla sprzedawców

| Wartość sprzedaży w bieżącym miesiącu | Wysokość premii |
|---------------------------------------|-----------------|
| Poniżej 10000,00 zł                   | 10%             |
| 10000,00 – 14999,99 zł                | 12%             |
| 15000,00 – 17999,99 zł                | 14%             |
| 18000,00 – 21999,99 zł                | 16%             |
| 22000 zł lub więcej                   | 18%             |

Przykładowo sprzedawca, który w ciągu miesiąca sprzedał towar na łączną kwotę 16000 zł, otrzyma premię w wysokości 14% wartości sprzedaży (2240 zł). Inny pracownik sprzedał towar na łączną kwotę 20000 zł i zarobi 16% wartości sprzedaży (3200 zł). Pracownik, który sprzedał towar na łączną kwotę 30000 zł, zarobi 18% wartości sprzedaży (5400 zł).

Ponieważ wynagrodzenie jest wypłacane raz w miesiącu, Kamil pozwala pracownikom na pobranie zaliczki w maksymalnej wysokości 2000 zł. Kiedy obliczane jest wynagrodzenie, wartość zaliczki jest odejmowana od naliczonej w danym miesiącu premii. Jeżeli wysokość premii jest mniejsza od pobranej zaliczki, pracownik musi zwrócić różnicę Kamilowi. Aby obliczyć miesięczne wynagrodzenie pracownika, Kamil korzysta z następującego wzoru:

$$\text{wynagrodzenie} = \text{wartość sprzedaży} \cdot \text{wysokość premii} - \text{pobrańa zaliczka}$$

Kamil poprosił Cię o napisanie programu, który ułatwi mu obliczanie wynagrodzenia. Poniższy algorytm prezentuje, w jaki sposób powinien działać program:

1. Pobierz miesięczną wartość sprzedaży dla danego sprzedawcy.
2. Pobierz wartość pobranej zaliczki.
3. Na podstawie wartości sprzedaży pobierz procentową wysokość premii.
4. Za pomocą przedstawionego wzoru oblicz wynagrodzenie sprzedawcy. Jeżeli wartość wynagrodzenia jest ujemna, wyświetl informację, że pracownik musi zwrócić firmie pieniądze.

Na listingu 6.8 przedstawiłem pseudokod programu, który podzieliłem na kilka funkcji. Zamiast prezentować program w całości przyjrzymy się najpierw modułowi `main`, a następnie każdej funkcji z osobna. Oto moduł `main`:

### **Listing 6.8 Program do obliczania premii: moduł main**

```

1 Module main()
2   //Zmienne lokalne
3   Declare Real sales, commissionRate, advancedPay
4
5   //Pobieramy wartość sprzedaży
6   Set sales = getSales()
7
8   //Pobieramy wartość pobranej zaliczki
9   Set advancedPay = getAdvancedPay()
10

```

```

11 // Określamy wysokość premii
12 Set commissionRate = determineCommissionRate(sales)
13
14 // Obliczamy wynagrodzenie
15 Set pay = sales * commissionRate - advancedPay
16
17 // Wyświetlamy wynagrodzenie
18 Display "Wynagrodzenie wynosi ", pay, " zł."
19
20 // Sprawdzamy, czy wynagrodzenie jest ujemne
21 If pay < 0 Then
22     Display "Pracownik musi zwrócić"
23     Display "pieniądze firmie."
24 End If
25 End Module
26

```

W linii 3. deklaruję zmienne, w których zapisane będą wartości sprzedaży, wysokość premii i pobranej zaliczki. W linii 6. wywołuję funkcję `getSales`, która pobiera od użytkownika wartość sprzedaży miesięcznej i zwraca tę wartość. Wartość zwróconą przez funkcję przypisuję do zmiennej `sales`. W linii 9. wywołuję funkcję `getAdvancedPay`, która pobiera od użytkownika wysokość wypłaconej zaliczki i zwraca tę wartość. Wartość zwróconą przez funkcję przypisuję do zmiennej `advancedPay`.

W linii 12. wywołuję funkcję `determineCommissionRate` i przekazuję do niej jako argument zmienną `sales`. Funkcja ta zwraca wysokość premii, która zależy od wartości sprzedaży. Wartość premii przypisuję do zmiennej `commissionRate`. W linii 15. obliczam wynagrodzenie, a następnie w linii 18. wyświetlам je na ekranie. Instrukcja

If-Then w liniach od 21. do 24. sprawdza, czy wartość wynagrodzenia jest ujemna — jeżeli tak, wyświetlam komunikat informujący, że sprzedawca musi zwrócić pieniądze firmie. Na rysunku 6.11 przedstawiłem schemat blokowy modułu `main`.

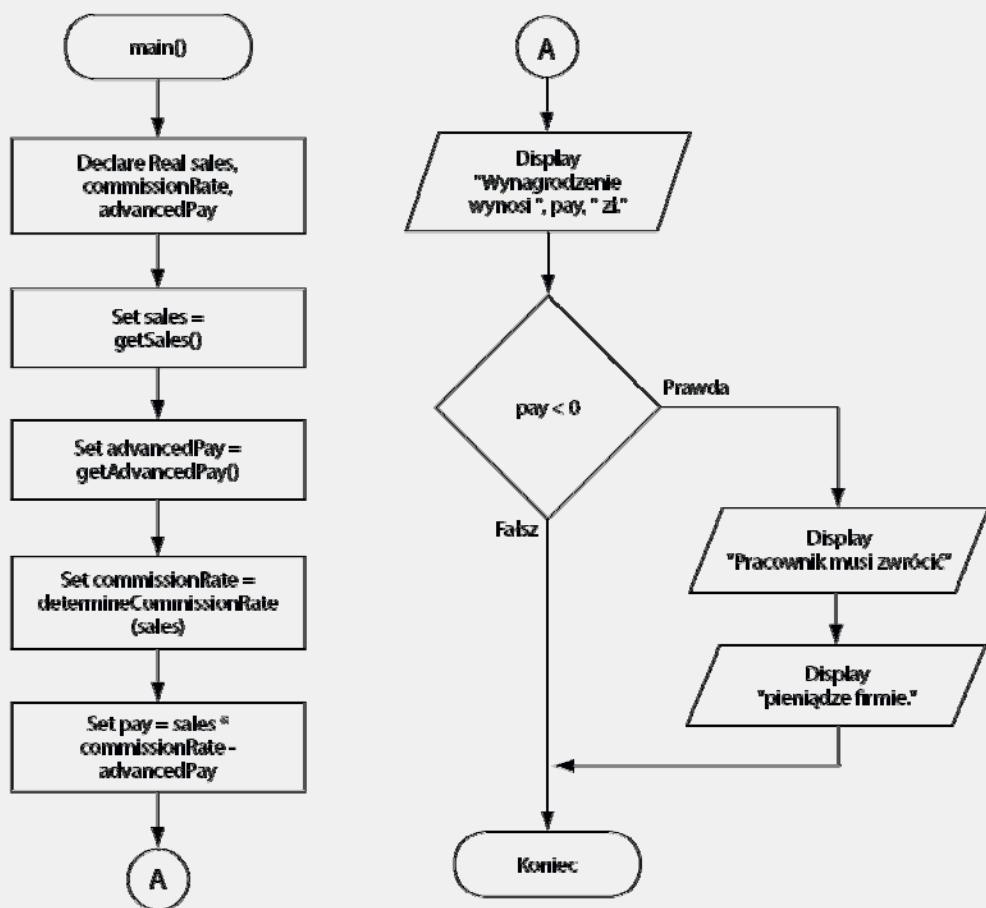
Poniżej znajduje się definicja funkcji `getSales`.

### **Listing 6.8 Program do obliczania premii (kontynuacja): funkcja `getSales`**

```

27 // Funkcja getSales pobiera od użytkownika
28 // wartość miesięcznej sprzedaży i zwraca
29 // tę wartość jako liczbę typu Real
30 Function Real getSales()
31     // Zmienna lokalna, w której zapiszemy wartość sprzedaży
32     Declare Real monthlySales
33
34     // Pobieramy wartość sprzedaży
35     Display "Wprowadź wartość miesięcznej sprzedaży pracownika."
36     Input monthlySales
37
38     // Zwracamy wartość miesięcznej sprzedaży
39     Return monthlySales
40 End Function
41

```



Rysunek 6.11. Schemat blokowy modułu main

Zadaniem funkcji `getSales` jest pobranie od użytkownika wartości miesięcznej sprzedaży pracownika i zwrócenie tej wartości. W linii 32. deklaruję zmienną lokalną `monthlySales`. W linii 35. proszę użytkownika o wprowadzenie wartości sprzedaży, a w linii 36. pobieram wprowadzone dane i zapisuję je w zmiennej `monthlySales`. W linii 39. zwracam wartość przypisaną do zmiennej `monthlySales`. Na rysunku 6.12 przedstawiłem schemat blokowy tej funkcji.

Poniżej znajduje się definicja funkcji `getAdvancedPay`.

#### **Listing 6.8 Program do obliczania premii (kontynuacja): funkcja getAdvancedPay**

```

42 // Funkcja getAdvancedPay pobiera wartość zaliczki
43 // wyplacanej danemu pracownikowi
44 // i zwraca ją jako liczbę typu Real
45 Function Real getAdvancedPay()
46     // Zmienna lokalna, w której zapiszemy wartość zaliczki
47     Declare Real advanced
48

```

```

49 // Pobieramy wartość wypłaconej zaliczki
50 Display "Wprowadź wartość wypłaconej zaliczki lub"
51 Display "0, jeżeli zaliczka nie została wypłacona."
52 Input advanced
53
54 // Zwracamy wartość zaliczki
55 Return advanced
56 End Function
57

```



Rysunek 6.12. Schemat blokowy funkcji getSales

Zadaniem funkcji getAdvancedPay jest pobranie od użytkownika wartości zaliczki wypłaconej pracownikowi i zwrocie tej wartości. W linii 47. deklaruję zmienną lokalną advanced. W liniach 50. i 51. proszę użytkownika o wprowadzenie wartości wypłaconej zaliczki (lub wpisanie 0, jeżeli zaliczka nie została wypłacona), a w linii 52. pobieram wprowadzone dane i zapisuję je w zmiennej advanced. W linii 55. zwracam wartość przypisaną do zmiennej advanced. Na rysunku 6.13 przedstawiłem schemat blokowy tej funkcji. Poniżej zamieściłem definicję funkcji determineCommissionRate.

#### **Listing 6.8 Program do obliczania premii (kontynuacja): funkcja determineCommissionRate**

```

58 // Funkcja determineCommissionRate przyjmuje jako argument
59 // wartość sprzedaży i zwraca
60 // wysokość premii jako liczbę typu Real
61 Function Real determineCommissionRate(Real sales)
62     // Zmienna lokalna, w której zapiszemy wysokość premii
63     Declare Real rate

```

```

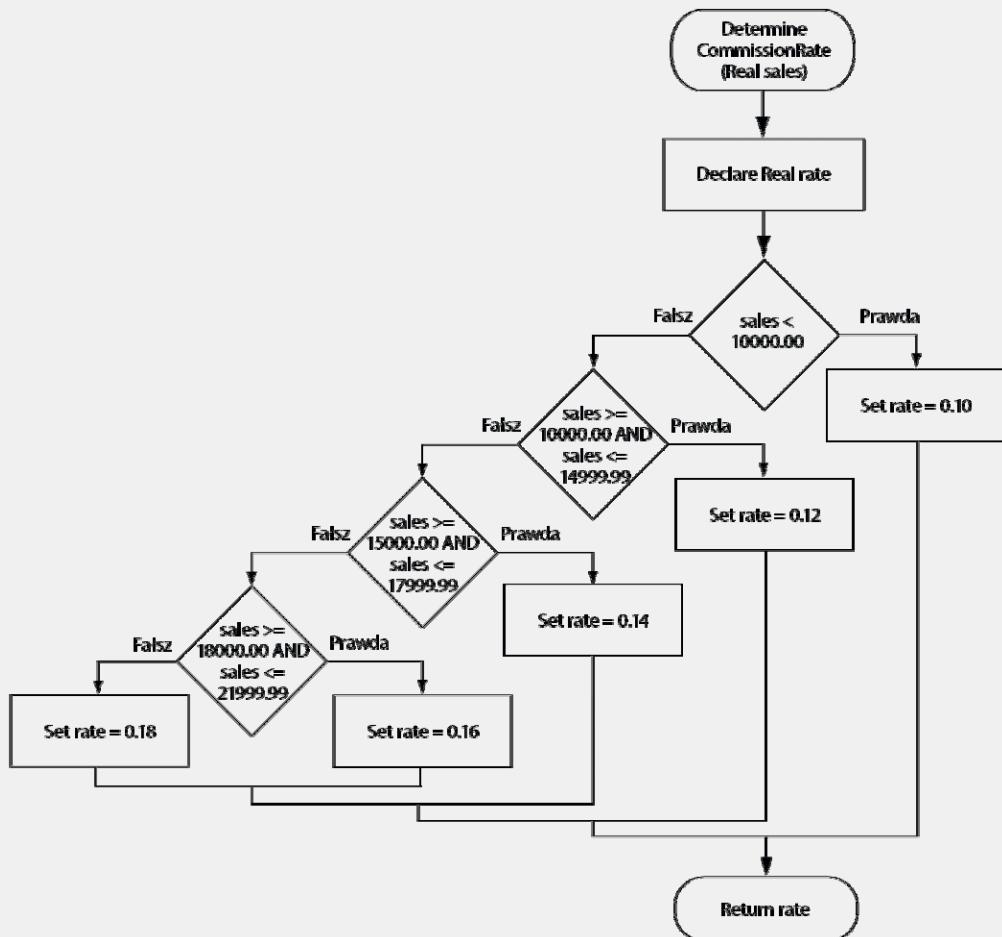
64 // Określamy wysokość premii
65 If sales < 10000.00 Then
66     Set rate = 0.10
67 Else If sales >= 10000.00 AND sales <= 14999.99 Then
68     Set rate = 0.12
69 Else If sales >= 15000.00 AND sales <= 17999.99 Then
70     Set rate = 0.14
71 Else If sales >= 18000.00 AND sales <= 21999.99 Then
72     Set rate = 0.16
73 Else
74     Set rate = 0.18
75 End If
77
78 // Zwracamy wysokość premii
79 Return rate
80 End Function

```



**Rysunek 6.13.** Schemat blokowy funkcji getAdvancedPay

Funkcja determineCommissionRate przyjmuje jako argument wartość sprzedaży i zwraca wysokość premii, która jest zależna od wysokości sprzedaży. W linii 63. deklaruję zmienną lokalną rate. Instrukcja If-Then-Else If w liniach od 66. do 76. sprawdza wartość parametru sales i przypisuje do zmiennej rate odpowiednią wysokość premii. W linii 79. zwracam wartość przypisaną do zmiennej rate. Na rysunku 6.14 przedstawiłem schemat blokowy tej funkcji.

**Rysunek 6.14.** Schemat blokowy funkcji**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wartość miesięcznej sprzedaży pracownika.  
**14650.00 [Enter]**

Wprowadź wartość wypłaconej zaliczki lub  
0, jeżeli zaliczka nie została wypłacona.  
**1000.00 [Enter]**

Wynagrodzenie wynosi 758.00 zł.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wartość miesięcznej sprzedaży pracownika.  
**9000.00 [Enter]**

Wprowadź wartość wypłaconej zaliczki lub  
0, jeżeli zaliczka nie została wypłacona.  
**0 [Enter]**

Wynagrodzenie wynosi 900.00 zł.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wartość miesięcznej sprzedaży pracownika.

**12000.00 [Enter]**

Wprowadź wartość wypłaconej zaliczki lub  
0, jeżeli zaliczka nie została wypłacona.

**2000.00 [Enter]**

Wynagrodzenie wynosi -560 zł.

Pracownik musi zwrócić  
 pieniądze firmie.

## Zwracanie ciągów znaków

Dotychczas przedstawiłem tylko przykłady funkcji, które zwracały liczbę. W większości języków programowania można także tworzyć funkcje, które zwracają ciąg znaków. Przykładowo funkcja przedstawiona poniżej prosi użytkownika o wprowadzenie imienia, a następnie zwraca ciąg, który wprowadził użytkownik:

```
Function String getName()
    // Zmienna lokalna, w której zapiszemy imię
    Declare String name

    // Pobieramy imię
    Display "Jak masz na imię?"
    Input name

    // Zwracamy imię
    Return name
End Function
```

## Zwracanie wartości typu Boolean

Większość języków programowania umożliwia także tworzenie funkcji boolowskich, które zwracają wartość **True** (prawda) lub **False** (fałsz). Za pomocą takiej funkcji można sprawdzić warunek i zwrócić wynik **True** lub **False** w zależności od tego, czy dany warunek jest spełniony, czy nie. Dzięki funkcjom boolowskim można znacznie uprościć warunki, które są sprawdzane w strukturach warunkowych i cyklicznych.

Załóżmy, że musimy stworzyć program, który będzie prosił użytkownika o wprowadzenie liczby, a następnie sprawdzał, czy dana liczba jest parzysta czy nieparzysta. Poniżej pokazałem, jak można to sprawdzić w pseudokodzie. Założymy, że zmienna **number** jest typu **Integer** i zawiera liczbę, którą wprowadził użytkownik.

```
If number MOD 2 == 0 Then
    Display "Liczba jest parzysta."
Else
    Display "Liczba jest nieparzysta."
End If
```

Przyjrzyjmy się z bliska, co sprawdza tak naprawdę instrukcja **If-Then**, ponieważ nie jest to takie oczywiste:

```
number MOD 2 == 0
```

W wyrażeniu pojawia się operator MOD, który opisałem w rozdziale 2. Przypominam, że operator MOD dzieli przez siebie dwie liczby i zwraca resztę z dzielenia. Tak więc instrukcję tę można rozumieć tak: „Jeśli reszta z dzielenia liczby number przez liczbę 2 jest równa 0, wyświetl komunikat, że liczba jest parzysta, w przeciwnym razie wyświetl komunikat, że liczba jest nieparzysta”.

Warunek taki zadziała poprawnie, ponieważ reszta z dzielenia liczby parzystej przez 2 jest zawsze równa 0. Pseudokod byłby jednak bardziej zrozumiały, gdybyśmy w jakiś sposób mogli go zapisać tak: „Jeśli liczba jest parzysta, wyświetl komunikat, że liczba jest parzysta, a w przeciwnym wypadku wyświetl komunikat, że liczba jest nieparzysta”. Jak się okazuje, można to zrobić za pomocą funkcji boolowskiej. W tym przypadku możemy zaprojektować funkcję boolowską o nazwie `isEven`, która będzie przyjmowała jako argument liczbę i zwracała wartość `True`, jeśli liczba ta jest parzysta, lub wartość `False`, jeśli liczba jest nieparzysta. Na poniższym pseudokodzie przedstawiłem taką funkcję:

```
Function Boolean isEven(Integer number)
    // Zmienna lokalna, w której będzie zapisana wartość True lub False
    Declare Boolean status

    // Sprawdzamy, czy liczba jest parzysta. Jeśli tak, ustawiamy wartość
    // zmiennej status na True, w przeciwnym razie ustawiamy na False
    If number MOD 2 == 0 Then
        Set status = True
    Else
        Set status = False
    End If

    // Zwracana wartość jest zapisana w zmiennej status
    Return status
End Function
```

Możemy teraz ponownie napisać instrukcję If-Then i wywołać funkcję `isEven`, która sprawdzi, czy liczba jest parzysta:

```
If isEven(number) Then
    Display "Liczba jest parzysta."
Else
    Display "Liczba jest nieparzysta."
End If
```

Nie tylko łatwiej jest taki program zrozumieć, ale masz także do dyspozycji funkcję, której możesz użyć w dowolnym miejscu programu, kiedy będziesz musiał sprawdzić, czy dana liczba jest parzysta.



## Punkt kontrolny

- 6.6. Do czego służy polecenie `Return`?
- 6.7. Spójrz na następującą definicję funkcji:

```
Function Integer doSomething(Integer number)
    Return number * 2
End Function
```

- Jak nazywa się ta funkcja?
- Jakiego typu dane zwraca ta funkcja?
- Co wyświetli się po wykonaniu poniższej instrukcji?

```
Display doSomething(10)
```

- 6.8. Co to jest funkcja boolowska?

## 6.3

# Inne funkcje biblioteczne



**UWAGA:** Funkcje biblioteczne, które zaprezentuję w tym podrozdziale, to uogólnione wersje funkcji, które można znaleźć w większości języków programowania. Nazwy funkcji, argumenty, które przyjmują, i ich działanie mogą się nieco różnić względem funkcji dostępnych w danym języku programowania.

## Funkcje matematyczne

W wielu językach programowania wśród funkcji bibliotecznych można znaleźć kilka przydatnych funkcji matematycznych. Przyjmują one zazwyczaj jeden lub kilka argumentów, wykonują określona operację matematyczną i zwracają jej wynik. Przykładowo dwiema popularnymi funkcjami matematycznymi są funkcje `sqrt` i `pow`. Przyjrzymy się im bliżej.

### Funkcja `sqrt`

Funkcja `sqrt` przyjmuje jako argument liczbę i zwraca pierwiastek kwadratowy tej liczby. Oto przykład jej wywołania:

```
Set result = sqrt(16)
```

W instrukcji tej wywołuję funkcję `sqrt` i przekazuję do niej liczbę 16. Funkcja zwraca pierwiastek kwadratowy liczby 16, a następnie przypisuje tę wartość do zmiennej `result`. Na listingu 6.9 pokazałem pseudokod, w którym wywołuję funkcję `sqrt`.

### Listing 6.9

```

1 // Deklaracja zmiennych
2 Declare Integer number
3 Declare Real squareRoot
4
5 // Pobieramy liczbę
6 Display "Wprowadź liczbę."
7 Input number
8
9 // Obliczamy i wyświetlamy pierwiastek kwadratowy liczby
10 Set squareRoot = sqrt(number)
11 Display "Pierwiastek kwadratowy jest równy ", squareRoot

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę.

**25 [Enter]**

Pierwiastek kwadratowy jest równy 5

Pseudokod na listingu 6.10 służy do obliczania długości przeciwnostokątnej trójkąta prostokątnego. W programie korzystam z następującego wzoru, znanego Ci zapewne z zajęć z trygonometrii:

$$c = \sqrt{a^2 + b^2}$$

We wzorze tym  $c$  oznacza długość przeciwnostokątnej, a  $a$  i  $b$  są długościami przyprostokątnych.

**Listing 6.10**

```

1 // Deklaracja zmiennych
2 Declare Real a, b, c
3
4 // Pobieramy długość przyprostokątnej A
5 Display "Wprowadź długość przyprostokątnej A."
6 Input a
7
8 // Pobieramy długość przyprostokątnej B
9 Display "Wprowadź długość przyprostokątnej B."
10 Input b
11
12 // Obliczamy długość przeciwprostokątnej
13 Set c = sqrt(a^2 + b^2)
14
15 // Wyświetlamy długość przeciwprostokątnej
16 Display "Długość przeciwprostokątnej wynosi ", c

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź długość przyprostokątnej A.

**5.0 [Enter]**

Wprowadź długość przyprostokątnej B.

**12.0 [Enter]**

Długość przeciwprostokątnej wynosi 13

Przyjrzyj się linii 13.:

```
Set c = sqrt(a^2 + b^2)
```

Instrukcja ta działa następująco: obliczana jest wartość wyrażenia  $a^2 + b^2$  i przekazywana do funkcji `sqrt` jako argument. Funkcja `sqrt` oblicza pierwiastek kwadratowy wartości przekazanej przez argument i przypisuje go do zmiennej  $c$ .

**Funkcja pow**

Inną popularną funkcją matematyczną jest `pow`. Służy ona do podnoszenia liczby do potęgi. Krótko mówiąc, działa tak samo jak operator `^`. W niektórych językach programowania nie istnieje operator służący do potęgowania — zamiast tego dostępna jest funkcja taka jak `pow`. Oto przykład wykorzystania funkcji `pow`:

```
Set area = pow(4, 2)
```

W instrukcji wywołuję funkcję pow i przekazuję do niej argumenty 4 i 2. Funkcja zwraca wartość równą liczbie 4 podniesionej do drugiej potęgi, a następnie wartość ta jest przypisywana do zmiennej area.

### Inne popularne funkcje matematyczne

W tabeli 6.2 przedstawiłem kilka innych funkcji matematycznych, które można znaleźć w większości języków programowania.

**Tabela 6.2.** Inne popularne funkcje matematyczne

| Nazwa funkcji | Opis i przykład użycia                                                                                                                                                                                                                                                                                      |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| abs           | Zwraca wartość bezwzględną liczby przekazanej jako argument.<br><br><i>Przykład:</i> Po wykonaniu poniższego polecenia w zmiennej y będzie się znajdowała wartość bezwzględna liczby w zmiennej x. Zmienna x pozostanie niezmieniona.<br><br>$y = \text{abs}(x)$                                            |
| cos           | Zwraca cosinus liczby przekazanej jako argument. Argument powinien być miarą kąta wyrażoną w radianach.<br><br><i>Przykład:</i> Po wykonaniu poniższego polecenia w zmiennej y będzie się znajdowała wartość cosinusa kąta zapisanego w zmiennej x. Zmienna x pozostanie niezmieniona.<br><br>$y = \cos(x)$ |
| round         | Przyjmuje jako argument liczbę rzeczywistą i zwraca wartość tej liczby zaokrągloną do najbliższej liczby całkowitej. Przykładowo round(3.5) zwróci 4, a round(3.2) zwróci 3.<br><br><i>Przykład:</i><br><br>$y = \text{round}(x)$                                                                           |
| sin           | Zwraca sinus liczby przekazanej jako argument. Argument powinien być miarą kąta wyrażoną w radianach.<br><br><i>Przykład:</i> Po wykonaniu poniższego polecenia w zmiennej y będzie się znajdowała wartość sinusa kąta zapisanego w zmiennej x. Zmienna x pozostanie niezmieniona.<br><br>$y = \sin(x)$     |
| tan           | Zwraca tangens liczby przekazanej jako argument. Argument powinien być miarą kąta wyrażoną w radianach.<br><br><i>Przykład:</i> Po wykonaniu poniższego polecenia w zmiennej y będzie się znajdowała wartość tangensa kąta zapisanego w zmiennej x. Zmienna x pozostanie niezmieniona.<br><br>$y = \tan(x)$ |

## Funkcje do konwertowania typów danych

W większości języków programowania można także znaleźć funkcje biblioteczne służące do konwertowania typów danych. Przykładowo dostępna jest funkcja, która zamienia liczbę rzeczywistą na całkowitą, lub funkcja, która zamienia liczbę całkowitą na liczbę rzeczywistą. W tej książce do zamiany liczby rzeczywistej na całkowitą będę się posługiwał funkcją `toInteger`, a do zamiany liczby całkowitej na liczbę rzeczywistą — funkcją `toReal`. Funkcje te opisalem w tabeli 6.3.

**Tabela 6.3.** Funkcje do konwertowania typów danych

| Funkcja                | Opis i przykład użycia                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>toInteger</code> | Funkcja <code>toInteger</code> przyjmuje jako argument liczbę rzeczywistą i zwraca ją w postaci liczby całkowitej. Jeżeli liczba rzeczywista ma część ułamkową, zostanie ona odrzucona. Przykładowo wywołanie <code>toInteger(2.5)</code> zwróci wartość 2.<br><br>Przykład: Jeżeli poniższy kod byłby prawdziwym programem, po wykonaniu polecenia w zmiennej <code>i</code> znalazłaby się liczba 2.<br><pre>Declare Integer i Declare Real r = 2.5 Set i = toInteger(r)</pre> |
| <code>toReal</code>    | Funkcja <code>toReal</code> przyjmuje jako argument liczbę całkowitą i zwraca ją w postaci liczby rzeczywistej.<br><br>Przykład: Jeżeli poniższy kod byłby prawdziwym programem, po wykonaniu polecenia w zmiennej <code>r</code> znalazłaby się liczba 2.<br><pre>Declare Integer i = 7 Declare Real r Set r = toReal(i)</pre>                                                                                                                                                  |

Jeżeli spróbujesz przypisać do zmiennej jednego typu wartość innego typu, w wielu językach programowania wystąpi błąd. Spójrz na ten przykładowy pseudokod:

```
Declare Integer number
Set number = 6.17 ← W wielu językach programowania tutaj pojawi się błąd!
```

W pierwszej linii deklaruję zmienną typu `Integer` o nazwie `number`. W drugiej linii próbuję do niej przypisać wartość `6.17`, która jest liczbą rzeczywistą. W większości języków programowania wystąpi w tym miejscu błąd, ponieważ zmienna typu całkowitego nie jest w stanie zapisać części dziesiętnej liczby. Błąd ten często jest nazywany **błędem niezgodności typów**.



**UWAGA:** W wielu językach można przypisywać liczbę całkowitą do zmiennej typu rzeczywistego, ponieważ taka operacja nie powoduje utraty danych. Gdybyś jednak chciał jawnie zapisać taką operację, nadal możesz skorzystać z funkcji do konwersji typów danych.

W pewnych sytuacjach możesz nieumyślnie zapisać instrukcję, która spowoduje powstanie błędu niezgodności typów. Przyjrzyj się przykładowi z listingu 6.11. Program ten służy do obliczania liczby osób, którym można podać lemoniadę, mając do dyspozycji określoną jej ilość.

### **Listing 6.11**

```

1 // Deklarujemy zmienną, w której zapiszemy
2 // liczbę uncji lemoniady, którą dysponujemy
3 Declare Real ounces
4
5 // Deklarujemy zmienną, w której zapiszemy
6 // liczbę osób, którym chcemy podać lemoniadę
7 Declare Integer numberofPeople
8
9 // Stała zawierająca liczbę uncji lemoniady przypadającą na jedną osobę
10 Constant Integer OUNCES_PER_PERSON = 8
11
12 // Pobieramy liczbę uncji lemoniady
13 Display "Ile jest uncji lemoniady?"
14 Input ounces
15
16 // Obliczamy liczbę osób, którym możemy podać lemoniadę
17 Set numberofPeople = ounces / OUNCES_PER_PERSON ←———— Błąd!
18
19 // Wyświetlamy liczbę osób, którym możemy podać lemoniadę
20 Display "Możesz podać lemoniadę ", numberofPeople, " osobom."

```

W linii 3. deklaruję zmienną `ounces`, w której zapiszę dostępną liczbę uncji lemoniady, a w linii 7. deklaruję zmienną `numberofPeople`, w której zapiszę liczbę osób, którym będzie można podać lemoniadę. W linii 10. inicjalizuję stałą `OUNCES_PER_PERSON` wartością 8. Stała ta wskazuje, że każda osoba dostanie 8 uncji lemoniady.

Gdy już pobiorę od użytkownika i zapiszę w zmiennej `ounces` ilość lemoniady (w linii 14.), w poleceniu w linii 17. próbuję obliczyć, ilu osobom będzie można podać lemoniadę. Jednak pojawia się tutaj pewien problem: zmienna `numberofPeople` jest zmienną typu `Integer`, a wyrażenie matematyczne `ounces / OUNCES_PER_PERSON` najprawdopodobniej zwróci wynik w postaci liczby rzeczywistej (np. jeżeli w zmiennej `ounces` znajdzie się liczba 12, wynik będzie równy 1,5). Kiedy polecenie będzie chciało przypisać do zmiennej `numberofPeople` wynik zwrócony przez wyrażenie matematyczne, pojawi się błąd.

Pierwsza myśl, która przychodzi do głowy w takiej sytuacji, jest taka, że być może wystarczy zmienić typ zmiennej `numberofPeople` na `Real`. Dzięki temu błąd się nie pojawi, jednak zapisywanie liczby osób w zmiennej typu `Real` nie ma sensu. Nie da się przecież podać lemoniady ułamkowi osoby! Lepszym rozwiązaniem będzie skonwertowanie wartości wyrażenia `ounces / OUNCES_PER_PERSON` na liczbę całkowitą i dopiero potem przypisanie tej wartości do zmiennej `numberofPeople`. Takie podejście przedstawiłem na listingu 6.12.

**Listing 6.12**

```

1 // Deklarujemy zmienną, w której zapiszemy
2 // liczbę uncji lemoniady, którą dysponujemy
3 Declare Real ounces
4
5 // Deklarujemy zmienną, w której zapiszemy
6 // liczbę osób, którym chcemy podać lemoniadę
7 Declare Integer numberOfPeople
8
9 // Stała zawierająca liczbę uncji lemoniady przypadającą na jedną osobę
10 Constant Integer OUNCES_PER_PERSON = 8
11
12 // Pobieramy liczbę uncji lemoniady
13 Display "Ile masz uncji lemoniady?"
14 Input ounces
15
16 // Obliczamy liczbę osób, którym możemy podać lemoniadę
17 Set numberOfPeople = toInteger(ounces / OUNCES_PER_PERSON)
18
19 // Wyświetlamy liczbę osób, którym możemy podać lemoniadę
20 Display "Możesz podać lemoniadę ", numberOfPeople, " osobom."

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Ile masz uncji lemoniady?

165 [Enter]

Możesz podać lemoniadę 20 osobom.

W tej wersji programu zmodyfikowałem linię 17. do następującej postaci:

```
Set numberOfPeople = toInteger(ounces / OUNCES_PER_PERSON)
```

Zobaczmy, jak zadziałała ta instrukcja w przykładowym wywołaniu programu. Po uruchomieniu polecenia najpierw zostanie obliczone wyrażenie `ounces / OUNCES_PER_PERSON`. Użytkownik wprowadził do zmiennej `ounces` liczbę 165, więc wyrażenie zwróci wartość 20.625. Wartość ta następnie jest przekazywana jako argument do funkcji `toInteger`. Funkcja `toInteger` odrzuca część dziesiętną liczby (.625) i zwraca liczbę całkowitą równą 20. Do zmiennej `numberOfPeople` zostanie więc przypisana wartość 20.

Funkcja `toInteger` w każdym przypadku odrzuca część dziesiętną liczby przekazanej jako argument. W tym przykładowym programie jest to dopuszczalne, ponieważ chcemy obliczyć liczbę osób, którym możemy podać określoną ilość lemoniady. Część ułamkowa, która została odrzucona, wskazuje, ile uncji lemoniady pozostało.

## Funkcje formatujące

W większości języków programowania dostępna jest co najmniej jedna funkcja służąca do formatowania danych liczbowych. Popularną operacją tego typu jest wyświetlanie liczby w formacie walutowym. W tej książce będę się posługiwał funkcją `currencyFormat`, która przyjmuje jako argument liczbę typu `Real` i zwraca ciąg znaków zawierający tę liczbę zapisaną w formacie walutowym. Poniższy pseudokod przedstawia sposób, w jaki można wykorzystać funkcję `currencyFormat`:

```
Declare Real amount = 6450.879
Display currencyFormat(amount)
```

Gdyby ten kod był prawdziwym programem, w wyniku wyświetliłby się tekst:

6 450,88 zł

Zwróć uwagę, że funkcja zwraca także symbol waluty (w tym przypadku złoty), wstawia w odpowiednich miejscach przecinek i znak spacji i zaokrąglą liczbę do dwóch miejsc po przecinku.



**UWAGA:** Wiele współczesnych języków programowania wspiera **lokalizację**, czyli dostosowanie pewnych elementów do standardów obowiązujących w danym kraju. W językach tych funkcja podobna do `currencyFormat` może zwracać inny symbol walutowy.

## Funkcje do przetwarzania ciągów znaków

Wiele typów programów służy głównie do przetwarzania danych tekstowych. Takie programy jak Notatnik czy procesor tekstu Microsoft Word przeprowadzają operacje niemal wyłącznie na tekście. Przeglądarki internetowe podobnie — kiedy załączujesz w przeglądarce stronę, odczytuje ona instrukcje formatujące umieszczone w treści strony internetowej.

W większości języków programowania dostępnych jest wiele funkcji bibliotecznych przeznaczonych do przetwarzania ciągów znaków. Omówię kilka najbardziej popularnych funkcji.

### Funkcja `length`

Funkcja `length` zwraca długość ciągu znaków. Jako argument przyjmuje串 znaków, a zwraca liczbę znaków występujących w danym ciągu. Zwracaną wartością jest liczba typu `Integer`. Na poniższym pseudokodzie przedstawiłem sposób, w jaki można wykorzystać funkcję `length`. W kodzie sprawdzam, czy wprowadzone hasło ma co najmniej sześć znaków.

```
Display "Wprowadź nowe hasło."
Input password
If length(password) < 6 Then
    Display "Hasło musi się składać z co najmniej sześciu znaków."
End If
```

### Funkcja `append`

Funkcja `append` przyjmuje jako argumenty dwa ciągi znaków — nazwijmy je `string1` i `string2`. Zwraca ona trzeci串 znaków, który jest wynikiem dołączenia do ciągu `string1` ciągu `string2`. Po wywołaniu funkcji ciągi `string1` i `string2` pozostaną niezmienione. Oto, w jaki sposób można wykorzystać tę funkcję:

```

Declare String lastName = "Kamiński"
Declare String salutation = "Pan "
Declare String properName
Set properName = append(salutation, lastName)
Display properName

```

Gdyby ten kod był prawdziwym programem, w wyniku wyświetliłby się tekst:

Pan Kamiński



**UWAGA:** Operację dołączania jednego ciągu znaków do innego nazywamy **konkatenacją**.

### Funkcje `toUpperCase` i `toLowerCase`

Funkcje `toUpperCase` i `toLowerCase` zmieniają wielkość znaków w ciągu znaków. Funkcja `toUpperCase` przyjmuje jako argument ciąg znaków i zwraca串, który jest jego kopią, ale wszystkie znaki zostały w nim zamienione na duże. Niezmienione zostaną tylko duże znaki i znaki niebędące literami alfabetu. Oto, w jaki sposób można wykorzystać tę funkcję:

```

Declare String str = "Witaj, świecie!"
Display toUpper(str)

```

Gdyby ten kod był prawdziwym programem, w wyniku wyświetliłby się tekst:

WITAJ, ŚWIECIE!

Funkcja `toLowerCase` przyjmuje jako argument ciąg znaków i zwraca串, który jest jego kopią, ale wszystkie znaki zostały w nim zamienione na małe. Niezmienione zostaną tylko małe znaki i znaki niebędące literami alfabetu. Oto, w jaki sposób można wykorzystać tę funkcję:

```

Declare String str = "UWAGA!"
Display toLower(str)

```

Gdyby ten kod był prawdziwym programem, w wyniku wyświetliłby się tekst:

uwaga!

Funkcje `toUpperCase` i `toLowerCase` okazują się bardzo pomocne, gdy chcemy porównać ciągi znaków, bez względu na wielkość znaków. W normalnej sytuacji podczas porównywania ciągów znaków rozróżniane są duże i małe litery. Przykładowo串, który jest zupełnie inny od ciągu "WITAJ", ponieważ różni je wielkość liter. Jednak w pewnych sytuacjach podczas porównywania ciągów nie będziemy chcieli, aby wielkość liter miała znaczenie. W takim przypadku串, który powinien być traktowany jako identyczny z ciągiem znaków "WITAJ" lub "Witaj".

Spójrz na następujący przykład:

```

Declare String again
Do
  Display "Witaj!"
  Display "Czy chcesz wyświetlić tekst jeszcze raz? (T = Tak)"
  Input again
  While toUpper(again) == "T"

```

Pętla wyświetla napis *Witaj!*, a następnie pyta użytkownika, czy chce go wyświetlić jeszcze raz — jeśli tak, użytkownik powinien nacisnąć klawisz „t”. Wyrażenie `toUpperCase(again) == "T"` zwróci prawdę zarówno wtedy, gdy użytkownik wprowadzi „t”, jak i wtedy, gdy wpisze „T”. Podobny rezultat można uzyskać za pomocą funkcji `toLowerCase`:

```
Declare String again
Do
    Display "Witaj!"
    Display "Czy chcesz wyświetlić tekst jeszcze raz? (T = Tak)"
    Input again
    While toLower(again) == "t"
```

## Funkcja substring

Funkcja `substring` zwraca **podciąg**, czyli fragment innego ciągu znaków. Przyjmuje ona zazwyczaj trzy argumenty: (1) ciąg znaków, którego fragment chcesz zwrócić, (2) początkową pozycję w ciągu znaków i (3) końcową pozycję w ciągu znaków.

Każdy znak w ciągu można wskazać za pomocą liczby. Pierwszy znak ma numer 0, drugi znak ma numer 1 itd. W poniższym przykładzie zwracam fragment ciągu znaków „Nowy Sącz” zaczynający się od znaku 5., a kończący się na znaku 7.:

```
Declare String str = "Nowy Sącz"
Declare String search
Set search = substring(str, 5, 7)
Display search
```

Gdyby ten kod był prawdziwym programem, w wyniku wyświetliłby się tekst:

Sąc

Za pomocą funkcji `substring` można także zwracać poszczególne znaki ciągu znaków. Spójrz na ten przykład:

```
Declare String name = "Karol"
Display substring(name, 2, 2)
```

W wyniku pojawi się:

r

Wywołanie funkcji `substring(name, 2, 2)` zwróci fragment ciągu zaczynający się i kończący na znaku numer 2. W tym przypadku jest to ciąg „r”. Na listingu 6.13 przedstawiłem kolejny przykład. Program prosi użytkownika o wprowadzenie ciągu znaków, a następnie zlicza, ile razy w ciągu występuje litera „T”.

### **Listing 6.13**

```
1 // Deklarujemy zmienną, w której zapiszemy ciąg znaków
2 Declare String str
3
4 // Deklarujemy zmienną, w której zapiszemy, ile razy
5 // w ciągu pojawia się litera "T"
6 Declare Integer numTs = 0
7
8 // Deklaracja zmiennej licznikowej
9 Declare Integer counter
```

```

10
11 // Pobieramy od użytkownika ciąg znaków
12 Display "Wprowadź ciąg znaków."
13 Input str
14
15 // Zliczamy, ile razy w ciągu występuje litera "T"
16 For counter = 0 To length(str)
17   If substring(str, counter, counter) == "T" Then
18     numTs = numTs + 1
19   End If
20 End For
21
22 // Wyświetlamy liczbę wystąpień litery "T"
23 Display "Ten ciąg znaków zawiera ", numTs
24 Display "litery T."

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź ciąg znaków.

**To bardzo Trudny Tekst! [Enter]**

Ten ciąg znaków zawiera 3

litery T.

**Funkcja contains**

Funkcja `contains` przyjmuje jako argumenty dwa ciągi znaków. Zwraca wartość `True`, jeżeli w pierwszym ciągu znaków występuje drugi ciąg znaków. W przeciwnym razie funkcja zwraca wartość `False`. Przykładowo na poniższym przykładzie sprawdzam, czy w ciągu znaków "Lat temu osiemdziesiąt i siedem" występuje słowo "siedem":

```

Declare string1 = "Lat temu osiemdziesiąt i siedem"
Declare string2 = "siedem"
If contains(string1, string2) Then
  Display string2, " występuje w ciągu znaków."
Else
  Display string2, " nie występuje w ciągu znaków."
End If

```

Gdyby ten kod był prawdziwym programem, w wyniku wyświetliłby się komunikat: `siedem występuje w ciągu znaków.`

**Funkcje `stringToInteger` i `stringToReal`**

Ciągi znaków składają się z szeregu znaków i służą do przechowywania różnych danych tekstowych, takich jak imię i nazwisko, adres, opis itp. Można w nich także zapisać liczbę. Jeżeli w programie umieścisz liczbę w cudzysłowie, stanie się ona ciągiem znaków. W tym przykładzie deklaruję zmienną typu `String` o nazwie `interestRate` i inicjalizuję ją wartością "4.3":

```
Declare String interestRate = "4.3"
```

W wyniku zapisywania liczby jako ciągu znaków mogą wyniknąć jednak problemy. Większości operacji, jakie przeprowadzamy na liczbach (np. działania matematyczne lub porównywanie liczb), nie można wykonać na ciągach znaków. Takie operacje są możliwe tylko na typach liczbowych, takich jak `Integer` lub `Real`.

W niektórych programach jednak dane będą pozyskiwane ze źródeł, w których liczby zapisane są jako ciągi znaków. Dzieje się tak często podczas odczytywania danych z plików. Ponadto w niektórych językach można odczytać dane z klawiatury tylko jako ciągi znaków. W takich przypadkach dane liczbowe trafiają do programu jako ciągi znaków i należy je skonwertować na wartości liczbowe.

W większości języków programowania dostępne są funkcje biblioteczne służące do zamiany ciągów znaków na liczby. W przykładach zamieszczonych poniżej zaprezentowałem funkcje `stringToInteger` i `stringToReal`. Funkcja `stringToInteger` przyjmuje jako argument ciąg znaków, a następnie zamienia go na liczbę typu `Integer` i zwraca tę wartość. Założmy, że w naszym programie jest zmienna `str` typu `String` i że przypisana jest do niej liczba całkowita zapisana jako ciąg znaków. Ta oto instrukcja zamienia ciąg znaków zapisany w zmiennej `str` na liczbę typu `Integer` i przypisuje ją do zmiennej `intNumber`:

```
Set intNumber = stringToInteger(str)
```

Funkcja `stringToReal` działa w podobny sposób, z tą różnicą, że zamienia ciąg znaków na wartość typu `Real`. Założmy, że w zmiennej `str` typu `String` zapisana jest jako ciąg znaków liczba rzeczywista. Poniższa instrukcja zamienia ciąg znaków zapisany w zmiennej `str` na liczbę typu `Real` i przypisuje ją do zmiennej `realNumber`:

```
Set realNumber = stringToReal(str)
```

Kiedy będziesz korzystać z tego typu funkcji, zawsze istnieje ryzyko wystąpienia błędu. Przyjrzyj się przykładowemu pseudokodowi:

```
Set intNumber = stringToInteger("123abc")
```

Oczywiście ciąg znaków "123abc" nie da się zamienić na liczbę typu `Integer`, ponieważ zawiera on litery. Oto inny przykład, w którym pojawi się błąd:

```
Set realNumber = stringToReal("3.14.159")
```

Ciągu znaków "3.14.159" nie da się zamienić na liczbę rzeczywistą, gdyż zawiera dwa separatory dziesiętne. Co się stanie w takich przypadkach, zależy tylko i wyłącznie od języka programowania.

## Funkcje `isInteger` i `isReal`

Aby uniknąć błędów podczas konwertowania ciągu znaków na liczbę, wiele języków programowania udostępnia funkcje, które sprawdzają, czy ciąg znaków da się zamienić na liczbę, i zwracają wartość `True` lub `False`. Poniżej pokazałem, w jaki sposób za pomocą funkcji `isInteger` i `isReal` można sprawdzić, czy ciąg znaków da się zamienić odpowiednio na liczbę typu `Integer` lub na liczbę typu `Real`. W tym przykładzie zaprezentowałem funkcję `isInteger`. Założmy, że zmienna `str` jest typu `String`, a zmienna `intNumber` jest typu `Integer`:

```
If isInteger(str) Then
    Set intNumber = stringToInteger(str)
Else
    Display "Błędne dane"
End If
```

Funkcja `isReal` działa analogicznie. Założymy, że zmienna `str` jest typu `String`, a zmienna `realNumber` jest typu `Real`:

```
If isReal(str) Then
    Set realNumber = stringToReal(str)
Else
    Display "Błędne dane"
End If
```

## 6.4

## Rzut oka na języki Java, Python i C++

W niniejszym podręczniku omówię sposoby implementowania w językach programowania Java, Python i C++ wielu zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

## Java

### Metody zwracające wartości

#### **Generowanie losowych liczb całkowitych w języku Java**

Aby wygenerować liczby losowe w Javie, należy utworzyć w pamięci obiekt nazywany obiektem `Random`, a następnie użyć go do wygenerowania kolejnych liczb losowych. Najpierw, na początku swojego programu, trzeba wpisać następującą instrukcję:

```
import java.util.Random;
```

Następnie w miejscu, w którym chcesz utworzyć obiekt `Random`, musisz umieścić poniższą instrukcję:

```
Random randomNumbers = new Random();
```

Instrukcja ta tworzy obiekt `Random` w pamięci i nadaje mu nazwę `randomNumbers`. (Możesz nadać obiektowi dowolną nazwę. W tym przypadku wybierzemy nazwę `randomNumbers`). Po utworzeniu obiektu `Random` można wywołać w nim metodę o nazwie `nextInt` zwracającą losową liczbę całkowitą. Metoda `nextInt` obiektu `Random` jest podobna do funkcji `random` biblioteki omawianej w niniejszej książce. Poniższy fragment kodu pokazuje przykład jego użycia:

```
int number;
number = randomNumbers.nextInt(10);
```

W tym przykładzie do metody `randomNumbers.nextInt` przekazujemy argument o wartości 10. Sprawia to, że metoda zwraca losową liczbę całkowitą z przedziału od 0 do 9. Argument, który przekazujemy do metody, stanowi górną granicę przedziału, jednocześnie nie zawierając się w tym przedziale. W poniższym kodzie przypisujemy losową liczbę do zmiennej `number`. Oto przykład, w jaki sposób wygenerowalibyśmy losową liczbę zawartą w przedziale od 1 do 100:

```
int number;
number = randomNumbers.nextInt(100) + 1;
```

W tym przykładzie do metody `randomNumbers.nextInt` przekazujemy argument o wartości 100, generując tym samym losową liczbę całkowitą z przedziału od 0 do 99. Następnie do uzyskanej wartości dodajemy 1, co spowoduje, że znajdzie się ona w przedziale od 1 do 100. Na koniec wynik tych działań przypisywany jest do zmiennej `number`.

### **Tworzenie własnych metod zwracających wartości w języku Java**

Podczas tworzenia metody zwracającej wartość trzeba zdecydować, jaki typ wartości będzie ona zwracała. Po prostu w nagłówku metody zamiast podawać parametr `void` należy określić typ danych zwracanej wartości. Metoda zwracająca wartość będzie używała w nagłówku zmiennych typu `int`, `double`, `String`, `boolean` lub innego poprawnego typu danych. Oto przykład metody, która zwraca wartość typu `int`:

```
public static int sum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    return result;
}
```

Nazwa tej metody to `sum`. Zwróć uwagę, że w nagłówku metody zamiast słowa `void` wpisujemy słowo `int` oznaczające zwracany typ. Ten przykład kodu definiuje metodę o nazwie `sum`, która przyjmuje dwa argumenty typu `int`. Argumenty przekazywane są do zmiennych parametrów `num1` i `num2`. Wewnątrz metody deklarowana jest zmienna lokalna o nazwie `result`. Zmienne parametrów `num1` i `num2` są dodawane do siebie, a ich suma zostaje przypisana do zmiennej `result`. Ostatni wiersz tej metody wygląda tak:

```
return result;
```

Umieszczenie instrukcji `return` w metodzie zwracającej wartość powoduje, że metoda kończy swoje działanie i zwraca wartość do instrukcji, która ją wywołała. W metodzie zwracającej wartość ogólny format instrukcji `return` jest następujący:

```
return Wyrażenie;
```

*Wyrażenie* to wartość, która ma zostać zwrócona. Może to być dowolne wyrażenie, które ma jakąś wartość, na przykład zmienna, literal lub wyrażenie matematyczne. W tym przypadku metoda `sum` zwraca wartość zmiennej `result`. Poniższy fragment kodu pokazuje, jak można wywołać metodę `sum`:

```
int total;
total = sum(2, 3);
```

Po wykonaniu tych instrukcji w zmiennej `total` znajdzie się wartość 5.

### **Zwracanie ciągów znaków przez metodę w języku Java**

Poniższy kod pokazuje, w jaki sposób metoda może zwrócić ciąg znaków. Zauważ, że nagłówek metody podaje typ `String` jako typ zwracany. Ta metoda jako argumenty przyjmuje dwa ciągi znaków (imię i nazwisko osoby). Otrzymane ciągi łączy następnie w jeden, składający się z imienia i nazwiska osoby. Na koniec zwracany jest uzyskany w ten sposób ciąg znaków.

```
public static String fullName(String firstName, String lastName)
{
    String name;

    name = firstName + " " + lastName;
    return name;
}
```

Poniższy fragment kodu pokazuje, jak można wywołać tę metodę:

```
String customerName;
customerName = fullName("Jaś", "Fasola");
```

Po wykonaniu tych instrukcji wartością zmiennej `customerName` będzie „Jaś Fasola”.

### Zwracanie wartości logicznych przez metodę w języku Java

Metody mogą również zwracać wartości typu `boolean`. Poniższa metoda przyjmuje argument i zwraca wartość `true`, jeśli wartość tego argumentu znajduje się w przedziale od 1 do 100, a w przeciwnym przypadku zwraca wartość `false`:

```
public static boolean isValid(int number)
{
    boolean status;

    if (number >= 1 && number <= 100)
        status = true;
    else
        status = false;
    return status;
}
```

Oto instrukcja `if-else`, w ramach której wywołana jest ta metoda:

```
int value = 20;
if (isValid(value))
    System.out.println("Wartość znajduje się w przedziale.");
else
    System.out.println("Wartość znajduje się poza przedziałem.");
```

Po uruchomieniu tego kodu wyświetli się komunikat „Wartość znajduje się w przedziale”.

## Python

### Funkcje zwracające wartość

#### Generowanie losowych liczb całkowitych w języku Python

Do pracy z liczbami losowymi Python udostępnia kilka funkcji bibliotecznych. W celu skorzystania z którejkolwiek z tych funkcji musimy na początku programu umieścić instrukcję `import`:

```
import random
```

Pierwsza funkcja generowania liczb losowych, którą tutaj omówimy, nosi nazwę `random.randint`. Poniższa instrukcja pokazuje, w jaki sposób można wywołać tę funkcję.

```
number = random.randint(1, 100)
```

Wywołaniem funkcji zajmuje się instrukcja `random.randint(1, 100)`. Zauważ, że w nawiasach pojawiają się dwa argumenty: 1 i 100. Informują one funkcję, że ma zwrócić losową liczbę całkowitą z przedziału od 1 do 100. (Wartości 1 i 100 są zawarte w tym przedziale.) Gdy funkcja jest wywoływana, generowana jest liczba losowa z przedziału od 1 do 100, a następnie jest ona **zwracana** do instrukcji wywołującej. Otrzymana w ten sposób liczba zostanie przypisana do zmiennej `number`.

### **Tworzenie własnych funkcji zwracających wartość w języku Python**

W Pythonie funkcję zwracającą wartość tworzy się w taki sam sposób jak prostą funkcję. Jest jednak pewna różnica: funkcja zwracająca wartość musi mieć instrukcję `return`. Oto ogólny format definicji funkcji zwracającej wartość w Pythonie:

```
def nazwa_funkcji():
    instrukcja
    instrukcja
    itd.
    return wyrażenie
```

Jedną z instrukcji funkcji musi być instrukcja `return`, która przyjmuje następującą postać:

```
return wyrażenie
```

Wartość *wyrażenia* następującego po słowie kluczowym `return` zostanie zwrócona do tej części programu, która tę funkcję wywołała. Może to być dowolna wartość, zmienna lub wyrażenie mające wartość (na przykład wyrażenie matematyczne). Oto prosty przykład funkcji zwracającej wartość:

```
def sum(num1, num2):
    result = num1 + num2
    return result
```

Zadaniem tej funkcji jest przyjęcie dwóch wartości całkowitych jako argumentów i zwrócenie ich sumy. Pierwsza instrukcja w bloku funkcji przypisuje wartość wyrażenia `num1 + num2` do zmiennej `result`. Następnie wykonywana jest instrukcja `return`, co powoduje, że funkcja kończy działanie i wysyła wartość zmiennej `result` z powrotem do części programu, która tę funkcję wywołała. Poniższa instrukcja pokazuje, jak możemy wywołać funkcję `sum`:

```
total = sum(2, 3)
```

Po wykonaniu tej instrukcji w zmiennej `total` znajdzie się wartość 5.

### **Zwracanie ciągu znaków przez metodę w języku Python**

Do tej pory przedstawiłem przykłady funkcji zwracających liczby. Oprócz tego można też tworzyć funkcje zwracające ciągi znaków. Na przykład poniższa funkcja prosi użytkownika o podanie imienia, a następnie zwraca串 znaków wprowadzony przez użytkownika:

```
def get_name():
    # Pobranie imienia użytkownika
    name = input('Jak masz na imię? ')

    # Zwrócenie imienia
    return name
```

### Zwracanie wartości logicznej przez metodę w języku Python

Python umożliwia także tworzenie **funkcji logicznych**, które zwracają wartość `True` lub `False`. Na przykład poniższa funkcja przyjmuje liczbę jako argument i zwraca wartość `True`, jeśli argument to liczba parzysta, a w przeciwnym wypadku zwraca wartość `False`:

```
def is_even(number):
    # Sprawdzenie, czy liczba jest parzysta
    # Jeżeli tak, to zmiennej status przypisywana jest wartość True
    # W przeciwnym przypadku zmieniąca status otrzymuje wartość False
    if (number % 2) == 0:
        status = True
    else:
        status = False

    # Zwrócenie wartości ze zmiennej status
    return status
```

Poniższy kod prosi użytkownika o wprowadzenie liczby, a następnie wywołuje funkcję, aby ustalić, czy ta liczba jest parzysta czy nieparzysta:

```
value = int(input('Podaj liczbę: '))
if is_even(value):
    print('Liczba jest parzysta.')
else:
    print('Liczba jest nieparzysta.')
```

## C++

### Funkcje zwracające wartość

#### Generowanie losowych liczb całkowitych w języku C++

Biblioteka języka C++ zawiera funkcję o nazwie `rand()`, która zwraca liczbę losową. (Funkcja `rand()` wymaga użycia dyrektywy `#include <cstdlib>`). Liczby losowe zwarcane przez funkcję `rand()` mają typ `int`. Oto przykład jej użycia:

```
y = rand();
```

Po wykonaniu tej instrukcji zmienna `y` będzie zawierała wylosowaną liczbę. W rzeczywistości numery tworzone przez funkcję `rand()` są pseudolosowe. Funkcja wykorzystuje algorytm, który generuje tę samą sekwencję liczb za każdym razem, gdy program jest uruchamiany w tym samym systemie. Na przykład założymy, że program wykona poniższe instrukcje:

```
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

Trzy wyświetcone liczby wydają się losowe, ale za każdym razem, gdy program zostanie uruchomiony, zostaną wygenerowane te same trzy wartości. W celu uzyskania wyników będących wartościami losowymi funkcji `rand()` należy użyć funkcji `srand()`. Funkcja `srand()` przyjmuje argument typu `unsigned int`, który używany jest jako wartość początkowa dla tego algorytmu. Podając różne wartości początkowe, sprawiamy, że funkcja `rand()` będzie generowała różne sekwencje liczb losowych.

Powszechną praktyką uzyskiwania unikatowych wartości początkowych jest wywoływanie funkcji `time()`, która jest częścią standardowej biblioteki języka C++. Funkcja `time()` zwraca liczbę sekund, które upłynęły od północy dnia 1 stycznia 1970 roku. Funkcja `time()` wymaga użycia dyrektywy `#include <ctime>`. Wywołując funkcję `time()`, należy podać jej w argumencie wartość 0. Oto przykład użycia tej funkcji:

```
// Pobranie czasu systemowego
unsigned seed = time(0);
// Wartość początkowa dla generatora liczb losowych
srand(seed);
// Wyświetlenie liczby losowej
cout << rand() << endl;
```

Jeśli chcesz ograniczyć zakres liczby losowej, użyj następującej formuły:

```
y = 1 + rand() % maxRange;
```

Zmienna `maxRange` definiuje górny limit zakresu generowanych liczb. Na przykład jeśli chcesz wygenerować liczbę losową z przedziału od 1 do 100, użyj następującej instrukcji:

```
y = 1 + rand() % 100;
```

Oto sposób działania tej instrukcji — spójrz na poniższe wyrażenie:

```
rand() % 100
```

Zakładając, że funkcja `rand()` zwróci liczbę 37894, wartość tego wyrażenia wyniesie 94. Dzieje się tak dlatego, że liczba 37894 podzielona przez 100 to 378 z resztą 94. (Operator dzielenia modulo zwraca resztę). Ale co się stanie, jeśli funkcja `rand()` zwróci liczbę, która będzie podzielna bez reszty przez 100, na przykład liczbę 500? Powyższe wyrażenie zwróci wartość 0. Jeśli chcemytrzymać liczbę z przedziału od 1 do 100, musimy do wyniku dodać 1. W tym celu użyjemy wyrażenia `1 + rand() % 100`.

### **Tworzenie własnych funkcji zwracających wartość w języku C++**

Kiedy tworzysz funkcję zwracającą wartość w C++, musisz zdecydować, jaki typ funkcja ma zwracać. Po prostu zamiast umieszczać w nagłówku funkcji słowo kluczowe `void` należy podać typ danych dla wartości, która zostanie zwrócona.

Funkcja zwracająca wartość może przyjąć w nagłówku typy danych `int`, `double`, `string`, `bool` lub innego poprawnego typu danych. Oto przykład funkcji, która zwraca wartość `int`:

```
int sum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    return result;
}
```

Ta funkcja nosi nazwę `sum`. Zauważ, że w nagłówku funkcji zamiast słowa kluczowego `void` widoczna jest nazwa typu danych `int` określająca typ zwracanych wartości. Powyższy kod definiuje funkcję o nazwie `sum`, która przyjmuje dwa argumenty typu `int`. Argumenty przekazywane są do zmiennych parametrów `num1` i `num2`. Wewnątrz funkcji deklarowana jest zmienna lokalna `result`. Zmienne parametrów `num1` i `num2` są do siebie dodawane, a ich suma zostaje przypisana do zmiennej `result`. Ostatnia instrukcja funkcji wygląda tak:

```
return result;
```

W funkcji zwracającej wartość musi znaleźć się instrukcja `return`. Sprawia ona, że funkcja kończy swoje działanie i zwraca wartość do instrukcji, która ją wywołała. A tak wygląda ogólny format instrukcji `return` umieszczanej w funkcji zwracającej wartość:

```
return Wyrażenie;
```

*Wyrażenie* to wartość, która zostanie zwrócona. Może to być dowolne wyrażenie, które ma wartość, na przykład zmienna, literal lub wyrażenie matematyczne. W tym przypadku funkcja `sum` zwraca wartość zmiennej `result`. Oto jak możemy wywołać funkcję `sum`:

```
int total;
total = sum(2, 3);
```

Po wykonaniu tego kodu w zmiennej `total` znajdzie się wartość 5.

### Zwracanie ciągów znaków przez funkcję w języku C++

Poniższy kod jest przykładem tego, w jaki sposób funkcja może zwrócić ciąg znaków. Zauważ, że nagłówek funkcji podaje typ `string` jako typ zwracany. Ta funkcja przyjmuje dwa argumenty typu `string` (imię i nazwisko osoby), a potem łączy te dwa ciągi w jeden, zawierający pełne imię i nazwisko osoby. Następnie zwracany jest uzyskany w ten sposób ciąg znaków:

```
string fullName(string firstName, string lastName)
{
    string name;
    name = firstName + " " + lastName;
    return name;
}
```

Poniższy fragment kodu pokazuje, jak możemy wywołać tę funkcję:

```
string customerName;
customerName = fullName("Jaś", "Fasola");
```

Po wykonaniu tego kodu wartością zmiennej `customerName` będzie ciąg znaków „Jaś Fasola”.

## Zwracanie wartości logicznej przez funkcję w języku C++

Funkcje mogą również zwracać wartości typu `bool`. Poniższa funkcja przyjmuje argument i zwraca wartość `true`, jeśli otrzymany argument będzie się zawierał się w przedziale od 1 do 100, a w przeciwnym wypadku zwróci wartość `false`:

```
bool isValid(int number)
{
    bool status;
    if (number >= 1 && number <= 100)
        status = true;
    else
        status = false;
    return status;
}
```

Oto instrukcja `if-else`, która wywołuje powyższą funkcję:

```
int value = 20;
if (isValid(value))
    cout << "Wartość znajduje się w przedziale." << endl;
else
    cout << "Wartość znajduje się poza przedziałem." << endl;
```

Po wykonaniu tego kodu wyświetli się komunikat „Wartość znajduje się w przedziale”.

# Pytania kontrolne

## Test jednokrotnego wyboru

1. Jest to gotowa funkcja wbudowana w dany język programowania.
  - a) funkcja standardowa
  - b) funkcja biblioteczna
  - c) funkcja użytkownika
  - d) funkcja kawiarniana
2. W ten sposób określa się mechanizm, w którym przekazuje się pewne dane, następnie mechanizm ten wykonuje na tych danych jakieś operacje i zwraca dane wyjściowe.
  - a) szklana skrzynka
  - b) biała skrzynka
  - c) ciemna skrzynka
  - d) czarna skrzynka
3. Tak nazywa się ta część definicji funkcji, w której umieszcza się typ zwracanej wartości.
  - a) nagłówek
  - b) stopka
  - c) ciało
  - d) instrukcja `Return`

4. Na tę część definicji funkcji składają się jedna lub więcej instrukcji, które zostaną wykonane, gdy funkcja zostanie wywołana.
  - a) nagłówek
  - b) stopka
  - c) ciało
  - d) instrukcja Return
5. Ta instrukcja pseudokodu powoduje zakończenie działania funkcji i zwrócenie wartości do miejsca, w którym funkcja została wywołana.
  - a) End
  - b) Send
  - c) Exit
  - d) Return
6. Jest to narzędzie, dzięki któremu możemy opisać, jakie dane wyjściowe przyjmuje funkcja, w jaki sposób je przetwarza i jakie dane zwraca.
  - a) schemat hierarchiczny
  - b) tabelka IPO
  - c) schemat datagramowy
  - d) schemat przetwarzania danych
7. Funkcje tego typu zwracają wartość True lub False.
  - a) funkcje binarne
  - b) funkcje TrueFalse
  - c) funkcje boolowskie
  - d) funkcje logiczne
8. Funkcja ta jest przykładową funkcją w pseudokodzie do konwertowania typów danych.
  - a) sqrt
  - b) toReal
  - c) substring
  - d) isNumeric
9. Tego typu błąd wystąpi w programie, gdy spróbowajesz zapisać do zmiennej jednego typu wartość innego typu.
  - a) błąd niezgodności typów
  - b) błąd logiczny
  - c) błąd relacyjny
  - d) błąd konwersji bitów
10. Jest to fragment ciągu znaków.
  - a) podciąg
  - b) ciąg wewnętrzny
  - c) miniciąg
  - d) ciąg fragmentaryczny

### Prawda czy fałsz?

1. Aby można było wywołać funkcję biblioteczną, jej kod należy umieścić w programie.
2. Złożone wyrażenie matematyczne można niekiedy uprościć, wydzielając z niego fragment i umieszczając go wewnątrz funkcji.

3. W wielu językach programowania podczas próby przypisania liczby rzeczywistej do zmiennej typu całkowitego pojawi się błąd.
4. W pewnych językach programowania, aby podnieść liczbę do potęgi, trzeba skorzystać z funkcji bibliotecznej.
5. W przypadku porównania ciągów znaków, gdy różzniane są duże i małe litery, ciągi "yoda" i "YODA" są sobie równoznaczne.

### Krótką odpowiedź

1. Czym różni się funkcja od modułu?
2. Za pomocą których cech opisuje się funkcję w tabelkach IPO?
3. Która z funkcji do konwertowania typów danych służy do zamiany liczby rzeczywistej na liczbę całkowitą? Co się dzieje z częścią ułamkową liczby rzeczywistej?
4. Co to jest podciąg?
5. W jaki sposób działają funkcje `stringToInteger` i `stringToReal`, które omówiłem w tym rozdziale?
6. W jaki sposób działają funkcje `isInteger` i `isReal`, które omówiłem w tym rozdziale?

### Warsztat projektanta algorytmów

1. Napisz instrukcję, która generuje liczbę losową z przedziału 1 – 100 i przypisuje ją do zmiennej `rand`.
2. Poniższa instrukcja wywołuje funkcję `half`, która zwraca wartość równą połowie liczby przekazanej jako argument funkcji (założymy, że zarówno `result`, jak i `number` są zmiennymi typu `Real`). Napisz definicję funkcji `half` za pomocą pseudokodu.

```
Set result = half(number)
```

3. Program zawiera następującą definicję funkcji:

```
Function Integer cube(Integer num)
    Return num * num * num
End Function
```

- Napisz polecenie, w którym wywołasz tę funkcję, przekażesz do niej wartość 4, a wynik zwrócony przez funkcję przypiszesz do zmiennej `result`.
4. Zaprojektuj w pseudokodzie funkcję o nazwie `timesTen`, która będzie przyjmowała argument typu `Integer`. Po wywoaniu funkcja powinna zwrócić wartość iloczynu wartości przekazanej do funkcji i liczby 10.
  5. Zaprojektuj w pseudokodzie funkcję o nazwie `getFirstName`, która będzie prosiła użytkownika o wprowadzenie imienia, a następnie będzie zwracała to imię.
  6. Założmy, że w programie są dwie zmienne typu `String`, o nazwach `str1` i `str2`. Napisz polecenie, w którym do zmiennej `str2` zostanie przypisana wersja ciągu `str1`, w której wszystkie litery zostały zamienione na duże.

## Ćwiczenia z wykrywania błędów

1. Programista chce, aby jego program wyświetlał trzy losowe liczby z przedziału 1 – 7. Jednak zgodnie z tym, czego dowiedziałeś się z tego rozdziału, w programie występuje błąd. Czy potrafisz go zlokalizować?

```
// Program wyświetla trzy losowe liczby
// z przedziału od 1 do 7
Declare Integer count
// Wyświetlamy trzy losowe liczby
For count = 1 To 3
    Display random(7, 1)
End For
```

2. Czy potrafisz w poniższym kodzie znaleźć błąd, który powoduje, że funkcja nie zwraca wartości zgodnej z komentarzem?

```
// Funkcja calcDiscountPrice przyjmuje cenę produktu
// i wartość rabatu, na podstawie tych wartości
// oblicza i zwraca cenę po rabacie
Function Real calcDiscountPrice(Real price, Real percentage)
    // Obliczamy rabat
    Declare Real discount = price * percentage

    // Odejmujemy rabat od ceny
    Declare Real discountPrice = price - discount

    // Zwracamy cenę po rabacie

    Return
End Function
```

3. Czy potrafisz w poniższym kodzie znaleźć błąd, który powoduje, że funkcja nie zwraca wartości zgodnej z komentarzem?

```
// Znajdź błąd w poniższym pseudokodzie
Module main()
    Declare Real value, result

    // Pobieramy od użytkownika wartość
    Display "Wprowadź wartość."
    Input value

    // Obliczamy 10% wprowadzonej wartości
    Call tenPercent(value)

    // Wyświetlamy 10% wprowadzonej wartości
    Display "10 procent liczby ", value, " wynosi ", result
End Module

// Funkcja tenPercent oblicza 10 procent
// wartości przekazanej jako argument
Function Real tenPercent(Real num)
    Return num * 0.1
End Function
```

# Ćwiczenia programistyczne

## 1. Pole powierzchni prostokąta

Pole powierzchni prostokąta obliczamy według następującego wzoru:

$$\text{pole} = \text{długość} \cdot \text{szerokość}$$

Zaprojektuj funkcję, która będzie przyjmowała jako argumenty długość i szerokość prostokąta, a następnie zwracała pole jego powierzchni. Wywołaj tę funkcję w programie, który poprosi użytkownika o wprowadzenie długości i szerokości prostokąta, a potem wyświetli pole jego powierzchni.

## 2. Stopły nacale

Jedna stopa odpowiada 12 calom. Zaprojektuj funkcję o nazwie `feetToInches`, która będzie przyjmowała jako argument liczbę stóp i zwracała odpowiadającą jej liczbę cali. Wywołaj tę funkcję w programie, który poprosi użytkownika o wprowadzenie liczby stóp, a następnie wyświetli odpowiadającą im liczbę cali.

## 3. Test matematyczny

Zaprojektuj program, który będzie generował proste zadania matematyczne. Program powinien wyświetlić dwie liczby losowe, które uczeń będzie musiał do siebie dodać:

$$\begin{array}{r} 247 \\ + 129 \\ \hline \end{array}$$

Program powinien poprosić ucznia o wprowadzenie wyniku dodawania i jeżeli odpowiedź jest prawidłowa, wyświetlić komunikat z gratulacjami, a jeśli odpowiedź jest błędna, wyświetlić komunikat z prawidłowym rozwiązaniem.

## 4. Wartość maksymalna dwóch liczb

Zaprojektuj funkcję o nazwie `max`, która będzie przyjmowała jako argumenty dwie liczby całkowite i zwracała wartość równą większej z tych liczb. Przykładowo po przekazaniu do funkcji liczb 7 i 12 funkcja powinna zwrócić wartość 12. Wywołaj tę funkcję w programie, który poprosi użytkownika o wprowadzenie dwóch liczb całkowitych, a następnie wyświetli wartość większą z nich.

## 5. Spadek swobodny

Odległość, jaką przebędzie obiekt w określonym czasie podczas spadku swobodnego, można obliczyć za pomocą następującego wzoru:

$$d = \frac{1}{2} g t^2$$

We wzorze  $d$  to odległość w metrach,  $g$  jest równe 9,8, a  $t$  to czas, przez jaki spada obiekt.

Zaprojektuj funkcję o nazwie `fallingDistance`, która będzie przyjmowała jako argument czas spadania. Funkcja powinna zwrócić odległość w metrach, którą przebędzie obiekt po wskazanym czasie. Zaprojektuj program, w którym wywołasz tę funkcję w pętli i przekażesz do niej po kolej wartości z przedziału 1 do 10, a następnie wyświetlisz wynik zwrócony przez funkcję.

## 6. Energia kinetyczna

Z fizyki wiemy, że poruszający się obiekt ma energię kinetyczną. Energię tę można obliczyć na podstawie następującego wzoru:

$$KE = \frac{1}{2} m v^2$$

We wzorze  $KE$  to energia kinetyczna,  $m$  to masa obiektu w kilogramach,  $v$  to prędkość obiektu w metrach na sekundę.

Zaprojektuj funkcję o nazwie `kineticEnergy`, która będzie przyjmowała jako argumenty masę obiektu i jego prędkość. Funkcja powinna zwracać wartość energii kinetycznej, jaką ma poruszający się obiekt. Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie masy i prędkości obiektu, a następnie wywoła funkcję `kineticEnergy` i wyświetli wartość energii kinetycznej.

## 7. Średni wynik ze sprawdzianu i ocena

Napisz program, który będzie prosił użytkownika o wprowadzenie wyników z pięciu sprawdzianów. Program powinien wyświetlić ocenę z każdego sprawdzianu i średni wynik z pięciu sprawdzianów. W programie zaprojektuj następujące funkcje:

- `calcAverage` — powinna przyjmować jako argumenty pięć wyników ze sprawdzianów i zwracać średni wynik;
- `determineGrade` — powinna przyjmować jako argument wynik ze sprawdzianu i zwracać (jako ciąg znaków) ocenę odpowiadającą temu wynikowi określonej na podstawie następującej

| Wynik      | Ocena |
|------------|-------|
| 90 – 100   | 5     |
| 80 – 89    | 4     |
| 70 – 79    | 3     |
| 60 – 69    | 2     |
| Poniżej 60 | 1     |

## 8. Zliczanie liczb parzystych i nieparzystych

Z tego rozdziału wiesz, w jaki sposób można zaprojektować algorytm sprawdzający, czy dana liczba jest parzysta czy nieparzysta (punkt „Zwracanie wartości typu Boolean” w podrozdziale 6.2). Zaprojektuj program, który będzie generował 100 losowych liczb i zliczał, ile z nich jest parzystych, a ile nieparzystych.

## 9. Zgadnij liczbę

Zaprojektuj program do odgadywania liczb. Program powinien generować liczbę losową, a następnie poprosić użytkownika o odgadnięcie numeru. Za każdym razem, gdy użytkownik zgłasza swoje przypuszczenia, program powinien wskazać, czy liczba była zbyt duża czy zbyt mała. Gra skończy się, gdy użytkownik prawidłowo odgadnie numer. Po zakończeniu gry program powinien wyświetlić liczbę prób wykonanych przez użytkownika.

## 10. Liczby pierwsze

Liczba pierwsza to taka liczba, która dzieli się tylko przez 1 i przez samą siebie. Przykładowo liczbą pierwszą jest 5, ponieważ dzieli się tylko przez 1 i przez 5. Ale liczba 6 nie jest liczbą pierwszą, dzieli się bowiem przez 1, 2, 3 i 6.

Zaprojektuj funkcję boolowską o nazwie `isPrime`, która będzie przyjmowała argument w postaci liczby całkowitej i zwracała wartość `True`, jeżeli liczba ta jest liczbą pierwszą lub — w przeciwnym wypadku —wartość `False`. Wywołaj tę funkcję w programie, który poprosi użytkownika o wprowadzenie liczby, a następnie wyświetli komunikat informujący, czy dana liczba jest liczbą pierwszą.



**WSKAZÓWKA:** Przypominam, że operator `MOD` dzieli przez siebie dwie liczby i zwraca resztę z dzielenia. Wynikiem wyrażenia `num1 MOD num2` będzie 0 tylko wtedy, gdy `num1` dzieli się przez `num2` bez reszty.

## 11. Lista liczb pierwszych

W tym ćwiczeniu zakładam, że zaprojektowałeś funkcję `isPrime` w ćwiczeniu 10. Zaprojektuj następny program, który będzie wyświetlał kolejne liczby pierwsze z przedziału 1 – 100. Program powinien zawierać pętlę i wywoływać funkcję `isPrime`.

## 12. Papier, kamień, nożyce

Zaprojektuj program, dzięki któremu użytkownik będzie mógł zagrać z komputerem w grę „papier, kamień, nożyce”. Program powinien wyglądać następująco:

- 1) Po uruchomieniu programu należy wygenerować losową liczbę z przedziału 1 – 3. Jeżeli liczba ta jest równa 1, oznacza to, że komputer wylosował kamień. Jeżeli liczba jest równa 2, komputer wylosował papier. Jeżeli liczba jest równa 3, komputer wylosował nożyce. Nie wyświetlaj na razie tego wyniku.
- 2) Użytkownik wprowadza za pomocą klawiatury, czy wybrał kamień, papier czy
- 3) Wyświetla się informacja, co wybrał komputer.
- 4) Komputer wyświetla komunikat informujący, czy wygrał użytkownik czy komputer. Zwycięzcę określa się za pomocą następujących reguł:
  - Jeżeli pierwszy gracz wybrał kamień, a drugi wybrał nożyce, wygrywa kamień (kamień tępi nożyce).
  - Jeżeli pierwszy gracz wybrał nożyce, a drugi wybrał papier, wygrywają nożyce (nożyce tną papier).
  - Jeżeli pierwszy gracz wybrał papier, a drugi wybrał kamień, wygrywa papier (papier owija kamień).
  - Jeżeli obaj gracze dokonają takiego samego wyboru, grę trzeba powtórzyć.

### 13. Symulator jednorękiego bandyty

Jednoręki bandyta to maszyna hazardowa, do której gracz wrzuca monety, po czym pociąga za dźwignię (lub naciska przycisk). Maszyna następnie wyświetla serię losowych symboli. Jeżeli co najmniej dwa symbole do siebie pasują, gracz wygrywa określona sumę pieniędzy, które maszyna mu wypłaca.

Zaprojektuj program, który będzie symulował działanie jednorękiego bandyty. Po uruchomieniu program powinien wykonać następujące czynności:

- Program prosi gracza o wrzucenie do maszyny dowolnej sumy pieniędzy.
- Zamiast symboli program wyświetla w sposób losowy kilka słów z następującej listy:

*Wiśnie, Pomarańcze, Śliwki, Dzwonki, Melony, Sztabki*

- Program trzykrotnie losuje i wyświetla słowo z tej listy.
- Jeżeli żadne z wyświetlonych słów do siebie nie pasują, program wyświetla komunikat, że wygrana wynosi 0. Jeżeli pasują do siebie dwa słowa, program informuje użytkownika, że wygrał kwotę równą dwukrotności sumy wrzuconej do maszyny. Jeżeli pasują do siebie wszystkie trzy słowa, program informuje użytkownika, że wygrał kwotę równą trzykrotności sumy wrzuconej do maszyny.
- Program pyta użytkownika, czy chce zagrać jeszcze raz. Jeśli tak, poprzednie kroki należy powtórzyć. Jeśli nie, program wyświetla sumaryczną wartość wrzuconych do maszyny pieniędzy i sumę wygranych.

### 14. Gra ESP

Zaprojektuj program, który przetestuje twoją ESP, czyli percepcję pozazmysłową. Program powinien losowo wybierać kolor, a użytkownik zostanie poproszony o przewidywanie wyboru programu przed jego ujawnieniem. Zaprojektuj program tak, aby losowo wybierał jedno z poniższych słów:

*czerwony, zielony, niebieski, pomarańczowy, żółty*

W celu wybrania słowa program może wygenerować liczbę losową. Na przykład jeśli wybraną liczbą będzie 0, to wybranym słowem będzie *czerwony*, natomiast jeśli wybrana zostanie liczba 1, to wybranym słowem będzie *zielony*, i tak dalej.

Następnie program powinien poprosić użytkownika o wprowadzenie koloru wybranego przez komputer. Po wprowadzeniu go przez użytkownika program powinien wyświetlić nazwę losowo wybranego koloru. Program powinien także powtórzyć tę operację 10 razy, a następnie wyświetlić liczbę razy, kiedy to użytkownik prawidłowo odgadł wybrany kolor.

## TEMATYKA

- |                                        |                                           |
|----------------------------------------|-------------------------------------------|
| 7.1 Garbage In, Garbage Out            | 7.3 Programowanie defensywne              |
| 7.2 Pętla walidacji danych wejściowych | 7.4 Rzut oka na języki Java, Python i C++ |

## 7.1

## Garbage In, Garbage Out

**WYJAŚNIENIE:** Jeśli program pobierze błędne dane wejściowe, w wyniku zwróci także błędne dane. Program powinien być zaprojektowany w taki sposób, aby odrzucać błędne dane wejściowe.

Często wśród programistów można usłyszeć wyrażenie „garbage in, garbage out”<sup>1</sup>. Określenie to, niekiedy skracane do akronimu GIGO, odnosi się do faktu, że komputer nie jest w stanie odróżnić poprawnych danych od danych niepoprawnych. Kiedy użytkownik programu wprowadzi błędne dane, program je przetworzy i w wyniku otrzymamy także błędne dane wyjściowe. Spójrz na przykładowy pseudokod programu do obliczania tygodniowego wynagrodzenia na listingu 7.1. Zwróć uwagę, co się stało, gdy podczas przykładowego uruchomienia programu użytkownik wprowadził błędne dane.

**Listing 7.1**

```
1 // Zmienne, w których zapiszemy liczbę przepracowanych godzin,  
2 // stawkę godzinową i wynagrodzenie całkowite  
3 Declare Real hours, payRate, grossPay  
4  
5 // Pobieramy liczbę przepracowanych godzin  
6 Display "Wprowadź liczbę przepracowanych godzin w tygodniu."  
7 Input hours  
8
```

<sup>1</sup> „śmieci na wejściu, śmiec na wyjściu” — przyp. tłum.

```

9 // Pobieramy stawkę godzinową
10 Display "Wprowadź stawkę godzinową."
11 Input payRate
12
13 // Obliczamy wynagrodzenie całkowite
14 Set grossPay = hours * payRate
15
16 // Wyświetlamy wynagrodzenie całkowite
17 Display "Wynagrodzenie całkowite wynosi ", currencyFormat(grossPay)

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę przepracowanych godzin.

**400 [Enter]**

Wprowadź stawkę godzinową.

**20 [Enter]**

Wynagrodzenie całkowite wynosi 8 000,00 zł

Czy zauważłeś miejsce, w którym użytkownik wprowadził błędne dane? Pracownik, który odbierze tygodniową wyplątę, będzie mile zaskoczony, ponieważ księgowa, która używała programu, wprowadziła liczbę przepracowanych godzin równą 400. Ponieważ tydzień ma mniej niż 400 godzin, księgowa najprawdopodobniej miała zamiar wpisać liczbę 40. Komputer, nieświadomy tego faktu, przetworzył te dane tak, jakby były poprawne. Czy widzisz w tym programie inny przypadek, wskutek którego program może wygenerować błędne dane? Jednym z nich jest ujemna liczba przepracowanych godzin, a drugim — błędna stawka godzinowa.

Czasami można usłyszeć w wiadomościach, jak w wyniku błędu komputera klienci dokonujący niewielkich zakupów zostali obciążeni gigantycznymi kwotami, albo o przypadku, gdy płatnik otrzymał nieuzasadniony zwrot podatku. Takie „błędy komputerów” są bardzo rzadko winą samych komputerów, lecz raczej skutkiem błędów w programie lub błędnych danych wprowadzonych do programu.

Integralność danych wyjściowych programu jest taka sama jak integralność danych wejściowych. Z tego powodu należy projektować program w taki sposób, aby błędne dane były przez niego odrzucane. Po wprowadzeniu danych, ale jeszcze przed ich przetwarzaniem, należy je wnikliwie przeanalizować. Jeśli dane wejściowe są błędne, program powinien je odrzucić i poprosić użytkownika o wprowadzenie prawidłowych danych. Proces ten nazywamy **walidacją danych wejściowych**. W tym rozdziale omówię kilka technik, z których będziesz mógł skorzystać podczas walidacji danych wejściowych.



## Punkt kontrolny

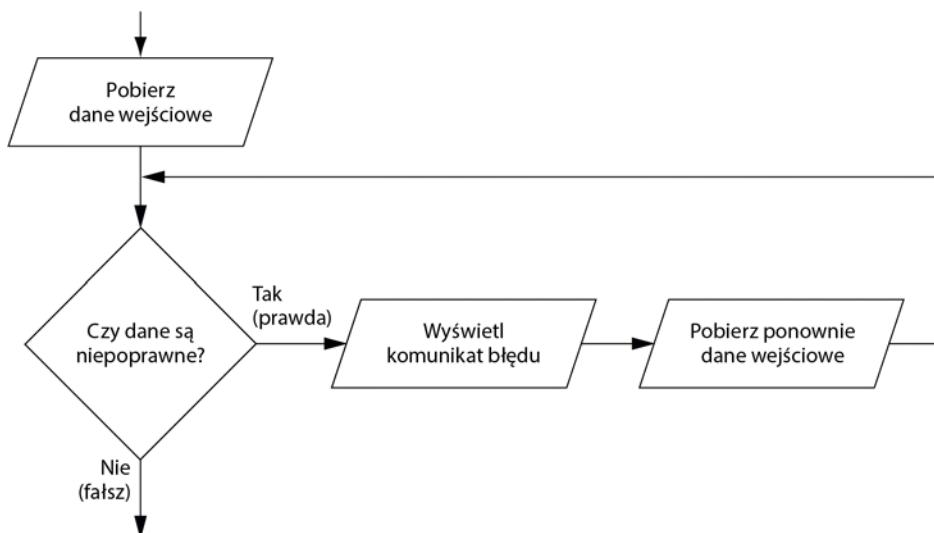
- 7.1. Co oznacza określenie „garbage in, garbage out”?
- 7.2. Opisz proces walidacji danych wejściowych.

## 7.2

# Pętla walidacji danych wejściowych

**WYJAŚNIENIE:** Walidacja danych wejściowych jest najczęściej wykonywana w pętli, która działa dopóty, dopóki dane wejściowe są niepoprawne.

Na rysunku 7.1 przedstawiłem bardzo popularną technikę walidacji danych wejściowych. Odczytujemy dane wejściowe, a następnie pojawia się pętla, w której warunek jest sprawdzany na początku. Jeśli dane są niepoprawne, wywoływane są instrukcje zawarte w ciele pętli — pętla wyświetla wtedy komunikat błędu, aby użytkownik wiedział, że dane są błędne, i program ponownie czeka na wprowadzenie poprawnych danych. Pętla będzie działała dopóty, dopóki dane wejściowe będą niepoprawne.



Rysunek 7.1. Zasada działania pętli walidacji danych wejściowych

Zwrócić uwagę, że na schemacie blokowym na rysunku 7.1 odczytujemy dane w dwóch miejscach: tuż przed pętlą i wewnętrznie pętli. Pierwszą z tych operacji — wystającą przed pętlą — nazywamy **odczytem wstępny**, a jej zadaniem jest pobranie od użytkownika początkowych danych, które będziemy walidować w pętli. Jeżeli dane okażą się błędne, program wykona kolejne operacje odczytu umieszczone wewnętrz pętli.

Przyjrzyjmy się przykładowi. Założymy, że projektujesz program, który będzie pobierał od użytkownika wyniki ze sprawdzianów, a Ty chcesz mieć pewność, że użytkownik nie wprowadzi liczby mniejszej niż 0. Na pseudokodzie pokazałem, w jaki sposób można utworzyć pętlę walidującą dane wejściowe, która odrzuca wyniki mniejsze od 0:

```

// Pobieramy wynik ze sprawdzianu
Display "Wprowadź wynik ze sprawdzianu."
Input score
  
```

```
// Sprawdzamy, czy wynik nie jest mniejszy od 0
While score < 0
    Display "BŁĄD! Wynik nie może być mniejszy niż 0."
    Display "Wprowadź prawidłowy wynik."
    Input score
End While
```

W zaprezentowanym pseudokodzie proszę użytkownika o wprowadzenie wyniku ze sprawdzianu (jest to odczyt wstępny), a następnie program przechodzi do pętli `While`. W rozdziale 5. wyjaśniłem, że pętla `While` jest typem pętli, w której warunek jest sprawdzany na początku, co oznacza, że warunek `score < 0` zostanie sprawdzony, zanim pętla wykona pierwszą iterację. Jeżeli użytkownik wprowadził prawidłowy wynik, wyrażenie zwróci prawdę i pętla nie wykona żadnej iteracji. Jeśli jednak dane będą nieprawidłowe, wyrażenie zwróci fałsz i wykonają się polecenia umieszczone w ciele pętli. Pętla wyświetli komunikat błędu i poprosi użytkownika o wprowadzenie poprawnego wyniku ze sprawdzianu. Pętla będzie wykonywała kolejne iteracje aż do momentu, gdy użytkownik wprowadzi prawidłowy wynik.



**UWAGA:** Pętla walidacji danych wejściowych — np. taka jak na rysunku 7.1 — nazywana jest niekiedy **pułapką na błędy lub obsługą błędu**.

Powyższy kod odrzuca tylko ujemne wyniki ze sprawdzianów. A co zrobić, jeśli chcielibyśmy dodatkowo odrzucać wyniki większe niż 100? Możemy w takim przypadku zmodyfikować pętlę walidującą dane wejściowe i umieścić w niej boolowskie wyrażenie złożone. Przedstawiłem to tutaj:

```
// Pobieramy wynik ze sprawdzianu
Display "Wprowadź wynik ze sprawdzianu."
Input score
// Walidujemy wynik ze sprawdzianu
While score < 0 OR score > 100
    Display "BŁĄD! Wynik nie może być mniejszy niż 0"
    Display "lub większy niż 100."
    Display "Wprowadź prawidłowy wynik."
    Input score
End While
```

Pętla w tym pseudokodzie sprawdza, czy zmienna `score` jest mniejsza niż 0 lub większa niż 100. Jeżeli któryś z tym warunków jest prawdziwy, wyświetli się komunikat błędu i użytkownik poproszony zostanie o wprowadzenie prawidłowego wyniku.



**UWAGA:** Aby sprawdzić, czy zmienna `score` nie mieści się w określonym przedziale, użyłem w pseudokodzie operatora `OR`. Pomyśl, co by się stało, gdybyśmy zamiast operatora `OR` użyli tutaj operatora `AND`:

`score < 0 AND score > 100`

Wyrażenie takie nigdy nie zwróci prawdy, ponieważ nie ma możliwości, aby liczba była jednocześnie mniejsza niż 0 i większa niż 100!

## W centrum uwagi

### Projektowanie pętli walidacji danych wejściowych



W rozdziale 5. przedstawiłem program, za pomocą którego Twoja przyjaciółka Agata mogła obliczać cenę detaliczną dostarczonych jej produktów (listing 5.6). Korzystając z programu, Agata natrafiła jednak na pewien problem. Niektóre produkty mają cenę hurtową równą 50 groszy. Agata wprowadza tę cenę w programie jako 0,50, a ponieważ klawisz 0 znajduje się zaraz obok klawisza -, niekiedy omyłkowo zdarzy jej się wprowadzić cenę ujemną. Poprosiła Cię więc o zmodyfikowanie programu w taki sposób, aby uniemożliwił wprowadzenie ujemnej ceny produktu.

Postanowileś dodać do modułu `showRetail1` pętlę walidującą dane wejściowe, która będzie odrzucała liczby ujemne wprowadzane przez użytkownika do zmiennej `wholesale`. Na listingu 7.2 znajduje się poprawiona wersja programu — kod odpowiadający za walidację danych umieszczony jest w liniach od 28. do 33.

Na rysunku 7.2 przedstawiłem schemat blokowy modułu `showRetail1`.

#### **Listing 7.2**

```

1 Module main()
2 // Zmienna lokalna
3 Declare String doAnother
4
5 Do
6 // Obliczamy i wyświetlamy cenę detaliczną
7 Call showRetail1()
8
9 // Jeszcze raz?
10 Display "Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)"
11 Input doAnother
12 While doAnother == "t" OR doAnother == "T"
13 End Module
14
15 // Moduł showRetail pobiera od użytkownika cenę hurtową
16 // i wyświetla cenę detaliczną
17 Module showRetail1()
18 // Zmienne lokalne
19 Declare Real wholesale, retail
20
21 // Stała zawierająca markę
22 Constant Real MARKUP = 2.50
23
24 // Pobieramy cenę hurtową
25 Display "Wprowadź cenę hurtową produktu."
26 Input wholesale
27
28 // Walidujemy cenę hurtową
29 While wholesale < 0

```

```

30     Display "Cena nie może być ujemna."
31     Display "Wprowadź prawidłową cenę hurtową."
32     Input wholesale
33 End While
34
35 // Obliczamy cenę detaliczną
36 Set retail = wholesale * MARKUP
37
38 // Wyświetlamy cenę detaliczną
39 Display "Cena detaliczna wynosi ", retail, " zł."
40 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę hurtową produktu.

**-0.50 [Enter]**

Cena nie może być ujemna.

Wprowadź prawidłową cenę hurtową.

**0.50 [Enter]**

Cena detaliczna wynosi 1.25 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)

**n [Enter]**

## Walidacja danych z wykorzystaniem pętli sprawdzającej warunek na końcu

Zapewne zastanawiasz się, czy zamiast odczytu wstępnego do walidowania danych wejściowych można wykorzystać pętlę, w której warunek jest sprawdzany na końcu. Oto, w jaki sposób do walidowania wyniku ze sprawdzianu można wykorzystać pętlę Do-While:

```

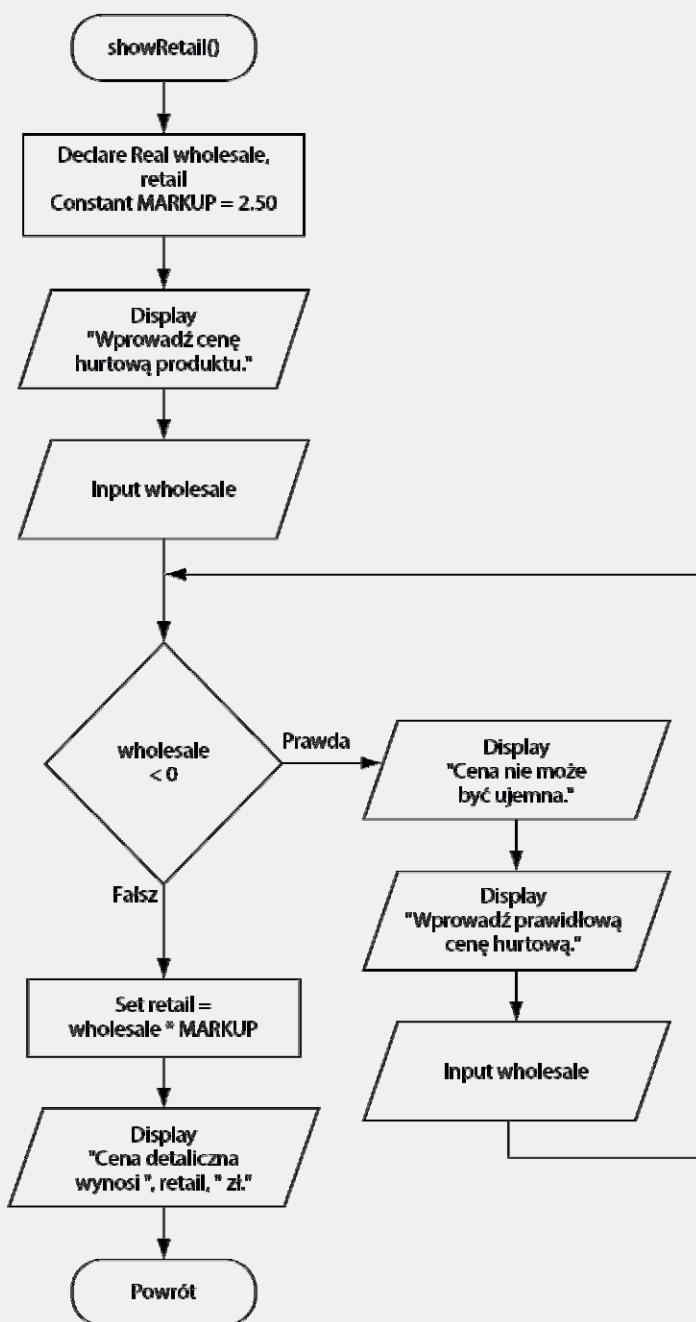
Do
    Display "Wprowadź wynik ze sprawdzianu."
    Input score
    While score < 0 OR score > 100

```

Powyższy kod działa prawidłowo, jednak gdy użytkownik wprowadzi nieprawidłowe dane, nie wyświetla on komunikatu błędu — po prostu w każdej iteracji wyświetla kolejną prośbę o wprowadzenie danych. Takie zachowanie programu może okazać się dla użytkownika niezrozumiałe, więc zazwyczaj lepiej jest dokonać odczytu wstępnego i wykorzystać podczas walidowania danych pętlę, w której warunek jest sprawdzany na początku.

## Tworzenie funkcji walidacyjnych

Przykłady walidacji danych wejściowych, które dotychczas przedstawiłem, były bardzo proste. Wiesz już, w jaki sposób tworzyć pętle walidujące i odrzucać w nich wartości ujemne i wartości spoza określonego przedziału. Jednak w wielu przypadkach walidacja danych jest znacznie bardziej skomplikowana.



Rysunek 7.2. Schemat blokowy modułu showRetail

Załóżmy, że projektujesz program, w którym użytkownik wprowadza numer modelu, a program powinien przyjmować tylko wartości 100, 200 lub 300. Możesz w takim przypadku zaprojektować następujący algorytm walidacji danych:

```
// Pobieramy numer modelu
Display "Wprowadź numer modelu."
Input model

While model != 100 AND model != 200 AND model != 300
    Display "Prawidłowe numery modeli to 100, 200 i 300."
    Display "Wprowadź prawidłowy numer modelu."
    Input model
End While
```

W pętli walidacji danych wejściowych używam złożonego wyrażenia boolowskiego. Pętla będzie wykonywała kolejne iteracje dopóty, dopóki zmienna `model` będzie różna od 100, 200 i 300. Mimo że takie rozwiązanie jest prawidłowe, można taką pętlę uprościć — możemy utworzyć nową funkcję boolowską, która będzie sprawdzała zmienną `model`, a następnie wywołać tę funkcję w pętli. Założymy, że będziemy przekazywać zmienną `model` jako argument funkcji `isValid`. Funkcja ta zwraca wartość `True`, gdy zmienna `model` ma nieprawidłową wartość — w przeciwnym przypadku zwraca wartość `False`. Możemy wtedy zmodyfikować pętlę do takiej postaci:

```
// Pobieramy numer modelu
Display "Wprowadź numer modelu."
Input model

While isValid(model)
    Display "Prawidłowe numery modeli to 100, 200 i 300."
    Display "Wprowadź prawidłowy numer modelu."
    Input model
End While
```

Dzięki takiemu rozwiązaniu program jest bardziej czytelny — jest oczywiste, że pętla wykonuje kolejne iteracje tak długo, jak zmienna `model` jest nieprawidłowa. Poniżej przedstawiłem pseudokod funkcji `isValid`. Przyjmuje ona jako argument numer modelu, a następnie zwraca wartość `True`, jeżeli przekazany argument nie jest równy 100, 200 i 300. W przeciwnym razie zwraca wartość `False`.

```
Function Boolean isValid(Integer model)
    // Zmienna lokalna, w której zapiszemy wartość True lub False
    Declare Boolean status

    // Jeżeli numer modelu jest nieprawidłowy, ustawiamy status na True
    // W przeciwnym razie ustawiamy status na False
    If model != 100 AND model != 200 AND model != 300 Then
        Set status = True
    Else
        Set status = False
    End If

    // Zwracamy status
    Return status
End Function
```

## Walidowanie ciągów znaków

W pewnych programach będziesz musiał walidować dane w postaci ciągów znaków. Przykładem niech będzie program, który zadaje użytkownikowi pytanie, a Ty chcesz mieć pewność, że użytkownik wprowadzi jedną z dwóch dozwolonych odpowiedzi: "tak" lub "nie". Poniższy pseudokod prezentuje jedno z rozwiązań tego problemu:

```
// Pobieramy odpowiedź na pytanie
Display "Czy Twój szef jest dobrym liderem?"
Input answer
// Walidujemy dane
While answer != "tak" AND answer != "nie"
    Display "Odpowiedz tak lub nie. Czy Twój szef jest"
    Display "dobrym liderem?"
    Input answer
End While
```

Pętla walidacji danych wejściowych odrzuca wszystkie dane, z wyjątkiem ciągów "tak" i "nie". Nie jest to jednak rozwiązanie zbyt elastyczne, ponieważ nie uwzględnia wielkości wprowadzonych znaków. Oznacza to, że pętla odrzuci także odpowiedzi "TAK", "NIE", "Tak" i "Nie". Aby program był bardziej przyjazny użytkownikowi, trzeba sprawić, aby przyjmował również odpowiedzi zawierające różne kombinacje małych i dużych znaków w słowach „tak” i „nie”. Jak zapewne pamiętasz, w rozdziale 6. omówiłem funkcje `toUpperCase` i `toLowerCase`, które przydają się podczas porównywania ciągów znaków, niezależnie od wielkości występujących w nich znaków. Na poniższym pseudokodzie pokazałem przykład wykorzystania funkcji `toLowerCase`:

```
// Pobieramy odpowiedź na pytanie
Display "Czy Twój szef jest dobrym liderem?"
Input answer
// Walidujemy dane
While toLower(answer) != "tak" AND toLower(answer) != "nie"
    Display "Odpowiedz tak lub nie. Czy Twój szef jest"
    Display "dobrym liderem?"
    Input answer
End While
```

W pewnych przypadkach to długość ciągu znaków pełni kluczową rolę. Znasz z pewnością portal internetowy lub inny system informatyczny, który wymaga, aby utworzyć w nim konto zabezpieczone hasłem. W niektórych systemach wymagane jest wprowadzenie hasła dłuższego niż określona liczba znaków. Aby sprawdzić długość ciągu znaków, można posłużyć się funkcją `length`, którą omówiłem w rozdziale 6. W poniższym pseudokodzie użyłem funkcji `length`, aby sprawdzić, czy hasło składa się z co najmniej sześciu znaków:

```
// Pobieramy hasło
Display "Wprowadź nowe hasło."
Input password
// Walidujemy długość hasła
While length(password) < 6
    Display "Hasło musi się składać z co najmniej"
    Display "sześciu znaków. Wprowadź nowe hasło."
    Input password
End While
```



## Punkt kontrolny

- 7.3. Opisz, jakie operacje mają miejsce podczas walidacji danych wejściowych w pętli.
- 7.4. Czym jest odczyt wstępny? Do czego służy?
- 7.5. Ile iteracji wykona pętla walidacyjna, gdy okaże się, że po odczycie wstępnym dane wejściowe są prawidłowe?

### 7.3

## Programowanie defensywne

**WYJAŚNIENIE:** Walidacja danych wejściowych jest częścią modelu zwanego programowaniem defensywnym. Dokładna walidacja danych wejściowych jest w stanie zabezpieczyć program zarówno przed oczywistymi, jak i nieoczywistymi błędami.

Programowanie defensywne polega na przewidywaniu ewentualnych błędów mogących wystąpić podczas działania programu i zaprojektowaniu go w taki sposób, aby ich uniknąć. Wszystkie przykłady algorytmów walidujących dane wejściowe, które przedstawiłem w tym rozdziale, są przykładami programowania defensywnego.

Niektóre typy błędów jest łatwo przewidzieć i im zapobiec. Przykładowo możemy nie dopuścić do tego, aby użytkownik wprowadził ujemną liczbę jako cenę lub wynik ze sprawdzianu. Niektóre błędy nie są już niestety tak oczywiste. Jednym z przykładów niech będzie próba odczytu **pustych danych**, która ma miejsce w momencie, gdy program chce odczytać dane wejściowe, ale tych danych mu nie dostarczymy. Taka sytuacja zachodzi na przykład, kiedy po wywołaniu instrukcji `Input` użytkownik naciśnie po prostu klawisz `Enter`, nie wpisując żadnej wartości. Pomimo faktu, że różne języki programowania obsługują puste odczyty w różny sposób, można w nich zazwyczaj zidentyfikować sytuację, gdy operacja pobierania danych się nie powiedzie.

Innym typem błędu, o którym często się zapomina, jest błąd typu wprowadzanych danych. Ma to miejsce na przykład w momencie, gdy program spodziewa się liczby całkowitej, a zamiast tego użytkownik wprowadza liczbę rzeczywistą lub ciąg znaków. W większości języków programowania do wychwytywania tego rodzaju błędów służą odpowiednie funkcje biblioteczne. Często znajdują się wśród nich funkcje takie jak `isInteger` czy `isReal`, które omówiłem w rozdziale 6. Aby skorzystać z tych funkcji w algorytmie walidacji danych wejściowych, trzeba zazwyczaj wykonać następujące kroki:

1. Odczytać dane wejściowe jako串 znaków.
2. Sprawdzić, czy串 znaków da się skonwertować na docelowy typ danych.
3. Jeżeli串 da się skonwertować, należy dokonać konwersji i przejść do dalszej części programu; w przeciwnym razie należy wyświetlić komunikat błędu i poprosić użytkownika o ponowne wprowadzenie danych.

Dokładna walidacja danych wejściowych wymaga też sprawdzenia ich sensowności. Nawet jeżeli użytkownik wprowadzi dane prawidłowego typu, mogą być one pozbawione sensu. Wyobraź sobie następujące scenariusze:

- Podczas wprowadzania adresu zamieszkania na terenie Stanów Zjednoczonych należy sprawdzić, czy skrót oznaczający stan składa się dokładnie z dwóch liter oraz czy prawidłowy jest kod pocztowy. Przykładowo nie istnieje stan o skrócie NW. Podobną walidację można przeprowadzić na adresach w innych krajach. Na przykład prowincje w Kanadzie oznaczane są skrótem dwuliterowym.
- Kiedy użytkownik wprowadza adres zamieszkania na terenie Stanów Zjednoczonych, należy zweryfikować, czy kod pocztowy ma prawidłowy format (5 lub 7 cyfr) oraz czy kod ten jest prawidłowy. Przykładowo kod pocztowy 99999 nie jest prawidłowy. Dodatkowo należy sprawdzić, czy dany kod pocztowy należy do określonego stanu. Bazy danych zawierające poprawne kody pocztowe można nabyć za niewielkie pieniądze — programista korzysta z takiej bazy podczas walidowania danych wejściowych.
- Należy sprawdzać kwoty wynagrodzenia i upewnić się, że są one wartościami liczbowymi mieszczącymi się w przedziale określonym przez dane przedsiębiorstwo.
- Należy weryfikować daty. Przykładowo data 29 lutego powinna zostać przyjęta tylko w przypadku, gdy rok jest przestępny, a nieprawidłowe daty, takie jak 30 lutego, powinny zostać odrzucone.
- Należy weryfikować wartości czasu. Przykładowo tydzień składa się ze 168 godzin, więc program obliczający tygodniowe wynagrodzenie powinien sprawdzać, czy użytkownik wprowadza prawidłową liczbę przepracowanych godzin — nie większą niż 168.
- Walidując dane, należy też zachować zdrowy rozsądek. Chociaż tydzień składa się ze 168 godzin, jest niemożliwe, aby pracownik pracował 24 godziny na dobę przez 7 dni w tygodniu. Podobnie jest z datami — przykładowo data urodzenia nie może być późniejsza niż bieżący dzień, a osoba nie może mieć więcej niż 150 lat. Kiedy użytkownik wprowadzi takie podejrzane dane, program powinien przynajmniej zapytać go, czy na pewno chce je wprowadzić.

## 7.4

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

## Java

W tym rozdziale nie omówię nowych funkcji Javy. Zaprezentuję natomiast program w pseudokodzie z listingu 7.2 zapisany w języku Java. Program widoczny na listingu 7.3 wykorzystuje pętlę walidacji wejścia z wierszy od 42. do 47. w celu sprawdzenia, czy wprowadzona przez użytkownika wartość nie jest ujemna.

### **Listing 7.3. (InputValidation.java)**

```

1 import java.util.Scanner;
2
3 public class InputValidation
4 {
5     public static void main(String[] args)
6     {
7         // Tworzenie obiektu Scanner w celu odczytania danych z klawiatury
8         Scanner keyboard = new Scanner(System.in);
9
10        // Zmienna lokalna
11        String doAnother;
12
13        do
14        {
15            // Obliczamy i wyświetlamy cenę detaliczną
16            showRetail();
17
18            // Jeszcze raz?
19            System.out.print("Czy chcesz przejść do kolejnego produktu? (Jeśli tak,
20                           wpisz t): ");
21            doAnother = keyboard.next();
22        } while (doAnother.equals("t") || doAnother.equals("T"));
23
24        // Moduł showRetail pobiera od użytkownika cenę hurtową
25        // produktu i wyświetla jego cenę detaliczną
26        public static void showRetail()
27        {
28            // Tworzenie obiektu Scanner w celu odczytania danych z klawiatury
29            Scanner keyboard = new Scanner(System.in);
30
31            // Zmienne lokalne
32            double wholesale, retail;
33
34            // Stała zawierająca marżę
35            final double MARKUP = 2.5;
36
37            // Pobieramy cenę hurtową
38            System.out.print("Wprowadź cenę hurtową produktu. ");
39            wholesale = keyboard.nextDouble();
40
41            // Walidujemy cenę hurtową
42            while (wholesale < 0)
43            {
44                System.out.println("Cena nie może być ujemna.");
45                System.out.print("Wprowadź prawidłową cenę hurtową.");
46                wholesale = keyboard.nextDouble();
47            }

```

```

48
49     // Obliczamy cenę detaliczną
50     retail = wholesale * MARKUP;
51
52     // Wyświetlamy cenę detaliczną
53     System.out.println("Cena detaliczna wynosi " + retail + " zł.");
54 }
55 }
```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę hurtową produktu. -1 [Enter]

Cena nie może być ujemna.

Wprowadź prawidłową cenę hurtową. 1.50 [Enter]

Cena detaliczna wynosi 3.75 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t) n [Enter]

## **Python**

W tym rozdziale nie omówię nowych funkcji Pythona. Zaprezentuję natomiast program w pseudokodzie z listingu 7.2 zapisany w języku Python. Program widoczny na listingu 7.4 wykorzystuje pętlę walidacji wejścia z wierszy od 28. do 30. w celu sprawdzenia, czy wprowadzona przez użytkownika wartość nie jest ujemna.

### **Listing 7.4. (retail.py)**

```

1  # Ten program wylicza ceny detaliczne
2
3  # Zmienna MARK_UP używana jest jako globalna stała
4  # wyznaczająca wysokość marży
5  MARK_UP = 2.5
6
7  # Funkcja main
8  def main():
9      # Zmienna sterująca pętlą
10     another = 'y'
11
12     # Obsługa przynajmniej jednego produktu
13     while another == 't' or another == 'T':
14         # Wyświetlenie ceny detalicznej produktu
15         show_retail()
16
17         # Jeszcze raz?
18         another = input('Czy chcesz przejść do kolejnego produktu? ' +
19                         '(Jeśli tak, wpisz t): ')
20
21     # Moduł showRetail pobiera od użytkownika cenę hurtową
22     # i wyświetla cenę detaliczną
23     def show_retail():
24         # Pobieramy cenę hurtową produktu
25         wholesale = float(input("Wprowadź cenę hurtową produktu. "))
26
27         # Walidujemy cenę hurtową
28         while wholesale < 0:
29             print('Cena nie może być ujemna.')
30             wholesale = float(input('Wprowadź prawidłową cenę hurtową. '))
```

```

31
32     # Obliczamy cenę detaliczną
33     retail = wholesale * MARK_UP
34
35     # Wyświetlamy cenę detaliczną
36     print('Cena detaliczna wynosi ', format(retail, '.2f'), ' zł.')
37
38     # Wywołanie funkcji main
39     main()

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę hurtową produktu. **-50 [Enter]**

Cena nie może być ujemna.

Wprowadź prawidłową cenę hurtową. **0.50 [Enter]**

Cena detaliczna wynosi 1.25 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t) **n [Enter]**

**C++**

W tym rozdziale nie omówię nowych funkcji C++. Zaprezentuję natomiast program w pseudokodzie z listingu 7.2 zapisany w języku C++. Program widoczny na listingu 7.5 wykorzystuje pętlę walidacji wejścia z wierszy od 41. do 46. w celu sprawdzenia, czy wprowadzona przez użytkownika wartość nie jest ujemna.

**Listing 7.5. (validation.cpp)**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // Prototyp funkcji
6 void showRetail();
7
8 int main()
9 {
10     // Zmienna lokalna
11     string doAnother;
12
13     do
14     {
15         // Obliczamy i wyświetlamy cenę detaliczną
16         showRetail();
17
18         // Jeszcze raz?
19         cout << "Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)" << endl;
20         cin >> doAnother;
21     } while (doAnother == "t" || doAnother == "T");
22
23     return 0;
24 }
25
26 // Moduł showRetail pobiera od użytkownika cenę hurtową
27 // i wyświetla cenę detaliczną
28 void showRetail()
29 {

```

```

30 // Zmienne lokalne
31 double wholesale, retail;
32
33 // Stała zawierająca marżę
34 const double MARKUP = 2.5;
35
36 // Pobieramy cenę hurtową
37 cout << "Wprowadź cenę hurtową produktu." << endl;
38 cin >> wholesale;
39
40 // Walidujemy cenę hurtową
41 while (wholesale < 0)
42 {
43     cout << "Cena nie może być ujemna." << endl;
44     cout << "Wprowadź prawidłową cenę hurtową." << endl;
45     cin >> wholesale;
46 }
47
48 // Obliczamy cenę detaliczną
49 retail = wholesale * MARKUP;
50
51 // Wyświetlenie ceny detalicznej.
52 cout << "Cena detaliczna wynosi " << retail << " zł." << endl;
53 }
```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź cenę hurtową produktu.

-1 [Enter]

Cena nie może być ujemna.

Wprowadź prawidłową cenę hurtową.

1.50 [Enter]

Cena detaliczna wynosi 3.75 zł.

Czy chcesz przejść do kolejnego produktu? (Jeśli tak, wpisz t)

n [Enter]

## **Pytania kontrolne**

### **Test jednokrotnego wyboru**

1. Skrót GIGO oznacza:
  - a) great input, great output
  - b) garbage in, garbage out
  - c) GIGahertz Output
  - d) GIGabyte Operation
2. Integralność danych wyjściowych jest taka jak integralność \_\_\_\_\_.
  - a) kompilatora
  - b) języka programowania
  - c) danych wejściowych
  - d) debuggera

3. Operacja odczytująca dane, która ma miejsce jeszcze przed pętlą walidacji danych, nazywa się \_\_\_\_\_.  
 a) odczytem przedwalidacyjnym  
 b) odczytem zasadniczym  
 c) odczytem inicjalizującym  
 d) odczytem wstępny
4. Pętlę walidacji danych wejściowych nazywa się także \_\_\_\_\_.  
 a) pułapką na błędy  
 b) pętlą zagłady  
 c) pętlą omijania błędów  
 d) pętlą programowania defensywnego
5. Pusty odczyt ma miejsce wtedy, gdy \_\_\_\_\_.  
 a) użytkownik wprowadzi znak *Spacji*, a następnie naciśnie klawisz *Enter*  
 b) instrukcja chce pobrać od użytkownika dane, ale użytkownik ich nie wprowadzi  
 c) użytkownik wprowadzi wartość 0, która jest niepoprawna  
 d) użytkownik wprowadzi błędne dane

### Prawda czy fałsz?

1. Proces walidacji danych wejściowych wygląda następująco: gdy użytkownik programu wprowadzi nieprawidłowe dane, program powinien wyświetlić komunikat: *Czy na pewno chcesz wprowadzić te dane?*. Gdy użytkownik odpowie „tak”, program powinien przyjąć wprowadzone dane.
2. Odczyt wstępny ma miejsce wewnętrz pętli walidującej dane wejściowe.
3. W przypadku wykorzystania do walidacji danych pętli, w której warunek jest sprawdzany na końcu, należy najpierw dokonać odczytu wstępnego.

### Krótką odpowiedź

1. Co oznacza określenie „garbage in, garbage out”?
2. Opisz krótko zasadę działania pętli walidacji danych wejściowych.
3. Do czego służy odczyt wstępny?
4. W rozdziale pokazałem, że do walidacji danych wejściowych można także wykorzystać pętlę, w której warunek sprawdzany jest na końcu. Dlaczego takie rozwiązanie nie jest zalecane?

### Warsztat projektanta algorytmów

1. Zaprojektuj algorytm, w którym użytkownik proszony jest o wprowadzenie dodatniej, niezerowej liczby i który dokonuje walidacji tej liczby.
2. Zaprojektuj algorytm, w którym użytkownik proszony jest o wprowadzenie liczby z przedziału 1 – 100 i który dokonuje walidacji tej liczby.
3. Zaprojektuj algorytm, w którym użytkownik proszony jest o wprowadzenie tekstu „tak” lub „nie” i który dokonuje walidacji tego ciągu znaków (porównaj tekst bez rozróżnienia wielkości znaków).
4. Zaprojektuj algorytm, w którym użytkownik proszony jest o wprowadzenie liczby większej niż 99 i który dokonuje walidacji tej liczby.

5. Zaprojektuj algorytm, w którym użytkownik proszony jest o wprowadzenie tajnego słowa. Słowo to powinno się składać z co najmniej 8 znaków. Dokonaj walidacji danych wejściowych.

## Ćwiczenia z wykrywania błędów

1. Dlaczego poniższy program nie zadziała zgodnie z opisem umieszczonym w komentarzach?

```
// Program prosi użytkownika o wprowadzenie liczby
// z przedziału od 1 do 10, a następnie ją waliduje
Declare Integer value

// Pobieramy liczbę od użytkownika
Display "Wprowadź liczbę z przedziału od 1 do 10."
Input value

// Sprawdzamy, czy liczba mieści się w przedziale od 1 do 10
While value < 1 AND value > 10
    Display "BŁĄD! Liczba musi mieścić się w przedziale od 1 do 10."
    Display "Wprowadź liczbę z przedziału od 1 do 10."
    Input value
End While
```

2. Dlaczego poniższy program nie zadziała zgodnie z opisem umieszczonym w komentarzach?

```
// Program pobiera od użytkownika cenę
// i ją waliduje
Declare Real amount

// Pobieramy od użytkownika
Display "Wprowadź cenę."
Input amount

// Sprawdzamy, czy cena jest mniejsza od 0. Jeśli tak,
// pobieramy cenę ponownie
While amount < 0
    Display "BŁĄD: Cena nie może być mniejsza niż 0."
    Display "Wprowadź cenę."
End While
```

3. Poniższy pseudokod działa prawidłowo, ale podczas walidacji danych rozróżnia wielkość znaków. W jaki sposób można poprawić ten algorytm, aby użytkownik, wprowadzając imię, nie musiał zwracać uwagi na wielkość znaków?

```
// Program prosi użytkownika o wprowadzenie ciągu znaków,
// a następnie go waliduje
Declare String choice

// Pobieramy od użytkownika odpowiedź na pytanie
Display "Oddaj swój głos na kapitana drużyny szachowej."
Display "Czy chcesz, aby kapitanem był Tomek czy Kasia?"
Input choice

// Walidujemy dane wejściowe
While choice != "Tomek" AND choice != "Kasia"
    Display "Wprowadź imiona Tomek lub Kasia."
```

```

Display "Oddaj swój głos na kapitana drużyny szachowej."
Display "Czy chcesz, aby kapitanem był Tomek czy Kasia?"
Input response
End While

```

## Ćwiczenia programistyczne

### 1. Obliczanie wynagrodzenia z walidacją danych wejściowych

Zaprojektuj program do obliczania wynagrodzenia, który będzie prosił użytkownika o wprowadzenie stawki godzinowej danego pracownika i liczby przepracowanych przez niego godzin. Dokonaj walidacji danych wejściowych, tak aby użytkownik mógł wprowadzić tylko wartości stawki godzinowej z przedziału od 7,50 do 18,25 złotych i liczbę przepracowanych godzin z przedziału od 0 do 40. Program powinien wyświetlić wynagrodzenie całkowite pracownika.

### 2. Wpływ ze sprzedaży biletów i walidacja danych wejściowych

W teatrze miejsca siedzące są podzielone na trzy sektory. Bilety w poszczególnych sektorach kosztują: w sektorze A: 20 złotych, w sektorze B: 15 złotych, a w sektorze C: 10 złotych. W sektorze A znajduje się 300 miejsc, w sektorze B — 500 miejsc, a w sektorze C — 200 miejsc. Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby sprzedanych biletów w każdym z sektorów, a następnie wyświetli sumę wpływów ze sprzedaży biletów. Program powinien dokonywać walidacji danych wejściowych.

### 3. Kalkulator dietetyczny

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie liczby gramów tłuszcza i kalorii zawartych w danym produkcie spożywczym.

Dokonaj następującej walidacji danych wejściowych:

- Liczba gramów tłuszcza i kalorii nie może być mniejsza niż 0.
- Z poniższego wzoru wynika, że liczba kalorii nie może być większa niż  $\text{gramy tłuszcza} \cdot 9$ . Upewnij się więc, że użytkownik nie będzie mógł wprowadzić w programie liczby kalorii większej niż  $\text{gramy tłuszcza} \cdot 9$ .

Po wprowadzeniu prawidłowych danych program powinien obliczyć i wyświetlić procent kalorii pochodzących z tłuszcza. Skorzystaj z następującego wzoru:

$$\text{procent kalorii z tłuszcza} = (\text{gramy tłuszcza} \cdot 9) : \text{liczba kalorii}$$

Według niektórych dietetyków produkt jest niskotłuszczowy wtedy, gdy mniej niż 30% zawartych w nim kalorii pochodzi z tłuszcza. Jeżeli wynikiem obliczeń będzie wartość mniejsza niż 0,3, program powinien wyświetlić informację, że dany produkt jest niskotłuszczowy.

### 4. Przekroczenie dopuszczalnej prędkości

Zaprojektuj program, który będzie obliczał i wyświetlał informację, o jaką wartość kierowca przekroczył dozwoloną prędkość. Program powinien poprosić

użytkownika o wprowadzenie wartości ograniczenia prędkości oraz prędkości, z jaką poruszał się samochód. Dokonaj walidacji danych wejściowych w następujący sposób:

- Ograniczenie prędkości musi wynosić co najmniej 40, ale nie może być wyższe niż 140.
- Prędkość samochodu powinna być co najmniej równa wprowadzonej wcześniej wartości ograniczenia prędkości (w przeciwnym razie kierowca nie popełniłby wykroczenia).

Po wprowadzeniu prawidłowych danych program powinien obliczyć i wyświetlić informację, o ile km/h kierowca przekroczył prędkość.

## 5. Papier, kamień, nożyce — modyfikacja

W ćwiczeniu programistycznym 11. zamieszczonym w rozdziale 6. poprosilem Cię o zaprojektowanie programu będącego grą w papier, kamień, nożyce. W programie tym użytkownik wprowadzał jeden z trzech ciągów znaków: "papier", "kamień", "nożyce". Dodaj do programu walidację danych wejściowych (bez rozróżnienia wielkości znaków), dzięki której użytkownik nie będzie mógł wprowadzić błędnego ciągu znaków.



**TEMATYKA**

- |                                        |                                           |
|----------------------------------------|-------------------------------------------|
| 8.1 Tablice — informacje podstawowe    | 8.4 Tablice równoległe                    |
| 8.2 Sekwencyjne przeszukiwanie tablicy | 8.5 Tablice dwuwymiarowe                  |
| 8.3 Przetwarzanie elementów tablicy    | 8.6 Tablice trój- i więcej wymiarowe      |
|                                        | 8.7 Rzut oka na języki Java, Python i C++ |

**8.1****Tablice — informacje podstawowe**

**WYJAŚNIENIE:** Dzięki tablicom możesz przechowywać w pamięci komputera pewną grupę elementów tego samego typu danych. Zazwyczaj przetwarzanie dużej liczby elementów zapisanych w tabeli jest znacznie szybsze niż przetwarzanie wartości zapisanych w osobnych zmiennych.

W programach, które tworzyliśmy do tej pory, wszystkie dane zapisane były w pamięci jako zmienne. W większości języków programowania to właśnie przechowywanie wartości w zmiennej jest najprostszym sposobem na jej zapisanie. Takie rozwiązanie sprawdza się w bardzo wielu przypadkach, jednak jest ono w pewien sposób ograniczające. Przykładowo zmienna może przechowywać tylko jedną wartość. Przyjrzyjmy się następującej deklaracji zmiennej:

```
Declare Integer number = 99
```

W instrukcji tej zadeklarowałem zmienną typu Integer o nazwie number i zainicjalizowałem ją wartością 99. Zwróć uwagę, co się stanie, jeżeli w dalszej części programu znajdzie się takie polecenie:

```
Set number = 5
```

Do zmiennej `number` zostanie przypisana wartość 5, która zastąpi zapisaną w niej wcześniej wartość 99. Ponieważ zmienna `number` jest zwykłą zmienną, można w niej zapisać tylko pojedynczą wartość.

Zmienne mogą przechowywać tylko jedną wartość, dlatego korzystanie z nich w programach przetwarzających całe listy danych jest bardzo uciążliwe. Założymy, że ktoś poprosił Cię o zaprojektowanie programu, który będzie przechowywał listę 50 pracowników. Wyobraź sobie, jak wyglądałby program, gdybyś musiał zadeklarować w tym celu 50 zmiennych:

```
Declare String employee1
Declare String employee2
Declare String employee3
itd.
Declare String employee50
```

Następnie wyobraź sobie kod programu, w którym musiałbyś wprowadzić 50 nazwisk:

```
// Pobieramy nazwisko pierwszego pracownika
Display "Wprowadź nazwisko pracownika 1."
Input employee1

// Pobieramy nazwisko drugiego pracownika
Display " Wprowadź nazwisko pracownika 2."
Input employee2

// Pobieramy nazwisko trzeciego pracownika
Display " Wprowadź nazwisko pracownika 3."
Input employee3

itd.

// Pobieramy nazwisko pięćdziesiątego pracownika
Display "Wprowadź nazwisko pracownika 50."
Input employee50
```

Jak widzisz, zmienne nie nadają się do przechowywania i przetwarzania dużych zbiorów danych. Dla każdego elementu w zbiorze musimy zadeklarować osobną zmienną, a następnie ją przetwarzać. Na szczęście w większości języków programowania występują **tablice**, które zostały zaprojektowane specjalnie z myślą o przetwarzaniu zbiorów danych. Podobnie jak zmienna, tablica to nazwane miejsce w pamięci komputera. Jednak w odróżnieniu od zmiennej w tablicy można zapisać wiele wartości. Wszystkie wartości zapisane w tablicy muszą być tego samego typu. Można więc stworzyć tablicę liczb typu `Integer` lub `Real` albo `String`, ale nie można w jednej tablicy mieścić tych typów. Oto, w jaki sposób będę deklarował tablicę w pseudokodzie:

```
Declare Integer units[10]
```

Zwróć uwagę, że polecenie to wygląda podobnie jak deklaracja zwykłej zmiennej, z tą różnicą, że na końcu znajdują się nawiasy kwadratowe. Liczba w tych nawiasach wskazuje **rozmiar tablicy**, czyli liczbę elementów, które będzie można zapisać w tablicy. Za pomocą powyższej instrukcji zadeklarowałem tablicę o nazwie `units`, w której będzie można zapisać 10 liczb całkowitych. W większości języków programowania rozmiar tablicy musi być liczbą całkowitą nieujemną. Oto inny przykład:

```
Declare Real salesAmounts[7]
```

Za pomocą tej instrukcji zadeklarowałem tablicę o nazwie `salesAmounts`, w której będzie można zapisać 7 liczb rzeczywistych. Poniżej przedstawiłem jeszcze jeden przykład. Za pomocą tego polecenia zadeklarowałem tablicę, w której będzie można zapisać 50 ciągów znaków. Tablica nazywa się `names`.

```
Declare String names[50]
```

W większości języków programowania nie można zmienić rozmiaru tablicy w trakcie działania programu. Jeżeli stworzysz program, w którym wykorzystasz tablicę, i w pewnym momencie okaże się, że ma ona nieodpowiedni rozmiar, będziesz musiał zmienić rozmiar tablicy w kodzie programu, a następnie ponownie skompilować program (lub w przypadku interpretera uruchomić go ponownie) już z poprawionym rozmiarem tablicy. Aby łatwiej było zarządzać tablicami, programiści często wolą zapisywać rozmiar tablicy w stałych nazwanych. Oto przykład:

```
Constant Integer SIZE = 10
Declare Integer units[SIZE]
```

Jak będziesz mieć okazję zaobserwować w dalszej części rozdziału, w przypadku wielu technik przetwarzania tablic trzeba się odnosić do ich rozmiaru. Określając rozmiar tablicy za pomocą stałej, można w algorytmach posługiwać się właśnie tą stałą. Jeżeli kiedykolwiek przyjdzie Ci zmodyfikować program tak, aby tablica miała inny rozmiar, wystarczy, że zmienisz tylko wartość przypisaną do stałej.



**UWAGA:** W tej książce do oznaczania rozmiaru tablicy będę używał nawiasów kwadratowych. W niektórych językach programowania służą do tego celu nawiasy okrągłe.

## Elementy tablicy i indeksy

Poszczególne komórki w tablicy nazywamy **elementami**. Elementy tablicy są zazwyczaj umieszczone w pamięci w następujących po sobie komórkach pamięci. Do każdego elementu tablicy jest przypisany unikatowy identyfikator liczbowy zwany **indeksem**. Za pomocą indeksów można wskazywać konkretne elementy tablicy. W większości języków programowania pierwszy element tablicy ma indeks 0, drugi element ma indeks 1 itd. Założmy, że w kodzie znajdują się następujące deklaracje:

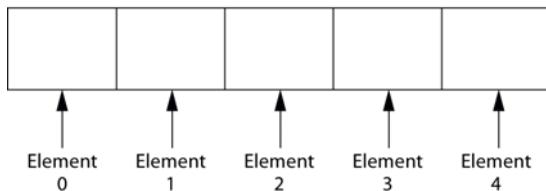
```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE]
```

Jak widać na rysunku 8.1, tablica `numbers` składa się z pięciu elementów. Do elementów są przypisane kolejno indeksy od 0 do 4 (ponieważ numeracja indeksów zaczyna się od 0, ostatni indeks jest o 1 mniejszy od rozmiaru tablicy).



**UWAGA:** W niektórych językach programowania numeracja indeksów zaczyna się od 1. Ponieważ jednak znacznie częściej można się spotkać z numerowaniem indeksów od 0, takiego systemu będę używał w tej książce.

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE]
```



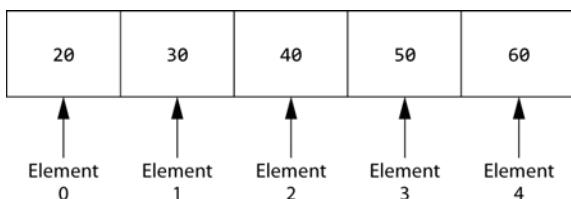
**Rysunek 8.1.** Indeksy tablicy

## Przypisywanie wartości do elementów tablicy

Do poszczególnych elementów tablicy odnosimy się za pomocą indeksów. W przypadku zadeklarowanej wcześniej tablicy `number` zawierającej liczby typu `Integer` możemy przypisać wartości do jej pięciu elementów w następujący sposób:

```
Set numbers[0] = 20
Set numbers[1] = 30
Set numbers[2] = 40
Set numbers[3] = 50
Set numbers[4] = 60
```

W powyższym pseudokodzie do elementu o indeksie 0 została przypisana wartość 20, do elementu 1 — wartość 30 itd. Na rysunku 8.2 widoczna jest zawartość tablicy po wykonaniu powyższych instrukcji.



**Rysunek 8.2.** Do elementów tablicy zostały przypisane wartości

## Wprowadzanie i wyświetlanie wartości zapisanych w tablicy

Podobnie jak w przypadku zmiennych, można zapisywać w tablicy wartości wprowadzone przez użytkownika i wyświetlać na ekranie wartość przypisaną do danego elementu tablicy. Pseudokod na listingu 8.1 prezentuje, w jaki sposób można zapisać w tablicy wartości wprowadzone przez użytkownika, a następnie wyświetlić je na ekranie.

Przyjrzyjmy się bliżej temu programowi. W linii 2. deklaruję stałą `SIZE` i inicjalizuję ją wartością 3. Następnie w linii 6. deklaruję tablicę typu `Integer` o nazwie `hours`.

**Listing 8.1**

```

1 // Tworzymy stałą, w której zapisana jest liczba pracowników
2 Constant Integer SIZE = 3
3
4 // Deklarujemy tablicę, w której zapisana będzie liczba godzin
5 // przepracowanych przez każdego pracownika
6 Declare Integer hours[SIZE]
7
8 // Pobieramy liczbę godzin przepracowanych przez pracownika 1
9 Display "Wprowadź liczbę godzin przepracowanych przez pracownika 1."
10 Input hours[0]
11
12 // Pobieramy liczbę godzin przepracowanych przez pracownika 2
13 Display "Wprowadź liczbę godzin przepracowanych przez pracownika 2."
14 Input hours[1]
15
16 // Pobieramy liczbę godzin przepracowanych przez pracownika 3
17 Display "Wprowadź liczbę godzin przepracowanych przez pracownika 3."
18 Input hours[2]
19
20 // Wyświetlamy wprowadzone wartości
21 Display "Wprowadziłeś następujące wartości:"
22 Display hours[0]
23 Display hours[1]
24 Display hours[2]

```

**Wynik działania programu (po grubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę godzin przepracowanych przez pracownika 1.

40 [Enter]

Wprowadź liczbę godzin przepracowanych przez pracownika 2.

20 [Enter]

Wprowadź liczbę godzin przepracowanych przez pracownika 3.

15 [Enter]

Wprowadziłłeś następujące wartości:

40

20

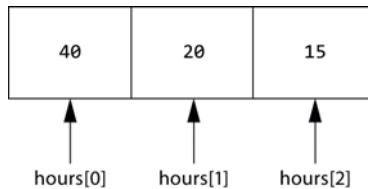
15

W deklaracji do określenia rozmiaru tablicy posługuję się stałą SIZE — tablica będzie się więc składała z trzech elementów. Polecenia Input w liniach 10., 14. i 18. odczytują wartości wprowadzone na klawiaturze i zapisują je do poszczególnych elementów tablicy hours. Następnie w liniach od 22. do 24. za pomocą polecenia Display wyświetlам na ekranie wartości przypisane do poszczególnych elementów tablicy.

Podczas przykładowego uruchomienia programu użytkownik wprowadził liczby 40, 20 i 15, które zostały zapisane w tablicy hours. Na rysunku 8.3 przedstawiona jest zawartość tablicy po przypisaniu do niej wartości.

**Śledzenie tablicy za pomocą pętli**

W większości języków programowania do elementu tablicy można się także odnosić poprzez liczbę zisaną w zmiennej. Dzięki temu do prześledzenia wszystkich elementów tablicy możemy wykorzystać pętlę. Spójrz na przykład na listingu 8.2.

**Rysunek 8.3.** Zawartość tablicy hours**Listing 8.2**

```

1 // Deklarujemy tablicę 10 elementów typu Integer
2 Declare Integer series[10]
3
4 // Deklarujemy zmienną, której użyjemy w pętli
5 Declare Integer index
6
7 // Do każdego elementu przypisujemy wartość 100
8 For index = 0 To 9
9     Set series[index] = 100
10 End For

```

W linii 2. deklaruję tablicę series typu Integer składającą się z 10 elementów, a w linii 5. deklaruję zmienną Integer o nazwie index. Zmienna index pełni funkcję licznika w pętli For, która pojawia się w liniach od 8. do 10. Do zmiennej index podczas działania pętli będą przypisywane kolejno wartości od 0 do 9. Podczas pierwszej iteracji pętli w zmiennej index będzie zapisana wartość 0, więc polecenie w linii 9. przypisze do elementu series[0] wartość 100. Podczas drugiej iteracji w zmiennej index będzie zapisana wartość 1, więc do elementu series[1] także zostanie przypisana wartość 100. Operacja ta będzie powtarzana aż do momentu, gdy w ostatniej iteracji do elementu series[9] zostanie przypisana wartość 100.

Spójrz na kolejny przykład. Program z listingu 8.1 możemy uprościć dzięki wykorzystaniu dwóch pętli For: jedna posłuży do wprowadzania wartości do tablicy, a druga do wyświetlania wartości w tablicy. Demonstруje to listing 8.3.

**Listing 8.3**

```

1 // Tworzymy stałą, w której zapisany będzie rozmiar tablicy
2 Constant Integer SIZE = 3
3
4 // Deklarujemy tablicę, w której zapisana będzie liczba godzin
5 // przepracowanych przez każdego pracownika
6 Declare Integer hours[SIZE]
7
8 // Deklarujemy zmienną, której użyjemy w pętli
9 Declare Integer index
10
11 // Pobieramy liczbę godzin przepracowanych przez każdego pracownika
12 For index = 0 To SIZE - 1
13     Display "Wprowadź liczbę godzin przepracowanych przez"
14     Display "pracownika numer ", index + 1
15     Input hours[index]

```

```

16 End For
17
18 // Wyświetlamy wprowadzone wartości
19 Display "Wprowadziłeś następujące wartości:"
20 For index = 0 To SIZE - 1
21   Display hours[index]
22 End For

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę godzin przepracowanych przez pracownika numer 1

**40 [Enter]**

Wprowadź liczbę godzin przepracowanych przez pracownika numer 2

**20 [Enter]**

Wprowadź liczbę godzin przepracowanych przez pracownika numer 3

**15 [Enter]**

Wprowadziłeś następujące wartości:

40

20

15

Przyjrzyjmy się bliżej pierwszej pętli For — pojawia się ona w liniach od 12. do 16. Oto pierwsza linia pętli:

```
For index = 0 To SIZE - 1
```

Wskazujmy tutaj, że do zmiennej index będą przypisywane w kolejnych iteracjach wartości od 0 do 2. Ale dlaczego użyłem tutaj jako wartości końcowej licznika wyrażenia SIZE - 1? Przypominam, że indeks ostatniego elementu w tablicy jest o 1 mniejszy od jej rozmiaru. W tym przypadku ostatni indeks powinien być równy 2, czyli wartość wyrażenia SIZE - 1.

Zwróć uwagę, że w pętli w linii 15. wykorzystuję zmienną index jako indeks tablicy:

```
Input hours[index]
```

Podczas pierwszej iteracji pętli w zmiennej index będzie zapisana wartość 0, więc liczba wprowadzona przez użytkownika zostanie zapisana do elementu hours[0]. Podczas kolejnej iteracji wartość zostanie zapisana do elementu hours[1]. Następnie, podczas ostatniej iteracji, będzie to element hours[2]. Zauważ, że w pętli do zmiennej index przypisywane są poprawne numery indeksów tablicy (od 0 do 2).

Pozostało mi jeszcze wyjaśnienie jednej rzeczy w programie z listingu 8.3. Program podczas odczytywania danych odnosi się do trzech pracowników jako: „pracownik numer 1”, „pracownik numer 2” i „pracownik numer 3”. Polecenia Display umieszczone w pętli w liniach 13. i 14. wyglądają następująco:

```

Display "Wprowadź liczbę godzin przepracowanych przez"
Display "pracownika numer ", index + 1

```

Zwróć uwagę, że w drugim poleceniu numer pracownika wyświetla się jako wartość wyrażenia index + 1. Jak myślisz, co by się stało, gdybyśmy usunęli z wyrażenia fragment + 1 i zapisali je w następujący sposób?

```
Display "Wprowadź liczbę godzin przepracowanych przez"
Display "pracownika numer ", index
```

Ponieważ do zmiennej `index` pętla będzie przypisywała kolejno wartości 0, 1 i 2, na ekranie wyświetliłby się tekst „pracownik numer 0”, „pracownik numer 1” i „pracownik numer 2”. Większość użytkowników programu uznałaby takie numerowanie osób lub przedmiotów za dziwne, więc użyłem wyrażenia `index + 1`, aby numeracja zaczynała się od 1.

## Przetwarzanie elementów tablicy

Przetwarzanie elementów tablicy nie różni się niczym od przetwarzania zwykłych zmiennych. W przedstawionych wcześniej przykładach pokazałem, w jaki sposób do elementów tablicy przypisuje się wartości, zapisuje się w nich dane wprowadzone przez użytkownika oraz jak można wyświetlić zawartość poszczególnych elementów tablicy. W sekcji „W centrum uwagi” pokażę, jak wykorzystać elementy tablicy w wyrażeniach matematycznych.

### W centrum uwagi

#### Korzystanie z elementów tablicy w wyrażeniach matematycznych

Gosia jest właścicielką małej kawiarni i zatrudnia na stanowisku baristy trzech pracowników. Wszyscy pracownicy mają taką samą stawkę godzinową. Gosia poprosiła Cię o zaprojektowanie programu, który umożliwi jej wprowadzenie liczby godzin przepracowanych przez każdego pracownika, a następnie wyświetli należne wynagrodzenie całkowite. Doszedłeś do wniosku, że program powinien wykonywać następujące operacje:

1. Pobranie liczby godzin przepracowanych przez każdego pracownika i zapisanie tych wartości w tablicy.
2. Dla każdego elementu tablicy obliczenie wynagrodzenia na podstawie zapisanych w niej danych, a następnie wyświetlenie go na ekranie.

Na listingu 8.4 znajduje się pseudokod programu, a na rysunku 8.4 jego schemat blokowy.

### Listing 8.4

```
1 // Stała zawierająca rozmiar tablicy
2 Constant Integer SIZE = 6
3
4 // Tablica z liczbą przepracowanych godzin
5 Declare Real hours[SIZE]
6
7 // Zmienna z wysokością stawki godzinowej
8 Declare Real payRate
```



```

9
10 // Zmienna z wynagrodzeniem całkowitym
11 Declare Real grossPay
12
13 // Zmienna licznikowa
14 Declare Integer index
15
16 // Pobieramy liczbę godzin przepracowanych przez każdego pracownika
17 For index = 0 To SIZE - 1
18   Display "Wprowadź liczbę godzin przepracowanych przez"
19   Display "pracownika ", index + 1, "."
20   Input hours[index]
21 End For
22
23 // Pobieramy stawkę godzinową
24 Display "Wprowadź stawkę godzinową."
25 Input payRate
26
27 // Wyświetlamy wynagrodzenie całkowite każdego pracownika
28 Display "Oto lista wynagrodzeń całkowitych:"
29 For index = 0 To SIZE - 1
30   Set grossPay = hours[index] * payRate
31   Display "Pracownik ", index + 1, ": ",
32     currencyFormat(grossPay)
33 End For

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę godzin przepracowanych przez  
pracownika 1.

**10 [Enter]**

Wprowadź liczbę godzin przepracowanych przez  
pracownika 2.

**20 [Enter]**

Wprowadź liczbę godzin przepracowanych przez  
pracownika 3.

**15 [Enter]**

Wprowadź liczbę godzin przepracowanych przez  
pracownika 4.

**40 [Enter]**

Wprowadź liczbę godzin przepracowanych przez  
pracownika 5.

**20 [Enter]**

Wprowadź liczbę godzin przepracowanych przez  
pracownika 6.

**18 [Enter]**

Wprowadź stawkę godzinową.

**12.75 [Enter]**

Oto lista wynagrodzeń całkowitych:

Pracownik 1: 127,50 zł

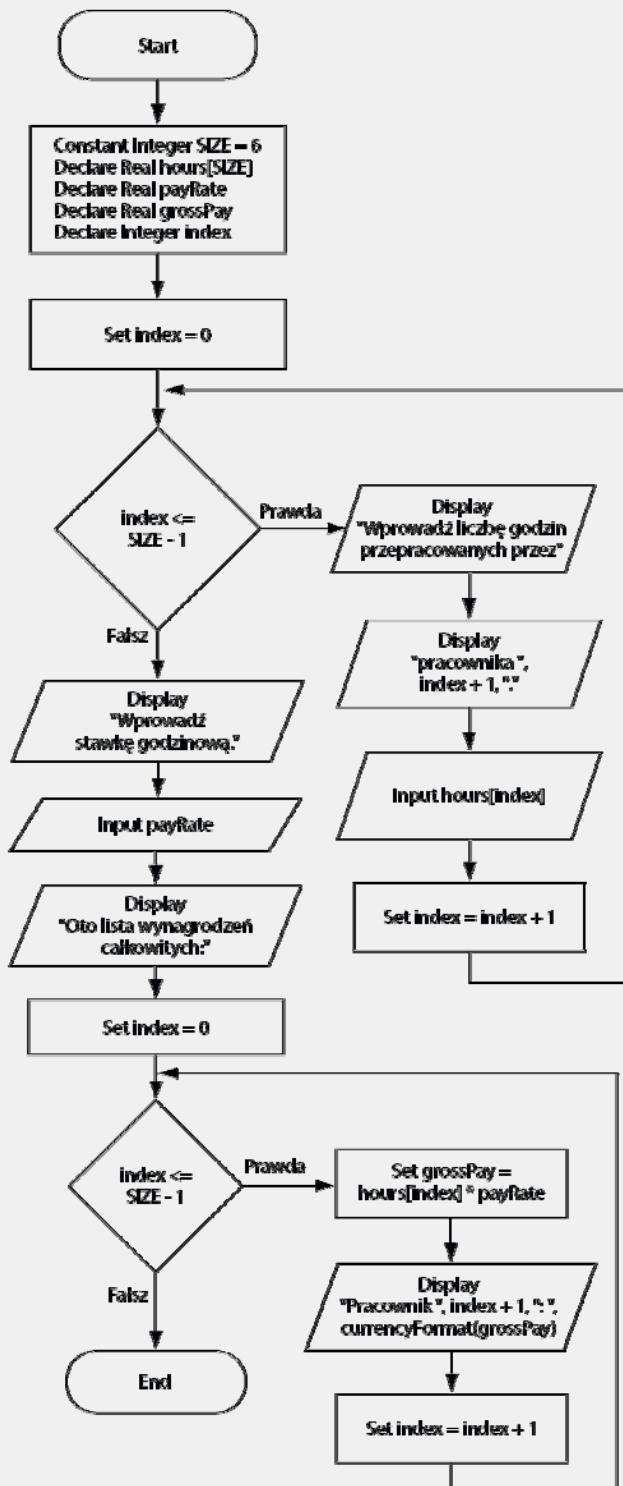
Pracownik 2: 255,00 zł

Pracownik 3: 191,25 zł

Pracownik 4: 510,00 zł

Pracownik 5: 255,00 zł

Pracownik 6: 229,50 zł



Rysunek 8.4. Schemat blokowy programu z listingu 8.4



**UWAGA:** Założmy, że kawiarnia cieszy się tak dużą popularnością, że Gosia postanowiła zatrudnić dwóch nowych baristów. Będziemy musieli zmienić program tak, aby zapisywał dane 8 pracowników, a nie 6. Z racji tego, że użyliśmy tutaj stałej nazwanej, modyfikacja będzie bardzo prosta — wystarczy w następujący sposób zmienić instrukcję w linii 2.:

```
Constant Integer SIZE = 8
```

Ponieważ stałej SIZE używamy jako rozmiaru tablicy podczas jej deklarowania w linii 5., tablica hours automatycznie zmieni rozmiar na 8. Ponadto stałej tej używamy jako wartości maksymalnej w pętli w liniach od 17. do 29., więc pętla także automatycznie wykona 8 iteracji.

Wyobraź sobie, o ile trudniejsza byłaby modyfikacja programu, gdybyśmy nie wykorzystali do określenia rozmiaru tablicy stałej nazwanej. Musielibyśmy zmienić poszczególne instrukcje, w których odwołujemy się do rozmiaru tablicy. Nie tylko mielibyśmy dodatkową pracę, ale także wzrosłoby ryzyko popełnienia błędu — jeśli zapomnielibyśmy zmodyfikować chociaż jedną instrukcję, w programie pojawiłby się błąd.



**WSKAZÓWKA:** W programach na listingach 8.1, 8.3 i 8.4 pokazałem, jak można przypisywać do poszczególnych elementów tablicy wartości wprowadzone przez użytkownika. Jednak w przypadku bardzo długiej listy wartości dane pozyskuje się zazwyczaj w inny sposób — na przykład odczytując je z pliku na dysku twardym. O odczytywaniu danych z plików i zapisywaniu ich w tablicy przeczytasz w rozdziale 10.

## Inicjalizowanie tablicy

W większości języków programowania tablicę można zainicjalizować podczas jej deklarowania. W tej książce będę inicjalizował tablicę w następujący sposób:

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE] = 10, 20, 30, 40, 50
```

Szereg wartości oddzielonych od siebie znakiem przecinka nazywamy **listą inicjalizacyjną**. Wartości te zostaną przypisane do kolejnych elementów tablicy. Pierwsza wartość (10) zostanie przypisana do elementu `numbers[0]`, druga wartość (20) zostanie przypisana do elementu `numbers[1]` itd. Oto inny przykład:

```
Constant Integer SIZE = 7
Declare String days[SIZE] = "Poniedziałek", "Wtorek", "Środa",
                           "Czwartek", "Piątek", "Sobota",
                           "Niedziela"
```

W powyższym pseudokodzie deklaruję tablicę siedmiu ciągów znaków i inicjalizuję jej pierwszy element wartością "Poniedziałek", drugi element — wartością "Wtorek" itd.

## Sprawdzanie zakresu indeksu

Większość języków programowania przeprowadza **sprawdzanie zakresu indeksu**, dzięki czemu odwołanie się w programie do nieistniejącego elementu tablicy jest niemożliwe. Spójrz na przykładowy pseudokod:

```
// Tworzymy tablice
Constant Integer SIZE = 5
Declare Integer numbers[SIZE]

// BŁĄD! W instrukcji odwołujemy się do nieistniejącego elementu!
Set numbers[5] = 99
```

W pseudokodzie deklaruję tablicę składającą się z pięciu elementów. Będzie ona więc miała indeksy od 0 do 4. W ostatniej instrukcji wystąpi błąd, ponieważ próbuję przypisać wartość do elementu `numbers[5]`, którego w tablicy nie ma.



**UWAGA:** Sprawdzanie zakresu indeksu ma zazwyczaj miejsce w momencie, gdy program jest już uruchomiony.

## Uważaj na błędy off-by-one

Ponieważ indeksy tablicy rozpoczynają się od 0, a nie od 1, musisz podczas pisania programu zachować ostrożność, aby nie popełnić błędu typu off-by-one. Błąd **off-by-one** polega na tym, że pętla wykonuje o jedną iterację za dużo lub za mało. Spójrz na następujący pseudokod:

```
// W tym kodzie pojawia się błąd off-by-one
Constant Integer SIZE = 100;
Declare Integer numbers[SIZE]
Declare Integer index
For index = 1 To SIZE - 1
    Set numbers[index] = 0
End For
```

Zadaniem tego kodu było utworzenie tablicy liczb całkowitych zawierającej 100 elementów, a następnie przypisanie do każdego elementu wartości 0. Jednak w kodzie występuje błąd off-by-one. W pętli używana jest zmienna licznikowa `index`, w której zapisywane są kolejne indeksy tablicy. W trakcie działania pętli zmienna `index` będzie przyjmowała wartości od 1 do 99, podczas gdy powinna przyjmować wartości od 0 do 99. W wyniku tego błędu pominięty zostanie pierwszy element tablicy o indeksie 0.

Załóżmy, że w poniższym przykładzie tablica `numbers` jest identyczna jak w zamieszczonym wyżej kodzie. Tutaj w pętli także wystąpi błąd off-by-one, ponieważ pętla prawidłowo rozpoczęcie działanie od indeksu 0, ale wykona o jedną iterację za dużo — zakończy działanie na indeksie 100.

```
// BŁĄD!
For index = 0 To SIZE
    Set numbers[index] = 0
End For
```

Ponieważ ostatni element tej tablicy ma indeks 99, pętla spowoduje wystąpienie błędu podczas sprawdzania zakresu indeksu.

## Tablice wypełnione częściowo

W niektórych zadaniach trzeba zapisać w tablicy szereg wartości, ale ich liczba nie będzie określona. Nie wiadomo więc, jak duża powinna być taka tablica. Jednym z rozwiązań jest stworzenie tablicy tak dużej, że będzie w stanie pomieścić najwięcej przewidywalną liczbę elementów. Prowadzi to jednak do pewnych problemów. Jeżeli liczba elementów zapisana w tablicy będzie mniejsza od jej rozmiaru, tablica będzie wypełniona tylko częściowo. Kiedy jednak będziesz przetwarzać elementy w tablicy, musisz uwzględnić tylko te, w których zapisane są prawidłowe dane.

Tablice wypełnionej częściowo towarzyszy zazwyczaj zmienna, w której zapisana jest informacja dotycząca liczby elementów przechowywanych w tablicy. Jeśli tablica jest pusta, w zmiennej tej powinna być zapisana wartość 0 — ponieważ w tablicy nie ma żadnego elementu. Za każdym razem, gdy do tablicy zostanie dodany nowy element, zmienną należy zwiększyć o 1. Następnie, gdy będziemy chcieli prześledzić w pętli zawartość tablicy, zamiast rozmiaru tablicy użyjemy zmiennej, w której zapisana jest właściwa liczba elementów. Na listingu 8.5 przedstawiłem przykład.

### **Listing 8.5**

```

1 // Deklarujemy stałą z rozmiarem tablicy
2 Constant Integer SIZE = 100
3
4 // Deklarujemy tablicę liczb całkowitych
5 Declare Integer values[SIZE]
6
7 // Deklarujemy zmienną typu Integer, gdzie zapiszemy
8 // liczbę elementów, które są tak naprawdę zapisane w tablicy
9 Declare Integer count = 0
10
11 // Deklarujemy zmienną, w której zapiszemy liczbę wprowadzoną przez użytkownika
12 Declare Integer number
13
14 // Deklarujemy zmienną licznikową
15 Declare Integer index
16
17 // Prosimy użytkownika o wprowadzenie liczby; jeżeli użytkownik wprowadzi
18 // wartość -1, zakończymy wprowadzanie danych
19 Display "Wprowadź liczbę lub -1, aby zakończyć wprowadzanie."
20 Input number
21
22 // Jeżeli użytkownik nie wprowadził -1 i tablica nie jest jeszcze zapełniona,
23 // przetwarzamy dane
24 While (number != -1 AND count < SIZE)
25     // Zapisujemy liczbę w tablicy
26     Set values[count] = number
27
28     // Inkrementujemy zmienną
29     count = count + 1
30

```

```

31 // Prosimy użytkownika o wprowadzenie kolejnej liczby
32 Display "Wprowadź liczbę lub -1, aby zakończyć wprowadzanie."
33 Input number
34 End While
35
36 // Wyświetlamy wartości zapisane w tablicy
37 Display "Oto wprowadzone przez Ciebie liczby:"
38 For index = 0 To count - 1
39   Display values[index]
40 End For

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę lub -1, aby zakończyć wprowadzanie.

**2 [Enter]**

Wprowadź liczbę lub -1, aby zakończyć wprowadzanie.

**4 [Enter]**

Wprowadź liczbę lub -1, aby zakończyć wprowadzanie.

**6 [Enter]**

Wprowadź liczbę lub -1, aby zakończyć wprowadzanie.

**-1 [Enter]**

Oto wprowadzone przez Ciebie liczby:

2

4

6

Przyjrzyjmy się dokładniej temu pseudokodowi. W linii 2. deklaruję stałą SIZE i inicjalizuję ją wartością 100. W linii 5. deklaruję tablicę liczb typu Integer o nazwie values i za pomocą stałej SIZE wskazuję jej rozmiar. W wyniku tego tablica będzie się składała ze 100 elementów. W linii 9. deklaruję zmienną typu Integer o nazwie count, w której będę zapisywać liczbę elementów tablicy values. Zwróć uwagę, że ponieważ na początku w tablicy nie będzie żadnego elementu, inicjalizuję tę zmienną wartością 0. W linii 12. deklaruję zmienną typu Integer o nazwie number, w której będę zapisywać wartość wprowadzoną przez użytkownika, a w linii 15. deklaruję zmienną typu Integer o nazwie index, która będzie pełniła w pętli funkcję zmiennej licznikowej.

W linii 19. proszę użytkownika o wprowadzenie liczby lub, jeżeli użytkownik chce zakończyć wprowadzanie liczb, wprowadzenie wartości -1. Wartość -1 pełni więc tutaj funkcje wartownika — jeżeli użytkownik wprowadzi -1, program przestanie odczytywać liczby wprowadzone z klawiatury. W linii 20. pobieram liczbę wprowadzoną przez użytkownika i zapisuję ją w zmiennej number. W linii 24. rozpoczyna się pętla While — działa ona dopóty, dopóki w zmiennej number nie ma wartości -1 i zmieniona count jest mniejsza od rozmiaru tablicy. W pętli w linii 26. wartość zmiennej number jest zapisywana w tablicy w elemencie values[count], a w linii 29. wartość zmiennej count jest inkrementowana. Wartość zmiennej count jest więc inkrementowana za każdym razem, gdy do tablicy zapisywana jest kolejna wartość — dzięki temu zmieniona ta będzie wskazywała liczbę elementów, jakie zostały zapisane w tablicy. W kolejnym kroku w linii 32. proszę użytkownika o wprowadzenie kolejnej liczby (lub wprowadzenie -1, jeżeli użytkownik chce zakończyć wprowadzanie liczb), a w linii 33. odczytuje wartość i zapisuję ją w zmiennej number. Następnie pętla zaczyna się od nowa.

Jeżeli użytkownik wprowadzi liczbę -1 lub gdy zmieniona count będzie równa rozmiarowi tablicy, program wyjdzie z pętli While. W pętli For, która rozpoczyna się w linii 38.,

wyświetlam zawartość wszystkich elementów zapisanych w tablicy. Jednak w tym przypadku zamiast prześledzić wszystkie elementy pętla odczytuje tylko te spośród nich, w których zapisana jest wartość. Zwróć uwagę, że zmienna licznikowa `index` przyjmuje wartości od 0 do `count - 1`. Dzięki temu, że ustawiłem wartość maksymalną licznika na `count - 1`, a nie na `SIZE - 1`, pętla zatrzyma się po odczytaniu ostatniej prawidłowej wartości, a nie po dojściu do końca tablicy.

## Zagadnienie opcjonalne: pętla For Each

W niektórych językach programowania można spotkać specjalną wersję pętli `For`, o nazwie `For Each`. Ułatwia ona nieco przetwarzanie elementów tablicy. Pętla `For Each` ma zazwyczaj następującą postać:

```
For Each zmienna In tablica
    instrukcja
    instrukcja
    instrukcja
    itd.
```

`End For`

*Zmienna* oznacza nazwę zmiennej, a *tablica* to nazwa tablicy. Pętla taka będzie przetwarzala po kolejny każdy element danej tablicy. Podczas każdej iteracji do zmiennej *zmienna* zostanie przypisana wartość równa kolejnemu elementowi tablicy. Przykładowo w pierwszej iteracji do zmiennej *zmienna* zostanie przypisana wartość elementu *tablica[0]*, w drugiej iteracji w zmiennej tej znajdzie się element *tablica[1]* itd. Proces ten będzie się powtarzał, aż pętla dojdzie do ostatniego elementu tablicy. Założmy, że mamy w kodzie następujące deklaracje:

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE] = 5, 10, 15, 20, 25
Declare Integer num
```

Aby wyświetlić wartość wszystkich elementów zapisanych w tablicy `numbers`, możemy w tym przypadku użyć następującej pętli `For Each`:

```
For Each num In numbers
    Display num
End For
```



**UWAGA:** W niektórych językach programowania pętla `For Each` jest niedostępna, więc w kolejnych przykładach będę korzystał ze zwykłej pętli `For`.



## Punkt kontrolny

- 8.1. Czy w jednej tablicy można zapisywać wartości różnych typów?
- 8.2. Co to jest rozmiar tablicy?
- 8.3. Czy w przypadku większości języków programowania można zmienić rozmiar tablicy w czasie działania programu?
- 8.4. Co to jest element tablicy?

- 8.5. Co to jest indeks tablicy?
- 8.6. Który indeks ma zazwyczaj pierwszy element tablicy?
- 8.7. Spójrz na poniższy pseudokod i odpowiedz na pytania od a. do d.

```
Constant Integer SIZE = 7
Declare Real numbers[SIZE]
```

- a. Jak nazywa się tablica, którą zadeklarowałem?
- b. Jaki jest rozmiar tablicy?
- c. Jakiego typu dane mogą zostać zapisane w tablicy?
- d. Jaki jest indeks ostatniego elementu tablicy?
- 8.8. Na czym polega sprawdzanie indeksu tablicy?
- 8.9. Na czym polega błąd typu off-by-one?

## 8.2

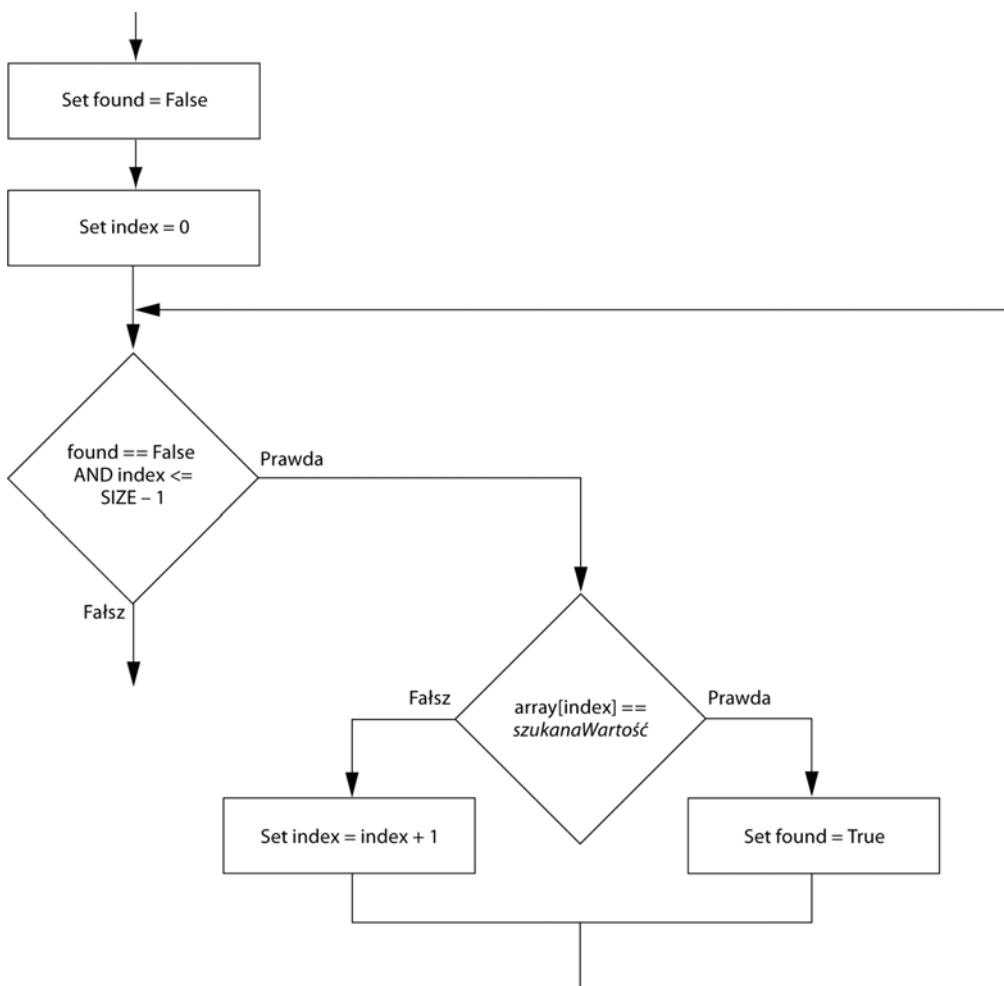
## Sekwencyjne przeszukiwanie tablicy

**WYJAŚNIENIE:** Sekwencyjne przeszukiwanie tablicy to prosta technika wyszukiwania elementu w danej tablicy. W technice tej pobieramy po kolei wartości kolejnych elementów tablicy i porównujemy je z wartością szukaną. Przeszukiwanie kończy się w momencie, gdy szukany element zostanie odnaleziony lub gdy program dojdzie do końca tablicy.

Zadaniem programu jest często wyszukanie określonej wartości wśród danych zawartych w tablicy. Opracowano w tym celu techniki zwane algorytmami wyszukiwania, które służą do odnajdywania określonej wartości w większym zbiorze danych — takim jak tablica. W tym podrozdziale wytłumaczę, jak sposób korzystać z najprostszego z tych algorytmów — przeszukiwania sekwencyjnego. W algorytmie przeszukiwania sekwencyjnego używamy pętli, która zaczynając od pierwszego elementu tablicy, porównuje wartości poszczególnych elementów z wartością szukaną. Przeszukiwanie trwa do momentu, gdy element zostanie odnaleziony lub gdy algorytm dotrze do końca tablicy. Jeżeli szukanego elementu nie ma w tablicy, algorytm prześledzi całą tablicę.

Na rysunku 8.5 przedstawiłem zasadę działania algorytmu przeszukiwania sekwencyjnego. Oto objaśnienie poszczególnych elementów schematu:

- array oznacza przeszukiwaną tablicę.
- szukanaWartość to wartość, której będzie szukał algorytm.
- found to zmienna typu Boolean, która pełni funkcję flagi. Jeżeli wartość zmiennej found jest równa False, oznacza to, że szukanaWartość nie została odnaleziona w tablicy. Wartość zmiennej found równa True oznacza, że szukanaWartość została odnaleziona.
- index to zmienna typu Integer, która pełni funkcję licznika.



**Rysunek 8.5.** Zasada działania przeszukiwania sekwencyjnego

Kiedy algorytm zakończy działanie i odnajdzie szukaną wartość w tablicy, zmienna found będzie ustawiona na wartość True. W takim przypadku w zmiennej index będzie się znajdował indeks elementu, w którym zapisana jest szukana wartość. Jeżeli algorytm nie odnajdzie szukanej wartości w tablicy, zmienna found będzie ustawiona na wartość False. Oto jak można zapisać ten algorytm w pseudokodzie:

```

Set found = False
Set index = 0
While found == False AND index <= SIZE - 1
  If array[index] == szukanaWartość Then
    Set found = True
  Else
    Set index = index + 1
  End If
End While
  
```

Pseudokod na listingu 8.6 demonstruje, jak można w programie zaimplementować algorytm przeszukiwania sekwencyjnego. W programie tym wyniki ze sprawdzianów są zapisane w tablicy. Program przeszukuje sekwencyjnie tablicę, aby znaleźć w niej wynik równy 100. Jeżeli wynik ten zostanie odnaleziony, program wyświetli numer sprawdzianu, którego dotyczy wynik.

### **Listing 8.6**



```

1 // Stała równa rozmiarowi tablicy
2 Constant Integer SIZE = 10
3
4 // Deklarujemy tablicę zawierającą wyniki ze sprawdzianów
5 Declare Integer scores[SIZE] = 87, 75, 98, 100, 82,
6                               72, 88, 92, 60, 78
7
8 // Deklarujemy zmienną boolowską pełniącą funkcję flagi
9 Declare Boolean found
10
11 // Deklarujemy zmienną licznikową
12 Declare Integer index
13
14 // Flaga musi być na początku ustawiona na False
15 Set found = False
16
17 // Ustawiamy licznik na 0
18 Set index = 0
19
20 // Przeszukujemy tablicę na obecność
21 // wyniku równego 100
22 While found == False AND index <= SIZE - 1
23     If scores[index] == 100 Then
24         Set found = True
25     Else
26         Set index = index + 1
27     End If
28 End While
29
30 // Wyświetlamy wynik wyszukiwania
31 If found Then
32     Display "Wynik równy 100 osiągnąłeś na sprawdzianie numer ", index + 1
33 Else
34     Display "Nie osiągnąłeś wyniku równego 100."
35 End If

```

#### **Wynik działania programu**

Wynik równy 100 osiągnąłeś na sprawdzianie numer 4

## **Przeszukiwanie tablicy zawierającej ciągi znaków**

Na listingu 8.6 pokazałem, jak wyszukiwać w tablicy liczby typu Integer. Na listingu 8.7 zobaczysz, że algorytm można także wykorzystać w przypadku tablicy zawierającej ciągi znaków.

**Listing 8.7**

```

1 // Deklarujemy stałą równą rozmiarowi tablicy
2 Constant Integer SIZE = 6
3
4 // Deklarujemy i inicjalizujemy tablicę ciągów znaków
5 Declare String names[SIZE] = "Mariusz Fiszer", "Krzysztof Jasiński",
6                               "Tomasz Głogowski", "Marta Jabłońska",
7                               "Renata Tomaszkiewicz", "Michał Adamek"
8
9 // Deklarujemy zmienną zawierającą szukaną wartość
10 Declare String searchValue
11
12 // Deklarujemy zmienną boolowską pełniącą funkcję flagi
13 Declare Boolean found
14
15 // Deklarujemy zmienną licznikową
16 Declare Integer index
17
18 // Flaga musi być na początku ustawiona na False
19 Set found = False
20
21 // Ustawiamy licznik na 0
22 Set index = 0
23
24 // Pobieramy szukany串 znaków
25 Display "Wprowadź osobę, którą chcesz wyszukać w tablicy."
26 Input searchValue
27
28 // Wyszukujemy w tablicy
29 // określona osobę
30 While found == False AND index <= SIZE - 1
31   If names[index] == searchValue Then
32     Set found = True
33   Else
34     Set index = index + 1
35   End If
36 End While
37
38 // Wyświetlamy wynik wyszukiwania
39 If found Then
40   Display "Osoba została odnaleziona przy indeksie ", index
41 Else
42   Display "Osoba nie została odnaleziona w tablicy."
43 End If

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź osobę, którą chcesz wyszukać w tablicy.

**Marta Jabłońska [Enter]**

Osoba została odnaleziona przy indeksie 3

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź osobę, którą chcesz wyszukać w tablicy.

**Marta [Enter]**

Osoba nie została odnaleziona w tablicy.

Program odnajdzie dany ciąg znaków w tablicy tylko wtedy, gdy użytkownik wprowadzi dokładne imię i nazwisko danej osoby. Przykładowo w pierwszym przypadku użytkownik wprowadził tekst „Marta Jabłońska” i program znalazł tę osobę w tablicy przy indeksie 3. Ale już w drugim przypadku — gdy użytkownik wprowadził tylko imię „Marta” — program wyświetlił informację, że dana osoba nie znajduje się w tablicy. Dzieje się tak dlatego, że ciąg "Marta" jest różny od ciągu "Marta Jabłońska".

Często program należy zaprojektować w taki sposób, aby można było wyszukiwać także fragment ciągu znaków. W wielu językach programowania można znaleźć funkcję biblioteczną, która sprawdza, czy dany ciąg znaków częściowo pokrywa się z innym ciągiem. W pseudokodzie używamy do tego celu funkcji `contains`. Przypomnij sobie informacje z rozdziału 6.: funkcja `contains` przyjmuje dwa ciągi znaków i zwraca wartość `True`, jeżeli w pierwszym ciągu znajduje się drugi ciąg; w przeciwnym razie funkcja zwraca wartość `False`. Na listingu 8.8 pokazałem, jak zmodyfikować program z listingu 8.7, wykorzystując funkcję `contains`.

### **Listing 8.8**

```

1 // Deklarujemy stałą równą rozmiarowi tablicy
2 Constant Integer SIZE = 6
3
4 // Deklarujemy i inicjalizujemy tablicę ciągów znaków
5 Declare String names[SIZE] = "Mariusz Fiszer", "Krzysztof Jasiński",
6                                     "Tomasz Głogowski", "Marta Jabłońska",
7                                     "Renata Tomaszecka", "Michał Adamek"
8
9 // Deklarujemy zmienną zawierającą szukaną wartość
10 Declare String searchValue
11
12 // Deklarujemy zmienną boolowską pełniącą funkcję flagi
13 Declare Boolean found
14
15 // Deklarujemy zmienną licznikową
16 Declare Integer index
17
18 // Flaga musi być na początku ustawiona na False
19 Set found = False
20
21 // Ustawiamy licznik na 0
22 Set index = 0
23
24 // Pobieramy szukany串 znaków
25 Display "Wprowadź osobę, którą chcesz wyszukać w tablicy."
26 Input searchValue
27
28 // Wyszukujemy w tablicy
29 // określona osobę
30 While found == False AND index <= SIZE - 1
31     If contains(names[index], searchValue) Then
32         Set found = True
33     Else
34         Set index = index + 1
35     End If
36 End While
37
38 // Wyświetlamy wynik wyszukiwania

```

```

39 If found Then
40   Display "Osoba pasuje do następującego elementu:"
41   Display names[index]
42 Else
43   Display "Osoba nie została odnaleziona w tablicy."
44 End If

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź osobę, którą chcesz wyszukać w tablicy.

**Marta [Enter]**

Osoba pasuje do następującego elementu:

Marta Jabłońska

**Punkt kontrolny**

- 8.10. Co to są algorytmy wyszukiwania?
- 8.11. Od którego elementu rozpoczyna działanie algorytm przeszukiwania sekwencyjnego?
- 8.12. Do czego służy pętla w algorytmie przeszukiwania sekwencyjnego?  
Co się dzieje, gdy szukana wartość nie zostanie odnaleziona w tablicy?
- 8.13. Ile elementów sprawdzi algorytm przeszukiwania sekwencyjnego w przypadku, gdy nie uda się odnaleźć szukanej wartości?
- 8.14. W jaki sposób podczas przeszukiwania tablicy można wyszukać fragment ciągu znaków?

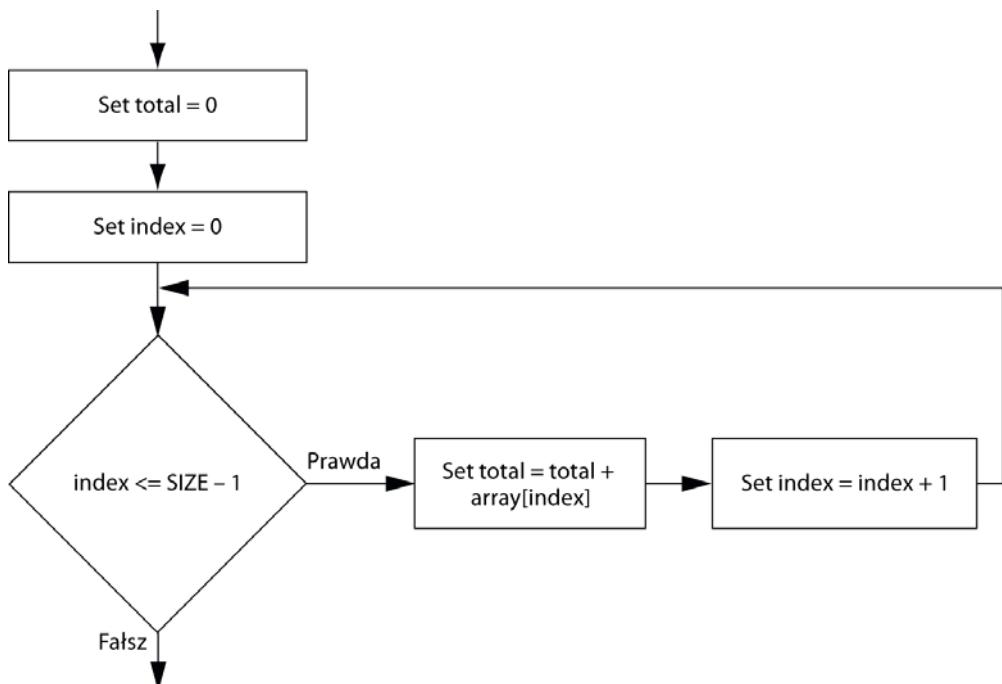
**8.3****Przetwarzanie elementów tablicy**

W tym rozdziale przedstawiłem kilka przykładów, w których do przetwarzania elementów tablicy wykorzystałem pętle. Używając pętli, można na elementach tablicy wykonywać różnego typu operacje — w tym podrozdziale przedstawię kilka spośród tych algorytmów.

**Sumowanie wartości elementów tablicy**

Aby obliczyć sumę wartości wszystkich elementów tablicy, wykorzystamy w pętli zmienną pełniąącą rolę akumulatora. Pętla pobiera po kolej elementy tablicy i dodaje je do akumulatora. Na rysunku 8.6 widoczna jest zasada działania tego algorytmu. W algorytmie tym zmienna `total` to akumulator, zmienna `index` to zmienna licznikowa, a `array` to tablica zawierająca dane liczbowe.

Na listingu 8.9 zademonstrowałem przykład wykorzystania algorytmu. Występuje w nim tablica `numbers` zawierająca liczby typu `Integer`.

**Rysunek 8.6.** Algorytm sumowania wartości elementów tablicy**Listing 8.9**

```

1 // Deklarujemy stałą równą rozmiarowi tablicy
2 Constant Integer SIZE = 5
3
4 // Deklarujemy i inicjalizujemy tablicę
5 Declare Integer numbers[SIZE] = 2, 4, 6, 8, 10
6
7 // Deklarujemy i inicjalizujemy zmienną akumulatora
8 Declare Integer total = 0
9
10 // Deklarujemy zmienną licznikową
11 Declare Integer index
12
13 // Obliczamy sumę wartości wszystkich elementów tablicy
14 For index = 0 To SIZE - 1
15     Set total = total + numbers[index]
16 End For
17
18 // Wyświetlamy sumę wartości wszystkich elementów tablicy
19 Display "Suma wartości elementów tablicy wynosi ", total
  
```

**Wynik działania programu**

Suma wartości elementów tablicy wynosi 30

## Uśrednianie wartości elementów tablicy

Pierwszym etapem obliczania średniej wartości elementów tablicy jest obliczenie sumy wszystkich wartości. Na poprzednim przykładzie pokazałem, jak można to zrobić. Drugim etapem jest podzielenie obliczonej sumy przez liczbę elementów tablicy. Przedstawiłem ten algorytm za pomocą pseudokodu na listingu 8.10.

### Listing 8.10

```

1 // Deklarujemy stałą równą rozmiarowi tablicy
2 Constant Integer SIZE = 5
3
4 // Deklarujemy i inicjalizujemy tablicę
5 Declare Real scores[SIZE] = 2.5, 8.3, 6.5, 4.0, 5.2
6
7 // Deklarujemy i inicjalizujemy zmienną akumulatora
8 Declare Real total = 0
9
10 // Deklarujemy zmienną, w której zapiszemy średnią wartość elementów
11 Declare Real average
12
13 // Deklarujemy zmienną licznikową
14 Declare Integer index
15
16 // Obliczamy sumę wartości wszystkich elementów tablicy
17 For index = 0 To SIZE - 1
18     Set total = total + numbers[index]
19 End For
20
21 // Obliczamy średnią wartość elementów tablicy
22 Set average = total / SIZE
23
24 // Wyświetlamy średnią wartość elementów tablicy
25 Display "Średnia wartość elementów tablicy jest równa ", average

```

### Wynik działania programu

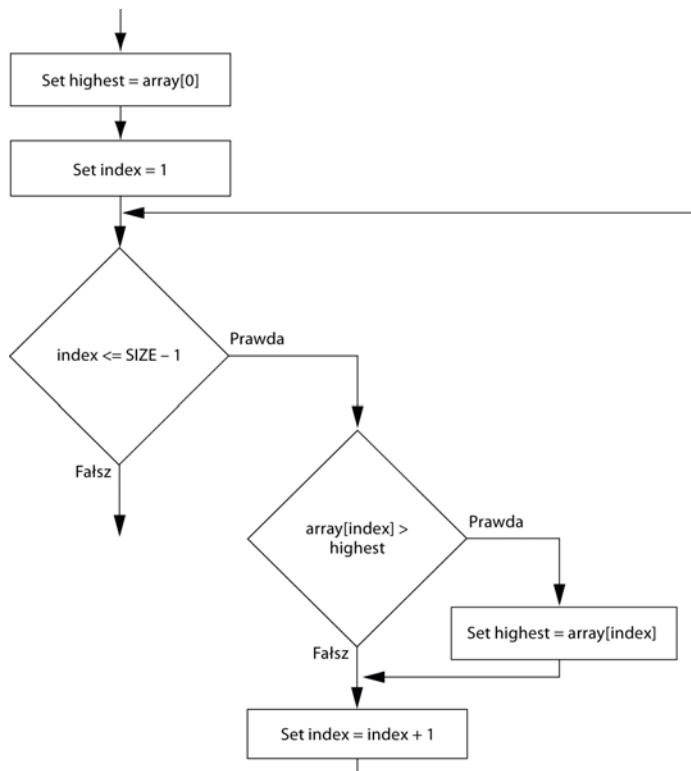
Średnia wartość elementów tablicy jest równa 5.3

## Wyszukiwanie elementu o największej wartości

Niektóre zadania programistyczne wymagają znalezienia elementu o największej wartości w zbiorze danych. Przykładem niech będą programy zwracające największą wartość sprzedają w określonym przedziale czasu, najwyższy wynik z kilku sprawdzianów, najwyższą temperaturę w zadany okresie itp.

Algorytm wyszukiwania elementu o największej wartości można opisać w następujący sposób: tworzymy zmienną, w której zapiszemy wartość największego elementu (w poniższych przykładach nazwałem tę zmienną `highest`). Następnie do zmiennej tej przypisujemy wartość elementu tablicy o indeksie 0. W kolejnym kroku korzystamy z pętli, aby prześledzić pozostałe elementy tablicy, począwszy od elementu o indeksie 1. W każdej iteracji pętli porównujemy wartość elementu tablicy ze zmienną `highest`. Jeżeli element tablicy jest większy niż zmienna `highest`, wtedy przypisujemy do tej zmiennej wartość tego elementu. Kiedy pętla zakończy działanie, w zmiennej

highest będzie się znajdowała wartość największego elementu tablicy. Schemat blokowy na rysunku 8.7 ilustruje zasadę działania tego algorytmu, a na listingu 8.11 zamieściłem przykład jego wykorzystania.



**Rysunek 8.7.** Schemat blokowy algorytmu wyszukiwania w tablicy elementu o największej wartości

### Listing 8.11

```

1 // Deklarujemy stałą równą rozmiarowi tablicy
2 Constant Integer SIZE = 5
3
4 // Deklarujemy i inicjalizujemy tablicę
5 Declare Integer numbers[SIZE] = 8, 1, 12, 6, 2
6
7 // Deklarujemy zmienną licznikową
8 Declare Integer index
9
10 // Deklarujemy zmienną, w której zapiszemy wartość największego elementu
11 Declare Integer highest
12
13 // Do zmiennej highest przypisujemy wartość pierwszego elementu
14 Set highest = numbers[0]
15
16 // Śledzimy pozostałe elementy tablicy,
17 // poczynając od indeksu 1; jeśli wartość elementu
18 // jest większa niż zmienna highest, przypisujemy
19 // ją do zmiennej highest
  
```

```

20 For index = 1 To SIZE - 1
21   If numbers[index] > highest Then
22     Set highest = numbers[index]
23   End If
24 End For
25
26 // Wyświetlamy największą wartość
27 Display "Największa wartość w tablicy jest równa ", highest

```

**Wynik działania programu**

Największa wartość w tablicy jest równa 12

## Wyszukiwanie elementu o najmniejszej wartości

W innych programach natomiast ważniejsze będzie wyszukanie w danym zbiorze danych elementu o najmniejszej wartości. Założmy, że projektujesz program, który będzie zapisywał w tablicy wyniki gry w golfa, i zechcesz wyszukać najlepszy wynik. W golfie wygrywa gracz z najmniejszym wynikiem, więc będziesz potrzebować algorytmu, który wyszuka w tablicy najmniejszą wartość.

Algorytm wyszukiwania najmniejszej wartości w tablicy jest bardzo podobny do algorytmu wyszukującego największą wartość. Działa on w następujący sposób: tworzymy zmienną, w której zapiszemy wartość najmniejszego elementu (w poniższych przykładach nazwałem tę zmienną `lowest`). Następnie do zmiennej tej przypisujemy wartość elementu tablicy o indeksie 0. W kolejnym kroku korzystamy z pętli, aby prześledzić pozostałe elementy tablicy, począwszy od elementu o indeksie 1. W każdej iteracji pętli porównujemy wartość elementu tablicy ze zmienną `lowest`. Jeżeli element tablicy jest mniejszy niż zmienna `lowest`, wtedy przypisujemy do tej zmiennej wartość tego elementu. Kiedy pętla zakończy działanie, w zmiennej `lowest` będzie się znajdowała wartość najmniejszego elementu tablicy. Na schemacie blokowym na rysunku 8.8 przedstawiłem zasadę działania tego algorytmu, a na listingu 8.12 zamieściłem prosty przykład jego wykorzystania.

**Listing 8.12**

```

1 // Deklarujemy stałą równą rozmiarowi tablicy
2 Constant Integer SIZE = 5
3
4 // Deklarujemy i inicjalizujemy tablice
5 Declare Integer numbers[SIZE] = 8, 1, 12, 6, 2
6
7 // Deklarujemy zmienną licznikową
8 Declare Integer index
9
10 // Deklarujemy zmienną, w której zapiszemy wartość najmniejszego elementu
11 Declare Integer lowest
12
13 // Do zmiennej lowest przypisujemy wartość pierwszego elementu
14 Set lowest = numbers[0]
15
16 // Śledzimy pozostałe elementy tablicy,
17 // począwszy od indeksu 1; jeśli wartość elementu
18 // jest mniejsza niż zmienna lowest, przypisujemy
19 // ją do zmiennej lowest

```

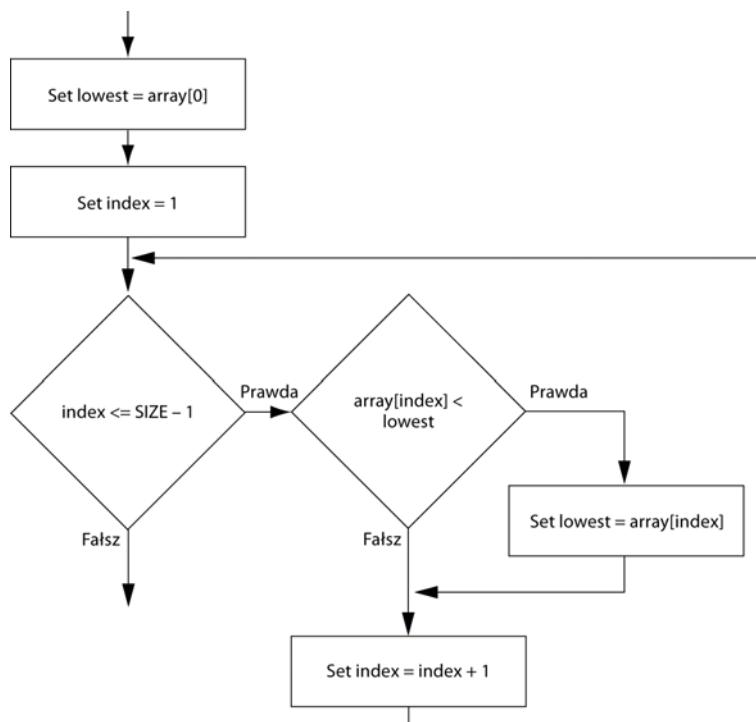
```

20 For index = 1 To SIZE - 1
21     If numbers[index] < lowest Then
22         Set lowest = numbers[index]
23     End If
24 End For
25
26 // Wyświetlamy najmniejszą wartość
27 Display "Najmniejsza wartość w tablicy jest równa ", lowest

```

**Wynik działania programu**

Najmniejsza wartość w tablicy jest równa 1



**Rysunek 8.8.** Schemat blokowy algorytmu wyszukiwania w tablicy elementu o najmniejszej wartości

## Kopiowanie tablicy

Aby skopiować zawartość tablicy do innej tablicy, w większości języków programowania trzeba operację kopiowania przeprowadzić na poszczególnych elementach tablicy. Najłatwiej jest to zrobić przy użyciu pętli. Spójrz na przykładowy pseudokod:

```

Constant Integer SIZE = 5
Declare Integer firstArray[SIZE] = 100, 200, 300, 400, 500
Declare Integer secondArray[SIZE]

```

Załóżmy, że chcemy skopiować zawartość tablicy `firstArray` do tablicy `secondArray`. Poniższy pseudokod kopiuje po kolei każdy element tablicy `firstArray` do odpowiadającego mu elementu w tablicy `secondArray`:

```

Declare Integer index
For index = 0 To SIZE - 1
    Set secondArray[index] = firstArray[index]
End For

```

## Przekazywanie tablicy jako argumentu modułu lub funkcji

W większości języków programowania jako argument modułu lub funkcji można przekazywać także tablice. Dzięki temu można podzielić na moduły wiele operacji, które wykonuje się na tablicach. Aby przekazać tablicę jako argument, należy zazwyczaj przekazać dwa argumenty: (1) tablicę i (2) liczbę całkowitą określającą, ile jest elementów w tablicy. Na listingu 8.13 pokazałem przykład funkcji, która przyjmuje jako argument tablicę liczb typu Integer. Funkcja zwraca sumę wszystkich elementów tablicy.

**Listing 8.13**



```

1 Module main()
2     // Stała równa rozmiarowi tablicy
3     Constant Integer SIZE = 5
4
5     // Tablica zainicjalizowana wartościami
6     Declare Integer numbers[SIZE] = 2, 4, 6, 8, 10
7
8     // Zmienna, w której zapiszemy sumę wartości elementów
9     Declare Integer sum
10
11    // Obliczamy sumę wartości elementów
12    Set sum = getTotal(numbers, SIZE)
13
14    // Wyświetlamy sumę wartości elementów
15    Display "Suma wartości elementów tablicy wynosi ", sum
16 End Module
17
18 // Funkcja getTotal przyjmuje jako argumenty tablicę liczb typu Integer
19 // i rozmiar tej tablicy; zwraca sumę
20 // wartości elementów tablicy
21 Function Integer getTotal(Integer array[], Integer arraySize)
22     // Zmienna licznikowa pętli
23     Declare Integer index
24
25     // Akumulator zainicjalizowany wartością 0
26     Declare Integer total = 0
27
28     // Obliczamy sumę wartości elementów tablicy
29     For index = 0 To arraySize - 1
30         Set total = total + array[index]
31     End For
32
33     // Zwracamy sumę
34     Return total
35 End Function

```

### Wynik działania programu

Suma wartości elementów tablicy wynosi 30

W module `main`, w linii 6., deklaruję tablicę liczb typu `Integer` i inicjalizuję ją pięcioma wartościami. W linii 12. wywołuję funkcję `getTotal` i przypisuję zwracającą przez nią wartość do zmiennej `sum` za pomocą następującego polecenia:

```
Set sum = getTotal(numbers, SIZE)
```

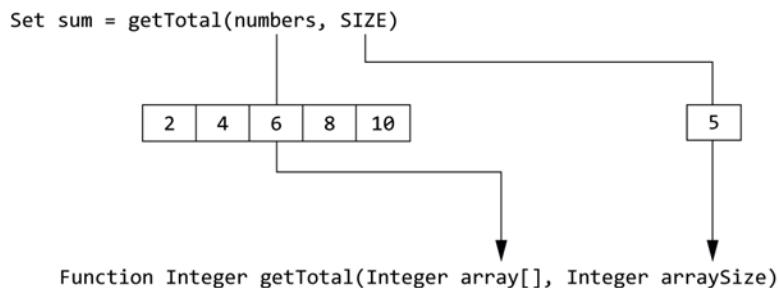
W poleceniu tym przekazuję do funkcji `getTotal` dwa argumenty: tablicę `numbers` i wartość stałej `SIZE`. Oto pierwsza linia funkcji `getTotal`, umieszczona w linii 21.:

```
Function Integer getTotal(Integer array[], Integer arraySize)
```

Zwróć uwagę, że funkcja ma dwa parametry:

- `Integer array[]` — zostanie do niego przekazana tablica liczb `Integer`;
- `Integer arraySize` — zostanie do niego przekazana liczba `Integer`, określająca liczbę elementów w tablicy.

Podczas wywołania funkcji w linii 12. do parametru `array` zostanie przekazana tablica `numbers`, a do parametru `arraySize` — stała `SIZE`. Przedstawiłem to na rysunku 8.9. Funkcja oblicza sumę wartości elementów tablicy `array`, a następnie zwraca tę sumę.



**Rysunek 8.9.** Przekazywanie argumentów do funkcji `getTotal`

## W centrum uwagi

### Przekazywanie tablicy

Doktor Mazurek przeprowadza na zajęciach z chemii cztery sprawdziany w ciągu semestru. Na koniec każdego semestru oblicza średni wynik każdego ucznia, odrzucając najniższy z nich. Doktor Mazurek poprosiła Cię o zaprojektowanie programu, który będzie przyjmował jako dane wejściowe wyniki z czterech sprawdzianów i obliczał średni wynik, odrzucając najgorszy. Opracowałaś następujący algorytm:

1. Odczytać wyniki z czterech sprawdzianów.
2. Obliczyć sumę wyników.
3. Wyszukać najniższy wynik.
4. Odjąć od sumy najniższy wynik.
5. Podzielić nową sumę przez 3, co daje średni wynik.
6. Wyświetlić średni wynik.

Na listingu 8.14 przedstawiłem program, który podzieliłem na moduły. Analizując go, rozpocznę od modułu `main`, następnie przedstawię pozostałe moduły.

**Listing 8.14. Program do obliczania średniego wyniku: moduł main**

```

1 Module main()
2   // Stała równa rozmiarowi tablicy
3   Constant Integer SIZE = 4
4
5   // Tablica, w której zapiszemy wyniki ze sprawdzianów
6   Declare Real testScores[SIZE]
7
8   // Zmienna, w której zapiszemy sumę wyników
9   Declare Real total
10
11  // Zmienna, w której zapiszemy najniższy wynik
12  Declare Real lowestScore
13
14  // Zmienna, w której zapiszemy średni wynik
15  Declare Real average
16
17  // Pobieramy od użytkownika wyniki ze sprawdzianów
18  Call getTestScores(testScores, SIZE)
19
20  // Obliczamy sumę wyników
21  Set total = getTotal(testScores, SIZE)
22
23  // Wyszukujemy najniższy wynik
24  Set lowestScore = getLowest(testScores, SIZE)
25
26  // Odejmujemy od zmiennej total najniższy wynik
27  Set total = total - lowestScore
28
29  // Obliczamy średnią: dzielimy przez 3,
30  // ponieważ odrzuciliśmy najniższy wynik
31  Set average = total / (SIZE - 1)
32
33  // Wyświetlamy średni wynik
34  Display "Średni wynik po odrzuceniu"
35  Display "najniższej wartości jest równy ", average
36 End Module
37

```

W liniach od 3. do 15. deklaruję następujące elementy:

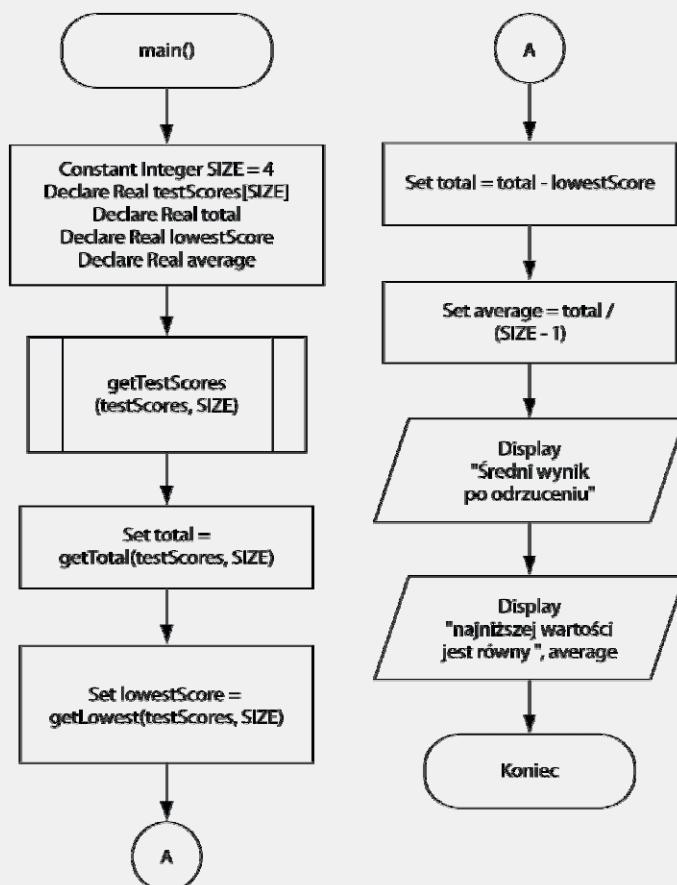
- SIZE — stała, za pomocą której określmy rozmiar tablicy;
- testScores — tablica liczb typu Real, w której zapiszemy wyniki ze sprawdzianów;
- total — zmienna typu Real, w której zapiszemy sumę wyników;
- lowestScore — zmienna typu Real, w której zapiszemy najniższy wynik;
- average — zmienna typu Real, w której zapiszemy średni wynik.

W linii 18. wywołuję moduł getTestScores i przekazuję do niego jako argumenty tablicę testScores i stałą SIZE. Jak za chwilę zobaczysz, tablica przekazywana jest przez referencję. W module pobieram od użytkownika wyniki ze sprawdzianów i zapisuję je w tablicy.

W linii 21. wywołuję funkcję `getTotal` i przekazuję do niej jako argumenty tablicę `testScores` i stałą `SIZE`. Funkcja zwraca sumę wartości wszystkich elementów tablicy. Sumę tę przypisuję do zmiennej `total`.

W linii 24. wywołuję funkcję `getLowest` i przekazuję do niej jako argumenty tablicę `testScores` i stałą `SIZE`. Funkcja zwraca wartość najmniejszego elementu w tablicy. Sumę tę przypisuję do zmiennej `lowestScore`.

W linii 27. od zmiennej `total` odejmuję wartość najniższego wyniku. Następnie w linii 31. obliczam średni wynik, dzieląc `total` przez `SIZE - 1` (użyłem wyrażenia `SIZE - 1`, ponieważ usunęliśmy z sumy wynik jednego sprawdzianu). W liniach 34. i 35. wyświetlам średni wynik. Na rysunku 8.10 przedstawiony jest schemat blokowy modułu `main`.



Rysunek 8.10. Schemat blokowy modułu `main`

Przejdźmy teraz do definicji modułu `getTestScores`.

**Listing 8.14.** Program do obliczania średniego wyniku (kontynuacja): moduł getTestScores

```

38 // Moduł getTestScores przyjmuje jako argumenty tablice (przez referencję)
39 // i jej rozmiar, prosi użytkownika o wprowadzenie wyników
40 // ze sprawdzianów i zapisuje je w tablicy
41 Module getTestScores(Real Ref scores[], Integer arraySize)
42     // Zmienna licznikowa
43     Declare Integer index
44
45     // Pobieramy poszczególne wyniki
46     For index = 0 To arraySize - 1
47         Display "Wprowadź wynik ze sprawdzianu numer ", index + 1
48         Input scores[index]
49     End For
50 End Module
51

```

Moduł getTestScores ma dwa parametry:

- scores[] — tablica liczb typu Real przekazywana przez referencję;
- arraySize — liczba Integer określająca rozmiar tablicy.

Zadaniem modułu jest pobranie od użytkownika wyników ze sprawdzianów i zapisanie ich w tablicy, którą przekazaliśmy w parametrze scores[]. Na rysunku 8.11 widoczny jest schemat blokowy modułu.

Teraz kolej na funkcję getTotal.

**Listing 8.14.** Program do obliczania średniego wyniku (kontynuacja): funkcja getTotal

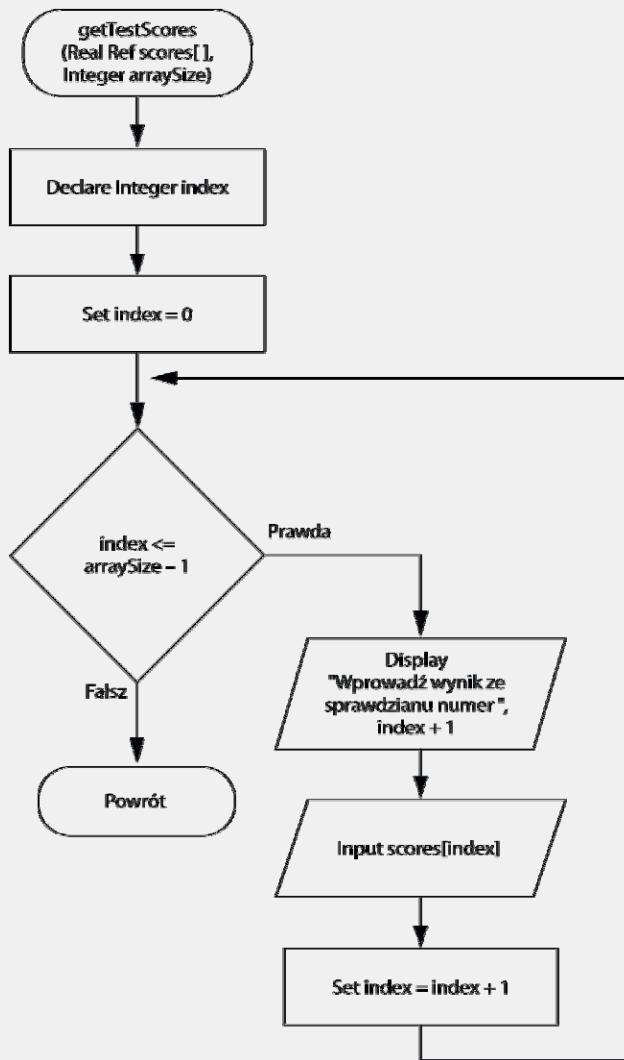
```

52 // Funkcja getTotal przyjmuje jako argumenty tablice liczb Real
53 // i jej rozmiar, a zwraca sumę wartości
54 // elementów tablicy
55 Function Real getTotal(Real array[], Integer arraySize)
56     // Zmienna licznikowa
57     Declare Integer index
58
59     // Akumulator zainicjalizowany wartością 0
60     Declare Real total = 0
61
62     // Obliczamy sumę wartości elementów tablicy
63     For index = 0 To arraySize - 1
64         Set total = total + array[index]
65     End For
66
67     // Zwracamy sumę
68     Return total
69 End Function
70

```

Funkcja getTotal ma dwa parametry:

- array[] — tablica liczb typu Real;
- arraySize — liczba Integer określająca rozmiar tablicy.



**Rysunek 8.11.** Schemat blokowy modułu `getTestScores`

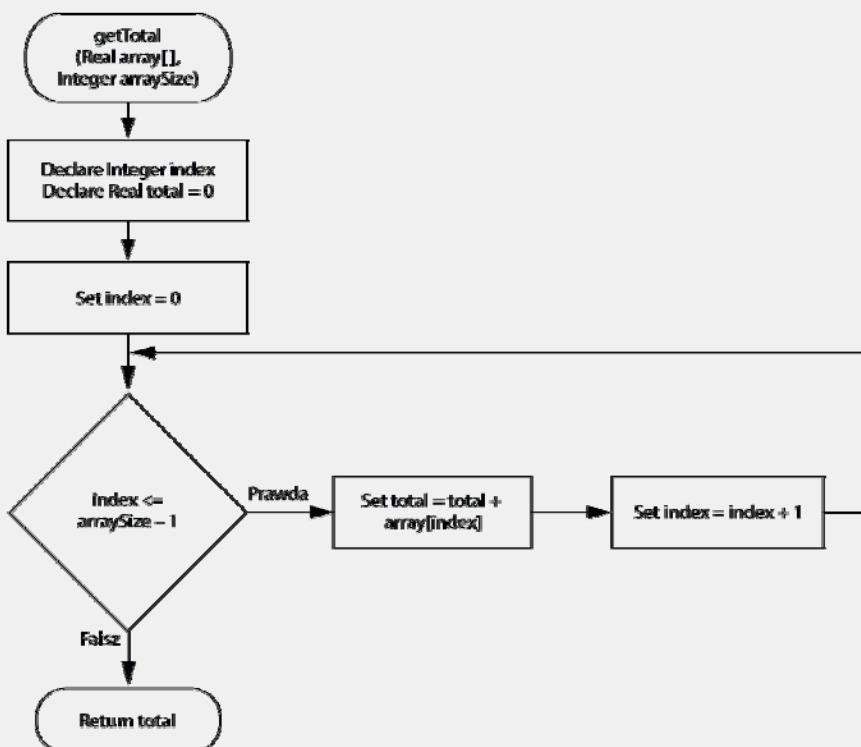
Funkcja zwraca sumę wartości elementów tablicy przekazanej w parametrze `array[]`. Na rysunku 8.12 widoczny jest schemat blokowy funkcji.

Kolejnym elementem programu jest funkcja `getLowest`.

Funkcja `getLowest` ma dwa parametry:

- `array[]` — tablica liczb typu `Real`;
- `arraySize` — liczba `Integer` określająca rozmiar tablicy.

Funkcja zwraca wartość najmniejszego elementu w tablicy przekazanej w parametrze `array[]`. Na rysunku 8.13 widoczny jest schemat blokowy funkcji.

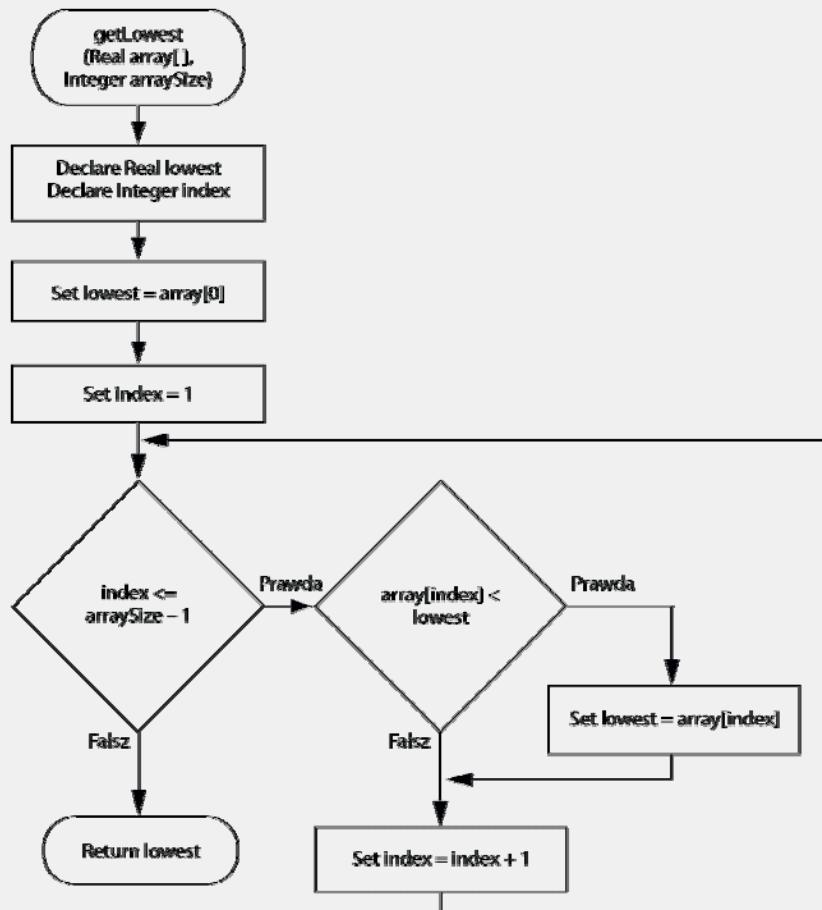


Rysunek 8.12. Schemat blokowy funkcji getTotal

#### **Listing 8.14. Program do obliczania średniego wyniku (kontynuacja): funkcja getLowest**

```

71 // Funkcja getLowest przyjmuje jako argumenty tablicę liczb typu Real
72 // i jej rozmiar, a następnie zwraca najmniejszą wartość
73 // w tablicy
74 Function Real getLowest(Real array[], Integer arraySize)
75   // Zmienna, w której zapiszemy najmniejszą wartość
76   Declare Real lowest
77
78   // Zmienna licznikowa
79   Declare Integer index
80
81   // Pobieramy pierwszy element tablicy
82   Set lowest = array[0]
83
84   // Śledzimy pozostałe elementy tablicy; jeżeli wartość elementu
85   // jest mniejsza niż zmienna lowest, przypisujemy ją do zmiennej lowest
86   For index = 1 To arraySize - 1
87     If array[index] < lowest Then
88       Set lowest = array[index]
89     End If
90   End For
91
92   // Zwracamy najmniejszą wartość
93   Return lowest
94 End Function
  
```



**Rysunek 8.13.** Schemat blokowy funkcji `getLowest`

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik ze sprawdzianu numer 1

**92 [Enter]**

Wprowadź wynik ze sprawdzianu numer 2

**67 [Enter]**

Wprowadź wynik ze sprawdzianu numer 3

**75 [Enter]**

Wprowadź wynik ze sprawdzianu numer 4

**88 [Enter]**

Średni wynik po odrzuceniu  
najniższej wartości jest równy 85



**UWAGA:** W najbardziej popularnych językach programowania (takich jak Java, C++ i C#) tablice zawsze przekazywane są przez referencję. Jest tak dlatego, że tworzenie kopii tablicy za każdym razem, gdy chcemy ją przekazać do funkcji lub modułu, jest nieefektywne. W językach tych, kiedy przekażesz tablicę do funkcji lub modułu, parametr będzie zawierał referencję do tablicy. Oznacza to, że funkcja lub moduł pracuje na oryginalnej wersji tablicy (a nie na jej kopii). Pisząc program w jednym z tych języków, zachowaj ostrożność, aby przypadkowo nie zmodyfikować elementów, które zawiera tablica przekazana przez argument.



## Punkt kontrolny

- 8.15. Opisz krótko, w jaki sposób można obliczyć sumę wartości wszystkich elementów tablicy.
- 8.16. Opisz krótko, w jaki sposób można obliczyć średnią wartość elementów tablicy.
- 8.17. Wyjaśnij, jak działa algorytm wyszukiwania największej wartości w tablicy.
- 8.18. Wyjaśnij, jak działa algorytm wyszukiwania najmniejszej wartości w tablicy.
- 8.19. W jaki sposób kopiuje się zawartość jednej tablicy do drugiej?

## 8.4

# Tablice równoległe

**WYJAŚNIENIE:** Korzystając z tego samego indeksu w dwóch różnych tablicach, można ustawić między zapisanymi w nich danymi relację.

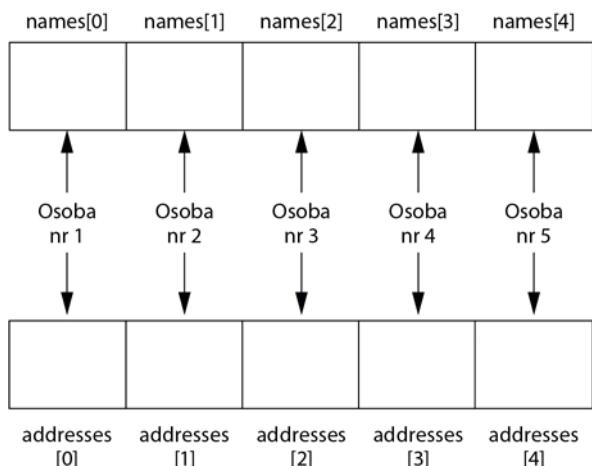
W pewnych sytuacjach pomocne okazuje się przechowywanie powiązanych ze sobą danych w dwóch lub większej liczbie tablic. Założmy, że zadeklarowałeś w programie następujące tablice:

```
Constant Integer SIZE = 5
Declare String names[SIZE]
Declare String addresses[SIZE]
```

W tablicy names są zapisane imiona, a w tablicy addresses — adresy zamieszkania pięciu osób. Dane poszczególnych osób są zapisane w odpowiadających im indeksem elementach tablic. Przykładowo imię pierwszej osoby jest zapisane w elemencie names[0], a jej adres zamieszkania — w elemencie addresses[0]. Przedstawiłem to na rysunku 8.14.

Aby odczytać te dane, należy posłużyć się tym samym indeksem w obu tablicach. Następująca pętla wyświetla imię i adres każdej z osób:

```
Declare Integer index
For index = 0 To SIZE - 1
    Display names[index]
    Display addresses[index]
End For
```

**Rysunek 8.14.** Tablice names i addresses

Tablice names i addresses są przykładami tablic równoległych. Tablice równoległe przechowują powiązane ze sobą dane — gdzie odpowiadające informacje zapisane są w obu tablicach pod tym samym indeksem.

## W centrum uwagi

### Korzystanie z tablic równoległych

W pierwszej sekcji „W centrum uwagi” zamieszczonej w tym rozdziale (program z listingu 8.4) Gosia poprosiła Cię o zaprojektowanie programu, który będzie obliczał wynagrodzenie pracownika na podstawie liczby przepracowanych przez niego godzin. W obecnej formie program odnosi się do każdego pracownika poprzez tekst „pracownik 1”, „pracownik 2”, itd. Gosia prosi Cię o zmodyfikowanie programu w taki sposób, aby mogła wprowadzać zarówno imię pracownika, jak i liczbę przepracowanych przez niego godzin oraz aby program wyświetlał imię i wynagrodzenie całkowite pracownika.

Obecnie w programie pojawia się jedna tablica o nazwie hours, w której zapisana jest liczba przepracowanych godzin każdego pracownika. Zdecydołeś się dodać do programu kolejną, równoległą tablicę o nazwie names, w której będą zapisane imiona pracowników. Dane dotyczące pierwszego pracownika znajdują się w elementach names[0] i hours[0], dane drugiego pracownika — w elementach names[1] i hours[1] itd.

Oto algorytm programu:

1. Dla każdego pracownika:
  - a) pobierz jego imię i zapisz je w tablicy names;
  - b) pobierz liczbę przepracowanych przez niego godzin i zapisz ją w tablicy hours.
2. Prześledź w pętli wszystkie elementy tablic równoległych i wyświetl imię pracownika i jego wynagrodzenie całkowite.

Na listingu 8.15 przedstawiłem poprawioną wersję programu, na rysunku 8.15 widoczny jest jego schemat blokowy.

### **Listing 8.15**

```

1 // Stała równa rozmiarowi tablicy
2 Constant Integer SIZE = 6
3
4 // Tablica zawierająca imiona pracowników
5 Declare String names[SIZE]
6
7 // Tablica zawierająca liczbę godzin przepracowanych przez pracowników
8 Declare Real hours[SIZE]
9
10 // Zmienna zawierająca stawkę godzinową
11 Declare Real payRate
12
13 // Zmienna, w której zapiszemy wynagrodzenie całkowite
14 Declare Real grossPay
15
16 // Zmienna pełniąca funkcję licznika w pętli
17 Declare Integer index
18
19 // Pobieramy dane każdego z pracowników
20 For index = 0 To SIZE - 1
21     // Pobieramy imię pracownika
22     Display "Wprowadź imię pracownika ", index + 1
23     Input names[index]
24
25     // Wprowadzamy liczbę przepracowanych godzin
26     Display "Ile godzin przepracował ten pracownik?"
27     Input hours[index]
28 End For
29
30 // Pobieramy stawkę godzinową
31 Display "Wprowadź stawkę godzinową."
32 Input payRate
33
34 // Wyświetlamy wynagrodzenie całkowite każdego pracownika
35 Display "Oto wynagrodzenia całkowite pracowników:"
36 For index = 0 To SIZE - 1
37     Set grossPay = hours[index] * payRate
38     Display names[index], ": ", currencyFormat(grossPay)
39 End For

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź imię pracownika 1

**Marcin [Enter]**

Ile godzin przepracował ten pracownik?

**10 [Enter]**

Wprowadź imię pracownika 2

**Magda [Enter]**

Ile godzin przepracował ten pracownik?

**20 [Enter]**

Wprowadź imię pracownika 3

**Kasia [Enter]**

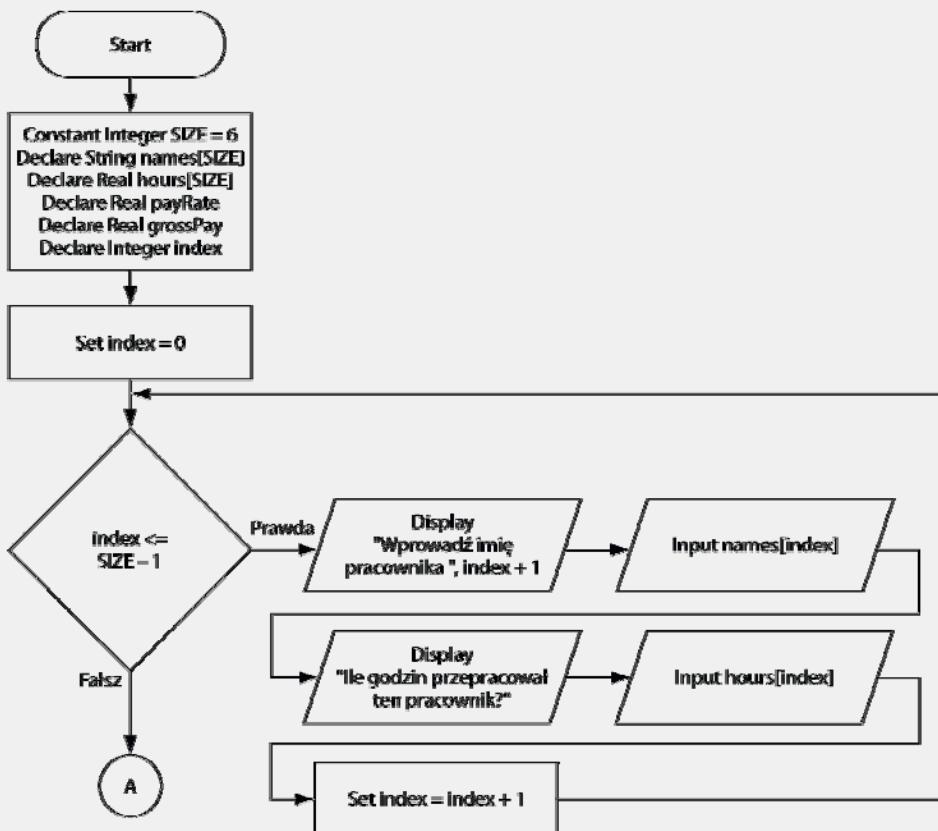
Ile godzin przepracował ten pracownik?

**15 [Enter]**

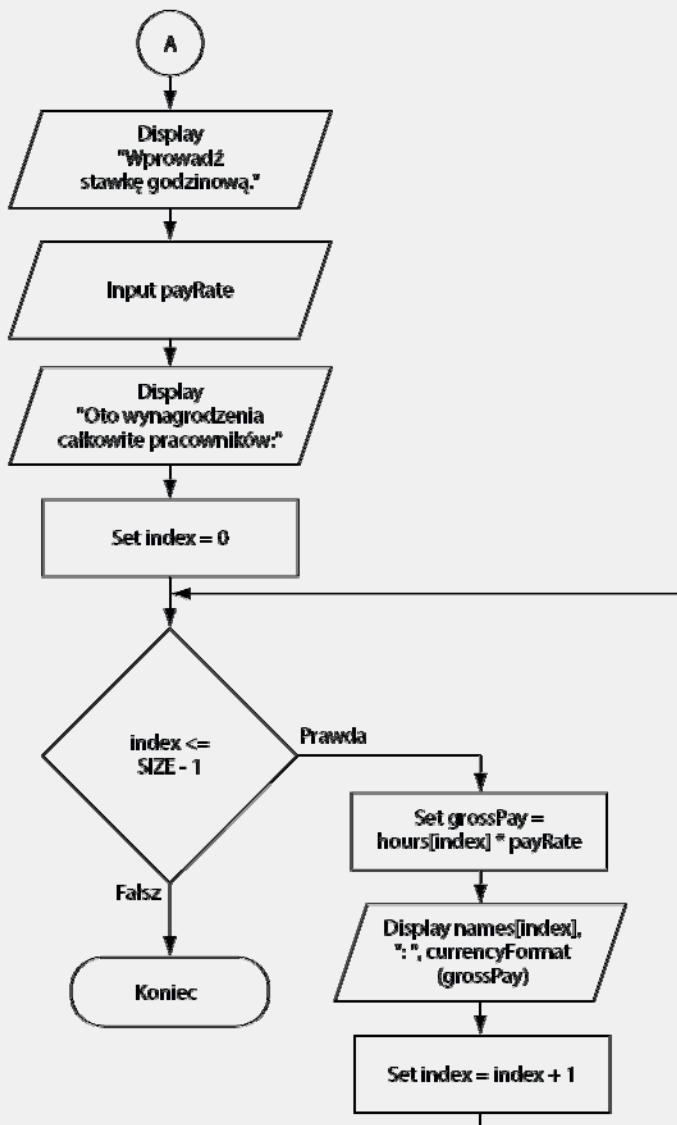
```

Wprowadź imię pracownika 4
Łukasz [Enter]
Ile godzin przepracował ten pracownik?
40 [Enter]
Wprowadź imię pracownika 5
Ania [Enter]
Ile godzin przepracował ten pracownik?
20 [Enter]
Wprowadź imię pracownika 6
Tomek [Enter]
Ile godzin przepracował ten pracownik?
18 [Enter]
Wprowadź stawkę godzinową
12,75 [Enter]
Oto wynagrodzenia całkowite pracowników:
Marcin: 127,50 zł
Magda: 255,00 zł
Kasia: 191,25 zł
Łukasz: 510,00 zł
Ania: 255,00 zł
Tomek: 229,50 zł

```



Rysunek 8.15. Schemat blokowy programu z listingu 8.15



Rysunek 8.15. (kontynuacja)



## Punkt kontrolny

- 8.20. W jaki sposób powiązane są ze sobą dane w tablicach równoległych?
- 8.21. W pewnym programie zadeklarowane są dwie tablice równolegle: names i creditScore. Tablica names przechowuje imię i nazwisko klienta, a tablica creditScore — wartość informującą o zdolności kredytowej klienta. Jeśli imię i nazwisko danego klienta zapisane jest w elemencie names[82], to w którym elemencie będą zapisane informacje dotyczące jego zdolności kredytowej?

## 8.5

## Tablice dwuwymiarowe

**WYJAŚNIENIE:** Tablica dwuwymiarowa jest podobna do kilku identycznych tabel złączonych ze sobą. Jest ona pomocna do przechowywania wielu zbiorów danych.

Tablice, które dotychczas omówiłem w tym rozdziale, są tablicami jednowymiarowymi. Nazywają się **jednowymiarowe**, ponieważ można w nich zapisać tylko jeden zbiór danych. Tablice dwuwymiarowe, zwane także tablicami 2D, mogą zawierać wiele zbiorów danych. Możesz sobie wyobrazić, że taka tablica składa się z elementów umieszczonych w wierszach i kolumnach, tak jak pokazałem to na rysunku 8.16. Widoczna na nim tablica składa się z trzech wierszy i czterech kolumn. Zwróć uwagę, że wiersze mają numery 0, 1 i 2, a kolumny mają numery 0, 1, 2 i 3. W sumie tablica składa się z 12 elementów.

|          | Kolumna 0 | Kolumna 1 | Kolumna 2 | Kolumna 3 |
|----------|-----------|-----------|-----------|-----------|
| Wiersz 0 |           |           |           |           |
| Wiersz 1 |           |           |           |           |
| Wiersz 2 |           |           |           |           |

Rysunek 8.16. Tablica dwuwymiarowa

Tablice dwuwymiarowe okazują się bardzo pomocne, gdy pracujemy na kilku zbiorach danych. Założmy przykładowo, że projektujesz dla nauczyciela program obliczający średni wynik. Nauczyciel ma sześciu uczniów, a w trakcie semestru przeprowadza pięć sprawdzianów. Jednym z rozwiązań takiego zadania jest stworzenie sześciu jednowymiarowych tablic — każda dla innego studenta. Każda tablica składałaby się z pięciu elementów — po jednym dla każdego sprawdzianu. Podejście takie byłoby jednak bardzo uciążliwe — musielibyśmy przetwarzać każdą tablicę z osobna. Znacznie lepszym rozwiązaniem jest zastosowanie jednej tablicy dwuwymiarowej zawierającej 6 wierszy (dla każdego ucznia) i 5 kolumn (dla każdego sprawdzianu). Przedstawiłem to na rysunku 8.17.

### Deklarowanie tablicy dwuwymiarowej

Podczas deklarowania tablicy dwuwymiarowej będziemy potrzebować dwóch rozmów: pierwszy wskazuje liczbę wierszy, a drugi liczbę kolumn. Poniżej zadeklarowałem przykładową tablicę dwuwymiarową:

```
Declare Integer values[3][4]
```

|                                    | Ta kolumna zawiera wyniki ze sprawdzianu nr 1 | Ta kolumna zawiera wyniki ze sprawdzianu nr 2 | Ta kolumna zawiera wyniki ze sprawdzianu nr 3 | Ta kolumna zawiera wyniki ze sprawdzianu nr 4 | Ta kolumna zawiera wyniki ze sprawdzianu nr 5 |
|------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| Kolumna 0                          |                                               |                                               |                                               |                                               |                                               |
| Ten wiersz reprezentuje uczeń nr 1 | → Wiersz 0                                    |                                               |                                               |                                               |                                               |
| Ten wiersz reprezentuje uczeń nr 2 | → Wiersz 1                                    |                                               |                                               |                                               |                                               |
| Ten wiersz reprezentuje uczeń nr 3 | → Wiersz 2                                    |                                               |                                               |                                               |                                               |
| Ten wiersz reprezentuje uczeń nr 4 | → Wiersz 3                                    |                                               |                                               |                                               |                                               |
| Ten wiersz reprezentuje uczeń nr 5 | → Wiersz 4                                    |                                               |                                               |                                               |                                               |
| Ten wiersz reprezentuje uczeń nr 6 | → Wiersz 5                                    |                                               |                                               |                                               |                                               |

Rysunek 8.17. Tablica dwuwymiarowa składająca się z 6 wierszy i 5 kolumn

W poleceniu tym deklaruję tablicę liczb typu Integer zawierającą 3 wiersze i 4 kolumny. Tablica nazywa się `values` i składa się w sumie z 12 elementów. Podobnie jak w przypadku tablic jednowymiarowych, deklarując rozmiary tablicy, warto posługiwać się stałymi. Oto przykład:

```
Constant Integer ROWS = 3
Constant Integer COLS = 4
Declare Integer values[ROWS][COLS]
```

Kiedy będziemy przetwarzali dane zawarte w tablicy dwuwymiarowej, będziemy się posługiwać dwoma indeksami: jeden będzie wskazywał wiersz, a drugi kolumnę. W przypadku tablicy `values` do elementów w pierwszym wierszu odniesiemy się w następujący sposób:

```
values[0][0]
values[0][1]
values[0][2]
values[0][3]
```

Elementy drugiego wiersza to:

```
values[1][0]
values[1][1]
values[1][2]
values[1][3]
```

A elementy trzeciego wiersza to:

```
values[2][0]
values[2][1]
values[2][2]
values[2][3]
```

Na rysunku 8.18 przedstawiłem tę tablicę i indeksy odpowiadające każdemu elementowi.

|          | Kolumna 0    | Kolumna 1    | Kolumna 2    | Kolumna 3    |
|----------|--------------|--------------|--------------|--------------|
| Wiersz 0 | values[0][0] | values[0][1] | values[0][2] | values[0][3] |
| Wiersz 1 | values[1][0] | values[1][1] | values[1][2] | values[1][3] |
| Wiersz 2 | values[2][0] | values[2][1] | values[2][2] | values[2][3] |

Rysunek 8.18. Indeksy poszczególnych elementów tablicy values

## Odwoływanie się do elementów w tablicy dwuwymiarowej

Aby odwołać się do elementu w tablicy dwuwymiarowej, musisz wskazać oba indeksy. Przykładowo to polecenie przypisuje do elementu `values[2][1]` wartość równą 95:

Set `values[2][1] = 95`

W programach przetwarzających tablice dwuwymiarowe pojawiają się najczęściej pętle zagnieżdżone. Program z listingu 8.16 jest tego przykładem. Deklaruję w nim tablicę składającą się z 2 wierszy i 3 kolumn, a następnie proszę użytkownika o wprowadzenie liczb, które zostaną zapisane w tablicy. Na końcu wyświetlam wszystkie elementy tablicy.

**Listing 8.16**



```

1 // Tworzymy tablicę 2D
2 Constant Integer ROWS = 2
3 Constant Integer COLS = 3
4 Declare Integer values[ROWS][COLS]
5
6 // Zmienne licznikowe dla wierszy i kolumn
7 Declare Integer row, col
8
9 // Pobieramy wartości do tablicy
10 For row = 0 To ROWS - 1
11   For col = 0 To COLS - 1
12     Display "Wprowadź dowolną liczbę."
13     Input values[row][col]
14   End For
15 End For
16
17 // Wyświetlamy wartości w tablicy
18 Display "Oto wprowadzone przez Ciebie liczby:"
19 For row = 0 To ROWS - 1
20   For col = 0 To COLS - 1
21     Display values[row][col]
22   End For
23 End For

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź dowolną liczbę.  
**1 [Enter]**

Wprowadź dowolną liczbę.  
**2 [Enter]**

Wprowadź dowolną liczbę.  
**3 [Enter]**

Wprowadź dowolną liczbę.  
**4 [Enter]**

Wprowadź dowolną liczbę.  
**5 [Enter]**

Wprowadź dowolną liczbę.  
**6 [Enter]**

Oto wprowadzone przez Ciebie liczby:

1

2

3

4

5

6



**WSKAZÓWKA:** W większości języków programowania podczas deklarowania tablicy dwuwymiarowej można ją także zainicjalizować. Składnia takiej inicjalizacji — w zależności od języka — wygląda różnie. Oto przykład pseudokodu, w którym inicjalizuję tablicę dwuwymiarową:

```
Declare Integer testScores[3][4] = 88, 72, 90, 92,
                    67, 72, 91, 85,
                    79, 65, 72, 84
```

W tym przypadku element `testScores[0][0]` zostanie zainicjalizowany wartością 88, element `testScores[0][1]` — wartością 72, element `testScores[0][2]` — wartością 90 itd.

W sekcji „W centrum uwagi” przedstawiłem kolejny przykład wykorzystania tablicy dwuwymiarowej. Program dodaje do akumulatora wartości wszystkich zapisanych w niej elementów.

## W centrum uwagi

### Korzystanie z tablic dwuwymiarowych



Przedsiębiorstwo Unique Candy posiada trzy oddziały: oddział 1 (wschodnie wybrzeże), oddział 2 (centrum), oddział 3 (zachodnie wybrzeże). Kierownik sprzedaje poprosił Cię o zaprojektowanie programu, w którym będzie można wprowadzić wartość sprzedaży uzyskaną w każdym z oddziałów w czterech kwartałach, a następnie wyświetlić całkowitą wartość sprzedaży we wszystkich oddziałach.

Program będzie musiał przetwarzać trzy zbiory danych:

- wartość sprzedaży w oddziale 1;
- wartość sprzedaży w oddziale 2;
- wartość sprzedaży w oddziale 3.

Każdy zbiór będzie zawierał cztery wartości:

- sprzedaż w kwartale 1;
- sprzedaż w kwartale 2;
- sprzedaż w kwartale 3;
- sprzedaż w kwartale 4.

Postanowileś zapisać dane dotyczące sprzedaży w tablicy dwuwymiarowej. Tablica będzie się składała z 3 wierszy (dla każdego oddziału) i 4 kolumn (dla każdego kwartału). Na rysunku 8.19 pokazałem, w jaki sposób dane dotyczące sprzedaży będą zapisane w tablicy.

|                 | <b>Kolumna 0</b>                                                       | <b>Kolumna 1</b>                                                       | <b>Kolumna 2</b>                                                       | <b>Kolumna 3</b>                                                       |
|-----------------|------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|
| <b>Wiersz 0</b> | <code>sales[0][0]</code><br>Zawiera sprzedaż w oddziale 1 w kwartale 1 | <code>sales[0][1]</code><br>Zawiera sprzedaż w oddziale 1 w kwartale 2 | <code>sales[0][2]</code><br>Zawiera sprzedaż w oddziale 1 w kwartale 3 | <code>sales[0][3]</code><br>Zawiera sprzedaż w oddziale 1 w kwartale 4 |
| <b>Wiersz 1</b> | <code>sales[1][0]</code><br>Zawiera sprzedaż w oddziale 2 w kwartale 1 | <code>sales[1][1]</code><br>Zawiera sprzedaż w oddziale 2 w kwartale 2 | <code>sales[1][2]</code><br>Zawiera sprzedaż w oddziale 2 w kwartale 3 | <code>sales[1][3]</code><br>Zawiera sprzedaż w oddziale 2 w kwartale 4 |
| <b>Wiersz 2</b> | <code>sales[2][0]</code><br>Zawiera sprzedaż w oddziale 3 w kwartale 1 | <code>sales[2][1]</code><br>Zawiera sprzedaż w oddziale 3 w kwartale 2 | <code>sales[2][2]</code><br>Zawiera sprzedaż w oddziale 3 w kwartale 3 | <code>sales[2][3]</code><br>Zawiera sprzedaż w oddziale 3 w kwartale 4 |

**Rysunek 8.19.** Tablica dwuwymiarowa, w której zapisane są dane dotyczące sprzedaży

W programie wykorzystamy dwie zagnieżdżone pętle, w których pobierzemy wartość sprzedaży. Następnie ponownie użyjemy dwóch zagnieżdżonych pętli, aby zsumować w akumulatorze wszystkie dane w tablicy. Oto podsumowanie algorytmu:

1. Dla każdego oddziału:

Dla każdego kwartału:

Pobierz wartość sprzedaży w danym kwartale i zapisz w tablicy.

2. Dla każdego wiersza tablicy:

Dla każdej kolumny tablicy:

Dodaj do akumulatora wartość z danej kolumny.

3. Wyświetl wartość zapisaną w akumulatorze.

Na listingu 8.17 przedstawiłem pseudokod programu.

**Listing 8.17**

```

1 // Stałe równe rozmiarom tablicy
2 Constant Integer ROWS = 3
3 Constant Integer COLS = 4
4
5 // Tablica, w której zapiszemy wartość sprzedaży
6 Declare Real sales[ROWS][COLS]
7
8 // Zmienne licznikowe
9 Declare Integer row, col
10
11 // Akumulator
12 Declare Real total = 0
13
14 // Wyświetlamy instrukcje
15 Display "Program oblicza wartość sprzedaży w przedsiębiorstwie."
16 Display "Wprowadź wartość sprzedaży uzyskaną w każdym kwartale"
17 Display "w każdym oddziale."
18
19 // Zagnieżdzona pętla, w której zapisywana jest wartość sprzedaży
20 // każdego oddziału w każdym z czterech kwartałów
21 For row = 0 To ROWS - 1
22   For col = 0 To COLS - 1
23     Display "Oddział ", row + 1, " kwartał ", col + 1
24     Input sales[row][col]
25   End For
26   // Wyświetlamy pustą linię
27   Display
28 End For
29
30 // W pętli zagnieżdzonej sumujemy wartości wszystkich elementów tablicy
31 For row = 0 To ROWS - 1
32   For col = 0 To COLS - 1
33     Set total = total + sales[row][col]
34   End For
35 End For
36
37 // Wyświetlamy sumę sprzedaży w przedsiębiorstwie
38 Display "Całkowita wartość sprzedaży wynosi: ",
39   currencyFormat(total)

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Program oblicza wartość sprzedaży w przedsiębiorstwie.

Wprowadź wartość sprzedaży uzyskaną w każdym kwartale  
w każdym oddziale.

Oddział 1 kwartał 1

**1000.00 [Enter]**

Oddział 1 kwartał 2

**1100.00 [Enter]**

Oddział 1 kwartał 3

**1200.00 [Enter]**

Oddział 1 kwartał 4

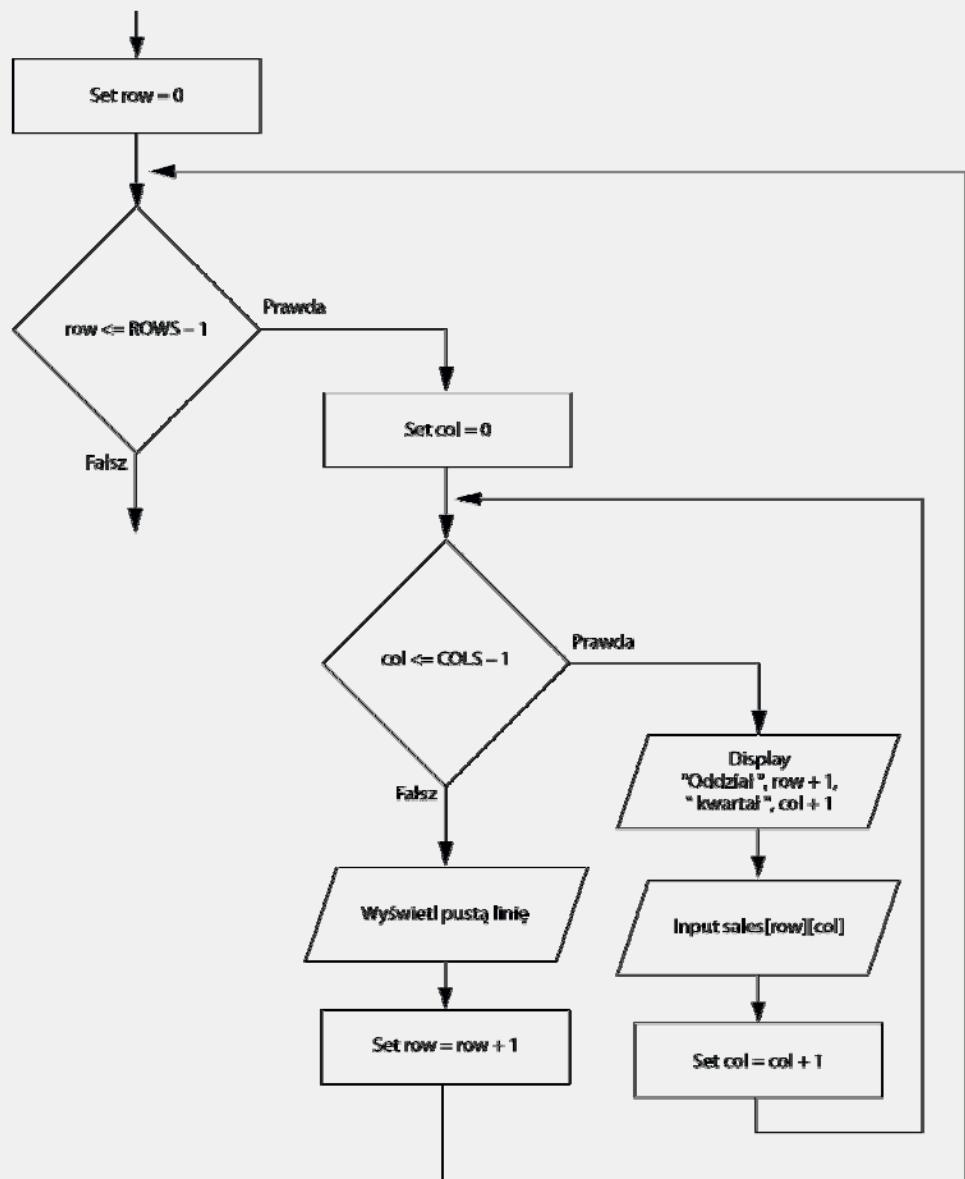
**1300.00 [Enter]**

Oddział 2 kwartał 1

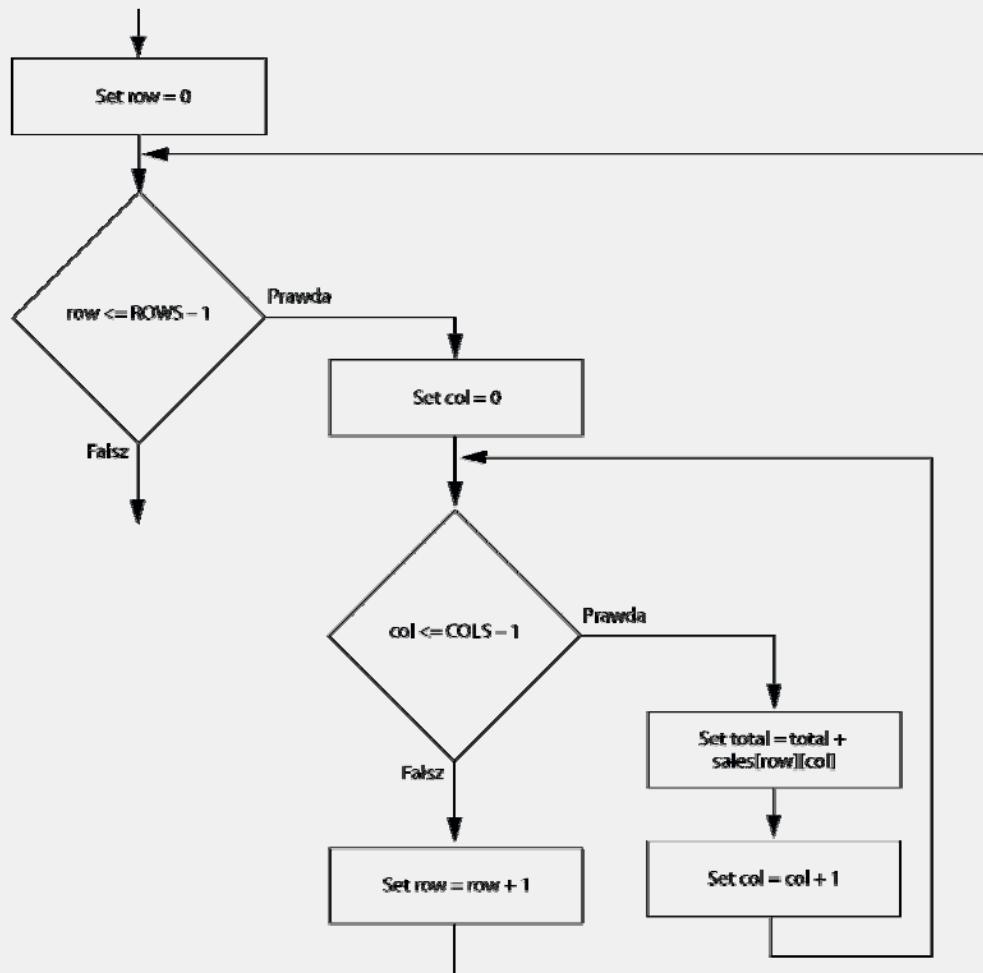
**2000.00 [Enter]**

Oddział 2 kwartał 2

**2100.00 [Enter]**



Rysunek 8.20. Schemat blokowy pierwszej pętli zagnieżdżonej (w liniach od 21. do 28.)



**Rysunek 8.21.** Schemat blokowy drugiej pętli zagnieżdżonej (w liniach od 31. do 35.)

Oddział 2 kwartał 3  
**2200.00 [Enter]**  
 Oddział 2 kwartał 4  
**2300.00 [Enter]**

Oddział 3 kwartał 1  
**3000.00 [Enter]**  
 Oddział 3 kwartał 2  
**3100.00 [Enter]**  
 Oddział 3 kwartał 3  
**3200.00 [Enter]**  
 Oddział 3 kwartał 4  
**3300.00 [Enter]**

Całkowita wartość sprzedaży wynosi: 25 800,00 zł

Pętla zagnieżdzona pojawia się po raz pierwszy w liniach od 21. do 28. W tym miejscu prosimy użytkownika o wprowadzenie danych dotyczących sprzedaży w każdym z oddziałów w każdym kwartale. Na rysunku 8.20 przedstawiłem schemat blokowy tej pętli.

Druga pętla zagnieżdzona pojawia się w liniach od 31. do 35. Śledzi ona po kolej wszystkie elementy tablicy sales i sumuje ich wartości w zmiennej total będącej akumulatorem. Na rysunku 8.21 widoczny jest schemat blokowy tej pętli. Po wyjściu z pętli w zmiennej total będzie zapisana suma wszystkich elementów tablicy sales.



## Punkt kontrolny

- 8.22. Z ilu wierszy i kolumn składa się poniższa tablica?

```
Declare Integer points[88] [100]
```

- 8.23. Napisz instrukcję w pseudokodzie, dzięki której w ostatnim elemencie tablicy zadeklarowanej w powyższym punkcie zostanie zapisana wartość 100.
- 8.24. Napisz polecenie deklarujące tablicę dwuwymiarową i inicjalizujące ją następującymi wartościami:

```
12 24 32 21 42  
14 67 87 65 90  
19 1 24 12 8
```

- 8.25. Założmy, że w programie pojawiają się następujące deklaracje:

```
Constant Integer ROWS = 100  
Constant Integer COLS = 50  
Declare Integer info[ROWS] [COLS]
```

Napisz fragment pseudokodu, w którym za pomocą zagnieżdzonej pętli przypiszesz do każdego elementu tablicy wartość równą 99.

### 8.6

## Tablice trój- i więcej wymiarowe

**WYJAŚNIENIE:** Aby zapisać w programie dane pochodzące z wielu zbiorów danych, w większości języków programowania można tworzyć tablice wielowymiarowe.

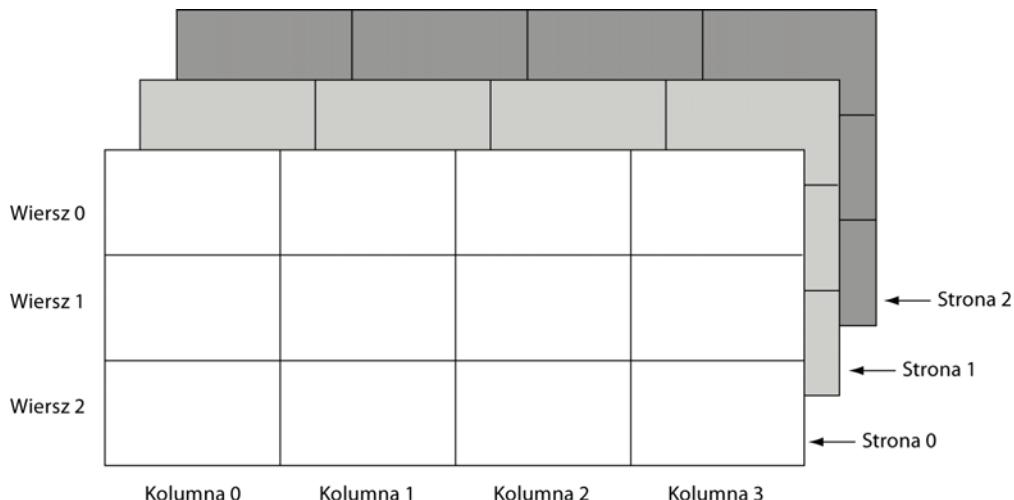
W poprzednim podrozdziale zaprezentowałem tablice dwuwymiarowe. Jednak w większości języków programowania można także tworzyć tablice mające trzy, a nawet więcej wymiarów. Oto przykład deklaracji tablicy trójwymiarowej w pseudokodzie:

```
Declare Real seats [3][5][8]
```

Możesz sobie wyobrazić taką tablicę jako trzy zestawy 5 wierszy, gdzie każdy wiersz zawiera 5 elementów. Tablicę taką można wykorzystać, aby zapisać ceny biletów na

widowni, na której w każdym rzędzie znajduje się 8 siedzeń, każda sekcja składa się z 5 rzędów, a w sumie są 3 sekcje.

Na rysunku 8.22 zilustrowałem tablicę trójwymiarową jako „strony” zawierające tablice dwuwymiarowe.



**Rysunek 8.22.** Tablica trójwymiarowa

Trudno jest sobie wyobrazić tablicę składającą się z więcej niż trzech wymiarów, jednak przydają się one w pewnych zadaniach programistycznych. Przykładem niech będzie magazyn w fabryce, gdzie urządzenia są przechowywane na paletach w pudełkach ustawionych jedno na drugim. Możemy w takim przypadku użyć tablicy czterowymiarowej i zapisać numer seryjny każdego urządzenia. Cztery indeksy takiej tablicy będą reprezentować numer palety, numer pudełka, numer rzędu i numer kolumny. Na podobnej zasadzie można wykorzystać tablicę pięciowymiarową, w której będzie można uwzględnić kilka magazynów.



## Punkt kontrolny

- 8.26. W księgarni książki przechowywane są na 50 regałach, z których każdy ma 10 półek. Każda półka mieści 25 książek. Zadeklaruj tablicę trójwymiarową zawierającą ciągi znaków, w której będzie można zapisać tytuły wszystkich książek w księgarni. Poszczególne wymiary tablicy powinny reprezentować regał, półkę i numer książki.

**8.7**

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

### **Java**

#### Tablice

##### **Deklaracje zmiennych tablicowych**

Oto przykład deklaracji tablicy w Javie:

```
int[] numbers = new int[6];
```

Powyższy kod deklaruje zmienną `numbers` jako tablicę typu `int`. Deklaracja rozmawia okręsła, że tablica ma 6 elementów. Jak wspomniałem wcześniej w tym rozdziale, dobrą praktyką jest użycie stałej nazwanej jako deklaratora rozmiaru:

```
final int SIZE = 6;
int[] numbers = new int[SIZE];
```

Oto inny przykład:

```
final int SIZE = 200;
double[] temperatures = new double[SIZE];
```

Ten fragment kodu deklaruje zmienną o nazwie `temperatures` jako tablicę zawierającą 200 elementów typu `double`. Oto jeszcze jeden przykład:

```
final int SIZE = 10;
String[] names = new String[SIZE];
```

Tutaj deklarowana jest zmienna o nazwie `names` jako tablica zawierająca 10 elementów typu `string`.

##### **Elementy tablic i ich indeksy w Javie**

Z pomocą indeksów możesz uzyskać dostęp do każdego elementu tablicy. Indeks pierwszego elementu to 0, drugiego to 1 i tak dalej. Natomiast indeks ostatniego elementu to rozmiar całej tablicy minus 1. Na przykład poniższy kod deklaruje tablicę typu `int` z trzema elementami, a następnie przypisuje wartość do każdego elementu; indeksy dla tych elementów to odpowiednio 0, 1 i 2:

```
final int SIZE = 3;
int[] numbers = new int[SIZE];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
```

## Inicjalizowanie tablicy w języku Java

W ramach deklaracji tablicy możesz też ją zainicjalizować wybranymi wartościami. Oto przykład:

```
int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Powyższa instrukcja deklaruje zmienną `days` jako tablicę typu `int` i przypisuje jej wartości początkowe. Seria wartości umieszczona w nawiasach klamrowych i oddzielona przecinkami jest nazywana listą inicjalizującą. Wartości te są przechowywane w elementach tablicy w kolejności, w jakiej pojawiają się na liście (pierwsza wartość, 31, jest przechowywana na pozycji `days[0]`, druga wartość, 28, jest przechowywana na pozycji `days[1]` i tak dalej.) Pamiętaj, że gdy korzystasz z listy inicjalizującej, nie używasz słowa kluczowego `new`. Java automatycznie utworzy tablicę i umieści wartości w liście inicjalizującej.

Kompilator Javy określa rozmiar tablicy według liczby elementów na liście inicjalizującej. Ze względu na to, że na tej liście znajduje się 12 pozycji, tablica będzie się składała z 12 elementów.

### Atrybut length

Każda tablica w Javie ma atrybut o nazwie `length`. Wartością atrybutu jest liczba elementów w tablicy. Na przykład przyjrzyj się tablicy utworzonej za pomocą następującej instrukcji:

```
double[] temperatures = new double[25];
```

Ponieważ tablica `temperatures` składa się z 25 elementów, poniższa instrukcja przypisze wartość 25 do zmiennej `size`:

```
size = temperatures.length;
```

Atrybut `length` może być przydatny podczas przetwarzania całej zawartości tablicy. Na przykład poniższa pętla przegląda zawartość tablicy i wyświetla wartość każdego elementu. Atrybut `length` jest używany w wyrażeniu sprawdzającym jako górny limit zmiennej sterującej pętlą:

```
for (int i = 0; i < temperatures.length; i++)
    System.out.println(temperatures[i]);
```

Trzeba tu zachować ostrożność, aby nie spowodować błędu „o jeden za dużo”, gdy atrybut `length` jest używany jako górny limit indeksu tablicy. Atrybut `length` zawiera liczbę elementów w tablicy, a maksymalny indeks w tablicy to wartość `length - 1`.

## Przekazywanie tablicy jako argumentu do metody w języku Java

Podczas przekazywania tablicy jako argumentu do metody w Javie nie ma konieczności definiowania osobnego argumentu wskazującego rozmiar tablicy. Wynika to z faktu, że tablice w Javie mają atrybut `length`, który zwraca ich rozmiar. Poniższy kod przedstawia metodę, która przyjmuje tablicę jako swój argument:

```
public static void showArray(int[] array)
```

```
{
    for (int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
}
```

Zauważ, że zmienna parametru `array` jest zadeklarowana jako tablica typu `int`. Podczas wywoływania tej metody musisz przekazać do niej tablicę typu `int` jako jej argument. Oto sposób wywołania metody `showArray` z argumentem `numbers` przy założeniu, że zmienna ta jest tablicą typu `int`:

```
showArray(numbers);
```

### Tablice dwuwymiarowe w języku Java

Oto przykład deklaracji dwuwymiarowej tablicy z trzema wierszami i czterema kolumnami:

```
double[][] scores = new double[3][4];
```

Dwa zestawy nawiasów kwadratowych w typie danych wskazują, że zmienna `scores` przechowuje tablicę dwuwymiarową. Liczby 3 i 4 deklarują rozmiar tej tablicy. Pierwszy deklarator rozmiaru określa liczbę wierszy, natomiast drugi — liczbę kolumn. Zauważ, że każdy deklarator rozmiaru jest zamknięty w osobnym zestawie nawiasów kwadratowych.

Podczas przetwarzania danych w dwuwymiarowej tablicy do każdego elementu tablicy można się odwoływać za pomocą dwóch indeksów: jednego dla wiersza i drugiego dla kolumny. W tablicy `scores` elementy w wierszu 0 są oznaczone w następujący sposób:

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

Elementy pierwszego wiersza to:

```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

Elementy drugiego wiersza to:

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

Aby odwołać się do elementu w tablicy dwuwymiarowej, musisz wskazać oba indeksy. Na przykład to polecenie przypisuje do elementu `scores[2][1]` wartość równą 95:

```
scores[2][1] = 95;
```

W programach przetwarzających tablice dwuwymiarowe pojawiają się najczęściej pętle zagnieżdżone. Na przykład na tym listingu kod programu prosi użytkownika o wprowadzenie wartości dla każdego elementu w tablicy:

```
final int ROWS = 3;
final int COLS = 4;
```

```

double[][] scores = new double[ROWS][COLS];
for (int row = 0; row < ROWS; row++)
{
    for (int col = 0; col < COLS; col++)
    {
        System.out.print("Podaj wartość: ");
        scores[row][col] = keyboard.nextDouble();
    }
}

```

Ten kod wyświetla wszystkie elementy tablicy scores:

```

for (int row = 0; row < ROWS; row++)
{
    for (int col = 0; col < COLS; col++)
    {
        System.out.println(scores[row][col]);
    }
}

```

### Tablice wielowymiarowe w języku Java

W Javie można tworzyć tablice o dowolnej liczbie wymiarów. Oto przykład deklaracji tablicy trójwymiarowej:

```
double[][][] seats = new double[3][5][8];
```

Tablicę tę można traktować jako 3 zestawy po 5 wierszy, z których każdy zawiera 8 elementów. Tablica może służyć do przechowywania cen poszczególnych miejsc w audytorium, gdzie znajduje się po osiem miejsc w rzędzie, sekcja ma po pięć rzędów i łącznie są trzy sekcje.

## Python

### Listy

W Pythonie zamiast tablic wykorzystuje się listy. **Lista** jest podobna do tablicy, ale zapewnia znacznie więcej możliwości niż tradycyjna tablica. Jest obiektem przechowującym wiele elementów danych. Każda pozycja na takiej liście nazywana jest **elementem**. Oto instrukcja, która tworzy listę liczb całkowitych:

```
even_numbers = [2, 4, 6, 8, 10]
```

Poszczególne pozycje umieszczone w nawiasach kwadratowych i oddzielone od siebie przecinkami stanowią wartości elementów listy. Oto inny przykład:

```
names = ['Magda', 'Stefan', 'Wojtek', 'Alicja', 'Adrian']
```

Ta instrukcja tworzy listę pięciu ciągów znaków.

Aby wyświetlić całą listę, możesz użyć funkcji print:

```
numbers = [5, 10, 15, 20]
print(numbers)
```

Po wywołaniu funkcji print w drugiej instrukcji elementy listy wyświetlane są w następujący sposób:

```
[5, 10, 15, 20]
```

## Indeksy i elementy listy w języku Python

Dostęp do każdego elementu listy możesz uzyskać za pomocą indeksu. Jak już wspomniałem, indeks pierwszego elementu to 0, drugiego to 1 i tak dalej. Natomiast indeks ostatniego elementu to rozmiar całej tablicy minus 1. Na przykład poniższy kod deklaruje listę o nazwie `numbers` z trzema elementami o wartości równej 0, a następnie przypisuje nowe wartości do każdego elementu; indeksy dla tych elementów to odpowiednio 0, 1 i 2:

```
numbers = [0, 0, 0]
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
```

## Stosowanie funkcji len z listami w języku Python

Jeśli w liście użyjesz nieprawidłowego indeksu, to pojawi się błąd. Na przykład spójrz na następujący kod:

```
# Ten kod spowoduje pojawienie się błędu
my_list = [10, 20, 30, 40]
index = 0
while index < 5:
    print(my_list[index])
    index += 1
```

Podczas ostatniej iteracji tej pętli do zmiennej `index` zostanie przypisana wartość 4, która nie będzie stanowiła poprawnego indeksu dla tej listy. W rezultacie instrukcja wywołująca funkcję `print` spowoduje błąd.

W celu uzyskania długości listy w Pythonie możesz wykorzystać funkcję `len`. Funkcja ta zwraca liczbę elementów zawartych w tej liście, która zostanie jej podana jako argument. Pokazany wcześniej błędny kod programu można zmodyfikować w następujący sposób:

```
my_list = [10, 20, 30, 40]
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

## Iterowanie listy za pomocą pętli for w języku Python

W Pythonie możesz z łatwością przeglądać zawartość listy za pomocą pętli `for`:

```
numbers = [99, 100, 101, 102]
for n in numbers:
    print(n)
```

Instrukcja `for` działa w następujący sposób: zmiennej `n` przypisywana jest kopia pierwszej wartości znajdującej się na liście, po czym wykonywane są kolejne instrukcje znajdujące się w bloku, potem zmiennej `n` przypisywana jest kopia następnej wartości z listy, a instrukcje z bloku są wykonywane ponownie. Jest to kontynuowane, dopóki zmiennej `n` nie zostanie przypisana ostatnia wartość znajdująca się na liście. Jeśli uruchomimy ten kod, wyświetli nam:

```
99
100
101
102
```

Należy jednak pamiętać, że podczas wykonywania się pętli `for` do zmiennej `n` przypisywana jest kopia elementów listy, a wszelkie dokonane w niej zmiany nie mają wpływu na samą listę. Oto przykład takiego działania:

```
1 numbers = [99, 100, 101, 102]
2 for n in my_list:
3     n = 0
4
5 print(my_list)
```

Instrukcja zawarta w wierszu 3. ponownie przypisuje zmiennej `n` inną wartość (0). Nie zmienia to elementu listy, do którego zmienna `n` odwoływała się przed wykonaniem tej instrukcji. Po wykonaniu tego kodu instrukcja w wierszu 5. wyświetli wynik:

```
99
100
101
102
```

### **Przekazywanie listy jako argumentu do funkcji w języku Python**

Podczas przekazywania listy jako argumentu do funkcji w Pythonie nie jest konieczne definiowanie oddzielnego argumentu podającego rozmiar tej listy. Wynika to z faktu, że Python udostępnia funkcję `len`, która zwraca rozmiar listy. Poniższy kod przedstawia funkcję, która przyjmuje listę jako swój argument:

```
def set_to_zero(numbers):
    index = 0
    while index < len(numbers):
        numbers[index] = 0
        index = index + 1
```

Parametr `numbers` w tej funkcji umożliwia odwoływanie się do listy. Kiedy wywołasz tę funkcję i przekażesz jej listę jako argument, pętla przypisze wartość 0 do każdego elementu listy. Oto przykład kodu, który wywołuje funkcję:

```
my_list = [1, 2, 3, 4, 5]
set_to_zero(my_list)
print(my_list)
```

Ostatni wiersz kodu spowoduje wyświetlenie:

```
0
0
0
0
0
```

### **Listy dwuwymiarowe w języku Python**

W Pythonie możesz utworzyć listę list, która działa podobnie jak tablica dwuwymiarowa. Oto przykład:

```
numbers = [ [1, 2, 3], [10, 20, 30] ]
```

Wykonanie tej instrukcji spowoduje utworzenie listy o nazwie `numbers` zawierającej dwa elementy. Pierwszym elementem jest ta lista:

```
[1, 2, 3]
```

Drugim elementem jest ta lista:

```
[10, 20, 30]
```

Poniższa instrukcja wypisuje zawartość wyrażenia `numbers[0]`, które jest pierwszym elementem:

```
print(numbers[0])
```

Jeżeli wykonamy to polecenie, na ekranie pojawi się:

```
[1, 2, 3]
```

Poniższa instrukcja wypisuje zawartość wyrażenia `numbers[1]`, które jest drugim elementem:

```
print(numbers[1])
```

Jeżeli wykonamy to polecenie, na ekranie pojawi się:

```
[10, 20, 30]
```

### **Wiersze i kolumny**

Jak wspominałem już w tym rozdziale, o tablicach dwuwymiarowych zwykle myślimy jak o zestawie wierszy i kolumn. Tej samej metafory możemy również użyć wobec list dwuwymiarowych. Założmy, że poniższa dwuwymiarowa lista zawiera zestawy wyników testów:

```
scores = [ [70, 80, 90],  
          [80, 60, 75],  
          [85, 75, 95] ]
```

Deklarując listę w taki sposób (z każdą listą będącą elementem wyświetlonym w oddzielnym wierszu), łatwo jest zobrazować nasze myślenie o liście jako o zestawie wierszy i kolumn.

Podczas przetwarzania danych w dwuwymiarowej liście każdy element określany jest przez dwa indeksy: jeden dla wiersza i drugi dla kolumny. Na liście `scores` elementy w wierszu 0 są oznaczone w następujący sposób:

```
scores[0][0]  
scores[0][1]  
scores[0][2]
```

Elementy pierwszego wiersza to:

```
scores[1][0]  
scores[1][1]  
scores[1][2]
```

Elementy drugiego wiersza to:

```
scores[2][0]  
scores[2][1]  
scores[2][2]
```

Aby uzyskać dostęp do jednego z elementów na liście dwuwymiarowej, trzeba użyć obu indeksów. Na przykład poniższa instrukcja wypisuje liczbę z listy scores[0][2]:

```
print(scores[0][2])
```

Następna instrukcja przypisuje liczbę 95 do elementu listy scores[2][1]:

```
scores[2][1] = 95
```

Listy dwuwymiarowe są przetwarzane za pomocą pętli zagnieżdzonych. Na przykład poniższy kod wyświetla wszystkie elementy z listy scores:

```
NUM_ROWS = 3
NUM_COLS = 3

row = 0
while row < NUM_ROWS:
    col = 0
    while col < NUM_COLS:
        print(scores[row][col])
        col = col + 1
    row = row + 1
```

Ten kod prosi użytkownika o wprowadzenie wartości dla każdego elementu znajdującego się na liście:

```
NUM_ROWS = 3
NUM_COLS = 3
row = 0
while row < NUM_ROWS:
    col = 0
    while col < NUM_COLS:
        scores[row][col] = int(input('Podaj wartość: '))
        col = col + 1
    row = row + 1
```

## C++

### Tablice

#### Deklaracje tablic w języku C++

Oto przykład deklaracji tablicy w C++:

```
int numbers [6];
```

Instrukcja ta deklaruje zmienną numbers jako tablicę typu int. Deklarator rozmiaru określa, że tablica zawiera 6 elementów. Jak już wspominałem, dobrą praktyką jest użycie stałej nazwanej jako deklaratora rozmiaru:

```
const int SIZE = 6;
int numbers[SIZE];
```

A tutaj inny przykład:

```
const int SIZE = 200;
double temperatures[SIZE];
```

Ten fragment kodu deklaruje zmienną `temperatures` jako tablicę 200 wartości typu `double`. Oto jeszcze jeden przykład:

```
const int SIZE = 10;
string names[SIZE];
```

Ta instrukcja deklaruje zmienną o nazwie `names` jako tablicę zawierającą 10 elementów typu `string`.

### **Indeksy i elementy tablicowe w języku C++**

Dostęp do każdego elementu tablicy możesz uzyskać za pomocą indeksu. Jak już wspominałem, indeks pierwszego elementu to 0, drugiego to 1 i tak dalej. Natomiast indeks ostatniego elementu to rozmiar całej tablicy minus 1. Na przykład poniższy kod deklaruje tablicę typu `int` z trzema elementami, a następnie przypisuje wartości do każdego z tych elementów; indeksy dla poszczególnych elementów to odpowiednio 0, 1 i 2:

```
const int SIZE = 3;
int numbers[SIZE];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
```

### **Inicjalizowanie tablicy w języku C++**

Podczas deklarowania tablicy możesz zainicjalizować ją różnymi wartościami. Oto przykład:

```
const int SIZE = 12;
int days[SIZE] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Instrukcje te deklarują zmienną `days` jako tablicę typu `int` i zapisują w niej wartości początkowe. Seria wartości zawartych w nawiasach klamrowych i oddzielonych przecinkami jest nazywana listą inicjalizującą. Wartości te są przechowywane w elementach tablicy w kolejności, w jakiej pojawiają się na liście (pierwsza wartość, 31, jest zapisywana w zmiennej `days[0]`, druga, 28, jest zapisywana w zmiennej `days[1]` i tak dalej.)

Podczas inicjalizowania tablicy nie ma konieczności podawania deklaratora rozmiaru. Kompilator C++ sam określi rozmiar tablicy na podstawie liczby elementów znajdujących się na liście inicjalizującej. Na przykład poprzednia deklaracja może być napisana w następujący sposób:

```
int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

### **Przekazywanie tablicy jako argumentu do funkcji w języku C++**

Podczas przekazywania tablicy jako argumentu do funkcji w C++ należy także przekazać oddzielnny argument typu `int` określający rozmiar tablicy. Poniższy kod przedstawia funkcję przyjmującą tablicę jako argument:

```
void showArray(int array[], int size)
{
    for (int i = 0; i < size; i++)
        cout << array[i] << " ";
```

Zauważ, że zmienna parametru `array` jest zadeklarowana jako tablica typu `int`, bez deklatora rozmiaru. Kiedy wywołujemy tę funkcję, musimy przekazać do niej tablicę typu `int` jako argument. Założmy, że zmienna `numbers` to nazwa tablicy typu `int`, a `SIZE`

jest stałą określającą jej rozmiar. Oto instrukcja wywołująca funkcję `showArray`, przekazującą jej jako argumenty tablicę `numbers` i stałą `SIZE`:

```
showArray(numbers, SIZE);
```

### Tablice dwuwymiarowe w języku C++

Oto przykład deklaracji tablicy dwuwymiarowej składającej się z trzech 3 i 4 kolumn:

```
double scores[3][4];
```

Dwa zestawy nawiasów kwadratowych umieszczone w typie danych wskazują, że zmienna `score` zawierać będzie tablicę dwuwymiarową. Liczby 3 i 4 są tutaj deklatorami rozmiaru. Pierwszy deklator określa liczbę wierszy, natomiast drugi — liczbę kolumn. Zauważ, że każdy deklator jest ujęty w osobne nawiasy kwadratowe.

Podczas przetwarzania danych w tablicy dwuwymiarowej każdemu elementowi nadawane są dwa indeksy: jeden dla wiersza i drugi dla kolumny. W tablicy `score` elementy w wierszu 0 są oznaczone w następujący sposób:

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

Elementy pierwszego wiersza to:

```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

Elementy drugiego wiersza to:

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

Aby uzyskać dostęp do jednego z elementów tablicy dwuwymiarowej, trzeba użyć obu indeksów. Na przykład ta instrukcja wypisuje liczbę 95 do elementu tablicy `scores[2][1]`:

```
scores[2][1] = 95;
```

Tablice dwuwymiarowe są przetwarzane za pomocą pętli zagnieżdżonych. Na przykład ten kod prosi użytkownika o podanie wartości dla każdego elementu tablicy:

```
const int ROWS = 3;
const int COLS = 4;
double scores[ROWS][COLS];
for (int row = 0; row < ROWS; row++)
{
```

```

for (int col = 0; col < COLS; col++)
{
    cout << "Podaj wartość." << endl;
    cin >> scores[row][col];
}
}

```

Poniższy kod wyświetla wszystkie elementy tablicy scores:

```

for (int row = 0; row < ROWS; row++)
{
    for (int col = 0; col < COLS; col++)
    {
        cout << scores[row][col] << endl;
    }
}

```

### Tablice wielowymiarowe w języku C++

W C++ można tworzyć tablice o dowolnej liczbie wymiarów. Oto przykład deklaracji tablicy trójwymiarowej:

```
double seats[3][5][8];
```

Tablicę tę można traktować jako 3 zestawy po 5 wierszy, z których każdy zawiera 8 elementów. Tablica może służyć do przechowywania cen poszczególnych miejsc w audytorium, gdzie znajduje się po osiem miejsc w rzędzie, sekcja ma po pięć rzędów i łącznie są trzy sekcje.

## Pytania kontrolne

### Test jednokrotnego wyboru

- Służy podczas deklarowania tablicy do określenia liczby elementów, z których będzie się ona składała.
  - indeks
  - rozmiar
  - nazwa
  - inicjalizacja
- Używa się ich do określania rozmiarów tablicy, aby łatwiej można było modyfikować program.
  - liczby rzeczywiste
  - wyrażenia z ciągiem znaków
  - wyrażenia matematyczne
  - stałe nazwane
- Jest to pojedyncza komórka tablicy.
  - element
  - kosz
  - kącik
  - rozmiar

4. Jest to liczba wskazująca konkretną komórkę w tablicy.
  - a) element
  - b) indeks
  - c) rozmiar
  - d) identyfikator
5. Jest to zazwyczaj numer pierwszego indeksu tablicy.
  - a) -1
  - b) 1
  - c) 0
  - d) rozmiar tablicy minus 1
6. Jest to zazwyczaj numer ostatniego indeksu tablicy.
  - a) -1
  - b) 99
  - c) 0
  - d) rozmiar tablicy minus 1
7. Ten algorytm wyszukiwania polega na prześledzeniu w pętli całej tablicy, począwszy do pierwszego elementu.
  - a) przeszukiwanie sekwencyjne
  - b) przeszukiwanie krok po kroku
  - c) przeszukiwanie elementarne
  - d) przeszukiwanie binarne
8. Z tej techniki korzysta wiele języków programowania, aby uniemożliwić odwołanie się do nieistniejącego elementu tablicy.
  - a) sprawdzanie
  - b) sprawdzanie zakresu indeksu
  - c) sprawdzanie kompatybilności typów
  - d) sprawdzanie składni
9. Nazywamy tak dwie lub więcej tablic, które zawierają powiązane ze sobą dane. Do powiązanych danych w kilku tablicach odwołujemy się poprzez ten sam indeks.
  - a) tablice synchroniczne
  - b) tablice asynchroniczne
  - c) tablice równoległe
  - d) tablice dwuwymiarowe
10. Tablicę dwuwymiarową można sobie wyobrazić jako:
  - a) linie i polecenia
  - b) rozdziały i strony
  - c) wiersze i kolumny
  - d) elementy poziome i pionowe

### Prawda czy fałsz?

1. W jednej tablicy można zapisać dane kilku typów.
2. W większości języków programowania nie można w trakcie działania programu zmienić rozmiaru tablicy.

3. Sprawdzanie zakresu indeksu ma najczęściej miejsce podczas działania programu.
4. Na tablicach można wykonywać wiele operacji, ale nie można ich przekazywać jako argument funkcji lub modułu.
5. Podczas deklarowania tablicy dwuwymiarowej wystarczy podać tylko jeden wymiar.

### Krótką odpowiedź

1. Na czym polega błąd typu off-by-one?
2. Spójrz na ten przykładowy pseudokod:

```
Constant Integer SIZE = 10
Declare Integer values[SIZE]
```

- a) Z ilu elementów składa się tablica?
  - b) Jaki indeks ma pierwszy element tablicy?
  - c) Jaki indeks ma ostatni element tablicy?
  3. Spójrz na ten przykładowy pseudokod:
- ```
Constant Integer SIZE = 3
Declare Integer numbers[SIZE] = 1, 2, 3
```
- a) Jaka wartość jest zapisana w elemencie numbers[2]?
  - b) Jaka wartość jest zapisana w elemencie numbers[0]?
  4. W pewnym programie występują dwie tablice równolegle: customerNumbers i balances. W tablicy customerNumbers są zapisane numery klientów, a w tablicy balances — salda ich rachunków. Zakładając, że numer klienta jest zapisany w elemencie customerNumbers[187], w którym elemencie będzie zapisane saldo jego
  5. Spójrz na następującą deklarację tablicy:

```
Declare Real sales[8][10]
```

- a) Z ilu wierszy składa się tablica?
- b) Z ilu kolumn składa się tablica?
- c) Z ilu elementów składa się tablica?
- d) Napisz polecenie, które przypisze dowolną liczbę do elementu w ostatnim wierszu i ostatniej kolumnie.

### Warsztat projektanta algorytmów

1. Napisz za pomocą pseudokodu deklarację tablicy zawierającej ciągi znaków i zainicjalizuj ją wartościami "Einstein", "Newton", "Kopernik" i "Kepler".
2. Założmy, że tablica names zawiera 20 liczb typu Integer. Zaprojektuj pętlę For, za pomocą której wyświetlisz wartości wszystkich elementów tablicy.
3. Założmy, że tablice numberArray1 i numberArray2 zawierają po 100 elementów. Zaprojektuj algorytm, który skopiuje wszystkie wartości z tablicy numberArray1 do tablicy numberArray2.
4. Narysuj schemat blokowy ilustrujący zasadę działania algorytmu sumującego wszystkie wartości w tablicy.

5. Narysuj schemat blokowy ilustrujący zasadę działania algorytmu wyszukującego wartość największego elementu tablicy.
6. Narysuj schemat blokowy ilustrujący zasadę działania algorytmu wyszukującego wartość najmniejszego elementu tablicy.
7. Założmy, że w programie pojawią się następujące deklaracje:

```
Constant Integer SIZE = 100
Declare Integer firstArray[SIZE]
Declare Integer secondArray[SIZE]
```

Założymy także, że do każdego elementu tablicy `firstArray` została przypisana wartość. Zaprojektuj algorytm, którego zadaniem będzie skopiowanie zawartości tablicy `firstArray` do tablicy `secondArray`.

8. Zaprojektuj funkcję przyjmującą jako argument tablicę liczb typu `Integer` i zwracającą sumę wartości wszystkich elementów tablicy.
9. Napisz w pseudokodzie algorytm, w którym za pomocą pętli `For Each` wyświetlisz wszystkie wartości zapisane w następującej tablicy:

```
Constant Integer SIZE = 10
Declare Integer values[SIZE] = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

## Ćwiczenia z wykrywania błędów

1. Na czym polega błąd w poniższym pseudokodzie?

```
// W programie wyświetlamy pięć imion zapisanych w tablicy
Constant Integer SIZE = 5
```

```
Declare String names[SIZE] = "Maria", "Jacek", "Stefan",
                    "Basia", "Lidia"
Declare Integer index
For index = 0 To SIZE
    Display names[index]
End For
```

2. Na czym polega błąd w poniższym pseudokodzie?

```
// Program wyświetla wartość największego elementu tablicy.
Declare Integer SIZE = 3
Declare Integer values[SIZE] = 1, 3, 4
Declare Integer index
Declare Integer highest

For index = 0 To SIZE - 1
    If values[index] > highest Then
        Set highest = values[index]
    End If
End For

Display "Największy element to ", highest
```

3. Na czym polega błąd w poniższym pseudokodzie?

```
// Funkcja searchName przyjmuje szukany ciąg znaków
// zawierający imię, tablicę ciągów znaków zawierającą imiona
// oraz liczbę całkowitą wskazującą rozmiar tablicy.
// Funkcja wyszukuje w tablicy dane imię. Jeżeli funkcja znajdzie imię,
// zwraca je; w przeciwnym razie zwraca komunikat
```

```

// informujący, że dane imię
// nie występuje w tablicy.
Function String searchName(String name, String names[], 
Integer size)
    Declare Boolean found
    Declare Integer index
    Declare String result

// Śledzimy tablicę i szukamy
// określonego imienia
    While found == False AND index <= size - 1
        If contains(names[index], name) Then
            Set found = True
        Else
            Set index = index + 1
        End If
    End While
// Określamy wynik wyszukiwania.
    If found == True Then
        Set result = names[index]
    Else
        Set result = "Tego imienia nie ma w tablicy."
    End If
    Return result
End Function

```

## Ćwiczenia programistyczne

### 1. Suma sprzedaży

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie wartości sprzedaży w każdym dniu tygodnia. Wartości powinny zostać zapisane w tablicy. Następnie za pomocą pętli oblicz sumę sprzedaży i wyświetl ją na ekranie.

### 2. Generator numeru losu na loterii

Zaprojektuj program, który będzie generował 7-cyfrowy numer losu. W programie powinna się znaleźć tablica zawierająca 7 liczb typu `Integer`. Stwórz pętlę, w której do każdego elementu tablicy przypiszesz losową liczbę z przedziału 0 – 9 (skorzystaj z funkcji `random`, którą opisałem w rozdziale 6.). Następnie za pomocą kolejnej pętli wyświetl zawartość tablicy.

### 3. Opady deszczu

Zaprojektuj program, w którym użytkownik będzie mógł zapisać w tablicy poziom opadów w każdym z 12 miesięcy. Program powinien obliczyć i wyświetlić sumę opadów w roku, średni miesięczny opad i miesiące z najwyższym i najniższym poziomem opadów.

### 4. Program do analizy liczbowej

Zaprojektuj program, w którym użytkownik będzie mógł wprowadzić 20 liczb. Program powinien zapisać liczby w tablicy, a potem wyświetlić następujące informacje:

- wartość najmniejszej liczby w tablicy;
- wartość największej liczby w tablicy;
- sumę wszystkich liczb w tablicy;
- średnią wartość wszystkich liczb w tablicy.

## 5. Sprawdzanie kodu doładowującego telefon

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie kodu doładowującego telefon. Program powinien sprawdzać, czy dany kod jest prawidłowy, porównując go z następującą listą prawidłowych kodów:

```
5658845 4520125 7895122 8777541 8451277 1302850
8080152 4562555 5552012 5050552 7825877 1250255
1005231 6545231 3852085 7576651 7881200 4581002
```

Kody te należy zapisać w tablicy. Aby sprawdzić poprawność kodu, wykorzystaj algorytm przeszukiwania sekwencyjnego. Jeżeli kod wprowadzony przez użytkownika znajduje się w tablicy, wyświetl komunikat informujący, że kod jest prawidłowy. Jeżeli kod nie znajduje się w tablicy, wyświetl komunikat informujący, że kod nie jest prawidłowy.

## 6. Liczba dni w poszczególnych miesiącach

Zaprojektuj program, który będzie wyświetlał liczbę dni w każdym z miesięcy. Program powinien wyświetlać następujące

```
Styczeń ma 31 dni.
Luty ma 28 dni.
Marzec ma 31 dni.
Kwiecień ma 30 dni.
Maj ma 31 dni.
Czerwiec ma 30 dni.
Lipiec ma 31 dni.
Sierpień ma 31 dni.
Wrzesień ma 30 dni.
Październik ma 31 dni.
Listopad ma 30 dni.
Grudzień ma 31 dni.
```

W programie powinny występować dwie tablice równolegle: 12-elementowa tablica zawierająca ciągi znaków zainicjalizowana nazwami miesięcy i 12-elementowa tablica liczb całkowitych Integer zainicjalizowana liczbą dni w poszczególnych miesiącach. Aby wyświetlić wynik, prześledź za pomocą pętli obie tablice, pobierając z nich nazwę miesiąca i liczbę dni.

## 7. Wyszukiwanie numeru telefonu

Zaprojektuj program, w którym wykorzystane będą dwie tablice równolegle: tablica ciągów znaków o nazwie `people` zainicjalizowana imionami siedmiorga Twoich przyjaciół i tablica ciągów znaków o nazwie `phoneNumbers` zainicjalizowana numerami telefonów Twoich przyjaciół. Program powinien umożliwić użytkownikowi wprowadzenie imienia, a następnie wyszukać je w tablicy `people`. Jeżeli imię zostanie odnalezione, program powinien odczytać numer telefonu danej osoby z tablicy `phoneNumbers` i wyświetlić go. Jeżeli dana osoba nie zostanie odnaleziona w tablicy `people`, program powinien o tym poinformować za pomocą odpowiedniego komunikatu.

## 8. Lista płac

Zaprojektuj program, w którym wykorzystasz następujące tablice równolegle:

- `empId` — tablica siedmiu liczb typu `Integer` zawierająca identyfikatory pracowników. Zainicjalizuj tablicę następującymi liczbami:  
56588 45201 78951 87775 84512 13028 75804
- `hours` — tablica siedmiu liczb typu `Integer` zawierająca liczbę godzin przepracowanych przez danego pracownika.
- `payRate` — tablica siedmiu liczb typu `Real` zawierająca stawkę godzinową danego pracownika.
- `wages` — tablica siedmiu liczb typu `Real` zawierająca całkowite wynagrodzenie danego pracownika.

Program powinien za pomocą wspólnego indeksu powiązać ze sobą te dane. Przykładowo indeks 0 powinien wskazywać w tablicy `hours` liczbę godzin przepracowanych przez pracownika o identyfikatorze zapisanym w elemencie o indeksie 0 w tablicy `empId`. Stawka godzinowa tego pracownika powinna być zapisana w elemencie o indeksie 0 w tablicy `payRate`.

Program powinien wyświetlać identyfikator pracownika i prosić użytkownika o wprowadzenie liczby przepracowanych przez niego godzin i wysokości stawki godzinowej. Następnie powinien obliczyć wynagrodzenie całkowite tego pracownika (liczba godzin pomnożona przez stawkę godzinową) i zapisać je w tablicy `wages`. Po wprowadzeniu danych dla wszystkich pracowników program powinien wyświetlić identyfikator każdego pracownika i wysokość całkowitego wynagrodzenia.

## 9. Egzamin na prawo jazdy

Ośrodek przeprowadzający egzaminy na prawo jazdy poprosił Cię o zaprojektowanie programu, który będzie oceniał teoretyczną część egzaminu. Egzamin składa się z 20 pytań jednokrotnego wyboru.

Oto prawidłowe odpowiedzi na pytania:

- |      |       |       |       |
|------|-------|-------|-------|
| 1. B | 6. A  | 11. B | 16. C |
| 2. D | 7. B  | 12. C | 17. C |
| 3. A | 8. A  | 13. D | 18. B |
| 4. A | 9. C  | 14. A | 19. D |
| 5. C | 10. D | 15. D | 20. A |

Umieść te odpowiedzi w tablicy (odpowiedź na każde pytanie zapisz w elemencie tablicy zawierającej ciągi znaków). Program powinien poprosić użytkownika o wprowadzenie odpowiedzi danego kandydata na każde z 20 pytań i zapisać je w drugiej tablicy. Po wprowadzeniu odpowiedzi program powinien wyświetlić informację, czy kandydat zdal egzamin teoretyczny, czy nie. Aby zdać egzamin, kandydat musi odpowiedzieć prawidłowo na 15 z 20 pytań. Następnie program powinien wyświetlić liczbę prawidłowych odpowiedzi, liczbę nieprawidłowych odpowiedzi oraz listę numerów pytań, na które kandydat odpowiedział błędnie.

## 10. Punkty siodłowe

Zaprojektuj program, który będzie tworzył tablicę dwuwymiarową z liczbami całkowitymi składającą się z 7 wierszy i 7 kolumn. Program powinien w każdym elemencie tablicy umieścić liczbę losową, następnie przeszukiwać tablicę pod kątem punktów siodłowych. Punkt siodłowy to element, którego wartość jest równa wszystkim innym wartościom w tym samym wierszu lub od nich mniejsza i jest równa innym wartościom w tej samej kolumnie lub do nich większa. Program powinien wyświetlać wartości wszystkich punktów siodłowych znalezionych w tablicy (jeśli takie istnieją).

## 11. Kółko i krzyżyk

Zaprojektuj program, który będzie działał jak gra w kółko i krzyżyk. Jako planszę do gry wykorzystaj dwuwymiarową tablicę ciągów znaków składającą się z 3 wierszy i 3 kolumn. Każdy element tablicy zainicjalizuj wartością "\*\*\*". Program powinien zawierać pętlę wykonującą następujące operacje:

- Wyświetlanie zawartości tablicy reprezentującej planszę.
- Umożliwienie graczu 1 postawienia w danym polu znaku X. Program powinien poprosić użytkownika o wprowadzenie numeru wiersza i kolumny.
- Umożliwienie graczu 2 postawienia w danym polu znaku O. Program powinien poprosić użytkownika o wprowadzenie numeru wiersza i kolumny.
- Sprawdzenie, czy któryś z graczy wygrał lub czy gra zakończyła się remisem.  
Jeżeli któryś z graczy wygrał, program powinien poinformować, kto jest zwycięzcą, i zakończyć działanie. Jeżeli nastąpił remis, program powinien o tym poinformować i rozpocząć grę od początku.
- Gracz 1 wygrywa wtedy, gdy w wierszu, kolumnie lub po przekątnej znajdują się trzy znaki X. Gracz 2 wygrywa wtedy, gdy w wierszu, kolumnie lub po przekątnej znajdują się trzy znaki O. Remis ma miejsce wtedy, gdy wszystkie miejsca na planszy zostaną zajęte, a zwycięzca nie zostanie wyłoniony.

## 12. Magiczny kwadrat Lo Shu

Magiczny kwadrat Lo Shu składa się z pól podzielonych na 3 wiersze i 3 kolumny, jak na rysunku 8.23

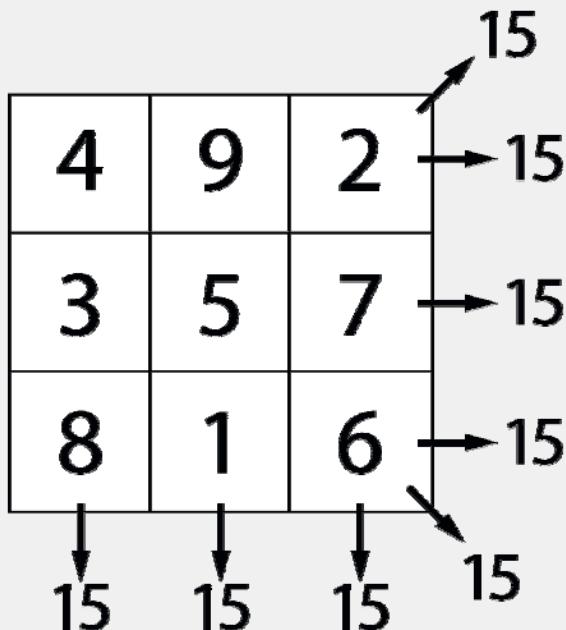
4	9	2
3	5	7
8	1	6

Rysunek 8.23. Magiczny kwadrat Lo Shu

Kwadrat ten cechuje się następującymi właściwościami:

- W polach umieszczone są niepowtarzające się liczby od 1 do 9.
- Sumy liczb w każdym wierszu, każdej kolumnie i po przekątnych są sobie równe. Przedstawia to rysunek 8.24.

Aby przedstawić taki kwadrat w programie, możesz wykorzystać tablicę dwuwymiarową. Zaprojektuj program, w którym zainicjalizujesz tablicę dwuwymiarową wartościami wprowadzonymi przez użytkownika. Program powinien następnie sprawdzić, czy kwadrat jest magicznym kwadratem Lo Shu.



**Rysunek 8.24.** Sumy wartości w wierszach, kolumnach i po przekątnych w magicznym kwadracie Lo Shu

# Sortowanie i przeszukiwanie tabel

## TEMATYKA

- |  |   |
|--|---|
| 9.1 Algorytm sortowania bąbelkowego      | 9.3 Algorytm sortowania przez wstawianie  |
| 9.2 Algorytm sortowania przez wybieranie | 9.4 Algorytm wyszukiwania binarnego       |
|  | 9.5 Rzut oka na języki Java, Python i C++ |

### 9.1

## Algorytm sortowania bąbelkowego

**WYJAŚNIENIE:** Zadaniem algorytmu sortowania jest uporządkowanie elementów tablicy w określonej kolejności. Przykładem prostego algorytmu sortowania jest algorytm sortowania bąbelkowego.

## Algorytmy sortowania

W wielu zadaniach programistycznych trzeba uporządkować dane zawarte w tablicy tak, aby występuły one w określonej kolejności. Przykładowo lista klientów jest zazwyczaj posortowana w kolejności alfabetycznej, oceny uczniów mogą być posortowane od najwyższej do najniższej, a identyfikatory produktów — w taki sposób, aby produkty w takim samym kolorze znajdowały się obok siebie. Aby posortować dane zawarte w tablicy, programista musi skorzystać z odpowiedniego algorytmu sortowania. **Algorytm sortowania** to technika polegająca na prześledzeniu elementów tablicy i uporządkowaniu ich w określonej kolejności.

Dane w tablicy można uporządkować w kolejności rosnącej lub malejącej. W tablicy, w której elementy uporządkowane są w **kolejności rosnącej**, wartości zapisane są od najmniejszej do największej. W tablicy, w której elementy są uporządkowane w **kolejności malejącej**, wartości zapisane są od największej do najmniejszej. W niniejszym

rozdziale przedstawię trzy algorytmy sortowania, za pomocą których można uporządkować elementy w tablicy: **sortowanie bąbelkowe**, **sortowanie przez wybieranie** i **sortowanie przez wstawianie**. W tym podrozdziale zajmę się algorytmem sortowania bąbelkowego.

## Sortowanie bąbelkowe

Z pomocą algorytmu **sortowania bąbelkowego** można bardzo łatwo uporządkować elementy tablicy w kolejności rosnącej lub malejącej. Nazwa tego algorytmu wynika stąd, że wykonujemy w nim kolejne przejścia względem całej tablicy, porównując ze sobą wartości elementów i przesuwając niektóre z nich w kierunku końca tablicy — przypomina to przemieszczanie się bąbelków w napoju gazowanym. Przykładowo, gdy skorzystamy z tego algorytmu, aby uporządkować elementy w kolejności rosnącej, wówczas elementy tablicy o większej wartości będą się przesuwały w kierunku jej końca. W przypadku sortowania w kolejności malejącej w kierunku końca tablicy będą się przesuwały jej elementy o mniejszej wartości. W tym podrozdziale pokażę, jak za pomocą algorytmu sortowania bąbelkowego można uporządkować elementy tablicy w kolejności rosnącej.

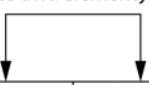
Załóżmy, że mamy do czynienia z tablicą taką jak na rysunku 9.1. Spójrzmy teraz, jak za pomocą sortowania bąbelkowego możemy uporządkować jej elementy w kolejności rosnącej.

7	2	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Rysunek 9.1. Tablica

Sortowanie bąbelkowe zaczynamy od porównania ze sobą dwóch pierwszych elementów tablicy. Jeżeli element o indeksie 0 jest większy od elementu o indeksie 1, zamieniamy je miejscami. Tablica będzie teraz wyglądała jak na rysunku 9.2.

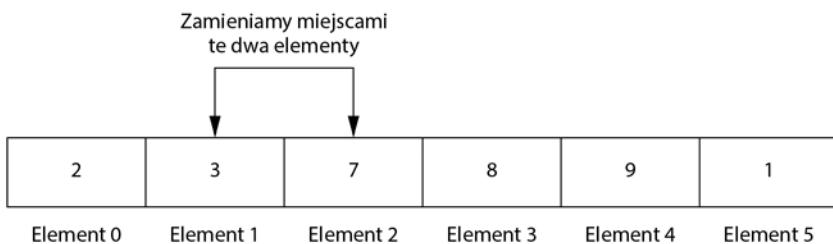
Zamieniamy miejscami te dwa elementy



2	7	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Rysunek 9.2. Zamienione miejscami elementy 0 i 1

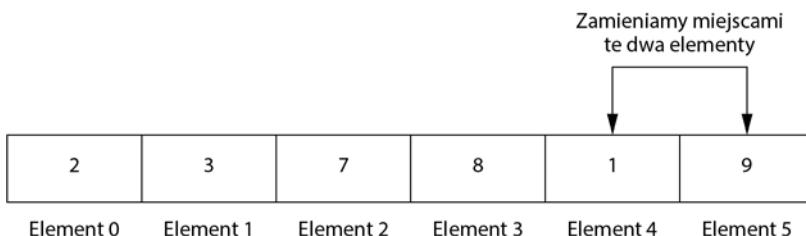
Następnie powtarzamy tę operację, ale tym razem porównujemy ze sobą elementy o indeksach 1 i 2. Jeżeli element o indeksie 1 jest większy od elementu o indeksie 2, zamieniamy je miejscami. Tablica będzie teraz wyglądała jak na rysunku 9.3.



Rysunek 9.3. Zamienione miejscami elementy 1 i 2

Następnie porównujemy elementy 2 i 3. W przypadku naszej tablicy elementy te są ustawione w prawidłowej kolejności (element o indeksie 2 jest mniejszy od elementu o indeksie 3), więc nie zamieniamy ich miejscami. W następnej kolejności porównujemy elementy o indeksach 3 i 4 i w tym przypadku również nie musimy ich przedstawiać.

Kiedy jednak porównamy elementy o indeksach 4 i 5, będziemy je musieli zamienić miejscami, ponieważ element o indeksie 4 ma większą wartość niż element o indeksie 5. Tablica będzie teraz wyglądała jak na rysunku 9.4.

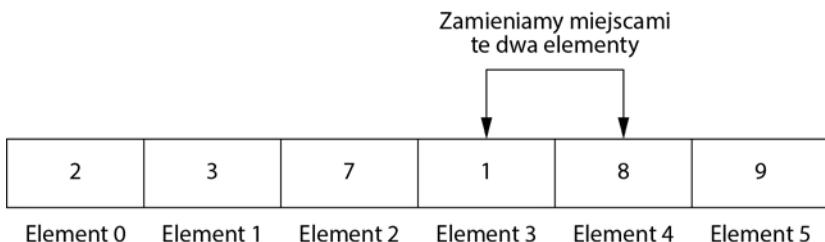


Rysunek 9.4. Zamienione miejscami elementy 4 i 5

Po tym kroku zakończyliśmy pierwsze przejście przez całą tablicę, a jej największa wartość, równa 9, znajduje się już na właściwym miejscu, ale inne elementy nie są jeszcze uporządkowane. Algorytm wykona więc kolejne przejście przez tablicę, porównując ze sobą sąsiadujące elementy. Jednak w tym przypadku porównywanie zakończymy na przedostatnim elemencie, ponieważ ostatni znajduje się już na właściwym miejscu.

Drugie przejście przez tablicę też zaczniemy od elementów o indeksach 0 i 1. Elementy te ustawione są w prawidłowej kolejności, więc nie zamieniamy ich miejscami. Następnie porównujemy elementy o indeksach 1 i 2 i ponownie okazuje się, że nie trzeba ich zamieniać. Dzieje się tak do momentu, gdy porównamy ze sobą elementy o indeksach 3 i 4. Ponieważ element o indeksie 3 ma większą wartość niż element o indeksie 4, zamieniamy je miejscami. Element o indeksie 4 jest ostatnim elementem, który porównujemy podczas tego przejścia. Tablica będzie teraz wyglądała jak na rysunku 9.5.

Po drugim przejściu przez tablicę na właściwym miejscu znajdują się dwa ostatnie elementy. Rozpoczynamy trzecie przejście, znów porównując ze sobą sąsiadujące elementy — jednak tym razem pomijamy dwa ostatnie, ponieważ zostały one już posortowane. Po zakończeniu trzeciego przejścia na właściwym miejscu znajdują się trzy ostatnie elementy, co przedstawiłem na rysunku 9.6.

**Rysunek 9.5.** Zamienione miejscami elementy 3 i 4

2	3	1	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

**Rysunek 9.6.** Tablica po trzecim przejściu

Po każdym przejściu przez tablicę liczba porównywanych elementów zmniejsza się o 1, a jedna z wartości trafia na właściwe miejsce. Kiedy algorytm wykona już wszystkie przejścia, tablica będzie wyglądała jak na rysunku 9.7.

1	2	3	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

**Rysunek 9.7.** Tablica z posortowanymi elementami

## Zamienianie elementów miejscami

Jak miałeś okazję zauważyć, algorytm sortowania bąbelkowego zamienia miejscami niektóre elementy tablicy. Omówmy więc krótko, jak wygląda sam proces zamiany dwóch elementów w pamięci komputera. Założymy, że w programie występują takie deklaracje zmiennych:

```
Declare Integer a = 1
Declare Integer b = 9
```

Powiedzmy, że chcemy zamienić wartości w tych zmiennych tak, aby w zmiennej **a** znalazła się wartość 9, a w zmiennej **b** znalazła się wartość 1. Na pierwszy rzut oka mogłoby się wydawać, że wystarczy, jeśli przypiszemy po prostu jedną zmienną do drugiej, w następujący sposób:

```
// BLĄD! Ten kod nie spowoduje zamiany wartości w zmiennych
Set a = b
Set b = a
```

Prześledźmy ten kod, aby zrozumieć, dlaczego nie zadziała on prawidłowo. Pierwsze polecenie to **Set a = b**. Po jego wykonaniu do zmiennej **a** zostanie przypisana wartość 9. Ale co się stało z wartością 1, uprzednio zapisaną w zmiennej **a**? Przypominię, że po przypisaniu do zmiennej nowej wartości zastąpi ona poprzednią wartość

przypisaną do tej zmiennej. Tak więc poprzednia wartość, równa 1, zostanie po prostu usunięta. Następne polecenie to Set  $b = a$ . Ponieważ w zmiennej  $a$  jest teraz zapisana wartość 9, do zmiennej  $b$  także zostanie przypisana wartość 9. Po wykonaniu tych poleceń zarówno zmieniona  $a$ , jak i zmieniona  $b$  będą równe 9.

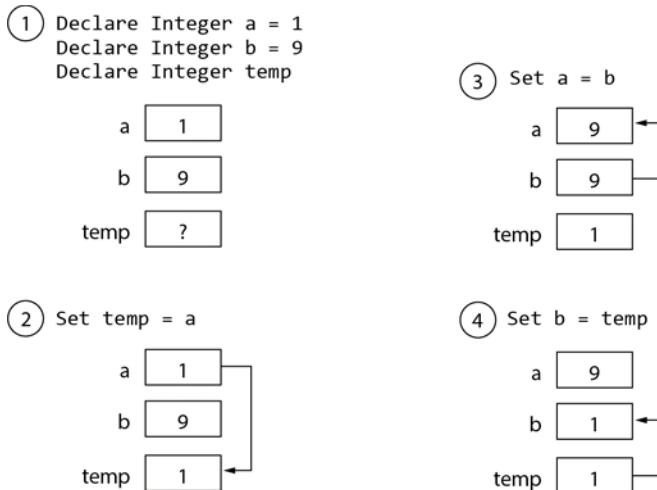
Aby prawidłowo zamienić wartości dwóch zmiennych, będziemy potrzebować trzeciej zmiennej, która posłuży jako tymczasowe miejsce zapisu:

```
Declare Integer temp
```

Teraz możemy przystąpić do zamiany wartości dwóch zmiennych w następujący sposób:

- do zmiennej  $temp$  przypisujemy wartość zmiennej  $a$ ;
- do zmiennej  $a$  przypisujemy wartość zmiennej  $b$ ;
- do zmiennej  $b$  przypisujemy wartość zmiennej  $temp$ .

Na rysunku 9.8 przedstawiłem wartości, jakie znajdą się w zmiennych podczas każdego z tych kroków. Zwróć uwagę, że po wykonaniu ostatniego kroku wartości zmienionych  $a$  i  $b$  są już zamienione.

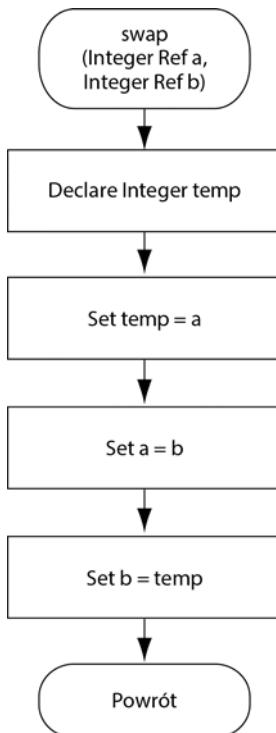


Rysunek 9.8. Zamiana wartości zmiennych a i b

Stwórzmy więc moduł o nazwie `swap`, który będzie zamieniał ze sobą dwie zmienne. Skorzystamy z tego modułu w algorytmie sortowania bąbelkowego. Na rysunku 9.9 widoczny jest schemat blokowy modułu `swap`. Zwróć uwagę, że w module znajdują się dwa parametry typu referencyjnego o nazwach  $a$  i  $b$ . Podczas wywołania tego modułu przekażemy do niego jako argumenty dwie zmienne (lub dwa elementy tablicy). Gdy moduł zakończy działanie, wartości zostaną zamienione.



**UWAGA:** Bardzo ważne jest, aby w module `swap` zadeklarować parametry typu referencyjnego, ponieważ moduł musi mieć możliwość modyfikacji zmiennych przekazanych jako argumenty.



**Rysunek 9.9.** Schemat blokowy modułu swap

Oto pseudokod modułu swap:

```

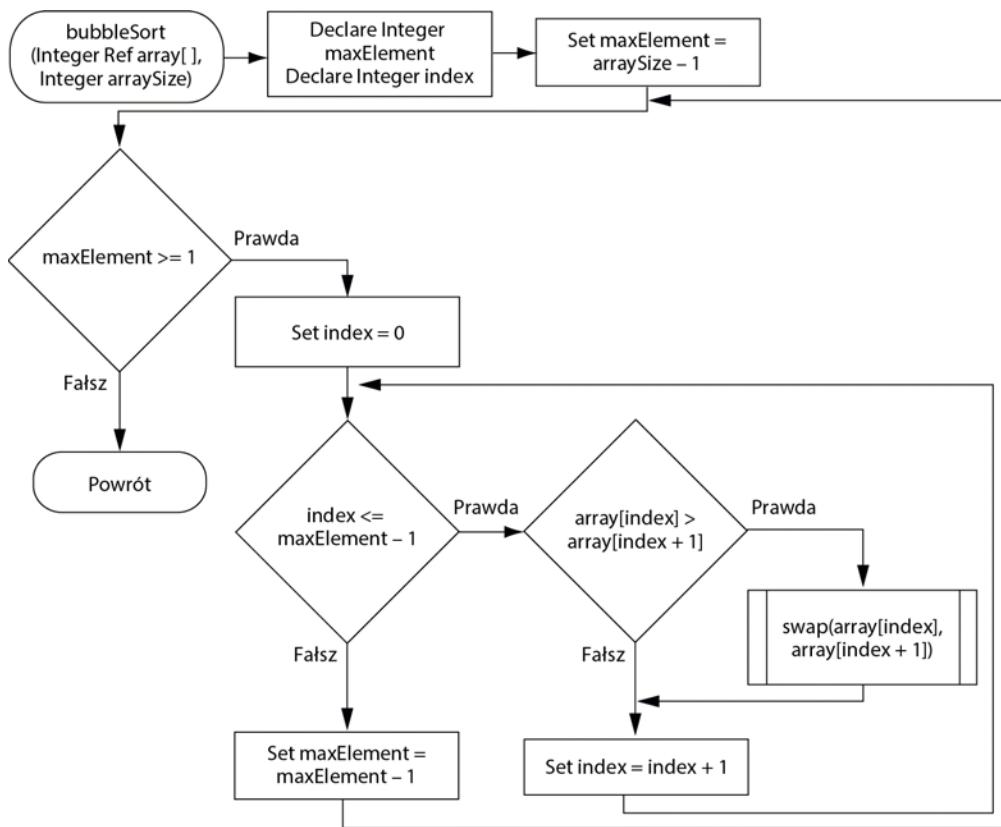
Module swap(Integer Ref a, Integer Ref b)
  // Tymczasowa zmieniona lokalna
  Declare Integer temp

  // Zamieniamy ze sobą wartości w zmienionych a i b
  Set temp = a
  Set a = b
  Set b = temp
End Module
  
```

Zaprezentowana wersja modułu swap będzie zamieniała tylko zmienne typu Integer. Jeżeli będziemy chcieli zamieniać wartości innych typów, będziemy musieli zmienić typ parametrów a i b oraz typ zmiennej temp.

## Projektowanie algorytmu sortowania bąbelkowego

Algorytm sortowania bąbelkowego zazwyczaj umieszcza się w jednym z modułów programu. Kiedy zechcesz posortować tablicę, wystarczy, że przekażesz ją do modułu, a on ją posortuje. Na rysunku 9.10 widoczny jest schemat blokowy modułu bubbleSort, którego zadaniem jest posortowanie tablicy zawierającej liczby Integer. Na listingu 9.1 przedstawiłem pseudokod tego modułu (zauważ, że na listingu 9.1 znajduje się tylko pseudokod modułu i nie stanowi on kompletnego programu).



Rysunek 9.10. Schemat blokowy algorytmu sortowania bąbelkowego

### Listing 9.1.

#### Moduł bubbleSort (to nie jest pełny program)



```

1 Module bubbleSort(Integer Ref array[], Integer arraySize)
2   // Zmienna maxElement będzie zawierała indeks ostatniego elementu tablicy,
3   // który ma zostać porównany
4   Declare Integer maxElement
5
6   // Zmienna index służy jako licznik
7   // w pętli wewnętrznej
8   Declare Integer index
9
10  // Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
11  // ostatniemu indeksowi tablicy, który ma zostać porównany w danym
12  // przejściu. Na samym początku zmienna maxElement będzie równa
13  // indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
14  // będzie ona dekrementowana o 1
15  For maxElement = arraySize - 1 To 0 Step -1
16
17  // Pętla wewnętrzna śledzi tablicę i porównuje
18  // sąsiadujące ze sobą elementy. Porównywane są
19  // elementy o indeksach od 0 do maxElement.
20  // Jeżeli kolejność elementów jest nieprawidłowa,
21  // zostają one zamienione miejscami
22  For index = 0 To maxElement - 1
  
```

```

23
24      // Porównujemy dwa sąsiadujące elementy
25      // i jeśli trzeba, zamieniamy je miejscami
26      If array[index] > array[index + 1] Then
27          Call swap(array[index], array[index + 1])
28      End If
29  End For
30 End For
31 End Module

```

W liniach 4. i 8. deklaruję następujące zmienne:

- `maxElement` — zapisuję w niej indeks ostatniego elementu, który w pętli ma zostać porównany z elementem sąsiednim;
- `index` — zapisuję w niej indeks tablicy w jednej z pętli.

W module pojawiają się dwie pętle `For` — jedna z nich jest zagnieźdzona w drugiej. Pętla zewnętrzna rozpoczyna się w linii 15. od następującego polecenia:

```
For maxElement = arraySize - 1 To 0 Step -1
```

Pętla ta wykona po jednej iteracji dla każdego elementu tablicy. Zmienna `maxElement` będzie przybierała wartości, począwszy od najwyższego indeksu tablicy do indeksu 0. Po zakończeniu każdej iteracji zmienna `maxElement` jest dekrementowana o 1.

Druga pętla, zagnieźdzona w pierwszej, rozpoczyna się w linii 22.:

```
For index = 0 To maxElement - 1
```

Wykonuje ona iterację dla każdego nieposortowanego elementu tablicy. Licznik `index` przybiera wartości od 0 do `maxElement - 1`. W każdej iteracji sprawdzany jest następujący warunek w linii 26.:

```
If array[index] > array[index + 1] Then
```

Za pomocą instrukcji `If` porównujemy `element[index]` z elementem sąsiednim `array[index + 1]`. Jeżeli element sąsiedni jest większy, zamieniamy je miejscami w linii 27. (moduł `swap` należy umieścić w każdym programie, w którym będziemy korzystać z modułu `bubbleSort`). W sekcji „**W centrum uwagi**” pokazałem, jak można wykorzystać algorytm sortowania bąbelkowego w pełnym programie.

## **W centrum uwagi**

### Korzystanie z algorytmu sortowania bąbelkowego

Gdy Kasia wystawi już oceny ze sprawdzianów, chciałaby, aby program wyświetlił je uporządkowane od najniższej do najwyższej. Poprosiła Cię więc o zaprojektowanie programu, który umożliwi jej wprowadzenie wyników ze sprawdzianów, a następnie wyświetli w kolejności rosnącej posortowaną listę tych wyników. Oto poszczególne kroki algorytmu takiego programu:



1. Pobierz od użytkownika serię wyników ze sprawdzianów i zapisz je w tablicy.
2. Posortuj tablicę w kolejności rosnącej.
3. Wyświetl zawartość tablicy.

Aby przetestować program, prosisz Kasię, aby sprawdziła go na swojej najmniej licznej klasie, składającej się z zaledwie sześciu uczniów. Jeśli będzie zadowolona z programu, zmodyfikujesz go, aby działał także w przypadku liczniejszych klas. Kasia zgodziła się na takie rozwiązanie.

Na listingu 9.2 znajduje się pseudokod programu, który podzieliłem na moduły. Nie pokazalem go jednak w całości, gdyż każdy moduł omówię oddzielnie. Na początek przyjrzyjmy się modułowi `main`.

### **Listing 9.2. Program sortujący wyniki w kolejności malejącej: moduł main**

```

1 Module main()
2   // Stała równa rozmiarowi tablicy
3   Constant Integer SIZE = 6
4
5   // Tablica, w której zapiszemy wyniki ze sprawdzianów
6   Declare Integer testScores[SIZE]
7
8   // Pobieramy wyniki ze sprawdzianów
9   getTestScores(testScores, SIZE)
10
11  // Sortujemy wyniki ze sprawdzianów
12  bubbleSort(testScores, SIZE)
13
14  // Wyświetlamy wyniki ze sprawdzianów
15  Display "Oto wyniki ze sprawdzianów"
16  Display "uporządkowane od najniższego do najwyższego:"
17  showTestScores(testScores, SIZE)
18 End Module
19

```

W linii 3. deklaruję stałą `SIZE` i zapisuję w niej rozmiar tablicy. W linii 6. deklaruję tablicę `testScores`, w której zapiszę wyniki ze sprawdzianów. W linii 9. przekazuję do modułu `getTestScores` tablicę `arrayScores` i stałą `SIZE`. Jak za chwilę zobaczysz, tablicę przekazuję przez referencję. Zadaniem modułu jest pobranie od użytkownika wyników ze sprawdzianów i zapisanie ich w tablicy.

W linii 12. przekazuję tablicę `testScores` i stałą `SIZE` do modułu `bubbleSort` (przez referencję). Po wywołaniu modułu elementy tablicy zostaną uporządkowane w kolejności rosnącej.

W linii 17. przekazuję tablicę `testScores` i stałą `SIZE` do modułu `showTestScores`. Zadaniem modułu jest wyświetlenie zawartości tablicy.

Następnie omówię moduł `getTestScores`.

**Listing 9.2.****Program sortujący wyniki w kolejności malejącej (kontynuacja):  
moduł getTestScores**

```

20 // Moduł getTestScores prosi użytkownika
21 // o wprowadzenie wyników ze sprawdzianów i zapisuje je w tablicy
22 // przekazanej jako argument
23 Module getTestScores(Integer Ref array[], Integer arraySize)
24     // Zmienna licznikowa
25     Declare Integer index
26
27     // Pobieramy wyniki ze sprawdzianów
28     For index = 0 to arraySize - 1
29         Display "Wprowadź wynik ze sprawdzianu numer ", index + 1
30         Input array[index]
31     End For
32 End Module
33

```

Moduł getTestScores ma dwa parametry:

- array[] — tablica liczb typu Integer przekazywana przez referencję;
- arraySize — zmienna typu Integer wskazująca rozmiar przekazywanej tablicy.

Zadaniem tego modułu jest pobranie od użytkownika wyników ze sprawdzianów i zapisanie ich w tablicy przekazanej przez argument array. W następnej kolejności przedstawię moduły bubbleSort i swap. Wyglądają one tak samo jak te, które przestawilem wcześniej w tym rozdziale.

**Listing 9.2.****Program sortujący wyniki w kolejności malejącej (kontynuacja):  
moduły bubbleSort i swap**

```

34 // Moduł bubbleSort przyjmuje jako argumenty tablicę liczb typu Integer
35 // i rozmiar tej tablicy. Gdy moduł zakończy działanie,
36 // wartości zapisane w tablicy będą posortowane w kolejności
37 // rosnącej
38 Module bubbleSort(Integer Ref array[], Integer arraySize)
39     // Zmienna maxElement będzie zawierała indeks ostatniego elementu tablicy,
40     // który ma zostać porównany
41     Declare Integer maxElement
42
43     // Zmienna index służy jako licznik
44     // w pętli wewnętrznej
45     Declare Integer index
46
47     // Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
48     // ostatniemu indeksowi tablicy, który ma zostać porównany w danym
49     // przejściu. Na samym pocztku zmienna maxElement będzie równa
50     // indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
51     // będzie ona dekrementowana o 1
52     For maxElement = arraySize - 1 To 0 Step -1
53
54         // Pętla wewnętrzna śledzi tablicę i porównuje
55         // sąsiadujące ze sobą elementy. Porównywane są
56         // elementy o indeksach od 0 do maxElement.
57         // Jeżeli kolejność elementów jest nieprawidłowa,
58         // zostają one zamienione miejscami
59         For index = 0 To maxElement - 1

```

```

60          // Porównujemy dwa sąsiadujące elementy
61          // i jeśli trzeba, zamieniamy je miejscami
62          If array[index] > array[index + 1] Then
63              Call swap(array[index], array[index + 1])
64          End If
65      End For
66  End For
68 End Module
69
70 // Moduł swap przyjmuje dwa argumenty typu Integer
71 // i zamienia ich wartości
72 Module swap(Integer Ref a, Integer Ref b)
73     // Tymczasowa zmienna lokalna
74     Declare Integer temp
75
76     // Zamieniamy wartości w zmiennych a i b
77     Set temp = a
78     Set a = b
79     Set b = temp
80 End Module
81

```

Na końcu programu pojawia się definicja modułu showTestScores.

**Listing 9.2. Program sortujący wyniki w kolejności malejącej (kontynuacja): moduł showTestScores**

```

82 // Moduł showTestScores wyświetla zawartość
83 // tablicy przekazanej jako argument
84 Module showTestScores(Integer array[], Integer arraySize)
85     // Zmienna licznikowa
86     Declare Integer index
87
88     // Wyświetlamy wyniki ze sprawdzianów
89     For index = 0 to arraySize - 1
90         Display array[index]
91     End For
92 End Module

```

Moduł showTestScores ma dwa parametry:

- array[] — tablica liczb typu Integer przekazywana przez referencję;
- arraySize — zmienna typu Integer wskazująca rozmiar przekazywanej tablicy.

Zadaniem tego modułu jest wyświetlenie zawartości tablicy przekazanej przez argument array.

**Wynik działania programu (po grubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik ze sprawdzianu numer 1

88 [Enter]

Wprowadź wynik ze sprawdzianu numer 2

92 [Enter]

Wprowadź wynik ze sprawdzianu numer 3

**73 [Enter]**

Wprowadź wynik ze sprawdzianu numer 4

**69 [Enter]**

Wprowadź wynik ze sprawdzianu numer 5

**98 [Enter]**

Wprowadź wynik ze sprawdzianu numer 6

**79 [Enter]**

Oto wyniki ze sprawdzianów

uporządkowane od najniższego do najwyższego:

69

73

79

88

92

98

## Sortowanie tablicy zawierającej ciągi znaków

W rozdziale 4. wspomniałem, że w większości języków programowania można sprawdzić, czy dany ciąg znaków jest większy, mniejszy, równy czy różny od drugiego ciągu znaków. Dzięki temu algorytm sortowania bąbelkowego można także zastosować do porządkowania ciągów znaków — na przykład aby posortować je w porządku alfabetycznym. Na listingu 9.3 pokazałem przykład. Zauważ, że moduły bubbleSort i swap dostosowałem tak, aby działały na tablicach zawierających ciągi znaków.

### Listing 9.3

```

1 Module main()
2 // Stała równa rozmiarowi tablicy
3 Constant Integer SIZE = 6
4
5 // Tablica ciągów znaków
6 Declare String names[SIZE] = "Dawid", "Anna", "Maria",
7                               "Beata", "Jurek", "Damian"
8
9 // Zmienna licznikowa
10 Declare Integer index
11
12 // Wyświetlamy elementy w pierwotnej kolejności
13 Display "Pierwotna kolejność:"
14 For index = 0 To SIZE - 1
15   Display names[index]
16 End For
17
18 // Sortujemy imiona
19 Call bubbleSort(names, SIZE)
20
21 // Wstawiamy pustą linię
22 Display
23
24 // Wyświetlamy posortowaną tablicę
25 Display "Posortowane imiona:"
26 For index = 0 To SIZE - 1

```

```

27     Display names[index]
28 End For
29 End Module
30
31 // Moduł bubbleSort przyjmuje jako argumenty tablicę ciągów znaków
32 // i rozmiar tej tablicy. Gdy moduł zakończy działanie,
33 // wartości zapisane w tablicy będą posortowane w kolejności
34 // alfabetycznej
35 Module bubbleSort(String Ref array[], Integer arraySize)
36     // Zmienna maxElement będzie zawierała indeks ostatniego elementu tablicy,
37     // który ma zostać porównany
38     Declare Integer maxElement
39
40     // Zmienna index służy jako licznik
41     // w pętli wewnętrznej
42     Declare Integer index
43
44     // Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
45     // ostatniemu indeksowi tablicy, który ma zostać porównany w danym
46     // przejściu. Na samym poczatku zmienna maxElement będzie równa
47     // indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
48     // będzie ona dekrementowana o 1
49     For maxElement = arraySize - 1 To 0 Step -1
50
51         // Pętla wewnętrzna śledzi tablicę i porównuje
52         // sąsiadujące ze sobą elementy. Porównywane są
53         // elementy o indeksach od 0 do maxElement.
54         // Jeżeli kolejność elementów jest nieprawidłowa,
55         // zostają one zamienione miejscami
56         For index = 0 To maxElement - 1
57
58             // Porównujemy dwa sąsiadujące elementy
59             // i jeśli trzeba, zamieniamy je miejscami
60             If array[index] > array[index + 1] Then
61                 Call swap(array[index], array[index + 1])
62             End If
63         End For
64     End For
65 End Module
66
67 // Moduł swap przyjmuje dwa argumenty typu String
68 // i zamienia ich wartości
69 Module swap(String Ref a, String Ref b)
70     // Tymczasowa zmienna lokalna
71     Declare String temp
72
73     // Zamieniamy wartości w zmiennych a i b
74     Set temp = a
75     Set a = b
76     Set b = temp
77 End Module

```

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Pierwotna kolejność:

Dawid  
Anna  
Maria  
Beata  
Jurek  
Damian

```
Posortowane imiona:
Anna
Beata
Damian
Dawid
Jurek
Maria
```



**WSKAZÓWKA:** Wszystkie przedstawione w tym rozdziale algorytmy sortowania możesz wykorzystać do sortowania ciągów znaków pod warunkiem, że będziesz pisać program w języku, który umożliwia porównywanie ciągów znaków.

## Sortowanie w kolejności malejącej

Algorytm sortowania bąbelkowego można w bardzo prosty sposób zmodyfikować tak, aby porządkował elementy w kolejności malejącej — od największej wartości do najmniejszej. Na listingu 9.4 przedstawiłem zmodyfikowaną wersję programu z listingu 9.3. Program sortuje tablicę names w odwróconym porządku alfabetycznym. Jedyna modyfikacja programu ma miejsce w linii 60.: zmieniłem instrukcję porównania, aby sprawdzała, czy element array[index] jest mniejszy od elementu array[index + 1].

**Listing 9.4**



```
1 Module main()
2   // Stała równa rozmiarowi tablicy
3   Constant Integer SIZE = 6
4
5   // Tablica ciągów znaków
6   Declare String names[SIZE] = "Dawid", "Anna", "Maria",
7     "Beata", "Jurek", "Damian"
8
9   // Zmienna licznikowa
10  Declare Integer index
11
12  // Wyświetlamy elementy w pierwotnej kolejności
13  Display "Pierwotna kolejność:"
14  For index = 0 To SIZE - 1
15    Display names[index]
16  End For
17
18  // Sortujemy imiona
19  Call bubbleSort(names, SIZE)
20
21  // Wstawiamy pustą linię
22  Display
23
24  // Wyświetlamy posortowaną tablicę
25  Display "Imiona posortowane w odwrotnej kolejności:"
26  For index = 0 To SIZE - 1
27    Display names[index]
28  End For
29 End Module
30
```

```

31 // Moduł bubbleSort przyjmuje jako argumenty tablicę ciągów znaków
32 // i rozmiar tej tablicy. Gdy moduł zakończy działanie,
33 // wartości zapisane w tablicy będą posortowane w kolejności
34 // alfabetycznej
35 Module bubbleSort(String Ref array[], Integer arraySize)
36     // Zmienna maxElement będzie zawierała indeks ostatniego elementu tablicy,
37     // który ma zostać porównany
38     Declare Integer maxElement
39
40     // Zmienna index służy jako licznik
41     // w pętli wewnętrznej
42     Declare Integer index
43
44     // Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
45     // ostatniemu indeksowi tablicy, który ma zostać porównany w danym
46     // przejściu. Na samym początku zmienna maxElement będzie równa
47     // indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
48     // będzie ona dekrementowana o 1
49     For maxElement = arraySize - 1 To 0 Step -1
50
51         // Pętla wewnętrzna śledzi tablicę i porównuje
52         // sąsiadujące ze sobą elementy. Porównywane są
53         // elementy o indeksach od 0 do maxElement.
54         // Jeżeli kolejność elementów jest nieprawidłowa,
55         // zostają one zamienione miejscami
56         For index = 0 To maxElement - 1
57
58             // Porównujemy dwa sąsiadujące elementy
59             // i jeśli trzeba, zamieniamy je miejscami
60             If array[index] < array[index + 1] Then
61                 Call swap(array[index], array[index + 1])
62             End If
63         End For
64     End For
65 End Module
66
67 // Moduł swap przyjmuje dwa argumenty typu String
68 // i zamienia ich wartości
69 Module swap(String Ref a, String Ref b)
70     // Tymczasowa zmienienna lokalna
71     Declare String temp
72
73     // Zamieniamy wartości w zmiennych a i b
74     Set temp = a
75     Set a = b
76     Set b = temp
77 End Module

```

### **Wynik działania programu**

Pierwotna kolejność:

Dawid  
Anna  
Maria  
Beata  
Jurek  
Damian

Imiona posortowane w odwrotnej kolejności:

Maria  
Jurek

Dawid  
Damian  
Beata  
Anna

## 9.2

**Algorytm sortowania przez wybieranie**

**WYJAŚNIENIE:** Algorytm sortowania przez wybieranie jest znacznie bardziej wydajny od algorytmu sortowania bąbelkowego. W algorytmie sortowania przez wybieranie każdy element jest od razu przemieszczany w tablicy na swoje docelowe miejsce.

Algorytm sortowania bąbelkowego jest bardzo prosty, ale także niezbyt wydajny, ponieważ wartości w tablicy przemieszczają się na prawidłowe miejsce pojedynczo. **Algorytm sortowania przez wybieranie** wykonuje zazwyczaj znacznie mniej operacji zamiany elementów, ponieważ przemieszcza je od razu na docelowe miejsce. Algorytm sortowania przez wybieranie działa w następujący sposób: odnajdujemy najmniejszą wartość w tablicy i przemieszczamy ją do elementu o indeksie 0, następnie znajdujemy kolejną najmniejszą wartość i przemieszczamy ją do elementu o indeksie 1 itd., aż do momentu, gdy wszystkie elementy tablicy znajdą się w określonej kolejności. Spójrzmy, jak działa algorytm sortowania przez wybieranie na przykładzie tablicy przedstawionej na rysunku 9.11.

5	7	2	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Rysunek 9.11. Wartości zapisane w tablicy

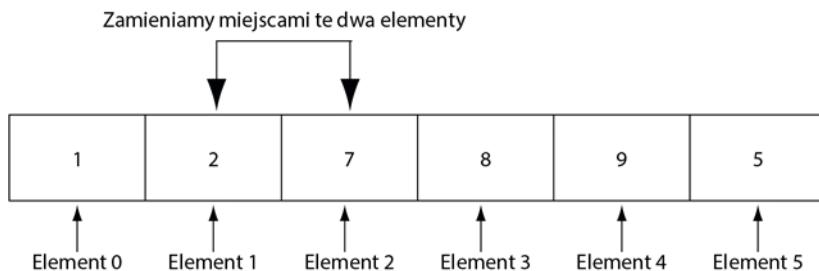
Algorytm śledzi całą tablicę, począwszy od elementu o indeksie 0, i odnajduje w niej element o najmniejszej wartości. Następnie element ten jest zamieniany miejscami z elementem o indeksie 0. W naszym przykładzie zamienione miejscami zostaną: wartość 1 zapisana w elemencie o indeksie 5 i wartość 5 zapisana w elemencie o indeksie 0. Po zamianie tablica będzie wyglądała jak na rysunku 9.12.

Zamieniamy miejscami te dwa elementy

1	7	2	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

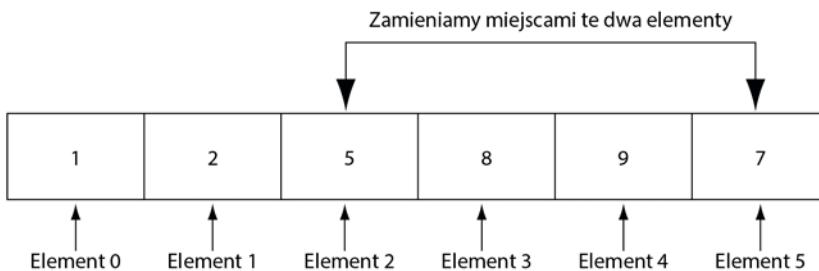
Rysunek 9.12. Tablica po pierwszej zamianie

Następnie algorytm powtarza tę operację, ale ponieważ element o indeksie 0 jest już na prawidłowej pozycji, można go tym razem pominąć. Algorytm prześledzi teraz tablicę od elementu o indeksie 1. W naszym przykładzie element o indeksie 2 zostanie zamieniony z elementem o indeksie 1. Po tej operacji tablica będzie wyglądała jak na rysunku 9.13.



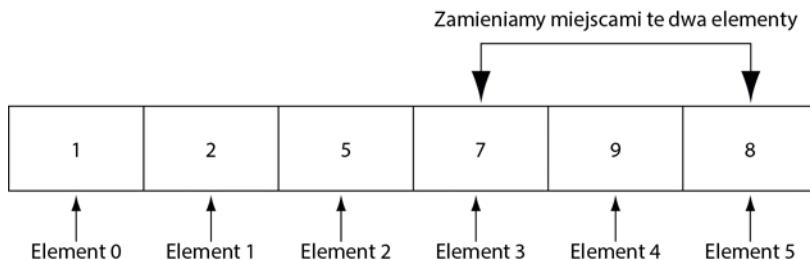
Rysunek 9.13. Tablica po drugiej zamianie

Ponownie powtarzamy całą operację, ale tym razem zaczynamy od elementu o indeksie 2. Okazuje się, że najmniejszą wartość ma element o indeksie 5. Zamieniamy go miejscami z elementem o indeksie 2, co spowoduje, że tablica będzie wyglądała jak na rysunku 9.14.



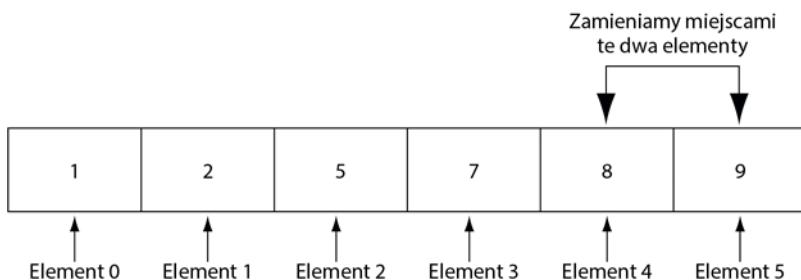
Rysunek 9.14. Tablica po trzeciej zamianie

Następnie śledzimy tablicę, poczawszy od elementu o indeksie 3, który zamieniamy z elementem o indeksie 5, dzięki czemu tablica będzie wyglądała jak na rysunku 9.15.



Rysunek 9.15. Tablica po czwartej zamianie

Na tym etapie pozostały nam do posortowania tylko dwa elementy. Algorytm rozpoznaje, że wartość elementu o indeksie 5 jest mniejsza od wartości elementu o indeksie 4, więc zamienia je miejscami. Tym samym dochodzimy do ostatecznej wersji tablicy, przedstawionej na rysunku 9.16.



Rysunek 9.16. Tablica po piątej zamianie

Na rysunku 9.17 przedstawiłem schemat blokowy modułu, w którym zawarty jest algorytm sortowania przez wybieranie. Moduł przyjmuje argumenty w postaci tablicy liczb typu Integer (przekazywanej przez referencję) i zmiennej typu Integer wskazującej rozmiar tablicy. Gdy moduł zakończy działanie, tablica będzie uporządkowana w kolejności rosnącej. Na listingu 9.5 przedstawiłem pseudokod modułu `selectionSort`.

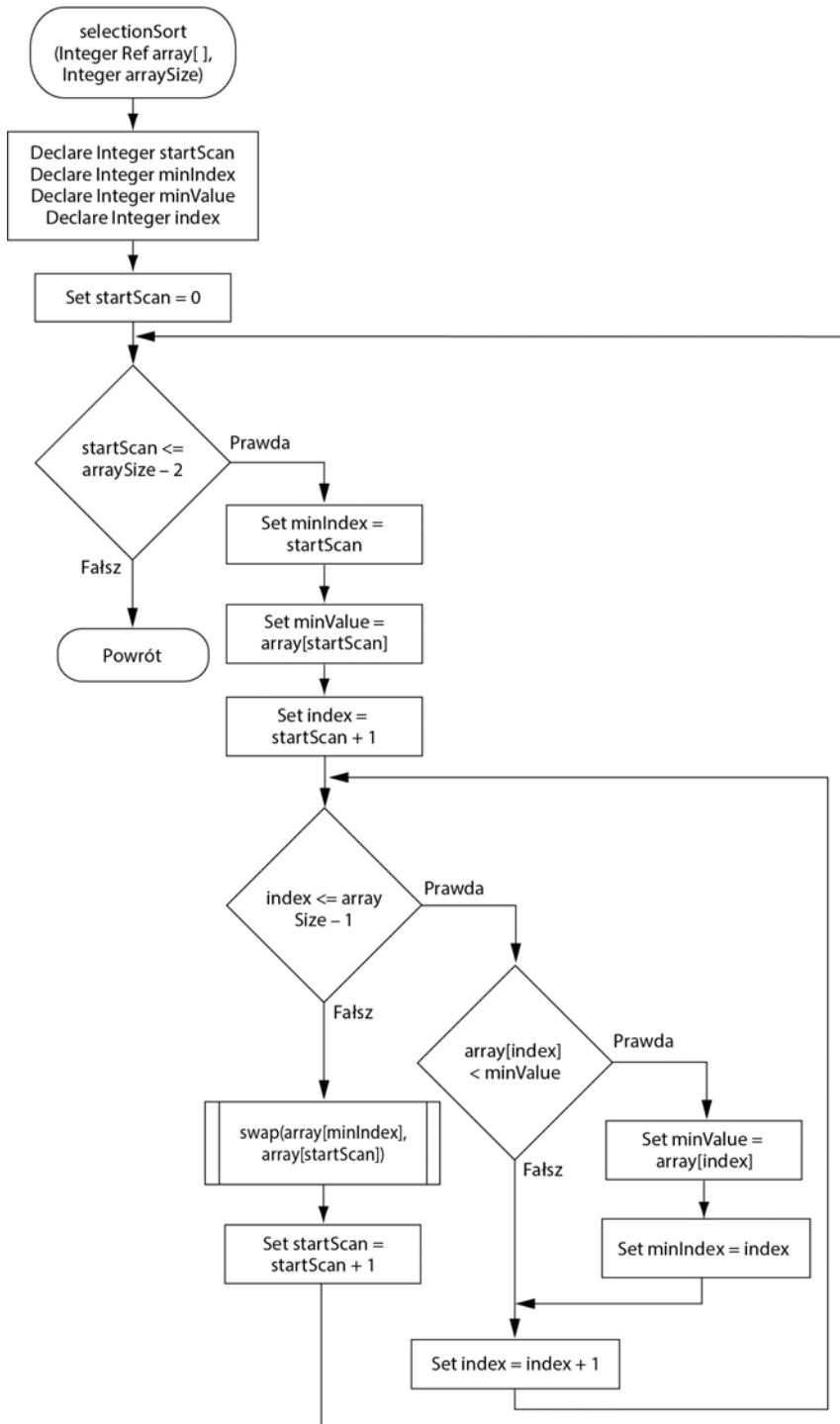
### Listing 9.5



```

1 Module main()
2   // Stała równa rozmiarowi tablicy
3   Constant Integer SIZE = 6
4
5   // Tablica liczb typu Integer
6   Declare Integer numbers[SIZE] = 4, 6, 1, 3, 5, 2
7
8   // Licznik pętli
9   Declare Integer index
10
11  // Wyświetlamy elementy tablicy w pierwotnej kolejności
12  Display "Pierwotna kolejność:"
13  For index = 0 To SIZE - 1
14    Display numbers[index]
15  End For
16
17  // Sortujemy liczby
18  Call selectionSort(numbers, SIZE)
19
20  // Wstawiamy pustą linię
21  Display
22
23  // Wyświetlamy posortowaną tablicę
24  Display "Posortowane liczby:"
25  For index = 0 To SIZE - 1
26    Display numbers[index]
27  End For
28 End Module
29

```



Rysunek 9.17. Schemat blokowy modułu selectionSort

```

30 // Moduł selectionSort przyjmuje jako argumenty tablicę liczb Integer
31 // i liczbę określającą rozmiar tablicy. Gdy moduł zakończy działanie,
32 // wartości zapisane w tablicy będą posortowane w kolejności
33 // rosnącej
34 Module selectionSort(Integer Ref array[], Integer arraySize)
35     // W zmiennej startScan zapiszemy początkowy indeks
36     Declare Integer startScan
37
38     // W zmiennej minIndex zapiszemy indeks elementu o najmniejszej wartości
39     // znajdującego się w przeszukiwanym fragmencie tablicy
40     Declare Integer minIndex
41
42     // W zmiennej minValue zapiszemy wartość najmniejszego elementu
43     // w przeszukiwanym fragmencie tablicy
44     Declare Integer minValue
45
46     // Zmienna index to licznik, w którym zapiszemy indeks
47     Declare Integer index
48
49     // Pętla zewnętrzna wykonuje iterację dla każdego elementu tablicy
50     // z wyjątkiem ostatniego. Zmienna startScan oznacza indeks,
51     // od którego należy rozpocząć przeszukiwanie
52     For startScan = 0 To arraySize - 2
53
54         // Na początku zakładamy, że to pierwszy element
55         // ma najniższą wartość
56         Set minIndex = startScan
57         Set minValue = array[startScan]
58
59         // Przeszukujemy tablicę, zaczynając od drugiego elementu
60         // w przeszukiwanym obszarze. Poszukujemy w tym obszarze
61         // elementu o najmniejszej wartości
62         For index = startScan + 1 To arraySize - 1
63             If array[index] < minValue Then
64                 Set minValue = array[index]
65                 Set minIndex = index
66             End If
67         End For
68
69         // Zamieniamy miejscami element, w którym jest zapisana najmniejsza wartość,
70         // i pierwszy element w przeszukiwanym obszarze
71         Call swap(array[minIndex], array[startScan])
72     End For
73 End Module
74
75 // Moduł swap przyjmuje dwa argumenty typu Integer
76 // i zamienia ich wartości
77 Module swap(Integer Ref a, Integer Ref b)
78     // Tymczasowa zmienna lokalna
79     Declare Integer temp
80
81     // Zamieniamy wartości w zmiennych a i b
82     Set temp = a
83     Set a = b
84     Set b = temp
85 End Module

```

**Wynik działania programu**

Pierwotna kolejność:

4  
6  
1

```

3
5
2

```

Posortowane liczby:

```

1
2
3
4
5
6

```



**UWAGA:** Zamieniając w następujący sposób operator „mniejsze niż” na operator „większe niż” w linii 63., można zmodyfikować moduł `selectionSort`, aby porządkował elementy w kolejności malejącej:

```
If array[indeks] > maxValue Then
```

Zwrót uwagę, że zmieniłem także nazwę zmiennej `minValue` na `maxValue`, która w przypadku sortowania w kolejności malejącej wydaje się bardziej stosowna. Nazwę tę musisz także zmienić w kilku innych miejscach w module.

## 9.3

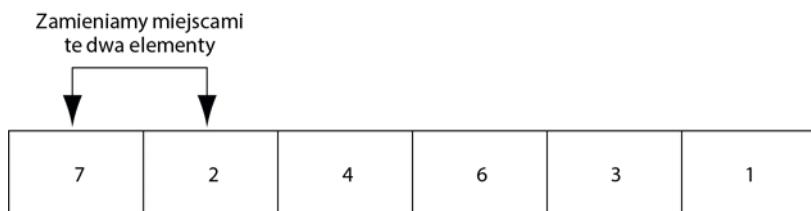
## Algorytm sortowania przez wstawianie

**WYJAŚNIENIE:** Algorytm sortowania przez wstawianie także jest bardziej wydajny niż algorytm sortowania bąbelkowego. Sortuje on dwa pierwsze elementy tablicy, które utworzą tym samym posortowany fragment tablicy. Następnie po kolejno wstawia pozostałe elementy tablicy w odpowiednie miejsca posortowanego fragmentu.

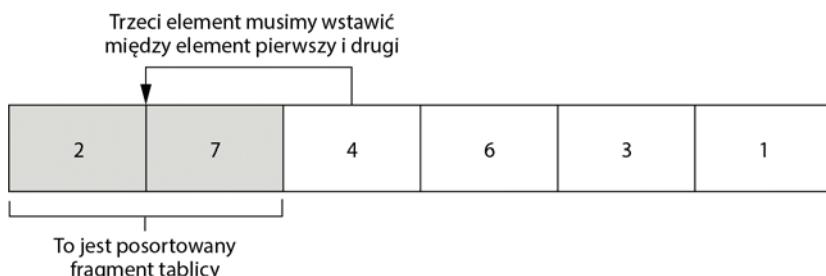
Algorytm sortowania przez wstawianie to kolejny przykład algorytmu, który charakteryzuje się większą wydajnością od algorytmu sortowania bąbelkowego. Sortowanie przez wstawianie rozpoczynamy od uporządkowania dwóch pierwszych elementów tablicy. Elementy te porównujemy i w razie konieczności są zamieniane miejscami. Tworzymy w ten sposób posortowany fragment tablicy.

Następnym zadaniem jest dołączenie do posortowanego fragmentu tablicy trzeciego elementu. Robimy to, wstawiając trzeci element tablicy na odpowiednie miejsce względem dwóch pierwszych elementów. Jeżeli trzeba, algorytm przesuwa pozostałe elementy. Po wstawieniu trzeciego elementu w prawidłowe miejsce tablicy (względem pozostałych dwóch elementów) będą one tworzyły nowy, posortowany fragment tablicy.

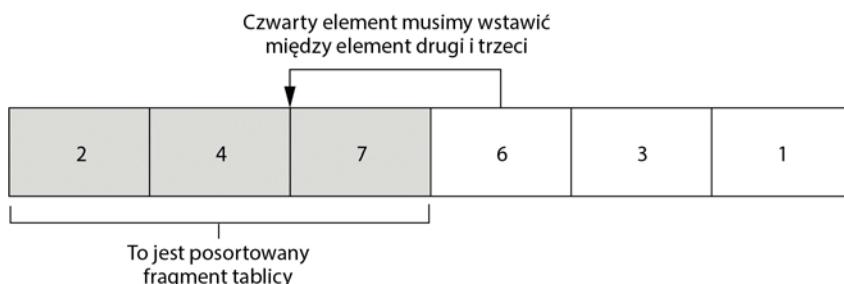
Działanie to powtarzamy z czwartym i każdym kolejnym elementem, aż do momentu, gdy wszystkie elementy znajdą się na odpowiednich miejscach. Przyjrzyjmy się przykładowi. Założymy, że zaczynamy pracę z tablicą liczb typu `Integer`, przedstawioną na rysunku 9.18. Jak widać, wartości pierwszego i drugiego elementu nie są ułożone w kolejności rosnącej, więc zamieniamy je miejscami.

**Rysunek 9.18.** Tablica nieuporządkowana

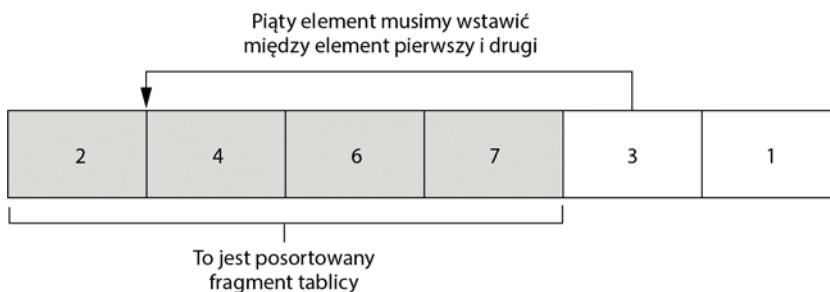
Po zamianie dwóch pierwszych elementów tablicy utworzą one nowy, posortowany fragment tablicy. Następnym zadaniem jest przesunięcie trzeciego elementu tak, aby był on uporządkowany względem dwóch pierwszych. Jak widać na rysunku 9.19, trzeci element należy wstawić między pierwszy i drugi.

**Rysunek 9.19.** Przemieszczamy trzeci element

Po wstawieniu trzeciego elementu w odpowiednie miejsce pierwsze trzy elementy utworzą nowy, posortowany fragment tablicy. Następnym krokiem jest przeniesienie czwartego elementu tak, aby znajdował się na odpowiednim miejscu względem trzech pierwszych. Jak widać na rysunku 9.20, czwarty element należy wstawić między drugi i trzeci.

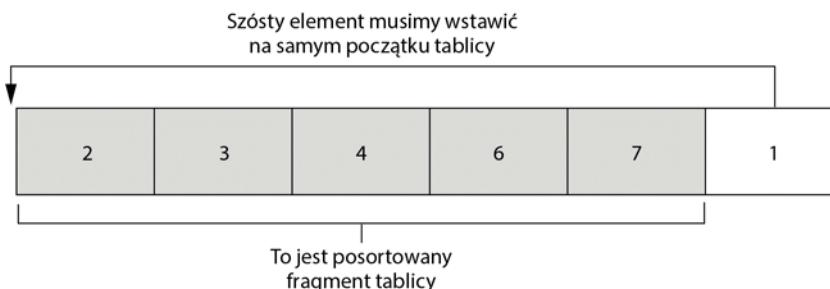
**Rysunek 9.20.** Przemieszczamy czwarty element

Po wstawieniu czwartego elementu w odpowiednie miejsce pierwsze cztery elementy utworzą nowy, posortowany fragment tablicy. Następnym krokiem jest przeniesienie piątego elementu tak, aby znajdował się na odpowiednim miejscu względem czterech pierwszych. Jak widać na rysunku 9.21, piąty element należy wstawić między pierwszy i drugi.



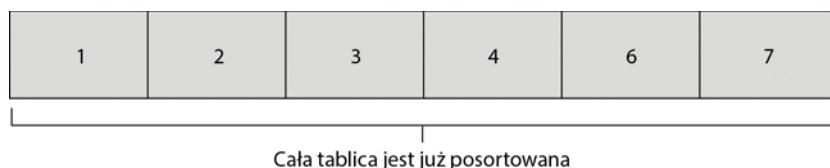
**Rysunek 9.21.** Przemieszczamy piąty element

Po wstawieniu piątego elementu w odpowiednie miejsce pierwszych pięciu elementów utworzy nowy, posortowany fragment tablicy. Następnym krokiem jest przeniesienie szóstego elementu tak, aby znajdował się na odpowiednim miejscu względem pięciu pierwszych. Jak widać na rysunku 9.22, szósty element należy wstawić na samym początku tablicy.



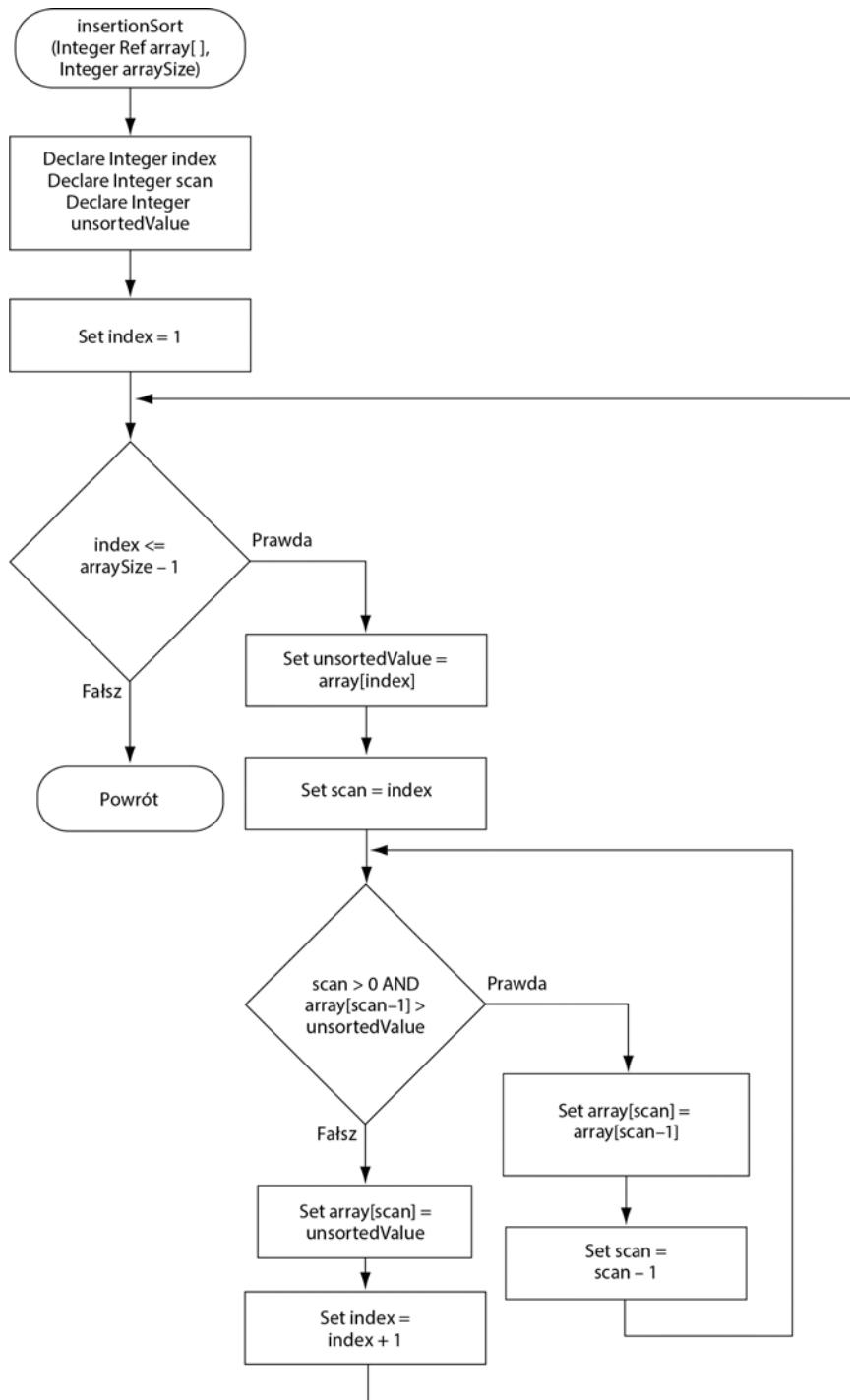
**Rysunek 9.22.** Przemieszczamy szósty element

Szósty element jest zarazem ostatnim elementem tablicy. Po wstawieniu go na odpowiednie miejsce otrzymamy posortowaną tablicę. Przedstawiłem ją na rysunku 9.23.



**Rysunek 9.23.** Wszystkie elementy są już na właściwych miejscach

Na rysunku 9.24 widoczny jest schemat blokowy modułu, w którym zawarty jest algorytm sortowania przez wstawianie. Moduł przyjmuje argumenty w postaci tablicy liczb typu Integer (przekazywanej przez referencję) i zmiennej typu Integer wskazującej rozmiar tablicy. Gdy moduł zakończy działanie, tablica będzie uporządkowana w kolejności rosnącej. Na listingu 9.6 przedstawiłem pseudokod modułu `insertionSort`.



Rysunek 9.24. Schemat blokowy modułu insertionSort

**Listing 9.6**

```

1 Module main()
2 // Stała równa rozmiarowi tablicy
3 Constant Integer SIZE = 6
4
5 // Tablica liczb typu Integer
6 Declare Integer numbers[SIZE] = 4, 6, 1, 3, 5, 2
7
8 // Licznik pętli
9 Declare Integer index
10
11 // Wyświetlamy elementy tablicy w pierwotnej kolejności
12 Display "Pierwotna kolejność:"
13 For index = 0 To SIZE - 1
14   Display numbers[index]
15 End For
16
17 // Sortujemy liczby
18 Call insertionSort(numbers, SIZE)
19
20 // Wstawiamy pustą linię
21 Display
22
23 // Wyświetlamy posortowaną tablicę
24 Display "Posortowane liczby:"
25 For index = 0 To SIZE - 1
26   Display numbers[index]
27 End For
28 End Module
29
30 // Moduł insertionSort przyjmuje jako argumenty tablicę liczb Integer
31 // i liczbę określającą rozmiar tablicy. Gdy moduł zakończy działanie,
32 // wartości zapisane w tablicy będą posortowane w kolejności
33 // rosnącej
34 Module insertionSort(Integer Ref array[], Integer arraySize)
35   // Zmienna licznikowa
36   Declare Integer index
37
38   // Zmienna, za pomocą której będziemy śledzili tablicę
39   Declare Integer scan
40
41   // Zmienna, w której zapiszemy pierwszą nieposortowaną wartość
42   Declare Integer unsortedValue
43
44   // W zmiennej index będą zapisywane w pętli kolejne indeksy,
45   // poczawszy od indeksu numer 1, dlatego,
46   // że element o indeksie 0 uważamy za już posortowany
47   For index = 1 To arraySize - 1
48
49     // Pierwszy element występujący zaraz za posortowanym fragmentem
50     // tablicy to array[index]. W zmiennej unsortedValue zapisujemy
51     // wartość przypisaną do tego elementu
52     Set unsortedValue = array[index]
53
54     // Śledzenie zaczynamy od indeksu pierwszego elementu,
55     // zaraz za posortowanym fragmentem tablicy
56     Set scan = index
57
58     // Przenosimy pierwszy element spoza posortowanego fragmentu

```

```

59      // w odpowiednie miejsce posortowanego fragmentu
60      While scan > 0 AND array[scan-1] > unsortedValue
61          Set array[scan] = array[scan-1]
62          Set scan = scan - 1
63      End While
64
65      // Wstawiamy wartość nieposortowaną we właściwe miejsce
66      // posortowanego fragmentu tablicy
67      Set array[scan] = unsortedValue
68  End For
69 End Module

```

**Wynik działania programu**

Pierwotna kolejność:

4  
6  
1  
3  
5  
2

Posortowane liczby:

1  
2  
3  
4  
5  
6**Punkt kontrolny**

- 9.1. Który z opisanych algorytmów sortowania polega na wielokrotnym śledzeniu całej tablicy i przesuwaniu kolejnych największych wartości w kierunku końca tablicy?
- 9.2. Jeden z algorytmów sortowania, które opisałem w tym rozdziale, działa w następujący sposób: sortowanie rozpoczyna się od uporządkowania dwóch pierwszych elementów tablicy, które tworzą posortowany fragment tablicy. Następnie na właściwe miejsce względem dwóch pierwszych elementów przesuwany jest trzeci element. Po wstawieniu trzeciego elementu otrzymujemy posortowany fragment tablicy składający się z trzech elementów. Operacja ta powtarzana jest w przypadku czwartego i dalszych elementów, aż do momentu, gdy cała tablica zostanie uporządkowana. Jak nazywa się ten algorytm?
- 9.3. Jeden z algorytmów sortowania, które opisałem w tym rozdziale, działa w następujący sposób: odnajdywany jest element o najmniejszej wartości, a następnie jest przenoszony do elementu o indeksie 0. W następnym kroku odnajdywany jest kolejny element o najmniejszej wartości i jest przenoszony do elementu o indeksie 1. Operacja ta jest powtarzana aż do momentu, gdy wszystkie elementy znajdą się na właściwych miejscach tablicy. Jak nazywa się ten algorytm?

**9.4**

## Algorytm wyszukiwania binarnego

**WYJAŚNIENIE:** Algorytm wyszukiwania binarnego jest znacznie bardziej wydajny od algorytmu wyszukiwania sekwencyjnego, omówionego w rozdziale 8. Algorytm wyszukiwania binarnego odnajduje dany element w tablicy poprzez cykliczne dzielenie jej na pół. Po podzieleniu tablicy z dalszego wyszukiwania wykluczana jest ta połówka, która nie zawiera szukanego elementu.

W rozdziale 8. umówilem algorytm wyszukiwania sekwencyjnego, który przeszukuje całą tablicę, począwszy od pierwszego elementu. Każdy element porównywany jest z poszukiwaną wartością, a wyszukiwanie kończy się w momencie, gdy element zostanie odnaleziony lub gdy algorytm dotrze do końca tablicy. Jeśli szukana wartość nie znajduje się w danej tablicy, tablica zostanie przeszukana w całości, ale wyszukiwanie zakończy się niepowodzeniem.

Zaletą tego algorytmu jest jego prostota — łatwo jest go zrozumieć i zaimplementować w programie. Ponadto algorytm ten nie wymaga, aby elementy tablicy znajdowały się w określonym porządku. Wadą jest niestety jego słaba wydajność. Jeżeli tablica zawiera 20000 elementów, a szukana wartość znajduje się w ostatnim elemencie, algorytm będzie musiał przeszukać wszystkie 20000 elementów.

W typowym przypadku prawdopodobieństwo tego, że szukany element znajduje się bliżej początku tablicy, jest takie samo jak prawdopodobieństwo, że element ten znajduje się bliżej końca tablicy. Jeżeli tablica składa się z  $n$  elementów, algorytm odnajdzie szukany element za  $n/2$  razem. Jeśli więc tablica zawiera 50000 elementów, algorytm wyszukiwania sekwencyjnego będzie musiał w typowym przypadku porównać 25000 elementów. Zakładam oczywiście, że poszukiwany element na pewno znajduje się w danej tablicy ( $n/2$  to średnia liczba porównań, a maksymalna liczba porównań jest zawsze równa  $n$ ).

Jeżeli szukanej wartości nie ma w danej tablicy, algorytm i tak musi porównać każdy element. W miarę wzrostu liczby nieudanych prób wyszukania wartości rośnie także średnia liczba operacji porównania. Algorytm wyszukiwania sekwencyjnego przydaje się w przypadku niewielkich tablic, jednak gdy mamy do czynienia z większymi tablicami i dodatkowo ważna jest szybkość działania programu, należy skorzystać z innego rozwiązania.

**Wyszukiwanie binarne** to bardzo sprytny algorytm, który jest znacznie bardziej wydajny od algorytmu wyszukiwania sekwencyjnego. Wymaga on jedynie, aby elementy tablicy były uporządkowane w kolejności rosnącej. Zamiast rozpoczynać wyszukiwanie od pierwszego elementu algorytm ten najpierw sprawdza środkowy element tablicy. Jeżeli element ten jest równy szukanej wartości, wyszukiwanie się kończy. W przeciwnym razie wartość środkowego elementu jest albo większa, albo mniejsza od wartości szukanej. Jeśli jest większa, wtedy jasne jest, że szukana wartość znajduje się gdzieś w pierwszej połówce tablicy (oczywiście pod warunkiem, że znajduje się ona w danej tablicy). Jeśli wartość jest mniejsza, wtedy szukana wartość znajduje

się w drugiej połówce tablicy (o ile w ogóle się w niej znajduje). W każdym z tych przypadków z dalszych poszukiwań wykluczana jest połowa tablicy.

Jeżeli szukana wartość nie znajduje się w środkowym elemencie, powtarzamy całą procedurę, ale sprawdzamy tylko tę połówkę tablicy, która potencjalnie może zawierać szukany element. Przykładowo, jeżeli do przeszukania została nam druga połówka tablicy, zaczniemy od sprawdzenia wartości środkowego elementu tej połówki. Jeżeli nie jest on równy szukanej wartości, zawiżamy poszukiwania do określonej części tablicy, znajdującej się przed albo za środkowym elementem danej połówki. Operację tę powtarzamy aż do momentu, gdy odnajdziemy szukaną wartość lub zabraknie elementów do sprawdzenia.

Na rysunku 9.25 przedstawiłem schemat blokowy funkcji, która korzysta z algorytmu wyszukiwania binarnego. Przyjmuje ona jako argumenty tablicę liczb typu Integer, liczbę Integer równą szukanej wartości oraz liczbę Integer wskazującą rozmiar tablicy. Jeżeli funkcja odnajdzie w tablicy szukaną wartość, zwraca indeks elementu, w którym ta wartość się znajduje. Jeżeli funkcja nie odnajdzie szukanej wartości, zwraca wartość -1. Na listingu 9.7 widoczny jest pseudokod funkcji `binarySearch`. Zauważ, że na listingu znajduje się tylko kod funkcji i nie jest on pełnym programem.

### **Listing 9.7. Funkcja `binarySearch` (tylko część programu)**



```

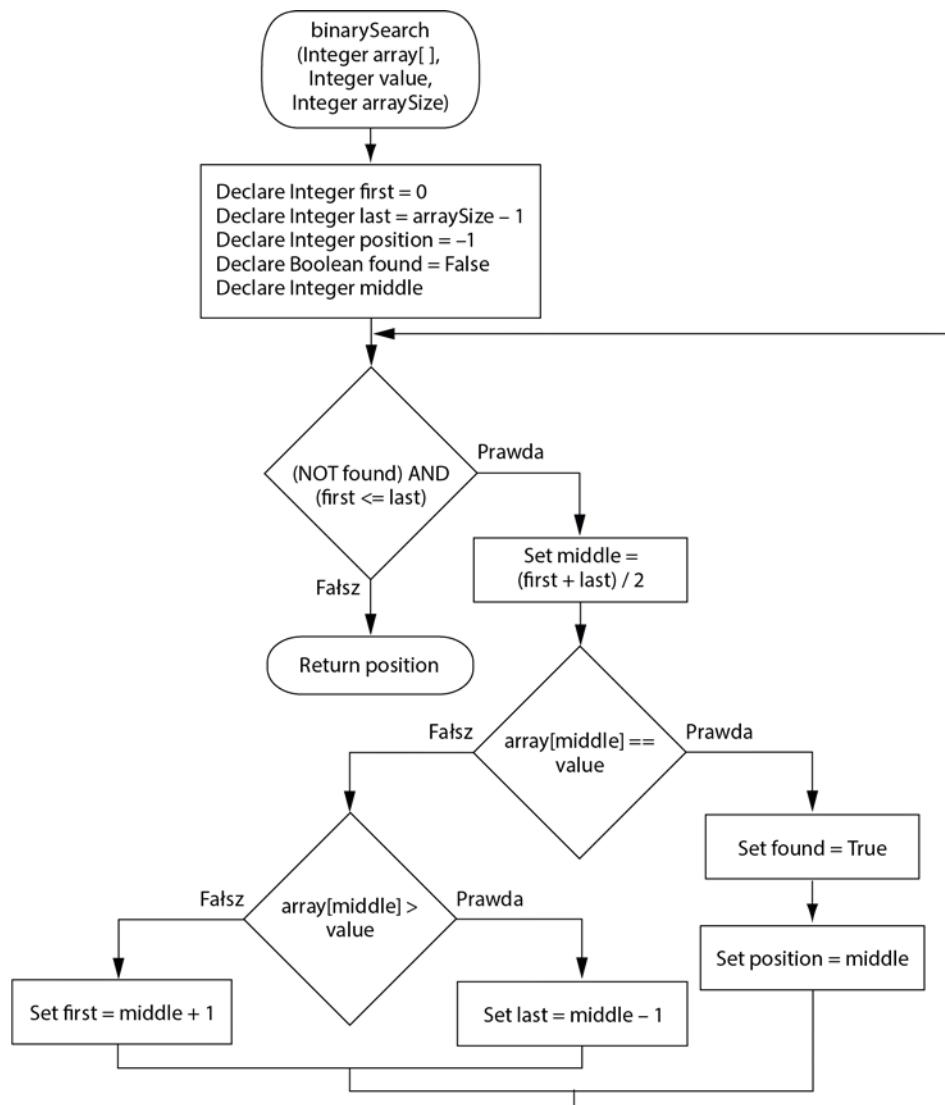
1 // Funkcja binarySearch przyjmuje jako argumenty tablicę liczb
2 // Integer, szukaną wartość oraz rozmiar tablicy.
3 // Gdy wartość zostanie odnaleziona w tablicy, funkcja zwraca
4 // indeks tego elementu. W przeciwnym razie funkcja zwraca -1,
5 // co oznacza, że wskazanej wartości nie udało się odnaleźć w tablicy
6 Function Integer binarySearch(Integer array[], Integer value,
7                               Integer arraySize)
8     // Zmienna, w której zapiszemy indeks pierwszego elementu
9     Declare Integer first = 0
10
11    // Zmienna, w której zapiszemy indeks ostatniego elementu
12    Declare Integer last = arraySize - 1
13
14    // Indeks elementu, w którym znajduje się szukana wartość
15    Declare Integer position = -1
16
17    // Flaga
18    Declare Boolean found = False
19
20    // Zmienna, w której zapiszemy indeks środkowego elementu
21    Declare Integer middle
22
23    While (NOT found) AND (first <= last)
24        // Obliczamy indeks środkowego elementu
25        Set middle = (first + last) / 2
26
27        // Sprawdzamy, czy szukana wartość jest zapisana w środkowym elemencie...
28        If array[middle] == value Then
29            Set found = True
30            Set position = middle
31
32        // ...jeśli nie, to wartość może się znajdować w pierwszej połówce...
33        Else If array[middle] > value Then

```

```

34     Set last = middle - 1
35
36 // ...jeśli nie, to wartość może się znajdować w drugiej połówce tablicy...
37 Else
38     Set first = middle + 1
39 End If
40 End While
41
42 // Zwracamy indeks, pod którym znajduje się szukana wartość, lub -1,
43 // jeśli wartości nie udało się odszukać
44 Return position
45 End Function

```



**Rysunek 9.25.** Schemat blokowy funkcji binarySearch

W algorytmie do zaznaczania indeksów w tablicy korzystam z trzech zmiennych: `first`, `last` i `middle`. Zmienne `first` i `last` określają przeszukiwany obecnie fragment tablicy. Inicjalizuję je wartościami równymi pierwszemu i ostatniemu indeksowi tablicy. Następnie obliczam indeks środkowego elementu, znajdującego się między indeksami `first` i `last`, i zapisuję go w zmiennej `middle`. Jeżeli element o indeksie równym `middle` nie jest równy szukanej wartości, modyfikuję wartość zmiennych `first` i `last` tak, aby wskazywały pierwszy i ostatni indeks fragmentu tablicy, który będę przeszukiwał w kolejnej iteracji. Dzięki temu po każdej nieudanej próbie odnalezienia wartości dzielimy przeszukiwany fragment tablicy na pół.

## Wydajność algorytmu wyszukiwania binarnego

Algorytm wyszukiwania binarnego jest znacznie bardziej wydajny od algorytmu wyszukiwania sekwencyjnego. Po każdym porównaniu i nieudanej próbie odnalezienia szukanej wartości przeszukiwany obszar tablicy zwiększa się o połowę. Założymy, że mamy do czynienia z tablicą zawierającą 1000 elementów. Jeżeli algorytm za pierwszym podejściem nie odnajdzie szukanej wartości, w kolejnej próbie pozostanie do przeszukania tylko 500 elementów. Jeżeli algorytm nie odnajdzie szukanej wartości za drugim podejściem, liczba elementów do przeszukania zwiększy się do 250. Taka operacja będzie się powtarzała aż do momentu, gdy algorytm odnajdzie szukaną wartość lub okaże się, że nie ma jej w danej tablicy. W przypadku tablicy zawierającej 1000 elementów będzie to nie więcej niż 10 operacji porównania (w przypadku algorytmu wyszukiwania sekwencyjnego jest to średnio 500 operacji!).

### W centrum uwagi

#### Korzystanie z algorytmu wyszukiwania binarnego

Karolina jest dyrektorką szkoły gastronomicznej, w której zatrudnia sześciu nauczycieli. Poprosiła Cię o zaprojektowanie programu, za pomocą którego będzie mogła wyszukać numer telefonu do danego nauczyciela. Zdecydowałeś się skorzystać z dwóch tablic równoległych: jedna z nich będzie się nazywała `names` i będzie zawierała nazwiska nauczycieli, a druga będzie się nazywała `phones` i będzie zawierała numery telefonów nauczycieli. Oto algorytm programu:

1. Pobierz od użytkownika nazwisko nauczyciela.
2. Wyszukaj nazwisko w tablicy `names`.
3. Jeżeli nazwisko znajduje się w tablicy, zapisz indeks tego elementu. Posługując się tym indeksem, wyświetl numer telefonu z tablicy równoległej `phones`. Jeżeli nazwiska nie ma w tablicy, wyświetl komunikat informujący o tym.

Na listingu 9.8 znajduje się pseudokod programu. Zauważ, że elementy tablicy są już uporządkowane w kolejności alfabetycznej. To bardzo ważne, ponieważ do odszukania danego nazwiska w tablicy `names` użyję algorytmu wyszukiwania binarnego.



**Listing 9.8**

```

1 Module main()
2   // Stała równa rozmiarowi tablicy
3   Constant Integer SIZE = 6
4
5   // Tablica zawierająca nazwiska nauczycieli
6   // posortowana w kolejności alfabetycznej
7   Declare String names[SIZE] = "Haręda", "Hibesz",
8           "Hołdys", "Kraśniski",
9           "Lipińska", "Piątek"
10
11  // Tablica równoległa zawierająca numery telefonów
12  Declare String phones[SIZE] = "555-678-323", "555-019-459",
13      "555-997-674", "555-237-627",
14      "555-777-432", "555-171-996"
15
16  // Zmienna, w której zapiszemy szukane nazwisko
17  Declare String searchName
18
19  // Zmienna, w której zapiszemy indeks, pod którym znajduje się nazwisko
20  Declare Integer index
21
22  // Zmienna sterująca pętlą
23  Declare String again = "T"
24
25  While (again == "T" OR again == "t")
26    // Pobieramy szukane nazwisko
27    Display "Wprowadź nazwisko, które chcesz wyszukać."
28    Input searchName
29
30  // Szukamy wskazanego nazwiska
31  index = binarySearch(names, searchName, SIZE)
32
33  If index != -1 Then
34    // Wyświetlamy numer telefonu
35    Display "Numer telefonu: ", phones[index]
36  Else
37    // Nie znaleźliśmy nazwiska w tablicy
38    Display searchName, " nie został znaleziony."
39  End If
40
41  // Wyszukać ponownie?
42  Display "Czy chcesz ponownie wyszukać nazwisko? (T = Tak, N = Nie)"
43  Input again
44 End While
45
46 End Module
47
48 // Funkcja binarySearch przyjmuje jako argumenty tablicę ciągów znaków liczb
49 // Integer, szukaną wartość oraz rozmiar tablicy.
50 // Gdy wartość zostanie odnaleziona w tablicy, funkcja zwraca
51 // indeks tego elementu. W przeciwnym razie funkcja zwraca -1,
52 // co oznacza, że wskazanej wartości nie udało się odnaleźć w tablicy
53 Function Integer binarySearch(String array[], String value,
54                                Integer arraySize)
55   // Zmienna, w której zapiszemy indeks pierwszego elementu
56   Declare Integer first = 0
57

```

```

58 // Zmienna, w której zapiszemy indeks ostatniego elementu
59 Declare Integer last = arraySize - 1
60
61 // Indeks elementu, w którym znajduje się szukana wartość
62 Declare Integer position = -1
63
64 // Flaga
65 Declare Boolean found = False
66
67 // Zmienna, w której zapiszemy indeks środkowego elementu
68 Declare Integer middle
69
70 While (NOT found) AND (first <= last)
    // Obliczamy indeks środkowego elementu
    Set middle = (first + last) / 2
71
72 // Sprawdzamy, czy szukana wartość jest zapisana w środkowym elemencie...
73 If array[middle] == value Then
    Set found = True
    Set position = middle
74
75 // ...jeśli nie, to wartość może się znajdować w pierwszej połówce...
76 Else If array[middle] > value Then
77     Set last = middle - 1
78
79 // ...jeśli nie, to wartość może się znajdować w drugiej połówce tablicy...
80 Else
81     Set first = middle + 1
82 End If
83 End While
84
85 // Zwracamy indeks, pod którym znajduje się szukana wartość, lub -1,
86 // jeśli wartości nie udało się odszukać
87 Return position
88 End Function

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź nazwisko, które chcesz wyszukać.

**Lipińska [Enter]**

Numer telefonu: 555-777-432

Czy chcesz ponownie wyszukać nazwisko? (T = Tak, N = Nie)

**T [Enter]**

Wprowadź nazwisko, które chcesz wyszukać.

**Hibsz [Enter]**

Numer telefonu: 555-019-459

Czy chcesz ponownie wyszukać nazwisko? (T = Tak, N = Nie)

**T [Enter]**

Wprowadź nazwisko, które chcesz wyszukać.

**Latański [Enter]**

Latański nie został znaleziony.

Czy chcesz ponownie wyszukać nazwisko? (T = Tak, N = Nie)

**N [Enter]**



## Punkt kontrolny

- 9.4. Wyjaśnij, czym różni się algorytm wyszukiwania sekwencyjnego od algorytmu wyszukiwania binarnego.
- 9.5. Ile operacji porównania będzie musiał średnio wykonać algorytm wyszukiwania sekwencyjnego w przypadku tablicy zawierającej 1000 elementów (przy założeniu, że szukana wartość znajduje się w tablicy)?
- 9.6. Ile wynosi maksymalna liczba operacji porównania, jakie będzie musiał wykonać algorytm wyszukiwania binarnego w przypadku tablicy zawierającej 1000 elementów?

9.5

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

### Java

#### Sortowanie i wyszukiwanie tablic

##### Sortowanie bąbelkowe w języku Java

Na listingu 9.9 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Javie wersja pseudokodu z listingu 9.1. Zajmujemy się tutaj algorytmem sortowania bąbelkowego.

##### **Listing 9.9**

```

1 public class BubbleSortAlgorithm
2 {
3     // Uwaga: To nie jest kompletny program
4     //
5     // Metoda bubbleSort korzysta z algorytmu sortowania bąbelkowego
6     // w celu posortowania tablicy typu int
7     // Zwróć uwagę na poniższe:
8     // (1) Nie musimy przekazywać funkcji wielkości tablicy, ponieważ
9     // w Javie tablice mają pole length
10    // (2) Nie mamy też osobnej metody podmieniającej elementy,
11    // co wynika to z faktu, że Java nie pozwala na przekazywanie parametrów
12    // przez referencję, podmiana wykonywana jest zatem w ramach metody
13
14    public static void bubbleSort(int[] array)
15    {

```

```

16     int maxElement; // Wyznacza ostatni element do porównania
17     int index;      // Indeks elementu do porównania
18     int temp;        // Używana podczas podmiany elementów
19
20     // Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
21     // ostatniemu indeksowi tablicy, który ma zostać porównany w danym
22     // przejściu. Na samym początku zmienna maxElement będzie równa
23     // indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
24     // będzie ona dekrementowana o 1
25     for (maxElement = array.length - 1; maxElement >= 0; maxElement--)
26     {
27         // Pętla wewnętrzna śledzi tablicę i porównuje
28         // sąsiadujące ze sobą elementy. Porównywane są
29         // elementy o indeksach od 0 do maxElement.
30         // Jeżeli kolejność elementów jest nieprawidłowa,
31         // zostają one zamienione miejscami
32         for (index = 0; index <= maxElement - 1; index++)
33         {
34             // Porównujemy dwa sąsiadujące elementy
35             if (array[index] > array[index + 1])
36             {
37                 // i jeśli trzeba, zamieniamy je miejscami
38                 temp = array[index];
39                 array[index] = array[index + 1];
40                 array[index + 1] = temp;
41             }
42         }
43     }
44 }
45 }
```

### Sortowanie przez wybieranie w języku Java

Na listingu 9.10 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Javie wersja pseudokodu z listingu 9.5. Zajmujemy się tutaj algorytmem sortowania przez wybieranie.

#### **Listing 9.10**

```

1 public class SelectionSortAlgorithm
2 {
3     // Uwaga: To nie jest kompletny program
4     //
5     // Metoda selectionSort sortuje tablice liczb całkowitych
6     // za pomocą algorytmu sortowania przez wybieranie
7     // Tablica zostaje posortowana w kolejności malejącej
8     public static void selectionSort(int[] array)
9     {
10         int startScan; // Początkowa pozycja skanowania
11         int index;    // Przechowuje aktualny indeks
12         int minIndex; // Element o najmniejszej wartości w danym skanowaniu
13         int minValue; // Najmniejsza wartość znaleziona w tym skanowaniu
14
15         // Pętla zewnętrzna wykonuje iterację dla każdego elementu tablicy
16         // z wyjątkiem ostatniego. Zmienna startScan oznacza indeks,
17         // od którego należy rozpoczęć przeszukiwanie
```

```

18     for (startScan = 0; startScan < (array.length-1); startScan++)
19     {
20         // Na początku zakładamy, że to pierwszy element
21         // ma najniższą wartość
22         minIndex = startScan;
23         minValue = array[startScan];
24
25         // Przeszukujemy tablicę, zaczynając od drugiego elementu
26         // w przeszukiwanym obszarze. Poszukujemy w tym obszarze
27         // elementu o najmniejszej wartości
28         for(index = startScan + 1; index < array.length; index++)
29         {
30             if (array[index] < minValue)
31             {
32                 minValue = array[index];
33                 minIndex = index;
34             }
35         }
36
37         // Zamieniamy miejscami element, w którym jest zapisana najmniejsza wartość,
38         // i pierwszy element w przeszukiwanym obszarze
39         array[minIndex] = array[startScan];
40         array[startScan] = minValue;
41     }
42 }
43 }
```

### Sortowanie przez wstawianie w języku Java

Na listingu 9.11 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Javie wersja pseudokodu modułu `insertionSort` z listingu 9.6. Zajmujemy się tutaj algorytmem sortowania przez wstawianie.

#### **Listing 9.11**

```

1 public class InsertionSortAlgorithm
2 {
3     //Uwaga: To nie jest kompletny program
4     //
5     //Metoda insertionSort sortuje tablicę liczb całkowitych
6     //za pomocą algorytmu sortowania przez wstawianie
7     //Tablica zostaje posortowana w kolejności malejącej
8     public static void insertionSort(int[] array)
9     {
10         int unsortedValue; // Pierwsza nieposortowana wartość
11         int scan;           // Używana podczas skanowania tablicy
12
13         // W zmiennej index będą zapisywane w pętli kolejne indeksy,
14         // począwszy od indeksu numer 1, dlatego,
15         // że element o indeksie 0 uważamy za już posortowany
16         for (int index = 1; index < array.length; index++)
17         {
18             // Pierwszy element występujący zaraz za posortowanym fragmentem
19             // tablicy to array[index]. W zmiennej unsortedValue zapisujemy
20             // wartość przypisaną do tego elementu
```

```

21     unsortedValue = array[index];
22
23     // Śledzenie zaczynamy od indeksu pierwszego elementu,
24     // zaraz za posortowanym fragmentem tablicy
25     scan = index;
26
27     // Przenosimy pierwszy element spoza posortowanego fragmentu
28     // w odpowiednie miejsce posortowanego fragmentu
29     while (scan > 0 && array[scan-1] > unsortedValue)
30     {
31         array[scan] = array[scan - 1];
32         scan--;
33     }
34
35     // Wstawiamy wartość nieposortowaną we właściwe miejsce
36     // posortowanego fragmentu tablicy
37     array[scan] = unsortedValue;
38 }
39 }
40 }
```

### Wyszukiwanie binarne w języku Java

Na listingu 9.12 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Javie wersja pseudokodu modułu `binarySearch` z listingu 9.7. Zajmujemy się tutaj algorytmem wyszukiwania binarnego.

#### **Listing 9.12**

```

1 public class BinarySearchAlgorithm
2 {
3     // Uwaga: To nie jest kompletny program
4     //
5     // Metoda binarySearch wykonuje operację szukania binarnego
6     // w tablicy ciągów znaków. W tablicy poszukiwany jest ciąg znaków
7     // podany w parametrze value. Jeżeli zostanie on znaleziony,
8     // to zwracany jest indeks w tablicy. W przeciwnym wypadku zwracana
9     // jest wartość -1, oznaczająca, że szukanej wartości nie ma w tablicy
10
11    public static int binarySearch(String[] array, String value)
12    {
13        int first;      // Pierwszy element tablicy
14        int last;       // Ostatni element tablicy
15        int middle;     // Środkowy punkt wyszukiwania
16        int position;   // Pozycja wartości szukanej
17        boolean found;  // Znacznik
18
19        // Ustalamy wartości początkowe
20        first = 0;
21        last = array.length - 1;
22        position = -1;
23        found = false;
24
25        // Poszukujemy wartości parametru value
26        while (!found && first <= last)
27        {
```

```

28     // Obliczamy indeks środkowego elementu
29     middle = (first + last) / 2;
30
31     // Sprawdzamy, czy szukana wartość jest zapisana w środkowym elemencie...
32     if (array[middle].equals(value))
33     {
34         found = true;
35         position = middle;
36     }
37     // ...jeśli nie, to wartość może się znajdować w pierwszej połówce...
38     else if (array[middle].compareTo(value) > 0)
39         last = middle - 1;
40     // ...jeśli nie, to wartość może się znajdować w drugiej połówce tablicy...
41     else
42         first = middle + 1;
43 }
44
45     // Zwracamy indeks, pod którym znajduje się szukana wartość, lub -1,
46     // jeśli wartości nie udało się odszukać
47     return position;
48 }
49 }
```

## Python

### Sortowanie i wyszukiwanie list

#### Sortowanie bąbelkowe w języku Python

Na listingu 9.13 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Pythonie wersja pseudokodu z listingu 9.1. Zajmujemy się tutaj algorytmem sortowania bąbelkowego.

**Listing 9.13** (bubble\_sort.py)

```

1  # Uwaga: To nie jest kompletny program
2 #
3 # Funkcja bubbleSort korzysta z algorytmu sortowania bąbelkowego
4 # w celu posortowania tablicy liczb całkowitych
5 # Zwróć uwagę na poniższe:
6 # (1) Nie musimy przekazywać funkcji wielkości tablicy, ponieważ
7 #     możemy skorzystać z funkcji len
8 # (2) Nie mamy też osobnej metody podmieniającej elementy,
9 #     podmiana wykonywana jest w ramach funkcji
10
11 def bubble_sort(arr):
12     # Zmiennej max_element przypisujemy długość tablicy
13     # zmniejszoną o 1, tego wymaga zewnętrzna pętla
14     max_element = len(arr) - 1
15
16     # Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
17     # ostatniemu indeksowi tablicy, który ma zostać porównany w danym
18     # przejściu. Na samym początku zmienna maxElement będzie równa
```

```

19  # indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
20  # będzie ona dekrementowana o 1
21  while max_element >= 0:
22      # Set index to 0, necessary for the inner loop.
23      index = 0
24
25      # Pętla wewnętrzna śledzi tablicę i porównuje
26      # sąsiadujące ze sobą elementy. Porównywane są
27      # elementy o indeksach od 0 do maxElement
28      # Jeżeli kolejność elementów jest nieprawidłowa,
29      # zostają one zamienione miejscami
30      while index <= max_element - 1:
31          # Porównujemy dwa sąsiadujące elementy
32          if arr[index] > arr[index + 1]:
33              # i jeśli trzeba, zamieniamy je miejscami
34              temp = arr[index]
35              arr[index] = arr[index + 1]
36              arr[index + 1] = temp
37          # Inkrementujemy indeks
38          index = index + 1
39      # Dekrementujemy zmienną max_element
40      max_element = max_element - 1

```

### Sortowanie przez wybieranie w języku Python

Na listingu 9.14 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Pythonie wersja pseudokodu modułu `selectionSort` z listingu 9.5. Zajmujemy się tutaj algorytmem sortowania przez wybieranie.

#### **Listing 9.14 (selection\_sort.py)**

```

1  # Uwaga: To nie jest kompletny program
2  #
3  # Funkcja selection_sort sortuje tablicę liczb całkowitych
4  # za pomocą algorytmu sortowania przez wybieranie
5
6  def selection_sort(arr):
7      # Zmiennej start_scan przypisujemy wartość 0,
8      # tego wymaga pętla zewnętrzna. W ten sposób wyznaczana
9      # jest pozycja startowa dla skanowania
10     start_scan = 0
11
12     # Pętla zewnętrzna wykonuje iterację dla każdego elementu tablicy
13     # z wyjątkiem ostatniego. Zmienna startScan oznacza indeks,
14     # od którego należy rozpocząć przeszukiwanie
15     while start_scan < len(arr) - 1:
16         # Na początku zakładamy, że to pierwszy element
17         # ma najniższą wartość
18         min_index = start_scan
19         min_value = arr[start_scan]
20
21         # Inicjalizowanie indeksu dla wewnętrznej pętli
22         index = start_scan + 1
23
24         # Przeszukujemy tablicę, zaczynając od drugiego elementu
25         # w przeszukiwanym obszarze. Poszukujemy w tym obszarze

```

```

26     # elementu o najmniejszej wartości
27     while index < len(arr):
28         if arr[index] < min_value:
29             min_value = arr[index]
30             min_index = index
31         # Increment index.
32         index = index + 1
33
34     # Zamieniamy miejscami element, w którym jest zapisana najmniejsza wartość,
35     # i pierwszy element w przeszukiwanym obszarze
36     arr[min_index] = arr[start_scan]
37     arr[start_scan] = min_value
38
39     # Inkrementujemy zmiennej start_scan
40     start_scan = start_scan + 1

```

### Sortowanie przez wstawianie w języku Python

Na listingu 9.15 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Pythonie wersja pseudokodu modułu `insertionSort` z listingu 9.6. Zajmujemy się tutaj algorytmem sortowania przez wstawianie.

**Listing 9.15** `(insertion_sort.py)`

```

1  # Uwaga: To nie jest kompletny program
2  #
3  # Funkcja insertion_sort sortuje tablicę liczb całkowitych
4  # za pomocą algorytmu sortowania przez wstawianie
5
6  def insertion_sort(arr):
7      # Początkowy indeks na potrzeby zewnętrznej pętli
8      index = 1
9
10     # W zmiennej index będą zapisywane w pętli kolejne indeksy,
11     # począwszy od indeksu numer 1, dla tego,
12     # że element o indeksie 0 uważamy za już posortowany
13     while index < len(arr):
14         # Pierwszy element występujący zaraz za posortowanym fragmentem
15         # tablicy to array[index]. W zmiennej unsortedValue zapisujemy
16         # wartość przypisaną do tego elementu
17         unsorted_value = arr[index]
18
19         // Śledzenie zaczynamy od indeksu pierwszego elementu,
20         # zaraz za posortowanym fragmentem tablicy
21         scan = index
22
23         # Przenosimy pierwszy element spoza posortowanego fragmentu
24         # w odpowiednie miejsce posortowanego fragmentu
25         while scan > 0 and arr[scan - 1] > unsorted_value:
26             arr[scan] = arr[scan - 1]
27             scan = scan - 1
28
29         # Wstawiamy wartość nieposortowaną we właściwe miejsce
30         # posortowanego fragmentu tablicy
31         arr[scan] = unsorted_value

```

```

32
33     # Inkrementujemy zmienną index
34     index = index + 1

```

### Wyszukiwanie binarne w języku Python

Na listingu 9.16 prezentuję tylko fragment programu. Widoczna jest na nim napisana w Pythonie wersja pseudokodu modułu `binarySearch` z listingu 9.7. Zajmujemy się tutaj algorytmem wyszukiwania binarnego.

#### **Listing 9.16** (binary\_search.py)

```

1  # Uwaga: To nie jest kompletny program
2  #
3  # Funkcja binary_search wykonuje operację szukania binarnego
4  # na liście ciągów znaków. Na liście poszukiwany jest ciąg znaków
5  # podany w parametrze value. Jeżeli zostanie on znaleziony,
6  # to zwracany jest indeks z listy, w przeciwnym wypadku zwracana
7  # jest wartość -1, oznaczająca, że szukanej wartości nie ma na liście
8
9 def binary_search(arr, value):
10    # Ustalamy wartości początkowe
11    first = 0
12    last = len(arr) - 1
13    position = -1
14    found = False
15
16    # Poszukujemy wartości parametru value
17    while not found and first <= last:
18        # Obliczamy indeks środkowego elementu
19        middle = (first + last) / 2
20
21        # Sprawdzamy, czy szukana wartość jest zapisana w środkowym elemencie...
22        if arr[middle] == value:
23            found = True
24            position = middle
25        # ...jeśli nie, to wartość może się znajdować w pierwszej połówce...
26        elif arr[middle] > value:
27            last = middle - 1
28        # ...jeśli nie, to wartość może się znajdować w drugiej połówce tablicy...
29        else:
30            first = middle + 1
31
32    # Zwracamy indeks, pod którym znajduje się szukana wartość, lub -1,
33    # jeśli wartości nie udało się odszukać
34    return position

```

## C++

### Sortowanie i wyszukiwanie tablic

#### Sortowanie bąbelkowe w języku C++

Na listingu 9.17 prezentuję tylko fragment programu. Widoczna jest na nim napisana w C++ wersja pseudokodu z listingu 9.1. Zajmujemy się tutaj algorytmem sortowania bąbelkowego.

**Listing 9.17** (BubbleSort.cpp)

```

1 // Uwaga: To nie jest kompletny program
2 // Funkcja bubbleSort
3 void bubbleSort(int array[], int size)
4 {
5     int maxElement; // Wyznacza ostatni element do porównania
6     int index;       // Indeks elementu do porównania
7
8     // Pętla zewnętrzna ustawia zmienną maxElement na wartość równą
9     // ostatniemu indeksowi tablicy, który ma zostać porównany w danym
10    // przejściu. Na samym początku zmienna maxElement będzie równa
11    // indeksowi ostatniego elementu tablicy. Podczas każdej kolejnej iteracji
12    // będzie ona dekrementowana o 1
13    for (maxElement = size - 1; maxElement >= 0; maxElement--)
14    {
15        // Pętla wewnętrzna śledzi tablicę i porównuje
16        // sąsiadujące ze sobą elementy. Porównywane są
17        // elementy o indeksach od 0 do maxElement.
18        // Jeżeli kolejność elementów jest nieprawidłowa,
19        // zostają one zamienione miejscami
20        for (index = 0; index <= maxElement - 1; index++)
21        {
22            // Porównujemy dwa sąsiadujące elementy
23            if (array[index] > array[index + 1])
24            {
25                // i jeśli trzeba, zamieniamy je miejscami
26                swap(array[index], array[index+1]);
27            }
28        }
29    }
30 }
31
32 // Funkcja swap przyjmuje dwa argumenty typu String
33 // i zamienia ich wartości
34 void swap(int &a, int &b)
35 {
36     int temp;
37     temp = a;
38     a = b;
39     b = temp;
40 }
```

### Sortowanie przez wybieranie w języku C++

Na listingu 9.18 prezentuję tylko fragment programu. Widoczna jest na nim napisana w C++ wersja pseudokodu modułu `selectionSort` z listingu 9.5. Zajmujemy się tutaj algorytmem sortowania przez wybieranie.

#### **Listing 9.18 (SelectionSort.cpp)**

```

1 // Uwaga: To nie jest kompletny program
2 //
3 // Funkcja selectionSort sortuje tablicę liczb całkowitych
4 // Tablica zostaje posortowana w kolejności malejącej
5
6 void selectionSort(int array[], int size)
7 {
8     int startScan;    // Początkowa pozycja skanowania
9     int index;        // Przechowuje wartość indeksu
10    int minIndex;    // Element o najmniejszej wartości w skanowaniu
11    int minValue;    // Najmniejsza wartość znaleziona w skanowaniu
12
13    // Pętla zewnętrzna wykonuje iterację dla każdego elementu tablicy
14    // z wyjątkiem ostatniego. Zmienna startScan oznacza indeks,
15    // od którego należy rozpocząć przeszukiwanie
16    for (startScan = 0; startScan < (size-1); startScan++)
17    {
18        // Na początku zakładamy, że to pierwszy element
19        // ma najniższą wartość
20        minIndex = startScan;
21        minValue = array[startScan];
22
23        // Przeszukujemy tablicę, zaczynając od drugiego elementu
24        // w przeszukiwanym obszarze. Poszukujemy w tym obszarze
25        // elementu o najmniejszej wartości
26        for(index = startScan + 1; index < size; index++)
27        {
28            if (array[index] < minValue)
29            {
30                minValue = array[index];
31                minIndex = index;
32            }
33        }
34
35        // Zamieniamy miejscami element, w którym jest zapisana najmniejsza wartość,
36        // i pierwszy element w przeszukiwanym obszarze
37        swap(array[minIndex], array[startScan]);
38    }
39 }
40
41 // Funkcja swap zamienia wartości dwóch przekazanych
42 // jej argumentów
43 void swap(int &a, int &b)
44 {
45     int temp;
46     temp = a;
47     a = b;
48     b = temp;
49 }
```

### Sortowanie przez wstawianie w języku C++

Na listingu 9.19 prezentuję tylko fragment programu. Widoczna jest na nim napisana w C++ wersja pseudokodu modułu `insertionSort` z listingu 9.6. Zajmujemy się tutaj algorytmem sortowania przez wstawianie.

#### **Listing 9.19 (InsertionSort.cpp)**

```

1 // Uwaga: To nie jest kompletny program
2 //
3 // Metoda insertionSort sortuje tablicę liczb całkowitych
4 // Tablica zostaje posortowana w kolejności malejącej
5
6 void insertionSort(int array[], int size)
7 {
8     int unsortedValue; // Pierwsza nieposortowana wartość
9     int scan;           // Używana podczas skanowania tablicy
10
11    // W zmiennej index będą zapisywane w pętli kolejne indeksy,
12    // począwszy od indeksu numer 1, dlatego,
13    // że element o indeksie 0 uważamy za już posortowany
14    for (int index = 1; index < size; index++)
15    {
16        // Pierwszy element występujący zaraz za posortowanym fragmentem
17        // tablicy to array[index]. W zmiennej unsortedValue zapisujemy
18        // wartość przypisaną do tego elementu
19        unsortedValue = array[index];
20
21        // Śledzenie zaczynamy od indeksu pierwszego elementu,
22        // zaraz za posortowanym fragmentem tablicy
23        scan = index;
24
25        // Przenosimy pierwszy element spoza posortowanego fragmentu
26        // w odpowiednie miejsce posortowanego fragmentu
27        while (scan > 0 && array[scan-1] > unsortedValue)
28        {
29            array[scan] = array[scan - 1];
30            scan--;
31        }
32
33        // Wstawiamy wartość nieposortowaną we właściwe miejsce
34        // posortowanego fragmentu tablicy
35        array[scan] = unsortedValue;
36    }
37 }
```

### Wyszukiwanie binarne w języku C++

Na listingu 9.20 prezentuję tylko fragment programu. Widoczna jest na nim napisana w C++ wersja pseudokodu modułu `binarySearch` z listingu 9.7. Zajmujemy się tutaj algorytmem wyszukiwania binarnego.

**Listing 9.20 (BinarySearch.cpp)**

```

1 // Uwaga: To nie jest kompletny program
2 //
3 // Metoda binarySearch wykonuje operację szukania binarnego
4 // w tablicy ciągów znaków. W tablicy poszukiwany jest ciąg znaków
5 // podany w parametrze value. Jeżeli zostanie on znaleziony,
6 // to zwracany jest indeks w tablicy, w przeciwnym wypadku zwracana
7 // jest wartość -1, oznaczająca, że szukanej wartości nie ma w tablicy
8
9 int binarySearch(string array[], string value, int size)
10 {
11     int first;          // Pierwszy element tablicy
12     int last;           // Ostatni element tablicy
13     int middle;         // Punkt środkowy wyszukiwania
14     int position;       // Pozycja wartości szukanej
15     bool found;         // Znacznik
16
17     // Ustalamy wartości początkowe
18     first = 0;
19     last = size - 1;
20     position = -1;
21     found = false;
22
23     // Poszukujemy wartości parametru value
24     while (!found && first <= last)
25     {
26         // Obliczamy indeks środkowego elementu
27         middle = (first + last) / 2;
28
29         // Sprawdzamy, czy szukana wartość jest zapisana w środkowym elemencie...
30         if (array[middle] == value)
31         {
32             found = true;
33             position = middle;
34         }
35         // ...jeśli nie, to wartość może się znajdować w pierwszej połówce...
36         else if (array[middle] > value)
37             last = middle - 1;
38         // ...jeśli nie, to wartość może się znajdować w drugiej połówce tablicy...
39         else
40             first = middle + 1;
41     }
42
43     // Zwracamy indeks, pod którym znajduje się szukana wartość, lub -1,
44     // jeśli wartości nie udało się odszukać
45     return position;
46 }
```

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Algorytmy tego typu służą do ustawiania elementów tablicy w określonej kolejności.
  - a) algorytmy wyszukiwania
  - b) algorytmy sortowania
  - c) algorytmy porządkowania
  - d) algorytm wybierania
2. Jeśli elementy tablicy są uporządkowane w tej kolejności, to wartości są uszeregowane od najmniejszej do największej.
  - a) kolejność asymptotyczna
  - b) kolejność logarytmiczna
  - c) kolejność rosnąca
  - d) kolejność malejąca
3. Jeśli elementy tablicy są uporządkowane w tej kolejności, to wartości są uszeregowane od największej do najmniejszej.
  - a) kolejność asymptotyczna
  - b) kolejność logarytmiczna
  - c) kolejność rosnąca
  - d) kolejność malejąca
4. Algorytm ten wykonuje kilka przejść przez całą tablicę, powodując, że elementy o większej wartości przesuwają się po każdym przejściu w kierunku końca tablicy.
  - a) sortownie bąbelkowe
  - b) sortowanie przez wybieranie
  - c) sortowanie przez wstawienie
  - d) sortowanie sekwencyjne
5. W algorytmie tym odnajdywany jest element tablicy o najmniejszej wartości, a następnie jest przenoszony do elementu o indeksie 0. W następnym kroku odnajdywany jest kolejny element tablicy o najmniejszej wartości i jest przenoszony do elementu o indeksie 1. Operacja ta jest powtarzana aż do momentu, gdy wszystkie elementy znajdą się na właściwych miejscach tablicy.
  - a) sortownie bąbelkowe
  - b) sortowanie przez wybieranie
  - c) sortowanie przez wstawienie
  - d) sortowanie sekwencyjne
6. Algorytm ten rozpoczyna się od uporządkowania dwóch pierwszych elementów tablicy, które tworzą posortowany fragment tablicy. Następnie na właściwe miejsce względem dwóch pierwszych elementów przesuwany jest trzeci element. Po wstawieniu trzeciego elementu otrzymujemy nowy, posortowany fragment tablicy, składający się z trzech elementów. Operacja ta jest powtarzana w przypadku czwartego i dalszych elementów tablicy, aż do momentu, gdy tablica zostanie w całości posortowana.

- a) sortownie bąbelkowe
  - b) sortowanie przez wybieranie
  - c) sortowanie przez
  - d) sortowanie sekwencyjne
7. W tym algorytmie wyszukiwania każdy element tablicy porównywany jest z wartością szukaną.
- a) algorytm wyszukiwania sekwencyjnego
  - b) algorytm wyszukiwania binarnego
  - c) algorytm wyszukiwania naturalnego
  - d) algorytm wyszukiwania przez wybieranie
8. W algorytmie tym sukcesywnie dzieli się przeszukiwany fragment tablicy na pół.
- a) algorytm wyszukiwania sekwencyjnego
  - b) algorytm wyszukiwania binarnego
  - c) algorytm wyszukiwania naturalnego
  - d) algorytm wyszukiwania przez wybieranie
9. Ten algorytm wyszukiwania sprawdza się w przypadku niewielkich tablic.
- a) algorytm wyszukiwania sekwencyjnego
  - b) algorytm wyszukiwania binarnego
  - c) algorytm wyszukiwania naturalnego
  - d) algorytm wyszukiwania przez wybieranie
10. Algorytm ten wymaga, aby zawartość tablicy była uporządkowana.
- a) algorytm wyszukiwania sekwencyjnego
  - b) algorytm wyszukiwania binarnego
  - c) algorytm wyszukiwania naturalnego
  - d) algorytm wyszukiwania przez wybieranie

### **Prawda czy fałsz?**

1. Dane posortowane w kolejności rosnącej to takie dane, które są uporządkowane od najmniejszej wartości do największej.
2. Dane posortowane w kolejności malejącej to takie dane, które są uporządkowane od największej wartości do najmniejszej.
3. W każdym języku programowania można uporządkować ciągi znaków za pomocą algorytmu sortowania bąbelkowego.
4. W algorytmie wyszukiwania sekwencyjnego **średnia** liczba operacji porównania w tablicy zawierającej  $n$  elementów jest równa  $n/2$  (przy założeniu, że szukana wartość znajduje się w tablicy).
5. W algorytmie wyszukiwania sekwencyjnego **maksymalna** liczba operacji porównania w tablicy zawierającej  $n$  elementów jest równa  $n/2$  (przy założeniu, że szukana wartość znajduje się w tablicy).

### **Krótką odpowiedź**

1. Ile elementów będzie musiała odczytać funkcja korzystająca z algorytmu wyszukiwania sekwencyjnego, aby odnaleźć wartość znajdująca się na samym końcu tablicy składającej się z 10000 elementów?

2. Ile razy funkcja wykorzystująca algorytm wyszukiwania sekwencyjnego będzie w typowym przypadku musiała odczytać wartości w tablicy zawierającej  $n$  elementów, aby odszukać określona wartość?
3. Założymy, że za pomocą funkcji korzystającej z algorytmu wyszukiwania binarnego szukamy pewnej wartości, która znajduje się dokładnie w środkowym elemencie tablicy. Ile razy funkcja ta będzie musiała odczytać wartości elementów tablicy, zanim odnajdzie szukaną wartość?
4. Jaka jest maksymalna liczba operacji porównania w przypadku wyszukiwania wartości w tablicy zawierającej 1000 elementów za pomocą algorytmu wyszukiwania binarnego?
5. Dlaczego algorytm sortowania bąbelkowego w przypadku bardzo dużych tablic jest mało wydajny?
6. Dlaczego w przypadku dużych tablic algorytm sortowania przez wybieranie jest bardziej wydajny od algorytmu sortowania bąbelkowego?
7. Wymień kroki, które będzie musiał wykonać algorytm sortowania przez wybieranie dla następującej listy wartości: 4, 1, 3, 2.
8. Wymień kroki, które będzie musiał wykonać algorytm sortowania przez wstawianie dla następującej listy wartości: 4, 1, 3, 2.

### **Warsztat projektanta algorytmów**

1. Zaprojektuj moduł swap, który będzie przyjmował dwa argumenty typu Real i zamieniał ich wartości.
2. Który algorytm został zaimplementowany w poniższym pseudokodzie?

```

Declare Integer maxElement
Declare Integer index

For maxElement = arraySize - 1 To 0 Step -1
    For index = 0 To maxElement - 1
        If array[index] > array[index + 1] Then
            Call swap(array[index], array[index + 1])
        End If
    End For
End For

```

1. Który algorytm został zaimplementowany w poniższym pseudokodzie?

```

Declare Integer index
Declare Integer scan
Declare Integer unsortedValue

For index = 1 To arraySize - 1
    Set unsortedValue = array[index]
    Set scan = index

    While scan > 0 AND array[scan-1] < array[scan]
        Call swap(array[scan-1], array[scan])
        Set scan = scan - 1

    End

    Set array[scan] = unsortedValue
End For

```

4. Który algorytm został zaimplementowany w poniższym pseudokodzie?

```

Declare Integer startScan

Declare Integer minIndex
Declare Integer minValue
Declare Integer index

For startScan = 0 To arraySize - 2
    Set minValue = startScan
    Set minValue = array[startScan]

    For index = startScan + 1 To arraySize - 1
        If array[index] < minValue
            Set minValue = array[index]
            Set minIndex = index
        End If
    End For

    Call swap(array[minIndex], array[startScan])
End For

```

## Ćwiczenia z wykrywania błędów

1. Założymy, że razem z poniższym modelem `main` w programie pojawia się także funkcja `binarySearch`, którą przedstawiłem w tym rozdziale. Dlaczego pseudokod w module `main` nie zadziała prawidłowo?

```

// Program za pomocą funkcji binarySearch wyszukuje w tablicy imię.
// Zakładamy, że funkcja binarySearch
// została już zdefiniowana
Module main()
    Constant Integer SIZE = 5

    Declare String names[SIZE] = "Zenon", "Jurek", "Patrycja",
                           "Marek", "Sylwia"
    Declare String searchName
    Declare Integer index
    Display "Wprowadź imię, którego szukasz."
    Input searchName

    Set index = binarySearch(names, searchName, SIZE)

    If index != -1 Then
        Display "Znalazłem ", searchName
    Else
        Display "Nie znalazłem ", searchName
    End If

End

```

## Ćwiczenia programistyczne

### 1. Posortowane wyniki gry w golfa

Zaprojektuj program, który będzie prosił użytkownika o wprowadzenie 10 wyników gry w golfa. Wyniki zapisz w tablicy liczb typu Integer. Następnie posortuj tablicę w kolejności rosnącej i wyświetl jej zawartość na ekranie.

### 2. Posortowana lista imion

Zaprojektuj program, który umożliwi użytkownikowi wprowadzenie 20 imion do tablicy zawierającej ciągi znaków. Posortuj tablicę w kolejności rosnącej (alfabetycznej) i wyświetl jej zawartość.

### 3. Modyfikacja programu do obliczania opadów deszczu

Przypomnij sobie ćwiczenie programistyczne 3. z rozdziału 8. Poprośłem Cię tam o zaprojektowanie programu, w którym użytkownik będzie mógł wprowadzić do tablicy poziom opadów deszczu w każdym z 12 miesięcy. Program powinien obliczać roczny poziom opadów, średni miesięczny poziom opadów oraz miesiące z najmniejszymi i największymi opadami. Rozwiń ten program tak, aby dodatkowo porządkował elementy tablicy w kolejności rosnącej i wyświetlał zawartość tablicy.

### 4. Wyszukiwanie imienia

Zmodyfikuj program stworzony w ćwiczeniu 2. tak, aby użytkownik mógł wyszukać w tablicy określone imię.

### 5. Sprawdzanie kodu doładowującego telefon

Przypomnij sobie ćwiczenie programistyczne 5. z rozdziału 8. Poprośłem Cię tam o zaprojektowanie programu, który będzie sprawdzał kod doładowujący. Program powinien sprawdzać, czy wprowadzony kod jest prawidłowy, porównując go z listą predefiniowanych kodów. Zmodyfikuj ten program tak, aby wyszukiwał kod za pomocą algorytmu wyszukiwania binarnego zamiast algorytmu wyszukiwania sekwencyjnego.

### 6. Wyszukiwanie numeru telefonu

Przypomnij sobie ćwiczenie programistyczne 7. z rozdziału 8. Poprośłem Cię tam o zaprojektowanie programu zawierającego dwie tablice równolegle: tablicę ciągów znaków o nazwie people i tablicę ciągów znaków o nazwie phoneNumbers. Program umożliwiał wyszukanie określonej osoby w tablicy people. Jeżeli osoba została odnaleziona, program wyświetlał jej numer telefonu. Zmodyfikuj program tak, aby wyszukiwał osobę za pomocą algorytmu wyszukiwania binarnego zamiast algorytmu wyszukiwania sekwencyjnego.

### 7. Test wydajności algorytmów wyszukiwania

Zaprojektuj program, w którym znajdzie się tablica zawierająca co najmniej 20 liczb typu Integer. Program powinien następnie wywołać moduł, który będzie wyszukiwał w tablicy określoną wartość za pomocą algorytmu wyszukiwania sekwencyjnego. Moduł ten powinien zapisać liczbę operacji porównania, które musiał wykonać, zanim odszukał w tablicy wartość. Następnie program

powinien wywołać kolejny moduł, w którym do odszukania wartości wykorzystany będzie algorytm wyszukiwania binarnego. Ten moduł także powinien zapisać liczbę operacji porównania. Na końcu **wyświetl na ekranie obie**

**8. Test wydajności algorytmów sortowania**

Zmodyfikuj moduły służące do sortowania bąbelkowego, sortowania przez wybieranie i sortowania przez wstawienie, które zaprezentowałem w tym rozdziale, tak, aby zapisywały liczbę operacji zamiany miejscami dwóch elementów tablicy.

Następnie zaprojektuj program, w którym znajdą się trzy identyczne tablice zawierające co najmniej 20 liczb typu Integer. Program powinien wywołać po kolei każdy moduł i przekazać do niego kolejne tablice, a następnie wyświetlić liczbę operacji zamiany miejscami elementów, jakie wykonał dany algorytm.

**TEMATYKA**

- |  |  |
|--|--|
| 10.1 Odczyt i zapis do plików — informacje wstępne | 10.4 Przetwarzanie rekordów                |
| 10.2 Przetwarzanie plików za pomocą pętli          | 10.5 Separatory sterowania                 |
| 10.3 Korzystanie z plików i tablic                 | 10.6 Rzut oka na języki Java, Python i C++ |

**10.1**

## Odczyt i zapis do plików — informacje wstępne

**WYJAŚNIENIE:** Kiedy program musi zapisać pewne dane, aby mógł z nich skorzystać w przyszłości, zapisuje je w pliku. Zapisane w ten sposób dane mogą zostać następnie odczytane.

Ponieważ zmienne zapisane w pamięci RAM znikają z niej po zamknięciu programu, we wszystkich programach, które do tej pory projektowaliśmy, użytkownik musiał po każdym uruchomieniu programu wprowadzać dane ponownie. Jeśli chcemy, aby program zapamiętał dane pomiędzy kolejnymi uruchomieniami programu, musimy je jakoś zapisać. Dane zapisujemy w pliku, który zazwyczaj przechowywany jest na dysku komputera. Gdy zapiszemy dane w pliku, pozostaną one w nim nawet wtedy, gdy program zakończy działanie. Zapisane w pliku dane można odczytać po kolejnym uruchomieniu programu.

Większość programów, z których korzystasz na co dzień, zapisuje dane w plikach. Oto kilka przykładów:

- **Procesory tekstowe.** Procesorów tekstowych używa się do tworzenia listów, notatek, raportów i innych dokumentów. Dokumenty te zapisywane są w plikach, dzięki czemu można je modyfikować i drukować.

- **Edytory graficzne.** Edytory graficzne służą do tworzenia grafiki i edycji zdjęć (np. zrobionych aparatem cyfrowym). Obrazy, które tworzy się lub edytuje w takim programie, także zapisywane są w plikach.
- **Arkusze kalkulacyjne.** Arkusze kalkulacyjne służą do wykonywania różnych operacji na danych liczbowych. W wierszach i kolumnach arkusza kalkulacyjnego można umieszczać wzory matematyczne. Arkusz taki można zapisać w pliku, aby można było z niego skorzystać ponownie w późniejszym czasie.
- **Gry.** Wiele gier komputerowych zapisuje dane w plikach. Przykładowo w wielu grach istnieje lista graczy i wyników uzyskanych przez nich w trakcie rozgrywki. W grach tego typu wyświetla się ekran zawierający nazwy graczy i ich wyniki w kolejności od najwyższego do najniższego. W niektórych grach można zapisać w pliku aktualny stan gry, aby można było z niej wyjść, a w przyszłości kontynuować rozgrywkę od danego miejsca — nie trzeba więc rozpoczynać gry od samego początku.
- **Przeglądarki internetowe.** Podczas przeglądania niektórych stron internetowych przeglądarka zapisuje na dysku mały plik zwany **ciasteczkami**. Plik ten zawiera zazwyczaj informacje dotyczące danej sesji, np. listę produktów umieszczonych w koszyku.

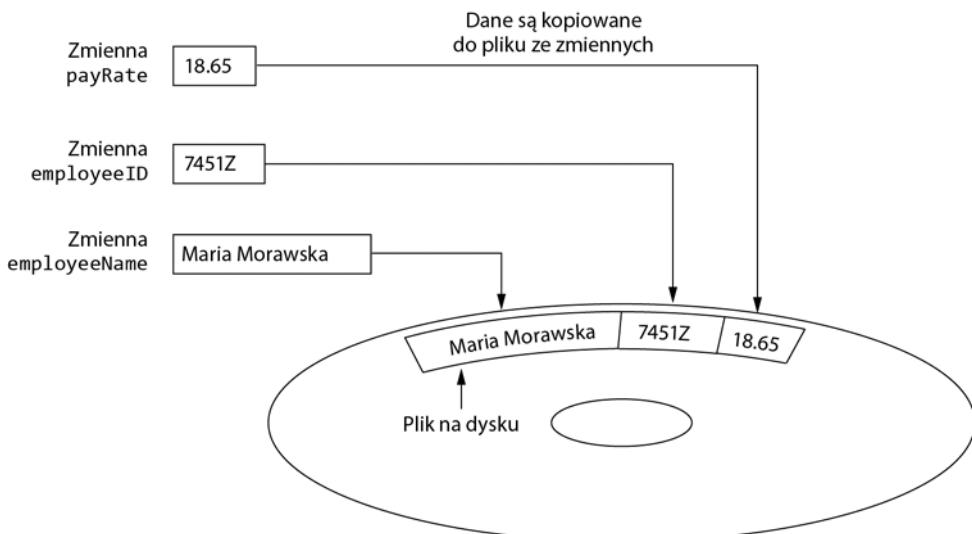
Programy używane na co dzień w biznesie w znacznym stopniu operują na danych zapisanych w plikach. Programy do obliczania wynagrodzenia zapisują w plikach dane dotyczące pracowników, programy magazynowe przechowują w plikach informacje dotyczące asortymentu, systemy księgowie przechowują w plikach dane finansowe dotyczące sprzedaży itp.

Odoszcząc się do procesu zapamiętywania danych w pliku, mówimy często o zapisywaniu danych do pliku. W momencie zapisywania danych do pliku dane są kopowane z pamięci RAM do pliku. Operację tę ilustruje rysunek 10.1. Plik, w którym zapisujemy dane, nazywamy **plikiem wyjściowym**. Nazwa ta wynika stąd, że program zapisuje w pliku dane wyjściowe.

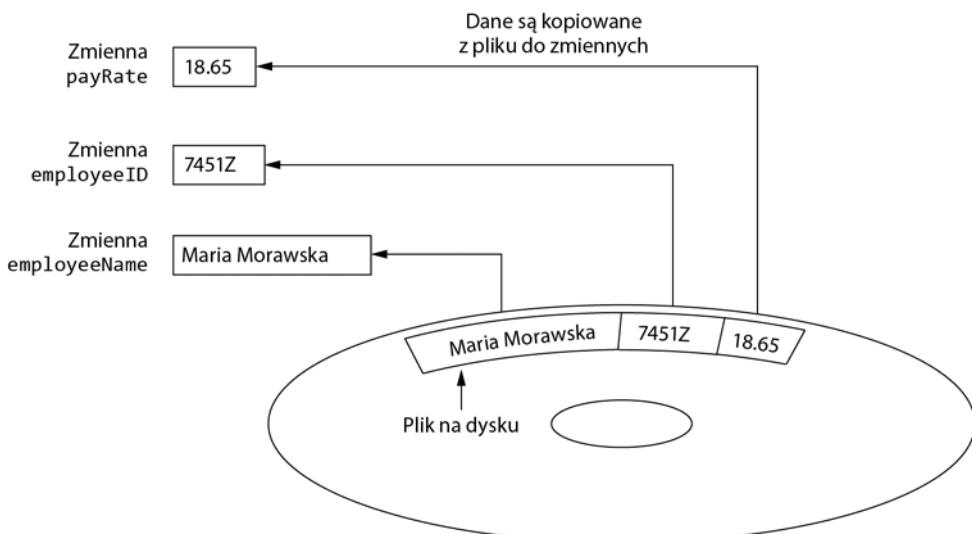
Operację pobierania danych zapisanych w pliku nazywamy odczytywaniem danych z pliku. Podczas odczytywania dane są kopowane do zmiennych znajdujących się w pamięci RAM. Ilustruje to rysunek 10.2. Plik, z którego odczytujemy dane, nazywamy **plikiem wejściowym**. Nazwa ta wynika stąd, że program pobiera z pliku dane wejściowe.

W tym rozdziale wyjaśnię, jak należy projektować programy, które zapisują dane do plików i odczytują dane z plików. Chcąc użyć w programie danego pliku, trzeba wykonać następujące trzy operacje:

1. **Otworzyć plik.** Po otwarciu pliku utworzona zostanie łączność między programem a plikiem. Otwarcie pliku do zapisu zazwyczaj spowoduje utworzenie go na dysku i umożliwi programowi zapisanie w nim danych. Otwarcie pliku do odczytu pozwoli programowi odczytać zawarte w nim dane.
2. **Przetworzyć plik.** Podczas tej operacji program zapisuje w pliku dane (jeśli plik został otwarty do zapisu) lub odczytuje z niego dane (jeśli został otwarty do odczytu).



**Rysunek 10.1.** Zapisywanie danych do pliku



**Rysunek 10.2.** Odczytywanie danych z pliku

3. **Zamknąć plik.** Kiedy program zakończy przetwarzanie pliku, plik należy zamknąć. Zamknięcie pliku rozłącza plik i program.

## Typy plików

Ogólnie rzecz biorąc, pliki można podzielić na dwa rodzaje: pliki tekstowe i pliki binarne. **Plik tekstowy** zawiera dane tekstowe zapisane za pomocą kodowania ASCII lub Unicode. Nawet jeśli plik będzie zawierał liczby, zostaną one zapisane w pliku jako zbiór znaków. Dzięki temu plik taki można otworzyć i podejrzeć w edytorze tekstowym,

takim jak na przykład Notatnik. **Plik binarny** zawiera dane, które nie zostały zapisane za pomocą kodowania znaków. Dlatego pliku binarnego nie da się podejrzeć w edytorze tekstowym.

## Metody dostępu do plików

Większość języków programowania zapewnia dostęp po pliku na dwa sposoby: poprzez dostęp sekwencyjny i dostęp swobodny. Gdy używamy **dostępu sekwencyjnego**, dane będziemy musieli odczytywać po kolej — od początku pliku do jego końca. Jeżeli przykładowo chcemy odczytać informację zapisaną na samym końcu pliku, będziemy musieli i tak odczytać wszystkie dane od początku tego pliku — nie możemy po prostu przeskoczyć do danego miejsca w pliku. W podobny sposób działają magnetofony kasetowe: jeżeli chcesz posłuchać utworu umieszczonego na końcu taśmy, musisz albo przewinąć taśmę do tego miejsca, albo odsłuchać po kolej utwory występujące wcześniej — nie można przejść bezpośrednio do konkretnego utworu.

W przypadku **dostępu swobodnego** (zwanego także **dostępem bezpośredniim**) możemy przeskoczyć do konkretnego miejsca w pliku bez potrzeby odczytywania danych zapisanych wcześniej. Podobnie wygląda słuchanie muzyki na odtwarzaczu CD czy MP3: w każdej chwili możesz przejść bezpośrednio do utworu, którego chcesz wysłuchać.

W tym rozdziale zajmiemy się sekwencyjnym dostępem do plików — jest on bardzo prosty, a dzięki niemu łatwo będzie Ci zrozumieć podstawowe operacje na plikach.

## Tworzenie pliku i zapisywanie w nim danych

Większość użytkowników komputerów jest już przyzwyczajona do tego, że pliki identyfikuje się za pomocą ich nazwy. Przykładowo kiedy utworzysz w procesorze tekstowym dokument i chcesz go zapisać, musisz wskazać jego nazwę. Kiedy sprawdzasz zawartość dysku za pomocą narzędzia takiego jak Eksplorator Windows, wyświetla się lista nazw plików. Na rysunku 10.3 przedstawiłem trzy przykładowe pliki widoczne w oknie Eksploratora Windows.



Rysunek 10.3. Trzy pliki

Każdy system operacyjny może się charakteryzować innymi zasadami nazewnictwa plików. W wielu systemach występują **rozszerzenia plików**, które mają postać kropki i kilku liter umieszczonych na końcu nazwy pliku. Przykładowo pliki widoczne na rysunku 10.3 mają rozszerzenia .jpg, .txt i .doc. Rozszerzenie wskazuje zazwyczaj typ danych, jakie są przechowywane w danym pliku. Na przykład rozszerzenie .jpg oznacza plik, w którym jest zapisany obraz skompresowany za pomocą standardu JPEG. Rozszerzenie .txt oznacza przeważnie plik, w którym zapisane są dane tekstowe. Rozszerzenie .doc to z kolei plik zawierający dokument programu Microsoft Word. W programach

przedstawionych w tej książce będę się posługiwał plikami z rozszerzeniem *.dat* — co będzie oznaczało po prostu, że w pliku zapisane są dane.

Wykonując w programie operacje na pliku, będziemy posługiwali się w kodzie dwiema nazwami. Pierwszą z nich jest nazwa pliku, która będzie wskazywała plik zapisany na dysku komputera. Drugą będzie wewnętrzna nazwa obiektu reprezentującego dany plik, podobna do nazwy zmiennej. Tak naprawdę obiekt ten deklarujemy w podobny sposób jak zwykłą zmienną. Oto deklaracja obiektu reprezentującego plik w pseudokodzie:

```
Declare outputFile customerFile
```

W poleceniu tym wskazujemy dwie rzeczy:

- słowo *OutputFile* wskazuje **tryb**, w jakim otwieramy plik. W tym przypadku wskazujemy, że chcemy otworzyć plik w trybie do zapisu;
- słowo *customerFile* to nazwa obiektu reprezentującego plik w programie.

Pomimo że składnia ta różni się nieco w zależności od języka programowania, zazwyczaj podczas deklarowania tego obiektu należy wskazać zarówno tryb, w jakim zamierzamy używać danego pliku, jak i nazwę obiektu, który będzie go reprezentował w programie.

Następnym krokiem jest otwarcie pliku. W pseudokodzie będziemy używali do tego celu polecenia *Open*. Oto przykład:

```
Open customerFile "customer.dat"
```

Po słowie *Open* pojawia się nazwa obiektu reprezentującego plik, a następnie ciąg znaków wskazujący nazwę pliku. Po wykonaniu tego polecenia na dysku komputera zostanie utworzony plik o nazwie *customers.dat*, a gdy w dalszej części programu będziemy chcieli zapisać w nim dane, odwołamy się do niego za pomocą nazwy *customerFile*.



**OSTRZEŻENIE!** Pamiętaj, że podczas otwierania pliku wyjściowego jest on tworzony na dysku. W większości języków programowania, gdy plik o wskazanej nazwie istnieje już na dysku, podczas jego otwierania do zapisu jego zawartość zostanie wykasowana.

## Zapisywanie danych do pliku

Gdy otworzymy już plik wyjściowy, możemy zapisać w nim dane. W pseudokodzie będziemy używali do tego celu polecenia *Write*. Przykładowo następujące polecenie:

```
Write customerFile "Konrad Grzesiak"
```

spowoduje, że w pliku powiązanym z obiektem *customerFile* zostanie zapisany ciąg znaków "Konrad Grzesiak". W pliku można także zapisać wartość przypisaną do zmiennej, na przykład w następujący sposób:

```
Declare String name = "Konrad Grzesiak"
Write customerFile name
```

Drugie polecenie spowoduje, że w pliku powiązanym z obiektem `customerFile` zostanie zapisana wartość przypisana do zmiennej `name`. W tym przykładzie jest to ciąg znaków, ale równie dobrze może to być wartość liczbową.

## Zamykanie pliku

Kiedy program zakończy przetwarzanie plików, należy taki plik zamknąć. Zamknięcie pliku powoduje odłączenie pliku od programu. W przypadku niektórych systemów niezamknięcie pliku może spowodować utratę zapisanych w nim danych. Dzieje się tak dla tego, że dane przed zapisaniem ich na dysku zapisywane są w **buforze**, czyli malej „przechowalni” w pamięci komputera. Kiedy bufor jest już pełny, system operacyjny zapisuje jego zawartość na dysku. Dzięki takiemu rozwiązaniu system jest wydajniejszy, ponieważ zapis w pamięci jest znacznie szybszy od zapisu na dysku. Operacja zamykania pliku powoduje, że wszystkie nie zapisane do tej pory dane znajdujące się w buforze zostają zapisane na dysku.

Do zamykania pliku z pseudokodzie będziemy używali polecenia `Close`. Na przykład polecenie:

```
Close customerFile
```

spowoduje zamknięcie pliku powiązanego z obiektem `customerFile`.

Na listingu 10.1 przedstawiłem pseudokod przykładowego programu, w którym otwieram plik, zapisuję w nim dane, a następnie go zamykam. Na rysunku 10.4 znajduje się schemat blokowy programu. Ponieważ polecenie `Write` jest polecienniem generującym dane wyjściowe, na schemacie blokowym oznaczamy je za pomocą równoległoboku.

**Listing 10.1**

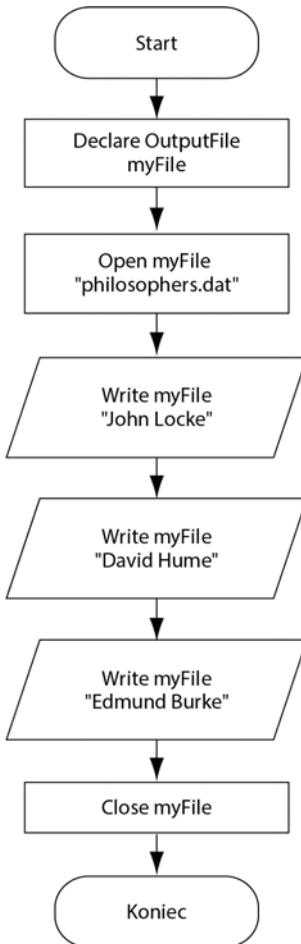


```

1 // Deklarujemy obiekt reprezentujący plik wyjściowy
2 Declare OutputFile myFile
3
4 // Otwieramy na dysku plik
5 // o nazwie philosophers.dat
6 Open myFile "philosophers.dat"
7
8 // Zapisujemy w pliku
9 // nazwiska trzech filozofów
10 Write myFile "John Locke"
11 Write myFile "David Hume"
12 Write myFile "Edmund Burke"
13
14 // Zamykamy plik
15 Close myFile

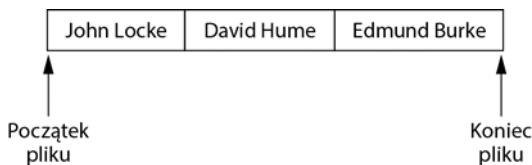
```

W linii 2. deklaruję obiekt reprezentujący plik wyjściowy o nazwie `myFile`. W linii 6. otwieram plik `philosophers.dat` i łączę go z obiektem `myFile`. Dzięki temu, zapisując dane w pliku `philosophers.dat`, będziemy się mogli posługiwać w programie nazwą `myFile`.



**Rysunek 10.4.** Schemat blokowy programu z listingu 10.1

Polecenia w liniach od 10. do 12. zapisuję w pliku trzy elementy: linii 10. ciąg znaków "John Locke", w linii 11. ciąg znaków "David Hume", a w linii 12. ciąg znaków "Edmund Burke". W linii 15. zamykam plik. Gdyby był to prawdziwy program, w pliku zapisane zostałyby trzy elementy przedstawione na rysunku 10.5.



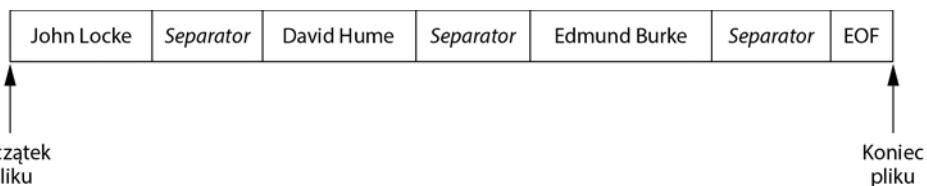
**Rysunek 10.5.** Zawartość pliku philosophers.dat

Zwróc uwagę, że elementy te występują w pliku w takiej samej kolejności, w jakiej je zapisywaliśmy. Pierwszym elementem jest „John Locke”, drugim — „David Hume”, a trzecim — „Edmund Burke”. Dlaczego jest to istotne, dowiesz się za chwilę, gdy zaczniemy odczytywać dane z tego pliku.

## Separatory i znaczniki EOF

Na rysunku 10.5 widać trzy elementy, które zapisaliśmy w pliku *philosophers.dat*. Jednak w wielu językach programowania struktura tego pliku będzie nieco bardziej złożona. W wielu językach występuje specjalny znak zwany separatorem, który umieszczany jest po każdym elemencie. **Separator** to predefiniowany znak lub ciąg znaków, które wskazują koniec pewnego fragmentu danych. Jego zadaniem jest oddzielenie od siebie poszczególnych informacji zapisanych w pliku. Znak ten (lub ciąg znaków) różni się w zależności od systemu operacyjnego.

Poza separatorami w wielu systemach na końcu pliku umieszczany jest inny znak specjalny lub ciąg znaków zwany **znacznikiem end-of-file** (EOF). Znacznik ten wskazuje, w którym miejscu kończą się dane zapisane w pliku. Znak, który reprezentuje znacznik EOF, różni się w zależności od systemu operacyjnego. Na rysunku 10.6 przedstawiłem wygląd pliku *philosophers.dat*, uwzględniając separatory i znacznik EOF.



**Rysunek 10.6.** Zawartość pliku *philosophers.dat* z uwzględnieniem separatorów i znacznika EOF

## Odczytywanie danych z pliku

Aby odczytać dane z pliku, najpierw musimy zadeklarować obiekt, za pomocą którego będziemy się odnosili do tego pliku. W pseudokodzie będę używał do tego celu polecenia **Declare**:

```
Declare inputFile inventoryFile
```

W poleceniu tym wskazujemy dwie rzeczy:

- słowo **InputFile** wskazuje tryb, w jakim otwieramy plik. W tym przypadku wskazujemy, że chcemy otworzyć plik w trybie do odczytu;
- słowo **inventoryFile** to nazwa obiektu reprezentującego plik w programie.

Jak wspomniałem wcześniej, sposób deklaracji trybu i obiektu reprezentującego plik może wyglądać nieco inaczej w każdym z języków programowania.

Następnym krokiem jest otwarcie pliku. W pseudokodzie będziemy używali do tego celu polecenia **Open**. Przykładowo w poleceniu:

```
Open inventoryFile "inventory.dat"
```

za słowem **Open** pojawia się nazwa obiektu reprezentującego plik, a następnie ciąg znaków zawierający nazwę pliku, który chcemy otworzyć. Po wykonaniu tego polecenia otwarty zostanie plik *inventory.dat* i podczas odczytywania danych będziemy mogli się do niego odnosić w programie za pomocą nazwy **inventoryFile**.

Ponieważ otwieramy plik, który zamierzamy odczytać, plik ten powinien się już znajdować na dysku. Podczas próby otwarcia pliku, którego nie ma na dysku, w większości systemów pojawi się błąd.

## Odczytywanie danych

Kiedy otworzymy już plik, możemy przystąpić do odczytania zawartych w nim danych. Aby odczytać z pliku porcję danych, w pseudokodzie będziemy używać polecenia Read. Oto przykład (zakładam, że zmienna `itemName` została już wcześniej zadeklarowana w programie):

```
Read inventoryFile itemName
```

Polecenie to odczyta porcję danych zapisanych w pliku powiązanym z obiektem `inventoryFile`. Dane te zostaną następnie zapisane w zmiennej `itemName`.

## Zamykanie pliku

Jak już wspomniałem, gdy program zakończy pracę z plikiem, należy taki plik zamknąć. Aby zamknąć plik wejściowy, podobnie jak to miało miejsce w przypadku pliku wyjściowego, będziemy używać w pseudokodzie polecenia Close. Przykładowo polecenie:

```
Close inventoryFile
```

spowoduje zamknięcie pliku powiązanego z obiektem `inventoryFile`.

Na listingu 10.2 przedstawiłem program, w którym otwieram plik utworzony wcześniej za pomocą programu z listingu 10.1, odczytuje z niego dane, zamykam go, a następnie wyświetlам odczytane nazwiska. Na rysunku 10.7 widoczny jest schemat blokowy programu. Zauważ, że polecenia Read także reprezentowane są przez równoległoboki.

### **Listing 10.2**



```

1 // Deklarujemy obiekt reprezentujący plik wejściowy
2 Declare InputFile myFile
3
4 // Deklarujemy trzy zmienne, w których zapiszemy nazwiska
5 // odczytane z pliku
6 Declare String name1, name2, name3
7
8 // Otwieramy plik philosophers.dat
9 // zapisany na dysku
10 Open myFile "philosophers.dat"
11
12 // Odczytujemy nazwiska trzech filozofów
13 // i zapisujemy je w zmiennych
14 Read myFile name1
15 Read myFile name2
16 Read myFile name3
17
18 // Zamykamy plik
19 Close myFile
20
21 // Wyświetlamy odczytane nazwiska
22 Display "Oto imiona i nazwiska trzech filozofów:"

```

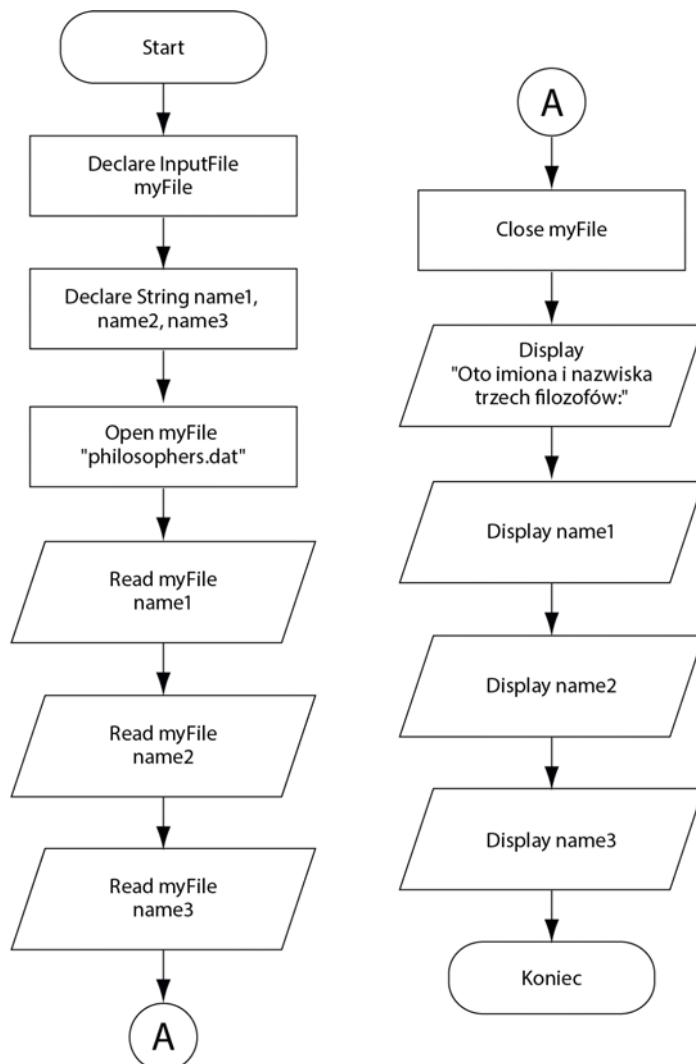
```

23 Display name1
24 Display name2
25 Display name3

```

### Wynik działania programu

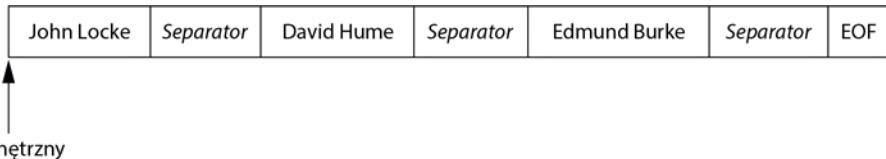
Oto imiona i nazwiska trzech filozofów:  
John Locke  
David Hume  
Edmund Burke



**Rysunek 10.7.** Schemat blokowy programu z listingu 10.2

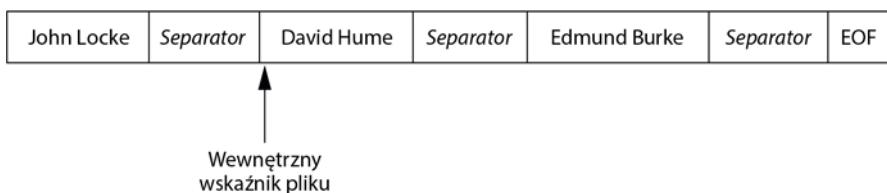
W linii 2. deklaruję obiekt `myFile` reprezentujący plik. W linii 6. deklaruję trzy ciągi znaków: `name1`, `name2` i `name3`. W zmiennych tych zapiszę wartości odczytane z pliku. W linii 10. otwieram plik `philosophers.dat` i łączę go z obiektem `myFile`. Dzięki temu, odczytując dane z pliku `philosophers.dat`, będę mógł się posługiwać nazwą `myFile`.

Podczas przetwarzania pliku program zapamiętuje specjalną wartość zwaną **wewnętrzny wskaźnikiem pliku**. Wskaźnik ten wskazuje miejsce w pliku, od którego rozpocznie się kolejny odczyt danych. Zaraz po otwarciu pliku wewnętrzny wskaźnik pliku jest ustawiany na samym początku pliku. Po wykonaniu polecenia w linii 10. wskaźnik pliku zostanie ustawiony tak, jak widać na rysunku 10.8.



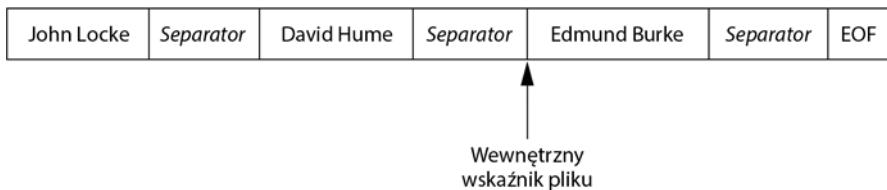
**Rysunek 10.8.** Początkowe położenie wewnętrznego wskaźnika pliku

Polecenie Read w linii 14. odczytuje z pliku element zaczynający się od miejsca, które wskazuje wewnętrzny wskaźnik pliku, a następnie zapisze ten element w zmiennej name1. Po wykonaniu tego polecenia w zmiennej name1 znajdzie się ciąg znaków "John Locke". Ponadto wewnętrzny wskaźnik pliku przesunie się do kolejnego elementu w pliku, co przedstawiłem na rysunku 10.9.



**Rysunek 10.9.** Położenie wewnętrznego wskaźnika pliku po wykonaniu pierwszego polecenia Read

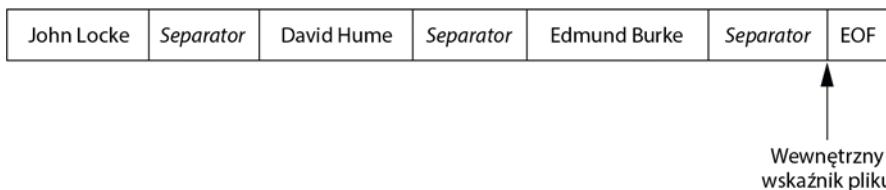
W linii 15. pojawia się kolejne polecenie Read. Odczyta ono element wskazywany przez wewnętrzny wskaźnik pliku i zapisze go w zmiennej name2. Po wykonaniu tego polecenia w zmiennej name2 znajdzie się ciąg znaków "David Hume". Wewnętrzny wskaźnik pliku przesunie się do kolejnego elementu w pliku, co przedstawiłem na rysunku 10.10.



**Rysunek 10.10.** Położenie wewnętrznego wskaźnika pliku po wykonaniu drugiego polecenia Read

Kolejne polecenie Read pojawia się w linii 16. Odczyta ono element wskazywany przez wewnętrzny wskaźnik pliku i zapisze go w zmiennej name3. Po wykonaniu tego polecenia

w zmiennej name3 znajdzie się ciąg znaków "Edmund Burke". Wewnętrzny wskaźnik pliku przesunie się doznacznika EOF, co przedstawiłem na rysunku 10.11.



**Rysunek 10.11.** Położenie wewnętrznego wskaźnika pliku po wykonaniu trzeciego polecenia Read

W linii 19. zamykam plik. Polecenia Display widoczne w liniach od 23. do 25. wyświetlały wartości przypisane do zmiennych name1, name2 i name3.



**UWAGA:** Czy zwróciłeś uwagę na to, że program z listingu 10.2 odczytuje kolejne elementy z pliku *philosophers.dat* w sposób sekwencyjny — od samego początku pliku do jego końca? Przypomnij sobie, o czym wspomniałem na początku tego rozdziału — w dokładnie taki sposób działa dostęp sekwencyjny do pliku.

## Dołączanie danych do istniejącego pliku

W większości języków programowania próba otwarcia do zapisu pliku, który już istnieje na dysku, spowoduje jego usunięcie i utworzenie nowego pliku o tej samej nazwie. W niektórych sytuacjach chcemy jednak, aby plik nie został usunięty, a nowe dane zostały dopisane na końcu tego pliku.

W większości języków programowania można także otworzyć plik w **trybie dołączania**, który działa w następujący sposób:

- Jeżeli plik już istnieje na dysku, nie zostanie on usunięty. Jeżeli pliku jeszcze nie ma na dysku, zostanie utworzony.
- Podczas zapisywania danych nowe dane zostaną zapisane na samym końcu pliku.

Składnia polecenia otwierającego plik w trybie dołączania różni się znaczco w zależności od języka. W pseudokodzie będziemy używać w tym celu słowa AppendMode, które dodamy do polecenia Declare, tak jak tutaj:

```
Declare outputFile AppendMode myFile
```

W poleceniu tym zadeklarowałem obiekt reprezentujący plik wyjściowy o nazwie *myFile*, który zostanie otwarty w trybie dołączania. Założmy, że plik *friends.dat* jest już zapisany na dysku i zawiera następujące imiona:

```
Jarek
Renata
Grzegorz
Grażyna
Kasia
```

W zamieszczonym poniżej pseudokodzie otwieram ten plik i dołączam do niego kolejne imiona:

```
Declare outputFile AppendMode myFile  
Open myFile "friends.dat"  
Write myFile "Mateusz"  
Write myFile "Krzysztof"  
Write myFile "Sylwia"  
Close myFile
```

Po uruchomieniu programu plik *friends.dat* będzie wyglądał następująco:

```
Jarek  
Renata  
Grzegorz  
Grażyna  
Kasia  
Mateusz  
Krzysztof  
Sylwia
```



## Punkt kontrolny

- 10.1. Gdzie przechowywane są zazwyczaj pliki?
- 10.2. Co to jest plik wyjściowy?
- 10.3. Co to jest plik wejściowy?
- 10.4. Jakie trzy operacje należy wykonać w programie, aby skorzystać z pliku?
- 10.5. Wymień dwa ogółem typy plików. Czym się one różnią?
- 10.6. Wymień dwie metody dostępu do pliku. Czym się one różnią?
- 10.7. Jeśli chcesz skorzystać w programie z pliku, jakie dwie nazwy musisz umieścić w kodzie?
- 10.8. Co się stanie w większości języków programowania, gdy spróbujesz w programie otworzyć do zapisu plik, który już istnieje na dysku?
- 10.9. Co ma na celu otwieranie pliku?
- 10.10. Co ma na celu zamykanie pliku?
- 10.11. Co to jest separator? W jaki sposób wykorzystuje się separatory w plikach?
- 10.12. Jaki znacznik jest zapisywany na końcu pliku w wielu systemach?
- 10.13. Co to jest wewnętrzny wskaźnik pliku? Które miejsce wskazuje wewnętrzny wskaźnik pliku zaraz po otwarciu pliku wejściowego?
- 10.14. W jakim trybie otworzysz plik, jeśli zechcesz dopisać do niego dane? W którym miejscu pliku zostaną dopisane nowe dane?

**10.2**

## Przetwarzanie plików za pomocą pętli

**WYJAŚNIENIE:** W plikach znajduje się przeważnie bardzo dużo danych, więc program zazwyczaj przetwarza pliki za pomocą pętli.

Istnieją programy, które przechowują w plikach tylko niewielką ilość danych, jednak z reguły w plikach zapisuje się duże zbiory informacji. Kiedy musimy w programie odczytać dane z tak dużego pliku, zwykle posługujemy się pętlą. Spójrz na przykładowy kod na listingu 10.3. W programie tym pobieram od użytkownika wartość sprzedaży w ciągu kilku dni, a następnie zapisuję te informacje w pliku *sales.dat*. Zadaniem użytkownika jest wprowadzenie liczby dni i wartości sprzedaży w każdym dniu. W przykładowym wywołaniu programu użytkownik wprowadził wartość sprzedaży z pięciu dni. Na rysunku 10.12 przedstawilem zawartość pliku *sales.dat* — są to dane, które wprowadził użytkownik podczas przykładowego wywołania programu. Na rysunku 10.13 widoczny jest schemat blokowy programu.

**Listing 10.3**

```

1 // Zmienna, w której zapiszemy liczbę dni
2 Declare Integer numDays
3
4 // Zmienna licznikowa
5 Declare Integer counter
6
7 // Zmienna, w której zapiszemy wartość sprzedaży
8 Declare Real sales
9
10 // Deklarujemy plik wyjściowy
11 Declare OutputFile salesFile
12
13 // Pobieramy liczbę dni
14 Display "Z ilu dni chcesz wprowadzić sprzedaż?"
15 Input numDays
16
17 // Otwieramy plik sales.dat
18 Open salesFile "sales.dat"
19
20 // Pobieramy wartość sprzedaży w każdym dniu
21 // i zapisujemy ją do pliku
22 For counter = 1 To numDays
23   // Pobieramy wartość sprzedaży w danym dniu
24   Display "Wprowadź wartość sprzedaży w dniu nr ", counter
25   Input sales
26
27 // Zapisujemy wartość do pliku
28 Write salesFile sales
29 End For
30
31 // Zamykamy plik
32 Close salesFile
33 Display "Zapisano dane w pliku sales.dat."

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Z ilu dni chcesz wprowadzić sprzedaży?

**5 [Enter]**

Wprowadź wartość sprzedaży w dniu nr 1

**1000.00 [Enter]**

Wprowadź wartość sprzedaży w dniu nr 2

**2000.00 [Enter]**

Wprowadź wartość sprzedaży w dniu nr 3

**3000.00 [Enter]**

Wprowadź wartość sprzedaży w dniu nr 4

**4000.00 [Enter]**

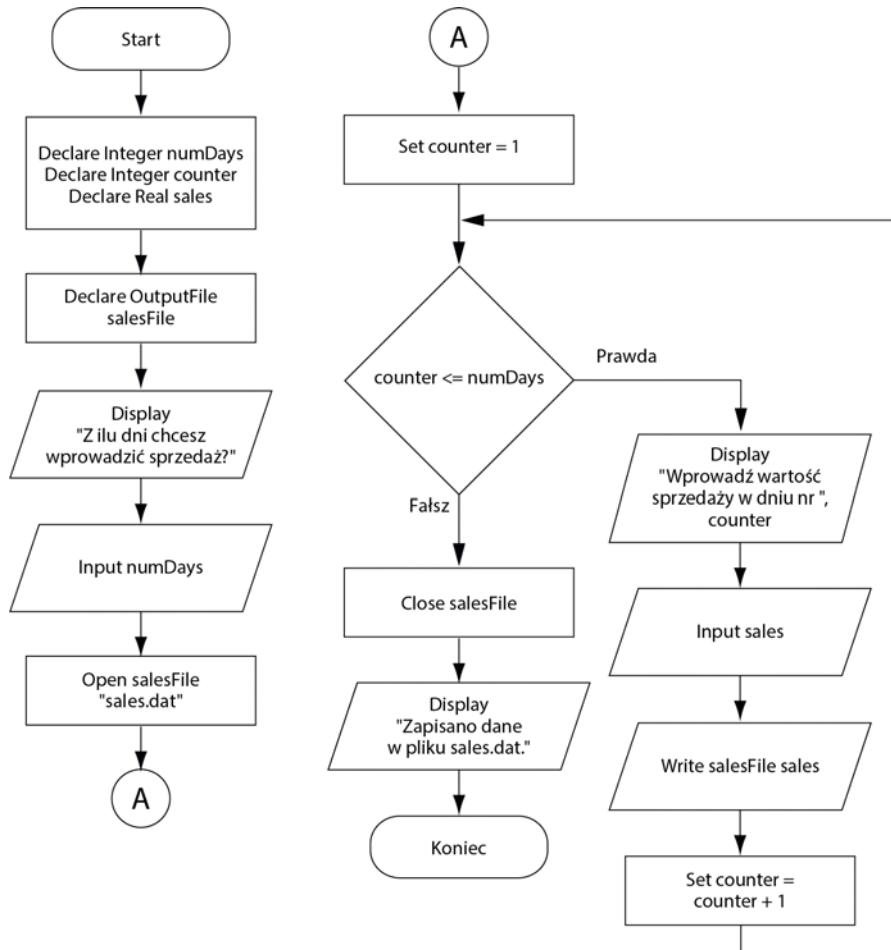
Wprowadź wartość sprzedaży w dniu nr 5

**5000.00 [Enter]**

Zapisano dane w pliku sales.dat.

1000.00	Separator	2000.00	Separator	3000.00	Separator	4000.00	Separator	5000.00	Separator	EOF
---------	-----------	---------	-----------	---------	-----------	---------	-----------	---------	-----------	-----

**Rysunek 10.12.** Zawartość pliku sales.dat



**Rysunek 10.13.** Schemat blokowy programu z listingu 10.3

## Odczytywanie pliku za pomocą pętli i wykrywanie znacznika EOF

Bardzo często program będzie musiał odczytać zawartość pliku, ale nie będzie wiedział, ile elementów jest w nim zapisanych. Przykładowo plik, który powstał po uruchomieniu programu z listingu 10.3, może zawierać dowolną liczbę elementów, ponieważ to użytkownik wprowadza liczbę dni i wartość sprzedaży. Jeśli użytkownik wprowadzi liczbę 5, program pobierze od niego i zapisze w pliku sprzedaż z pięciu dni. Jeśli użytkownik wprowadzi liczbę 100, program pobierze i zapisze w pliku sprzedaż ze stu dni.

Stwarza to pewien problem, ponieważ Twoim zadaniem jest napisanie programu, który będzie przetwarzał wszystkie elementy zapisane w pliku, niezależnie od ich liczby. Założymy, że chcemy teraz napisać program, który odczyta zapisane w pliku wartości sprzedaży i obliczy ich sumę. Możesz wykorzystać w tym przypadku pętlę, ale gdy program spróbuje odczytać dane spoza pliku, pojawi się błąd. Program musi więc w jakiś sposób wiedzieć, w którym miejscu kończą się dane zapisane w pliku i dalsze ich odczytywanie nie ma sensu.

W większości języków programowania dostępna jest funkcja biblioteczna rozwiązuająca ten problem. W pseudokodzie będziemy używali w tym celu funkcji o nazwie eof. Ma ona następującą postać:

eof(*nazwaObiektuReprezentującegoPlik*)

Funkcja eof przyjmuje jako argument obiekt reprezentujący plik i zwraca wartość True, gdy program odczyta już wszystkie dane zapisane w pliku. Gdy w pliku znajdują się jeszcze jakieś dane, funkcja eof zwraca wartość False. Na listingu 10.4 przedstawiony przykład z wykorzystaniem funkcji eof. Program ten wyświetla wartości sprzedaży zapisane w pliku sales.dat.

**Listing 10.4**



```

1 // Deklarujemy plik wejściowy
2 Declare inputFile salesFile
3
4 // Deklarujemy zmienną, w której zapiszemy wartość sprzedaży
5 // odczytaną z pliku
6 Declare Real sales
7
8 // Otwieramy plik sales.dat
9 Open salesFile "sales.dat"
10
11 Display "Oto wartości sprzedaży:"
12
13 // Odczytujemy wszystkie wartości z pliku
14 // i wyświetlamy je
15 While NOT eof(salesFile)
16   Read salesFile sales
17   Display currencyFormat(sales)
18 End While
19
20 // Zamykamy plik
21 Close salesFile

```

**Wynik działania programu**

Oto wartości sprzedaży:

1 000,00 zł  
2 000,00 zł  
3 000,00 zł  
4 000,00 zł  
5 000,00 zł

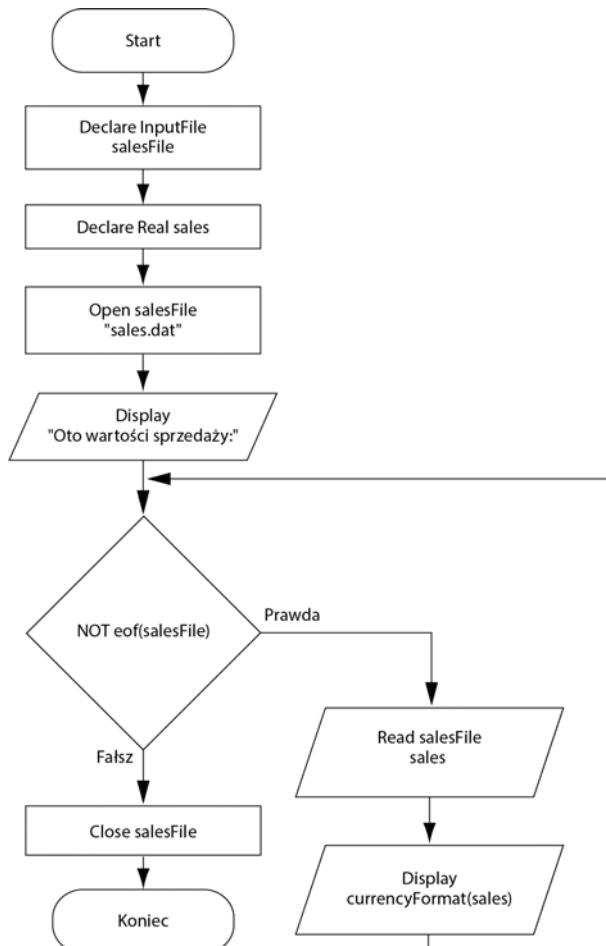
Przyjrzyj się bliżej linii 15.:

```
While NOT eof(salesFile)
```

Tę linie kodu można odczytać jako: *Dopóki NIE dojdiesz do końca pliku...*

```
While eof(salesFile) == False
```

Pomimo że oba polecenia są równoważne, większość programistów najpewniej skorzysta z pierwszej formy, z użyciem operatora NOT, ponieważ jest ona bardziej przejrzysta. Na rysunku 10.14 przedstawiłem schemat blokowy programu.



Rysunek 10.14. Schemat blokowy programu z listingu 10.4



## W centrum uwagi

### Korzystanie z plików

Krystian tworzy reklamy telewizyjne dla lokalnych przedsiębiorców. Tworząc daną reklamę, nagrywa zazwyczaj kilka krótkich ujęć, które następnie montuje, aby uzyskać ostateczną wersję reklamy. Poprosił Cię o zaprojektowania dwóch programów:

1. Program, który pozwoli mu wprowadzić czas trwania każdego ujęcia (w sekundach). Czasy te chciałby zapisać w pliku.
2. Program, który odczyta utworzony wcześniej plik, wyświetli czas trwania każdego ujęcia, a następnie wyświetli czas trwania całej reklamy.

Oto algorytm pierwszego programu:

1. Pobierz od użytkownika liczbę ujęć w danej reklamie.
2. Otwórz plik wyjściowy.
3. Dla każdego ujęcia:
  - Pobierz czas trwania ujęcia.
  - Zapisz czas trwania ujęcia do pliku.
4. Zamknij plik.

Na listingu 10.5 przedstawiłem pseudokod pierwszego programu. Na rysunku 10.15 jest widoczny jego schemat blokowy.

**Listing 10.5**



```

1 // Deklarujemy plik wyjściowy
2 Declare outputFile videoFile
3
4 // Zmienna, w której zapiszemy liczbę ujęć
5 Declare Integer numVideos
6
7 // Zmienna, w której zapiszemy czas trwania ujęcia
8 Declare Real runningTime
9
10 // Zmienna licznikowa
11 Declare Integer counter
12
13 // Pobieramy liczbę ujęć
14 Display "Wprowadź liczbę ujęć, z których składa się reklama."
15 Input numVideos
16
17 // Otwieramy plik wyjściowy, w którym zapiszemy czasy trwania ujęć
18 Open videoFile "video_times.dat"
19
20 // Zapisujemy w pliku czas trwania każdego ujęcia
21 For counter = 1 To numVideos
22   // Pobieramy czas trwania ujęcia
23   Display "Wprowadź czas trwania ujęcia nr ", counter
24   Input runningTime
25
26   // Zapisujemy w pliku czas trwania ujęcia
27   Write videoFile runningTime
28 End For

```

```

29
30 // Zamykamy plik
31 Close videoFile
32 Display "Czasy trwania ujęć zostały zapisane w pliku video_times.dat."

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę ujęć, z których składa się reklama.

**6 [Enter]**

Wprowadź czas trwania ujęcia nr 1

**24.5 [Enter]**

Wprowadź czas trwania ujęcia nr 2

**12.2 [Enter]**

Wprowadź czas trwania ujęcia nr 3

**14.6 [Enter]**

Wprowadź czas trwania ujęcia nr 4

**20.4 [Enter]**

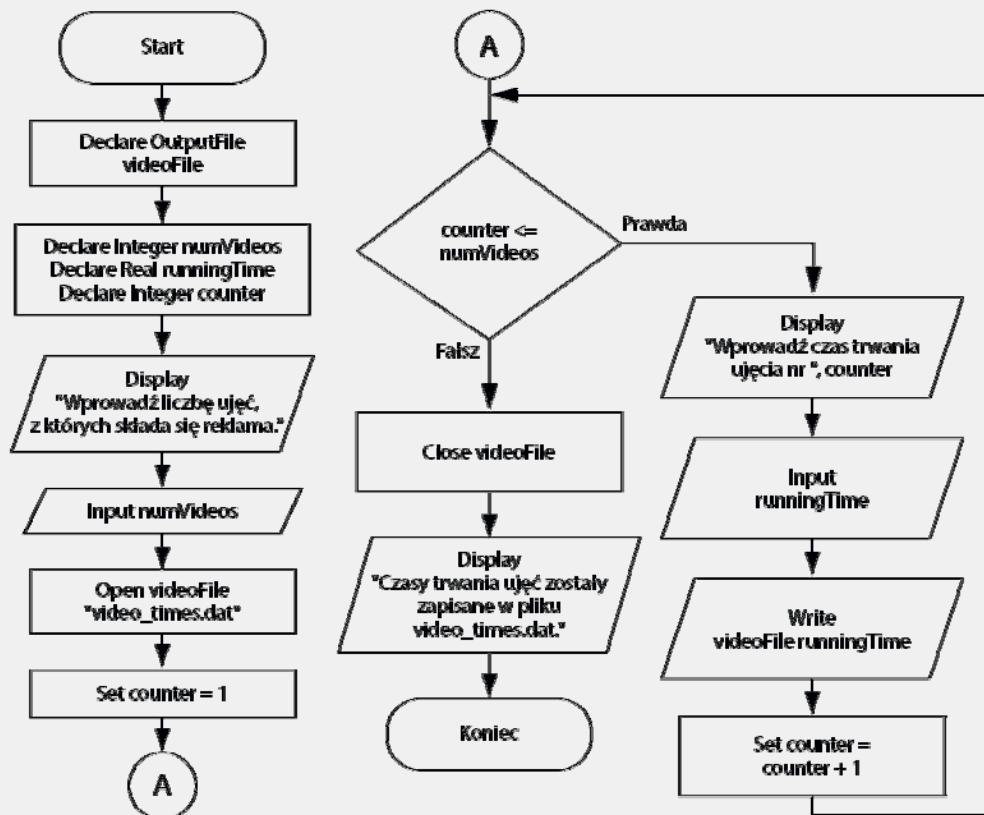
Wprowadź czas trwania ujęcia nr 5

**22.5 [Enter]**

Wprowadź czas trwania ujęcia nr 6

**19.3 [Enter]**

Czasy trwania ujęć zostały zapisane w pliku video\_times.dat.



Rysunek 10.15. Schemat blokowy programu z listingu 10.5

Oto algorytm drugiego programu:

1. Zainicjalizuj akumulator wartością 0.
2. Otwórz plik wejściowy.
3. Dopóki program nie dotrze do końca pliku:
4. Odczytaj wartość z pliku.
5. Dodaj odczytaną wartość do akumulatora.
6. Zamknij plik.
7. Wyświetl zawartość akumulatora, który będzie zawierał całkowity czas trwania reklamy.

Na listingu 10.6 przedstawiłem pseudokod drugiego programu. Na rysunku 10.16 jest widoczny jego schemat blokowy.

### **Listing 10.6**



```

1 // Deklarujemy plik wejściowy
2 Declare InputFile videoFile
3
4 // Zmienna, w której zapiszemy
5 // czas trwania ujęcia odczytany z pliku
6 Declare Real runningTime
7
8 // Akumulator, w którym zapiszemy całkowity czas trwania reklamy,
9 // zainicjalizowany wartością 0
10 Declare Real total = 0
11
12 // Otwieramy plik video_times.dat
13 Open videoFile "video_times.dat"
14
15 Display "Oto czasy trwania (wyrażone w sekundach) ",
16     "wszystkich ujęć w reklamie:"
17
18 // Odczytujemy czasy trwania ujęć z pliku,
19 // wyświetlamy je i obliczamy ich sumę
20 While NOT eof(videofile)
21     // Odczytujemy czas trwania ujęcia
22     Read videoFile runningTime
23
24     // Wyświetlamy czas trwania ujęcia
25     Display runningTime
26
27     // Sumujemy czas trwania ujęcia
28     Set total = total + runningTime
29 End While
30
31 // Zamykamy plik
32 Close videoFile
33
34 // Wyświetlamy całkowity czas trwania reklamy
35 Display "Całkowity czas trwania reklamy wynosi",
36     total, " s."

```

### **Wynik działania programu**

Oto czasy trwania (wyrażone w sekundach) wszystkich ujęć w reklamie:

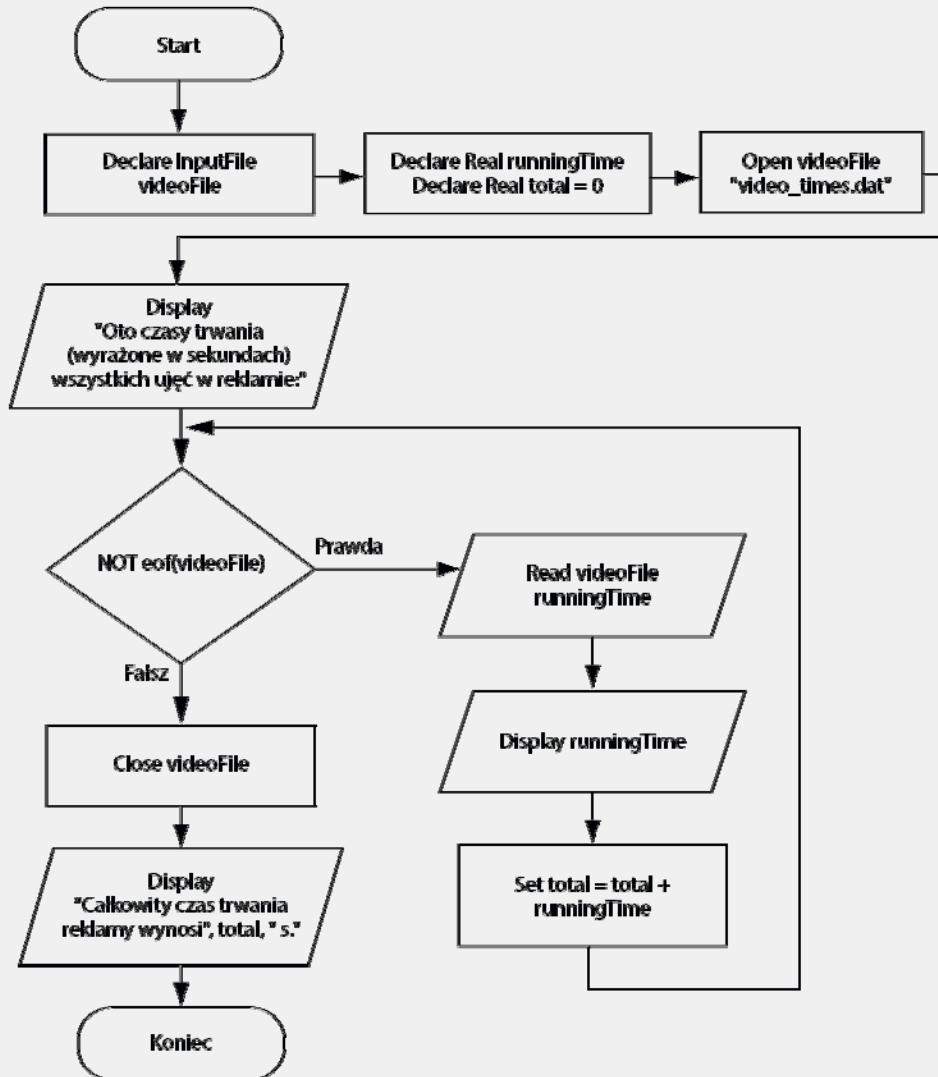
24.5  
12.2  
14.6

20.4

22.5

19.3

Całkowity czas trwania reklamy wynosi 113.5 s.

**Rysunek 10.16.** Schemat blokowy programu z listingu 10.6

### Punkt kontrolny

10.15. Zaprojektuj algorytm, który za pomocą pętli For zapisze w pliku liczby od 1 do 10.

- 10.16. Do czego służy funkcja eof?
- 10.17. Czy w programie można odczytać dane znajdujące się poza plikiem?
- 10.18. Co oznacza sytuacja, w której wyrażenie eof(myFile) zwraca wartość True?
- 10.19. Której z przedstawionych poniżej pętli użyjesz, aby odczytać wszystkie elementy zapisane w pliku powiązanym z obiektem myFile?
- `While eof(myFile)  
    Read myFile item  
End While`
  - `While NOT eof(myFile)  
    Read myFile item  
End While`

### 10.3

## Korzystanie z plików i tablic

**WYJAŚNIENIE:** W przypadku niektórych algorytmów można bardzo efektywnie wykorzystać równocześnie pliki i tablice. Bardzo łatwo jest napisać pętlę, która będzie zapisywała zawartość tablicy w pliku albo zawartość pliku w tablicy.

W niektórych programach trzeba zapisać w pliku tablicę w taki sposób, aby można było skorzystać z zapisanych w niej danych w późniejszym czasie. W innych przypadkach trzeba odczytać przechowywane w pliku dane i zapisać je w tablicy. Założmy, że w pliku są zapisane liczby ułożone w przypadkowej kolejności, a Ty chcesz je uporządkować. Jednym z rozwiązań takiego zadania będzie odczytanie liczb zapisanych w pliku, zapisanie ich w tablicy, posortowanie elementów tablicy, a następnie zapisanie uporządkowanych liczb z powrotem do pliku.

Zapisywanie elementów tablicy w pliku jest bardzo proste: wystarczy otworzyć plik i za pomocą pętli odczytać po kolei każdy element tablicy, a potem zapisać go do pliku. Założmy, że w programie występuje taka oto deklaracja tablicy:

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE] = 10, 20, 30, 40, 50
```

Z pomocą poniższego pseudokodu otwieram plik *values.dat* i zapisuję do niego po kolei każdy element tablicy:

```
// Zmienna licznikowa, której użyjemy w pętli
Declare Integer index
// Deklarujemy plik wyjściowy
Declare OutputFile numberFile
// Otwieramy plik values.dat
Open numberFile "values.dat"
// Zapisujemy w pliku po kolei każdy element tablicy
For index = 0 To SIZE - 1
    Write numberFile numbers[index]
End For
// Zamkamy plik
Close numberFile
```

Odczytanie pliku i zapisanie danych w tablicy jest równie proste: otwórz plik i za pomocą pętli odczytaj z niego kolejne wartości, zapisując każdą z nich w kolejnym elemencie tablicy. Pętla powinna zakończyć działanie, gdy tablica będzie już zapełniona lub program dojdzie do końca pliku. Założmy, że w programie występuje taka oto deklaracja tablicy:

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE]
```

Z pomocą poniższego pseudokodu otwieram plik *values.dat* i odczytuję z niego wartości, które następnie zapisuję w tablicy:

```
// Zmienna licznikowa, której użyjemy w pętli, zainicjalizowana
// wartością 0
Declare Integer index = 0
// Deklarujemy plik wejściowy
Declare InputFile numberFile
// Otwieramy plik values.dat
Open numberFile "values.dat"
// Odczytujemy plik i zapisujemy dane w tablicy
While (index <= SIZE - 1) AND (NOT eof(numberFile))
    Write numberFile numbers[index]
    Set index = index + 1
End While
// Zamkamy plik
Close numberFile
```

Zwróć uwagę, że w pętli `While` sprawdzam dwa warunki. Pierwszy z nich to `index <= SIZE - 1` — dzięki niemu pętla nie zapisze danych poza zakresem indeksów tablicy. Kiedy tablica będzie pełna, pętla zakończy działanie. Drugi warunek to `NOT eof(numberFile)` — dzięki niemu pętla nie odczyta wartości spoza pliku. Kiedy pętla dojdzie do końca pliku, także zakończy działanie.

## 10.4

## Przetwarzanie rekordów

**WYJAŚNIENIE:** Dane przechowywane w plikach są często zapisane w formie rekordów. Rekord to kompletny zbiór danych na temat elementu, a pole to jedna, konkretna właściwość elementu zapisanego w rekordzie.

Dane zapisywane do pliku często zorganizowane są w formie rekordów zawierających pola. **Rekord** do kompletny zbiór danych opisujących konkretny element, a **pole** to jedna z właściwości tego elementu. Założmy, że chcesz zapisać w pliku dane na temat pracowników. Plik taki będzie zawierał rekordy opisujące każdego pracownika. Każdy rekord będzie zawierał szereg pól, takich jak imię i nazwisko, identyfikator czy dział. Ilustruje to rysunek 10.17.

**Rysunek 10.17.** Rekord i pola

## Zapisywanie rekordu

Aby zapisać w pliku dany rekord, w pseudokodzie będziemy używali polecenia `Write`. Założymy, że w zmiennych `name`, `idNumber` i `department` zapisane są informacje dotyczące pracownika, a `employeeFile` jest obiektem reprezentującym plik, w którym chcemy zapisać dane. Wartości przypisane do zmiennych możemy wtedy zapisać w pliku za pomocą następującego polecenia:

```
Write employeeFile name, idNumber, department
```

W poleceniu tym po nazwie obiektu reprezentującego plik zapisałem nazwy wszystkich zmiennych i oddzieliłem je przecinkami. Na listingu 10.7 przedstawiłem kompletny program, w którym wykorzystałem to polecenie.

**Listing 10.7**

```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String name
3 Declare Integer idNumber
4 Declare String department
5
6 // Zmienna, w której zapiszemy liczbę rekordów zawierających dane o pracownikach
7 Declare Integer numEmployees
8
9 // Zmienna licznikowa
10 Declare Integer counter
11
12 // Deklarujemy plik wyjściowy
13 Declare OutputFile employeeFile
14
15 // Pobieramy liczbę pracowników
16 Display "Ile rekordów zawierających dane o pracownikach ",
17     "chcesz utworzyć?"
18 Input numEmployees
19
20 // Otwieramy plik o nazwie employees.dat
21 Open employeeFile "employees.dat"
22
23 // Pobieramy dane każdego pracownika
24 // i zapisujemy je w pliku
25 For counter = 1 To numEmployees
  
```

```

26 // Pobieramy imię i nazwisko pracownika
27 Display "Wprowadź imię i nazwisko pracownika nr ", counter
28 Input name
29
30 // Pobieramy identyfikator pracownika
31 Display "Wprowadź identyfikator pracownika."
32 Input idNumber
33
34 // Pobieramy dział pracownika
35 Display "Wprowadź dział pracownika."
36 Input department
37
38 // Zapisujemy rekord do pliku
39 Write employeeFile name, idNumber, department
40
41 // Wyświetlamy pustą linię
42 Display
43 End For
44
45 // Zamkamy plik
46 Close employeeFile
47 Display "Dane pracowników zostały zapisane w pliku employees.dat."

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Ile rekordów zawierających dane o pracownikach chcesz utworzyć?

**3 [Enter]**

Wprowadź imię i nazwisko pracownika nr 1

**Grażyna Matusik [Enter]**

Wprowadź identyfikator pracownika.

**7311 [Enter]**

Wprowadź dział pracownika.

**Księgowość [Enter]**

Wprowadź imię i nazwisko pracownika nr 2

**Robert Jakubiak [Enter]**

Wprowadź identyfikator pracownika.

**8996 [Enter]**

Wprowadź dział pracownika.

**Ochrona [Enter]**

Wprowadź imię i nazwisko pracownika nr 3

**Beata Koroniewska [Enter]**

Wprowadź identyfikator pracownika.

**2301 [Enter]**

Wprowadź dział pracownika.

**Marketing [Enter]**

Dane pracowników zostały zapisane w pliku employees.dat.

W liniach od 16. do 18. proszę użytkownika o wprowadzenie liczby pracowników.

Następnie w pętli pobieram imię i nazwisko, identyfikator oraz dział pracownika.

W linii 39. zapisuję wprowadzone dane do pliku. Pętla wykonuje po jednej iteracji dla każdego pracownika.

W przykładowym wywołaniu programu użytkownik wprowadził dane trzech pracowników. W tabeli na rysunku 10.18 pokazałem, w jaki sposób możesz sobie wyobrazić rekordy zapisane w pliku. Plik zawiera trzy rekordy, po jednym dla każdego pracownika, a każdy rekord składa się z trzech pól.

Imię i nazwisko	Identyfikator	Dział
Grażyna Matusik	7311	Księgowość
Robert Jakubiak	8996	Ochrona
Beata Koroniewska	2301	Marketing

**Rysunek 10.18.** Rekordy zapisane w pliku employees.dat

Sposób, w jaki faktycznie zostaną zapisane rekordy w pliku, różni się w zależności od języka. Wspomniałem wcześniej, że w wielu systemach po każdym elemencie zapisanym w pliku wstawiany jest separator. Na rysunku 10.19 przedstawiłem fragment pliku, w którym po każdym polu pojawia się separator.

Grażyna Matusik	Separator	7311	Separator	Księgowość	Separator	Robert Jakubiak	Separator	8996	...	Itd.
-----------------	-----------	------	-----------	------------	-----------	-----------------	-----------	------	-----	------

**Rysunek 10.19.** Zawartość pliku z uwzględnionymi separatorami pomiędzy poszczególnymi polami

**UWAGA:** W niektórych systemach podczas zapisywania rekordów w pliku pola oddzielane są za pomocą jednego typu separatora, a rekordy za pomocą innego.

## Odczytywanie rekordów

Do odczytania rekordu z pliku w pseudokodzie będziemy używali pojedynczej instrukcji Read. Oto polecenie, za pomocą którego odczytuje trzy wartości z pliku employeeFile i zapisuję je w zmiennych name, idNumber i department:

Read employeeFile name, idNumber, department

Pseudokod zamieszczony na listingu 10.8 przedstawia program, który odczytuje rekordy z pliku utworzonego wcześniej za pomocą programu z listingu 10.7.

### Listing 10.8



```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String name
3 Declare Integer idNumber
4 Declare String department
5
6 // Deklarujemy plik wejściowy
7 Declare InputFile employeeFile
8
9 // Otwieramy plik o nazwie employees.dat
10 Open employeeFile "employees.dat"
11
12 Display "Rekordy zawierające dane pracowników"
13
14 // Wyświetlamy rekordy zapisane w pliku

```

```

15 While NOT eof(employeeFile)
16   // Odczytujemy rekord z pliku
17   Read employeeFile name, idNumber, department
18
19   // Wyświetlamy rekord
20   Display "Imię i nazwisko: ", name
21   Display "Identyfikator: ", idNumber
22   Display "Dział: ", department
23
24   // Wyświetlamy pustą linię
25   Display
26 End For
27
28 // Zamykamy plik
29 Close employeeFile

```

### **Wynik działania programu**

Rekordy zawierające dane pracowników

Imię i nazwisko: Grażyna Matusik

Identyfikator: 7311

Dział: Księgowość

Imię i nazwisko: Robert Jakubiak

Identyfikator: 8996

Dział: Ochrona

Imię i nazwisko: Beata Koroniewska

Identyfikator: 2301

Dział: Marketing

## **Specyfikacja pliku**

Jeśli będziesz pracować jako programista w firmie lub instytucji, najprawdopodobniej będziesz tworzyć między innymi programy odczytujące dane z już istniejących plików. Pliki te będą przeważnie zapisane na firmowym serwerze lub na innym komputerze będącym częścią infrastruktury informatycznej organizacji. W takim przypadku nie będziesz wiedział, w jaki sposób dane są zapisywane w plikach. Właśnie z tego powodu w przedsiębiorstwach tworzy się **specyfikacje plików**. Specyfikacja taka opisuje, jak w danym pliku są zapisane pola i jaki jest ich typ danych. Programista, który nie miał do tej pory styczności z tymi plikami, może zajrzeć do ich specyfikacji i dowiedzieć się, w jaki sposób dane są zapisane w określonym pliku.

Specyfikacje plików mogą być w danej organizacji przechowywane w plikach procesora tekstuowego, w plikach PDF lub w zwykłych plikach tekstowych — w niektórych przypadkach mogą być także dostępne w formie papierowej. Każde przedsiębiorstwo ma swoje zasady dotyczące tego, jak ma wyglądać specyfikacja pliku, ale zawsze będzie ona zawierała informacje, których potrzebuje programista, aby móc przetworzyć w programie dany plik. Na rysunku 10.20 przedstawiłem przykładową specyfikację pliku *employees.dat*, z którego korzystałem w programach z listingów 10.7 i 10.8.

W przedstawionym przykładzie specyfikacja zawiera nazwę pliku, krótki opis zawartości pliku oraz listę pól, z których składa się rekord. Wskazany jest także typ danych każdego pola. Ponadto pola są wymienione w takiej samej kolejności, w jakiej

<b>Nazwa pliku:</b>	employees.dat
<b>Opis:</b>	każdy rekord zawiera informacje dotyczące pracownika
<b>Nazwa pola</b>	<b>Typ danych</b>
Imię i nazwisko	String
Identyfikator	Integer
Dział	String

Rysunek 10.20. Przykładowa specyfikacja pliku

umieszczone są w rekordzie. W tym przypadku pierwsze pole zawiera imię i nazwisko pracownika, drugie pole zawiera identyfikator pracownika, a trzecie — nazwę działu, do którego dany pracownik należy.

## Zarządzanie rekordami

Aplikacje, które zapisują dane za pomocą rekordów zazwyczaj muszą charakteryzować się znacznie większą funkcjonalnością niż zapis i odczyt rekordów. W sekcji „W centrum uwagi” przyjrzymy się algorytmom dodawania rekordów do pliku oraz wyszukiwania, modyfikowania i usuwania określonego rekordu.

### W centrum uwagi

#### Dodawanie i wyświetlanie rekordów



Midnight Coffee Roasters to mała firma zajmująca się importem ziaren kawy z całego świata i palenia ich w celu uzyskania różnych odmian wybornej kawy. Julia jest właścicielką tej firmy i poprosiła Cię o zaprojektowanie kilku programów, dzięki którym będzie mogła zarządzać stanem magazynowym kawy. Po rozmowie z Julią doszedłeś do wniosku, że do zapisywania rekordów dotyczących stanu kawy będziesz potrzebować pliku. Każdy rekord powinien składać się z dwóch pól, w których będą zapisane następujące dane:

- nazwa produktu — ciąg znaków będący nazwą odmiany kawy;
- ilość na stanie — liczba określająca ilość kawy w kilogramach (w postaci liczby rzeczywistej).

Twoje pierwsze zadanie będzie polegało na zaprojektowaniu programu, za pomocą którego będzie można dodawać do pliku nowe rekordy. Na listingu 10.9 przedstawiłem pseudokod tego programu, a na rysunku 10.21 widoczny jest jego schemat blokowy. Zwróć uwagę, że plik wyjściowy otworzyłem w trybie dołączania. Po każdym uruchomieniu programu do pliku zostanie dodany kolejny rekord.

Kolejnym zadaniem będzie zaprojektowanie programu, który wyświetli wszystkie rekordy zapisane w pliku. Na listingu 10.10 widoczny jest pseudokod programu, a na rysunku 10.22 jego schemat blokowy.

**Listing 10.9**

```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String description
3 Declare Real quantity
4
5 // Zmienna sterująca pętlą
6 Declare String another = "T"
7
8 // Deklarujemy plik wyjściowy w trybie dodawania
9 Declare OutputFile AppendMode coffeeFile
10
11 // Otwieramy plik
12 Open coffeeFile "coffee.dat"
13
14 While toUpper(another) == "T"
15     // Pobieramy nazwę produktu
16     Display "Wprowadź nazwę produktu."
17     Input description
18
19     // Pobieramy ilość produktu
20     Display "Wprowadź ilość produktu ",
21         "(w kilogramach)."
22     Input quantity
23
24     // Dобавляем рекорд в файл
25     Write coffeeFile description, quantity
26
27     // Проверяем, есть ли пользовательский ввод для следующего рекорда
28     // следующий рекорд
29     Display "Czy chcesz wprowadzić kolejny rekord?"
30     Display "Jeśli tak, wpisz T, w przeciwnym razie wpisz dowolny inny znak."
31     Input another
32
33     // Wyświetlamy pustą linię
34     Display
35 End While
36
37 // Zamkamy plik
38 Close coffeeFile
39 Display "Dane zostały dopisane do pliku coffee.dat."

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź nazwę produktu.

**Brazilian Dark Roast [Enter]**

Wprowadź ilość produktu (w kilogramach).

**18 [Enter]**

Czy chcesz wprowadzić kolejny rekord?

Jeśli tak, wpisz T, w przeciwnym razie wpisz dowolny inny znak.

**t [Enter]**

Wprowadź nazwę produktu.

**Sumatra Medium Roast [Enter]**

Wprowadź ilość produktu (w kilogramach).

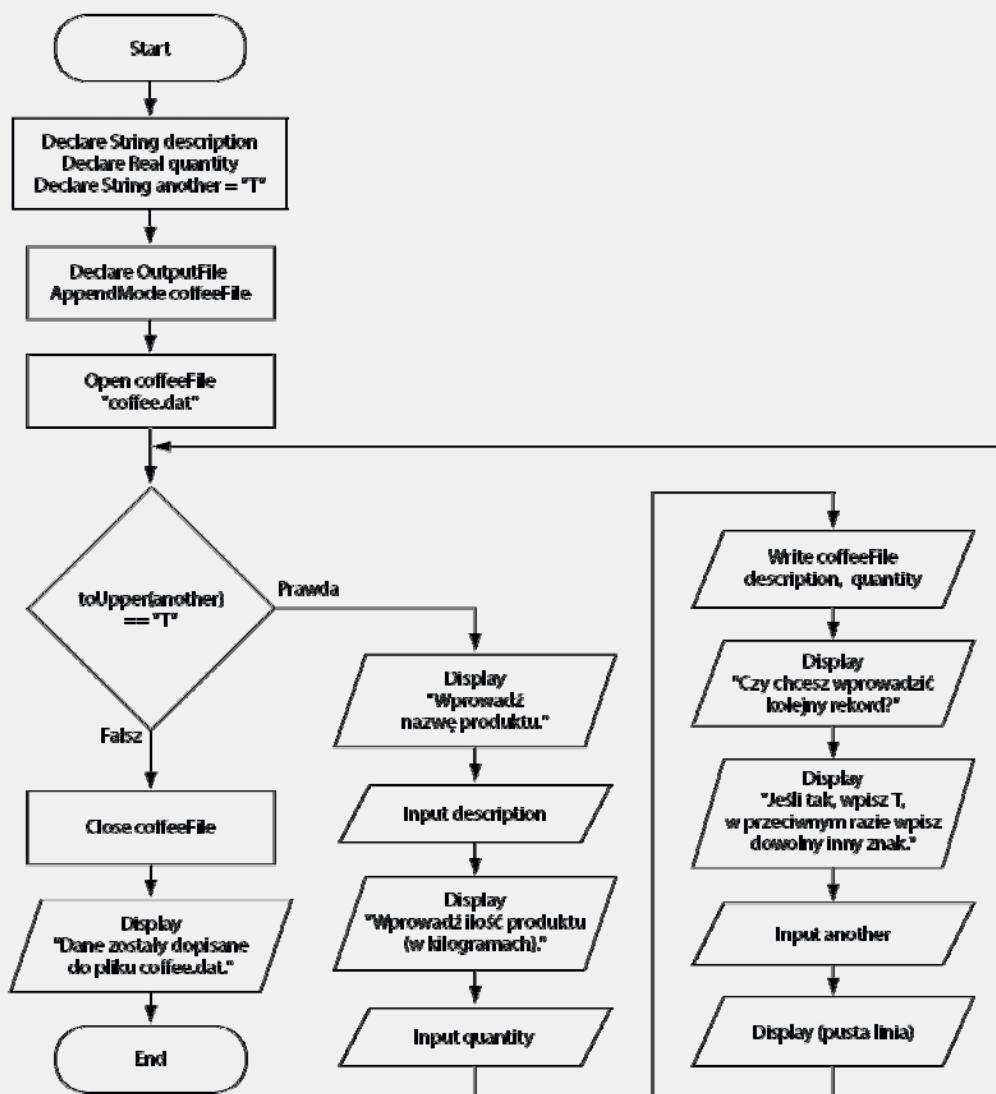
**25 [Enter]**

Czy chcesz wprowadzić kolejny rekord?

Jeśli tak, wpisz T, w przeciwnym razie wpisz dowolny inny znak.

**n [Enter]**

Dane zostały dopisane do pliku coffee.dat.



Rysunek 10.21. Schemat blokowy programu z listingu 10.9

**Listing 10.10**

```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String description
3 Declare Real quantity
4
5 // Deklarujemy plik wejściowy
6 Declare InputFile coffeeFile
7
8 // Otwieramy plik
9 Open coffeeFile "coffee.dat"
10
  
```

```

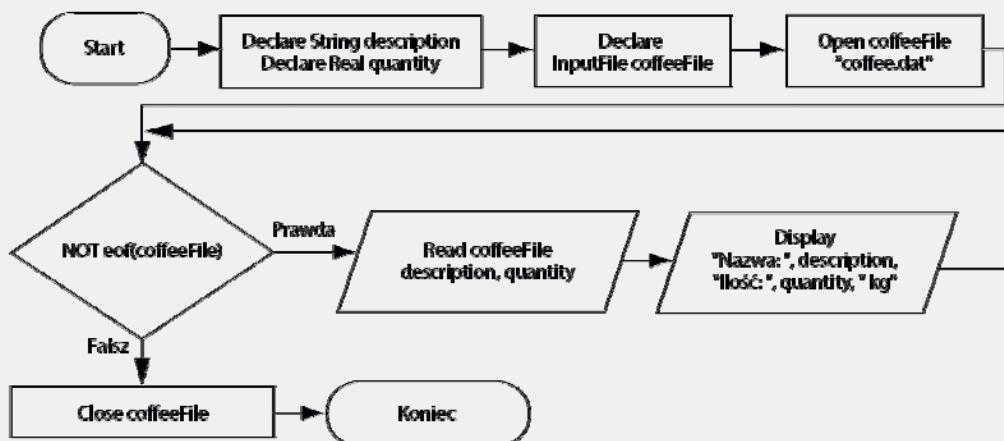
11 While NOT eof(coffeeFile)
12   // Zapisujemy rekord do pliku
13   Read coffeeFile description, quantity
14
15   // Wyświetlamy rekord
16   Display "Nazwa: ", description,
17     "Ilość: ", quantity, " kg"
18 End While
19
20 // Zamkamy plik
21 Close coffeeFile

```

### Wynik działania programu

Nazwa: Brazilian Dark Roast Ilość: 18 kg

Nazwa: Sumatra Medium Roast Ilość: 25 kg



Rysunek 10.22. Schemat blokowy programu z listingu 10.10

## W centrum uwagi

### Wyszukiwanie rekordu

Julia od pewnego czasu korzysta z dwóch programów, które dla niej zaprojektowałeś. W pliku *coffee.dat* jest już więc zapisanych kilka rekordów. Julia poprosiła Cię, abyś tym razem zaprojektował program, który umożliwi jej wyszukanie rekordów. Chce, aby można było wprowadzić nazwę kawy, a program powinien wyświetlić listę rekordów, które zawierają w nazwie wprowadzony ciąg znaków. Założmy, że w pliku znajdują się następujące rekordy:

Nazwa	Ilość
Sumatra Dark Roast	12
Sumatra Medium Roast	30
Sumatra Decaf	20
Sumatra Organic Medium Roast	15

Kiedy Julia wprowadzi tekst „Sumatra”, program powinien wyświetlić wszystkie rekordy z tym słowem. Na listingu 10.11 przedstawiłem pseudokod tego programu, a na rysunku 10.23 widoczny jest jego schemat blokowy.

**Listing 10.11**


```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String description
3 Declare Real quantity
4
5 // Zmienna, w której zapiszemy szukaną wartość
6 Declare String searchValue
7
8 // Flaga wskazująca, czy wartość została odnaleziona
9 Declare Boolean found = False
10
11 // Deklarujemy plik wejściowy
12 Declare InputFile coffeeFile
13
14 // Pobieramy szukaną wartość
15 Display "Której kawy szukasz?"
16 Input searchValue
17
18 // Otwieramy plik
19 Open coffeeFile "coffee.dat"
20
21 While NOT eof(coffeeFile)
22     // Odczytujemy rekord z pliku
23     Read coffeeFile description, quantity
24
25     // Jeśli rekord zawiera szukaną wartość,
26     // wyświetlamy go
27     If contains(description, searchValue) Then
28         // Wyświetlamy rekord
29         Display "Nazwa: ", description,
30             "Ilość: ", quantity, " kg"
31
32     // Ustawiamy flagę na True
33     Set found = True
34 End If
35 End While
36
37 // Jeśli szukanej wartości nie udało się odnaleźć,
38 // wyświetlamy odpowiedni komunikat
39 If NOT found Then
40     Display "Nie znaleziono: ", searchValue
41 End If
42
43 // Zamkamy plik
44 Close coffeeFile

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Której kawy szukasz?

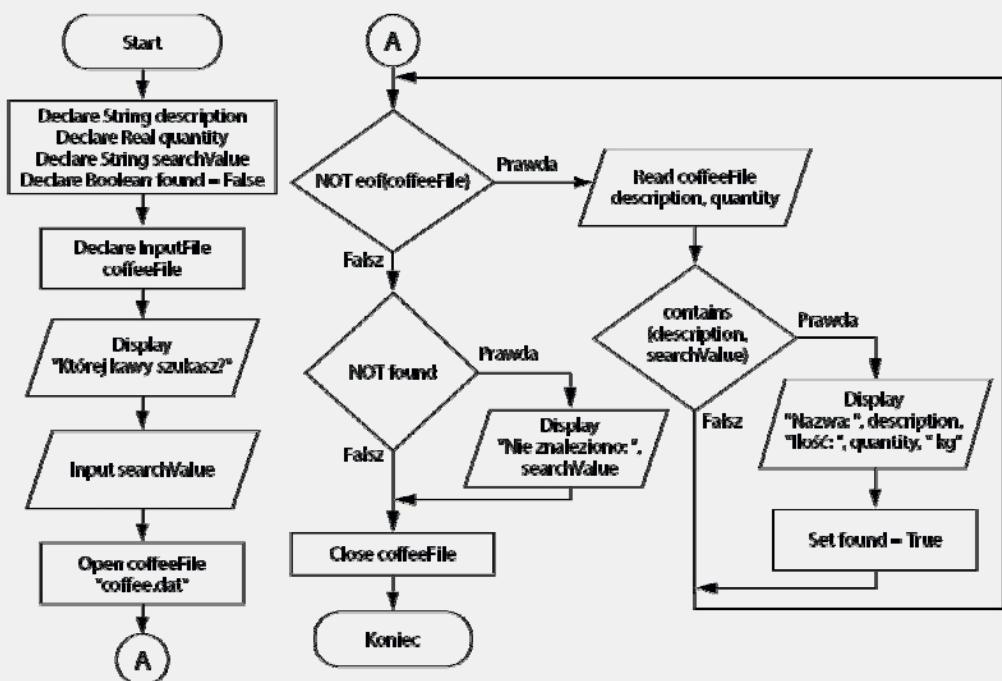
**Sumatra [Enter]**

Nazwa: Sumatra Dark Roast Ilość: 12 kg

Nazwa: Sumatra Medium Roast Ilość: 30 kg

Nazwa: Sumatra Decaf Ilość: 20 kg

Nazwa: Sumatra Organic Medium Roast Ilość: 15 kg



**Rysunek 10.23.** Schemat blokowy programu z listingu 10.11

Zwróc uwagę, że w linii 27. użyłem funkcji `contains`. Przypomnij sobie, że funkcja `contains`, którą omówiłem w rozdziale 6., zwraca wartość `True`, jeżeli ciąg znaków przekazany jako pierwszy argument zawiera ciąg znaków przekazany jako drugi argument.

## W centrum uwagi

### Modyfikowanie rekordów

Julia jest bardzo zadowolona z programów, które dla niej zaprojektowałeś. Twoim kolejnym zadaniem będzie zaprojektowanie programu, za pomocą którego Julia będzie mogła modyfikować wartość w polu *Ilość*. Dzięki temu będzie mogła aktualizować ilość danego rodzaju kawy, gdy zostanie on sprzedany lub gdy się powiększy.

Aby zmodyfikować rekord zapisany w pliku sekwencyjnym, należy utworzyć drugi, tymczasowy plik. Do pliku tymczasowego kopujemy całą zawartość pliku pierwotnego, ale kiedy dojdziemy do rekordu modyfikowanego, zamiast jego pierwotnej wersji zapisujemy wersję zmodyfikowaną. Następnie kopujemy pozostałe rekordy zapisane w pliku pierwotnym.

Na końcu plik pierwotny zastępujemy plikiem tymczasowym — wystarczy usunąć plik pierwotny i zmienić nazwę pliku tymczasowego na taką, jaką miał plik pierwotny. Oto algorytm takiego programu:

1. Otwórz pierwotny plik wejściowy i utwórz tymczasowy plik wyjściowy.
2. Pobierz od użytkownika nazwę kawy, której zapas chcesz zmodyfikować, oraz nową wartość ilości.
3. Dopóki nie dojdiesz do końca pliku pierwotnego:  
    Odczytaj rekord.  
    Jeśli w odczytanym rekordzie wartość w polu z nazwą jest taka sama jak nazwa kawy wprowadzona przez użytkownika:  
        Zapisz w pliku tymczasowym nowe dane.  
    W przeciwnym razie przepisz rekord z pliku pierwotnego do pliku tymczasowego.
4. Zamknij plik pierwotny i plik tymczasowy.
5. Usuń plik pierwotny.
6. Zmień nazwę pliku tymczasowego na taką, jaką miał plik pierwotny.

Zwróć uwagę, że na końcu algorytmu usuwamy plik pierwotny, a następnie zmieniamy nazwę pliku tymczasowego. W większości języków programowania dostępne są polecenia, które wykonują te operacje. W pseudokodzie do usuwania pliku z dysku będziemy używali polecenia Delete. Wystarczy, że wskażesz nazwę pliku, który chcesz usunąć:

`Delete "coffee.dat"`

Aby zmienić nazwę pliku, będziemy korzystali z polecenia Rename. Oto przykład:

`Rename "temp.dat", "coffee.dat"`

Po wykonaniu tego polecenia plik `temp.dat` zmieni nazwę na `coffee.dat`.

Na listingu 10.12 zamieściłem pseudokod programu, a na rysunkach 10.24 i 10.25 widoczny jest jego schemat blokowy.

### Listing 10.12



```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String description
3 Declare Real quantity
4
5 // Zmienna, w której zapiszemy szukaną wartość
6 Declare String searchValue
7
8 // Zmienna, w której zapiszemy nową ilość produktu
9 Declare Real newQuantity
10
11 // Flaga wskazująca, czy wartość została odnaleziona
12 Declare Boolean found = False
13
14 // Deklarujemy plik wejściowy
15 Declare InputFile coffeeFile
16
17 // Deklarujemy plik wyjściowy, do którego skopiujemy
18 // plik pierwotny
19 Declare OutputFile tempFile
20
21 // Otwieramy plik pierwotny
22 Open coffeeFile "coffee.dat"
23
24 // Otwieramy plik tymczasowy

```

```

25 Open tempFile "temp.dat"
26
27 // Pobieramy szukaną wartość
28 Display "Wprowadź nazwę kawy, której ilość chcesz zmodyfikować."
29 Input searchValue
30
31 // Pobieramy nową ilość produktu
32 Display "Wprowadź nową ilość."
33 Input newQuantity
34
35 While NOT eof(coffeeFile)
36   // Odczytujemy rekord z pliku
37   Read coffeeFile description, quantity
38
39   // Gdy rekord zawiera szukaną wartość, zapisujemy w pliku tymczasowym nowy rekord
40   // W przeciwnym razie w pliku tymczasowym zapisujemy
41   // odczytany rekord
42   If description == searchValue Then
43     Write tempFile description, newQuantity
44     Set found = True
45   Else
46     Write tempFile description, quantity
47   End If
48 End While
49
50 // Zamykamy plik pierwotny
51 Close coffeeFile
52
53 // Zamykamy plik tymczasowy
54 Close tempFile
55
56 // Usuwamy plik pierwotny
57 Delete "coffee.dat"
58
59 // Zmieniamy nazwę pliku tymczasowego
60 Rename "temp.dat", "coffee.dat"
61
62 // Wyświetlamy informację, czy operacja się powiodła
63 If found Then
64   Display "Rekord został zaktualizowany."
65 Else
66   Display searchValue, ": nie ma takiej kawy w pliku."
67 End If

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

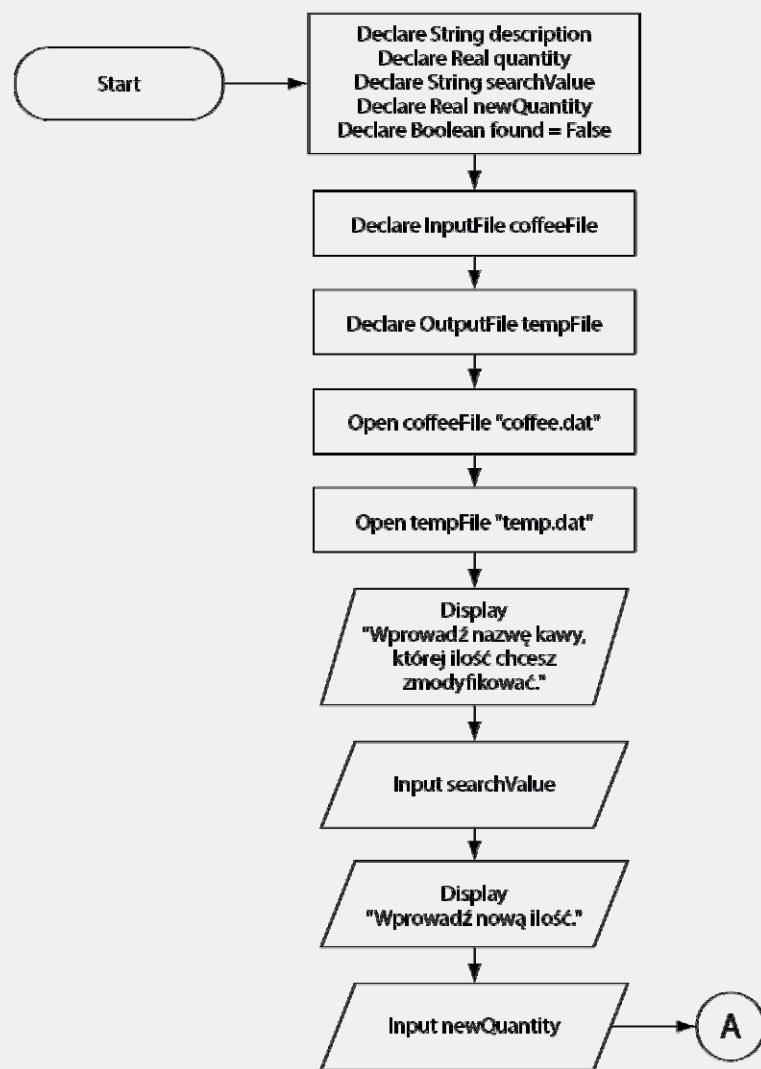
Wprowadź nazwę kawy, której ilość chcesz zmodyfikować.

**Sumatra Medium Roast [Enter]**

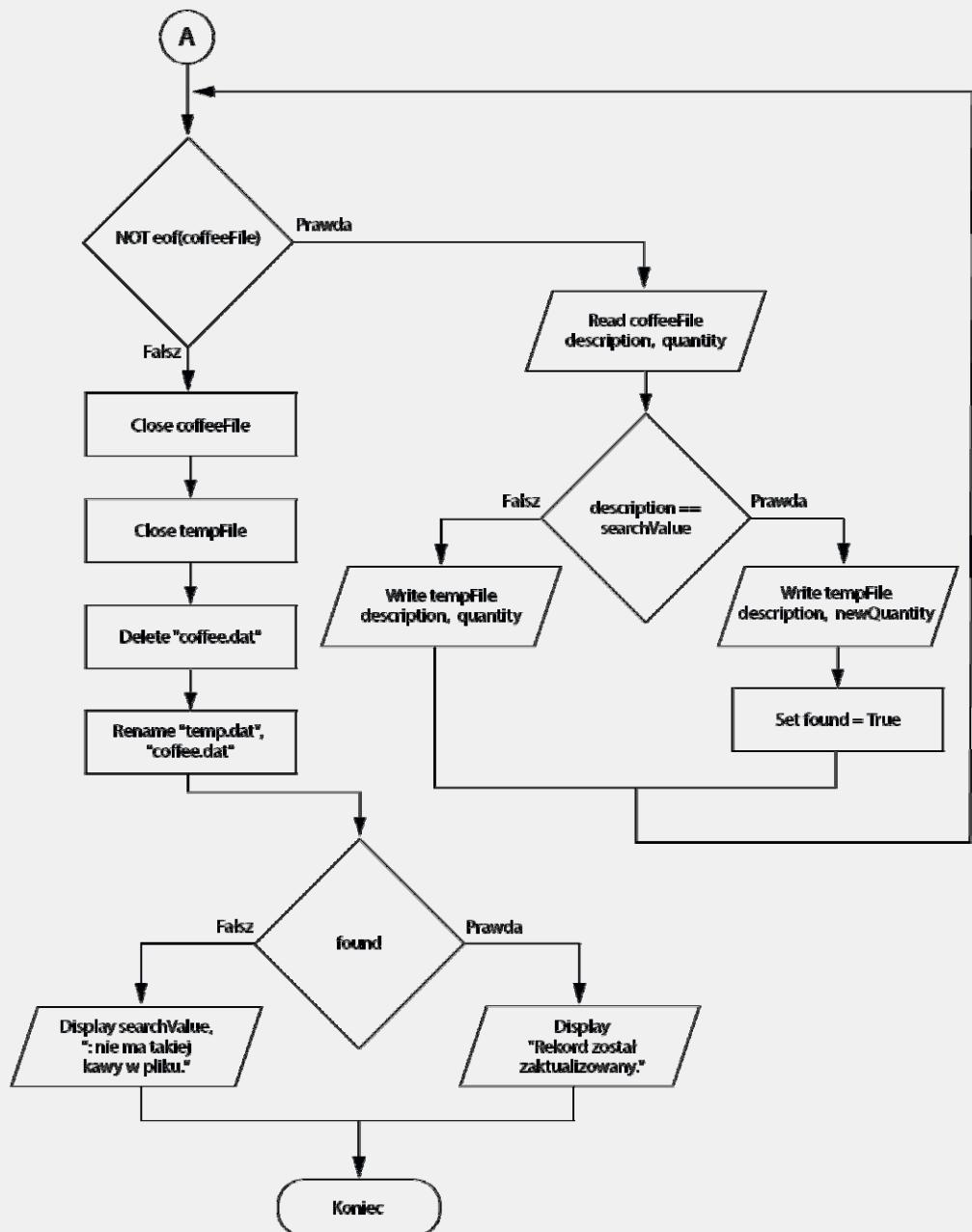
Wprowadź nową ilość.

**18 [Enter]**

Rekord został zaktualizowany.



Rysunek 10.24. Schemat blokowy programu z listingu 10.12, część 1.



Rysunek 10.25. Schemat blokowy programu z listingu 10.12, część 2.



**WSKAZÓWKA:** Jeżeli będziesz tworzyć program w języku, w którym nie ma wbudowanych funkcji służących do usuwania i zmiany nazw plików, możesz po zamknięciu plików pierwotnego i tymczasowego wykonać następujące operacje:

1. Otwórz plik pierwotny w trybie do zapisu (operacja ta spowoduje usunięcie z pliku wszystkich danych).
2. Otwórz plik tymczasowy w trybie do odczytu.
3. Odczytaj po kolejni każdy rekord z pliku tymczasowego i zapisz w pliku pierwotnym (dzięki temu skopujesz wszystkie rekordy z pliku tymczasowego do pliku pierwotnego).
4. Zamknij plik pierwotny i plik tymczasowy.

Wadą tego rozwiązania jest to, że dodatkowe operacje kopiowania pliku tymczasowego do pliku pierwotnego spowodują, że program będzie działał nieco wolniej. Kolejną wadą jest to, że plik tymczasowy po zakończeniu kopiowania pozostanie nadal na dysku. Jeśli plik ten będzie zawierał bardzo dużo danych, należy otworzyć go ponownie w trybie do zapisu, a następnie zamknąć. Operacja ta spowoduje usunięcie danych zapisanych w pliku.



## W centrum uwagi

### Usuwanie rekordów

Twoje ostatnie zadanie będzie polegało na zaprojektowaniu programu, za pomocą którego Julia będzie mogła usuwać rekordy z pliku *coffee.dat*. Podobnie jak w przypadku modyfikowania rekordu, operacja usunięcia rekordu z pliku sekwencyjnego wymaga utworzenia drugiego, tymczasowego pliku. Do pliku tymczasowego kopiujemy wszystkie rekordy z pliku pierwotnego, z wyjątkiem rekordu usuwanego. Następnie musimy zastąpić plik pierwotny plikiem tymczasowym — usuwamy więc plik pierwotny i zmieniamy nazwę pliku tymczasowego na taką, jaką miał plik pierwotny. Oto ogólny algorytm programu:

1. Otwórz pierwotny plik wejściowy i utwórz tymczasowy plik wyjściowy.
2. Pobierz od użytkownika nazwę kawy, którą chce usunąć.
3. Dopóki nie dojdzieś do końca pliku pierwotnego:
  - Odczytaj rekord.
  - Jeśli w odczytanym rekordzie wartość w polu z nazwą nie jest taka sama jak nazwa kawy wprowadzona przez użytkownika:
    - Zapisz w pliku tymczasowym odczytany rekord.
4. Zamknij plik pierwotny i plik tymczasowy.
5. Usuń plik pierwotny.
6. Zmień nazwę pliku tymczasowego na taką, jaką miał plik pierwotny.

Na listingu 10.13 widoczny jest pseudokod programu, a na rysunku 10.26 przedstawiłem jego schemat blokowy.

**Listing 10.13**

```

1 // Zmienne, w których zapiszemy pola rekordu
2 Declare String description
3 Declare Real quantity
4
5 // Zmienna, w której zapiszemy szukaną wartość
6 Declare String searchValue
7
8 // Deklarujemy plik wejściowy
9 Declare InputFile coffeeFile
10
11 // Deklarujemy plik wyjściowy, do którego skopiujemy
12 // plik pierwotny
13 Declare OutputFile tempFile
14
15 // Otwieramy oba pliki
16 Open coffeeFile "coffee.dat"
17 Open tempFile "temp.dat"
18
19 // Pobieramy szukaną wartość
20 Display "Wprowadź nazwę kawy, którą chcesz usunąć."
21 Input searchValue
22
23 While NOT eof(coffeeFile)
24     // Odczytujemy rekord z pliku
25     Read coffeeFile description, quantity
26
27     // Jeśli nie jest to rekord, który mamy zamierzyć usunąć,
28     // zapisujemy go w pliku tymczasowym
29     If description != searchValue Then
30         Write tempFile description, quantity
31     End If
32 End While
33
34 // Zamkamy oba pliki
35 Close coffeeFile
36 Close tempFile
37
38 // Usuwamy plik pierwotny
39 Delete "coffee.dat"
40
41 // Zmieniamy nazwę pliku tymczasowego
42 Rename "temp.dat", "coffee.dat"
43
44 Display "Plik został zaktualizowany."

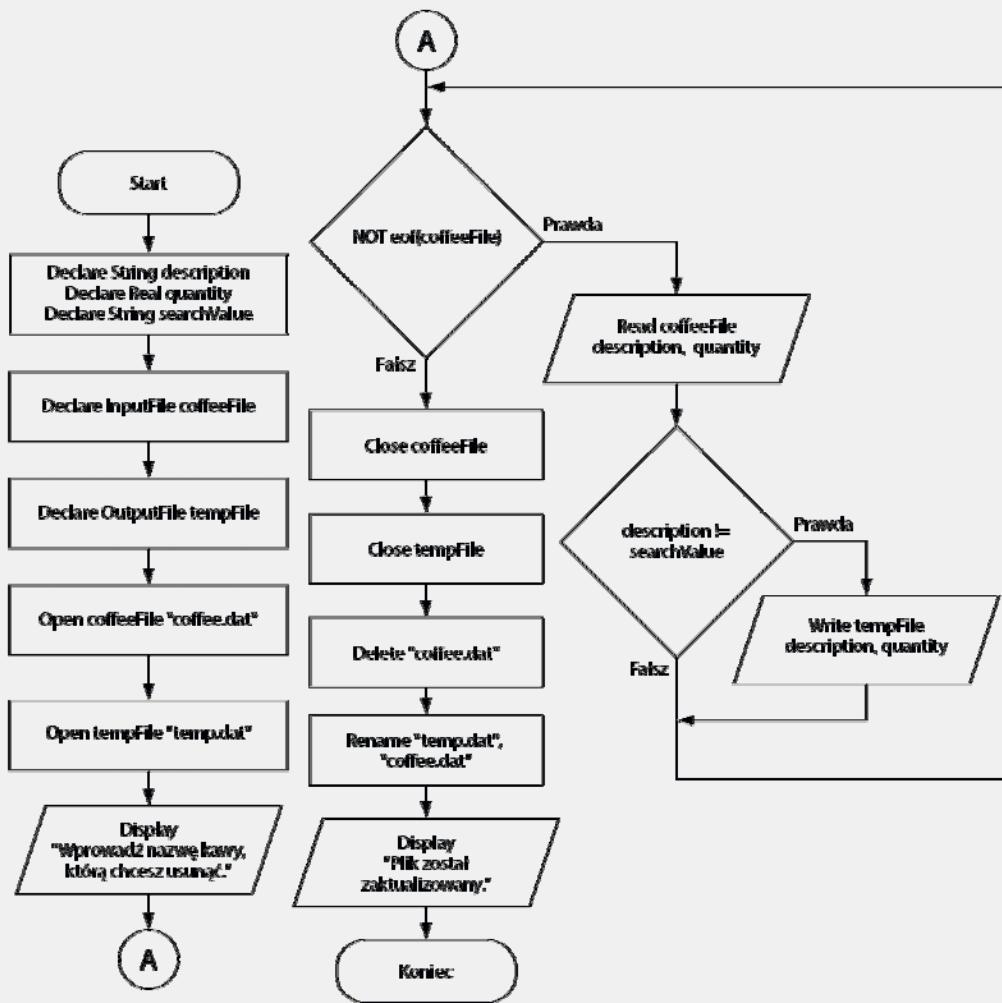
```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź nazwę kawy, którą chcesz usunąć.

**Sumatra Organic Medium Roast [Enter]**

Plik został zaktualizowany.



Rysunek 10.26. Schemat blokowy programu z listingu 10.13



## Punkt kontrolny

- 10.20. Co to jest rekord? Co to jest pole?
- 10.21. Wyjaśnij, w jaki sposób za pomocą pliku tymczasowego można zmodyfikować rekord w pliku z dostępem sekwencyjnym.
- 10.22. Wyjaśnij, w jaki sposób za pomocą pliku tymczasowego można usunąć rekord w pliku z dostępem sekwencyjnym.

**10.5**

## Separatory sterowania

**WYJAŚNIENIE:** Separatory sterowania mają na celu chwilowe przerwanie przetwarzania danych i wykonanie innej operacji w momencie, gdy ulegnie zmianie wartość przypisana do zmiennej sterującej lub gdy w zmiennej pojawi się określona wartość. Po wykonaniu operacji program kontynuuje przetwarzanie danych.

Załóżmy, że pewne przedsiębiorstwo posiada dziewięć sklepów ulokowanych w trzech województwach: trzy w województwie śląskim (ŚL), trzy w województwie wielkopolskim (WP) i trzy w województwie małopolskim (MP). W systemie księgowym przedsiębiorstwa znajduje się plik o nazwie *sales.dat*, w którym zapisane są wartości sprzedaży w poszczególnych sklepach. Każdy rekord w pliku ma następujące pola:

- numer sklepu — liczba typu Integer;
- województwo — ciąg znaków;
- wartość sprzedaży — liczba typu Real.

Oto przykładowe dane zapisane w tym pliku:

Numer sklepu	Województwo	Wartość sprzedaży w złotych
101	ŚL	10000,00 zł
102	ŚL	11000,00 zł
103	ŚL	12000,00 zł
201	WP	7000,00 zł
202	WP	8000,00 zł
203	WP	9000,00 zł
301	MP	12000,00 zł
302	MP	13000,00 zł
303	MP	14000,00 zł

Mamy zamiar napisać program, który wyświetli raport sprzedaży zawierający sumę sprzedaży w poszczególnych województwach. Raport powinien wyglądać następująco:

```
Raport sprzedaży z podziałem na województwa
Nr sklepu    Woj.      Wart. sprzedaży
=====
101          ŚL        10 000,00 zł
102          ŚL        11 000,00 zł
103          ŚL        12 000,00 zł
Suma wart. sprzed. w ŚL: 33 000,00 zł
201          WP        7 000,00 zł
202          WP        8 000,00 zł
203          WP        9 000,00 zł
Suma wart. sprzed. w WP: 24 000,00 zł
301          MP        12 000,00 zł
302          MP        13 000,00 zł
303          MP        14 000,00 zł
Suma wart. sprzed. w MP: 39 000,00 zł
```

Aby uzyskać taki efekt, posłużymy się **separatormi sterowania**. Krótko mówiąc, technika ta polega na tym, że chwilowo przerywamy normalne przetwarzanie danych (np. odczytywanie danych z pliku) w momencie, gdy zmieniąca zmienia wartość. Wykonujemy wtedy jakąś inną operację, a następnie kontynuujemy przetwarzanie danych.

W naszym przykładzie program będzie odczytywał kolejne rekordy z pliku *sales.dat* i zapisywał sumę wartości sprzedaży. Gdy program zauważy, że w danym rekordzie zmieniło się województwo, wyświetli sumę wartości sprzedaży i ustawi sumę z powrotem na 0. Program będzie powtarzał tę operację aż do momentu, gdy odczyta z pliku ostatni rekord.

Separatory sterowania są często używane w programach do drukowania raportów, w których dane są pogrupowane według kategorii. W sekcji „W centrum uwagi” pokażę przykład wykorzystania separatorów sterowania i zaprezentuję nowe polecenie pseudokodu: **Print**. Polecenia **Print** używamy dokładnie tak samo jak polecenia **Display** — różnica polega na tym, że **Print** przesyła dane wyjściowe do drukarki (sam proces przekazywania danych do drukarki wygląda różnie, w zależności od systemu operacyjnego).

## W centrum uwagi

### Korzystanie z separatorów sterowania

Doktor Stankiewicz jest dyrektorką Akademii Handlowej. Zorganizowała ona na uczelni zbiórkę pieniędzy, w której może wziąć udział każdy student. Poprosiła Cię, abyś zaprojektował program, który będzie drukował raport zawierający listę wpłat. W raporcie powinny być ujęte poszczególne wpłaty zebrane przez danego studenta, suma wszystkich wpłat zebranych przez danego studenta oraz suma wpłat zebranych przez wszystkich studentów.

Doktor Stankiewicz przekazała Ci plik *donations.dat*, w którym są zapisane wszystkie dane, jakich będziesz potrzebować, aby wygenerować raport. Na rysunku 10.27 przedstawiłem specyfikację tego pliku.

<b>Nazwa pliku:</b>	<i>donations.dat</i>
<b>Opis:</b>	Zawiera wartości wpłat zebranych przez studenta o danym identyfikatorze
<b>Nazwa pola</b>	<b>Typ danych</b>
Identyfikator studenta	Integer
Kwota	Real

Rysunek 10.27. Specyfikacja pliku *donations.dat*

W pliku znajdują się rekordy odpowiadające poszczególnym wpłatom. Każdy rekord składa się z dwóch pól: identyfikator studenta, który zebrał pieniądze (liczba Integer), oraz zebrana przez niego kwota (liczba Real). Rekordy w pliku zostały już uporządkowane według identyfikatorów.

Oto wygląd przykładowego raportu:

Raport ze zbiórki pieniędzy w Akademii Handlowej	
Ident. studenta	Kwota
104	250,00 zł
104	100,00 zł
104	500,00 zł
Suma wpłat zebranych przez studenta: 850,00 zł	
105	100,00 zł
105	800,00 zł
105	400,00 zł
Suma wpłat zebranych przez studenta: 1 300,00 zł	
106	350,00 zł
106	450,00 zł
106	200,00 zł
Suma wpłat zebranych przez studenta: 1 000,00 zł	
Suma wszystkich wpłat: 3 150,00 zł	

Na listingu 10.14 przedstawiłem pseudokod programu. Przyjrzyjmy się modułom `main` i `printHeader`.

**Listing 10.14. Program generujący raport ze zbiórki pieniędzy:  
moduły main i printHeader**

```

1 Module main()
2   //Drukujemy nagłówek raportu
3   Call printHeader()
4
5   //Drukujemy treść raportu
6   Call printDetails()
7 End Module
8
9 //Moduł printHeader drukuje nagłówek raportu
10 Module printHeader()
11   Print "Raport ze zbiórki pieniędzy w Akademii Handlowej"
12   Print
13   Print "Ident. studenta      Kwota"
14   Print "====="
15 End Module
16

```

W module `main`, w linii 3., wywołuję moduł `printHeader`, którego zadaniem jest wydrukowanie nagłówka raportu. Następnie, w linii 6., wywołuję moduł `printDetails`, którego zadaniem jest wydrukowanie treści raportu. Poniżej zamieściłem pseudokod modułu `printDetails`.

**Listing 10.14.** Program generujący raport ze zbiórki pieniędzy (kontynuacja): moduł printDetails

```

17 // Moduł printDetails drukuje treść raportu
18 Module printDetails()
19     // Zmienne przeznaczone na pola rekordu
20     Declare Integer studentID
21     Declare Real donation
22
23     // Zmienne pełniące role akumulatorów
24     Declare Real studentTotal = 0
25     Declare Real total = 0
26
27     // Zmienna, którą wykorzystamy w mechanizmie
28     // separatorka sterowania
29     Declare Integer currentID
30
31     // Deklarujemy plik wejściowy i otwieramy go
32     Declare InputFile donationsFile
33     Open donationsFile "donations.dat"
34
35     // Odczytujemy pierwszy rekord
36     Read donationsFile studentID, donation
37
38     // Zapisujemy identyfikator studenta
39     Set currentID = studentID
40
41     // Drukujemy treść raportu
42     While NOT eof(donationsFile)
43         // Sprawdzamy, czy zmieniła się wartość
44         // w polu studentID
45         If studentID != currentID Then
46             // Drukujemy sumę wpłat zebranych przez studenta,
47             // a następnie pustą linię
48             Print "Suma wpłat zebranych przez studenta: ",
49                 currencyFormat(studentTotal)
50             Print
51
52         // Zapisujemy identyfikator kolejnego studenta
53         Set currentID = studentID
54
55         // Zerujemy wartość akumulatora
56         Set studentTotal = 0
57     End If
58
59     // Drukujemy informacje dotyczące wpłat
60     Print studentID, Tab, currencyFormat(donation)
61
62     // Aktualizujemy wartości akumulatorów
63     Set studentTotal = studentTotal + donation
64     Set total = total + donation
65
66     // Odczytujemy kolejny rekord
67     Read donationsFile studentID, donation
68 End While
69
70     // Drukujemy sumę wpłat zebranych przez ostatniego studenta
71     Print "Suma wpłat zebranych przez studenta: ",
72                 currencyFormat(studentTotal)
73
74     // Drukujemy sumę wszystkich wpłat

```

```

75     Print "Suma wszystkich wpłat: ",
76         currencyFormat(total)
77
78 // Zamkamy plik
79 Close donationsFile
80 End Module

```

Przyjrzyjmy się bliżej modułowi `printDetails`. Oto informacje na temat zadeklarowanych zmiennych:

- W liniach 20. i 21. deklaruję zmienne `studentID` i `donation`, w których zapiszę pola rekordów odczytanych w pliku.
- W liniach 24. i 25. deklaruję zmienne `studentTotal` i `total`. Zmienna `studentTotal` jest akumulatorem, w którym zapiszę sumaryczną kwotę zebraną przez danego studenta. Zmienna `total` jest akumulatorem, w którym zapiszę sumę wszystkich wpłat zebranych przez wszystkich studentów.
- W linii 29. deklaruję zmienną `currentID`. Zapiszę w niej identyfikator studenta, dla którego będę w danej chwili obliczał sumę wpłaconych kwot.
- W linii 32. deklaruję zmienną `donationsFile`, która jest obiektem reprezentującym plik `donations.dat`.

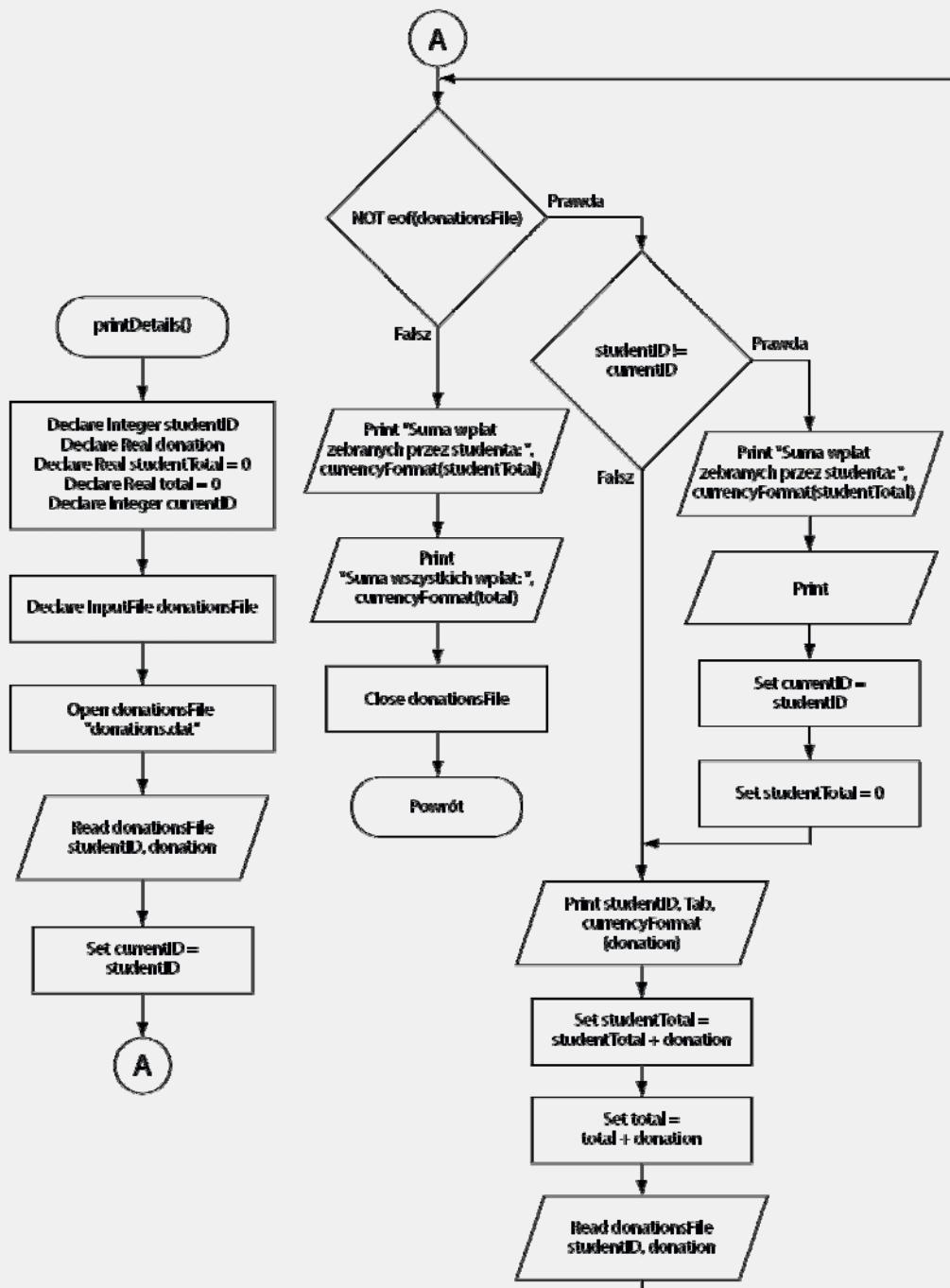
W linii 33. otwieram plik `donations.dat`, a w linii 36. odczytuję z niego pierwszy rekord. Odczytane wartości zapisuję w zmiennych `studentID` i `donation`.

W linii 39. przypisuję do zmiennej `currentID` identyfikator studenta odczytany z pliku. Zmienna ta będzie zawierała identyfikator studenta, którego rekordy będę przetwarzał w danym momencie.

W linii 42. rozpoczyna się pętla, za pomocą której przetwarzam dane zapisane w pliku. Polecenie `If` znajdujące się w liniach od 45. do 57. implementuje technikę separatorów sterowania. Za pomocą polecenia `If` sprawdzam, czy zmienna sterująca `studentID` jest różna od zmiennej `currentID`. Jeśli te zmienne są różne, to znaczy, że program odczytał właśnie rekord dotyczący studenta o innym identyfikatorze niż ten zapisany w zmiennej `currentID`. Informuje nas to o tym, że zakończyliśmy odczytywanie rekordów studenta o identyfikatorze zapisanym w zmiennej `currentID` i możemy chwilowo przerwać przetwarzanie danych i wyświetlić informację o sumie wpłat zebranych przez studenta (w liniach 48. i 49.), następnie zapisać w zmiennej `currentID` identyfikator kolejnego studenta (linia 53.) i wyzerować wartość akumulatora `studentTotal` (linia 56.).

W linii 60. drukuję zawartość aktualnego rekordu. W liniach 63. i 64 aktualizuję wartości akumulatorów. W linii 67. odczytuję z pliku kolejny rekord. Po przetworzeniu wszystkich rekordów w liniach 71. i 72. wyświetlам sumę wpłat zebranych przez ostatniego studenta, w liniach 75. i 76. wyświetlam sumę wpłat wszystkich studentów, a w linii 79. zamknię plik. Raport wydrukowany przez program będzie wyglądał podobnie do przykładowego raportu, który przedstawiłem wcześniej.

Na rysunku 10.28 przedstawiłem schemat blokowy modułu `printDetails`.



Rysunek 10.28. Schemat blokowy modułu printDetails



**UWAGA:** W algorytmie przedstawionego wyżej programu założyłem, że rekordy w pliku *donations.dat* zostały już posortowane według identyfikatorów studentów. Jeśli rekordy nie byłyby uporządkowane, program nie wyświetliłby poprawnie sumy wpłat zebranych przez każdego studenta.

## Schematy rozmieszczenia znaków

Przy tworzeniu programów drukujących raporty na kartce papieru pomocne okazują się **schematy rozmieszczenia znaków**, za pomocą których można zaprojektować wygląd raportu. Schemat rozmieszczenia znaków ma postać kartki papieru z naniesioną siatką, podobnie jak w zeszycie w kratkę. Na rysunku 10.29 przedstawiłem przykład takiego schematu. Każda kratka na schemacie oznacza miejsce na pojedynczy znak. Cyfry umieszczone na górze i z lewej strony kartki ułatwiają odmierzanie odstępów. Wystarczy, że wypełnisz kratki odpowiednim tekstem. Następnie, pisząc program, możesz wykorzystać taki schemat, aby określić, w którym miejscu ma się pojawić określona informacja, ile znaków spacji należy wstawić pomiędzy kolejnymi elementami raportu itp.

**Rysunek 10.29.** Schemat rozmieszczenia znaków

Na rysunku 10.29 widoczny jest schemat rozmieszczenia znaków dla programu generującego raport ze zbiórki pieniężny z listingu 10.14. Zwróć uwagę, że nagłówek raportu oraz inne, niezmieniające się teksty umieszczone są na schemacie dokładnie tak, jak wygląda to na wygenerowanym raporcie. Miejsce, w którym program ma wydrukować identyfikatory studentów i kwoty wpłat, zaznaczyłem na schemacie cyframi 9.



**UWAGA:** Aby zaznaczyć na schemacie miejsca, w których pojawiają się zmienne informacje, można zamiast cyfr 9 stosować znak X. Przyjęło się, że miejsca, w których ma pojawić się liczba, oznaczamy cyfrą 9, a miejsca, w których ma pojawić się litera — znakiem X.

## 10.6

# Rzut oka na języki Java, Python i C++

W niniejszym podręczniku omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawcy: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

## Java

### Pliki

#### Otwieranie pliku i zapisywanie w nim danych za pomocą języka Java

W celu rozpoczęcia pracy z plikami trzeba umieścić w kodzie na początku programu w Javie następującą instrukcję:

```
import java.io.*;
```

Następnie w metodzie, w której chcemy otworzyć plik i zapisać do niego dane, tworzymy obiekt `PrintWriter`. Oto przykład instrukcji, która tworzy ten obiekt:

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```

Ta instrukcja wykonuje następujące czynności:

- Tworzy obiekt `PrintWriter` w pamięci. Nazwa obiektu `PrintWriter` to `outputFile`. Wykorzystamy go w naszym programie do zarządzania plikiem.
- Otwiera na dysku plik o nazwie `StudentData.txt`.

(Zwrót uwagi, że jeśli plik o nazwie `StudentData.txt` nie istnieje, instrukcja ta utworzy go, natomiast jeśli plik już istnieje, jego zawartość zostanie usunięta. Tak czy inaczej, po wykonaniu tej instrukcji na dysku zostanie utworzony pusty plik o nazwie `StudentData.txt`).

Po utworzeniu obiektu `PrintWriter` i otwarciu pliku można zapisać do niego dane za pomocą metody `println`. Wiesz już, jak używać funkcji `println` z przestrzeni nazw `System.out`, służącej do wyświetlania danych na ekranie. W taki sam sposób można jej użyć z obiektem `PrintWriter`, aby zapisać dane do pliku. Na przykład, zakładając, że `outputFile` jest obiektem typu `PrintWriter`, poniższa instrukcja zapisuje do pliku ciąg znaków "Jan":

```
outputFile.println("Jan");
```

Zakładając, że `payRate` jest zmienną, poniższa instrukcja zapisuje wartość tej zmiennej do pliku:

```
outputFile.println(payRate);
```

### Zamykanie pliku w języku Java

Oto przykład wywołania metody `close` w celu zamknięcia pliku, przy założeniu, że `outputfile` jest nazwą obiektu `PrintWriter`:

```
outputFile.close();
```

Po zamknięciu pliku połączenie między nim a obiektem `PrintWriter` zostanie usunięte. Aby wykonać dalsze operacje na tym pliku, należy go ponownie otworzyć.

Na listingu 10.15 można zobaczyć, jak utworzyć obiekt `PrintWriter` (i otworzyć plik do zapisu), zapisać do niego wybrane dane, po czym go zamknąć. Jest to wersja pseudokodu pokazanego już wcześniej na listingu 10.1 napisana dla języka Java.

#### **Listing 10.15 (FileWriteDemo.java)**

```
1 import java.io.*;
2
3 public class FileWriteDemo
4 {
5     public static void main(String[] args) throws IOException
6     {
7         // Deklarujemy zmienną PrintWriter o nazwie myFile
8         // i otwieramy plik o nazwie philosophers.dat
9         PrintWriter myFile = new PrintWriter("philosophers.dat");
10
11        // Zapisujemy w pliku nazwiska trzech filozofów
12        myFile.println("John Locke");
13        myFile.println("David Hume");
14        myFile.println("Edmund Burke");
15
16        // Zamkamy plik
17        myFile.close();
18    }
19 }
```

Zauważ, że w wierszu 5. nagłówek dla metody `main` kończy się klauzulą wyjątku `throws IOException`. Jest to wymagane z powodu zaawansowanego mechanizmu obsługi błędów w Javie. Nie musisz się martwić szczegółami klauzuli, ale trzeba pamiętać, że wymagana jest ona dla każdej metody, która używa technik związanych z pracą na plikach pokazanych w tym podrozdziale.

Po uruchomieniu tego programu wiersz 9. tworzy na dysku plik o nazwie `philosophers.dat`, a wiersze od 12. do 14. zapisują do tego pliku ciągi znaków „John Locke”, „David Hume” i „Edmund Burke”. Następnie w wierszu 17. plik jest zamknięty.

## Otwieranie pliku i odczytywanie z niego danych za pomocą języka Java

Obiekt Scanner można użyć do odczytania danych nie tylko z klawiatury, ale również z pliku. Najpierw potrzebujemy poniższej instrukcji na początku naszego programu:

```
import java.util.Scanner;
```

Następnie wewnątrz metody, która wymaga odczytania danych z pliku, tworzymy obiekt Scanner i łączymy go z tym plikiem. Oto przykład:

```
Scanner inputFile = new Scanner (new File ("StudentData.txt"));
```

Powyzsza instrukcja tworzy w pamięci obiekt Scanner o nazwie `inputFile`. Plik `StudentData.txt` jest otwierany do odczytu, a obiekt Scanner jest do niego podłączony. Po wykonaniu tej instrukcji będzie można użyć obiektu Scanner do odczytania danych z tego pliku.

Po podłączeniu obiektu Scanner do pliku można wykorzystać metodę `nextLine`, aby odczytać znajdujący się w nim ciąg znaków, metodę `nextInt`, aby odczytać liczbę całkowitą, lub metodę `nextDouble`, aby odczytać liczbę typu `double`.

Program na listingu 10.16 pokazuje przykład, który odczytuje ciągi znaków z pliku. Program ten otwiera plik o nazwie `philosophers.dat`, który został utworzony przez program z listingu 10.1. Jest to wersja programu w pseudokodzie z listingu 10.2 napisana dla języka Java. Oto kilka ważnych punktów dotyczących tego programu:

- Wiersz 14. tworzy obiekt Scanner o nazwie `myFile`, otwiera plik na dysku o nazwie `philosophers.dat`, a następnie łączy obiekt z plikiem.
- Wiersz 18. odczytuje wiersz tekstu z pliku i przypisuje go do zmiennej `name1`.
- Wiersz 19. odczytuje następny wiersz tekstu z pliku i przypisuje go do zmiennej `name2`.
- Wiersz 20. odczytuje następny wiersz tekstu z pliku i przypisuje go do zmiennej `name3`.
- Wiersz 29. zamyka plik.

### **Listing 10.16**

```
1 import java.io.*;
2 import java.util.Scanner;
3
4 public class FileReadDemo
5 {
6     public static void main(String[] args) throws IOException
7     {
8         // Deklarujemy trzy zmienne, w których zapiszemy nazwiska
9         // odczytane z pliku
10        String name1, name2, name3;
11
12        // Deklarujemy zmienną typu Scanner o nazwie myFile
13        // i otwieramy plik philosophers.dat
14        Scanner myFile = new Scanner(new File("philosophers.dat"));
15
16        // Odczytujemy nazwiska trzech filozofów
```

```

17     // i zapisujemy je w zmiennych
18     name1 = myFile.nextLine();
19     name2 = myFile.nextLine();
20     name3 = myFile.nextLine();
21
22     // Wyświetlamy odczytane nazwiska
23     System.out.println("Oto imiona i nazwiska trzech filozofów:");
24     System.out.println(name1);
25     System.out.println(name2);
26     System.out.println(name3);
27
28     // Zamkamy plik
29     myFile.close();
30 }
31 }
```

### **Wynik działania programu**

Oto imiona i nazwiska trzech filozofów:

John Locke  
David Hume  
Edmund Burke

### **Dołączanie danych do istniejącego pliku w języku Java**

Czasami chcesz otworzyć istniejący już plik do zapisu i zachować jego zawartość w taki sposób, aby nowe dane mogły zostać dołączone do istniejącej już zawartości. Wówczas należy utworzyć obiekt `PrintWriter` za pomocą następującej instrukcji:

```
PrintWriter outputFile =
    new PrintWriter(new FileWriter("MyFriends.txt", true));
```

Instrukcja ta otwiera istniejący już plik `MyFriends.txt`. Jeśli użyjemy metody `println` do zapisania danych do tego pliku, jego zawartość nie zostanie usunięta, a dane te zostaną dołączone na końcu tego pliku.

### **Wykrywanie końca pliku w języku Java**

Czasami musisz odczytać zawartość pliku, nie znając liczby zapisanych w nim elementów. W takiej sytuacji możesz skorzystać z metody `hasNext` obiektu `Scanner`, aby sprawdzić, czy w pliku znajduje się jeszcze kolejny element danych. Jeśli istnieje więcej danych, które można odczytać, to metoda `hasNext` zwraca wartość `true`. Po dotarciu do końca pliku, gdzie nie ma już więcej danych do odczytania, metoda zwróci wartość `false`.

Na przykład poniższy fragment kodu otwiera plik o nazwie `sales.txt`, a następnie odczytuje każdy wiersz tego pliku i wyświetla go na ekranie:

```
Scanner salesFile = new Scanner(new File("sales.dat"));
while (salesFile.hasNext())
{
    sales = salesFile.nextDouble();
    System.out.println(sales);
}
salesFile.close();
```

## Python

### Pliki

#### Otwieranie pliku za pomocą języka Python

W Pythonie, aby otworzyć plik, wykorzystujemy funkcję `open`. Tworzy ona obiekt pliku i kojarzy go z plikiem istniejącym już na dysku. Oto ogólny format użycia funkcji `open`:

```
zmienna_pliku = open(nazwa_pliku, tryb)
```

Oto kilka zasad dla języka Python:

- `zmienna_pliku` to nazwa zmiennej, w której znajdzie się obiekt pliku.
- `nazwa_pliku` jest ciągiem znaków określającym nazwę pliku.
- `tryb` to ciąg znaków określający tryb pracy (odczyt, zapis itp.), w którym plik zostanie otwarty. W tabeli 10.1 przedstawiam trzy ciągi znaków, których można użyć do określenia trybu pracy z plikiem. (Istnieją także inne, bardziej złożone tryby pracy, ale tych, które zamieściłem w tabeli 10.1, będziemy używać w tej książce).

**Tabela 10.1** Niektóre tryby plików Pythona

Tryb pracy	Opis działania
'r'	Otwórz plik w trybie tylko do odczytu. Nie można zmienić ani zapisać pliku.
'w'	Otwórz plik w trybie do zapisu. Jeśli plik już istnieje, usuń jego zawartość. Jeśli nie istnieje, utwórz go.
'a'	Otwórz plik w trybie do zapisu. Wszystkie dane zapisane w pliku zostaną dołączone na końcu tego pliku. Jeśli plik nie istnieje, utwórz go.

Założymy na przykład, że plik `customers.txt` zawiera dane klientów i chcemy go otworzyć w trybie tylko do odczytu. Oto przykład, jak wywołalibyśmy funkcję `open`:

```
customer_file = open('cusomters.txt', 'r')
```

Po wykonaniu tej instrukcji plik `customers.txt` zostanie otwarty, a zmiennej `customer_file` zostanie przypisany obiekt pliku, którego możemy użyć do odczytania danych.

Założymy, że chcemy utworzyć plik o nazwie `sales.txt` i zapisać do niego dane. W tym celu wywołujemy funkcję `open`:

```
sales_file = open('sales.txt', 'w')
```

Po wykonaniu tego polecenia zostanie utworzony plik o nazwie `sales.txt`, a zmiennej `sales_file` zostanie przypisany obiekt pliku, którego możemy użyć do zapisu danych. (Pamiętaj, że podczas pracy w trybie '`w`' tworzymy plik na dysku. Jeżeli w chwili tworzenia pliku o określonej nazwie taki już istnieje, to zawartość istniejącego pliku zostanie usunięta).

### Zapisywanie danych do pliku w języku Python

Chcąc zapisać nowe dane do pliku otwartego w trybie do zapisu, wykorzystujemy metodę `write` obiektu pliku. Oto ogólny format wywołania tej metody:

```
zmienna_pliku.write(ciąg_znaków)
```

W takim formacie `zmienna_pliku` jest zmienną, która odwołuje się do obiektu pliku, a `ciąg_znaków` jest ciągiem, który zostanie zapisany do pliku. Plik taki musi być otwarty w trybie do zapisu (przy użyciu trybu pracy '`w`' lub '`a`') albo wystąpi błąd.

Załóżmy, że w zmiennej `customer_file` znajduje się obiekt pliku, a plik został otworzony w trybie do zapisu z wykorzystaniem znacznika '`w`'. Oto przykład, w jaki sposób można zapisać do takiego pliku ciąg znaków „Konrad Grzesiak”:

```
customer_file.write('Konrad Grzesiak')
```

Oto inny przykład:

```
name = 'Konrad Grzesiak'
customer_file.write(name)
```

Druga instrukcja zapisuje wartość znajdująca się w zmiennej `name` do pliku powiązanego ze zmienną `customer_file`. W tym przypadku zapisałaby do pliku ciąg znaków „Konrad Grzesiak”. (Te przykłady pokazują, że oprócz zapisu ciągu znaków do pliku można także zapisywać wartości liczbowe).

### Zamykanie pliku w języku Python

W Pythonie w celu zamknięcia pliku wykorzystamy metodę `close` obiektu pliku. Na przykład poniższa instrukcja zamyka plik powiązany ze zmienną `customer_file`:

```
customer_file.close()
```

Po zamknięciu pliku połączenie między nim a obiektem pliku zostanie usunięte. Aby wykonać dalsze operacje na pliku, należy go ponownie otworzyć.

Program przedstawiony na listingu 10.17 pokazuje napisany w Pythonie kompletny program, który otwiera plik wyjściowy, zapisuje w nim dane, a następnie go zamyka. Jest to wersja programu pseudokodu z listingu 10.1 napisana w języku Python.

#### **Listing 10.17 (file\_write\_demo.py)**

```
1 # Ten program zapisuje trzy wiersze danych
2 # do pliku
3 def main():
4     # Otwieramy plik philosophers.txt
5     outfile = open('philosophers.txt', 'w')
6
7     # Zapisujemy w pliku nazwiska trzech filozofów
8     outfile.write('John Locke\n')
9     outfile.write('David Hume\n')
10    outfile.write('Edmund Burke\n')
11
12    # Zamykamy plik
13    outfile.close()
```

```

14
15 # Wywołujemy funkcję main
16 main()

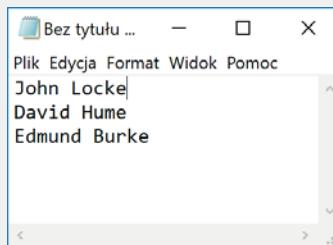
```

Po uruchomieniu tego programu wiersz 5. kodu utworzy na dysku plik o nazwie *philosophers.dat*, a wiersze od 8. do 10. zapiszą w nim ciągi znaków „John Locke\n”, „David Hume\n” i „Edmund Burke\n”. Wiersz 13. zamyka plik.

### Zapis znaków końca wiersza w pliku w języku Python

Zwróć uwagę na użycie znaku \n, który pojawia się wewnątrz ciągów zapisywanych do pliku w wierszach 8., 9. i 10. Sekwencja znaków \n jest znana jako znak modyfikacji. **Znak modyfikacji** jest specjalnym znakiem poprzedzonym lewym ukośnikiem (\), pojawiającym się wewnątrz ciągu znaków. Gdy ciąg znaków zawierający znaki modyfikacji jest wyświetlany na ekranie lub zapisywany do pliku, znaki te są traktowane jako specjalne polecenia osadzone wewnątrz ciągu znaków.

Sekwencja znaków \n jest znakiem **nowego wiersza** i służy do oznaczania miejsca, w którym rozpoczyna się nowy wiersz tekstu w pliku. Jeśli otworzysz plik w edytorze tekstu, to możesz się przekonać, jak działa taki znak. Na przykład na rysunku 10.30 pokazano plik *philosophers.txt* wyświetlony w Notatniku.



**Rysunek 10.30.** Zawartość pliku *philosophers.txt* wyświetlona w Notatniku (dzięki uprzejmości Microsoft Corporation)

### Otwieranie pliku i czytanie danych przy użyciu języka Python

Jeśli plik został otwarty tylko do odczytu (za pomocą trybu 'r'), można użyć metody `readline` z obiektu pliku do odczytu wiersza tekstu z pliku. Metoda ta zwraca wiersz jako串 znaków, w tym znak \n znajdujący się na końcu wiersza. Program widoczny na listingu 10.18 pokazuje, jak można użyć metody `readline` do odczytania zawartości pliku *philosophers.txt*, odczytując kolejno po jednym wierszu. (Jest to wersja programu pseudokodu z listingu 10.2 napisana w języku Python).

#### **Listing 10.18 (file\_read\_demo.py)**

```

1 # Ten program odczytuje zawartość pliku
2 # philosophers.txt po jednym wierszu
3 def main():
4     # Otwieramy plik o nazwie philosophers.txt

```

```

5   infile = open('philosophers.txt', 'r')
6
7   # Odczytujemy trzy wiersze z pliku
8   line1 = infile.readline()
9   line2 = infile.readline()
10  line3 = infile.readline()
11
12  # Zamkamy plik
13  infile.close()
14
15  # Wpisujemy teksty odczytane z pliku
16  print('Oto imiona i nazwiska trzech filozofów:')
17  print(line1)
18  print(line2)
19  print(line3)
20
21 # Wywołujemy funkcję main
22 main()

```

### **Wynik działania programu**

Oto imiona i nazwiska trzech filozofów:

John Locke

David Hume

Edmund Burke

Instrukcja w wierszu 5. otwiera plik *philosophers.txt* tylko do odczytu, używając trybu 'r'. Tworzy również obiekt pliku i przypisuje go do zmiennej typu `infile`. Kiedy plik jest otwierany do odczytu, jest dla niego tworzona specjalna wewnętrzna wartość znana jako **pozycja odczytu**. Pozycja ta wskazuje położenie następnego elementu, który zostanie odczytany z pliku. Początkowo pozycja odczytu jest ustawiona na początek pliku.

W celu odczytania z pliku pierwszego wiersza tekstu instrukcja z wiersza 8. wywołuje metodę `infile.readline()`. Wiersz zwracany jako ciąg znaków jest przypisywany do zmiennej `line1`. Po wykonaniu tej instrukcji zmiennej `line1` zostanie przypisany ciąg znaków „John Locke\n”. Ponadto pozycja odczytu pliku zostanie przesunięta do następnego wiersza znajdującego się w tym pliku.

Następnie instrukcja z wiersza 9. odczytuje z pliku wiersz tekstu, rozpoczynając od aktualnej pozycji odczytu. Cały odczytany wiersz tekstu jest przypisywany do zmiennej `line2`. Po wykonaniu tej instrukcji zmiennej `line2` zostanie przypisany ciąg znaków „David Hume\n”. Pozycja odczytu pliku zostanie ponadto przesunięta do następnego wiersza znajdującego się w tym pliku.

Następnie instrukcja w wierszu 10. odczytuje z pliku następny wiersz tekstu, rozpoczynając od aktualnej pozycji odczytu. Wiersz tekstu jest przypisywany do zmiennej `line3`. Po wykonaniu tej instrukcji zmiennej `line3` zostanie przypisany ciąg znaków „Edmund Burke\n”. Pozycja odczytu pliku zostanie przesunięta na koniec pliku.

Instrukcja z wiersza 13. zamknie plik. Instrukcje z wierszy od 17. do 19. wyświetlają zawartość zmiennych `line1`, `line2` i `line3`.



**UWAGA:** Jeśli ostatni wiersz w pliku nie zostanie zakończony znakiem \n, to metoda readline zwróci wiersz bez znaku \n.

### Odczytywanie ciągu znaków i usuwanie z niego znaku nowego wiersza w języku Python

Czasami znak \n pojawiający się na końcu ciągu znaków zwracanych przez metodę readline powoduje różnorakie komplikacje. Na przykład można zauważać, że po każdym wierszu wypisywanego wyniku pojawia się pusty wiersz, tak jak ma to miejsce w programie na listingu 10.18. Dzieje się tak, ponieważ każdy z ciągów drukowanych w wierszach od 17. do 19. kończy się sekwencją znaku specjalnego \n. Podczas wypisywania ciągów znaków znak \n powoduje pojawienie się dodatkowego pustego wiersza.

W pliku znak \n ma do wykonania konkretne zadanie: oddziela elementy przechowywane w tym pliku. Jednak w większości przypadków po odczytaniu z pliku danych chcemy go usunąć z ciągu znaków. Każdy ciąg znaków w Pythonie ma metodę o nazwie rstrip, która usuwa określone znaki z jego końca. Oto przykład użycia metody rstrip:

```
name = 'Joanna Makowska\n'
name = name.rstrip('\n')
```

Pierwsza instrukcja przypisuje ciąg znaków „Joanna Makowska\n” do zmiennej name. (Zauważ, że ciąg kończy się sekwencją znaku wyjścia \n). Natomiast druga instrukcja wywołuje metodę name.rstrip('\'\n'). Metoda ta zwraca kopię ciągu znaków niezawierającą już znaku \n, a następnie ciąg ten jest przypisywany do zmiennej name. W rezultacie końcowy znak \n jest usuwany z tego ciągu.

Program przedstawiony na listingu 10.19 to zupełnie inny program, który ma za zadanie odczytanie i wyświetlanie zawartości pliku philosophers.txt. Program wykorzystuje metodę rstrip do usunięcia znaków \n z poszczególnych ciągów po odczytaniu ich z pliku, ale jeszcze przed wyświetlaniem ich na ekranie. W rezultacie dodatkowe puste wiersze nie będą się już pojawiały.

#### **Listing 10.19 (strip\_newline.py)**

```
1 # Ten program odczytuje zawartość pliku
2 # philosophers.txt po jednej linii
3 def main():
4     # Otwieramy plik o nazwie philosophers.txt
5     infile = open('philosophers.txt', 'r')
6
7     # Odczytujemy trzy wiersze z pliku
8     line1 = infile.readline()
9     line2 = infile.readline()
10    line3 = infile.readline()
11
12    # Usuwamy znak \n z wszystkich ciągów znaków
13    line1 = line1.rstrip('\n')
14    line2 = line2.rstrip('\n')
15    line3 = line3.rstrip('\n')
16
```

```

17 # Zamykamy plik
18 infile.close()
19
20 # Wpisujemy odczytane teksty
21 print('Oto imiona i nazwiska trzech filozofów:')
22 print(line1)
23 print(line2)
24 print(line3)
25
26 # Wywołujemy funkcję main
27 main()

```

### **Wynik działania programu**

Oto imiona i nazwiska trzech filozofów:

John Locke  
David Hume  
Edmund Burke

### **Dodawanie znaku końca wiersza do ciągu znaków w języku Python**

Program przedstawiony na listingu 10.17 zapisał do pliku trzy literały ciągów znaków, a każdy z tych ciągów zakończył się znakiem \n. W większości przypadków elementy danych zapisywane do pliku nie są literałami ciągu znaków, lecz wartościami w pamięci dostępnymi poprzez zmienne. W ten właśnie sposób działa program, który prosi użytkownika o wprowadzenie danych, a następnie zapisuje je do pliku.

Gdy program zapisuje do pliku dane wprowadzone przez użytkownika, zwykle konieczne będzie uprzednie połączenie sekwencji znaku \n z danymi. W ten sposób każda część danych zostanie zapisana w oddzielnym wierszu pliku. Program z listingu 10.20 pokazuje, jak się to odbywa.

### **Listing 10.20 (write\_names.py)**

```

1 # Program pobiera od użytkownika trzy imiona,
2 # a następnie zapisuje je do pliku
3
4 def main():
5     # Pobieramy trzy imiona
6     print('Wpisz imiona trojga przyjaciół.')
7     name1 = input('Przyjaciel #1: ')
8     name2 = input('Przyjaciel #2: ')
9     name3 = input('Przyjaciel #3: ')
10
11    # Otwieramy plik o nazwie friends.txt
12    myfile = open('friends.txt', 'w')
13
14    # Zapisujemy imiona do pliku
15    myfile.write(name1 + '\n')
16    myfile.write(name2 + '\n')
17    myfile.write(name3 + '\n')
18
19    # Zamykamy plik
20    myfile.close()
21    print('Imiona zostały zapisane do pliku friends.txt.')

```

```

22
23 # Wywołujemy funkcję main
24 main()

```

### **Wynik działania programu (pogrubieniem wyróżniono dane wprowadzone przez użytkownika)**

Wpisz imiona trojga przyjaciół.

Przyjaciel #1: **Janek** [Enter]

Przyjaciel #2: **Róża** [Enter]

Przyjaciel #3: **Grześ** [Enter]

Imiona zostały zapisane do pliku friends.txt.

W wierszach od 7. do 9. program prosi użytkownika o podanie trzech imion, a imiona te są przypisane do zmiennych name1, name2 i name3. Natomiast w wierszu 12. plik friends.txt otwierany jest w trybie do zapisu. Następnie w wierszach od 15. do 17. zapisywane są imiona wprowadzone przez użytkownika, a każde z nich otrzymuje na końcu znak '\n'. W rezultacie po zapisaniu w pliku każde z tych imion będzie miało dodaną sekwencję \n.

### **Zapisywanie danych liczbowych do pliku tekstowego w języku Python**

Ciągi znaków mogą być zapisywane bezpośrednio do pliku za pomocą metody `write`. Natomiast liczby, zanim będą mogły zostać zapisane do pliku, muszą zostać przekonwertowane na ciągi znaków. Python ma wbudowaną funkcję o nazwie `str`, która konwertuje wartość na ciąg znaków. Na przykład, zakładając, że zmiennej `num` zostanie przypisana wartość 99, wyrażenie `str(num)` zwróci ciąg znaków „99”.

Program przedstawiony na listingu 10.21 pokazuje sposób użycia funkcji `str`, aby przekonwertować liczbę na ciąg znaków, a następnie ciąg wynikowy zapisać do pliku.

#### **Listing 10.21 (write\_numbers.py)**

```

1 # Ten program pokazuje, w jaki sposób liczby
2 # muszą zostać przekonwertowane na ciągi znaków,
3 # zanim zostaną zapisane w pliku tekstowym
4
5 def main():
6     # Otwieramy plik do zapisu
7     outfile = open('numbers.txt', 'w')
8
9     # Pobieramy trzy liczby od użytkownika
10    num1 = int(input('Wprowadź pierwszą liczbę: '))
11    num2 = int(input('Wprowadź drugą liczbę: '))
12    num3 = int(input('Wprowadź trzecią liczbę: '))
13
14    # Zapisujemy liczby w pliku
15    outfile.write(str(num1) + '\n')
16    outfile.write(str(num2) + '\n')
17    outfile.write(str(num3) + '\n')
18
19    # Zamkamy plik
20    outfile.close()
21    print('Dane zostały zapisane do pliku numbers.txt')
22
23 # Wywołujemy funkcję main
24 main()

```

**Wynik działania programu (pogrubioną czcionką pokazano dane wprowadzone przez użytkownika)**

```
Wprowadź pierwszą liczbę: 22 [Enter]
Wprowadź drugą liczbę: 14 [Enter]
Wprowadź trzecią liczbę: -99 [Enter]
Dane zostały zapisane do pliku numbers.txt
```

Instrukcja w wierszu 7. otwiera do zapisu plik *numbers.txt*, a następnie instrukcje w wierszach od 10. do 12. proszą użytkownika o wprowadzenie trzech liczb, które zostaną przypisane do zmiennych *num1*, *num2* i *num3*.

Przyjrzyj się instrukcji w wierszu 15., która zapisuje do pliku wartość znajdująca się w zmiennej *num1*:

```
outfile.write(str(num1) + '\n')
```

Wyrażenie *str(num1) + '\n'* konwertuje wartość ze zmiennej *num1* na ciąg znaków, a następnie łączy go z sekwencją *\n*. W przykładowym uruchomieniu programu użytkownik wprowadził jako pierwszą liczbę 22, a następnie powyższe wyrażenie utworzyło ciąg znaków „22\n”. W rezultacie w pliku zapisany został ciąg znaków „22\n”. Wiersze 16. i 17. wykonują podobne operacje, zapisując w pliku wartości zapisane w zmiennech *num2* i *num3*.

### Odczytywanie danych liczbowych z pliku tekstowego w języku Python

Kiedy odczytujemy liczby z pliku tekstowego, są one zawsze odczytywane jako ciągi znaków. Założmy, że program używa następującego kodu do odczytania pierwszego wiersza z pliku *numbers.txt*, który został utworzony przez program z listingu 10.5:

```
1 infile = open('numbers.txt', 'r')
2 value = infile.readline()
3 infile.close()
```

Instrukcja w wierszu 2. wywołuje metodę *readline* w celu odczytania wiersza z pliku. Po wykonaniu tej instrukcji w zmiennej *value* znajdzie się ciąg znaków „22\n”. Może to spowodować pewien problem, jeżeli zamierzamy na przykład wykonać operację matematyczną na wartości ze zmiennej, ponieważ nie można wykonywać takich operacji na ciągach znaków. W takim przypadku musimy przekonwertować ciąg znaków na postać liczbową.

Python ma już wbudowaną funkcję *int*, która służy do konwersji ciągu znaków na liczbę całkowitą, natomiast wbudowana funkcja *float* przekształca ciąg znaków na liczbę zmiennoprzecinkową. Na przykład możemy zmodyfikować w następujący sposób kod pokazany wcześniej:

```
1 infile = open('numbers.txt', 'r')
2 string_input = infile.readline()
3 value = int(string_input)
4 infile.close()
```

Instrukcja w wierszu 2. odczytuje wiersz z pliku i przypisuje go do zmiennej *string\_input*. W rezultacie w tej zmiennej znajdzie się ciąg znaków „22\n”. Następnie instrukcja w wierszu 3. wywołuje funkcję *int* w celu skonwertowania wartości zmiennej

`string_input` na liczbę całkowitą, a potem wynik przypisze do zmiennej `value`. Po wykonaniu tej instrukcji w zmiennej `value` znajdzie się liczba całkowita 22. (Zarówno `int`, jak i `float` ignorują znaki `\n` pojawiające się na końcu ciągów znaków przekazywanych jako ich argumenty).

Powyższy kod demonstruje kroki związane z odczytywaniem ciągu znaków z pliku za pomocą metody `readline` i późniejszym przekształcaniem go w liczbę całkowitą za pomocą funkcji `int`. W wielu sytuacjach taki kod można jednak uprościć. Lepszym sposobem będzie odczytanie ciągu znaków z pliku i przekonwertowanie go w ramach jednej instrukcji:

```
1 infile = open('numbers.txt', 'r')
2 value = int(infile.readline())
3 infile.close()
```

Zauważ, że w wierszu 2. wywołanie metody `readline` jest używane jako argument funkcji `int`. Oto jak działa powyższy kod programu: wywoływana jest metoda `readline`, która zwraca ciąg znaków, następnie jest on przekazywany do funkcji `int`, która konwertuje go na liczbę całkowitą, a wynik operacji jest przypisywany do zmiennej `value`.

### Wykrywanie końca pliku w języku Python

W Pythonie metoda `readline` zwraca pusty串 znaków ('') w przypadku próby odczytu danych poza końcem pliku. Aby zapobiec takiej sytuacji, musimy wykorzystać pętlę `while`, która określi, kiedy osiągnięto koniec pliku. Program przedstawiony na listingu 10.22 demonstruje sposób, w jaki można wykorzystać taką pętlę w kodzie programu. Program odczytuje i wyświetla wszystkie wartości w pliku `sales.txt`. (Jest to wersja pseudokodu z listingu 10.3 napisana w języku Python).

#### **Listing 10.22 (read\_sales.py)**

```
1 # Ten program odczytuje wszystkie
2 # wartości z pliku sales.txt.
3
4 def main():
5     # Otwieramy plik sales.txt do odczytu
6     sales_file = open('sales.txt', 'r')
7
8     # Odczytujemy pierwszy wiersz z pliku,
9     # ale bez konwertowania go na liczbę
10    # Najpierw trzeba sprawdzić, czy jest to pusty串
11    line = sales_file.readline()
12
13    print('Oto kwoty sprzedaży:')
14
15    # Kontynuujemy przetwarzanie, dopóki funkcja readline
16    # nie zwróci pustego串 znaków
17    while line != '':
18        # Konwersja wiersza tekstu na wartość typu float
19        amount = float(line)
20
21        # Formatujemy i wyświetlamy kwoty
22        print(format(amount, '.2f'), 'zł')
23
```

```

24     # Odczytujemy następny wiersz
25     line = sales_file.readline()
26
27     # Zamkamy plik
28     sales_file.close()
29
30 # Wywołujemy funkcję main
31 main()

```

### **Wynik działania programu (pogrubioną czcionką pokazano dane wprowadzone przez użytkownika)**

Oto kwoty sprzedaży:

```

1000.00 zł
2000.00 zł
3000.00 zł
4000.00 zł
5000.00 zł

```

### **Wykorzystywanie pętli for do odczytu wiersza za pomocą języka Python**

W poprzednim przykładzie można było zobaczyć, jak metoda `readline` zwraca pusty ciąg znaków po osiągnięciu końca pliku. Python umożliwia również napisanie pętli `for`, która automatycznie odczytuje wiersze z pliku bez sprawdzania ich pod kątem jakiegokolwiek szczególnego warunku sygnalizującego koniec tego pliku. Pętla `for` nie wymaga operacji odczytu wstępnego i zatrzymuje się automatycznie po osiągnięciu końca pliku. Kiedy chcemy po prostu znajdujące się w pliku wiersze czytać jeden po drugim, ta technika będzie prostsza i bardziej elegancka niż technika wykorzystująca pętlę `while`, która wymaga jawnego sprawdzania, czy dotarliśmy już do końca pliku. Oto ogólny format tej pętli:

```

for zmienna in obiekt_pliku:
    instrukcja
    instrukcja
    itd.

```

W takim ogólnym formacie `zmienna` jest nazwą zmiennej, a `obiekt_pliku` jest zmienną przechowującą obiekt pliku. Pętla `for` będzie iterować po wszystkich wierszach tekstu z tego pliku. Przy pierwszej iteracji pętli `zmienna` otrzyma zawartość pierwszego wiersza tekstu znajdującego się w pliku (czyli ciąg znaków), przy drugim obiegu pętli `for` w `zmiennej` znajdzie się zawartość drugiego wiersza i tak dalej. Program widoczny na listingu 10.23 demonstruje działanie takiej pętli, która odczytuje i wyświetla wszystkie pozycje znajdujące się w pliku `sales.txt`.

#### **Listing 10.23 (read\_sales2.py)**

```

1 # Ten program używa pętli for do odczytu
2 # wszystkich wartości w pliku sales.txt
3
4 def main():
5     # Otwieramy plik sales.txt do odczytu
6     sales_file = open('sales.txt', 'r')
7
8     # Odczytujemy wszystkie wiersze z pliku
9     for line in sales_file:

```

```

10      # Konwersja wiersza tekstu na wartość typu float
11      amount = float(line)
12      # Formatujemy i wyświetlamy kwoty
13      print(format(amount, '.2f'), ' zł')
14
15      # Zamkamy plik
16      sales_file.close()
17
18 # Wywołujemy funkcję main
19 main()

```

**Wynik działania programu**

1000.00 zł  
2000.00 zł  
3000.00 zł  
4000.00 zł  
5000.00 zł

**C++****Pliki****Otwieranie i zapisywanie pliku zawierającego dane w języku C++**

Aby pracować z plikami w C++, na początku kodu programu należy umieścić poniższą dyrektywę `#include`:

```
#include <fstream>
```

Następnie w funkcji, w której chcesz otworzyć plik i zapisać do niego dane, należy zadeklarować obiekt typu `ofstream`. Oto przykład instrukcji deklarującej obiekt typu `ofstream`:

```
ofstream outputFile;
```

Instrukcja ta deklaruje obiekt typu `ofstream` o nazwie `outputFile`. Z kolei poniższa instrukcja wykorzystuje go do otwarcia pliku:

```
outputFile.open("StudentData.txt");
```

Po wykonaniu tej instrukcji będziemy już mogli korzystać z obiektu `ofstream` o nazwie `outputFile` do zapisywania danych w pliku `StudentData.txt`. Można to sobie wyobrazić w następujący sposób: w pamięci mamy obiekt typu `ofstream`, do którego odwołujemy się w kodzie programu poprzez nazwę `outputFile`. Obiekt ten jest połączony ze znajdującym się na dysku plikiem o nazwie `StudentData.txt`. Chcąc zapisać do niego dane, musimy użyć obiektu typu `ofstream`. (Zwróć uwagę, że jeśli plik `StudentData.txt` nie istnieje, to instrukcja go utworzy, a jeśli już istnieje, to jego zawartość zostanie usunięta. Tak czy inaczej, po wykonaniu tej instrukcji na dysku powstanie pusty plik o nazwie `StudentData.txt`).

Po utworzeniu obiektu typu `ofstream` i otwarciu pliku można do niego zapisywać dane za pomocą operatora wstawiania do strumienia (`<<`). Jak już wiesz, operatora `<<` w połączeniu z funkcją `cout` można używać do wyświetlania danych na ekranie.

Operator ten jest stosowany w ten sam sposób z obiektem typu `ofstream` w celu zapisania danych do pliku. Zakładając, że funkcja  `outputFile`  jest obiektem typu `ofstream`, poniższa instrukcja zapisze do pliku ciąg znaków „Jaś”:

```
outputFile << "Jaś" << endl;
```

Zakładając, że `payRate` jest zmienną, ta instrukcja zapisuje jej wartość do pliku:

```
outputFile << payRate << endl;
```

### Zamykanie pliku w języku C++

Gdy program zakończy zapisywanie danych do pliku, musi go zamknąć. Zakładając, że `outputfile` jest nazwą obiektu typu `ofstream`, poniżej przedstawiam przykład wywołania funkcji `close` w celu zamknięcia pliku:

```
outputFile.close();
```

Po zamknięciu pliku połączenie między nim a obiektem typu `ofstream` zostaje usunięte. Aby wykonać dalsze operacje na tym pliku, należy go ponownie otworzyć.

Program widoczny na listingu 10.24 pokazuje, jak utworzyć obiekt typu `ofstream` (i otworzyć plik w trybie do zapisu), zapisać pewne dane do pliku, po czym go zamknąć. Jest to wersja programu w pseudokodzie z listingu 10.1 napisana w języku C++.

#### **Listing 10.24 (WriteNames.cpp)**

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     // Deklarujemy obiekt o nazwie myFile
8     // i otwieramy plik philosophers.txt
9     ofstream myFile;
10    myFile.open("philosophers.txt");
11
12    // Zapisujemy w pliku nazwiska trzech filozofów
13    myFile << "John Locke" << endl;
14    myFile << "David Hume" << endl;
15    myFile << "Edmund Burke" << endl;
16
17    // Zamykamy plik
18    myFile.close();
19    return 0;
20 }
```

Po uruchomieniu tego programu w wierszu 9. deklarowany jest obiekt typu `ofstream` o nazwie `myFile`, a w wierszu 10. jest on używany do otwarcia znajdującego się na dysku pliku `philosophers.txt`. Wiersze od 13. do 15. zapisują do pliku ciągi znaków „John Locke”, „David Hume” i „Edmund Burke”. Natomiast wiersz 18. zamyka ten plik.

Program znajdujący się na listingu 10.25 pokazuje inny przykład. Otwiera on plik o nazwie *numbers.txt*, a następnie korzysta z pętli w celu zapisania w tym pliku liczb od 1 do 5.

### **Listing 10.25 (WriteNumbers.cpp)**

```

1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     // Stała wyznaczająca maksymalną liczbę liczb
8     const int MAX_NUMS = 5;
9
10    // Zmienna
11    int counter;
12
13    // Deklarujemy obiekt o nazwie myFile
14    // i otwieramy plik o nazwie numbers.txt
15    ofstream myFile;
16    myFile.open("numbers.txt");
17
18    // Zapisujemy w pliku liczby od 1 do 5
19    for (counter = 1; counter <= MAX_NUMS; counter++)
20    {
21        // Zapisujemy liczby do pliku
22        myFile << counter << endl;
23    }
24
25    // Zamkamy plik
26    myFile.close();
27    return 0;
28 }
```

### **Otwieranie pliku i odczytywanie z niego danych w języku C++**

Teraz omówimy, jak odczytać dane z pliku w języku C++. W naszym programie potrzebna jest następująca dyrektywa `#include`:

```
#include <fstream>
```

Następnie w funkcji, za pomocą której chcemy otworzyć plik i odczytać z niego dane, zadeklarujemy obiekt typu `ifstream`. Oto przykład instrukcji deklarującej obiekt typu `ifstream`:

```
ifstream inputFile;
```

Ta instrukcja deklaruje obiekt typu `ifstream` o nazwie `inputFile`, a następnie używa go, aby otworzyć plik. Oto przykład:

```
inputFile.open("numbers.txt");
```

Po wykonaniu tej instrukcji będziemy mogli użyć obiektu typu `ifstream` o nazwie `inputFile`, aby odczytać dane z pliku *numbers.txt*. Można to przeanalizować w następujący sposób: w pamięci mamy obiekt typu `ifstream`, do którego odwołujemy się

w naszym kodzie za pomocą zmiennej `inputFile`; obiekt ten jest połączony ze znajdującym się na dysku plikiem o nazwie `numbers.txt`. Chcąc odczytać z niego dane, musimy skorzystać z obiektu typu `ifstream`.

Po utworzeniu obiektu typu `ifstream` i po otwarciu pliku można odczytać z niego dane za pomocą operatora wyjścia ze strumienia (`>>`). Wiemy już, jak możemy użyć tego operatora w połączeniu z funkcją `cin`, aby odczytać dane wejściowe z klawiatury. W taki sam sposób możemy też go zastosować, aby odczytać dane z pliku, wykorzystując do tego obiekt typu `ifstream`. Zakładając, że `inputFile` jest obiektem typu `ifstream`, poniższa instrukcja odczyta fragment danych z pliku i zapisze go w zmiennej `value`:

```
inputFile >> value;
```

### Zamykanie pliku za pomocą języka C++

Zakładając, że `inputFile` jest nazwą obiektu typu `ifstream`, poniżej przedstawiam przykład wywołania funkcji `close` w celu zamknięcia pliku:

```
inputFile.close();
```

Po zamknięciu pliku połączenie między nim a obiektem typu `ifstream` zostaje usunięte. Aby wykonać dalsze operacje na pliku, należy go ponownie otworzyć.

Na listingu 10.25 zaprezentowałem przykład programu, który utworzył plik i zapisał w nim liczby od 1 do 5. Natomiast program z listingu 10.26 przedstawia sposób na odczytanie listy liczb z pliku i wyświetlenie ich na ekranie.

#### **Listing 10.26 (ReadNumbers.cpp)**

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     // Stała określająca maksymalną liczbę liczb
8     const int MAX_NUMS = 5;
9
10    // Zmienne
11    int counter, number;
12
13    // Deklarujemy obiekt typu ifstream o nazwie myFile
14    // i otwieramy plik numbers.txt w trybie do odczytu
15    ifstream myFile;
16    myFile.open("numbers.txt");
17
18    // Poniższa pętla odczytuje 5 liczb
19    // z pliku i je wyświetla
20    for (counter = 1; counter <= MAX_NUMS; counter++)
21    {
22        // Odczytujemy liczby z pliku
23        myFile >> number;
24
25        // Wyświetlamy liczby
26        cout << number << endl;
27    }
```

```

28
29 // Zamykamy plik
30 myFile.close();
31 return 0;
32 }
```

**Wynik działania programu**

```

1
2
3
4
5
```

**Korzystanie z funkcji getline do odczytu ciągów znaków**

Jak już wspomniałem, aby odczytać z klawiatury ciąg znaków zawierający wiele słów oddzielonych spacjami, musimy użyć funkcji `getline`. To samo dotyczy przypadku, gdy chcemy odczytać z pliku ciąg znaków zawierający spacje. Funkcja `getline` może odczytywać cały wiersz danych wejściowych z pliku, w tym także spacje, a następnie umieszczać je w zmiennej `string`.

Załóżmy, że `inputFile` jest nazwą obiektu typu `ifstream` i że właśnie otworzyliśmy plik. Założymy również, że `line` jest nazwą zmiennej typu `string`. Poniższa instrukcja pokazuje, w jaki sposób możemy użyć funkcji `getline`, aby odczytać wiersz danych wejściowych z pliku, a następnie zapisać go do zmiennej `line`:

```
getline(inputFile, line);
```

Program przedstawiony na listingu 10.27 pokazuje przykład użycia tej funkcji. Otwierany jest tutaj plik *philosophers.txt*, który został utworzony przez program z listingu 10.24. Jest to wersja programu w pseudokodzie z listingu 10.2 zapisana w języku C++.

**Listing 10.27**

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 using namespace std;
5
6 int main()
7 {
8     // Deklarujemy trzy zmienne, które będą
9     // zawierać wartości odczytane z pliku
10    string name1, name2, name3;
11
12    // Deklarujemy obiekt typu ifstream o nazwie myFile
13    // i otwieramy plik philosophers.txt
14    ifstream myFile;
15    myFile.open("philosophers.txt");
16
17    // Odczytujemy z pliku nazwiska trzech filozofów
18    // i zapisujemy je w zmiennych
19    getline(myFile, name1);
20    getline(myFile, name2);
21    getline(myFile, name3);
```

```

22 // Wyświetlamy teksty odczytane z pliku
23 cout << "Oto imiona i nazwiska trzech filozofów:" << endl;
24 cout << name1 << endl;
25 cout << name2 << endl;
26 cout << name3 << endl;
27
28 // Zamkamy plik
29 myFile.close();
30 return 0;
31 }

```

### **Wynik działania programu**

Oto imiona i nazwiska trzech filozofów:

John Locke  
David Hume  
Edmund Burke

### **Dołączanie danych do istniejącego pliku za pomocą języka C++**

Jeżeli użyjesz obiektu typu `ofstream` do otwarcia istniejącego pliku, to jego zawartość zostanie usunięta. Czasami jednak chcemy otworzyć istniejący plik bez usuwania jego aktualnej zawartości, a następnie zapisać do niego nowe dane na jego końcu. Taka operacja nazywa się to **dołączaniem** danych do pliku.

Kiedy chcemy dołączyć dane do istniejącej zawartości pliku, w języku C++ deklarujemy obiekt typu `fstream`. Oto przykład deklarowania takiego obiektu o nazwie `myFile`, a następnie użycia go do otwarcia pliku `friends.txt`, przy czym istniejąca zawartość pliku nie zostanie usunięta:

```

fstream myFile;
myFile.open("friends.txt", ios::app);

```

Zauważ, że do funkcji `open` przekazywane są dwa argumenty:

- nazwa pliku — w tym przypadku `friends.txt`;
- specjalna wartość `ios::app`, która określa, że wszelkie dane zapisywane do tego pliku powinny zostać dołączone do istniejącej zawartości.

### **Wykrywanie końca pliku za pomocą języka C++**

Czasami istnieje potrzeba odczytu zawartości pliku o nieznanej liczbie przechowywanych w nim elementów. W takim przypadku możemy najpierw otworzyć ten plik, a następnie użyć pętli, aby wielokrotnie odczytać z niego dane, po czym wyświetlić je na ekranie. Natomiast w przypadku, gdy program spróbuje odczytać dane znajdujące się poza końcem pliku, wystąpi błąd. Program musi wiedzieć, kiedy znajdzie się na końcu pliku, żeby nie próbować dalszego czytania. Na szczęście operator `>>` nie tylko odczytuje dane z pliku, ale także zwraca wartość `true` lub `false`, wskazującą, czy dane zostały pomyślnie odczytane, czy nie. Jeśli operator zwraca wartość `true`, oznacza to, że wartość została pomyślnie odczytana. W przeciwnym wypadku, czyli gdy operator zwróci wartość `false`, z pliku nie została odczytana żadna wartość.

Program z listingu 10.28 pokazuje, jak sprawnie korzystać z tej techniki. Otwiera on plik `numbers.txt`, który został utworzony przez program z listingu 10.25, a następnie odczytuje go i wyświetla każdy element danych znajdujący się w tym pliku.

**Listing 10.28**

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7     // Deklarujemy obiekt pliku wejściowego
8     ifstream salesFile;
9
10    // Deklarujemy zmienną do przechowywania liczb
11    int number;
12
13    // Otwieramy plik numbers.txt
14    salesFile.open("numbers.txt");
15
16    // Odczytujemy i wyświetlamy elementy z pliku
17    while (salesFile >> number)
18    {
19        cout << number << endl;
20    }
21
22    // Zamkamy plik
23    salesFile.close();
24    return 0;
25 }
```

**Wynik działania programu**

```
1
2
3
4
5
```

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Plik, do którego zapisujemy dane, nazywamy plikiem \_\_\_\_\_.
  - a) wejściowym
  - b) wyjściowym
  - c) o dostępie sekwencyjnym
  - d) binarnym
2. Plik, z którego odczytujemy dane, nazywamy plikiem \_\_\_\_\_.
  - a) wejściowym
  - b) wyjściowym
  - c) o dostępie sekwencyjnym
  - d) binarnym

3. Zanim zaczniemy przetwarzać plik w programie, należy go najpierw \_\_\_\_\_.  
a) sformatować  
b) zaszyfrować  
c) zamknąć  
d) otworzyć
4. Gdy zakończymy pracę z plikiem, należy go \_\_\_\_\_.  
a) wykasować  
b) otworzyć  
c) zamknąć  
d) zaszyfrować
5. Zawartość tego rodzaju plików można podejrzeć w edytorze takim jak Notatnik.  
a) pliki tekstowe  
b) pliki binarne  
c) pliki w języku angielskim  
d) pliki czytelne dla człowieka
6. Ten typ plików zawiera dane, które nie zostały skonwertowane na tekst.  
a) pliki tekstowe  
b) pliki binarne  
c) pliki Unicode  
d) pliki symboliczne
7. Pracując z plikami tego typu, dane odczytujemy po kolej, od początku pliku do jego końca.  
a) pliki o dostępie uporządkowanym  
b) pliki o dostępie binarnym  
c) pliki o dostępie bezpośrednim  
d) pliki o dostępie sekwencyjnym
8. Pracując z plikami tego typu, możemy przejść do dowolnego miejsca pliku, bez potrzeby odczytywania danych zapisanych wcześniej.  
a) pliki o dostępie uporządkowanym  
b) pliki o dostępie binarnym  
c) pliki o dostępie bezpośrednim  
d) pliki o dostępie sekwencyjnym
9. Jest to „przechowalnia” w pamięci komputera, w której system zapisuje dane przed zapisaniem ich do pliku.  
a) bufor  
b) zmienna  
c) plik wirtualny  
d) plik tymczasowy
10. Jest to znak lub seria znaków, które wskazują miejsce, gdzie kończy się dany element zapisany w pliku.  
a) mediana  
b) separator  
c) wskaźnik graniczny  
d) wskaźnik EOF

11. Jest to znak lub seria znaków, które wskazują miejsce, gdzie kończy się zawartość pliku.
  - a) mediana
  - b) separator
  - c) wskaźnik graniczny
  - d) wskaźnik EOF
12. \_\_\_\_\_ wskazuje miejsce w pliku, z którego zostanie odczytany kolejny element.
  - a) wskaźnik wejściowy
  - b) separator
  - c) wskaźnik
  - d) wewnętrzny wskaźnik pliku
13. Kiedy plik otwarty jest w tym trybie, nowe dane zostaną dodane na jego końcu.
  - a) tryb wyjściowy
  - b) tryb dołączania
  - c) tryb dublowania
  - d) tryb tylko do odczytu
14. Wyrażenie NOT eof(myFile) jest równoznaczne z wyrażeniem \_\_\_\_\_.
  - a) eof(myFile) == True
  - b) eof(myFile)
  - c) eof(myFile) == False
  - d) eof(myFile) < 0
15. Jest to pojedyncza właściwość opisująca element zapisany w rekordzie.
  - a) pole
  - b) zmienna
  - c) separator
  - d) podrekord

### **Prawda czy fałsz?**

1. Pracując z plikiem otwartym w trybie sekwencyjnym, można przejść do dowolnego jego miejsca bez potrzeby odczytywania danych zapisanych wcześniej.
2. W większości języków programowania otwarcie pliku wyjściowego, który już istnieje na dysku, zakończy się wykasowaniem dotychczasowej zawartości pliku.
3. Plik należy otworzyć tylko wtedy, gdy chcemy odczytać z niego dane. Pliki, do których chcemy zapisać dane, otwierają się automatycznie w momencie, gdy zapisujemy w nich dane.
4. Zadaniem wskaźnika EOF jest informowanie o tym, gdzie kończy się dane pole. Typowy plik zawiera kilka wskaźników EOF.
5. Zaraz po otwarciu pliku do odczytu jego wskaźnik wewnętrzny wskazuje jego początek.
6. Po otwarciu pliku wyjściowego w trybie dołączania jego dotychczasowa zawartość zostanie wykasowana.
7. W mechanizmie separatorów sterowania program przetwarza dane (np. dane zapisane w pliku), a w momencie, gdy zmienna sterująca zmieni wartość lub będzie równa określonej wartości, program natychmiast kończy działanie.

## Krótką odpowiedź

1. Wymień i opisz trzy kroki, które należy wykonać, aby skorzystać z pliku w programie.
2. Dlaczego po zakończeniu pracy z plikiem należy go zamknąć?
3. Co to jest wewnętrzny wskaźnik pliku? Które miejsce on wskazuje zaraz po otwarciu pliku do odczytu?
4. Co się stanie z dotychczasową zawartością pliku, jeśli otworzymy go w trybie dołączania?
5. Co się stanie w większości języków programowania podczas próby otwarcia w trybie dołączania pliku nieistniejącego na dysku?
6. Do czego służy funkcja eof, którą omówilem w tym rozdziale?
7. Co to są separatory sterowania?

## Warsztat projektanta algorytmów

1. Zaprojektuj program, który otwiera plik wyjściowy i przypisuje go do obiektu reprezentującego plik o nazwie *my\_name.dat*. Następnie zapisz w pliku Twoje imię i nazwisko i zamknij go.
2. Zaprojektuj program, który otwiera plik *my\_name.dat* utworzony w poprzednim punkcie, odczytuje z niego Twoje imię i nazwisko, wyświetla je na ekranie, a następnie zamknięcie pliku.
3. Zaprojektuj algorytm wykonujący następujące operacje: otwarcie pliku wyjściowego i przypisanie go do obiektu reprezentującego plik o nazwie *number\_list.dat*; zapisanie w pliku liczb od 1 do 100 za pomocą pętli; zamknięcie pliku.
4. Zaprojektuj algorytm wykonujący następujące operacje: otwarcie pliku o nazwie *number\_list.dat* utworzonego w poprzednim punkcie; odczytanie z pliku i wyświetlenie wszystkich liczb; zamknięcie pliku.
5. Zmodyfikuj algorytm z punktu 4. tak, aby dodawał wszystkie liczby i wyświetlał ich sumę.
6. Napisz za pomocą pseudokodu program, w którym otworzysz plik wyjściowy i przypiszesz go do obiektu reprezentującego plik o nazwie *number\_list.dat*, jednocześnie nie usuwając dotychczasowej zawartości tego pliku.
7. Na dysku zapisany jest plik o nazwie *students.dat*. Zawiera on kilka rekordów, z których każdy składa się z dwóch pól: nazwisko studenta i wynik studenta z egzaminu końcowego. Zaprojektuj algorytm, za pomocą którego można będzie usunąć rekord studenta o nazwisku Jarosław Piekarczyk.
8. Na dysku zapisany jest plik o nazwie *students.dat*. Zawiera on kilka rekordów, z których każdy składa się z dwóch pól: nazwisko studenta i wynik studenta z egzaminu końcowego. Zaprojektuj algorytm, za pomocą którego można będzie zmienić wynik Joanny Makowieckiej na 100.

## Ćwiczenia z wykrywania błędów

- Dlaczego poniższy moduł nie zadziała zgodnie z opisem umieszczonym w komentarzach?

```

// Metoda readFile przyjmuje jako argument ciąg znaków zawierający nazwę pliku
// oraz odczytuje i wyświetla wszystkie elementy zapisane w pliku
Module readFile(String filename)
    // Deklarujemy plik wejściowy
    Declare InputFile file

    // Zmienna, w której zapiszemy element odczytany z pliku
    Declare String item

    // Otwieramy wskazany plik
    Open file filename

    // Odczytujemy wszystkie elementy zapisane w pliku i je wyświetlamy
    While eof(file)
        Read file item
        Display item
    End While
End Module

```

## Ćwiczenia programistyczne

- Wyświetlanie danych zapisanych w pliku

Załóżmy, że na dysku znajduje się plik o nazwie *numbers.dat*, w którym zapisany jest szereg liczb. Zaprojektuj program, który wyświetli wszystkie liczby zapisane w tym pliku.

- Licznik elementów**

Załóżmy, że na dysku znajduje się plik o nazwie *names.dat*, w którym zapisany jest szereg imion (jako ciągi znaków). Zaprojektuj program, który wyświetli liczbę imion zapisanych w tym pliku.

*Podpowiedź:* Otwórz plik i odczytaj z niego po kolei każdy ciąg znaków. Po odczytaniu każdego ciągu zwiększą zmienną licznikową. Po odczytaniu ostatniego ciągu znaków zmienna licznikowa będzie równa liczbie imion zapisanych w pliku.

- Suma liczb**

Załóżmy, że na dysku znajduje się plik o nazwie *numbers.dat*, w którym zapisany jest szereg liczb całkowitych. Zaprojektuj program, który odczyta wszystkie liczby zapisane w tym pliku i wyświetli ich sumę.

- Średnia wartość liczb**

Załóżmy, że na dysku znajduje się plik o nazwie *numbers.dat*, w którym zapisany jest szereg liczb całkowitych. Zaprojektuj program, który obliczy średnią wartość liczb zapisanych w pliku.

## 5. Największa liczba

Załóżmy, że na dysku znajduje się plik o nazwie *numbers.dat*, w którym zapisany jest szereg liczb całkowitych. Zaprojektuj program, który określi wartość największej z liczb zapisanych w tym pliku.

*Podpowiedź:* Skorzystaj z opisanej w rozdziale 8. techniki wyszukiwania największej wartości w tablicy. Nie musisz w tym przypadku kopiować liczb z pliku do tablicy — wystarczy, że dostosujesz opisany algorytm do potrzeb tego ćwiczenia.

## 6. Wyniki gry w golfa

Klub golfowy Springfork Amateur organizuje w każdy weekend zawody. Prezes klubu poprosił Cię o zaprojektowanie dwóch programów:

1. Program, który umożliwi wprowadzenie nazwiska i wyniku każdego zawodnika, a następnie zapisze te dane jako rekordy w pliku *golf.dat*. Każdy rekord musi zawierać pola z nazwiskiem zawodnika i jego wynikiem.
2. Program, który odczyta rekordy zapisane w pliku *golf.dat*, a następnie je wyświetli.

## 7. Najlepszy wynik gry w golfa

Zmodyfikuj program nr 2. z ćwiczenia 6. tak, aby wyświetlał także nazwisko gracza, który uzyskał najlepszy (najniższy) wynik.

*Podpowiedź:* Skorzystaj z opisanej w rozdziale 8. techniki wyszukiwania najmniejszej wartości w tablicy. Nie musisz w tym przypadku kopiować wyników do tablicy — wystarczy, że dostosujesz algorytm do potrzeb tego ćwiczenia.

## 8. Raport sprzedaży

Firma AutoCentrum zajmuje się sprzedażą używanych samochodów i zatrudnia kilku przedstawicieli handlowych. Właściciel firmy przekazał Ci plik, w którym dla każdego przedstawiciela zapisane są rekordy sprzedaży w poprzednim miesiącu. Każdy rekord zawiera następujące dwa pola:

- identyfikator przedstawiciela handlowego, będący liczbą całkowitą;
- wartość sprzedaży, będąca liczbą rzeczywistą.

Rekordy zostały już uporządkowane w pliku według identyfikatorów przedstawicieli handlowych. Właściciel firmy poprosił Cię, abyś zaprojektował program, który będzie drukował raport sprzedaży. Raport powinien zawierać zarówno poszczególne operacje, jak i sumaryczną wartość sprzedaży dla każdego przedstawiciela handlowego. W raporcie powinna się także znaleźć sumaryczna wartość sprzedaży wszystkich przedstawicieli handlowych. Oto jak powinien wyglądać przykładowy raport:

AutoCentrum  
Raport sprzedaży

ID przedstawiciela	Wartość sprzedaży
100	10 000,00 zł
100	12 000,00 zł
100	5 000,00 zł
Suma sprzedaży dla tego przedstawiciela: 27 000,00 zł	

101	14 000,00 zł
101	18 000,00 zł
101	12 500,00 zł
Suma sprzedaży dla tego przedstawiciela: 44 500,00 zł	

102	13 500,00 zł
102	14 500,00 zł
102	20 000,00 zł
Suma sprzedaży dla tego przedstawiciela: 48 000,00 zł	
Całkowita wartość sprzedaży: 119 500,00 zł	

### 9. Najniższe i najwyższe ceny benzyny

Założmy, że mamy plik przechowujący tygodniowe średnie ceny za litr benzyny w Polsce. Dane w tym pliku zawierają statystykę prowadzoną od 3 ostatnich lat. Dane te są przechowywane jako rekordy. Każdy rekord zawiera średnią cenę litra benzyny w określonym dniu i składa się z następujących pól:

- miesiąc zapisany jako liczba całkowita: styczeń = 1, luty = 2 itd.;
- dzień miesiąca zapisany jako liczba całkowita;
- rok zapisany jako liczba całkowita;
- średnia cena litra benzyny w określonym dniu zapisana jako liczba rzeczywista zaokrąglona do 3. miejsca po przecinku.

Zaprojektuj program, który odczyta plik oraz wyświetli najniższe i najwyższe ceny benzyny, a także daty tych cen.

### 10. Wyliczanie średniej liczby kroków

Personal Fitness Tracker to urządzenie ubieralne, które monitoruje m.in. aktywność fizyczną, spalone kalorie, tętno i jakość snu. Jedną z typowych aktywności fizycznych monitorowanych przez większość tego typu urządzeń jest liczba kroków, jakie przechodzisz każdego dnia.

Założmy, że mamy plik zawierający liczbę kroków, które dana osoba przeszła każdego dnia w ciągu roku. Dane te są przechowywane w pliku jako rekordy. Każdy rekord zawiera liczbę kroków wykonanych w określonym dniu. Składa się on z następujących pól:

- miesiąc zapisany jako liczba całkowita: styczeń = 1, luty = 2 itd.;
- dzień miesiąca zapisany jako liczba całkowita;
- rok zapisany jako liczba całkowita;
- liczba kroków wykonanych w określonym dniu zapisana jako liczba całkowita.

Ponadto zapisy te są przechowywane w kolejności dat. Na przykład pierwszy zapis w tym pliku jest na dzień 1 stycznia, a ostatni rekord przypada na dzień 31 grudnia. Zaprojektuj program, który odczyta plik i wyświetli średnią liczbę kroków wykonanych w każdym miesiącu.

## TEMATYKA

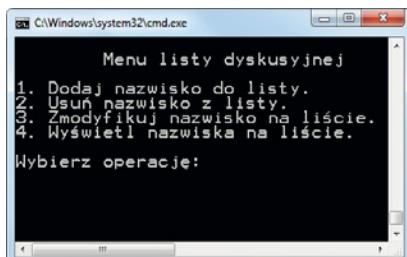
- |   |  |
|---|--|
| 11.1 Wprowadzenie do programów sterowanych za pomocą menu | 11.3 Ponowne wyświetlanie menu za pomocą pętli |
| 11.2 Modularyzacja programu sterowanego za pomocą menu    | 11.4 Menu wielopoziomowe                       |
|   | 11.5 Rzut oka na języki Java, Python i C++     |

## 11.1

## Wprowadzenie do programów sterowanych za pomocą menu

**WYJAŚNIENIE:** Menu to lista operacji, którą wyświetla program. Użytkownik może wybrać którąś z operacji, a program ją wykona.

Program sterowany za pomocą menu wyświetla na ekranie listę operacji, jakie może wykonać, i umożliwia użytkownikowi wybranie jednej z nich. Listę operacji wyświetlonych na ekranie nazywamy **menu**. Menu przykładowego programu do zarządzania internetową listą dyskusyjną może wyglądać jak na rysunku 11.1.



Rysunek 11.1. Menu

Zwróć uwagę, że przed każdą pozycją w menu pojawia się liczba. Użytkownik wybiera operację poprzez wprowadzenie widocznej przed nią liczby. Przykładowo wprowadzenie liczby 1 umożliwi użytkownikowi dodanie kolejnej osoby do listy dyskusyjnej, a wprowadzenie liczby 4 spowoduje wyświetlenie członków listy dyskusyjnej. W programach sterowanych za pomocą menu przed każdą pozycją wyświetla się jakiś znak, np. cyfra. Użytkownik programu musi wprowadzić ten znak, gdy chce, aby program wykonał określzoną operację.



**UWAGA:** W programach wyposażonych w graficzny interfejs użytkownika (GUI) użytkownik zazwyczaj wybiera operację w menu za pomocą myszy. O graficznych interfejsach użytkownika przeczytasz w rozdziale 15.

## Wybieranie pozycji w menu za pomocą struktury warunkowej

Kiedy użytkownik wybierze element z menu, program musi za pomocą struktury warunkowej wykonać odpowiednią operację. Dobrym rozwiązaniem jest w tym przypadku skorzystanie ze struktury decyzyjnej. Przejrzyjmy się prostemu przykładowi. Założymy, że chcemy stworzyć program, który będzie służył do zamiany następujących jednostek z systemu anglosaskiego na system metryczny:

- cala na centymetry,
- stopy na metry,
- mile na kilometry.

Oto wzory, za pomocą których należy dokonać konwersji:

$$\text{centymetry} = \text{cal} \cdot 2,54$$

$$\text{metry} = \text{stopy} \cdot 0,3048$$

$$\text{kilometry} = \text{mile} \cdot 1,609$$

Program powinien umożliwić użytkownikowi wybranie rodzaju konwersji, wyświetlając następujące menu:

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

Na listingu 11.1 przestawiłem pseudokod programu i cztery przykładowe wywołania programu. W liniach od 21. do 48. umieściłem strukturę decyzyjną, której zadaniem jest wykonanie wybranej przez użytkownika operacji. Zwróć uwagę na klauzulę Default w liniach od 45. do 47. Klauzula ta służy do walidacji danych wprowadzonych przez użytkownika. Jeśli użytkownik wprowadzi inną liczbę niż 1, 2 lub 3, wyświetli się komunikat błędu. Pierwsze trzy przykładowe wywołania programu pokazują, co się stanie, gdy użytkownik wprowadzi prawidłową wartość, odpowiadającą pozycji w menu. W ostatnim wywołaniu widać natomiast, co się stanie, gdy użytkownik dokona nieprawidłowego wyboru. Na rysunku 11.2 widoczny jest schemat blokowy programu.

**Listing 11.1**

```

1 // Deklarujemy zmienne, w której zapiszemy
2 // pozycję menu, którą wybrał użytkownik
3 Declare Integer menuSelection
4
5 // Deklarujemy zmienne,
6 // w których zapiszemy jednostki
7 Declare Real inches, centimeters, feet, meters,
8 miles, kilometers
9
10 // Wyświetlamy menu
11 Display "1. Zamiana cali na centymetry."
12 Display "2. Zamiana stóp na metry."
13 Display "3. Zamiana mil na kilometry."
14 Display
15
16 // Prosimy użytkownika o wybranie operacji
17 Display "Wybierz operację."
18 Input menuSelection
19
20 // Wykonujemy wybraną operację
21 Select menuSelection
22 Case 1:
23     // Zamieniamy cala na centymetry
24     Display "Wprowadź liczbę cali."
25     Input inches
26     Set centimeters = inches * 2.54
27     Display "To ", centimeters,
28         " cm."
29
30 Case 2:
31     // Zamieniamy stopy na metry
32     Display "Wprowadź liczbę stóp."
33     Input feet
34     Set meters = feet * 0.3048
35     Display "To ", meters, " m."
36
37 Case 3:
38     // Zamieniamy mile na kilometry
39     Display "Wprowadź liczbę mil."
40     Input miles
41     Set kilometers = miles * 1.609
42     Display "To ", kilometers,
43         " km."
44
45 Default:
46     // Wyświetlamy komunikat błędu
47     Display "Nieprawidłowa operacja."
48 End Select

```

**Tutaj wyświetlamy menu i prosimy użytkownika o wybór operacji. Wybrana operacja zostanie przypisana do zmiennej menuSelection.**

**Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi 1.**

**Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi 2.**

**Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi 3.**

**Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi cokolwiek innego.**

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**1 [Enter]**

Wprowadź liczbę cali.

**10 [Enter]**

To 25.4 cm.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**2 [Enter]**

Wprowadź liczbę stóp.

**10 [Enter]**

To 3.048 m.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**3 [Enter]**

Wprowadź liczbę mil.

**10 [Enter]**

To 16.09 km.

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**4 [Enter]**

Nieprawidłowa operacja.

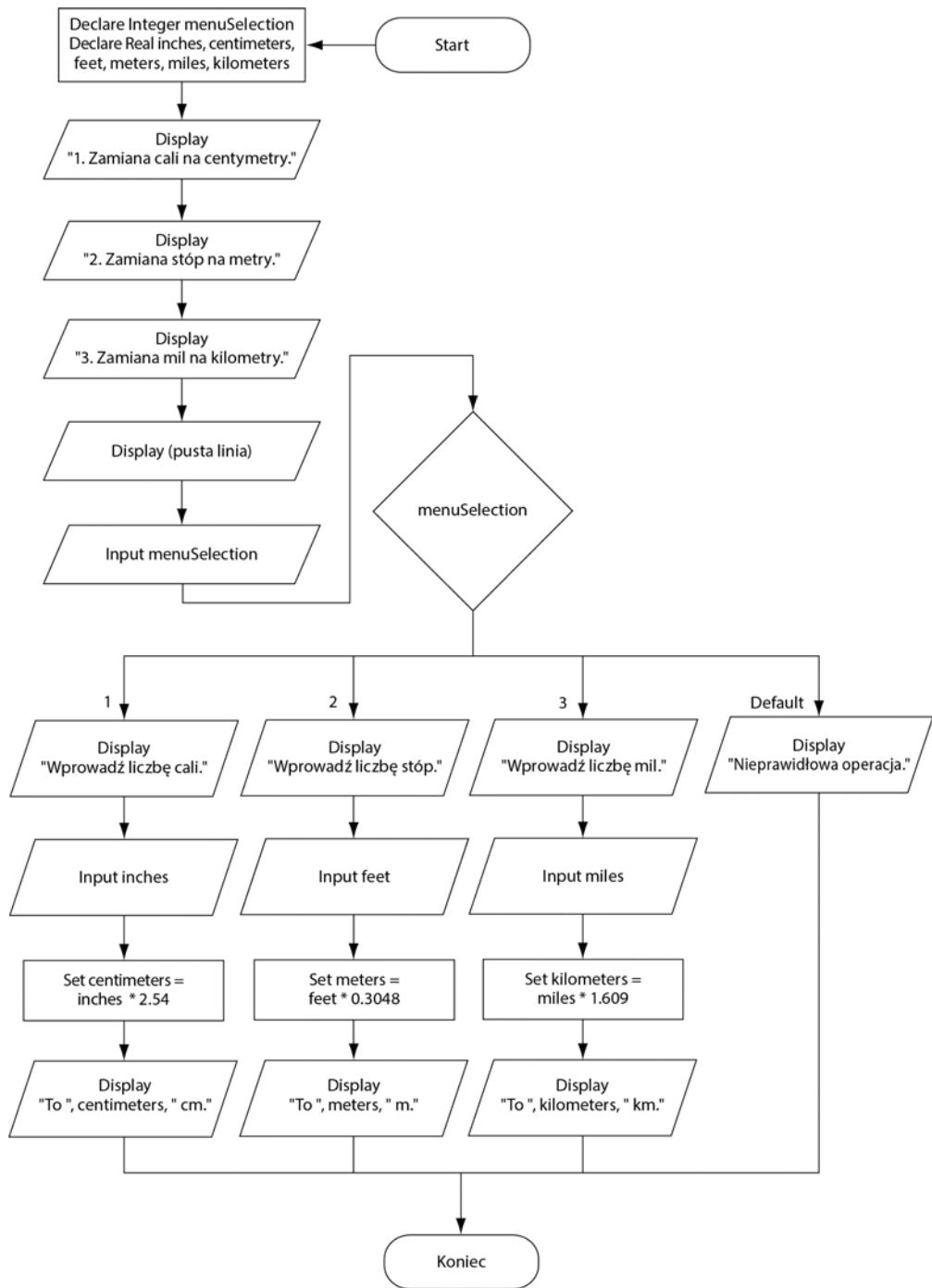
Mimo że w przypadku programów sterowanych za pomocą menu skorzystanie ze struktury decyzyjnej wydaje się najprostsze i najłatwiejsze, można także użyć innych mechanizmów. Przykładowo w programie z listingu 11.2 wykorzystałem kilka zagnieżdzonych instrukcji If-Then-Else. Na rysunku 11.3 widoczny jest schemat blokowy tego programu.



**UWAGA:** Trzecią alternatywą jest wykorzystanie w programie instrukcji If-Then-Else If.

## Walidowanie operacji wybranej przez użytkownika z menu

Program, który pozwala użytkownikowi wybrać jedną z operacji w menu, powinien także w jakiś sposób walidować wprowadzone dane. W programie z listingu 11.1 walidowałem wybraną przez użytkownika operację w strukturze decyzyjnej (w liniach od 45. do 47.) za pomocą klauzuli Default. W programie z listingu 11.2 walidowałem dane za pomocą klauzuli Else (w liniach od 43. do 45.).



Rysunek 11.2. Schemat blokowy programu z listingu 11.1

**Listing 11.2**

```

1 // Deklarujemy zmienne, w której zapiszemy
2 // pozycję menu wybraną przez użytkownika
3 Declare Integer menuSelection
4
5 // Deklarujemy zmienne,
6 // w których zapiszemy jednostki
7 Declare Real inches, centimeters, feet, meters,
8           miles, kilometers
9
10 // Wyświetlamy menu
11 Display "1. Zamiana cali na centymetry."
12 Display "2. Zamiana stóp na metry."
13 Display "3. Zamiana mil na kilometry."
14 Display
15
16 // Prosimy użytkownika o wybranie operacji
17 Display "Wybierz operację."
18 Input menuSelection
19
20 // Wykonujemy wybraną operację
21 If menuSelection == 1 Then
22     // Zamieniamy cale na centymetry
23     Display "Wprowadź liczbę cali."
24     Input inches
25     Set centimeters = inches * 2.54
26     Display "To ", centimeters,
27             " cm."
28 Else
29     If menuSelection == 2 Then
30         // Zamieniamy stopy na metry
31         Display "Wprowadź liczbę stóp."
32         Input feet
33         Set meters = feet * 0.3048
34         Display "To ", meters, " m."
35 Else
36     If menuSelection == 3 Then
37         // Zamieniamy mile na kilometry
38         Display "Wprowadź liczbę mil."
39         Input miles
40         Set kilometers = miles * 1.609
41         Display "To ", kilometers,
42                 " km."
43 Else
44     // Wyświetlamy komunikat błędu
45     Display "Nieprawidłowa operacja."
46 End If
47 End If
48 End If

```

} Tutaj wyświetlamy menu i prosimy użytkownika o wybranie operacji. Wybrana operacja zostanie przypisana do zmiennej menuSelection.

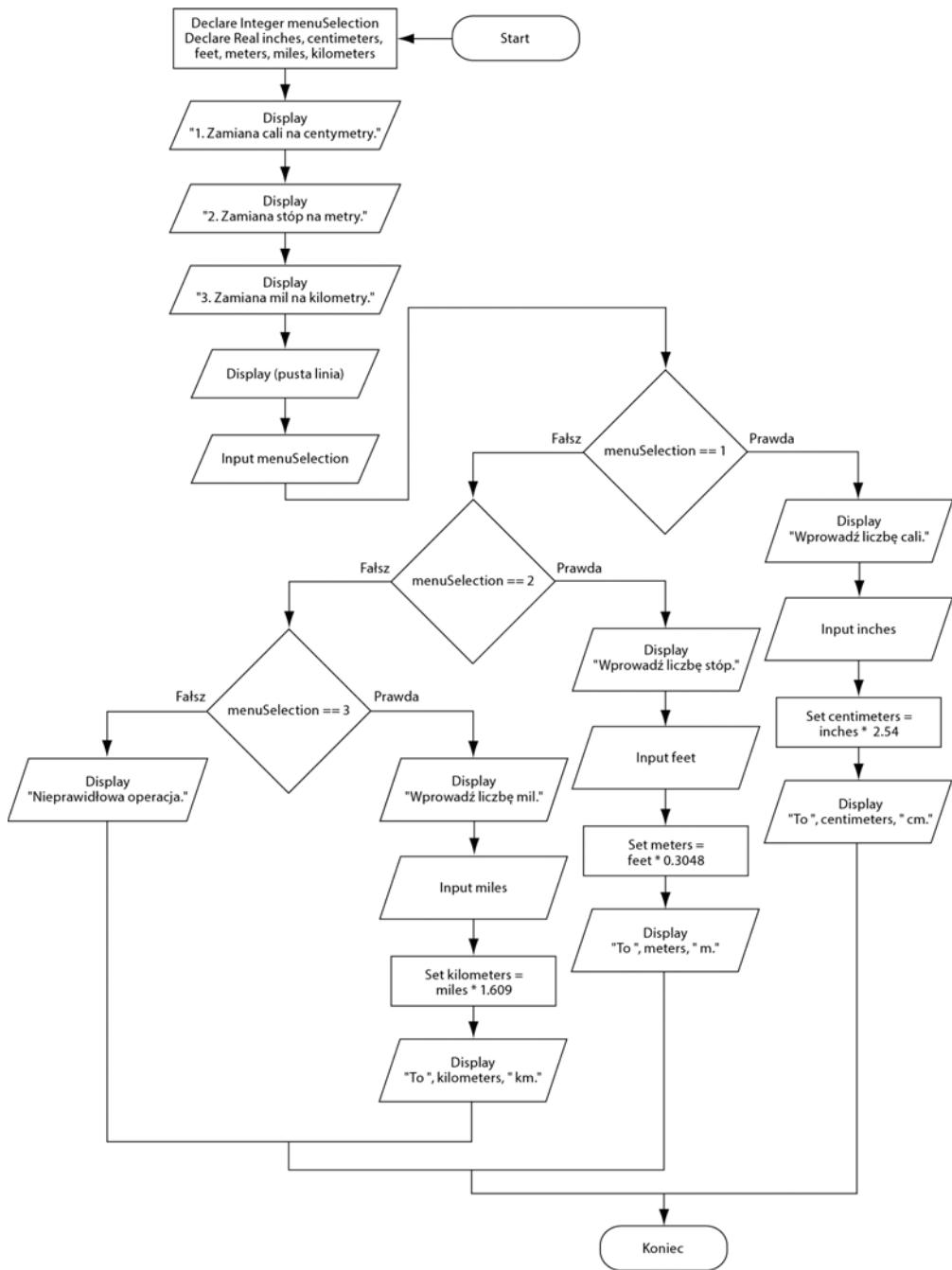
} Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi 1.

} Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi 2.

} Ten fragment kodu uruchomi się, gdy użytkownik wprowadzi 3.

} Komunikat błędu.

**Wynik działania tego programu jest taki sam jak na listingu 11.1**



**Rysunek 11.3.** Schemat blokowy programu z listingu 11.2

Alternatywnym rozwiążaniem jest walidowanie danych od razu po ich wprowadzeniu, czyli zaraz po instrukcji `Input`. Jeśli operacja jest nieprawidłowa, program wyświetli komunikat błędu i poprosi użytkownika o ponowne wprowadzenie operacji. Pętla będzie działała dopóty, dopóki użytkownik będzie wprowadzał nieprawidłową operację.

Na listingu 11.3 pokazałem, w jaki sposób można zmodyfikować program z listingu 11.1, aby validacja danych miała miejsce w pętli. Pętla walidująca dane wejściowe znajduje się w liniach od 20. do 25. Zwróć uwagę, że tym razem struktura decyzyjna nie zawiera klauzuli Default. Dzięki wykorzystaniu pętli walidacyjnej mamy pewność, że gdy program przejdzie do struktury decyzyjnej, w zmiennej menuSelection znajdzie się prawidłowa wartość z przedziału 1 – 3. Na rysunku 11.4 widoczny jest schemat blokowy programu.

### **Listing 11.3**



```

1 // Deklarujemy zmienną, w której zapiszemy
2 // pozycję menu wybraną przez użytkownika
3 Declare Integer menuSelection
4
5 // Deklarujemy zmienne,
6 // w których zapiszemy jednostki
7 Declare Real inches, centimeters, feet, meters,
8                 miles, kilometers
9
10 // Wyświetlamy menu
11 Display "1. Zamiana cali na centymetry."
12 Display "2. Zamiana stóp na metry."
13 Display "3. Zamiana mil na kilometry."
14 Display
15
16 // Prosimy użytkownika o wybranie operacji
17 Display "Wybierz operację."
18 Input menuSelection
19
20 // Walidujemy dane wprowadzone przez użytkownika
21 While menuSelection < 1 OR menuSelection > 3
22     Display "Nieprawidłowa operacja. ",
23             "Wprowadź 1, 2 lub 3."
24     Input menuSelection
25 End While
26
27 // Wykonujemy wybraną operację
28 Select menuSelection
29     Case 1:
30         // Zamieniamy cala na centymetry
31         Display "Wprowadź liczbę cali."
32         Input inches
33         Set centimeters = inches * 2.54
34         Display "To ", centimeters,
35                 " cm."
36
37     Case 2:
38         // Zamieniamy stopy na metry
39         Display "Wprowadź liczbę stóp."
40         Input feet
41         Set meters = feet * 0.3048
42         Display "To ", meters, " m."
43
44     Case 3:
45         // Zamieniamy mile na kilometry
46         Display "Wprowadź liczbę mil."
47         Input miles
48         Set kilometers = miles * 1.609

```

```

49     Display "To ", kilometers,
50             " km."
51 End Select

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**4 [Enter]**

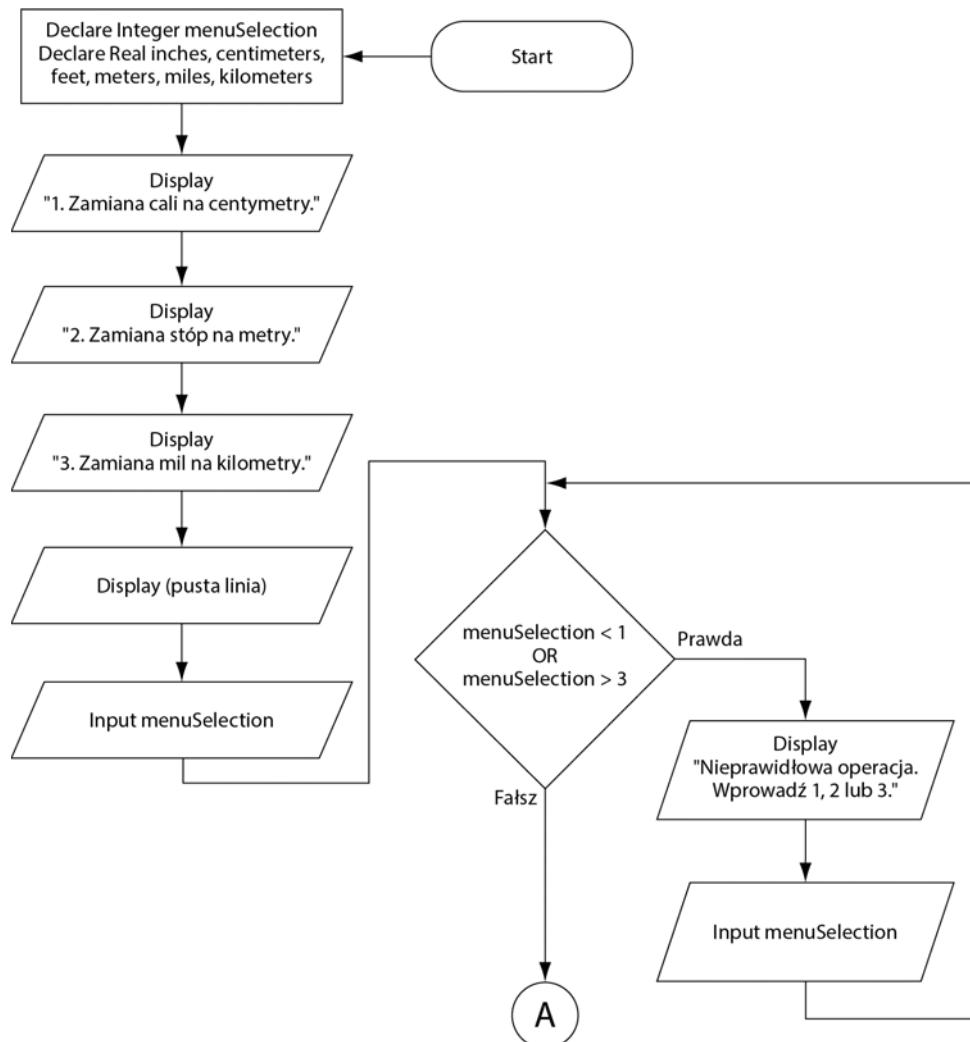
Nieprawidłowa operacja. Wprowadź 1, 2 lub 3.

**1 [Enter]**

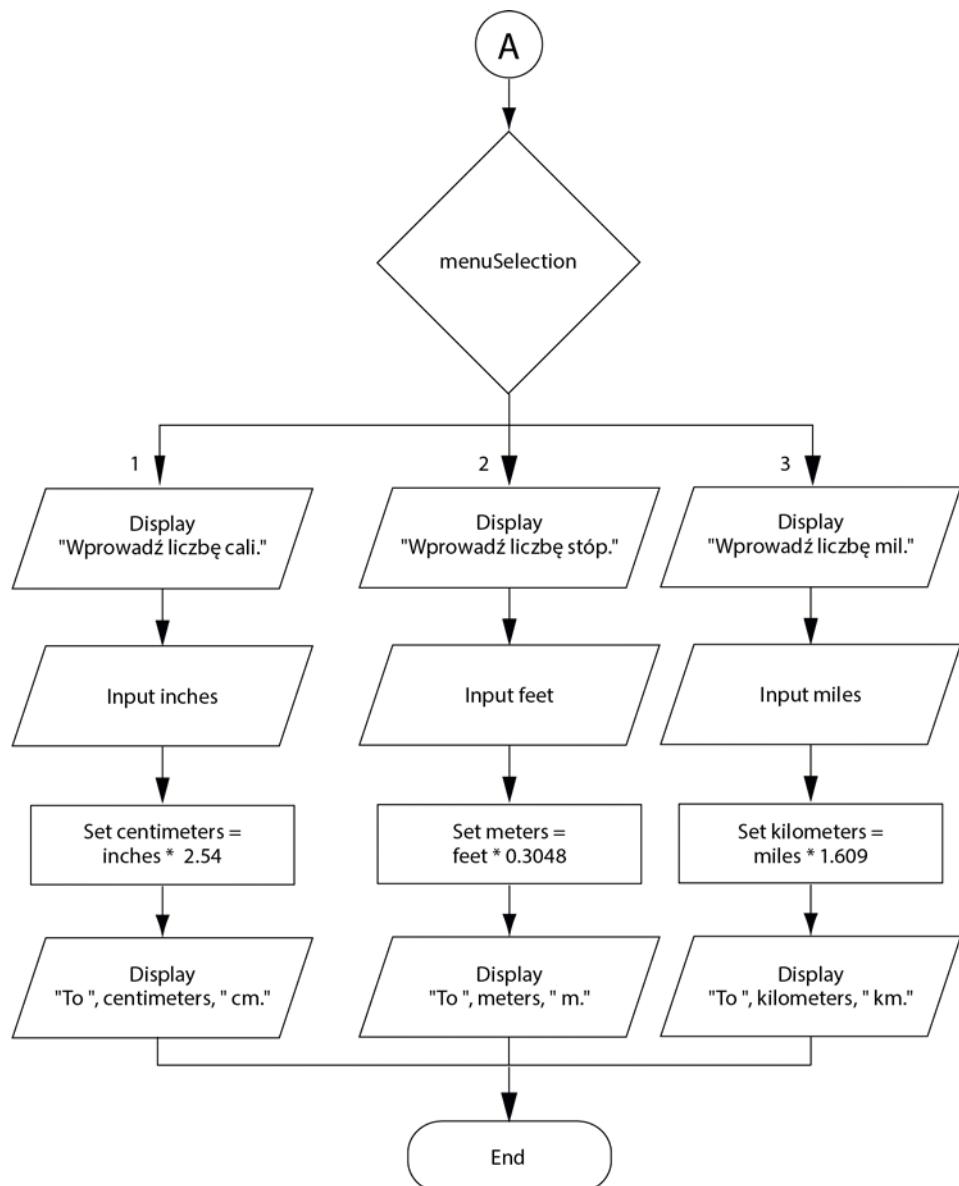
Wprowadź liczbę cali.

**10 [Enter]**

To 25.4 cm.



**Rysunek 11.4.** Schemat blokowy programu z listingu 11.3



**Rysunek 11.4.** Schemat blokowy programu z listingu 11.3 (ciąg dalszy)



## Punkt kontrolny

- 11.1. Co to jest program sterowany za pomocą menu?
- 11.2. Operacje wyświetlane w menu poprzedza się zazwyczaj liczbą, literą lub innym znakiem. W jakim celu się to robi?
- 11.3. Jakiej struktury używamy w programie, aby wykonać operację wybraną przez użytkownika w menu?

**11.2**

## Modularyzacja programu sterowanego za pomocą menu

**WYJAŚNIENIE:** Programy sterowne za pomocą menu zazwyczaj dzieli się na moduły, umieszczając każdą operację w osobnym module.

Program sterowany za pomocą menu wykonuje zazwyczaj kilka operacji, a użytkownik programu może wybrać, którą z nich program ma wykonać. W większości przypadków program sterowany za pomocą menu należy podzielić na moduły, z których każdy powinien wykonywać inną operację. Spójrz na przykładowy pseudokod na listingu 11.4. Jest on poprawioną wersją programu z listingu 11.3 — podzielilem po prostu program na mniejsze części, którymi łatwiej będzie zarządzać.

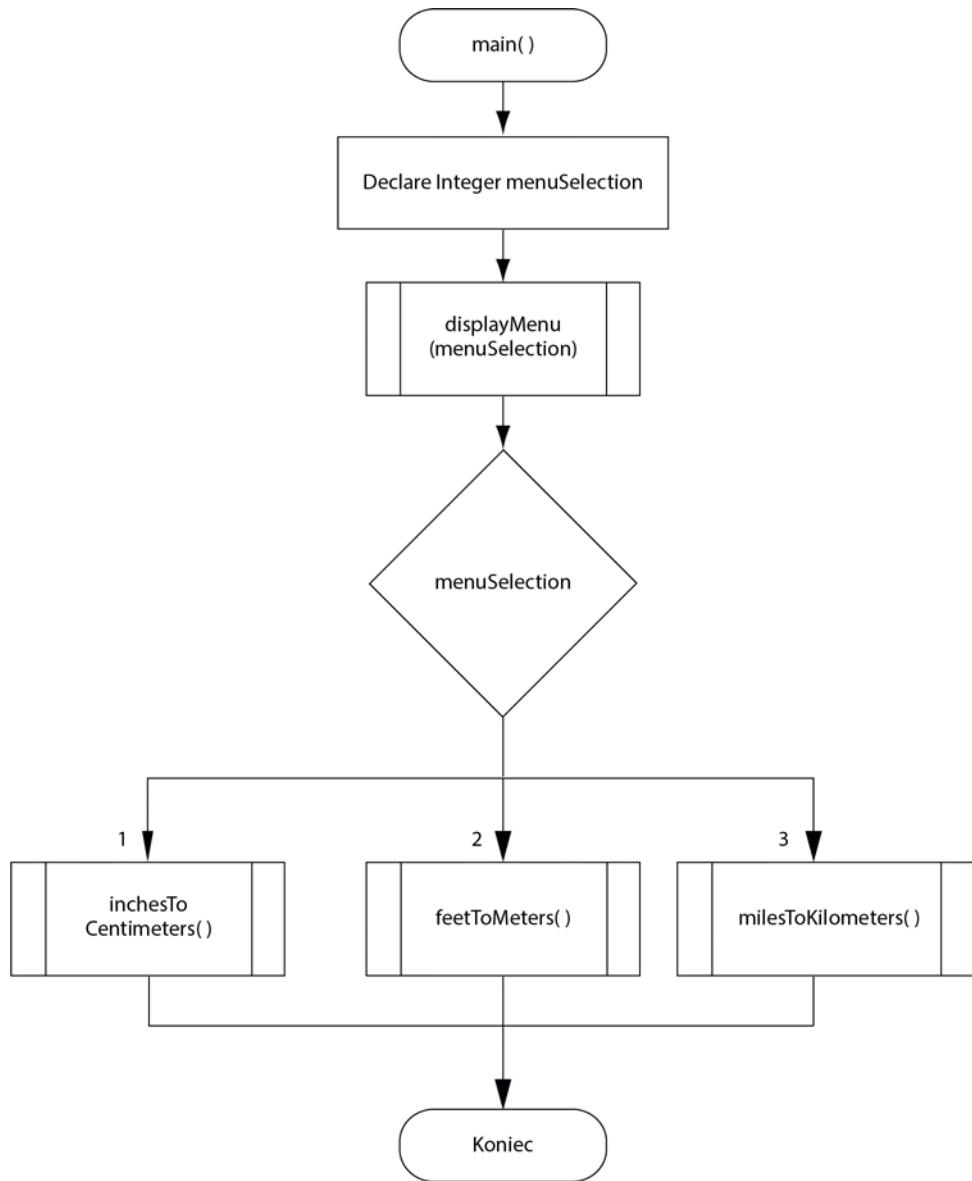
### Listing 11.4

```

1 Module main()
2   // Deklarujemy zmienną, w której zapiszemy
3   // pozycję menu wybranej przez użytkownika
4   Declare Integer menuSelection
5
6   // Wyświetlamy menu
7   // i prosimy użytkownika o wybranie operacji
8   Call displayMenu(menuSelection)
9
10  // Wykonujemy wybraną operację
11  Select menuSelection
12    Case 1:
13      Call inchesToCentimeters()
14
15    Case 2:
16      Call feetToMeters()
17
18    Case 3:
19      Call milesToKilometers()
20  End Select
21 End Module
22
23 // Moduł displayMenu wyświetla menu
24 // i prosi użytkownika o wybranie operacji. Wprowadzona wartość
25 // jest zapisywana w parametrze selection,
26 // który przekazywany jest przez referencję
27 Module displayMenu(Integer Ref selection)
28   // Wyświetlamy menu
29   Display "1. Zamiana cali na centymetry."
30   Display "2. Zamiana stóp na metry."
31   Display "3. Zamiana mil na kilometry."
32   Display
33
34   // Prosimy użytkownika o dokonanie wyboru
35   Display "Wybierz operację."
36   Input selection
37
38   // Walidujemy dane wprowadzone przez użytkownika
39   While selection < 1 OR selection > 3
40     Display "Nieprawidłowa operacja. ",
```

```
41           "Wprowadź 1, 2 lub 3."
42   Input selection
43 End While
44 End Module
45
46 // Moduł inchesToCentimeters zamienia
47 // cala na centymetry
48 Module inchesToCentimeters()
49   //Zmienne lokalne
50   Declare Real inches, centimeters
51
52   //Pobieramy liczbę cali
53   Display "Wprowadź liczbę cali."
54   Input inches
55
56   //Zamieniamy cala na centymetry
57   Set centimeters = inches * 2.54
58
59   // Wyświetlamy wynik
60   Display "To ", centimeters,
61         " cm."
62 End Module
63
64 // Moduł feetToMeters zamienia
65 // stopy na metry
66 Module feetToMeters()
67   //Zmienne lokalne
68   Declare Real feet, meters
69
70   //Pobieramy liczbę stóp
71   Display "Wprowadź liczbę stóp."
72   Input feet
73
74   //Zamieniamy stopy na metry
75   Set meters = feet * 0.3048
76
77   // Wyświetlamy wynik
78   Display "To ", meters, " m."
79 End Module
80
81 // Moduł milesToKilometers zamienia
82 // mile na kilometry
83 Module milesToKilometers()
84   //Zmienne lokalne
85   Declare Real miles, kilometers
86
87   //Pobieramy liczbę mil
88   Display "Wprowadź liczbę mil."
89   Input miles
90
91   //Zamieniamy mile na kilometry
92   Set kilometers = miles * 1.609
93
94   // Wyświetlamy wynik
95   Display "To ", kilometers,
96         " km."
97 End Module
```

**Wynik działania tego programu jest taki sam jak na [listingu 11.3](#)**



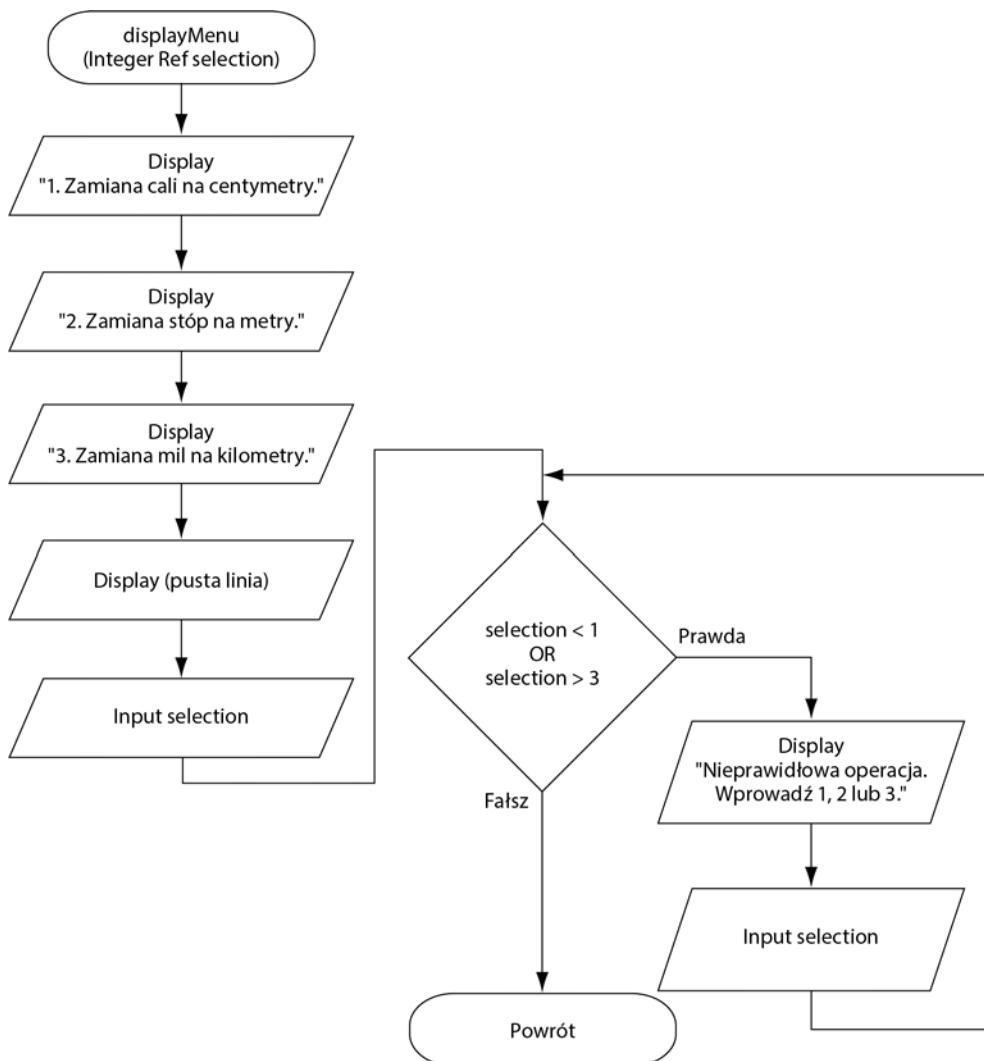
**Rysunek 11.5.** Schemat blokowy modułu main w programie z listingu 11.4

Oto informacje podsumowujące moduły, które utworzyłem w programie z listingu 11.4:

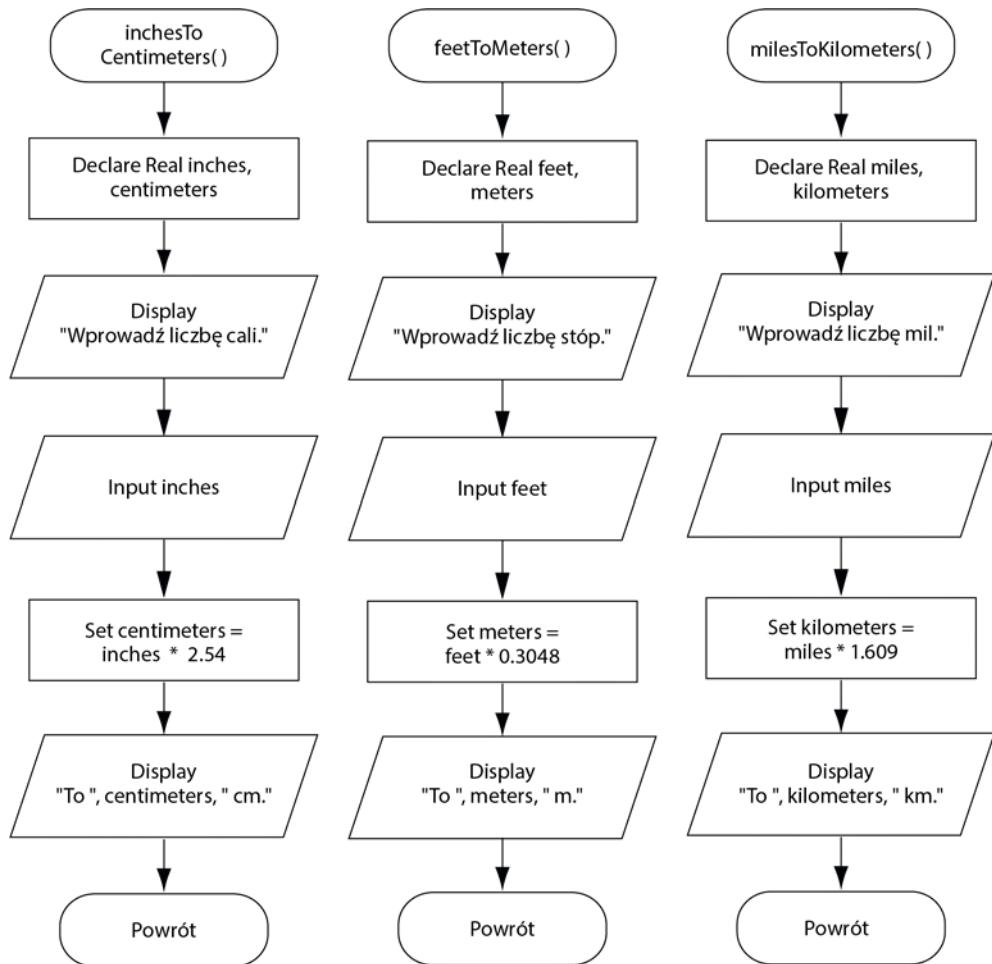
- **main** — moduł main to punkt startowy programu: wywołuję w nim inne moduły.
- **displayMenu** — zadaniem modułu displayMenu jest wyświetlenie menu na ekranie i pobranie od użytkownika operacji, którą chce wykonać.
- **inchesToCentimeters** — moduł inchesToCentimeters просi użytkownika o wprowadzenie liczby cali i wyświetla odpowiadającą tej wartości liczbę centymetrów. Moduł ten jest wywoływany w module main (w linii 13.) wtedy, gdy użytkownik wybierze operację 1.

- `feetToMeters` — moduł `feetToMeters` проси użytkownika o wprowadzenie liczby stóp i wyświetla odpowiadającą tej wartości liczbę metrów. Moduł ten jest wywoływany w module `main` (w linii 16.) wtedy, gdy użytkownik wybierze operację 2.
- `milesToKilometers` — moduł `milesToKilometers` проси użytkownika o wprowadzenie liczby mil i wyświetla odpowiadającą tej wartości liczbę kilometrów. Moduł ten jest wywoływany w module `main` (w linii 19.) wtedy, gdy użytkownik wybierze operację 3.

Na rysunku 11.5 widoczny jest schemat blokowy modułu `main`. Jeśli porównasz go ze schematem blokowym programu z listingu 11.3 (rysunek 11.4), to zauważysz, jak bardzo podzielenie programu na moduły uprościło cały projekt. Na rysunku 11.6 przedstawiłem schematy blokowe pozostałych modułów programu.



Rysunek 11.6. Schemat blokowy pozostałych modułów w programie z listingu 11.4



**Rysunek 11.6.** Schemat blokowy pozostałych modułów w programie z listingu 11.4 (ciąg dalszy)

11.3

## Ponowne wyświetlanie menu za pomocą pętli

**WYJAŚNIENIE:** W większości programów sterowanych za pomocą menu po wykonaniu operacji program ponownie wyświetla menu — wykorzystywana jest w tym celu pętla.

Programy, które do tej pory przedstawiłem w tym rozdziale, kończyły pracę zaraz po wykonaniu operacji wybranej przez użytkownika. Jeśli użytkownik chciałby następnie wybrać inną operację, musiałby ponownie uruchomić program. Takie wielokrotne uruchamianie programu jest dla użytkownika bardzo niewygodne, dlatego w większości

programów sterowanych za pomocą menu po wykonaniu operacji program za pomocą pętli ponownie wyświetla menu. Kiedy użytkownik chce zakończyć działanie programu, wybiera po prostu z menu operację *Zakończ program*.

Program na listingu 11.5 jest zmodyfikowaną wersją programu z listingu 11.4. Przedstawiłem w nim, w jaki sposób za pomocą pętli Do-While umieszczonej w module main można cyklicznie wyświetlać menu — aż do momentu, gdy użytkownik postanowi zakończyć działanie programu. Wybranie operacji 4. *Zakończ program* spowoduje zatrzymanie pętli i zakończy działanie programu. Na rysunku 11.7 przedstawiłem schemat blokowy modułu main.

### **Listing 11.5**

```

1 Module main()
2   // Deklarujemy zmienną, w której zapiszemy
3   // pozycję menu wybraną przez użytkownika
4   Declare Integer menuSelection
5
6   Do
7     // Wyświetlamy menu
8     // i prosimy użytkownika o wybranie operacji
9     Call displayMenu(menuSelection)
10
11    // Wykonujemy wybraną operację
12    Select menuSelection
13      Case 1:
14        Call inchesToCentimeters()
15
16      Case 2:
17        Call feetToMeters()
18
19      Case 3:
20        Call milesToKilometers()
21    End Select
22  While menuSelection != 4
23 End Module
24
25 // Moduł displayMenu wyświetla menu
26 // i prosi użytkownika o wybranie operacji. Wprowadzona wartość
27 // jest zapisywana w parametrze selection,
28 // który przekazywany jest przez referencję
29 Module displayMenu(Integer Ref selection)
30   // Wyświetlamy menu
31   Display "1. Zamiana cali na centymetry."
32   Display "2. Zamiana stóp na metry."
33   Display "3. Zamiana mil na kilometry."
34   Display "4. Zakończ program."
35   Display
36
37   // Prosimy użytkownika o wybranie operacji
38   Display "Wybierz operację."
39   Input selection
40

```

```
41 // Walidujemy dane wprowadzone przez użytkownika
42 While selection < 1 OR selection > 4
43     Display "Nieprawidłowa operacja. ",
44         "Wprowadź 1, 2, 3 lub 4."
45     Input selection
46 End While
47 End Module
48
49 // Moduł inchesToCentimeters zamienia
50 // cale na centymetry
51 Module inchesToCentimeters()
52     //Zmienne lokalne
53     Declare Real inches, centimeters
54
55     //Pobieramy liczbę cali
56     Display "Wprowadź liczbę cali."
57     Input inches
58
59     //Zamieniamy cale na centymetry
60     Set centimeters = inches * 2.54
61
62     // Wyświetlamy wynik
63     Display "To ", centimeters,
64         " cm."
65
66     // Wyświetlamy pustą linię
67     Display
68 End Module
69
70 // Moduł feetToMeters zamienia
71 // stopy na metry
72 Module feetToMeters()
73     //Zmienne lokalne
74     Declare Real feet, meters
75
76     //Pobieramy liczbę stóp
77     Display "Wprowadź liczbę stóp."
78     Input feet
79
80     //Zamieniamy stopy na metry
81     Set meters = feet * 0.3048
82
83     // Wyświetlamy wynik
84     Display "To ", meters, " m."
85
86     // Wyświetlamy pustą linię
87     Display
88 End Module
89
90 // Moduł milesToKilometers zamienia
91 // mile na kilometry
92 Module milesToKilometers()
93     //Zmienne lokalne
94     Declare Real miles, kilometers
95
```

```

96 // Pobieramy liczbę mil
97 Display "Wprowadź liczbę mil."
98 Input miles
99
100 // Zamieniamy mile na kilometry
101 Set kilometers = miles * 1.609
102
103 // Wyświetlamy wynik
104 Display "To ", kilometers,
105     " km."
106
107 // Wyświetlamy pustą linię
108 Display
109 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.
4. Zakończ program.

Wybierz operację.

**1 [Enter]**

Wprowadź liczbę cali.

**10 [Enter]**

To 25.4 cm.

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.
4. Zakończ program.

Wybierz operację.

**2 [Enter]**

Wprowadź liczbę stóp.

**10 [Enter]**

To 3.048 m.

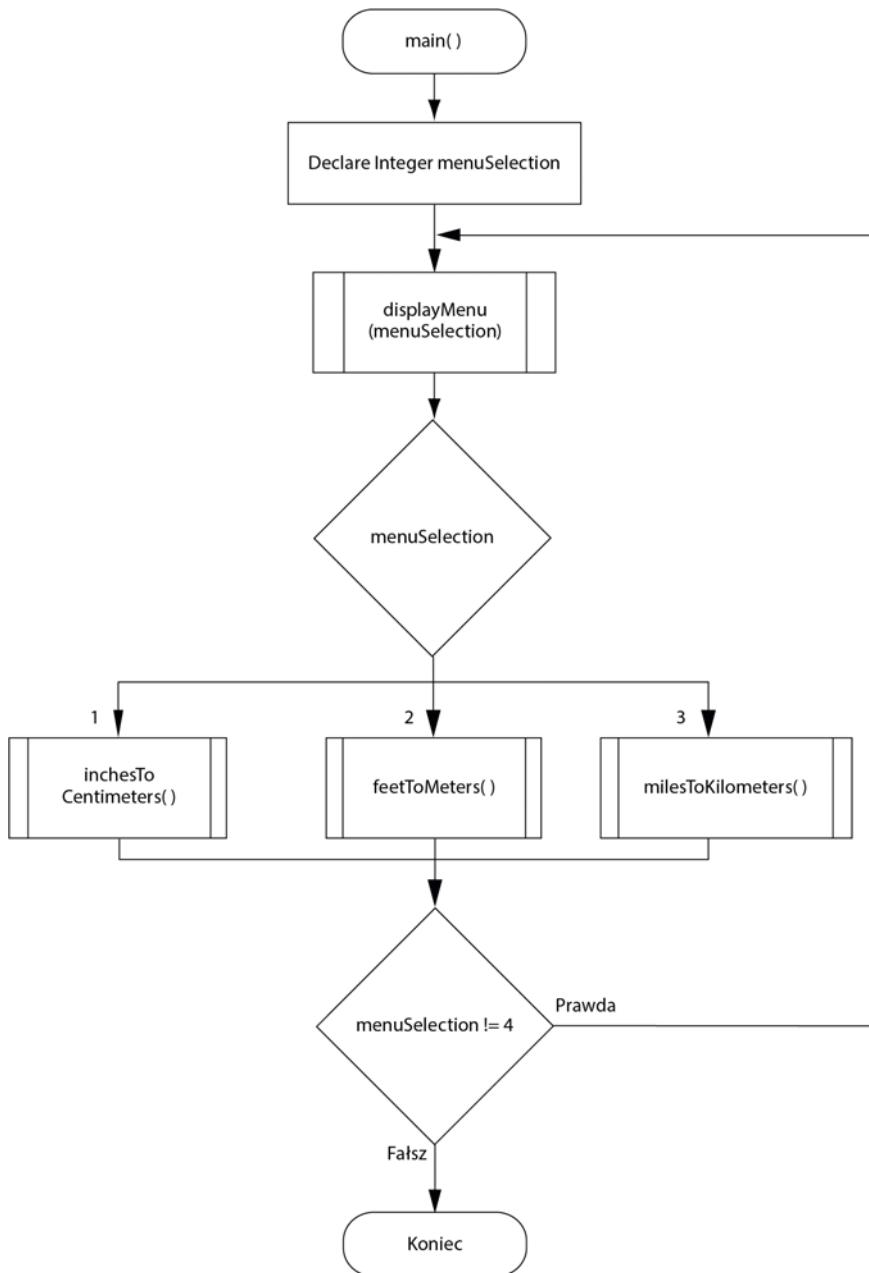
1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.
4. Zakończ program.

Wybierz operację.

**4 [Enter]**



**WSKAZÓWKA:** W programie na listingu 11.5 wykorzystałem pętlę Do-While, ponieważ jest to pętla, w której warunek sprawdzany jest na końcu — sprawi to, że menu wyświetli się co najmniej jednokrotnie. Mógłbym także wykorzystać pętlę While, jednak jest to pętla, w której warunek jest sprawdzany na początku, więc musiałbym także zainicjalizować zmienną menuSelection wartością różną od 4.



Rysunek 11.7. Schemat blokowy modułu main w programie z listingu 11.5



## W centrum uwagi

### Projektowanie programu sterowanego za pomocą menu

W kilku sekcjach „W centrum uwagi” zamieszczonych w rozdziale 10. miałeś okazję zaprojektować zestaw programów dla firmy Midnight Coffee Roasters. Programy te służyły głównie do kontrolowania stanu magazynowego kawy. Każdy dostępny rodzaj kawy miał odpowiadający mu rekord w pliku. Każdy rekord składał się z pól zawierających nazwę kawy i jej ilość. Programy, które przedstawiłem w rozdziale 10., wykonywały następujące operacje:

- dodawanie rekordu do pliku;
- wyszukiwanie rekordu;
- modyfikowanie wartości ilości w istniejącym już rekordzie;
- usuwanie rekordu z pliku;
- wyświetlanie wszystkich rekordów zapisanych w pliku.

Obecnie każda z tych operacji wykonywana jest przez oddzielny program. Julia, właścicielka firmy, poprosiła Cię, abyś zebrał wszystkie operacje i umieścił w jednym programie, wyposażonym w menu.

Postanowileś więc zaprojektować program, w którym będą się znajdowały następujące moduły:

- `main` — uruchomi się wraz ze startem programu. Korzystam w nim z pętli, w której wywołuję odpowiednie moduły wyświetlające menu, pobierające od użytkownika dane i wykonujące wybraną operację.
- `displayMenu` — wyświetla następujące menu:

```
Menu stanu magazynowego
1. Dodaj rekord.
2. Wyszukaj rekord.
3. Zmodyfikuj rekord.
4. Usuń rekord.
5. Wyświetl wszystkie rekordy.
6. Zakończ program
```

W module `displayMenu` pobieram także operację wybraną przez użytkownika i waliduję wprowadzone dane.

- `addRecord` — jest wywoływany, gdy użytkownik wybierze w menu operację 1. Umożliwia on użytkownikowi dodanie rekordu do pliku.
- `searchRecord` — jest wywoływany, gdy użytkownik wybierze w menu operację 2. Umożliwia on użytkownikowi wyszukanie konkretnego rekordu.
- `modifyRecord` — jest wywoływany, gdy użytkownik wybierze w menu operację 3. Umożliwia on użytkownikowi zmodyfikowanie ilości wybranego rodzaju kawy zapisanego w istniejącym już rekordzie.
- `deleteRecord` — jest wywoływany, gdy użytkownik wybierze w menu operację 4. Umożliwia on użytkownikowi usunięcie rekordu z pliku.
- `displayRecords` — jest wywoływany, gdy użytkownik wybierze w menu operację 5. Wyświetla on wszystkie rekordy zapisane w pliku.

Na listingu 11.6 przedstawiłem pseudokod modułu `main`, a na rysunku 11.8 jego schemat blokowy.

### **Listing 11.6 Program do zarządzania stanem magazynowym kawy: moduł main**

```

1 Module main()
2   // Zmienna, w której zapiszemy pozycję menu wybraną przez użytkownika
3   Declare Integer menuSelection
4
5   Do
6     // Wyświetlamy menu
7     Call displayMenu(menuSelection)
8
9     // Wykonujemy wybraną operację
10    Select menuSelection
11      Case 1:
12        Call addRecord()
13
14      Case 2:
15        Call searchRecord()
16
17      Case 3:
18        Call modifyRecord()
19
20      Case 4:
21        Call deleteRecord()
22
23      Case 5:
24        Call displayRecords()
25    End Select
26  While menuSelection != 6
27 End Module
28

```

W następnej kolejności omówię pseudokod modułu `displayMenu`. Na rysunku 11.9 widoczny jest jego schemat blokowy.

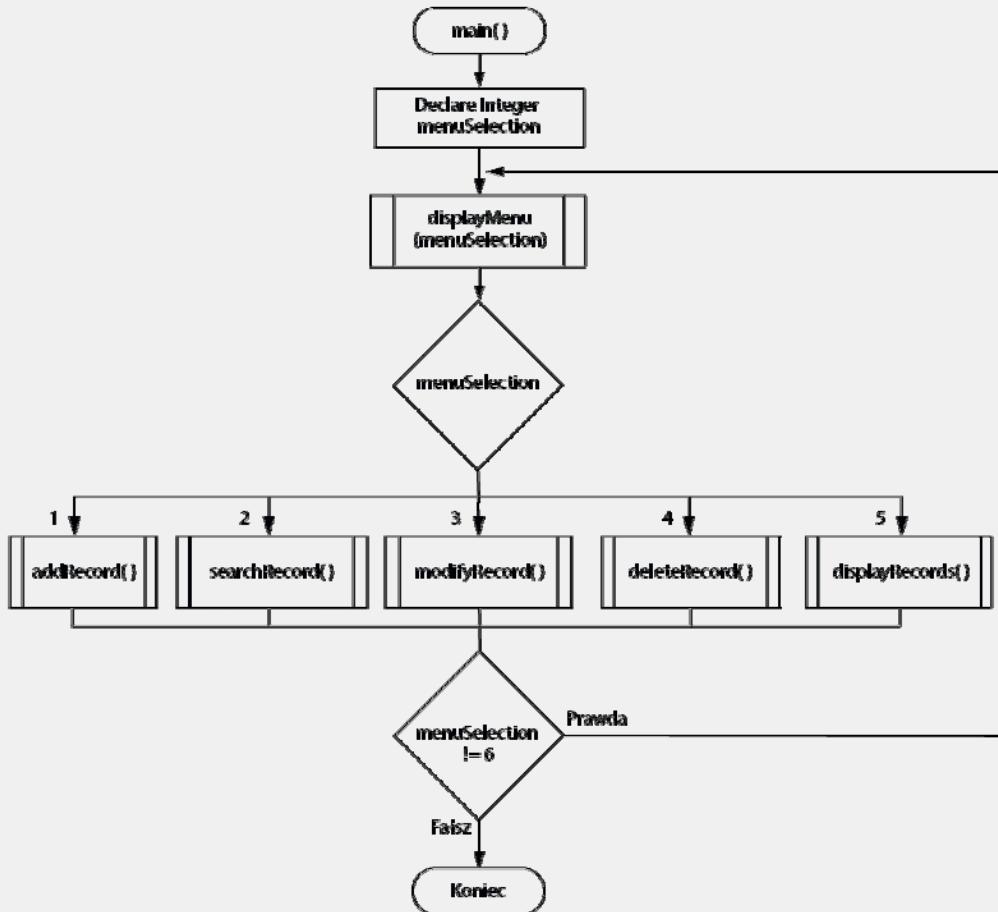
### **Listing 11.6. Program do zarządzania stanem magazynowym kawy (kontynuacja): moduł displayMenu**

```

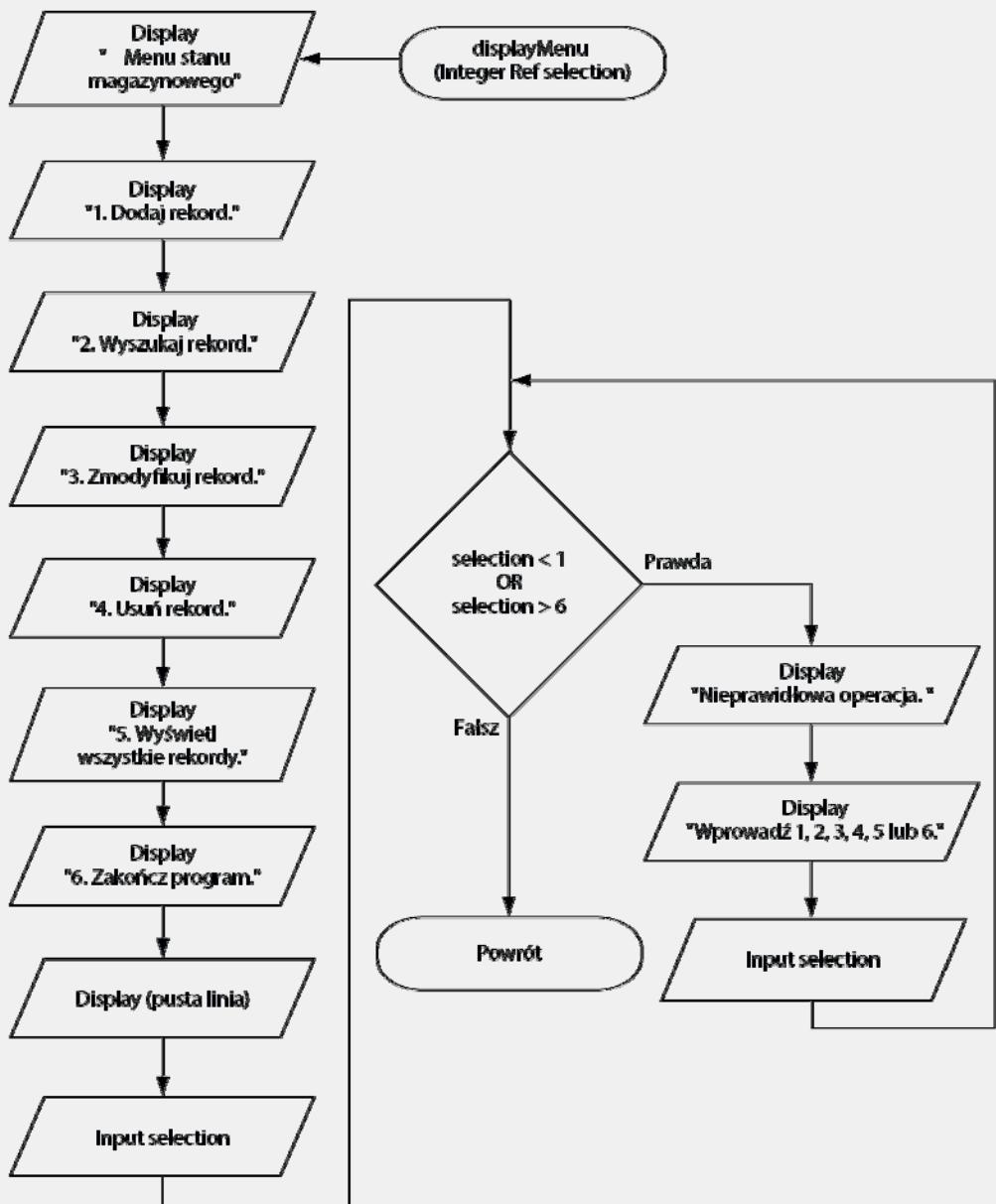
29 // Moduł displayMenu wyświetla menu, prosi użytkownika o wybranie operacji
30 // i waliduje wprowadzone dane
31 Module displayMenu(Integer Ref selection)
32   // Wyświetlamy menu
33   Display " Menu stanu magazynowego"
34   Display "1. Dodaj rekord."
35   Display "2. Wyszukaj rekord."
36   Display "3. Zmodyfikuj rekord."
37   Display "4. Usuń rekord."
38   Display "5. Wyświetl wszystkie rekordy."
39   Display "6. Zakończ program."
40   Display
41
42   // Prosimy użytkownika o wybranie operacji
43   Display "Wybierz operację."
44   Input selection

```

```
45 // Walidujemy dane wprowadzone przez użytkownika
46 While selection < 1 OR selection > 6
47     Display "Nieprawidłowa operacja. ",
48     Display "Wprowadź 1, 2, 3, 4, 5 lub 6."
49     Input selection
50
51 End While
52 End Module
53
```



Rysunek 11.8. Schemat blokowy modułu main w programie z listingu 11.6

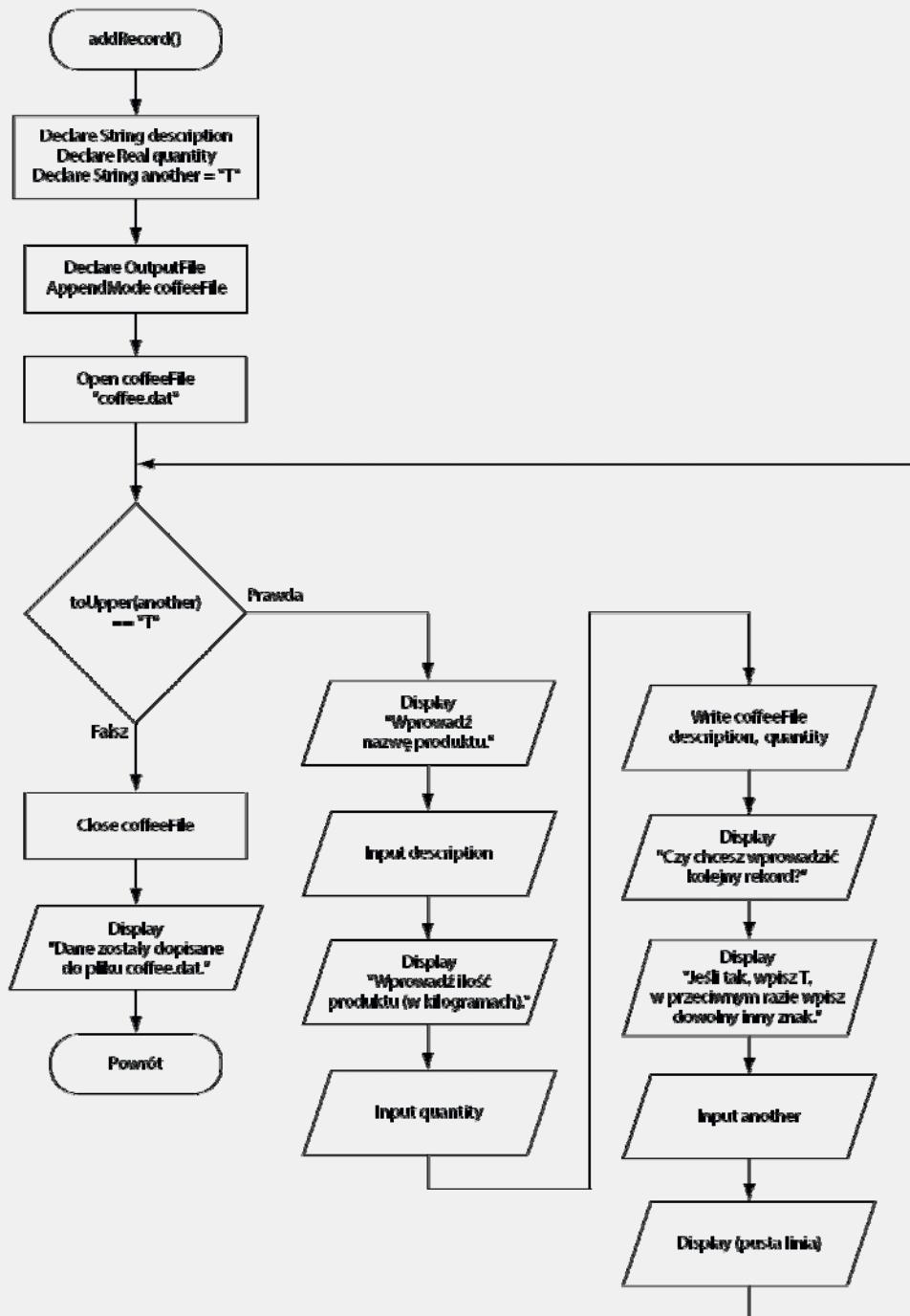


**Rysunek 11.9.** Schemat blokowy modułu `displayMenu` w programie z listingu 11.6

W następnej kolejności zaprezentuję moduł addRecord. Na rysunku 11.10 widoczny jest jego schemat blokowy.

**Listing 11.6 Program do zarządzania stanem magazynowym kawy (kontynuacja):  
moduł addRecord**

```
54 // Moduł addRecord umożliwia dodanie
55 // rekordu do pliku
56 Module addRecord()
57     // Zmienne, w których zapiszemy pola rekordu
58     Declare String description
59     Declare Real quantity
60
61     // Zmienna sterująca pętlą
62     Declare String another = "T"
63
64     // Deklarujemy plik wyjściowy w trybie dołączania
65     Declare OutputFile AppendMode coffeeFile
66
67     // Otwieramy plik
68     Open coffeeFile "coffee.dat"
69
70     While toUpper(another) == "T"
71         // Pobieramy opis
72         Display "Wprowadź nazwę produktu."
73         Input description
74
75         // Pobieramy zapas produktu
76         Display "Wprowadź ilość produktu ",
77             "(w kilogramach)."
78         Input quantity
79
80         // Dołączmy rekord do pliku
81         Write coffeeFile description, quantity
82
83         // Sprawdzamy, czy użytkownik ma zamiar wprowadzić
84         // kolejny rekord
85         Display "Czy chcesz wprowadzić kolejny rekord?"
86         Display "Jeśli tak, wpisz T, w przeciwnym razie wpisz dowolny inny znak."
87         Input another
88
89         // Wyświetlamy pustą linię
90         Display
91     End While
92
93     // Zamkamy plik
94     Close coffeeFile
95     Display "Dane zostały dopisane do pliku coffee.dat."
96 End Module
97
```



**Rysunek 11.10.** Schemat blokowy modułu `addRecord` w programie z listingu 11.6

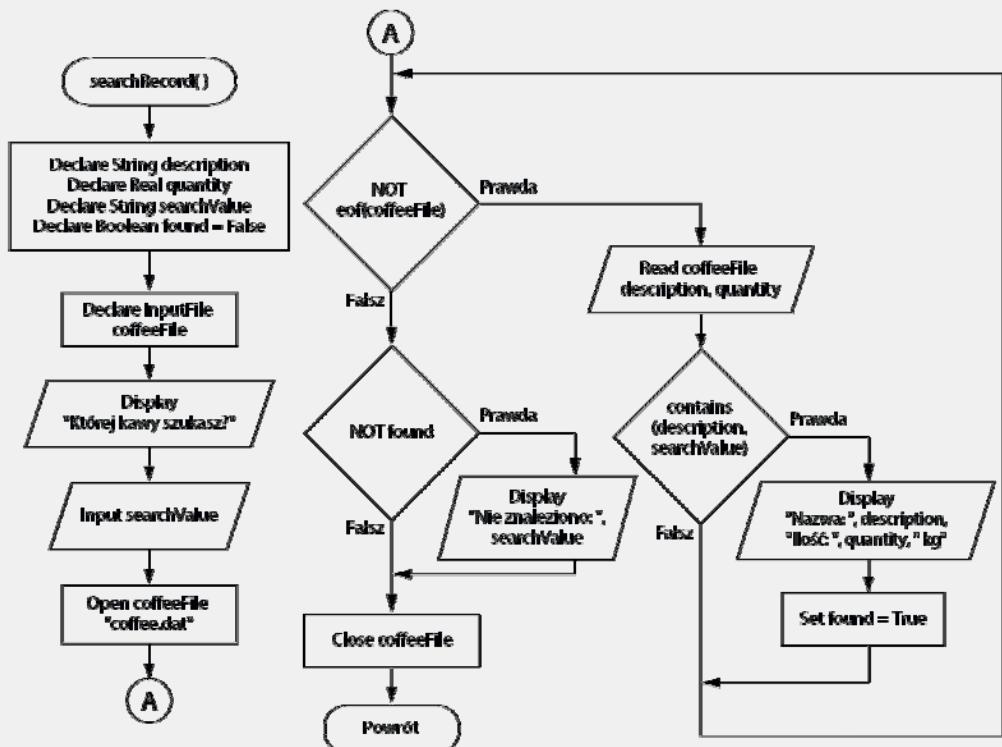
Poniżej znajduje się pseudokod modułu searchRecord. Na rysunku 11.11 widoczny jest jego schemat blokowy.

**Listing 11.6. Program do zarządzania stanem magazynowym kawy (kontynuacja): moduł searchRecord**

```

98 // Moduł searchRecord umożliwia wyszukanie
99 // określonego rekordu w pliku
100 Module searchRecord()
101     // Zmienne, w których zapiszemy pola rekordu
102     Declare String description
103     Declare Real quantity
104
105     // Zmienna, w której zapiszemy szukaną wartość
106     Declare String searchValue
107
108     // Flaga wskazująca, czy wartość została odnaleziona
109     Declare Boolean found = False
110
111     // Deklarujemy plik wejściowy
112     Declare InputFile coffeeFile
113
114     // Pobieramy szukaną wartość
115     Display "Której kawy szukasz?"
116     Input searchValue
117
118     // Otwieramy plik
119     Open coffeeFile "coffee.dat"
120
121     While NOT eof(coffeeFile)
122         // Odczytujemy rekord z pliku
123         Read coffeeFile description, quantity
124
125         // Jeśli rekord zawiera szukaną wartość,
126         // wyświetlamy go
127         If contains(description, searchValue) Then
128             // Wyświetlamy rekord
129             Display "Nazwa: ", description,
130                 "Ilość: ", quantity, " kg"
131
132         // Ustawiamy flagę na True
133         Set found = True
134     End If
135 End While
136
137     // Jeśli szukanej wartości nie udało się odnaleźć,
138     // wyświetlamy odpowiedni komunikat
139     If NOT found Then
140         Display "Nie znaleziono: ", searchValue
141     End If
142
143     // Zamkamy plik
144     Close coffeeFile
145 End Module
146

```



**Rysunek 11.11.** Schemat blokowy modułu searchRecord w programie z listingu 11.6

Poniżej znajduje się pseudokod modułu modifyRecord. Na rysunkach 11.12 i 11.13 widoczny jest jego schemat blokowy.

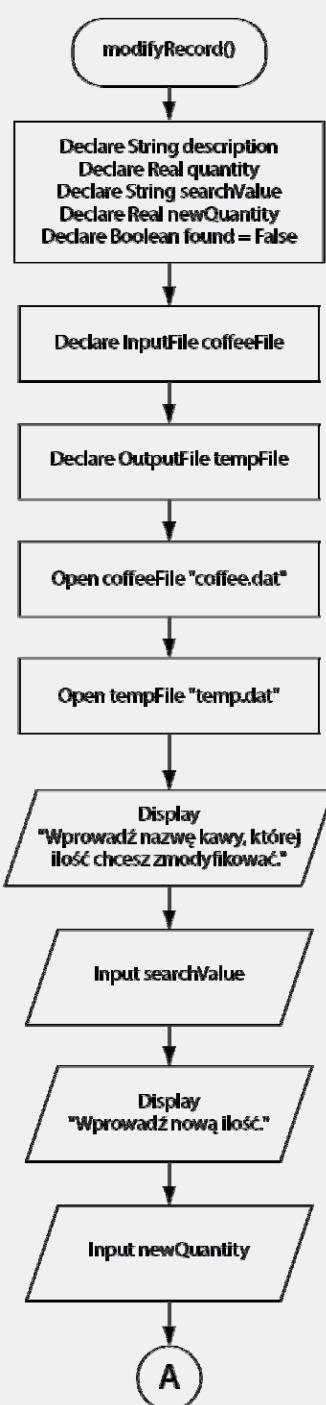
#### **Listing 11.6 Program do zarządzania stanem magazynowym kawy (kontynuacja): moduł modifyRecord**

```

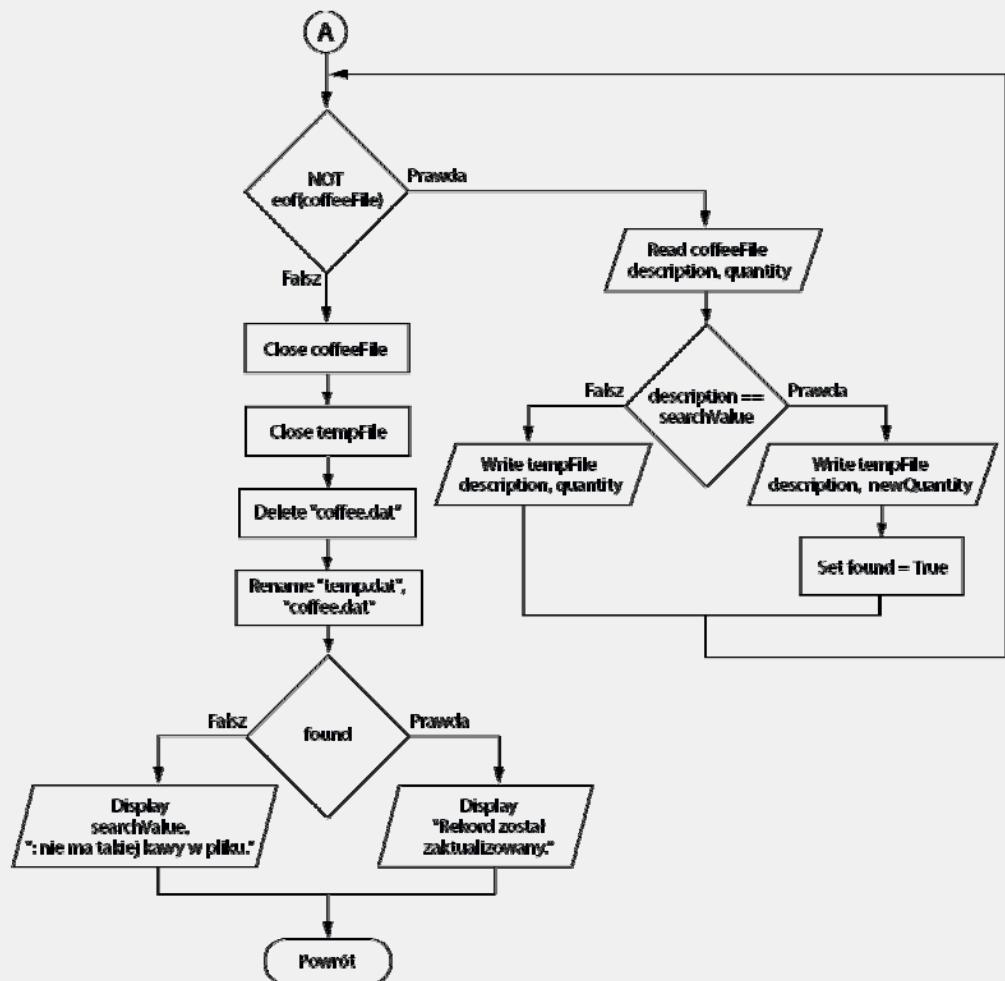
147 // Moduł modifyRecord umożliwia modyfikację
148 // istniejącego już rekordu
149 Module modifyRecord()
150     // Zmienne, w których zapiszemy pola rekordu
151     Declare String description
152     Declare Real quantity
153
154     // Zmienna, w której zapiszemy szukaną wartość
155     Declare String searchValue
156
157     // Zmienna, w której zapiszemy nową ilość produktu
158     Declare Real newQuantity
159
160     // Flaga wskazująca, czy wartość została odnaleziona
161     Declare Boolean found = False
162
163     // Deklarujemy plik wejściowy
164     Declare InputFile coffeeFile
165
166     // Deklarujemy plik wyjściowy, do którego skopiujemy
167     // plik pierwotny

```

```
168 Declare outputFile tempFile
169
170 // Otwieramy pliki
171 Open coffeeFile "coffee.dat"
172 Open tempFile "temp.dat"
173
174 // Pobieramy szukaną wartość
175 Display "Wprowadź nazwę kawy, której ilość chcesz zmodyfikować."
176 Input searchValue
177
178 // Pobieramy nowy zapas produktu
179 Display "Wprowadź nową ilość."
180 Input newQuantity
181
182 While NOT eof(coffeeFile)
183     // Odczytujemy rekord z pliku
184     Read coffeeFile description, quantity
185
186     // Gdy rekord zawiera szukaną wartość, zapisujemy w pliku tymczasowym nowy rekord
187     // W przeciwnym razie w pliku tymczasowym zapisujemy
188     // odczytany rekord
189     If description == searchValue Then
190         Write tempFile description, newQuantity
191         Set found = True
192     Else
193         Write tempFile description, quantity
194     End If
195 End While
196
197 // Zamkamy oba pliki
198 Close coffeeFile
199 Close tempFile
200
201 // Usuwamy plik pierwotny
202 Delete "coffee.dat"
203
204 // Zmieniamy nazwę pliku tymczasowego
205 Rename "temp.dat", "coffee.dat"
206
207 // Wyświetlamy informację, czy operacja się powiodła
208 If found Then
209     Display "Rekord został zaktualizowany."
210 Else
211     Display searchValue, ": nie ma takiej kawy w pliku."
212 End If
213 End Module
214
```



**Rysunek 11.12.** Pierwszy fragment schematu blokowego modułu `modifyRecord` w programie z listingu 11.6

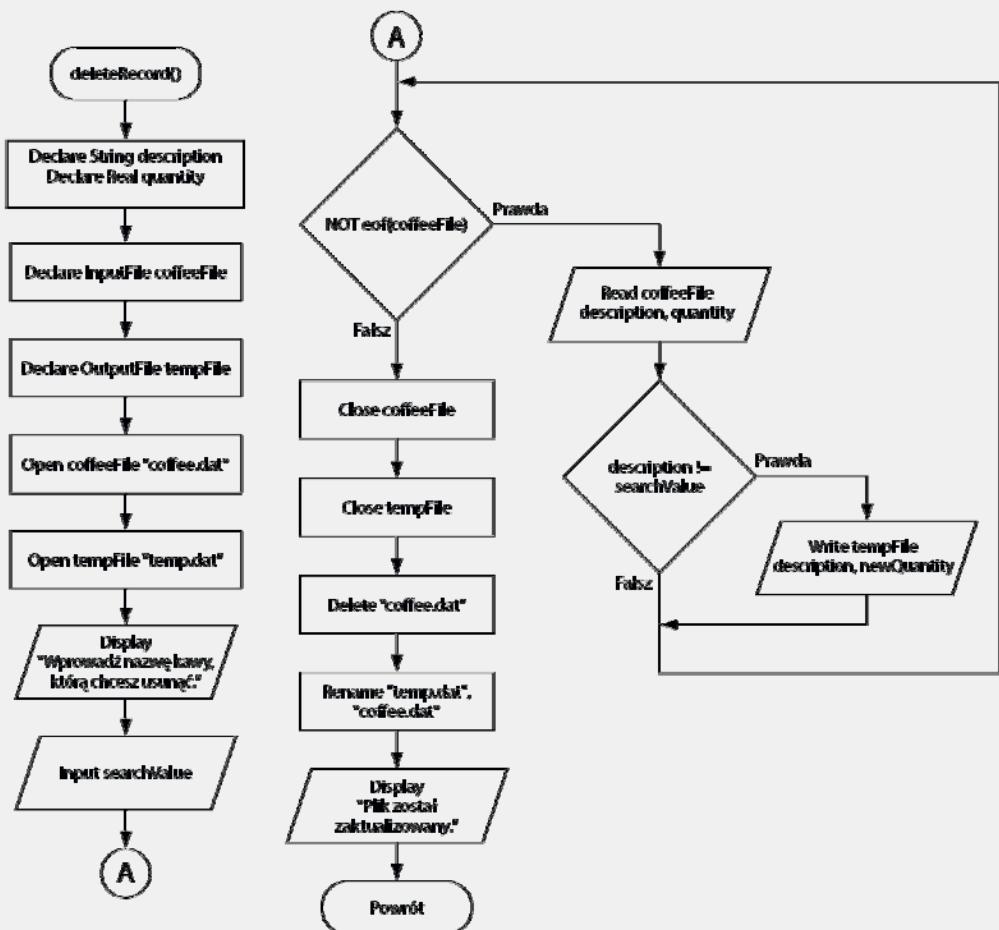


**Rysunek 11.13.** Drugi fragment schematu blokowego modułu `modifyRecord` w programie z listingu 11.6

Poniżej znajduje się pseudokod modułu `deleteRecord`. Na rysunku 11.14 widoczny jest jego schemat blokowy.

**Listing 11.6****Program do zarządzania stanem magazynowym kawy (kontynuacja):  
moduł deleteRecord**

```
215 // Moduł deleteRecord umożliwia usunięcie
216 // rekordu zapisanego w pliku
217 Module deleteRecord()
218     // Zmienne, w których zapiszemy pola rekordu
219     Declare String description
220     Declare Real quantity
221
222     // Zmienna, w której zapiszemy szukaną wartość
223     Declare String searchValue
224
225     // Deklarujemy plik wejściowy
226     Declare InputFile coffeeFile
227
228     // Deklarujemy plik wyjściowy, do którego skopiujemy
229     // plik pierwotny
230     Declare OutputFile tempFile
231
232     // Otwieramy oba pliki
233     Open coffeeFile "coffee.dat"
234     Open tempFile "temp.dat"
235
236     // Pobieramy szukaną wartość
237     Display "Wprowadź nazwę kawy, którą chcesz usunąć."
238     Input searchValue
239
240     While NOT eof(coffeeFile)
241         // Odczytujemy rekord z pliku
242         Read coffeeFile description, quantity
243
244         // Jeśli nie jest to rekord, który mamy zamiar usunąć,
245         // zapisujemy go w pliku tymczasowym
246         If description != searchValue Then
247             Write tempFile description, newQuantity
248         End If
249     End While
250
251     // Zamkamy oba pliki
252     Close coffeeFile
253     Close tempFile
254
255     // Usuwany plik pierwotny
256     Delete "coffee.dat"
257
258     // Zmieniamy nazwę pliku tymczasowego
259     Rename "temp.dat", "coffee.dat"
260
261     Display "Plik został zaktualizowany."
262 End Module
263
```



Rysunek 11.14. Schemat blokowy modułu `deleteRecord` w programie z listingu 11.6

Poniżej znajduje się pseudokod modułu `displayRecords`. Na rysunku 11.15 widoczny jest jego schemat blokowy.

#### **Listing 11.6.**

#### **Program do zarządzania stanem magazynowym kawy (kontynuacja): moduł `displayRecords`**

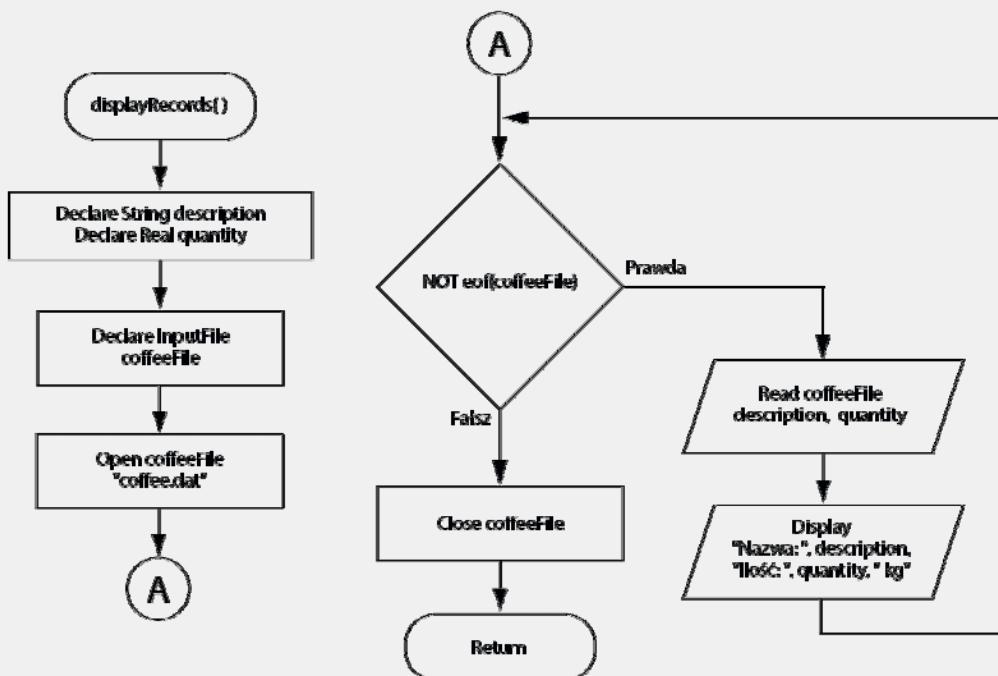
```

264 // Moduł displayRecords wyświetla wszystkie rekordy
265 // zapisane w pliku
266 Module displayRecords()
267   // Zmienne, w których zapiszemy pola rekordu
268   Declare String description
269   Declare Real quantity
270
271   // Deklarujemy plik wejściowy
272   Declare InputFile coffeeFile
273
274   // Otwieramy plik
275   Open coffeeFile "coffee.dat"
  
```

```

276 While NOT eof(coffeeFile)
277     // Odczytujemy rekord z pliku
278     Read coffeeFile description, quantity
279
280     // Wyświetlamy rekord
281     Display "Nazwa: ", description,
282             "Ilość: ", quantity, " kg"
283
284 End While
285
286 // Zamkamy plik
287 Close coffeeFile
288 End Module

```



Rysunek 11.15. Schemat blokowy modułu `displayRecords()` w programie z listingu 11.6



## Punkt kontrolny

- 11.4. Wyjaśnij, dlaczego w większości programów sterowanych za pomocą menu do ponownego wyświetlania menu wykorzystuje się pętlę.
- 11.5. W jaki sposób użytkownik może zakończyć działanie programu, gdy menu programu wyświetla się w sposób cykliczny za pomocą pętli?

**11.4**

## Menu wielopoziomowe

**WYJAŚNIENIE:** Menu wielopoziomowe składa się z menu głównego i co najmniej jednego podmenu.

Programy, które przedstawiłem w tym rozdziale, były na tyle proste, że wszystkie operacje mogliśmy umieścić w pojedynczym menu. Gdy użytkownik wybrał jedną z operacji w menu, program ją natychmiast wykonał, a następnie ponownie wyświetlił menu (lub zakończył działanie, gdy nie skorzystaliśmy z pętli). Takie menu nazywamy **menu jednopoziomowym**.

Bardzo często jednak programy są tak złożone, że jedno menu nie wystarczy, aby pomieścić wszystkie dostępne operacje. Założmy, że projektujesz program dla firmy, która będzie za jego pomocą wykonywała następujące operacje:

1. Analiza sprzedaży.
2. Analiza zyskowności.
3. Dodawanie rekordu do asortymentu.
4. Wyszukiwanie rekordu w asortymencie.
5. Modyfikowanie rekordu w asortymencie.
6. Usuwanie rekordu z asortymentu.
7. Drukowanie raportu z asortymentem.
8. Drukowanie asortymentu uporządkowanego według kosztu.
9. Drukowanie asortymentu uporządkowanego według wieku.
10. Drukowanie asortymentu uporządkowanego według ceny detalicznej.

Ponieważ lista operacji jest bardzo dłużna, wyświetlanie ich w jednym menu nie jest najlepszym pomysłem. Użytkownicy takiego programu mieliby kłopot z przeglądaniem tak rozbudowanego menu, zawierającego aż tyle operacji.

Znacznie lepszym pomysłem jest zastosowanie menu wielopoziomowego. Program, w którym wykorzystuje się **menu wielopoziomowe**, zaraz po uruchomieniu wyświetla **menu główne** (na którym są widoczne tylko wybrane elementy), a gdy użytkownik wybierze którąś z pozycji, program wyświetla mniejsze **podmenu**. Przykładowe menu główne może wyglądać następująco:

Menu główne

1. Analiza sprzedaży i zyskowności
2. Aktualizowanie asortymentu
3. Drukowanie raportów
4. Zakończ program

Kiedy użytkownik wybierze w menu głównym pozycję 1., wyświetli się następujące podmenu:

Menu analizy sprzedaży i zyskowności

1. Analiza sprzedaży
2. Analiza zyskowności
3. Powrót do menu głównego

Kiedy użytkownik wybierze w menu głównym pozycję 2., wyświetli się następujące podmenu:

Menu aktualizowania asortymentu

1. Dodawanie rekordu
2. Wyszukiwanie rekordu
3. Modyfikowanie rekordu
4. Usuwanie rekordu
5. Powrót do menu głównego

Kiedy użytkownik wybierze w menu głównym pozycję 3., wyświetli się następujące podmenu:

Menu drukowania raportów

1. Drukuj raport asortymentu
2. Drukuj asortyment uporządkowany według kosztu
3. Drukuj asortyment uporządkowany według wieku
4. Drukuj asortyment uporządkowany według ceny detalicznej
5. Powrót do menu głównego

Przyjrzyjmy się teraz, w jaki sposób można zaprojektować taki program. Nie będę tutaj omawiał wszystkich modułów programu — przedstawię tylko te, których zadaniem jest wyświetlenie menu i zareagowanie na wybór pozycji. Na rysunku 11.16 pokażę, jak zaprojektować moduł `main`. Najpierw wywołuję moduł `displayMainMenu`. Zadaniem tego modułu jest wyświetlenie menu głównego i pobranie od użytkownika wybranej pozycji. Następnie w strukturze decyzyjnej wywoływane są następujące moduły:

- `saleOrReturn` — jeśli użytkownik wybierze pozycję 1.
- `updateInventory` — jeśli użytkownik wybierze pozycję 2.
- `inventoryReport` — jeśli użytkownik wybierze pozycję 3.

Jeśli użytkownik wybierze pozycję 4., program zakończy działanie.

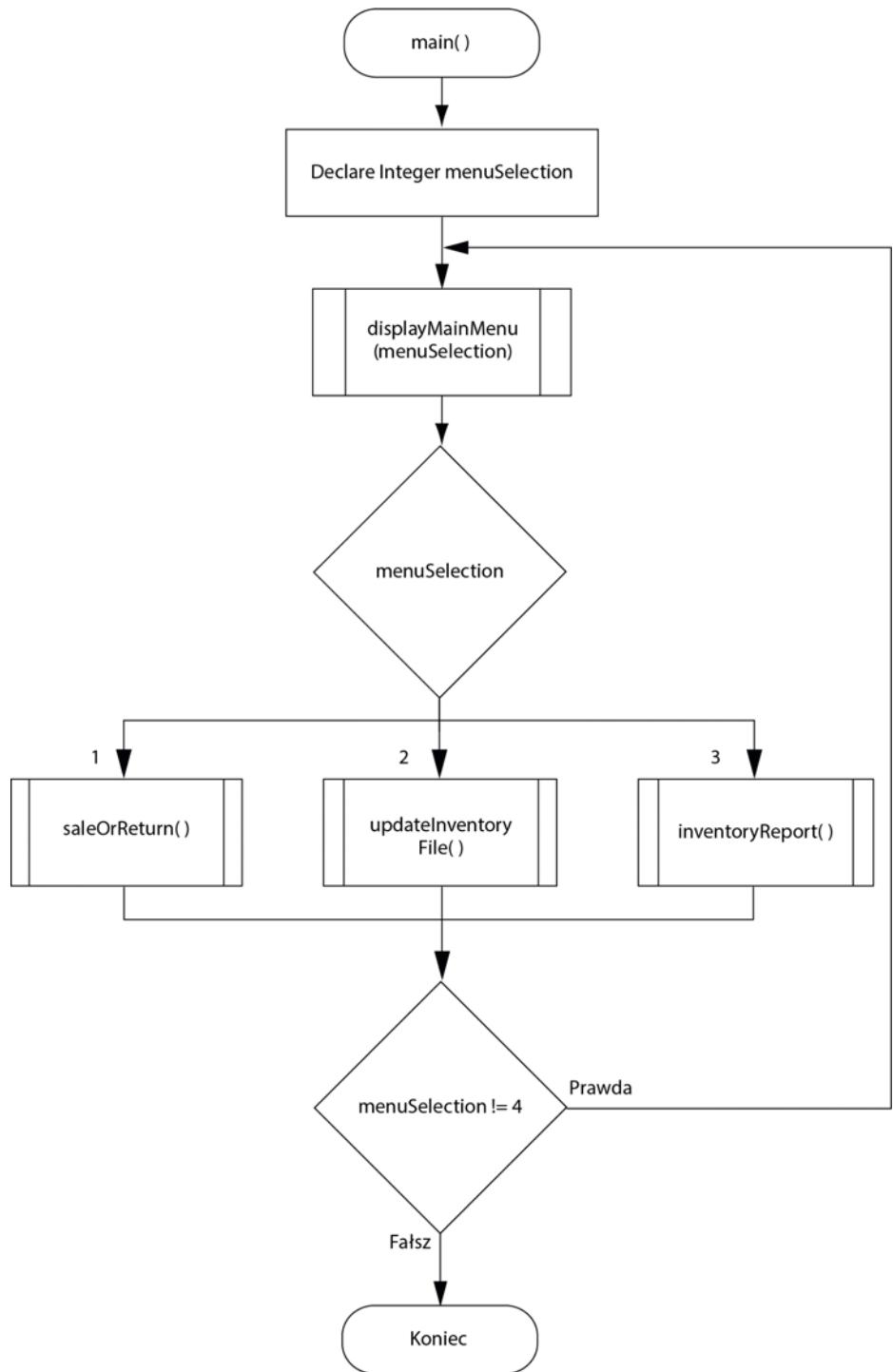
Na rysunku 11.17 przedstawiłem sposób działania modułu `saleOrReturn`. Najpierw wywołuję moduł `displaySaleOrReturnMenu`. Jego zadaniem jest wyświetlenie menu analizy sprzedaży i zyskowności i pobranie od użytkownika wybranej pozycji. Następnie w strukturze decyzyjnej wywoływane są następujące moduły:

- `processSale` — jeśli użytkownik wybierze pozycję 1.
- `processReturn` — jeśli użytkownik wybierze pozycję 2.

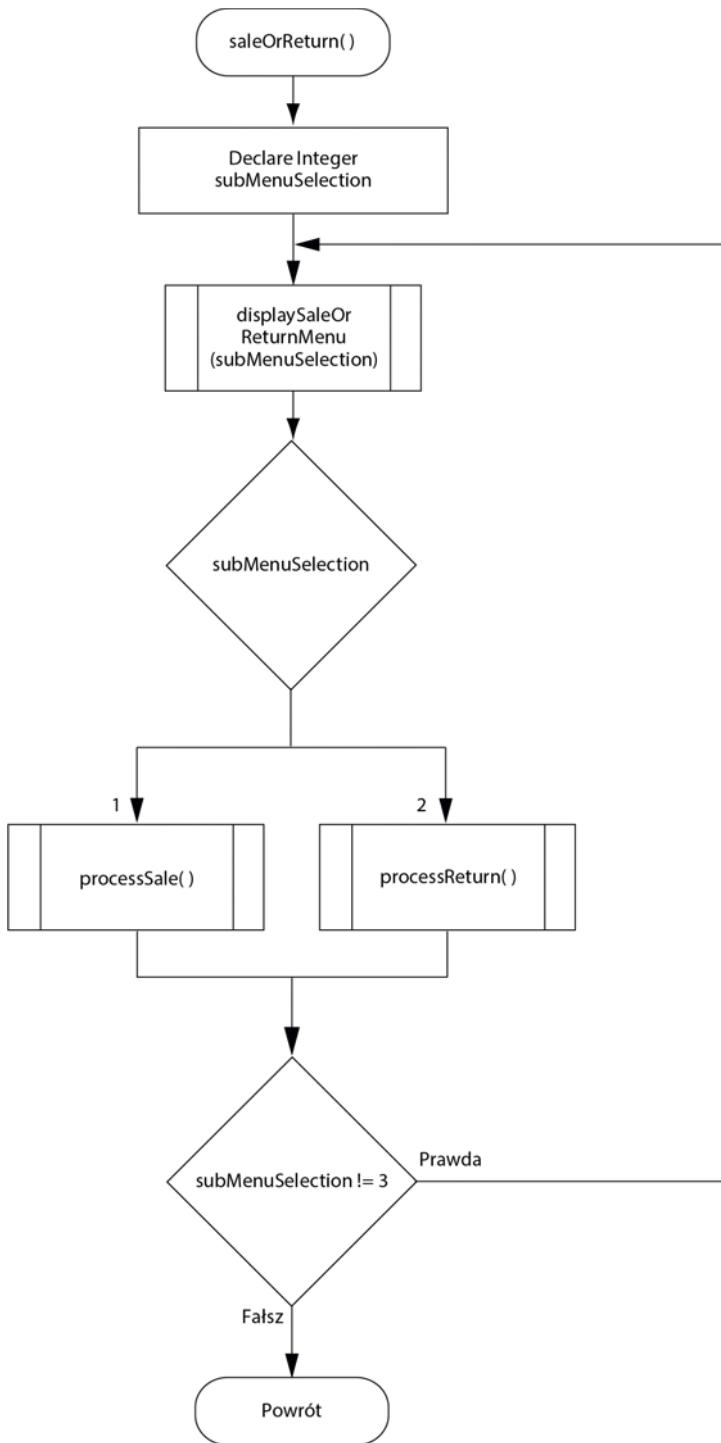
Jeśli użytkownik wybierze pozycję 3. *Powrót do menu głównego*, program powróci do modułu `main` i ponownie wyświetli menu główne.

Na rysunku 11.18 przedstawiłem sposób działania modułu `updateInventory`. Najpierw wywołuję moduł `displayUpdateInventoryMenu`. Jego zadaniem jest wyświetlenie menu aktualizowania asortymentu i pobranie od użytkownika wybranej pozycji. Następnie w strukturze decyzyjnej wywoływane są następujące moduły:

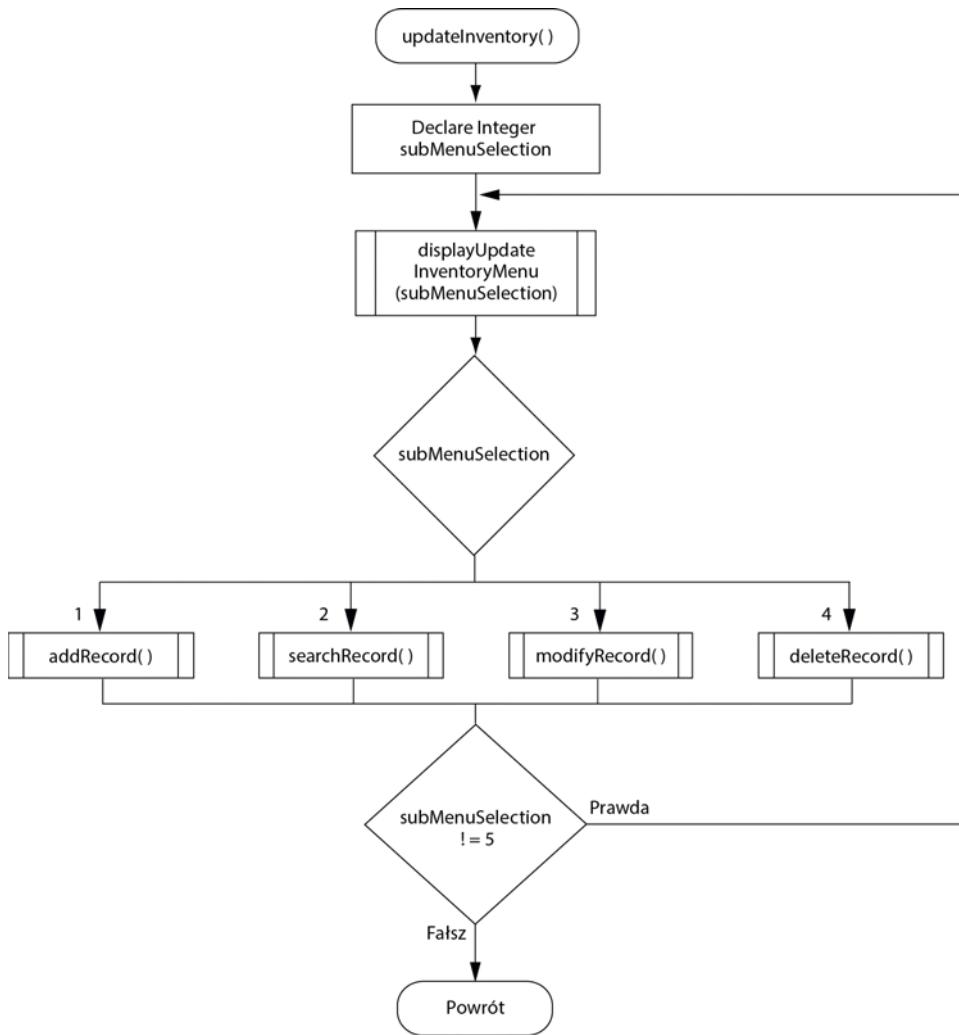
- `addRecord` — jeśli użytkownik wybierze pozycję 1.
- `searchRecord` — jeśli użytkownik wybierze pozycję 2.
- `modifyRecord` — jeśli użytkownik wybierze pozycję 3.
- `deleteRecord` — jeśli użytkownik wybierze pozycję 4.



**Rysunek 11.16.** Zasada działania modułu main



Rysunek 11.17. Zasada działania modułu `saleOrReturn`



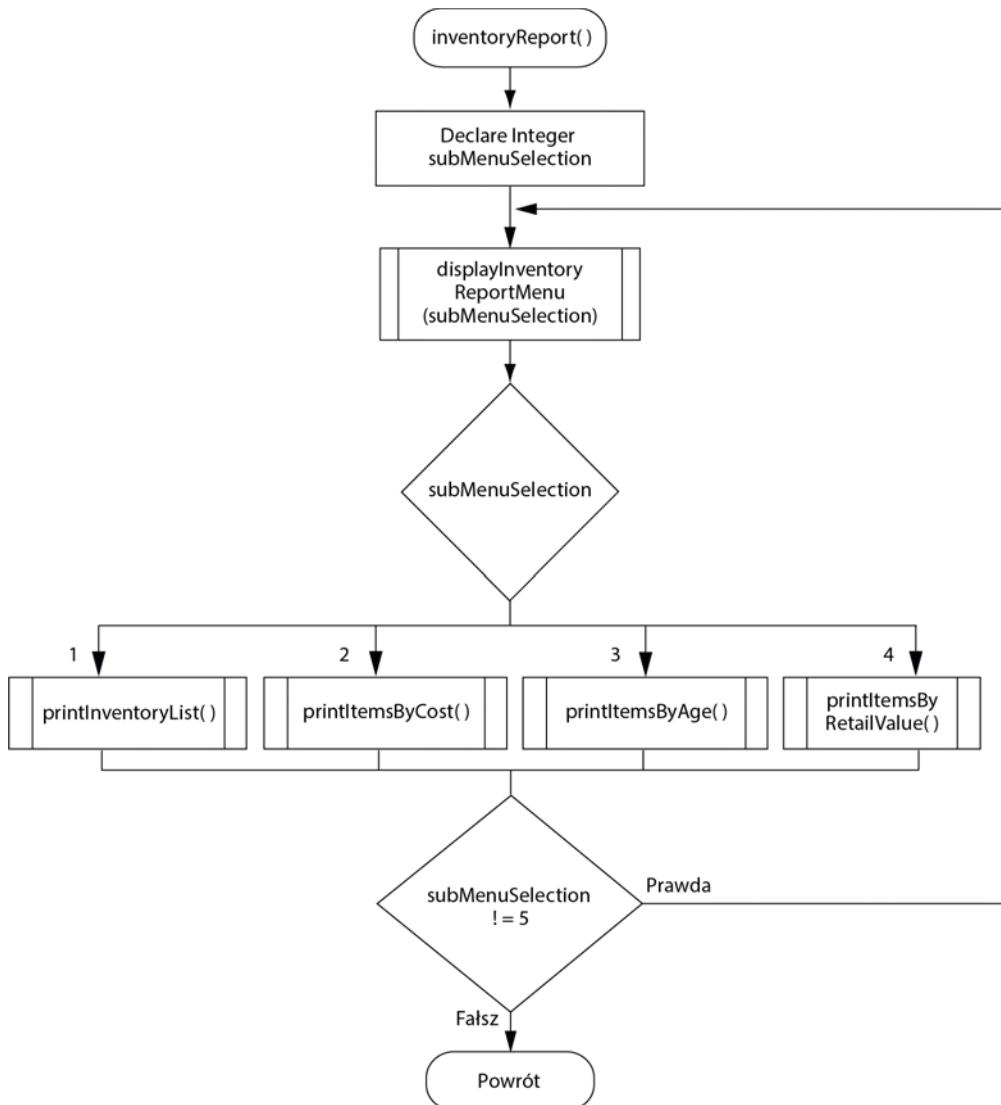
**Rysunek 11.18.** Zasada działania modułu `updateInventory`

Jeśli użytkownik wybierze pozycję 5. *Powrót do menu głównego*, program powróci do modułu `main` i ponownie wyświetli menu główne.

Na rysunku 11.19 przedstawiłem sposób działania modułu `inventoryReport`. Najpierw wywołuję moduł `displayInventoryReportMenu`. Jego zadaniem jest wyświetlenie menu drukowania raportów i pobranie od użytkownika wybranej pozycji. Następnie w strukturze decyzyjnej wywoływane są następujące moduły:

- `printInventoryList` — jeśli użytkownik wybierze pozycję 1.
- `printItemsByCost` — jeśli użytkownik wybierze pozycję 2.
- `printItemsByAge` — jeśli użytkownik wybierze pozycję 3.
- `printItemsByRetailValue` — jeśli użytkownik wybierze pozycję 4.

Jeśli użytkownik wybierze pozycję 5. *Powrót do menu głównego*, program powróci do modułu `main` i ponownie wyświetli menu główne.



Rysunek 11.19. Zasada działania modułu `inventoryReport`



## Punkt kontrolny

- 11.6. Co to jest menu jednopoziomowe?
- 11.7. Co to jest menu wielopoziomowe?
- 11.8. Dlaczego w przypadku programu charakteryzującego się dużą liczbą możliwych do wykonania operacji powinno się unikać wyświetlania wszystkich elementów w pojedynczym menu?

**11.5****Rzut oka na języki Java, Python i C++**

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/troya.zip>.

**Java**

W tym rozdziale nie omawiam już nowych funkcji języka programowania, pokażę tylko prosty program w Javie sterowany przez menu. Program przedstawiony na listingu 11.7 to napisana w Javie wersja programu w pseudokodzie z listingu 11.3.

**Listing 11.7 (MenuDriven.java)**

```

1 import java.util.Scanner;
2
3 public class MenuDriven
4 {
5     public static void main(String[] args)
6     {
7         // Deklarujemy zmienne, w której zapiszemy
8         // pozycję menu wybraną przez użytkownika
9         int menuSelection;
10
11        // Deklarujemy zmienne,
12        // w których zapiszemy jednostki
13        double inches, centimeters, feet, meters,
14            miles, kilometers;
15
16        // Tworzymy obiekt do wprowadzania danych z klawiatury
17        Scanner keyboard = new Scanner(System.in);
18
19        // Wyświetlamy menu
20        System.out.println("1. Zamiana cali na centymetry.");
21        System.out.println("2. Zamiana stóp na metry.");
22        System.out.println("3. Zamiana mil na kilometry.");
23        System.out.println();
24
25        // Prosimy użytkownika o wybranie operacji
26        System.out.print("Wybierz operację.");
27        menuSelection = keyboard.nextInt();
28
29        // Walidujemy dane wprowadzone przez użytkownika
30        while (menuSelection < 1 || menuSelection > 3)
31        {
32            System.out.println("Nieprawidłowa operacja.");
33            System.out.print("Wprowadź 1, 2 lub 3.");
34            menuSelection = keyboard.nextInt();
35        }
36

```

```

37     // Wykonujemy wybraną operację
38     switch(menuSelection)
39     {
40         case 1:
41             // Zamieniamy cali na centymetry
42             System.out.print("Wprowadź liczbę cali. ");
43             inches = keyboard.nextDouble();
44             centimeters = inches * 2.54;
45             System.out.println("To " + centimeters +
46                               " cm.");
47             break;
48
49         case 2:
50             // Zamieniamy stopy na metry
51             System.out.print("Wprowadź liczbę stóp.");
52             feet = keyboard.nextDouble();
53             meters = feet * 0.3048;
54             System.out.println("To " + meters +
55                               " m.");
56             break;
57
58         case 3:
59             // Zamieniamy mile na kilometry
60             System.out.print("Wprowadź liczbę mil.");
61             miles = keyboard.nextDouble();
62             kilometers = miles * 1.609;
63             System.out.println("To " + kilometers +
64                               " km.");
65             break;
66     }
67 }
68 }
```

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację. 1 [Enter]  
 Wprowadź liczbę cali. 10 [Enter]  
 To 25,4 cm.

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację. 2 [Enter]  
 Wprowadź liczbę stóp. 10 [Enter]  
 To 3,048 m.

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację. 4 [Enter]  
 Nieprawidłowa operacja.  
 Wprowadź 1, 2 lub 3. 3 [Enter]  
 Wprowadź liczbę mil. 10 [Enter]  
 To 16,09 km.

## Python

W tym rozdziale nie omawiam już nowych funkcji języka programowania, pokażę tylko prosty program w Pythonie sterowany przez menu. Program przedstawiony na listingu 11.8 to napisana w Pythonie wersja programu w pseudokodzie z listingu 11.3.

### **Listing 11.8 (menu\_driven.py)**

```

1 # Wyświetlamy menu
2 print('1. Zamiana cali na centymetry.')
3 print('2. Zamiana stóp na metry.')
4 print('3. Zamiana mil na kilometry.')
5 print()
6
7 # Prosimy użytkownika o wybranie operacji
8 menu_selection = int(input('Wybierz operację.'))
9
10 # Walidujemy dane wprowadzone przez użytkownika
11 while menu_selection < 1 or menu_selection > 3
12     print('Nieprawidłowa operacja.')
13     menu_selection = int(input('Wprowadź 1, 2 lub 3. '))
14
15 # Wykonujemy wybraną operację
16 if menu_selection == 1:
17     # Zamieniamy cale na centymetry
18     inches = float(input('Wprowadź liczbę cali. '))
19     centimeters = inches * 2.54
20     print('To', centimeters, 'cm.')
21 elif menu_selection == 2:
22     # Zamieniamy stopy na metry
23     feet = float(input('Wprowadź liczbę stóp. '))
24     meters = feet * 0.3048
25     print('To', meters, 'm.')
26 elif menu_selection == 3:
27     # Zamieniamy mile na kilometry
28     miles = float(input('Wprowadź liczbę mil. '))
29     kilometers = miles * 1.609
30     print('To', kilometers, 'km.')

```

### **Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację. 1 [Enter]

Wprowadź liczbę cali. 10 [Enter]

To 25,4 cm.

### **Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację. 2 [Enter]

Wprowadź liczbę stóp. 10 [Enter]

To 3,048 m.

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację. 4 [Enter]

Nieprawidłowa operacja.

Wprowadź 1, 2 lub 3. 3 [Enter]

Wprowadź liczbę mil. 10 [Enter]

To 16,09 km.

**C++**

W tym rozdziale nie omawiam już nowych funkcji języka programowania, pokażę tylko prosty program w C++ sterowany przez menu. Program przedstawiony na listingu 11.9 to napisana w C++ wersja programu w pseudokodzie z listingu 11.3.

**Listing 11.9 (MenuDriven.cpp)**

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     // Deklarujemy zmenną, w której zapiszemy
7     // pozycję menu wybraną przez użytkownika
8     int menuSelection;
9
10    // Deklarujemy zmienne,
11    // w których zapiszmy jednostki
12    double inches, centimeters, feet, meters,
13        miles, kilometers;
14
15    // Wyświetlamy menu
16    cout << "1. Zamiana cali na centymetry." << endl;
17    cout << "2. Zamiana stóp na metry." << endl;
18    cout << "3. Zamiana mil na kilometry." << endl;
19    cout << endl;
20
21    // Prosimy użytkownika o wybranie operacji
22    cout << "Wybierz operację." << endl;
23    cin >> menuSelection;
24
25    // Walidujemy dane wprowadzone przez użytkownika
26    while (menuSelection < 1 || menuSelection > 3)
27    {
28        cout << "Nieprawidłowa operacja." << endl;
29        cout << "Wprowadź 1, 2 lub 3." << endl;
30        cin >> menuSelection;
31    }
32
33    // Wykonujemy wybraną operację
34    switch(menuSelection)

```

```

35     {
36     case 1:
37         // Zamieniamy cali na centymetry
38         cout << "Wprowadź liczbę cali." << endl;
39         cin >> inches;
40         centimeters = inches * 2.54;
41         cout << "To " << centimeters
42             << " cm." << endl;
43         break;
44
45     case 2:
46         // Zamieniamy stopy na metry
47         cout << "Wprowadź liczbę stóp." << endl
48         cin >> feet;
49         meters = feet * 0.3048;
50         cout << "To " << meters
51             << " m." << endl;
52         break;
53
54     case 3:
55         // Zamieniamy mile na kilometry
56         cout << "Wprowadź liczbę mil." << endl;
57         cin >> miles;
58         kilometers = miles * 1.609;
59         cout << "To " << kilometers
60             << " km." << endl;
61         break;
62     }
63     return 0;
64 }
```

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**1 [Enter]**

Wprowadź liczbę cali.

**10 [Enter]**

To 25,4 cm.

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**2 [Enter]**

Wprowadź liczbę stóp.

**10 [Enter]**

To 3,048 m.

**Wynik działania programu**

1. Zamiana cali na centymetry.
2. Zamiana stóp na metry.
3. Zamiana mil na kilometry.

Wybierz operację.

**4 [Enter]**

Nieprawidłowa operacja.

Wprowadź 1, 2 lub 3:

**3 [Enter]**

Wprowadź liczbę mil.

**10 [Enter]**

To 16,09 km.

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Menu to \_\_\_\_\_.
  - a) struktura decyzyjna, za pomocą której można wybrać określoną operację
  - b) zbiór modułów wykonujących poszczególne zadania
  - c) wyświetlana na ekranie lista operacji, które może wybrać użytkownik
  - d) tabela operacji boolowskich
2. Kiedy użytkownik wybierze pozycję z menu, program musi wykonać wybraną operację za pomocą struktury \_\_\_\_\_.
  - e) cyklicznej
  - f) sekwencyjnej
  - g) wyboru menu
  - h) warunkowej
3. W programie, w którym do ponownego wyświetlania menu po wykonaniu operacji wybranej przez użytkownika wykorzystuje się pętlę, w menu powinna się także znajdować pozycja umożliwiająca \_\_\_\_\_.
  - a) zakończenie programu
  - b) ponowne wykonanie tej samej operacji
  - c) wycofanie poprzedniej operacji
  - d) zrestartowanie komputera
4. Program wyposażony w menu wielopoziomowe wyświetla na początku \_\_\_\_\_.
  - a) ostrzeżenie
  - b) menu główne
  - c) podmenu
  - d) menu nadzędne
5. Kiedy użytkownik wybierze pozycję w programie wyposażonym w menu wielopoziomowe, na ekranie może wyświetlić się \_\_\_\_\_.
  - a) menu główne
  - b) formularz zawierający dane użytkownika
  - c) podmenu
  - d) komunikat pytający użytkownika, czy chce kontynuować pracę
6. Kiedy użytkownik wybierze operację w programie wyposażonym w \_\_\_\_\_, program wykona wybraną operację i ponownie wyświetli menu (lub zakończy działanie, jeśli nie użyjemy pętli).
  - a) menu wielopoziomowe
  - b) menu jednopoziomowe

- c) podmenu  
d) menu nadzędne
7. Kiedy użytkownik wybierze operację w programie wyposażonym w \_\_\_\_\_, na ekranie może wyświetlić się kolejne menu.  
a) menu wielopoziomowe  
b) menu jednopoziomowe  
c) podmenu  
d) menu z przeplotem

### **Prawda czy fałsz?**

1. Aby wykonać operację wybraną przez użytkownika z menu, nie można skorzystać z zagnieżdzonych poleceń If-Then-Else.
2. Walidacja wybranej przez użytkownika pozycji w menu zazwyczaj nie jest konieczna.
3. W większości przypadków program sterowany za pomocą menu należy podzielić na moduły.
4. Jeśli w programie sterowanym za pomocą menu nie skorzystamy z pętli, która będzie ponownie wyświetlała menu po wykonaniu wybranej operacji, użytkownik będzie musiał ponownie uruchomić program, aby wykonać kolejną operację.
5. W przypadku menu jednopoziomowego użytkownik po wybraniu pozycji w menu głównym zobaczy na ekranie podmenu.

### **Krótką odpowiedź**

1. Jakiej struktury użyjesz w programie, aby wykonać operację wybraną przez użytkownika z menu?
2. Jakie sposoby walidacji pozycji wybranej przez użytkownika w menu omówiłem w tym rozdziale?
3. Jakiego mechanizmu użyjesz podczas projektowania programu sterowanego za pomocą menu, jeśli będziesz chciał ponownie wyświetlić menu po wykonaniu operacji wybranej przez użytkownika?
4. Czym różni się program z menu jednopoziomowym od programu z menu wielopoziomowym?
5. Dlaczego w przypadku programu charakteryzującego się dużą liczbą możliwych do wykonania operacji powinno się unikać wyświetlania wszystkich pozycji w jednym menu?

### **Warsztat projektanta algorytmów**

1. Zaprojektuj algorytm programu, który będzie wyświetlał poniższe menu, pobierał od użytkownika operację, a następnie walidował wprowadzone dane.

Menu główne  
 1. Otwórz dokument.  
 2. Zamknij bieżący dokument.  
 3. Wydrukuj bieżący dokument.  
 4. Zakończ program.  
 Wybierz operację.

2. Zaprojektuj strukturę decyzyjną, z której będziesz mógł skorzystać w programie z punktu 1. Po wybraniu przez użytkownika pozycji 1. struktura decyzyjna powinna wywołać moduł `openDocument`. Po wybraniu przez użytkownika pozycji 2. powinna wywołać moduł `closeDocument`, a po wybraniu pozycji 3. — moduł `printDocument`.
3. Połącz algorytmy, które zaprojektowałeś w punktach 1. i 2., i umieść je w pętli, która ponownie wyświetli menu, gdy program zakończy wykonywanie operacji wybranej przez użytkownika, lub zakończy program, gdy użytkownik wybierze w menu operację 4.
4. Pomyśl, jak można zmodularyzować algorytm zaprojektowany w punkcie 3., i odpowiednio go zmodyfikuj.

## Ćwiczenia programistyczne

### 1. Tłumacz

Zaprojektuj program, który będzie wyświetlał następujące menu:

*Wybierz język, a ja Cię w nim przywitam.*

1. Angielski
2. Włoski
3. Hiszpański
4. Niemiecki
5. Zakończ program

*Wybierz pozycję menu.*

Jeśli użytkownik wybierze pozycję 1., program powinien wyświetlić tekst *Good morning*. Jeśli użytkownik wybierze pozycję 2., program powinien wyświetlić tekst *Buongiorno*. Jeśli użytkownik wybierze pozycję 3., program powinien wyświetlić tekst *Buenos días*. Jeśli użytkownik wybierze pozycję 4., program powinien wyświetlić tekst *Guten Morgen*. Jeśli użytkownik wybierze pozycję 5., program powinien zakończyć działanie.

### 2. Wybór uniwersyteckiego planu posiłków

Stółówka uniwersytecka umożliwia wykupienie następujących planów posiłków:

Plan 1: 7 posiłków w tygodniu za 560 zł na semestr.

Plan 2: 14 posiłków w tygodniu za 1095 zł na semestr.

Plan 3: Nieograniczona liczba posiłków za 1500 zł na semestr.

Zaprojektuj program sterowany za pomocą menu, dzięki któremu użytkownik będzie mógł wybrać odpowiadający mu plan posiłków. Program powinien poprosić użytkownika o wprowadzenie liczby semestrów, a następnie wyświetlić całkowity koszt wybranego planu.

### 3. Kalkulator geometryczny

Napisz program, który będzie wyświetlał następujące menu:

*Kalkulator geometryczny*

1. Obliczanie pola powierzchni okręgu
2. Obliczanie pola powierzchni prostokąta

3. Obliczanie pola powierzchni trójkąta
  4. Wyjście
- Wybierz operację (1-4).

Jeśli użytkownik wybierze pozycję 1., program powinien poprosić go o wprowadzenie promienia okręgu, a następnie wyświetlić jego pole. Aby obliczyć pole powierzchni okręgu, posłuż się następującym wzorem:

$$\text{pole powierzchni} = \pi r^2$$

$\pi$  jest równe 3,14159, a  $r$  to promień okręgu.

Jeśli użytkownik wybierze pozycję 2., program powinien poprosić go o wprowadzenie długości boków prostokąta, a następnie wyświetlić jego pole. Aby obliczyć pole powierzchni prostokąta, posłuż się następującym wzorem:

$$\text{pole powierzchni} = \text{długość} \cdot \text{szerokość}$$

Jeśli użytkownik wybierze pozycję 3., program powinien poprosić go o wprowadzenie długości podstawy i wysokości trójkąta, a następnie wyświetlić jego pole. Aby obliczyć pole powierzchni trójkąta, posłuż się następującym wzorem:

$$\text{pole powierzchni} = \text{podstawa} \cdot \text{wysokość} \cdot 0,5$$

Jeśli użytkownik wybierze pozycję 4., program powinien zakończyć działanie.

#### 4. Pomocnik astronomiczny

Stwórz aplikację, która będzie wyświetlała następujące menu:

- Wybierz planetę
1. Merkury
  2. Wenus
  3. Ziemia
  4. Mars
  5. Zakończ program
- Wybierz pozycję

Kiedy użytkownik wybierze którąś z planet, program powinien wyświetlić następujące informacje na jej temat: średnia odległość od Słońca, masa planety, temperatura na jej powierzchni. W programie wykorzystaj poniższe dane.

##### Merkury

Średnia odległość od Słońca	57,9 miliona km
Masa	$3,31 \cdot 10^{23}$ kg
Temperatura na powierzchni	od -173 do 430°C

##### Wenus

Średnia odległość od Słońca	108,2 miliona km
Masa	$4,87 \cdot 10^{24}$ kg
Temperatura na powierzchni	472°C

##### Ziemia

Średnia odległość od Słońca	149,6 miliona km
Masa	$5,967 \cdot 10^{24}$ kg
Temperatura na powierzchni	od -50 do 50°C

**Mars**

Średnia odległość od Słońca	227,9 miliona km
Masa	$0,6424 \cdot 10^{24}$ kg
Temperatura na powierzchni	od -140 do 20°C

**5. Modyfikacja programu wyświetlającego wyniki gry w golfa**

W ćwiczeniu programistycznym 6. z rozdziału 10. zaprojektowałeś następujące dwa programy dla klubu golfowego Springfolk Amateur:

1. Program, który umożliwia wprowadzenie nazwiska i wyniku zawodnika, a następnie zapisuje te dane jako rekordy w pliku *golf.dat*.
2. Program, który odczytuje rekordy zapisane w pliku *golf.dat*, a następnie je wyświetla.

Połącz te dwa programy w jeden program, który będzie wyświetlał menu i pozwalał użytkownikowi wybrać operację.

**6. Książka telefoniczna**

Zaprojektuj program, dzięki któremu będziesz mógł zapisać w pliku nazwiska i numery telefonów do swoich przyjaciół. Program powinien być sterowany za pomocą menu i udostępniać następujące operacje:

1. Dodaj nowy rekord
2. Wyszukaj numer telefonu osoby
3. Zmień numer telefonu
4. Usuń rekord
5. Zakończ program

**7. Prędkość dźwięku**

W poniższej tabeli zamieścilem wartości prędkości dźwięku rozchodzącego się w powietrzu, wodzie i stali.

<b><u>Medium</u></b>	<b><u>Prędkość dźwięku</u></b>
Powietrze	300 m/s
Woda	1500 m/s
Stal	6000 m/s

Zaprojektuj program, który będzie wyświetlał menu umożliwiające użytkownikowi wybór powietrza, wody lub stali. Gdy użytkownik wybierze jedną z opcji, program powinien poprosić go o wprowadzenie czasu, przez jaki dźwięk rozchodził się w danym medium, a następnie wyświetlić informację, jaką odległość dźwięk w tym czasie przebył.



**TEMATYKA**

12.1 Wstęp

12.3 Rzut oka na języki Java, Python i C++

12.2 Przetwarzanie poszczególnych znaków  
w ciągu**12.1****Wstęp**

W pewnych przypadkach projektowany przez Ciebie programie będzie musiał przetwarzać dane tekstowe. Przykładami tego typu aplikacji są procesory tekstu, programy wysyłające wiadomości tekstowe, programy pocztowe, przeglądarki internetowe, korektory tekstu.

W poprzednich rozdziałach pokazałem Ci kilka technik przetwarzania tekstu, takie jak porównywanie ciągów znaków (uwzględniające lub ignorujące wielkość znaków), sortowanie ciągów zawartych w tablicy lub wyszukiwanie fragmentu tekstu w ciągu znaków. Ponadto w rozdziale 6. przedstawiłem kilka funkcji bibliotecznych, które wykonują określone operacje na ciągach znaków. Dla wygody zaprezentowałem te funkcje ponownie w tabeli 12.1.

Funkcje przedstawione w tabeli 12.1 są bardzo pomocne, ale w wielu przypadkach trzeba operować na ciągach w bardziej szczegółowy sposób. Niekiedy musimy pobić się po poszczególne znaki w ciągu znaków i nimi manipulować. Miałeś zapewne styczność z programami lub serwisami internetowymi, które wymagają utworzenia hasła spełniającego określone reguły, takie jak określona minimalna długość, co najmniej jeden mały znak, jeden duży znak i jedna cyfrę. Reguły te mają na celu zapobieganie sytuacji, gdy użytkownik tworzy bardzo łatwe do złamania hasło składające się z powszechnie występującego słowa. Podczas tworzenia hasła system musi więc w jakiś sposób zweryfikować każdy jego znak i określić, czy dane hasło jest zgodne z regułami. W kolejnym podrozdziale przedstawię przykład algorytmu, który wykonuje taką operację. Wcześniej jednak pokażę, jak można sprawdzać i modyfikować poszczególne znaki w ciągu.

**Tabela 12.1.** Popularne funkcje na ciągach znaków

Funkcja	Opis
<code>length(<i>ciąg znaków</i>)</code>	Zwraca liczbę znaków w <i>ciągu znaków</i> . Przykładowo wyrażenie <code>length("Test")</code> zwróci wartość równą 4.
<code>append(<i>ciąg znaków 1</i>, <i>ciąg znaków 2</i>)</code>	Zwraca串 znaków powstały w wyniku dołączenia na końcu <i>ciągu znaków 1</i> , <i>ciągu znaków 2</i> .
<code>toUpper(<i>ciąg znaków</i>)</code>	Przykładowo wyrażenie <code>append("Witaj, " , "świecie!")</code> zwróci串 znaków "Witaj, świecie!".
<code>toLower(<i>ciąg znaków</i>)</code>	Zwraca kopię <i>ciągu znaków</i> , w której wszystkie znaki zostały zamienione na duże. Przykładowo wyrażenie <code>toUpper("Test")</code> zwróci串 znaków "TEST".
<code>substring(<i>ciąg znaków</i>, <i>początek</i>, <i>koniec</i>)</code>	Zwraca podciąg <i>ciągu znaków</i> , począwszy od znaku o indeksie <i>początek</i> , a kończąc na znaku o indeksie <i>koniec</i> . Pierwszy znak ma indeks 0. Przykładowo wyrażenie <code>substring("Karol", 2, 4)</code> zwróci串 znaków "rol".
<code>contains(<i>ciąg znaków</i>)</code>	Zwraca wartość <code>True</code> , jeśli <i>ciąg znaków 1</i> zawiera <i>ciąg znaków 2</i> , lub <code>False</code> w przeciwnym przypadku. Przykładowo wyrażenie <code>contains("Program", "gram")</code> zwróci wartość <code>True</code> , a wyrażenie <code>contains("Program", "xyz")</code> zwróci wartość <code>False</code> .
<code>stringToInteger(<i>ciąg znaków</i>)</code>	Zamienia <i>ciąg znaków</i> na liczbę całkowitą <code>Integer</code> i ją zwraca. Przykładowo wyrażenie <code>stringToInteger("77")</code> zwróci liczbę całkowitą 77.
<code>stringToReal(<i>ciąg znaków</i>)</code>	Zamienia <i>ciąg znaków</i> na liczbę rzeczywistą <code>Real</code> i ją zwraca. Przykładowo wyrażenie <code>stringToReal("1.5")</code> zwróci liczbę rzeczywistą 1.5.
<code>isInteger(<i>ciąg znaków</i>)</code>	Zwraca wartość <code>True</code> , jeśli <i>ciąg znaków</i> da się skonwertować na liczbę całkowitą, lub <code>False</code> w przeciwnym przypadku. Przykładowo wyrażenie <code>isInteger("77")</code> zwróci wartość <code>True</code> , a wyrażenie <code>isInteger("x4yz")</code> zwróci wartość <code>False</code> .
<code>isReal(<i>ciąg znaków</i>)</code>	Zwraca wartość <code>True</code> , jeśli <i>ciąg znaków</i> da się skonwertować na liczbę rzeczywistą, lub <code>False</code> w przeciwnym przypadku. Przykładowo wyrażenie <code>isReal("3.2")</code> zwróci wartość <code>True</code> , a wyrażenie <code>isReal("x4yz")</code> zwróci wartość <code>False</code> .

**12.2**

## Przetwarzanie poszczególnych znaków w ciągu

**WYJAŚNIENIE:** W niektórych zadaniach programistycznych trzeba przetwarzać poszczególne znaki, z których składa się ciąg.

W każdym języku programowania odwoływanie się do poszczególnych znaków ciągu może odbywać się nieco inaczej. W wielu językach można to robić, posługując się indeksami. Dzięki temu praca z ciągiem znaków wygląda identycznie jak praca z tablicą zawierającą znaki. Do pierwszego znaku w ciągu odwołujemy się poprzez indeks 0, do drugiego poprzez indeks 1 itd. Indeks ostatniego znaku w ciągu jest o 1 mniejszy od jego długości. Właśnie takie podejście zademonstrowałem na przykładowym pseudokodzie na listingu 12.1.

**Listing 12.1**

```

1 // Deklarujemy i inicjalizujemy ciąg znaków
2 Declare String name = "Jacek"
3
4 // Posługując się indeksami,
5 // wyświetlamy poszczególne litery, z których składa się ciąg znaków
6 Display name[0]
7 Display name[1]
8 Display name[2]
9 Display name[3]
10 Display name[4]
```

**Wynik działania programu**

```
J
a
c
e
k
```

W linii 2. deklaruję zmienną typu `String` o nazwie `name` i inicjalizuję ją ciągiem znaków "Jacek". Ciąg ten składa się z pięciu znaków, więc odwołuję się do nich za pomocą indeksów od 0 do 4 — co widać w liniach od 6. do 10. Gdy spróbujesz odwołać się do nieistniejącego indeksu, podobnie jak w przypadku tablic, pojawi się błąd.

Na listingu 12.2 pokazalem, że do kolejnych znaków w ciągu można się także odwoływać za pomocą pętli. Zauważ, że zmienna licznikowa `index` w pętli `For` (w linii 8.) przyjmuje wartości od 0 do `length(name)-1`.

**Listing 12.2**

```

1 // Deklarujemy i inicjalizujemy ciąg znaków
2 Declare String name = "Jacek"
3
4 // Deklarujemy zmienną licznikową
5 Declare Integer index
```

```

6
7 // Wyświetlamy litery, z których składa się ciąg znaków
8 For index = 0 To length(name) - 1
9   Display name[index]
10 End For

```

**Wynik działania programu**

J  
a  
c  
e  
k

Na listingu 12.3 przedstawiłem przykładowy program, za pomocą którego modyfikuję poszczególne znaki w ciągu. Program odczytuje wprowadzony na klawiaturze ciąg znaków, a następnie zamienia każdą występującą w ciągu literę „t” na literę „d”.

**Listing 12.3**

```

1 // Deklarujemy ciąg znaków, w którym zapiszemy dane wejściowe
2 Declare String str
3
4 // Deklarujemy zmienną, za pomocą której będziemy śledzić poszczególne znaki w ciągu
5 Declare Integer index
6
7 // Prosimy użytkownika o wprowadzenie dowolnego zdania
8 Display "Wprowadź dowolne zdanie."
9 Input str
10
11 // Zamieniamy litery t na litery d
12 For index = 0 To length(str) - 1
13   If str[index] == "t" Then
14     Set str[index] = "d"
15   End If
16 End For
17
18 // Wyświetlamy zmodyfikowany ciąg znaków
19 Display str

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź dowolne zdanie.

**Spójrz, jaki milutki kotek! [Enter]**

Spójrz, jaki miludki kodek!

Programy, które do tej pory przedstawiłem w tym rozdziale, demonstrowały, jak można odwoływać się do znaków znajdujących się w określonym miejscu w ciągu i je zmieniać. W większości języków programowania do poszczególnych znaków ciągu można się odwoływać za pomocą indeksów lub innego mechanizmu. W takim przypadku wskazany indeks musi być prawidłowy — w przeciwnym razie w programie wystąpi błąd. Przykładowo w ciągu składającym się z czterech znaków nie możemy użyć poleceń, które dołączy do niego piąty znak. Ilustruje to poniższy pseudokod:

```

Declare String word = "plik"    // Ten ciąg składa się z 4 znaków
Set word[4] = "i"                // Błąd!

```

W przypadku pierwszego polecenia zadeklarowałem zmienią typu `String` o nazwie `word` i zainicjalizowałem ją ciągiem znaków "plik" (składającym się z czterech znaków). Indeks ostatniego znaku jest więc równy 3. W drugim poleceniu próbuję do elementu `word[4]` przypisać znak "i", ale w tym momencie wystąpi błąd, ponieważ znaku o takim indeksie nie ma w tym ciągu. Jeśli chcesz dołączyć do ciągu znaków określony znak, najczęściej będziesz musiał się posłużyć operatorem lub przeznaczoną do tego funkcją biblioteczną.



**OSTRZEŻENIE!** Błąd pojawi się także wtedy, gdy odwołasz się za pomocą indeksu do niezainicjalizowanej zmiennej typu `String`. Ponieważ zmienność taka nie ma żadnej wartości, nie można jej też zmodyfikować ani odwołać się do niej.

## Funkcje sprawdzające znaki

Poza funkcjami bibliotecznymi do przetwarzania ciągów znaków przedstawionymi w tabeli 12.1 w większości języków programowania można się posługiwać także funkcjami bibliotecznymi, które przetwarzają pojedyncze znaki. Najpopularniejsze spośród nich zaprezentowałem w tabeli 12.2. Zauważ, że każda z tych funkcji zwraca wartość boolowską równą `True` albo `False`.

**Tabela 12.2.** Popularne funkcje sprawdzające znaki

Funkcja	Opis
<code>isDigit(znak)</code>	Zwraca wartość <code>True</code> , jeśli <code>znak</code> jest cyfrą, lub <code>False</code> w przeciwnym przypadku.
<code>isLetter(znak)</code>	Zwraca wartość <code>True</code> , jeśli <code>znak</code> jest literą, lub <code>False</code> w przeciwnym przypadku.
<code>isLower(znak)</code>	Zwraca wartość <code>True</code> , jeśli <code>znak</code> jest małą literą, lub <code>False</code> w przeciwnym przypadku.
<code>isUpper(znak)</code>	Zwraca wartość <code>True</code> , jeśli <code>znak</code> jest dużą literą, lub <code>False</code> w przeciwnym przypadku.
<code>isWhiteSpace(znak)</code>	Zwraca wartość <code>True</code> , jeśli <code>znak</code> jest znakiem niedrukowalnym, lub <code>False</code> w przeciwnym przypadku. Znakami niedrukowalnymi są np. spacja, znak tabulacji czy znak końca linii.

Na listingu 12.4 przedstawiłem przykład wykorzystania jednej z tych funkcji. Program odczytuje wprowadzony za pomocą klawiatury ciąg znaków i zlicza liczbę występujących w nim dużych liter.

Pętla `For`, która pojawia się w liniach od 16. do 20., sprawdza po kolejny każdy znak w zmiennej `str`. W poleceniu `If-Then` rozpoczynającym się w linii 17. wywołuję funkcję `isUpper` i przekazuję do niej jako argument wyrażenie `str[index]`. Jeśli dany znak jest dużą literą, funkcja zwróci wartość `True`, a wartość zapisana w zmiennej `upperCaseCount`

**Listing 12.4**

```

1 // Deklarujemy zmienną typu String, w której zapiszemy wprowadzony ciąg znaków
2 Declare String str
3
4 // Deklarujemy zmienną, za pomocą której będziemy śledzić poszczególne znaki w ciągu
5 Declare Integer index
6
7 // Deklarujemy zmienną pełniącą funkcję akumulatora,
8 // za pomocą której będziemy zliczać duże znaki
9 Declare Integer upperCaseCount = 0
10
11 // Prosimy użytkownika o wprowadzenie dowolnego zdania
12 Display "Wprowadź dowolne zdanie."
13 Input str
14
15 // Zliczamy w zdaniu duże litery
16 For index = 0 To length(str) - 1
17     If isUpper(str[index]) Then
18         Set upperCaseCount = upperCaseCount + 1
19     End If
20 End For
21
22 // Wyświetlamy liczbę dużych liter
23 Display "Liczba dużych liter w tym zdaniu wynosi: ", upperCaseCount

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź dowolne zdanie.

**Pan Wróbel przejeżdża już DZISIAJ! [Enter]**

Liczba dużych liter w tym zdaniu wynosi: 9

zostanie zinkrementowana (linia 18.). Kiedy pętla zakończy działanie, w zmiennej upperCaseCount będzie zapisana wartość równa liczbie dużych liter występujących w ciągu znaków przypisanym do zmiennej str.

## W centrum uwagi

### Sprawdzanie hasła



W wielu systemach zabezpieczanych za pomocą hasła użytkownik ma możliwość jego zmiany. Aby zapewnić bezpieczeństwo, systemy takie najczęściej dbają, by hasło spełniało określone wymagania. Kiedy użytkownik tworzy nowe hasło, system sprawdza, czy spełnia ono te wymagania. Jeśli hasło nie spełnia wymagań, system je odrzuca i prosi użytkownika o wprowadzenie nowego, bezpieczniejszego hasła.

Na listingu 12.5 przedstawiłem pseudokod programu, który sprawdza, czy hasło spełnia następujące wymagania:

- hasło musi składać się z co najmniej 8 znaków;
- hasło musi zawierać co najmniej 1 dużą literę;
- hasło musi zawierać co najmniej 1 małą literę;
- hasło musi zawierać co najmniej 1 cyfrę.

Pseudokod podzieliłem na moduły, z których każdy zawiera funkcję sprawdzającą określone wymaganie dotyczące hasła. W module `main` pobieram od użytkownika hasło, a potem waliduję je, wywołując następujące funkcje:

- funkcja biblioteczna `length` sprawdza, jaka jest długość hasła;
- funkcja `numberUpperCase` zwraca liczbę dużych liter występujących w ciągu znaków przekazanym poprzez parametr `password`;
- funkcja `numberLowerCase` zwraca liczbę małych liter występujących w ciągu znaków przekazanym poprzez parametr `password`;
- funkcja `numberDigits` zwraca liczbę cyfr występujących w ciągu znaków przekazanym poprzez parametr `password`.

Teraz zaprezentuję po kolej wszytkie moduły programu, począwszy od modułu `main`.

#### **Listing 12.5. Program walidujący hasło: moduł main**

```

1 Module main()
2   // Stała równa minimalnej długości hasła
3   Constant Integer MIN_LENGTH = 8
4
5   // Zmienna lokalna, w której zapiszemy hasło wprowadzone przez użytkownika
6   Declare String password
7
8   // Wyświetlamy informacje na temat programu
9   Display "Program sprawdza, czy wprowadzone hasło"
10  Display "spełnia następujące wymagania:"
11  Display "(1) Hasło musi się składać z co najmniej 8 znaków."
12  Display "(2) Hasło musi zawierać co najmniej 1 dużą literę."
13  Display "(3) Hasło musi zawierać co najmniej 1 małą literę."
14  Display "(4) Hasło musi zawierać co najmniej 1 cyfrę."
15  Display
16
17  // Pobieramy hasło od użytkownika
18  Display "Wprowadź hasło."
19  Input password
20
21  // Walidujemy hasło
22  If length(password) >= MIN_LENGTH AND
23    numberUpperCase(password) >= 1 AND
24    numberLowerCase(password) >= 1 AND
25    numberDigits(password) >= 1 Then
26    Display "Hasło jest prawidłowe."
27  Else
28    Display "Hasło nie spełnia wymagań."
29  End If
30 End Module
31

```

W linii 3. deklaruję stałą, w której zapisuję minimalną długość hasła, a w linii 6. deklaruję zmienną typu `String` o nazwie `password`, w której zapiszę hasło wprowadzone przez użytkownika. W liniach od 9. do 15. wyświetlам na ekranie komunikat informujący użytkownika o wymaganiach dotyczących hasła. W liniach 18. i 19. proszę użytkownika o wprowadzenie hasła i zapisuję je w zmiennej `password`.

W poleceniu If-Then-Else rozpoczynającym się od linii 22. używam boolowskiego wyrażenia złożonego. Można je rozumieć w następujący sposób:

Jeśli długość hasła jest równa co najmniej 8 i  
 liczba dużych liter w haśle jest równa co najmniej 1 i  
 liczba małych liter w haśle jest równa co najmniej 1 i  
 liczba cyfr w haśle jest równa co najmniej 1, wtedy hasło jest prawidłowe.  
 W przeciwnym przypadku  
 hasło nie spełnia wymagań.

W następnej kolejności omówię funkcję numberUpperCase.

### **Listing 12.5. Program walidujący hasło: moduł numberUpperCase**

```

32 // Funkcja numberUpperCase przyjmuje jako argument
33 // ciąg znaków i zwraca liczbę występujących w nim
34 // dużych liter
35 Function Integer numberUpperCase(String str)
36     // Zmienna, w której zapiszemy liczbę dużych liter
37     Declare Integer count = 0
38
39     // Zmienna, za pomocą której będziemy śledzić poszczególne znaki w ciągu str
40     Declare Integer index
41
42     // Śledzimy poszczególne znaki w ciągu
43     // i liczymy duże litery
44     For index = 0 To length(str) - 1
45         If isUpper(str[index]) Then
46             Set count = count + 1
47         End If
48     End For
49
50     // Zwracamy liczbę dużych liter
51     Return count
52 End Function
53

```

Funkcja ta przyjmuje jako argument ciąg znaków, który przekazywany jest do parametru str. W linii 37. deklaruję zmienną typu Integer o nazwie count, która inicjalizuję wartością 0. Wykorzystam tę zmienną w roli akumulatora, w którym będę zapisywać liczbę dużych liter występujących w ciągu znaków przypisanym do parametru str. W linii 40. deklaruję kolejną zmienną typu Integer o nazwie index. Korzystam z niej w pętli rozpoczynającej się od linii 22., gdzie pełni rolę licznika śledzącego kolejne znaki w ciągu znaków str. W instrukcji If-Then rozpoczynającej się w linii 45. wywołuję funkcję biblioteczną isUpper, dzięki której sprawdzam, czy znak str[index] jest dużą literą. Jeśli tak, w linii 46. inkrementuję zmienną count. Kiedy pętla zakończy działanie, w zmiennej count będzie się znajdowała liczba informująca o liczbie dużych liter występujących w parametrze str. W linii 51. funkcja zwraca tę wartość.

Następnie omówię funkcję numberLowerCase.

**Listing 12.5. Program walidujący hasło: moduł numberLowerCase**

```

54 // Funkcja numberLowerCase przyjmuje jako argument
55 // ciąg znaków i zwraca liczbę występujących w nim
56 // małych liter
57 Function Integer numberLowerCase(String str)
58 // Zmienna, w której zapiszemy liczbę małych liter
59 Declare Integer count = 0
60
61 // Zmienna, za pomocą której będziemy śledzić poszczególne znaki w ciągu str
62 Declare Integer index
63
64 // Śledzimy poszczególne znaki w ciągu
65 // i zliczamy małe litery
66 For index = 0 To length(str) - 1
67     If isLower(str[index]) Then
68         Set count = count + 1
69     End If
70 End For
71
72 // Zwracamy liczbę małych liter
73 Return count
74 End Function
75

```

Funkcja ta wygląda niemal identycznie jak funkcja numberUpperCase, z wyjątkiem linii 67., gdzie wywołuję funkcję biblioteczną isLower, która sprawdza, czy znak str[index] jest małą literą. Na końcu funkcji, w linii 73., zwracam wartość przypisaną do zmiennej count, która tym razem jest równa liczbie małych liter występujących w parametrze str.

W następnej kolejności omówię funkcję numberDigits.

**Listing 12.5. Program walidujący hasło: moduł numberDigits**

```

76 // Funkcja numberDigits przyjmuje jako argument
77 // ciąg znaków i zwraca liczbę występujących w nim
78 // cyfr
79 Function Integer numberDigits(String str)
80 // Zmienna, w której zapiszemy liczbę cyfr
81 Declare Integer count = 0
82
83 // Zmienna, za pomocą której będziemy śledzić poszczególne znaki w ciągu str
84 Declare Integer index
85
86 // Śledzimy poszczególne znaki w ciągu
87 // i zliczamy cyfry
88 For index = 0 To length(str) - 1
89     If isDigit(str[index]) Then
90         Set count = count + 1
91     End If
92 End For
93
94 // Zwracamy liczbę cyfr
95 Return count
96 End Function

```

Funkcja ta wygląda niemal identycznie jak funkcje `numberUpperCase` i `numberLowerCase`, z wyjątkiem linii 89., gdzie za pomocą funkcji bibliotecznej `isDigit` sprawdzam, czy znak `str[index]` jest cyfrą. Na końcu funkcji, w linii 95., zwracam wartość przypisaną do zmiennej `count`, która tym razem jest równa liczbie cyfr występujących w parametrze `str`.

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Program sprawdza, czy wprowadzone hasło spełnia następujące wymagania:

- (1) Hasło musi się składać z co najmniej 8 znaków.
- (2) Hasło musi zawierać co najmniej 1 dużą literę.
- (3) Hasło musi zawierać co najmniej 1 małą literę.
- (4) Hasło musi zawierać co najmniej 1 cyfrę.

Wprowadź hasło.

**love [Enter]**

Hasło nie spełnia wymagań.

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Program sprawdza, czy wprowadzone hasło spełnia następujące wymagania:

- (1) Hasło musi się składać z co najmniej 8 znaków.
- (2) Hasło musi zawierać co najmniej 1 dużą literę.
- (3) Hasło musi zawierać co najmniej 1 małą literę.
- (4) Hasło musi zawierać co najmniej 1 cyfrę.

Wprowadź hasło.

**1oVe679g [Enter]**

Hasło jest prawidłowe.

## **Wstawianie i usuwanie znaków w ciągu znaków**

W większości języków programowania dostępne są funkcje lub moduły biblioteczne służące do wstawiania i usuwania znaków w ciągu. W pseudokodzie będziemy do tego celu używać modułów bibliotecznych opisanych w tabeli 12.3.

**Tabela 12.3.** Moduły do wstawiania i usuwania znaków w ciągach znaków

Moduł	Opis
<code>insert(ciąg znaków 1, pozycja, ciąg znaków 2)</code>	<code>ciąg znaków 1</code> jest typu <code>String</code> , <code>pozycja</code> jest typu <code>Integer</code> , a <code>ciąg znaków 2</code> jest typu <code>String</code> . Moduł wstawia do <code>ciągu znaków 2</code> <code>ciąg znaków 1</code> w miejscu wskazanym przez parametr <code>pozycja</code> .
<code>delete(ciąg znaków, początek, koniec)</code>	<code>ciąg znaków</code> jest typu <code>String</code> , a <code>początek</code> i <code>koniec</code> są typu <code>Integer</code> . Moduł usuwa z <code>ciągu znaków</code> wszystkie znaki, począwszy od pozycji wskazanej przez parametr <code>początek</code> , a kończąc na pozycji wskazanej przez parametr <code>koniec</code> — łącznie ze znakiem na pozycji <code>koniec</code> .

Oto przykład wykorzystania modułu `insert`:

```
Declare String str = "New City"
insert(str, 4, "York ")
Display str
```

Drugie polecenie wstawia do ciągu znaków `str` ciąg znaków "York ", zaczynając od pozycji 4. Znaki, które znajdowały się dotychczas w zmiennej `str`, poczawszy od pozycji 4, zostaną odpowiednio przesunięte w prawo. Zmienna `str` zostanie także odpowiednio rozszerzona w pamięci komputera, aby pomieścić wstawione znaki. Gdyby powyższy kod był prawdziwym programem, po jego uruchomieniu na ekranie wyświetliłby się napis *New York City*.

A oto przykład wykorzystania modułu `delete`:

```
Declare String str = "Zjadłem dzisiaj 1000 jagód!"
delete(str, 18, 19)
Display str
```

Drugie polecenie usunie z ciągu znaków `str` znaki znajdujące się na pozycjach od 18 do 19. Znaki znajdujące się za pozycją 20 zostaną odpowiednio przesunięte w lewo. Gdyby powyższy kod był prawdziwym programem, po jego uruchomieniu na ekranie wyświetliłby się napis *Zjadłem dzisiaj 10 jagód!*.

## **W centrum uwagi**

### Formatowanie numeru telefonu i usuwanie formatowania



Numery telefonów w Stanach Zjednoczonych mają najczęściej następujący format:

(XXX)XXX-XXXX

X oznacza pojedynczą cyfrę. Trzy cyfry umieszczone w nawiasach to numer kierunkowy. Trzy kolejne cyfry stanowią prefiks, a ostatnie cztery cyfry występujące po łączniku to numer abonenta. Oto przykładowy numer telefonu:

(919)555-1212

Nawiasy i łącznik sprawiają, że numer telefonu jest dla użytkownika bardziej czytelny, jednak znaki te nie są przetwarzane przez komputer. Numery telefonów zazwyczaj zapisuje się w systemach komputerowych bez żadnego formatowania, jako ciąg cyfr, w następujący sposób:

9195551212

Program, w którym przetwarzane są numery telefonów, po wprowadzeniu numeru przez użytkownika usuwa formatowanie. Oznacza to, że przed zapisaniem numeru telefonu w pliku lub dalszym jego przetwarzaniem zostaną z niego usunięte nawiasy i łącznik. Ponadto gdy program wyświetla lub drukuje numer telefonu, musi go sformatować, dodając nawiasy i łącznik.

Na listingu 12.6 przedstawiłem pseudokod algorytmu, który usuwa formatowanie z numeru telefonu. W module `main` proszę użytkownika o wprowadzenie numeru

telefonu. Następnie, aby sprawdzić, czy numer telefonu jest odpowiednio sformatowany, wywołuję funkcję `isValidFormat`. Jeśli numer ma prawidłowy format, za pomocą modułu `unformat` usuwam z niego nawiasy i łącznik. Na końcu wyświetlам niesformatowany numer telefonu. Przyjrzymy się najpierw modułowi `main`:

#### **Listing 12.6. Program usuwający formatowanie z numeru telefonu: moduł main**

```

1 Module main()
2 // Deklarujemy zmienną, w której zapiszemy numer telefonu
3 Declare String phoneNumber
4
5 // Prosimy użytkownika o wprowadzenie numeru telefonu
6 Display "Wprowadź numer telefonu. Numer powinien mieć"
7 Display "następujący format: (XXX)XXX-XXXX."
8 Input phoneNumber
9
10 // Jeśli wprowadzone dane mają prawidłowy format, usuwany z numeru telefonu formatowanie
11 If isValidFormat(phoneNumber) Then
12     unformat(phoneNumber)
13     Display "Niesformatowany numer telefonu: ", phoneNumber
14 Else
15     Display "Numer telefonu ma niewłaściwy format."
16 End If
17 End Module
18

```

W linii 3. deklaruję zmienną typu `String` o nazwie `phoneNumber`, w której zapiszę numer telefonu wprowadzony przez użytkownika. W liniach od 6. do 8. proszę użytkownika o wprowadzenie prawidłowo sformatowanego numeru telefonu, następnie go odczytuję i zapisuję w zmiennej `phoneNumber`.

W instrukcji `If-Then-Else` rozpoczynającej się od linii 11. przekazuję do funkcji `isValidFormat` argument w postaci zmiennej `phoneNumber`. Gdy numer telefonu ma prawidłowy format, funkcja zwraca wartość `True`. W przeciwnym razie zwraca wartość `False`. Jeśli funkcja zwróci wartość `True`, w linii 12. przekazuję zmienną jako argument modułu `unformat`. Moduł `unformat` przyjmuje argument przez referencję i usuwa z numeru telefonu nawiasy i łącznik. W linii 13. wyświetlam niesformatowany numer telefonu.

Jeśli użytkownik wprowadzi błędnie sformatowany numer telefonu, funkcja `isValidFormat` w linii 11. zwróci wartość `False` i wykona się polecenie `Display` w linii 15.

Następnie omówię funkcję `isValidFormat`.

#### **Listing 12.6. Program usuwający formatowanie z numeru telefonu (kontynuacja): funkcja isValidFormat**

```

19 // Funkcja isValidFormat przyjmuje jako argument ciąg znaków
20 // i sprawdza, czy jest on odpowiednio sformatowanym numerem telefonu
21 // w Stanach Zjednoczonych
22 // Numer powinien mieć format (XXX)XXX-XXXX
23 // Jeśli numer telefonu ma prawidłowy format, funkcja zwraca wartość True
24 // W przeciwnym razie zwraca wartość False
25 Function Boolean isValidFormat(str)

```

```

26 // Zmienna lokalna wskazująca, czy numer jest prawidłowo sformatowany
27 Declare Boolean valid
28
29 // Sprawdzamy, czy ciąg znaków str jest prawidłowo sformatowany
30 If length(str) == 13 AND str[0] == "(" AND
31     str[4] == ")" AND str[8] == "-" Then
32     Set valid = True
33 Else
34     Set valid = False
35 End If
36
37 // Zwracamy wartość zapisaną w zmiennej valid
38 Return valid
39 End Function
40

```

Funkcja ta przyjmuje jako argument ciąg znaków, który jest przekazywany do parametru `str`. W linii 27. deklaruję zmienną lokalną typu Boolean o nazwie `valid`, która będzie pełniła funkcję flagi informującej, czy ciąg zapisany w zmiennej `str` jest prawidłowo sformatowanym numerem telefonu.

Polecenie `If-Then-Else` rozpoczynające się od linii 30. sprawdza złożone wyrażenie boolowskie. Polecenie to można opisać w następujący sposób:

Jeśli długość ciągu znaków jest równa 13 i znak na pozycji 0 jest równy "(" i znak na pozycji 4 jest równy ")" i znak na pozycji 8 jest równy "-", wtedy ustaw zmienną `valid` na `True`.

W przeciwnym przypadku  
Ustaw zmienną `valid` na `False`.

Po wykonaniu polecenia `If-Then-Else`, w zależności od tego, czy zmienna `str` jest prawidłowo sformatowana, w zmiennej `valid` będzie zapisana wartość `True` lub `False`. W linii 38. funkcja zwraca wartość zapisaną w zmiennej `valid`.

Następnie opiszę moduł `unformat`.

#### **Listing 12.6. Program usuwający formatowanie z numeru telefonu (kontynuacja): moduł unformat**

```

41 // Moduł unformat przyjmuje jako argument ciąg znaków przekazywany przez referencję
42 // Zakładamy, że ciąg znaków to prawidłowo sformatowany numer telefonu
43 // w postaci (XXX)XXX-XXXX.
44 // Moduł usuwa z ciągu znaków formatowanie
45 // (nawiasy i łącznik)
46 Module unformat(String Ref str)
47     // Na początku usuwamy lewy nawias na pozycji 0
48     delete(str, 0, 0)
49
50     // Następnie usuwamy prawy nawias; po usunięciu lewego nawiasu
51     // prawy nawias będzie się znajdował
52     // na pozycji 3
53     delete(str, 3, 3)
54
55     // Następnie usuwamy łącznik; po usunięciu obu nawiasów
56     // łącznik będzie się znajdował

```

```

57 // na pozycji 6
58 delete(str, 6, 6)
59 End Module

```

Moduł przyjmuje jako argument ciąg znaków przekazywany przez parametr typu referencyjnego o nazwie str. Zakładam tutaj, że ciąg znaków jest sformatowany jako (XXX)XXX-XXXX. W linii 48. usuwam znak na pozycji 0 (czyli "("). Pozostałe znaki przesuną się automatycznie w lewo, zapełniając puste miejsce po usuniętym znaku. Następnie w linii 53. usuwam znak na pozycji 3 (czyli ")"). Pozostałe znaki, od znaku na pozycji 4, przesuną się automatycznie w lewo, zapełniając puste miejsce po usuniętym znaku. Następnie w linii 58. usuwam znak na pozycji 6 (czyli łącznik). Pozostałe znaki, znajdujące się na prawo do łącznika, przesuną się automatycznie w lewo o jedno miejsce. Po wykonaniu tego polecenia w zmiennej str będzie się znajdował niesformatowany numer telefonu, składający się jedynie z cyfr.

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź numer telefonu. Numer powinien mieć następujący format: (XXX)XXX-XXXX.

**(919)555-1212 [Enter]**

Niesformatowany numer telefonu: 9195551212

Przyjrzymy się teraz algorytmowi, który jako dane wejściowe przyjmuje niesformatowany numer telefonu (czyli ciąg dziesięciu cyfr) i formatuje go, wstawiając w odpowiednie miejsca nawiasy i łącznik. Poniżej przedstawiam moduł main.

#### **Listing 12.7. Program formatujący numer telefonu: moduł main**

```

1 Module main()
2 // Deklarujemy zmienną, w której zapiszemy numer telefonu
3 Declare String phoneNumber
4
5 // Prosimy użytkownika o wprowadzenie numeru telefonu
6 Display "Wprowadź niesformatowany numer telefonu składający się z 10 cyfr."
7 Input phoneNumber
8
9 // Jeśli wprowadzony ciąg znaków składa się z 10 znaków, formatujemy go
10 If length(phoneNumber) == 10 Then
11     format(phoneNumber)
12     Display "Sformatowany numer telefonu: ", phoneNumber
13 Else
14     Display "Numer nie składa się z 10 cyfr."
15 End If
16 End Module
17

```

W linii 3. deklaruję zmienną typu String o nazwie phoneNumber, w której zapiszę numer telefonu wprowadzony przez użytkownika. W linii 6. proszę użytkownika o wprowadzenie niesformatowanego numeru telefonu składającego się z 10 cyfr, a w linii 7. zapisuję go w zmiennej phoneNumber. W poleceniu If-Then rozpoczynającym się od

linii 10. wywołuję funkcję biblioteczną `length`, dzięki której weryfikuję, czy użytkownik wprowadził ciąg znaków składający się dokładnie z 10 znaków. Jeśli numer ma prawidłową długość, w linii 11. wywołuję moduł `format` i przekazuję do niego jako argument zmiennej `phoneNumber`. Moduł `format` przyjmuje argument przez referencję, a jego zadanie polega na wstawieniu w odpowiednie miejsca ciągu znaków i łącznika tak, aby numer telefonu miał postać (XXX)XXX-XXXX. Sformatowany numer telefonu wyświetlam w linii 12. Jeśli numer wprowadzony przez użytkownika ma długość różną od 10, w linii 14. wyświetlам komunikat o błędzie.

Następnie omówię moduł `format`.

### **Listing 12.7**

```

18 // Moduł format module przyjmuje jako argument ciąg znaków przekazywany przez referencję
19 // Zakładamy, że ciąg znaków to niesformatowany 10-cyfrowy numer telefonu
20 // Moduł formatuje ciąg znaków
21 // w następujący sposób: (XXX)XXX-XXXX
22 Module format(String Ref str)
23     //Najpierw na pozycji 0 wstawiamy lewy nawias
24     insert(str, 0, "(")
25
26     //Następnie na pozycji 4 wstawiamy prawy nawias
27     insert(str, 4, ")")
28
29     //Następnie na pozycji 8 wstawiamy łącznik
30     insert(str, 8, "-")
31 End Module

```

Moduł przyjmuje jako argument ciąg znaków przekazywany przez parametr typu referencyjnego. W linii 24. wywołuję moduł biblioteczny `insert`, który wstawia znak "(" na pozycji 0. Wszystkie znaki zostaną więc przesunięte o jedno miejsce w prawo tak, aby pomieścić wstawiony znak. W linii 27. wstawiam na pozycji 4 znak ")", przesuwając tym samym wszystkie kolejne znaki o jedno miejsce w prawo. W linii 30. wstawiam na pozycji 8 znak "-", przesuwając kolejne znaki o jedno miejsce w prawo. Po wykonaniu tego polecenia ciąg znaków zapisany w zmiennej `str` będzie miał format (XXX)XXX-XXXX.

### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź niesformatowany numer telefonu składający się z 10 cyfr.

**9195551212 [Enter]**

Sformatowany numer telefonu: (919)555-1212



### **Punkt kontrolny**

12.1. Założmy, że w programie pojawia się następująca deklaracja zmiennej:

```
Declare String name = "Jan"
```

Co wyświetli się po wykonaniu poniższego polecenia?

```
Display name[2]
```

12.2. Założmy, że w programie pojawia się następująca deklaracja zmiennej:

```
Declare String str = "Taran"
```

Napisz polecenie, które zamieni pierwszą literę w zmiennej str na „B”.

12.3. Zaprojektuj algorytm, który będzie sprawdzał, czy pierwszy znak w zmiennej str jest cyfrą, a jeśli tak, usunie ten znak z ciągu.

12.4. Zaprojektuj algorytm, który będzie sprawdzał, czy pierwszy znak w zmiennej str jest dużą literą, a jeśli tak, zamieni ten znak na "0".

12.5. Założmy, że w programie pojawia się następująca deklaracja zmiennej:

```
Declare String str = "świecie!"
```

Napisz polecenie, które wstawi na początku zmiennej str ciąg znaków "Witaj, ". Po wykonaniu polecenia, w zmiennej str powinien być zapisany ciąg znaków "Witaj, świecie!".

12.6. Założmy, że w programie pojawia się następująca deklaracja zmiennej:

```
Declare String city = "Boston"
```

Napisz polecenie, które ze zmiennej city usunie trzy pierwsze znaki.

## 12.3

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

## Java

### Przetwarzanie tekstu

#### Literały ciągów znaków w języku Java

W Javie występuje różnica między literałem ciągu znaków a literałem pojedynczego znaku. Literały ciągów znaków są ujęte w cudzysłowy, natomiast literały pojedynczych znaków są zawarte w apostrofach. Na przykład poniższy tekst jest literałem ciągu znaków:

```
"A"
```

A oto literal pojedynczego znaku:

```
'A'
```

## **Przetwarzanie tekstu znak po znaku w języku Java**

W Javie ciągi znaków są obiektami niezmiennymi, co oznacza, że po ich utworzeniu w pamięci nie można już zmienić ich wartości. Z powodu tego ograniczenia należy podczas przetwarzania tekstu wykorzystać zmienną typu *String*. Jako alternatywę dla niej Java pozwala utworzyć obiekt typu *StringBuilder*. W obiektach tych można przechowywać ciągi znaków, umożliwiając one także bezpośrednie modyfikowanie poszczególnych znaków składających się na taki ciąg. W tym rozdziale będziemy pracować z obiektami typu *StringBuilder*.

Oto przykład inicjalizacji obiektu *StringBuilder*:

```
StringBuilder cityName = new StringBuilder("Wrocław");
```

Instrukcja ta tworzy obiekt typu *StringBuilder* o nazwie *cityName*. Obiekt zawiera ciąg znaków "Wrocław". Możemy użyć go tutaj do operowania poszczególnymi znakami ciągu.

Aby uzyskać dostęp do określonego znaku znajdującego się w określonej lokalizacji, wykorzystamy metodę *charAt* w obiekcie *StringBuilder*. Założymy na przykład, że mamy następującą deklarację:

```
StringBuilder name = new StringBuilder("Jakub");
```

Poniższy kod wyświetla pierwszy znak, który jest literą J:

```
System.out.println(name.charAt(0));
```

### **Zmiana wartości określonego znaku**

Metoda *charAt* obiektu *StringBuilder* zwraca wartość znaku znajdującego się w określonej lokalizacji. Jeśli chcemy zmienić wartość określonego znaku, musimy użyć metody *setCharAt*. Metoda ta przyjmuje dwa argumenty: pozycję znaku, który chcemy zmienić, i sam znak, który jest przeznaczony do zmiany. Oto przykład użycia tej metody:

```
StringBuilder str = new StringBuilder("bilet");
str.setCharAt(2, 'd');
System.out.println(str);
```

Kod ten wyświetli taki tekst:

```
bidet
```

### **Metody testowania znaków**

Java udostępnia metody podobne do funkcji bibliotecznych testowania znaków pokazanych w tabeli 12.2. Metody Javy przedstawia tabela 12.4.

### **Wstawianie i usuwanie znaków w metodzie *StringBuilder***

Istnieją także metody *StringBuilder*, które służą do wstawiania i usuwania znaków znajdujących się w ciągu. Metody te, zamieszczone w tabeli 12.5, są podobne do metod znanych już z modułów bibliotecznych przedstawionych w tabeli 12.3.

**Tabela 12.4.** Metody testowania znaków

Funkcja	Opis
<code>Character.isDigit(znak)</code>	Zwraca wartość True, jeśli <i>znak</i> jest cyfrą, a w przeciwnym razie zwraca wartość False.
<code>Character.isLetter(znak)</code>	Zwraca wartość True, jeśli <i>znak</i> jest literą alfabetu, a w przeciwnym razie zwraca wartość False.
<code>Character.isLowerCase(znak)</code>	Zwraca wartość True, jeśli <i>znak</i> jest małą literą, a w przeciwnym razie zwraca wartość False.
<code>Character.isUpperCase(znak)</code>	Zwraca wartość True, jeśli <i>znak</i> jest wielką literą, a w przeciwnym razie zwraca wartość False.
<code>Character.isWhiteSpace(znak)</code>	Zwraca wartość True, jeśli <i>znak</i> jest znakiem białym, a w przeciwnym razie zwraca wartość False. (Znak biały to znak spacji, tabulacji lub nowego wiersza).

**Tabela 12.5.** Metody StringBuilder do wstawiania i usuwania znaków

Funkcja	Opis
<code>nazwaObiektu.insert(pozycja, ciągZnaków)</code>	<i>nazwaObiektu</i> to obiekt typu <code>StringBuilder</code> , <i>pozycja</i> to wartość typu <code>int</code> , a <i>ciągZnaków</i> to wartość typu <code>String</code> . Metoda wstawia串 znaków do obiektu <code>StringBuilder</code> , zaczynając od <i>pozycja</i> .
<code>nazwaObiektu.delete(początek, koniec)</code>	<i>nazwaObiektu</i> to obiekt typu <code>StringBuilder</code> , <i>początek</i> i <i>koniec</i> to wartości typu <code>int</code> . Metoda usuwa z obiektu <code>StringBuilder</code> wszystkie znaki rozpoczynające się od pozycji określonej przez wartość <i>początek</i> , a kończące się na pozycji określonej przez wartość <i>koniec</i> . Znak znajdujący się w pozycji końcowej NIE jest uwzględniony przy usuwaniu.

Oto przykład, w jaki sposób możemy użyć metody `insert`:

```
StringBuilder str = new StringBuilder("New City");
str.insert(4, "York ");
System.out.println(str);
```

Druga instrukcja wstawia do obiektu `StringBuilder`串 znaków "York ", rozpoczynając od pozycji 4. Znaki, które są obecnie przechowywane w metodzie `StringBuilder`, począwszy od pozycji 4, są przenoszone w prawą stronę. Pamiętaj, że metoda `StringBuilder` jest automatycznie powiększana tak, aby pomieścić wstawione znaki. Jeśli te instrukcje byłyby kompletnym programem, a my byśmy go uruchomili, na ekranie wyświetliłyby się napis New York City.

Oto przykład, w jaki sposób możemy użyć metody `delete`:

```
StringBuilder str = new StringBuilder("Zjadłem dzisiaj 1000 jagód!");
str.delete(8, 10);
System.out.println(str)
```

W powyższym kodzie, w metodzie `StringBuilder`, druga instrukcja usuwa znaki znajdujące się na pozycjach od 8. do 9. Należy zauważyć, że pozycja określona przez drugi argument nie jest uwzględniona w tym usunięciu. (Działa zatem inaczej od modułu `delete` opisywanego wcześniej w niniejszej książce). Znaki, które poprzednio znajdowały się na pozycjach od 10. w góre, zostały przesunięte w lewą stronę w taki sposób, aby zająć miejsce pozostałe po usuniętych znakach. Jeśli te instrukcje byłyby kompletnym programem i zostały on uruchomiony, to zobaczylibyśmy napis `Zjadłem dzisiaj 10 jagód!` wyświetlony na ekranie.

## Python

### Przetwarzanie tekstu

#### Sposoby dostępu do poszczególnych znaków w ciągu w języku Python

Jak już wspomniałem, Python pozwala pobierać pojedyncze znaki znajdujące się w ciągu przez podanie ich indeksu. Na przykład poniższy kod tworzy串 znaków „Witaj”, a następnie używa indeksów w celu wypisania pierwszego znaku umieszczonego w tym ciągu:

```
greeting = 'Witaj'
print(greeting[0])
```

Notacji z indeksami można użyć w celu pobrania wybranych znaków znajdujących się w ciągu, ale nie można z niej skorzystać do zmiany wartości znaku znajdującego się w tym ciągu. Wynika to stąd, że ciągi znaków w Pythonie są niezmienne, co oznacza, że po ich utworzeniu nie można już ich zmienić.

Ze względu na to, że ciągi w Pythonie są niezmienne, nie można już umieścić wyrażenia `ciągZnaków[indeks]` po lewej stronie operatora przypisania. Na przykład ten kod wywoła błąd:

```
# Przypisanie tekstu 'Darek' do zmiennej friend
friend = 'Darek'
# Czy można zmienić pierwszy znak na "J"?
friend[0] = 'J'      # Nie, to spowoduje błąd!
```

Ostatnia instrukcja w tym kodzie spowoduje błąd, ponieważ próbuje zmienić wartość pierwszego znaku w ciągu 'Darek'. Ponieważ ciągi znaków są niezmienne, nie ma możliwości wstawiania i usuwania poszczególnych znaków w ciągu, więc w tym podrozdziale nie będziemy się już zajmować tym tematem.

#### Metody testowania znaków

Python udostępnia metody, które są podobne do funkcji bibliotecznych testowania znaków pokazanych w tabeli 12.2. Metody Pythona przedstawią tabela 12.6.

**Tabela 12.6.** Metody testowania znaków

Metoda	Opis
isalnum()	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko litery lub cyfry i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> .
isalpha()	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko litery i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> .
isdigit()	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko cyfry i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> .
islower()	Zwraca wartość <code>true</code> , jeśli wszystkie litery w ciągu znaków są małymi literami, a ciąg zawiera co najmniej jedną literę. W przeciwnym razie zwróci wartość <code>false</code> .
isspace()	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko białe znaki i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> . (Białe znaki to znaki spacji, nowego wiersza ( <code>\n</code> ) i tabulacji ( <code>\t</code> )).
isupper()	Zwraca wartość <code>true</code> , jeśli wszystkie litery w ciągu znaków są wielkimi literami, a ciąg zawiera co najmniej jedną literę. W przeciwnym razie zwróci wartość <code>false</code> .

Różnica między tymi metodami a funkcjami testowania znaków omówionymi wcześniej w tej książce polega na tym, że funkcje Pythona działają na całym ciągu znaków. Na przykład poniższy kod określa, czy wszystkie znaki zawarte w ciągu, do którego odnosi się zmienna `my_string`, są napisane wielką literą:

```
my_string = "ABC"
if my_string.isupper():
    print('Ten tekst jest napisany wielkimi literami.')
```

Kod ten wypisze na ekranie komunikat *Ten tekst jest napisany wielkimi literami*, ponieważ wszystkie znaki znajdujące się w ciągu przypisany do zmiennej `my_string` są wielkimi literami.

Powyższe metody można też zastosować do pojedynczego znaku znajdującego się w ciągu. Oto przykład:

```
my_string = "Abc"
if my_string[0].isupper():
    print('Pierwszy znak jest napisany wielką literą.')
```

Ten kod sprawdza, czy znak o indeksie 0 znajdujący się w zmiennej `my_string` jest napisany wielką literą (i w tym przypadku jest).

## C++

### Przetwarzanie tekstu

#### Literały znaków w języku C++

W języku C++ istnieje różnica między literałem ciągu znaków a literałem znaku. Literały ciągu są ujęte w podwójne cudzysłowy, natomiast literały pojedynczych znaków są zawarte w pojedynczych cudzysłowach. Na przykład poniższy tekst jest literałem ciągu znaków:

```
"A"
```

A oto literal pojedynczego znaku:

```
'A'
```

#### Przetwarzanie tekstu znak po znaku w języku C++

W C++ możliwa jest praca z poszczególnymi znakami znajdującymi się w ciągu przy wykorzystaniu notacji z indeksami, zgodnie z wcześniejszym opisem zawartym w tej książce. Oto przykład takiego działania:

```
string name = "Jakub";
cout << name[0] << endl;
cout << name[1] << endl;
cout << name[2] << endl;
cout << name[3] << endl;
cout << name[4] << endl;
```

Kod ten wyświetla następujący napis:

```
J
a
k
u
b
```

Poniższy kod pokazuje, w jaki sposób można wykorzystać notację z indeksami do zmiany określonego znaku w ciągu:

```
string str = "bilet";
str[2] = 'd';
cout << str << endl;
```

Kod ten wyświetli:

```
bidet
```

Zmienne typu `string` mają wbudowaną funkcję `length()`, która zwraca liczbę znaków znajdującej się w ciągu. Poniższy kod jest przykładem, w którym wykorzystujemy pętlę do przejścia przez wszystkie znaki zawarte w ciągu.

```
string name = "Jakub";
for (index = 0; index < name.length(); index++)
    cout << name[index] << endl;
```

Kod ten wyświetla następujący napis:

```
J  
a  
k  
u  
b
```

### Funkcje sprawdzania znaków

C++ udostępnia funkcje podobne do funkcji bibliotecznych testowania znaków pokazanych w tabeli 12.2. Funkcje C++ przedstawia tabela 12.7. (Aby skorzystać z tych funkcji, musisz pamiętać o wpisaniu w swoim programie dyrektywy `#include <cctype>`).

**Tabela 12.7.** Funkcje sprawdzania znaków

Metoda	Opis
<code>isalnum()</code>	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko litery lub cyfry i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> .
<code>isalpha()</code>	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko litery i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> .
<code>isdigit()</code>	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko cyfry i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> .
<code>islower()</code>	Zwraca wartość <code>true</code> , jeśli wszystkie litery znajdujące się w ciągu znaków są małymi literami, a ciąg zawiera co najmniej jedną literę. W przeciwnym razie zwróci wartość <code>false</code> .
<code>isspace()</code>	Zwraca wartość <code>true</code> , jeśli ciąg znaków zawiera tylko znaki białe i składa się z co najmniej jednego znaku. W przeciwnym razie zwróci wartość <code>false</code> . (Białe znaki to znaki spacji, nowej linii ( <code>\n</code> ) i tabulacji ( <code>\t</code> )).
<code>isupper()</code>	Zwraca wartość <code>true</code> , jeśli wszystkie litery zawarte w ciągu znaków są wielkimi literami, a ciąg zawiera co najmniej jedną literę alfabetu. W przeciwnym razie zwróci wartość <code>false</code> .

### Wstawianie i usuwanie znaków znajdujących się w ciągu znaków w języku C++

Istnieją także funkcje ciągów znaków przeznaczone do wstawiania i usuwania znaków znajdujących się w ciągu. Funkcje te, zamieszczone w tabeli 12.8, są podobne do metod znanych już z modułów bibliotecznych przedstawionych w tabeli 12.3.

Oto przykład, w jaki sposób możemy użyć funkcji `insert`:

```
string str = "New City";
str.insert(4, "York ");
cout << str << endl;
```

Druga instrukcja, rozpoczynając od pozycji mającej wartość 4, wstawia ciąg znaków "York " do funkcji `string`. Znaki, które są obecnie przechowywane w funkcji `string`, począwszy od pozycji zawierającej wartość 4, są przenoszone w prawą stronę. Pamięć

**Tabela 12.8.** Funkcje ciągów znaków do wstawiania i usuwania znaków

Funkcja	Opis
<code>ciagZnaków.insert(pozycja, ciag2)</code>	<code>ciagZnaków</code> to nazwa zmiennej typu <code>string</code> , <code>pozycja</code> to wartość typu <code>int</code> , a <code>ciag2</code> to ciąg znaków. Funkcja wstawia ciąg znaków <code>ciag2</code> do zmiennej <code>ciagZnaków</code> , zaczynając od <code>pozycja</code> .
<code>ciagZnaków.erase(start, liczbaZnaków)</code>	<code>ciagZnaków</code> to nazwa zmiennej typu <code>string</code> , <code>start</code> i <code>liczbaZnaków</code> to wartości typu <code>int</code> . Funkcja usuwa liczbę znaków określonyą przez zmienną <code>liczbaZnaków</code> , rozpoczynając od pozycji określonej przez zmienną <code>start</code> .

przeznaczona na ciąg znaków jest automatycznie powiększana tak, aby pomieścić wstawione znaki. Jeśli te instrukcje byłyby kompletnym programem, a my byśmy go uruchomiliśmy, na ekranie wyświetliły się napis New York City.

Oto przykład, w jaki sposób możemy użyć funkcji `erase`:

```
string str = "Zjadłem dzisiaj 1000 jagód!";
str.erase(8, 2);
cout << str << endl;
```

W powyższym kodzie druga instrukcja usuwa dwa znaki, począwszy od pozycji 8. w ciągu znaków. Znaki, które poprzednio pojawiły się od pozycji 10., będą przesunięte w lewo tak, aby zająć miejsce pozostałe po usuniętych znakach. Jeśli te instrukcje byłyby kompletnym programem i zostały on uruchomiony, to zobaczylibyśmy napis Zjadłem dzisiaj 10 jagód! wyświetlony na ekranie.

## Pytania kontrolne

### Test jednokrotnego wyboru

- Które z poniższych poleceń wyświetli pierwszy znak ciągu znaków zapisanego w zmiennej `str`?
  - Display `str[1]`
  - Display `str[0]`
  - Display `str[first]`
  - Display `str`
- Które z poniższych poleceń wyświetli ostatni znak ciągu znaków zapisanego w zmiennej `str`?
  - Display `str[-1]`
  - Display `str[length(str)]`
  - Display `str[last]`
  - Display `str[length(str) - 1]`
- Które z poniższych poleceń zamieni wartość przypisaną do zmiennej `str` na "blackberry", jeśli w zmiennej `str` znajduje się ciąg znaków "berry"?

- a) Set str[0] = "black"  
 b) Set str = str + "black"  
 c) insert(str, 0, "black")  
 d) insert(str, 1, "black")
4. Które z poniższych poleceń zamieni wartość przypisaną do zmiennej str na "Red", jeśli w zmiennej str znajduje się ciąg znaków "Redmond"?
- a) delete(str, 3, length(str))  
 b) delete(str, 3, 6)  
 c) Set str = str - "mond"  
 d) Set str[0] = "Red"
5. Co się stanie po wykonaniu poniższego polecenia?
- ```
Declare String name = "Adrian"
Set name[6] = "a"
```
- a) Wystąpi błąd  
 b) W zmiennej name zostanie zapisany ciąg znaków "Adriana"  
 c) W zmiennej name zostanie zapisany ciąg znaków "Adrian"  
 d) W zmiennej name zostanie zapisany ciąg znaków "Adrian a"

### Prawda czy fałsz?

- Jeśli do wskazania znaku w ciągu znaków użyjemy indeksu, pierwszy znak będzie miał indeks równy 0.
- Jeśli do wskazania znaku w ciągu znaków użyjemy indeksu, ostatni znak będzie miał indeks równy długości tego ciągu.
- Jeśli zmienna typu String o nazwie str zawiera ciąg znaków "Liczب", to polecenie str[5] = "y" zmieni ciąg znaków na "Liczبy".
- Moduł biblioteczny insert automatycznie rozszerza ciąg znaków tak, aby pomieścił wstawione do niego znaki.
- Moduł biblioteczny delete nie usuwa znaków z ciągu, lecz jedynie zastępuje je znakami spacji.
- Funkcja biblioteczna toUpper zmienia znak na duży, a funkcja biblioteczna toLower zmienia znak na mały.
- Odwołanie się za pomocą indeksu w pustym ciągu znaków spowoduje wystąpienie błędu.

### Krótką odpowiedź

1. Jakie numery mają indeksy pierwszego i ostatniego znaku w ciągu?

2. Jaki wynik wyświetli się po wykonaniu poniższego pseudokodu?

```
Declare String greeting = "Wszystkiego"
insert(greeting, 0, "Najlepszego")
Display greeting
```

3. Jaki wynik wyświetli się po wykonaniu poniższego pseudokodu?

```
Declare String str = "Bla bla bla"
delete(str, 3, 7)
Display str
```

4. Jaki wynik wyświetli się po wykonaniu poniższego pseudokodu?

```
Declare String str = "AaBbCcDd"
Declare Integer index
For Index = 0 To length(str) - 1
    If isLower(str[index]) Then
        Set str[index] = "_"
    End If
End For
Display str
```

5. Jaki wynik wyświetli się po wykonaniu poniższego pseudokodu?

```
Declare String str = "AaBbCcDd"
delete(str, 0, 0)
delete(str, 3, 3)
delete(str, 3, 3)
Display str
```

### **Warsztat projektanta algorytmów**

- Zaprojektuj algorytm, który będzie obliczał liczbę cyfr występujących w ciągu znaków str.
- Zaprojektuj algorytm, który będzie obliczał liczbę małych liter w ciągu znaków str.
- Zaprojektuj algorytm, który będzie obliczał liczbę dużych liter w ciągu znaków str.
- Zaprojektuj algorytm, który będzie usuwał pierwszą i ostatnią literę w ciągu znaków str.
- Zaprojektuj algorytm, który będzie zamieniał każdą literę „t” występującą w ciągu znaków str na literę „T”.
- Zaprojektuj algorytm, który będzie zamieniał każdą literę „X” występującą w ciągu znaków str na spację.
- Załóżmy, że w programie pojawia się deklaracja następującej zmiennej:

```
Declare String str = "P. Krawczyk"
```

Zaprojektuj algorytm, który będzie zamieniał skrót „P.” na słowo „Pan”.

## **Ćwiczenia z wykrywania błędów**

1. W którym miejscu poniższego pseudokodu jest błąd?

```
// Program przypisuje literę do pierwszego elementu
// w ciągu znaków
Declare String letters
Set letters[0] = "A"
Display "Pierwszą literą alfabetu jest ", letters
```

2. W którym miejscu poniższego pseudokodu jest błąd?

```
// Program sprawdza, czy użytkownik wprowadził
// pojedynczą cyfrę
Declare Integer digit
```

```

// Pobieramy dane od użytkownika
Display "Wprowadź cyfrę."
Input digit

// Sprawdzamy, czy użytkownik wprowadził pojedynczą cyfrę
If isDigit(digit[0]) AND length(digit) == 1 Then
    Display digit, " jest cyfrą."
Else
    Display digit, " NIE jest cyfrą."
End If

```

3. Dlaczego poniższy program nie zadziała zgodnie z opisem umieszczonym w komentarzach?

```

// Program oblicza liczbę znaków w ciągu
Declare String word
Declare Integer index
Declare Integer letters = 0

// Pobieramy dane od użytkownika
Display "Wprowadź słowo."
Input word

// Zliczamy znaki w ciągu
For index = 0 To length(word)
    Set count = count + 1
End For

Display "Słowo składa się z ", count, " znaków."

```

## Ćwiczenia programistyczne

### 1. Odwracanie ciągu znaków

Zaprojektuj program, który poprosi użytkownika o wprowadzenie ciągu znaków, a następnie wyświetli go w odwrotnej kolejności. Przykładowo jeśli użytkownik wprowadzi słowo grawitacja, program powinien wyświetlić tekst *ajcatiwarg*.

### 2. Duże znaki na początku zdania

Zaprojektuj program, który poprosi użytkownika o wprowadzenie ciągu znaków składającego się z kilku zdań. Następnie program powinien wyświetlić wprowadzone zdania, ale pierwszą literę w każdym zdaniu zamieniając na dużą. Przykładowo jeśli użytkownik wprowadzi tekst *cześć. mam na imię Jarek. a ty jak masz na imię?*, program powinien wyświetlić tekst *Cześć. Mam na imię Jarek. A ty jak masz na imię?*.

*Podpowiedź:* Poszczególne znaki można zamieniać na duże za pomocą funkcji bibliotecznej *ToUpper*.

### 3. Spółgłoski i samogłoski

Zaprojektuj program, który poprosi użytkownika o wprowadzenie ciągu znaków. Program powinien wyświetlić liczbę spółgłosek i liczbę samogłosek występujących w ciągu znaków.

#### 4. Sumowanie cyfr w ciągu znaków

Zaprojektuj program, który poprosi użytkownika o wprowadzenie ciągu znaków składającego się z ciągu cyfr. Program powinien wyświetlić sumę wszystkich cyfr w ciągu znaków. Przykładowo jeśli użytkownik wprowadzi ciąg 2514, program powinien wyświetlić liczbę 12, czyli sumę cyfr 2, 5, 1 i 4.

*Podpowiedź:* Aby zamienić znak na liczbę całkowitą, skorzystaj z funkcji bibliotecznej `stringToInteger`.

#### 5. Najczęściej występujący znak

Zaprojektuj program, który poprosi użytkownika o wprowadzenie ciągu znaków, a następnie wyświetli znak, który występuje w ciągu najczęściej.

#### 6. Konwertowanie alfanumerycznego numeru telefonu

Wiele amerykańskich firm prezentuje swój numer telefonu w następujący sposób: 555-GET-FOOD. Dzięki temu klientom łatwiej jest go zapamiętać. W tradycyjnym telefonie do każdej cyfry są przypisane litery, według następującego klucza:

A, B, C = 2  
D, E, F = 3  
G, H, I = 4  
J, K, L = 5  
M, N, O = 6  
P, Q, R, S = 7  
T, U, V = 8  
W, X, Y, Z = 9

Zaprojektuj program, który poprosi użytkownika o wprowadzenie 10-znakowego numeru telefonu, w formacie XXX-XXX-XXXX. Program powinien następnie wyświetlić numer telefonu, w którym litery zostaną zastąpione odpowiadającym im cyfrom. Przykładowo jeśli użytkownik wprowadzi numer telefonu 555-GET-FOOD, program powinien wyświetlić 555-438-3663.

#### 7. Oddzielanie słów

Zaprojektuj program, który jako dane wejściowe przyjmie zdanie, w którym wszystkie wyrazy zostały ze sobą połączone, a pierwszy znak każdego słowa jest dużą literą. Zamień to zdanie na taki ciąg znaków, w którym wyrazy są oddzielone spacjami, a tylko pierwszy znak w zdaniu jest dużą literą. Przykładowo ciąg znaków **Susza Wywarła Wielkie Spustoszenie** powinien zostać zamieniony na ciąg **Susza wywarła wielkie spustoszenie**.

*Podpowiedź:* Aby zamienić literę na małą, skorzystaj z funkcji bibliotecznej `toLower`.

#### 8. Świńska łacina

Zaprojektuj program, który jako dane wejściowe przyjmie zdanie w języku angielskim, a następnie każde z występujących w nim słów zamieni na słowo

w „świńskiej łacinie”. Jeden z wariantów świńskiej łaciny polega na przesunięciu pierwszej litery w wyrazie na jego koniec i dodaniu do wyrazu końcówki „ay”. Oto przykład:

Język angielski: I SLEPT MOST OF THE NIGHT  
 Świńska łacina: IAY LEPTSAY OSTMAY FOAY HETAY IGHTNAY

### 9. Tłumaczenie na alfabet Morse'a

Zaprojektuj program, który poprosi użytkownika o wprowadzenie ciągu znaków, a następnie zamieni go na ciąg zapisany za pomocą alfabetu Morse'a. W alfabetie Morse'a litery i znaki przestankowe są zapisywane za pomocą ciągu kropek i kresek. W tabeli 12.9 przedstawiłem fragment tego alfabetu.

**Tabela 12.9.** Alfabet Morse'a

| Znak      | Kod    | Znak | Kod   | Znak | Kod  | Znak | Kod  |
|-----------|--------|------|-------|------|------|------|------|
| spacja    | spacja | 6    | -.... | G    | --.  | Q    | --.. |
| przecinek | ----   | 7    | ---.. | H    | .-   | R    | .-.  |
| kropka    | .-..   | 8    | ----. | I    | ..   | S    | ...  |
| ?         | .....  | 9    | ----- | J    | ---- | T    | -    |
| 0         | ----   | A    | .-    | K    | --.  | U    | ..-  |
| 1         | .---   | B    | -...  | L    | .-.. | V    | ...- |
| 2         | ---    | C    | -..   | M    | --   | W    | --   |
| 3         | ---    | D    | -..   | N    | -.   | X    | --.. |
| 4         | ....   | E    | .     | O    | ---  | Y    | --.. |
| 5         | ....   | F    | ...   | P    | .... | Z    | --.. |

### 10. Szyfrowanie pliku

Szyfrowanie plików polega na zapisywaniu ich zawartości za pomocą specjalnego kodu. W tym ćwiczeniu zaprojektujesz program, za pomocą którego otworzysz plik i zaszyfrujesz jego zawartość. Założmy, że szyfrowany plik zawiera szereg ciągów znaków.

Program powinien otworzyć plik i odczytać z niego po kolejni każdy串 znaków. Po odczytaniu każdego串 powinien zamienić występujące w nim znaki na inne, a następnie zapisać powstały w ten sposób串 znaków w drugim pliku. Gdy program zakończy działanie, w drugim pliku będzie się znajdowała zakodowana zawartość pierwszego pliku.

### 11. Odszyfrowywanie pliku

Zaprojektuj program, który odszyfruje plik powstały po uruchomieniu programu z ćwiczenia programistycznego 10. Program powinien odczytać zawartość zaszyfrowanego pliku, a następnie odszyfrować zapisane w nim串 znaków i zapisać je w innym pliku.

## 12. Wykrywanie słabości hasła

Zaprojektuj program, który poprosi użytkownika o wprowadzenie hasła, a następnie przeanalizuje to hasło pod kątem następujących słabych punktów:

- składa się z mniej niż 8 znaków;
- nie zawiera co najmniej jednej dużej litery i jednej małej litery;
- nie zawiera co najmniej jednej cyfry;
- nie zawiera co najmniej jednego znaku specjalnego (znaku, który nie jest literą ani cyfrą);
- jest ciągiem kolejnych wielkich liter (takich jak ABCDE);
- jest ciągiem kolejnych małych liter (takich jak abcde);
- jest sekwencją kolejnych cyfr numerycznych (takich jak 12345);
- jest sekwencją powtarzających się znaków (np. ZZZZZ lub 55555).

Program powinien wyświetlać komunikaty wskazujące, czy w haśle użytkownika została wykryta któraś z tych słabości.



## TEMATYKA

- |                                               |                                            |
|-----------------------------------------------|--------------------------------------------|
| 13.1 Wprowadzenie do rekurencji               | 13.3 Przykłady algorytmów rekurencyjnych   |
| 13.2 Rozwiązywanie zadań za pomocą rekurencji | 13.4 Rzut oka na języki Java, Python i C++ |

## 13.1

## Wprowadzenie do rekurencji

**WYJAŚNIENIE:** Moduł rekurencyjny to taki moduł, który wywołuje sam siebie.

Miałeś już okazję zaobserwować sytuacje, w których jeden moduł wywoływał inny moduł. Przykładowo moduł `main` wywoływał moduł `A`, który z kolei wywoływał moduł `B`. Dany moduł może także wywołać sam siebie. Moduł, który wywołuje sam siebie, nazywamy **modułem rekurencyjnym**. Spójrz na przykładowy moduł `message`, który przedstawiłem na listingu 13.1.

**Listing 13.1**

```
1 Module main()
2   Call message()
3 End Module
4
5 Module message()
6   Display "To jest moduł rekurencyjny."
7   Call message()
8 End Module
```

**Wynik działania programu**

```
To jest moduł rekurencyjny.
To jest moduł rekurencyjny.
To jest moduł rekurencyjny.
To jest moduł rekurencyjny.
... ten tekst będzie się powtarzał w nieskończoność!
```

Moduł `message` wyświetla na ekranie komunikat *To jest moduł rekurencyjny*, a następnie wywołuje sam siebie. Operacja ta powtórzy się po każdym kolejnym wywołaniu modułu. Czy widzisz tutaj pewien problem? Takiego rekurencyjnego wywołania modułu nie da się zatrzymać. Ponieważ nie ma w nim instrukcji, która mogłaby zatrzymać jego wywoływanie, moduł działa jak pętla nieskończona.

Podobnie jak w przypadku pętli, moduł rekurencyjny musi w jakiś sposób kontrolować, ile razy ma się uruchomić. Na listingu 13.2 pokazałem zmodyfikowany pseudokod modułu `message`. W programie tym `message` przyjmuje argument w postaci liczby typu `Integer` który określa, ile razy moduł powinien wyświetlić komunikat.

### Listing 13.2



```

1 Module main()
2   //Przekazując do modułu argument 5,
3   //informujemy go, że chcemy wyświetlić komunikat
4   //pięć razy
5   Call message(5)
6 End Module
7
8 Module message(Integer times)
9   If times > 0 Then
10     Display "To jest moduł rekurencyjny."
11     Call message(times - 1)
12   End If
13 End Module

```

### Wynik działania programu

```

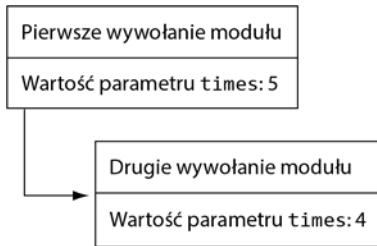
To jest moduł rekurencyjny.

```

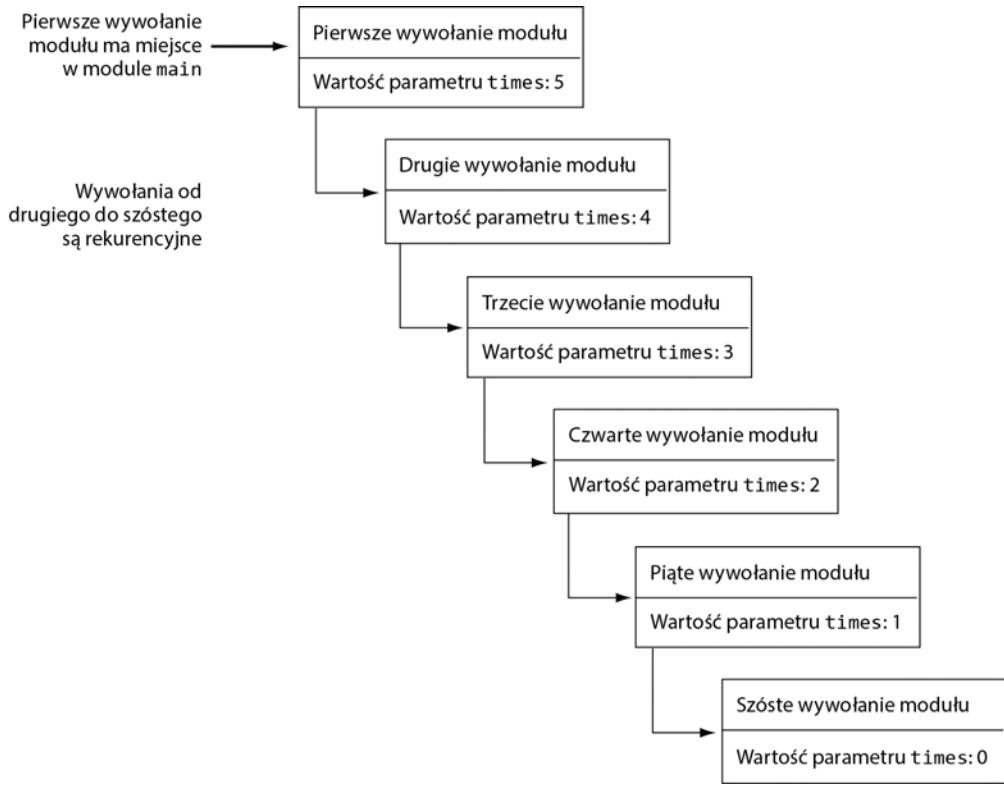
W module `message` umieściłem instrukcję `If-Then` (w liniach od 9. do 12.), dzięki czemu mogę kontrolować liczbę wywołań modułu. Program będzie wyświetlał komunikat i wywoływał moduł dopóty, dopóki parametr `times` będzie większy niż zero. Podczas każdego wywołania samego siebie do modułu przekazywana będzie wartość `times - 1`.

W module `main` wywołujemy moduł `message`, przekazując do niego argument równy 5. Podczas pierwszego wywołania modułu instrukcja `If-Then` wyświetli komunikat, a następnie moduł wywoła sam siebie, przekazując tym razem jako argument wartość 4. Ilustruje to rysunek 13.1.

Na diagramie przedstawionym na rysunku 13.1 widać dwa odrębne wywołania modułu `message`. Podczas każdego wywołania w pamięci komputera tworzona jest nowa instancja parametru `times`. Podczas pierwszego wywołania modułu parametr `times` jest równy 5. Gdy moduł wywoła sam siebie, utworzona zostanie nowa instancja parametru `times` i przekazana do niej zostanie wartość 4. Cykl ten będzie się powtarzał aż do momentu, gdy do modułu zostanie przekazana wartość równa 0. Ilustruje to rysunek 13.2.



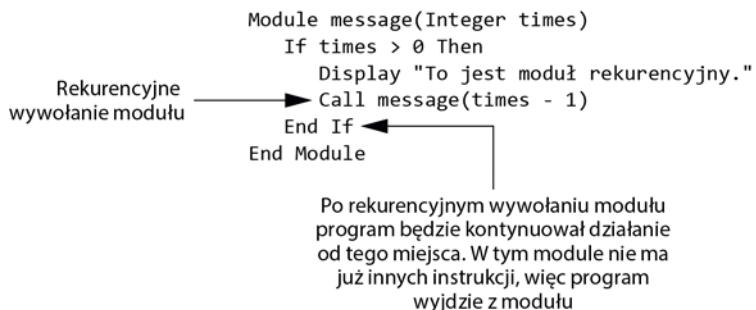
Rysunek 13.1. Dwa pierwsze wywołania modułu



Rysunek 13.2. Sześć wywołań modułu message

Jak widać, moduł `message` został wywołany sześciokrotnie — raz w module `main`, a kolejnych pięć razy wywołał sam siebie. To, ile razy moduł wywołał sam siebie, nazywamy głębokością rekurencji. W naszym przykładzie głębokość rekurencji jest równa 5. Podczas szóstego wywołania modułu parametr `times` będzie równy 0. W tym momencie warunek sprawdzany za pomocą instrukcji If-Then nie zostanie spełniony i moduł zakończy działanie. Program wyjdzie z szóstej instancji modułu do piątej instancji, zaraz za instrukcją wywołania modułu. Ilustruje to rysunek 13.3.

Ponieważ po wywołaniu modułu nie ma już żadnych innych instrukcji w module, program wyjdzie z niej i powróci do czwartej instancji modułu. Proces ten będzie się powtarzał aż do momentu, gdy program powróci do modułu `main`.



**Rysunek 13.3.** Po wywołaniu modułu sterowanie przekazywane jest do miejsca za wywołaniem

## 13.2

## Rozwiązywanie zadań za pomocą rekurencji

**WYJAŚNIENIE:** Jeśli dane zadanie można podzielić na mniejsze, identyczne zadania, to do jego rozwiązania można wykorzystać rekurencję.

Pseudokod przedstawiony na listingu 13.2 demonstruje wykorzystanie modułu rekurencyjnego. Rekurencja to potężne narzędzie, dzięki któremu można rozwiązywać zadania związane z cyklicznym przetwarzaniem danych, i często jest ona omawiana na kursach programowania. Zapewne jednak nie widzisz jeszcze, w jaki sposób można za jej pomocą rozwiązać jakiekolwiek zadanie.

Na wstępnie chciałem zaznaczyć, że aby wykonać zadanie, niekoniecznie trzeba sięgać po rekurencję. Każde zadanie, które można rozwiązać za pomocą rekurencji, można także rozwiązać za pomocą zwykłej pętli. Algorytmy rekurencyjne są zazwyczaj mniej wydajne niż pętle. To dlatego, że wywołanie modułu wymaga wykonania przez komputer kilku dodatkowych operacji. Jest to między innymi zaalokowanie pamięci na zmienne parametry i zmienne lokalne oraz zapisanie adresu, do którego ma wrócić program po wykonaniu modułu. Operacje te, zwane **narzutem**, mają miejsce podczas każdego wywołania modułu. Narzut taki nie występuje podczas korzystania z pętli.

Jednak pewne zadania programistyczne łatwiej jest wykonać za pomocą rekurencji niż za pomocą pętli. Można powiedzieć, że algorytmy z wykorzystaniem pętli komputer wykonuje szybciej, ale programista szybciej stworzy algorytm rekurencyjny. Ogólnie rzecz biorąc, moduł rekurencyjny działa w następujący sposób:

- jeśli zadanie można wykonać w danej chwili, moduł je wykonuje i zwraca wynik;
- jeśli zadania nie da się wykonać w danej chwili, moduł dzieli je na podobne, mniejsze zadanie i wywołuje sam siebie, aby je wykonać.

Aby zaimplementować to podejście, musimy najpierw wskazać przynajmniej jeden przypadek, w którym dane zadanie da się rozwiązać bez użycia rekurencji. Nazywamy go **przypadkiem bazowym**. Następnie musimy określić, w jaki sposób za pomocą

rekurencji rozwiążemy zadanie w pozostałych przypadkach. Nazywamy je **przypadkami rekurencyjnym**. W przypadku rekurencyjnym musimy ograniczyć zadanie do prostszej wersji zadania pierwotnego. Ograniczając zadanie przy każdym wywołaniu rekurencyjnym, dojdziemy w pewnym momencie do przypadku bazowego i zatrzymamy rekurencję.

## Obliczanie silni liczby za pomocą rekurencji

Na zamieszczonych wcześniej przykładach przedstawiłem moduły rekurencyjne. W większości języków programowania można także tworzyć funkcje rekurencyjne. Weźmy pewne zadanie matematyczne i przyjrzyjmy się aplikacji, w której do jego rozwiązania użyjemy funkcji rekurencyjnych. Symbol  $n!$  w matematyce oznacza silnię liczby  $n$ . Silnię nieujemnej liczby  $n$  można obliczyć za pomocą następujących reguł:

$$\begin{aligned} \text{jeśli } n = 0, \text{ wtedy } n! &= 1 \\ \text{jeśli } n > 0, \text{ wtedy } n! &= 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \end{aligned}$$

Zastąpmy teraz zapis  $n!$  zapisem  $\text{factorial}(n)$ , który bardziej przypomina kod programu komputerowego. Możemy teraz zapisać następujące reguły:

$$\begin{aligned} \text{jeśli } n = 0, \text{ wtedy } \text{factorial}(n) &= 1 \\ \text{jeśli } n > 0, \text{ wtedy } \text{factorial}(n) &= 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \end{aligned}$$

Zasady te wskazują, że kiedy liczba  $n$  jest równa 0, to jej silnia jest równa 1. Jeśli liczba  $n$  jest większa niż 0, wówczas jej silnia jest równa iloczynowi kolejnych liczb całkowitych od 1 do  $n$ . Przykładowo  $\text{factorial}(6)$  obliczamy jako  $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6$ .

Projektując algorytm rekurencyjny do obliczania silni danej liczby, musimy określić przypadek bazowy, będący wariatem zadania, które można rozwiązać, nie korzystając z rekurencji. W tym przypadku jest to sytuacja, gdy  $n$  jest równe 0:

$$\text{jeśli } n = 0, \text{ wtedy } \text{factorial}(n) = 1$$

Wiemy już więc, jak rozwiązać zadanie, gdy  $n$  jest równe 0, ale co zrobić, gdy  $n$  jest większe niż 0? Jest to przypadek rekurencyjny lub wariant zadania, które rozwiążemy za pomocą rekurencji. Możemy go wyrazić w następujący sposób:

$$\text{jeśli } n > 0, \text{ wtedy } \text{factorial}(n) = n \cdot \text{factorial}(n - 1)$$

Oznacza to, że jeśli  $n$  jest większe niż 0, wtedy silnia liczby  $n$  jest równa  $n$  razy silnia liczby  $n - 1$ . Zauważ, w jaki sposób ograniczyłem tutaj zadanie do prostszego zadania obliczenia silni liczby  $n - 1$ . Możemy więc zapisać reguły obliczania silni liczby następująco:

$$\begin{aligned} \text{jeśli } n = 0, \text{ wtedy } \text{factorial}(n) &= 1 \\ \text{jeśli } n > 0, \text{ wtedy } \text{factorial}(n) &= n \cdot \text{factorial}(n - 1) \end{aligned}$$

Pseudokod na listingu 13.3 przedstawia sposób, w jaki można w programie zapisać funkcję `factorial`.

**Listing 13.3**

```

1 Module main()
2 // Zmienna lokalna, w której zapiszemy
3 // liczbę wprowadzoną przez użytkownika
4 Declare Integer number
5
6 // Zmienna lokalna, w której zapiszemy
7 // silnię liczby
8 Declare Integer numFactorial
9
10 // Pobieramy liczbę od użytkownika
11 Display "Wprowadź nieujemną liczbę całkowitą."
12 Input number
13
14 // Obliczamy silnię liczby
15 Set numFactorial = factorial(number)
16
17 // Wyświetlamy silnię liczby
18 Display "Silnia liczby ", number,
19         " wynosi ", numFactorial
20 End Module
21
22 // Funkcja factorial oblicza silnię liczby
23 // przekazanej jako argument
24 // Zakładamy, że liczba ta jest nieujemna
25 Function Integer factorial(Integer n)
26     If n == 0 Then
27         Return 1
28     Else
29         Return n * factorial(n - 1)
30     End If
31 End Function

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź nieujemną liczbę całkowitą.

**4 [Enter]**

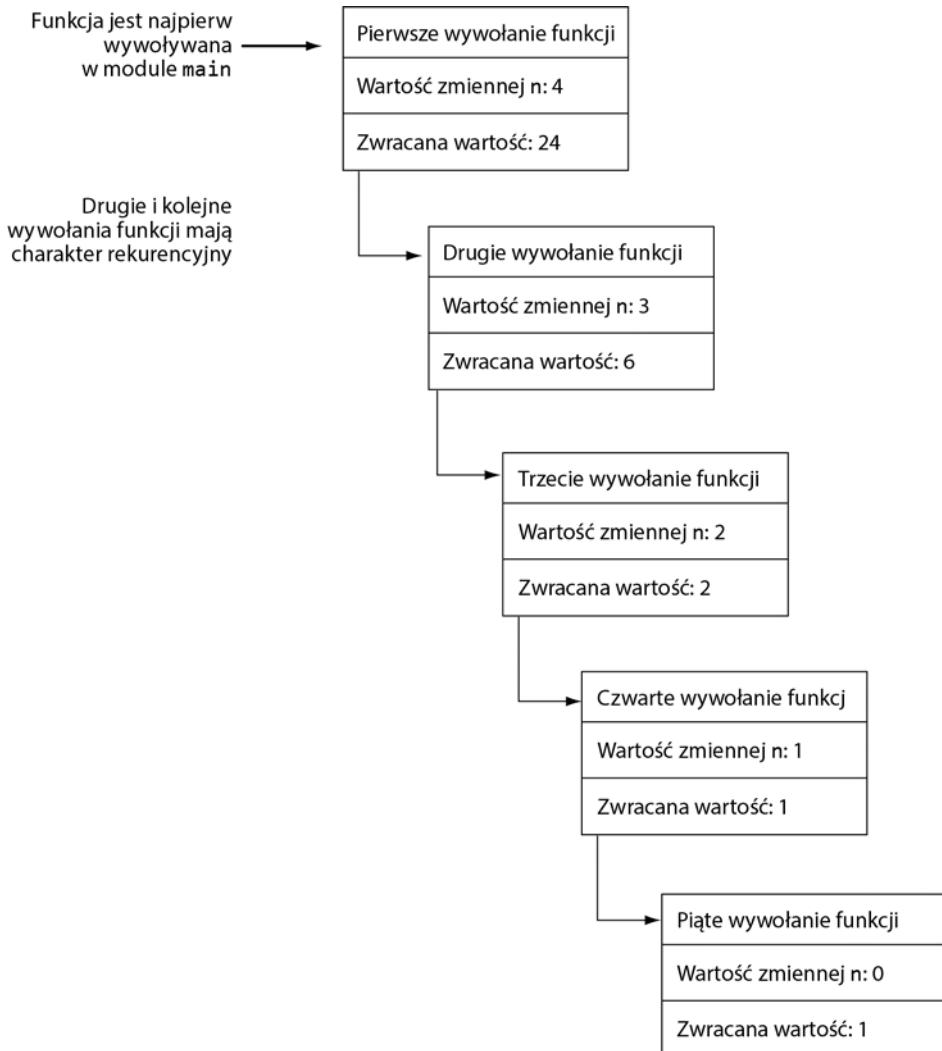
Silnia liczby 4 wynosi 24

W przykładowym wywołaniu programu wywołuję funkcję `factorial` i przekazuję do parametru `n` argument równy 4. Ponieważ `n` jest równe od 0, w pętli `If` wykonana zostanie klauzula `Else` zawierająca następujące polecenie:

`Return n * factorial(n - 1)`

Pomimo że jest to instrukcja `Return`, nie spowoduje ona natychmiastowego powrotu do głównej części programu. Zanim funkcja będzie mogła zwrócić wartość, musi najpierw obliczyć wartość zwracaną przez wyrażenie `factorial(n - 1)`. Funkcja `factorial` będzie wywoływaną rekurencyjnie aż do momentu, gdy podczas piątego jej wywołania parametr `n` będzie równy 0. Na rysunku 13.4 przedstawiłem wartość parametru `n` i zwracaną wartość podczas każdego wywołania funkcji.

Rysunek ilustruje powód, dla którego w algorytmie rekurencyjnym musimy z każdym wywołaniem funkcji upraszczać zadanie pierwotne. Ostatecznie rekurencja się zakończy i zwróci wynik końcowy.



**Rysunek 13.4.** Wartość zmiennej n i wartość zwracana podczas poszczególnych wywołań funkcji

Z każdym rekurencyjnym wywołaniem funkcja rozwiązuje prostszą wersję pierwotnego zadania, aż dojdzie do przypadku bazowego. Przypadek bazowy nie wymaga stosowania rekurencji i zatrzymuje serię wywołań rekurencyjnych funkcji.

Zadanie ograniczymy zazwyczaj poprzez zmniejszanie w każdym kolejnym wywołaniu wartości jednego z parametrów funkcji. W przypadku funkcji obliczającej silnię wartość parametru n będzie z każdym wywołaniem funkcji coraz bliższa wartości 0. Gdy parametr będzie równy 0, funkcja zwróci po prostu wartość i nie wywoła samej siebie po raz kolejny.

## Rekurencja bezpośrednia i rekurencja pośrednia

Przykłady modułów i funkcji rekurencyjnych, które dotychczas przedstawiłem, są wywoływane bezpośrednio. Taką sytuację nazywamy **rekurencją bezpośrednią**. Można jednak w programie wywołać **rekurencję pośrednią**. Ma to miejsce wtedy, gdy moduł A wywołuje moduł B, który z kolei wywołuje moduł A. W takim scenariuszu może brać udział nawet większa liczba modułów. Przykładowo moduł A wywołuje moduł B, który wywołuje moduł C, który wywołuje moduł A.



### Punkt kontrolny

- 13.1. Faktem jest, że algorytm rekurencyjny wiąże się z większym narzutem niż algorytm iteracyjny. Dlaczego tak jest?
- 13.2. Czym w rekurencji jest przypadek bazowy?
- 13.3. Co to jest przypadek rekurencyjny?
- 13.4. Co powoduje, że moduł rekurencyjny przestaje wywoływać sam siebie?
- 13.5. Co to jest rekurencja bezpośrednią? Co to jest rekurencja pośrednia?

### 13.3

## Przykłady algorytmów rekurencyjnych

### Sumowanie wartości fragmentu tablicy za pomocą rekurencji

W poniższym przykładzie przyjrzymy się funkcji `rangeSum`, która za pomocą rekurencji oblicza sumę wartości elementów tablicy z określonego przedziału indeksów. Funkcja przyjmuje następujące argumenty: tablica liczb `Integer`, liczba `Integer` wskazująca indeks, od którego ma rozpocząć się sumowanie, liczba `Integer` określająca indeks elementu, na którym ma zakończyć się sumowanie. Oto przykład użycia takiej funkcji:

```
Constant Integer SIZE = 9
Declare Integer numbers[SIZE] = 1, 2, 3, 4, 5, 6, 7, 8, 9
Declare Integer sum;
Set sum = rangeSum(numbers, 3, 7)
```

W ostatnim poleceniu pseudokodu wskazuję, że funkcja `rangeSum` ma zwrócić sumę elementów od 3. do 7. zawartych w tablicy `numbers`. Zwrócona wartość, która w tym przypadku będzie równa 30, zostanie przypisana do zmiennej `sum`. Oto definicja funkcji `rangeSum`:

```
Function Integer rangeSum(Integer array[], Integer start,
                           Integer end)
  If start > end Then
    Return 0
  Else
    Return array[start] + rangeSum(array, start + 1, end)
  End If
End Function
```

Przypadkiem bazowym tej funkcji jest sytuacja, gdy parametr `start` jest większy niż parametr `end`. Kiedy nastąpi taka sytuacja, funkcja zwróci wartość 0. W przeciwnym razie funkcja wykona następujące polecenie:

```
Return array[start] + rangeSum(array, start + 1, end)
```

Polecenie to zwraca wartość równą sumie elementu `array[start]` i wartości zwanej przez rekurencyjne wywołanie funkcji `rangeSum`. Zwróć uwagę, że w wywołaniu rekurencyjnym początkowy indeks elementu jest równy `start + 1`. Można to wyrazić w następujący sposób: „Zwróć wartość pierwszego elementu z określonego przedziału i dodaj do niego sumę pozostałych elementów z określonego przedziału”. Na listingu 13.4 zademonstrowałem, jak można wykorzystać tę funkcję.

#### **Listing 13.4**

```

1 Module main()
2 // Deklarujemy stałą równą rozmiarowi tablicy
3 Constant Integer SIZE = 9
4
5 // Deklarujemy tablicę liczb typu Integer
6 Declare Integer numbers[SIZE] = 1, 2, 3, 4, 5, 6, 7, 8, 9
7
8 // Deklarujemy zmienną, w której zapiszemy sumę
9 Declare Integer sum
10
11 // Obliczamy sumę elementów od 2. do 5.
12 Set sum = rangeSum(numbers, 2, 5)
13
14 // Wyświetlamy sumę
15 Display "Suma elementów od 2. do 5. wynosi ", sum
16 End Module
17
18 // Funkcja rangeSum zwraca sumę wartości elementów tablicy
19 // z określonego przedziału indeksów. Parametr start
20 // określa indeks pierwszego elementu, a parametr end określa
21 // indeks ostatniego elementu w przedziale
22 Function Integer rangeSum(Integer array[], Integer start,
23                         Integer end)
24     If start > end Then
25         Return 0
26     Else
27         Return array[start] + rangeSum(array, start + 1, end)
28     End If
29 End Function

```

#### **Wynik działania programu**

Suma elementów od 2. do 5. wynosi 18

## **Ciąg Fibonacciego**

Z pomocą rekurencji można rozwiązywać niektóre zadania matematyczne. Jednym z najpopularniejszych jest generowanie ciągu Fibonacciego. Ciąg Fibonacciego, nazwany od nazwiska włoskiego matematyka Leonardo Fibonacciego (ur. ok. 1170 r.), to następująca sekwencja liczb:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 ...

Zwróć uwagę, że począwszy do drugiej liczby, każda kolejna liczba jest równa sumie dwóch poprzednich liczb. Ciąg Fibonacciego można zdefiniować w następujący sposób:

```
jeśli n = 0, wtedy Fib(n) = 0
jeśli n = 1, wtedy Fib(n) = 1
jeśli n >= 2, wtedy Fib(n) = Fib(n - 1) + Fib (n - 2)
```

Oto funkcja rekurencyjna wyznaczająca n-tą liczbę ciągu Fibonacciego:

```
Function Integer fib(Integer n)
    If n == 0 then
        Return 0
    Else If n == 1 Then
        Return 1
    Else
        Return fib(n - 1) + fib(n - 2)
    End If
End Function
```

Zwróć uwagę, że mamy tutaj dwa przypadki bazowe: kiedy  $n$  jest równe 0 i kiedy  $n$  jest równe 1. W każdym z tych przypadków funkcja zwraca wartość i nie wywołuje rekurencyjnie samej siebie. Na listingu 13.5 przedstawiłem pseudokod, w którym za pomocą funkcji `fib` wyświetlam 10 pierwszych liczb ciągu Fibonacciego.

### **Listing 13.5**

```
1 Module main()
2 //Zmienna lokalna pełniąca funkcję licznika
3 Declare Integer counter
4
5 // Wyświetlamy komunikat początkowy
6 Display "10 pierwszych liczb ciągu Fibonacciego ",
7         "to:"
8
9 // Wywołujemy w pętli funkcję fib, przekazując do niej
10 // jako argumenty liczby od 1 do 10
11 For counter = 1 To 10
12     Display fib(counter)
13 End For
14 End Module
15
16 // Funkcja fib zwraca wartość n-tego
17 // elementu ciągu Fibonacciego
18 Function Integer fib(Integer n)
19     If n == 0 then
20         Return 0
21     Else If n == 1 Then
22         Return 1
23     Else
24         Return fib(n - 1) + fib(n - 2)
25     End If
26 End Function
```

### **Wynik działania programu**

10 pierwszych liczb ciągu Fibonacciego to:  
0 1 1 2 3 5 8 13 21 34

## Wyznaczanie największego wspólnego dzielnika

Kolejnym przykładem algorytmu rekurencyjnego jest wyznaczanie największego wspólnego dzielnika dwóch liczb. Największy wspólny dzielnik dwóch liczb dodatnich  $x$  i  $y$  wyznaczamy w następujący sposób:

jeśli  $x$  można podzielić bez reszty przez  $y$ , wtedy  $\text{gcd}(x, y) = y$   
w przeciwnym wypadku  $\text{gcd}(x, y) = \text{gcd}(y, \text{reszta z dzielenia } x/y)$

Definicja ta wskazuje, że gdy  $x$  dzieli się bez reszty przez  $y$ , wówczas największy wspólny dzielnik jest równy  $y$ . Jest to przypadek bazowy. W przeciwnym razie wynik jest równy największemu wspólnemu dzielnikowi liczb  $y$  i reszty z dzielenia  $x/y$ . Na listingu 13.6 przedstawiłem pseudokod programu, który wyznacza największy wspólny dzielnik dwóch liczb.

### Listing 13.6

```

1 Module main()
2   // Zmienne lokalne, w których zapiszemy liczby wprowadzone przez użytkownika
3   Declare Integer num1, num2
4
5   // Pobieramy liczbę od użytkownika
6   Display "Wprowadź liczbę całkowitą."
7   Input num1
8
9   // Pobieramy kolejną liczbę od użytkownika
10  Display "Wprowadź kolejną liczbę całkowitą."
11  Input num2
12
13  // Wyświetlamy największy wspólny dzielnik
14  Display "Największy wspólny dzielnik tych liczb"
15  Display "jest równy ", gcd(num1, num2)
16 End Module
17
18 // Funkcja gcd zwraca największy wspólny dzielnik
19 // liczb przekazanych jako argumenty x i y
20 Function Integer gcd(Integer x, Integer y)
21   // Sprawdzamy, czy x dzieli się bez reszty przez y
22   // Jeśli tak, dotarliśmy do przypadku bazowego
23   If x MOD y == 0 Then
24     Return y
25   Else
26     // Tutaj jest przypadek rekurencyjny
27     Return gcd(x, x MOD y)
28   End If
29 End Function

```

### Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)

Wprowadź liczbę całkowitą.

49 [Enter]

Wprowadź kolejną liczbę całkowitą.

28 [Enter]

Największy wspólny dzielnik tych liczb  
jest równy 7

## Rekurencyjny algorytm wyszukiwania binarnego

W rozdziale 9. pokazałem algorytm wyszukiwania binarnego i przykład programu, w którym do wyszukiwania wykorzystałem pętlę. Algorytm ten można także zaimplementować w sposób rekurencyjny. Przykładowo można go przedstawić następująco:

*Jeśli element array[middle] jest równy wyszukiwanej wartości, wtedy zadanie jest zakończone.*

*W przeciwnym razie, jeśli element array[middle] jest mniejszy niż szukana wartość, wtedy zastosuj algorytm wyszukiwania binarnego na górną połówce tablicy.*

*W przeciwnym razie, jeśli element array[middle] jest większy niż szukana wartość, wtedy zastosuj algorytm wyszukiwania binarnego na dolną połówce tablicy.*

Kiedy porównamy algorytm rekurencyjny z jego odpowiednikiem w postaci algorytmu wykorzystującego pętlę, staje się oczywiste, że wersja rekurencyjna jest bardziej przejrzysta i bardziej zrozumiała. Rekurencyjny algorytm wyszukiwania binarnego jest także dobrym przykładem tego, jak dzielimy zadanie na mniejsze części aż do momentu, gdy odnajdziemy rozwiązanie. Oto pseudokod rekurencyjnej funkcji binarySearch:

```

Function Integer binarySearch(Integer array[],  
                           Integer first, Integer last, Integer value)  
    // Zmienna lokalna, w której zapiszemy indeks środkowego elementu  
    // przeszukiwanego fragmentu tablicy  
    Declare Integer middle  
  
    // Najpierw sprawdzamy, czy pozostały jakieś elementy do przeszukiwania  
    If first > last Then  
        Return -1  
    End If  
  
    // Obliczamy indeks środkowego elementu przeszukiwanego fragmentu tablicy  
    Set middle = (first + last) / 2  
  
    // Sprawdzamy, czy szukana wartość znajduje się w środkowym elemencie  
    If array[middle] == value Then  
        Return middle  
    End If  
  
    // Przeszukujemy dolną albo górną połówkę fragmentu tablicy  
    If array[middle] < value Then  
        Return binarySearch(array, middle + 1, last, value)  
    Else  
        Return binarySearch(array, first, middle - 1, value)  
    End If  
End Function

```

Pierwszy parametr funkcji o nazwie `array` jest tablicą, którą mamy zamiar przeszukać. Kolejny parametr, o nazwie `first`, reprezentuje indeks pierwszego elementu fragmentu tablicy, który będziemy przeszukiwać. Następny parametr, o nazwie `last`, wskazuje indeks ostatniego elementu fragmentu tablicy, który będziemy przeszukiwać. Ostatni parametr, o nazwie `value`, reprezentuje szukaną wartość. Podobnie jak w przypadku funkcji `binarySearch` przedstawionej w rozdziale 9., funkcja zwraca indeks szukanego elementu lub wartość równą `-1`, jeśli szukanego elementu nie udało się odszukać w tablicy. Na listingu 13.7 zademonstrowałem tę funkcję.

**Listing 13.7**

```

1 Module main()
2   // Deklarujemy stałą równą rozmiarowi tablicy
3   Constant Integer SIZE = 20
4
5   // Deklarujemy tablicę zawierającą identyfikatory pracowników
6   Declare Integer numbers[SIZE] = 101, 142, 147, 189, 199,
7                               207, 222, 234, 289, 296,
8                               310, 319, 388, 394, 417,
9                               429, 447, 521, 536, 600
10
11  // Deklarujemy zmienną, w której zapiszemy identyfikator pracownika
12  Declare Integer empID
13
14  // Deklarujemy zmienną, w której zapiszemy wynik wyszukiwania
15  Declare Integer results
16
17  // Pobieramy identyfikator szukanego pracownika
18  Display "Wprowadź identyfikator pracownika."
19  Input empID
20
21  // Wyszukujemy identyfikator w tablicy
22  result = binarySearch(numbers, 0, SIZE - 1, empID)
23
24  // Wyświetlamy wynik wyszukiwania
25  If result == -1 Then
26    Display "Nie udało się wyszukać pracownika o takim identyfikatorze."
27  Else
28    Display "Znaleziono pracownika o takim identyfikatorze ",
29    "przy indeksie ", result
30  End If
31
32 End Module
33
34 // Funkcja binarySearch wykonuje rekurencyjne wyszukiwanie binarne
35 // we wskazanym fragmencie tablicy. Parametr array
36 // to przeszukiwana tablica. Parametr first
37 // to indeks pierwszego elementu przeszukiwanego fragmentu,
38 // a parametr last to indeks ostatniego elementu przeszukiwanego fragmentu.
39 // Parametr value to szukana wartość. Gdy wartość zostanie odnaleziona,
40 // funkcja zwraca indeks szukanego elementu. W przeciwnym razie
41 // funkcja zwraca wartość -1
42 Function Integer binarySearch(Integer array[],
43                           Integer first, Integer last, Integer value)
44   // Zmienna lokalna, w której zapiszemy indeks środkowego elementu
45   // przeszukiwanego fragmentu tablicy
46   Declare Integer middle
47
48   // Najpierw sprawdzamy, czy pozostały jakieś elementy do przeszukiwania
49   If first > last Then
50     Return -1
51   End If
52
53   // Obliczamy indeks środkowego elementu przeszukiwanego fragmentu tablicy
54   Set middle = (first + last) / 2
55
56   // Sprawdzamy, czy szukana wartość znajduje się w środkowym elemencie
57   If array[middle] == value Then
58     Return middle

```

```

59     End If
60
61 // Przeszukujemy dolną albo górną połówkę fragmentu tablicy
62 If array[middle] < value Then
63     Return binarySearch(array, middle + 1, last, value)
64 Else
65     Return binarySearch(array, first, middle - 1, value)
66 End If
67 End Function

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

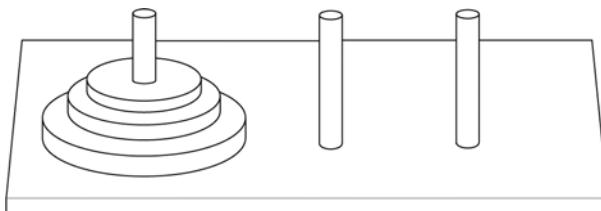
Wprowadź identyfikator pracownika.

**521 [Enter]**

Znaleziono pracownika o takim identyfikatorze przy indeksie 17

## Wieże Hanoi

Wieże Hanoi to gra matematyczna, która w podręcznikach do nauki programowania bardzo często służy jako przykład ilustrujący możliwości rekurencji. W grze występują trzy słupki i zestaw krążków z otworami pośrodku. Krążki ułożone są na jednym ze słupków (rysunek 13.5).



**Rysunek 13.5.** Słupki i krążki w grze wieże Hanoi

Zwróć uwagę, że krążki umieszczone są kolejno od największego do najmniejszego na lewym słupku. Gra opiera się na legendzie, która głosi, że grupa mnichów ze świątyni w Hanoi posiada podobny zestaw, ale składający się z 64 krążków. Zadaniem mnichów jest przemieszczenie krążków z pierwszego słupka na trzeci słupek. Na środkowym słupku można umieszczać krążki tymczasowo. Ponadto, przemieszczając krążki, mnisi muszą przestrzegać następujących zasad:

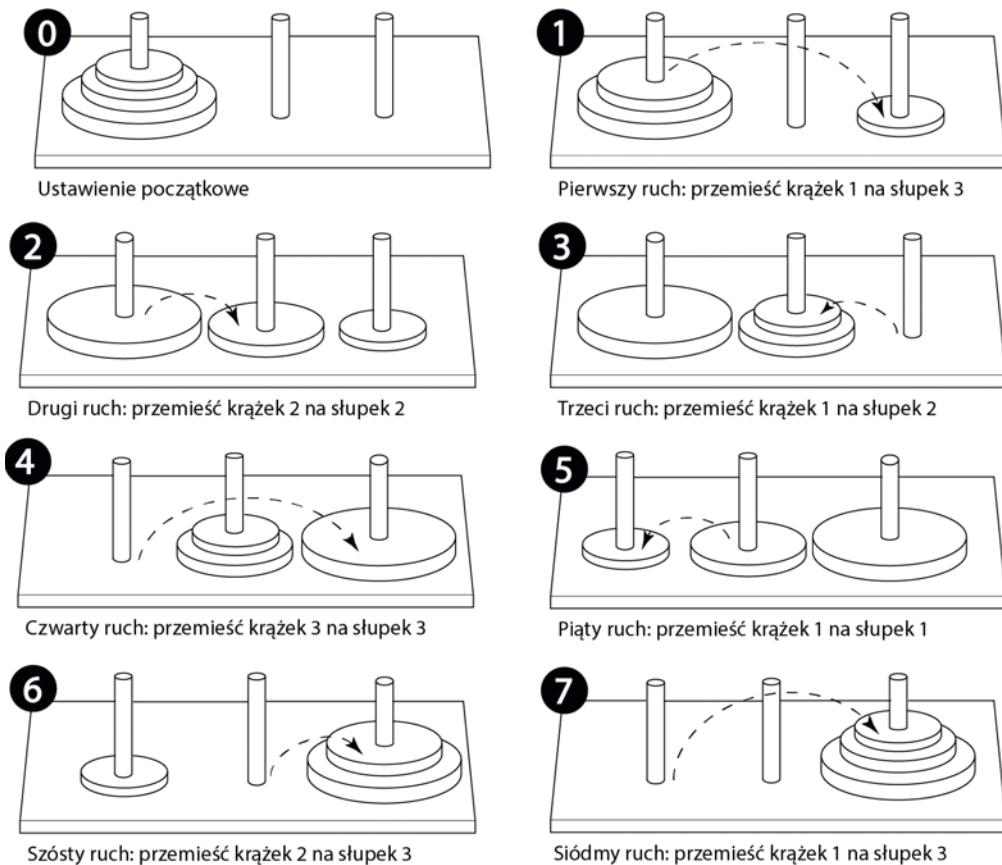
- w danym momencie można przemieścić tylko jeden krążek;
- większego krążka nie można umieścić na mniejszym krążku;
- wszystkie krążki muszą przez cały czas znajdować się na słupkach, z wyjątkiem momentu, gdy dany krążek jest przemieszczany.

Legenda głosi, że kiedy mnisi przemieszczą wszystkie krążki z pierwszego słupka na trzeci, nastąpi koniec świata.

Grając w tę grę, musisz trzymać się tych samych zasad co mnisi. Spójrzmy, jak będzie wyglądało rozwiązanie tego zadania, gdy w grze pojawi się różna liczba krążków. Kiedy krążek jest tylko jeden, rozwiązanie jest banalnie proste: wystarczy przemieścić krążek ze słupka pierwszego na trzeci. W przypadku dwóch krążków rozwiązanie będzie wymagało trzech ruchów:

- przemieścić krążek 1 na słupek 2;
- przemieścić krążek 2 na słupek 3;
- przemieścić krążek 1 na słupek 3.

Zauważ, że w tym rozwiążaniu słupek 2 pełni rolę tymczasowego miejsca, na które przemieszczamy krążek. Wraz ze wzrostem liczby krążków rośnie złożoność zadania. Aby przemieścić trzy krążki, będziemy potrzebować siedmiu ruchów — ilustruje to rysunek 13.6.



**Rysunek 13.6.** Poszczególne kroki przemieszczania trzech krążków

Ogólne rozwiążanie tego zadania można określić następującym stwierdzeniem:

*Przemieść n krążków ze słupka 1 na słupek 3 i wykorzystaj słupek 2 jako słupek tymczasowy.*

Oto algorytm rekurencyjny, za pomocą którego możemy zasymulować rozwiązanie zadania (zwróć uwagę, że w algorytmie używam zmiennych A, B i C, w których zapisuję numery słupków):

*Aby przemieścić n krążków ze słupka A na słupek C, używając tymczasowo słupka B, postępuj w sposób następujący:*

jeśli  $n > 0$ , wtedy

przenieść  $n - 1$  krążków ze słupka A na słupek B, używając tymczasowo słupka C;

przenieść pozostały krążek ze słupka A na słupek C;

przenieść  $n - 1$  krążków ze słupka B na słupek C, używając tymczasowo słupka A.

Koniec

Przypadkiem bazowym tego algorytmu jest sytuacja, gdy zabraknie krążków, które możemy przemieścić. Poniższy pseudokod przedstawia moduł, w którym zaimplementowałem ten algorytm. Zauważ, że moduł tak naprawdę niczego nie przemieszcza — zamiast tego wyświetla informacje dotyczące kolejnych ruchów.

```
Module moveDiscs(Integer num, Integer fromPeg,
                  Integer toPeg, Integer tempPeg)
    If num > 0 Then
        moveDiscs(num - 1, fromPeg, tempPeg, toPeg)
        Display "Przenieść krążek ze słupka ", fromPeg,
                " na słupek ", toPeg
        moveDiscs(num - 1, tempPeg, toPeg, fromPeg)
    End If
End Module
```

Moduł przyjmuje następujące parametry:

|         |                                            |
|---------|--------------------------------------------|
| num     | — liczba krążków, które należy przemieścić |
| fromPeg | — słupek, z którego należy zdjąć krążek    |
| toPeg   | — słupek, na którym należy umieścić krążek |
| tempPeg | — słupek tymczasowy                        |

Parametr num równy 0 oznacza, że nie ma już krążków, które możemy przemieścić. Pierwsze wywołanie rekurencyjne wygląda następująco:

```
moveDiscs(num - 1, fromPeg, tempPeg, toPeg)
```

Polecenie to wskazuje, że należy przemieścić wszystkie krążki z wyjątkiem jednego ze słupka fromPeg na słupek tempPeg, używając słupka toPeg jako słupka tymczasowego. Kolejne polecenie wygląda następująco:

```
Display "Przenieść krążek ze słupka ", fromPeg,
        " na słupek ", toPeg
```

Polecenie to wyświetla po prostu informację, że krążek należy przemieścić ze słupka fromPeg na słupek toPeg. Następnie pojawia się kolejne wywołanie rekurencyjne:

```
moveDiscs(num - 1, tempPeg, toPeg, fromPeg)
```

Polecenie to wskazuje, że należy przemieścić wszystkie krążki z wyjątkiem jednego ze słupka tempPeg na słupek toPeg, używając słupka fromPeg jako słupka tymczasowego. Na listingu 13.8 widoczny jest ten moduł i rozwiążanie gry wieże Hanoi.

### Listing 13.8



```
1 Module main()
2 //Stała równa liczbie krążków, które należy przemieścić
3 Constant Integer NUM_DISCS = 3
4
5 //Stała równa początkowemu słupkowi, z którego zdejmujemy krążek
6 Constant Integer FROM_PEG = 1
```

```

7 // Stała równa początkowemu słupkowi, na którym umieszczamy krążek
8 Constant Integer TO_PEG = 3
9
10 // Stała równa początkowemu słupkowi tymczasowemu
11 Constant Integer TEMP_PEG = 2
12
13
14 // Zaczynamy grę
15 Call moveDiscs(NUM_DISCS, FROM_PEG, TO_PEG, TEMP_PEG)
16 Display "Wszystkie krążki zostały już przemieszczone!"
17 End Module
18
19
20 // Funkcja moveDiscs wyświetla ruch
21 // w grze wieże Hanoi
22 // Jej parametry to:
23 // num — liczba krążków, które należy przemieścić
24 // fromPeg — słupek, z którego należy zdjąć krążek
25 // toPeg — słupek, na którym należy umieścić krążek
26 // tempPeg — słupek tymczasowy
27 Module moveDiscs(Integer num, Integer fromPeg,
28                     Integer toPeg, Integer tempPeg)
29     If num > 0 Then
30         moveDiscs(num - 1, fromPeg, tempPeg, toPeg)
31         Display "Przemieść krążek ze słupka ", fromPeg,
32                 " na słupek ", toPeg
33         moveDiscs(num - 1, tempPeg, toPeg, fromPeg)
34     End If
35 End Module

```

### Wynik działania programu

```

Przemieść krążek ze słupka 1 na słupek 3
Przemieść krążek ze słupka 1 na słupek 2
Przemieść krążek ze słupka 3 na słupek 2
Przemieść krążek ze słupka 1 na słupek 3
Przemieść krążek ze słupka 2 na słupek 1
Przemieść krążek ze słupka 2 na słupek 3
Przemieść krążek ze słupka 1 na słupek 3
Wszystkie krążki zostały już przemieszczone!

```

## Rekurencja a pętle

Każdy algorytm zapisany z wykorzystaniem rekurencji można także zapisać za pomocą pętli. Dzięki obu tym podejściom możemy wykonywać powtarzalne operacje, ale które z nich wybrać?

Jest kilka powodów przemawiających za tym, aby unikać rekurencji. Algorytmy rekurencyjne są zazwyczaj mniej wydajne od algorytmów iteracyjnych. Każde wywołanie modułu wiąże się z pewnym narzutem, co nie ma miejsca w przypadku pętli. Ponadto rozwiązania z wykorzystaniem pętli są znacznie czytelniejsze od rozwiązań rekurencyjnych. Tak naprawdę większość cyklicznych zadań programistycznych najlepiej jest rozwiązywać za pomocą pętli.

Jednak niektóre zadania łatwiej jest rozwiązać za pomocą rekurencji niż pętli. Przykładem niech będzie wzór matematyczny do obliczania największego wspólnego dzielnika, który idealnie wpasowuje się w naturę rekurencji. Prędkość współczesnych komputerów

i ilość ich pamięci powoduje, że argument dotyczący niższej wydajności algorytmów rekurencyjnych przestaje być tak znaczący. Obecnie wybór jednej czy drugiej techniki to głównie kwestia samego projektu — jeśli dane zadanie łatwiej będzie Ci rozwiązać za pomocą pętli, wykorzystaj ją, a jeśli zastosowanie rekurencji poprawi projekt program, zastosuj rekurencję.

## 13.4

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawcy: <ftp://ftp.helion.pl/przyklady/pkpro5.zip>.

### Java

W tym rozdziale nie będę omawiał żadnych nowych funkcji Javy, lecz jedynie technikę rekursywnego wywoływania metod. Zaprezentuję tu dwa programy omawiane wcześniej w tym rozdziale. Program na listingu 13.9 to napisana w Javie wersja programu w pseudokodzie z listingu 13.2.

**Listing 13.9** *(RecursionDemo.java)*

```

1 public class RecursionDemo
2 {
3     public static void main(String[] args)
4     {
5         // Przekazując do metody argument 5,
6         // informujemy go, że chcemy wyświetlić komunikat pięć razy
7         message(5);
8     }
9
10    public static void message(int n)
11    {
12        if (n > 0)
13        {
14            System.out.println("To jest metoda rekurencyjna.");
15            message(n - 1);
16        }
17    }
18 }
```

### Wynik działania programu

To jest metoda rekurencyjna.  
 To jest metoda rekurencyjna.  
 To jest metoda rekurencyjna.  
 To jest metoda rekurencyjna.  
 To jest metoda rekurencyjna.

Program z listingu 13.10 to napisana w Javie wersja programu w pseudokodzie z listingu 13.3. Program ten rekurencyjnie oblicza silnię liczbę.

### **Listing 13.10 (RecursiveFactorial.java)**

```

1 import java.util.Scanner;
2
3 public class RecursiveFactorial
4 {
5     public static void main(String[] args)
6     {
7         int number;      // Przechowuje liczbę
8
9         // Tworzymy obiekt Scanner do wprowadzania danych z klawiatury
10        Scanner keyboard = new Scanner(System.in);
11
12        // Pobieramy liczbę od użytkownika
13        System.out.print("Wprowadź nieujemną liczbę całkowitą. ");
14        number = keyboard.nextInt();
15
16        // Wyświetlamy silnię liczby
17        System.out.println("Silnia liczby " + number +
18                           " wynosi " + factorial(number));
19    }
20
21    // Metoda factorial używa rekursji do obliczenia
22    // silni argumentu, który z założenia ma być
23    // liczbą nieujemną
24
25    private static int factorial(int n)
26    {
27        if (n == 0)
28            return 1;  // Przypadek bazowy
29        else
30            return n * factorial(n - 1);
31    }
32 }
```

#### **Wynik działania programu**

Wprowadź nieujemną liczbę całkowitą. 7 [Enter]  
Silnia liczby 7 wynosi 5040

## **Python**

W tym rozdziale nie będę omawiał żadnych nowych funkcji Pythona, lecz jedynie technikę rekursywnego wywoływanego metod. Zaprezentuję tu dwa programy omawiane wcześniej w tym rozdziale. Program na listingu 13.11 to napisana w Pythonie wersja programu w pseudokodzie z listingu 13.2.

Program z listingu 13.12 to napisana w Pythonie wersja programu w pseudokodzie z listingu 13.3. Program ten rekurencyjnie oblicza silnię liczby.

**Listing 13.11 (recursion\_demo.py)**

```

1 # Ten program ma funkcję rekursywną
2
3 def main():
4     # Przekazując do funkcji message argument 5,
5     # informujemy go, że chcemy wyświetlić
6     # komunikat pięć razy
7     message(5)
8
9 def message(times):
10    if (times > 0):
11        print('To jest funkcja rekurencyjna.')
12        message(times - 1)
13
14 # Wywołujemy funkcję main
15 main()

```

**Wynik działania programu**

To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.

**Listing 13.12 (factorial.py)**

```

1 # Ten program używa rekursji do obliczenia
2 # silni podanej liczby
3
4 def main():
5     # Pobieramy liczbę od użytkownika
6     number = int(input('Wprowadź nieujemną liczbę całkowitą. '))
7
8     # Obliczamy silnię liczby
9     fact = factorial(number)
10
11    # Wyświetlamy silnię liczby
12    print('Silnia liczby', number, 'wynosi', fact)
13
14 # Funkcja factorial używa rekurencji do obliczenia
15 # silni argumentu, który jest uważany za nieujemny
16
17 def factorial(num):
18     if num == 0:
19         return 1
20     else:
21         return num * factorial(num - 1)
22
23 # Wywołujemy funkcję main
24 main()

```

**Wynik działania programu (pogrubioną czcionką zaznaczone dane wprowadzone przez użytkownika)**

Wprowadź nieujemną liczbę całkowitą. 7 [Enter]  
 Silnia liczby 7 wynosi 5040

## C++

W tym rozdziale nie będę omawiał żadnych nowych funkcji C++, lecz jedynie technikę rekursywnego wywoływania metod. Zaprezentuję tu dwa programy omawiane wcześniej w tym rozdziale. Program na listingu 13.13 to napisana w C++ wersja programu w pseudokodzie z listingu 13.2.

### **Listing 13.13 (RecursionDemo.cpp)**

```

1 #include <iostream>
2 using namespace std;
3
4 // Prototyp funkcji
5 void message(int);
6
7 int main()
8 {
9     // Przekazując do funkcji message argument 5,
10    // informujemy go, że chcemy wyświetlić komunikat pięć razy
11    message(5);
12
13    return 0;
14 }
15
16 void message(int n)
17 {
18     if (n > 0)
19     {
20         cout << "To jest funkcja rekurencyjna." << endl;
21         message(n - 1);
22     }
23 }
```

### **Wynik działania programu**

To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.  
 To jest funkcja rekurencyjna.

Program z listingu 13.14 to napisana w C++ wersja programu w pseudokodzie z listingu 13.3. Program ten rekurencyjnie oblicza silnię liczb.

### **Listing 13.14**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // Prototyp funkcji
6 int factorial(int);
7
8 int main()
9 {
10     int number;      // Przechowuje liczbę wprowadzoną przez użytkownika
```

```

11 int numFactorial; // Przechowuje silnię liczby
12
13 // Pobieramy liczbę od użytkownika
14 cout << "Wprowadź nieujemną liczbę całkowitą." << endl;
15 cin >> number;
16
17 // Obliczamy silnię liczby
18 numFactorial = factorial(number);
19
20 // Wyświetlamy silnię liczby
21 cout << "Silnia liczby " << number
22     << " wynosi " << numFactorial << endl;
23
24 return 0;
25 }
26
27 // Funkcja factorial używa rekursji do obliczenia silni argumentu,
28 // który jest uważany za liczbę nieujemną
29
30 int factorial(int n)
31 {
32     if (n == 0)
33         return 1; // Przypadek bazowy
34     else
35         return n * factorial(n - 1);
36 }
```

**Wynik działania programu**

Wprowadź nieujemną liczbę całkowitą.

7 [Enter]

Silnia liczby 7 wynosi 5040

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Moduł rekurencyjny \_\_\_\_\_.  
 a) wywołuje inny moduł  
 b) powoduje nieoczekiwane zamknięcie programu  
 c) wywołuje sam siebie  
 d) można wywołać tylko jednokrotnie
2. Pewien moduł wywoływany jest przez moduł `main`, a następnie pięć razy wywołuje sam siebie. Głębokość rekurencji wynosi w tym przypadku \_\_\_\_\_.  
 a) jeden  
 b) cztery  
 c) pięć  
 d) dziewięć
3. Część zadania, którą wykonujemy bez pomocy rekurencji, nazywamy przypadkiem \_\_\_\_\_.  
 a) bazowym  
 b) rozwiązywalnym

- c) znanym  
d) iteracyjnym
4. Część zadania, którą wykonujemy w sposób rekurencyjny, nazywamy przypadkiem \_\_\_\_\_.  
a) bazowym  
b) iteracyjnym  
c) nieznanym  
d) rekurencyjnym
5. Kiedy moduł w sposób jawnym wywołuje sam siebie, mamy do czynienia z rekurencją \_\_\_\_\_.  
a) jawną  
b) modalną  
c) bezpośrednią  
d) pośrednią
6. Kiedy moduł A wywołuje moduł B, który z kolei wywołuje moduł A, mamy do czynienia z rekurencją \_\_\_\_\_.  
a) niejawną  
b) modalną  
c) bezpośrednią  
d) pośrednią
7. Każde zadanie, które można rozwiązać za pomocą rekurencji, można także rozwiązać za pomocą \_\_\_\_\_.  
a) struktury warunkowej  
b) pętli  
c) struktury sekwencyjnej  
d) struktury decyzyjnej
8. Operacje mające miejsce podczas wywoływanego modułu, takie jak alokacja pamięci dla parametrów i zmiennych lokalnych, nazywamy \_\_\_\_\_.  
a) narzutem  
b) instalacją  
c) czyszczeniem  
d) synchronizacją
9. Przypadek rekurencyjny w algorytmie rekurencyjnym ma na celu \_\_\_\_\_.  
a) rozwiązać zadanie bez pomocy rekurencji  
b) ograniczyć problem do uproszczonej wersji zadania pierwotnego  
c) przechwycić błąd programu i zamknąć program  
d) rozszerzyć zadanie do bardziej skomplikowanej wersji zadania pierwotnego
10. Przypadek bazowy w algorytmie rekurencyjnym ma na celu \_\_\_\_\_.  
a) rozwiązać zadanie bez pomocy rekurencji  
b) ograniczyć problem do uproszczonej wersji zadania pierwotnego  
c) przechwycić błąd programu i zamknąć program  
d) rozszerzyć zadanie do bardziej skomplikowanej wersji zadania pierwotnego

**Prawda czy fałsz?**

- Algorytm wykorzystujący pętlę będzie zazwyczaj działał szybciej od odpowiadającego mu algorytmu rekurencyjnego.

2. Niektóre zadania programistyczne można rozwiązać tylko za pomocą rekurencji.
3. We wszystkich algorytmach rekurencyjnych należy określić przypadek bazowy.
4. W przypadku bazowym metoda rekurencyjna wywołuje sama siebie, rozwiązując uproszoną wersję zadania pierwotnego.

### Krótką odpowiedź

1. Jak wygląda przypadek bazowy w programie z listingu 13.2?
2. W niniejszym rozdziale wyjaśniłem, że aby obliczyć silnię liczb, należy kierować się następującymi zasadami:

jeśli  $n = 0$ , wtedy  $\text{factorial}(n) = 1$

jeśli  $n > 0$ , wtedy  $\text{factorial}(n) = n \cdot \text{factorial}(n - 1)$

Jak zdefiniowałbyś przypadek bazowy, gdybyś na podstawie tych zasad miał zaprojektować moduł? Jak zdefiniowałbyś przypadek rekurencyjny?

3. Czy do rozwiązania zadania zawsze trzeba korzystać z rekurencji? Z jakiej innej techniki można skorzystać, rozwiązując zadanie polegające na cyklicznym wykonywaniu pewnej operacji?
4. Dlaczego w przypadku rekurencji, aby wykonać uproszoną wersję zadania, moduł wywołuje sam siebie?
5. W jaki sposób ogranicza się zazwyczaj zadanie w module rekurencyjnym?

### Warsztat projektanta algorytmów

1. Co wyświetli się po uruchomieniu poniższego programu?

```
Module main()
    Declare Integer num = 0
    Call showMe(num)
End Module

Module showMe(Integer arg)
    If arg < 10 Then
        Call showMe(arg + 1)
    Else
        Display arg
    End If
End Module
```

2. Co wyświetli się po uruchomieniu poniższego programu?

```
Module main()
    Declare Integer num = 0
    Call showMe(num)
End Module

Module showMe(Integer arg)
    Display arg
    If arg < 10 Then
        Call showMe(arg + 1)
    End If
End Module
```

3. W poniższym module wykorzystano pętlę. Przepisz program tak, aby za pomocą rekurencji wykonywał tę samą operację.

```
Module trafficSign(int n)
    While n > 0
        Display "Zakaz parkowania"
        Set n = n - 1
    End While
End Module
```

## Ćwiczenia programistyczne

### 1. Mnożenie rekurencyjne

Zaprojektuj funkcję rekurencyjną, która przyjmuje dwa argumenty i przekazuje je do parametrów  $x$  i  $y$ . Funkcja powinna zwrócić wartość równą  $x$  razy  $y$ . Pamiętaj, że operację mnożenia można przedstawić jako określoną liczbę operacji dodawania:

$$7 \cdot 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4$$

Dla uproszczenia obliczeń załóż, że w zmiennych  $x$  i  $y$  będą przekazywane dodatnie liczby całkowite różne od zera.

### 2. Największy element

Zaprojektuj funkcję, która będzie przyjmowała jako argumenty tablicę liczb całkowitych `Integer` i rozmiar tej tablicy oraz zwracała wartość największego elementu tej tablicy. Aby odnaleźć element o największej wartości, skorzystaj z rekurencji.

### 3. Sumowanie elementów tablicy za pomocą rekurencji

Zaprojektuj funkcję, która będzie przyjmowała jako argumenty tablicę liczb całkowitych `Integer` i rozmiar tej tablicy. Funkcja powinna obliczyć za pomocą rekurencji sumę wszystkich elementów tablicy i zwrócić tę wartość.

### 4. Suma liczb

Zaprojektuj funkcję, która będzie przyjmowała jako argument liczbę całkowitą i zwracała sumę wszystkich liczb, począwszy od 1 aż do przekazanej przez argument liczby. Przykładowo po przekazaniu do funkcji liczby 50 funkcja powinna zwrócić sumę liczb 1, 2, 3, 4 ..., 50. Oblicz sumę za pomocą rekurencji.

### 5. Rekurencyjna funkcja podnosząca liczbę do potęgi

Zaprojektuj funkcję, która za pomocą algorytmu rekurencyjnego będzie podnosiła daną liczbę do potęgi. Funkcja powinna przyjmować dwa argumenty: liczbę, którą należy podnieść do potęgi, i wykładnik. Przyjmij założenie, że wykładnik jest liczbą nieujemną.

### 6. Funkcja Ackermann'a

Funkcja Ackermann'a to matematyczny algorytm rekurencyjny, który można wykorzystać, aby przetestować wydajność algorytmu rekurencyjnego na danym komputerze. Zaprojektuj funkcję `ackermann(m, n)` i zaimplementuj w niej funkcję Ackermann'a. Funkcja powinna działać w następujący sposób:

jeśli  $m = 0$ , wtedy zwróć  $n + 1$

jeśli  $n = 0$ , wtedy zwróć wartość  $\text{ackermann}(m - 1, 1)$

w przeciwnym przypadku zwróć wartość  $\text{ackermann}(m - 1, \text{ackermann}(m, n - 1))$

## TEMATYKA

- |                                                           |                                                          |
|-----------------------------------------------------------|----------------------------------------------------------|
| 14.1 Programowanie proceduralne i programowanie obiektowe | 14.4 Wyznaczanie klas i ich zakresu obowiązków w zadaniu |
| 14.2 Klasы                                                | 14.5 Dziedziczenie                                       |
| 14.3 Projektowanie klas za pomocą języka UML              | 14.6 Polimorfizm                                         |
|                                                           | 14.7 Rzut oka na języki Java, Python i C++               |

## 14.1

## Programowanie proceduralne i programowanie obiektowe

**WYJAŚNIENIE:** Programowanie proceduralne to jedna z metod tworzenia oprogramowania. Skupia się ona na procedurach lub operacjach, które mają miejsce w programie. Programowanie obiektowe koncentruje się na obiektach. Są one abstrakcyjnymi typami danych, w których znajdują się zarówno dane, jak i funkcje.

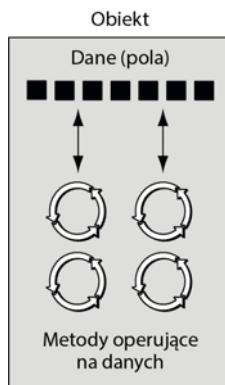
Obecnie korzysta się z dwóch głównych metod programowania: programowania proceduralnego i programowania obiektowego. Pierwsze języki programowania były językami proceduralnymi, co znaczy, że programy składały się z jednej lub większej liczby procedur. **Procedura** to po prostu moduł lub funkcja, które realizują określone zadanie, takie jak pobieranie danych od użytkownika, wykonywanie obliczeń, odczytywanie i zapisywanie danych do plików, wyświetlanie danych na ekranie itp. Programy, które dotychczas tworzyliśmy, były programami proceduralnymi.

Procedury operują zazwyczaj na danych zewnętrznych. W programowaniu proceduralnym dane są najczęściej przekazywane z procedury do procedury. Programowanie proceduralne skupia się więc na tworzeniu procedur operujących na pewnych danych.

Takie oddzielenie od siebie procedur i danych w przypadku bardziej skomplikowanych i rozbudowanych programów prowadzi do pewnych problemów.

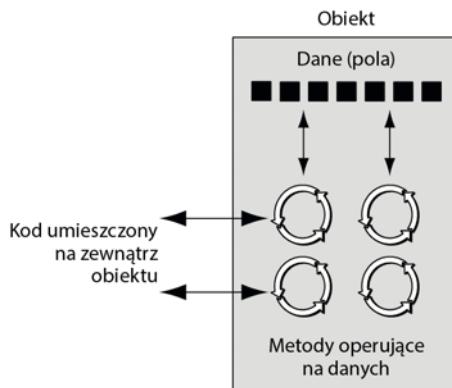
Załóżmy, że należysz do zespołu programistów, którego zadaniem jest stworzenie bardzo rozbudowanej aplikacji bazodanowej. Program został pierwotnie zaprojektowany tak, że nazwisko, adres i numer telefonu klienta były zapisywane w zmiennych typu *String*. Twoje zadanie polegało na zaprojektowaniu kilku procedur, które będą przekazywały te trzy zmienne jako argumenty i wykonywały na nich określone operacje. Tak zaprojektowany program był używany z powodzeniem przez pewien czas, ale Twój zespół otrzymał zadanie polegające na dodaniu do niego kilku nowych funkcji. Przyjrzał się temu zadaniu, starszy programista poinformował Cię, że od tej chwili nazwisko, adres i numer telefonu nie będą już zapisywane w zmiennych typu *String*, lecz w tablicy elementów typu *String*. Oznacza to, że musisz przeprojektować wszystkie stworzone do tej pory procedury tak, aby zamiast trzech zmiennych typu *String* przejmowały tablicę elementów typu *String*. Takie zadanie oznacza nie tylko bardzo dużo pracy, ale także możliwość popełnienia błędów w kodzie programu.

Podczas gdy programowanie proceduralne skupia się na tworzeniu procedur (zwanych także modułami lub funkcjami), **programowanie obiektowe** (ang. *object-oriented programming*) koncentruje się na tworzeniu obiektów. **Obiekt** to taki byt, który zawiera zarówno dane, jak i procedury. Dane zawarte w obiekcie nazywamy **polami**. Są to po prostu zmienne, tablice lub inne struktury przechowywane w obiekcie. Procedury, które może wykonywać obiekt, nazywamy **metodami**. Metoda to nic innego jak moduł lub funkcja. Obiekt jest więc samodzielną jednostką zawierającą dane (pola) i procedury (metody). Ilustruje to rysunek 14.1.



**Rysunek 14.1.** Obiekt składa się z danych i metod

Programowanie obiektowe rozwiązuje problem oddzielenia od siebie danych i kodu. Łączy się z nim także pojęcie hermetyzacji. **Hermetyzacja** polega na ukrywaniu danych przed kodem znajdującym się na zewnątrz obiektu. Do odczytywania i modyfikacji danych bezpośredni dostęp mają jedynie metody. Obiekt zazwyczaj **ukrywa dane**, ale umożliwia wywoływanie umieszczonej w nim metod. Na rysunku 14.2 przestawiłem, jak kod znajdujący się na zewnątrz modułu może odwoływać się dzięki metodom w sposób pośredni do danych.



**Rysunek 14.2.** Kod znajdujący się na zewnątrz obiektu wywołuje metody zawarte w obiekcie

Dane, do których mają dostęp tylko metody umieszczone wewnętrz obiektu, są ukryte przed zewnętrznym kodem i tym samym są chronione przed przypadkowym uszkodzeniem. Ponadto kod umieszczony na zewnątrz obiektu nie musi wiedzieć nic o tym, jaki format i strukturę mają dane umieszczone w obiekcie. Wystarczy, że kod będzie wywoływał udostępnione metody. Kiedy programista decyduje się zmodyfikować strukturę danych, modyfikuje również metody, tak aby działały one poprawnie ze zmodyfikowaną strukturą. Nie zmienia się jednak sposób, w jaki zewnętrzny kod odwołuje się do metod obiektu.

## Wielokrotne używanie obiektów

Poza tym, że programowanie obiektowe rozwiązuje problem separacji danych i kodu, sprzyja także koncepcji **wielokrotnego używania obiektów**. Sam obiekt nie jest samodzielnym programem, ale zaimplementowana w nim funkcjonalność może zostać wykorzystana w wielu programach. Przykładowo Karolina stworzyła obiekt, który służy do renderowania obrazów 3D. Karolina bardzo interesuje się matematyką i posiada szeroką wiedzę z dziedziny grafiki komputerowej, więc zaprojektowała obiekt tak, że wykonuje wszystkie potrzebne obliczenia w przestrzeni trójwymiarowej i współpracuje z kartą graficzną komputera. Tomek z kolei tworzy oprogramowanie dla pracowni architektonicznych i chce, aby jego program wyświetlał trójwymiarowe modele budynków. Ponieważ terminy gonią Tomka, a nie posiada on zbyt dużej wiedzy na temat grafiki komputerowej, będzie mógł wykorzystać stworzony przez Karolinę obiekt (oczywiście za niewielką opłatą!) i wyświetlać w swoim programie obiekty trójwymiarowe.

## Przykładowy obiekt z życia codziennego

Pomyśl przez chwilę o budziku jako obiekcie. Ma on następujące pola:

- bieżąca sekunda (wartość z zakresu 0 – 59);
- bieżąca minuta (wartość z zakresu 0 – 59);
- bieżąca godzina (wartość z zakresu 1 – 12);

- czas włączenia alarmu (godzina i minuta);
- stan budzika (włączony albo wyłączony).

Jak widzisz, pola to po prostu wartości, które opisują, w jakim **stanie** znajduje się budzik. Ty, jako użytkownik obiektu budzik, nie możesz bezpośrednio operować tymi polami, ponieważ są one **prywatne**. Aby zmienić wartość przypisaną do danego pola, musisz wywołać metodę obiektu. Oto kilka przykładowych metod, które udostępnia obiekt budzik:

- **Ustaw czas**
- **Ustaw czas budzenia**
- **Włącz budzik**
- **Wyłącz budzik**

Każda z tych metod modyfikuje co najmniej jedno pole. Przykładowo metoda **Ustaw czas** umożliwia ustawienie czasu. Można sobie wyobrazić, że metodę tę aktywujemy po naciśnięciu odpowiedniego przycisku na budziku. Naciskając inny przycisk, możemy aktywować metodę **Ustaw czas budzenia**.

Kolejny przycisk może służyć do aktywowania metod **Włącz budzik** i **Wyłącz budzik**. Zauważ, że wszystkie wymienione metody aktywuje osoba korzystająca z budzika. Metody, które wywoływane są na zewnątrz danego obiektu, nazywamy **metodami publicznymi**.

Budzik ma także **metody prywatne**, które można wywoływać tylko wewnątrz niego samego. Były zewnętrzne (czyli na przykład Ty jako użytkownik budzika) nie mogą wywoływać metod prywatnych. Obiekt jest zaprojektowany w taki sposób, że wywołuje te metody automatycznie, ukrywając przed Tobą szczegóły implementacji. Oto kilka metod prywatnych budzika:

- **Inkrementuj liczbę sekund**
- **Inkrementuj liczbę minut**
- **Inkrementuj liczbę godzin**
- **Aktywuj budzik**

Metoda **Inkrementuj liczbę sekund** jest wywoływana co sekundę. Jej zadanie polega na zwiększeniu o 1 bieżącej wartości pola, w którym zapisane są sekundy. Jeśli bieżąca wartość w tym polu jest równa 59, metoda ustawi w nim wartość 0, a następnie wywoła metodę **Inkrementuj liczbę minut**. Ta metoda z kolei zwiększa o 1 liczbę minut, chyba że wartość ta jest równa 59 — wtedy metoda wyzeruje liczbę minut i wywoła metodę **Inkrementuj liczbę godzin**. Metoda **Inkrementuj liczbę minut** sprawdza ponadto bieżący czas i porównuje go z czasem budzenia. Jeśli oba czasy są identyczne, włączany jest budzik — czyli wywoływana jest metoda **Aktywuj budzik**.



## Punkt kontrolny

- 14.1. Co to jest obiekt?
- 14.2. Co to jest hermetyzacja?

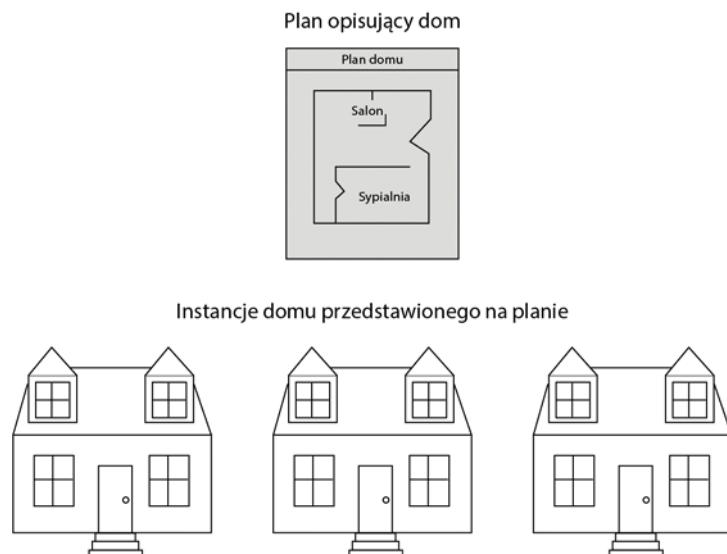
- 14.3. Dlaczego wewnętrzne dane obiektu są zazwyczaj niewidoczne na zewnątrz niego?
- 14.4. Co to są metody publiczne? Co to są metody prywatne?

## 14.2

# Klasy

**WYJAŚNIENIE:** Klasa to kod, za pomocą którego określamy pola i metody danego typu obiektu.

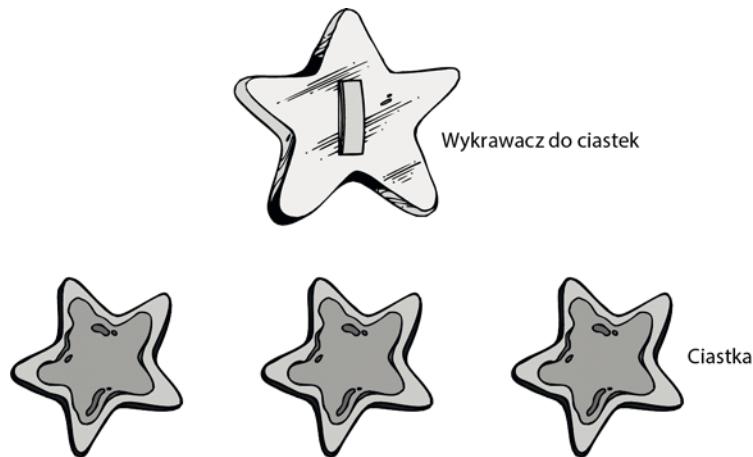
Omówię teraz, jak tworzy się obiekty w kodzie programu. Zanim stworzymy obiekt, należy go zaprojektować. Programista określa w tym celu wszystkie wymagane pola i metody, a następnie tworzy **klasę**. Klasa to kod, za pomocą którego określamy pola i metody danego typu obiektu. Wyobraź sobie klasę jako „plan”, na podstawie którego będzie tworzony dany obiekt. Ma on podobne przeznaczenie jak plan budowy domu. Kiedy na podstawie planu budujemy dom, możemy powiedzieć, że budujemy instancję budynku określonego na planie. Każdy z wybudowanych domów jest oddzielną instancją domu przedstawionego na planie. Ilustruje to rysunek 14.3.



Rysunek 14.3. Plan i domy wybudowane zgodnie z tym planem

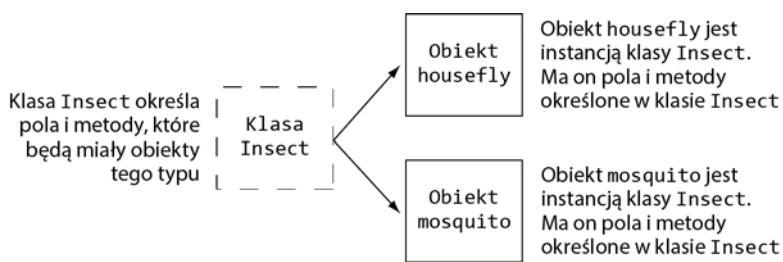
Inną analogią pozwalającą zrozumieć różnicę między klasą a obiektem jest wykrawacz do ciastek i ciastko. Sam wykrawacz nie jest ciastkiem, ale określa kształt ciastka. Jak pokazałem na rysunku 14.4, można go użyć, aby stworzyć wiele ciastek. Klasa to wykrawacz do ciastek, a obiekty utworzone na podstawie tej klasy to ciastka.

Klasa nie jest więc obiektem, lecz stanowi jego opis. Po uruchomieniu programu możemy za pomocą klasy tworzyć w pamięci komputera dowolną liczbą obiektów określonego typu. Każdy z tych obiektów nazywamy **instancją** klasy.



**Rysunek 14.4.** Analogia z wykrawaczem do ciastek

Przykładowo Kasia jest entomologiem (badaczem owadów), ale lubi także tworzyć programy komputerowe. Postanowiła zaprojektować program, za pomocą którego będzie mogła katalogować różne gatunki owadów. Fragmentem programu jest klasa o nazwie `Insect`, w której znajdują się pola i metody wspólne dla wielu gatunków owadów. Klasa `Insect` nie jest obiektem, ale stanowi opis tego, jak będą wyglądać obiekty utworzone na jej podstawie. Kasia następnie zapisuje polecenia, za pomocą których utworzy obiekt o nazwie `housefly`, będący instancją klasy `Insect`. Obiekt ten zostanie umieszczony w pamięci komputera i będą w nim zapisane informacje na temat muchy domowej. Będzie miał pola i metody określone w klasie `Insect`. W kolejnej kolejności Kasia zapisuje polecenia, za pomocą których utworzy obiekt o nazwie `mosquito`, także będący instancją klasy `Insect`. Zajmie on osobne miejsce w pamięci komputera, a zapisane w nim będą informacje dotyczące komara. Pomimo że obiekty `housefly` i `mosquito` zajmują osobne miejsca w pamięci komputera, oba zostały utworzone na podstawie klasy `Insect`. Oznacza to, że każdy z obiektów ma pola i metody zdefiniowane w klasie `Insect`. Ilustruje to rysunek 14.5.



**Rysunek 14.5.** Obiekty `housefly` i `mosquito` to instancje klasy `Insect`

## Tworzenie klasy krok po kroku

W pseudokodzie do tworzenia klasy będziemy używać następującej definicji:

```
Class NazwaKlasy
    deklaracje pól i definicje metod...
End Class
```

Pierwsza linia rozpoczyna się od słowa `Class`, a po nim następuje nazwa klasy. W większości języków programowania zasady nazewnictwa klas są identyczne jak zasady nazewnictwa zmiennych. Następnie deklarujemy pola klasy i definiujemy metody klasy. Pola i metody należące do danej klasy nazywamy **składowymi klasy**. Definicję klasy kończą słowa `End Class`.

Teraz zademonstruję, jak tworzy się zazwyczaj klasę w języku obiektowym. Ponieważ klasa składa się z kilku elementów, nie przedstawię od razu całej jej definicji — będę ją tworzył stopniowo, krok po kroku.

Załóżmy, że tworzymy oprogramowanie dla firmy Wireless Solutions, która zajmuje się sprzedażą telefonów komórkowych i urządzeń bezprzewodowych. Program będzie służył do zarządzania asortymentem telefonów znajdujących się w ofercie firmy. Oto informacje, których będziemy potrzebować:

- nazwa producenta telefonu;
- model telefonu;
- cena detaliczna telefonu.

Gdybyśmy tworzyli program proceduralny, zapisalibyśmy te informacje po prostu w zmiennych. W tym przypadku projektujemy program obiektowy i do opisania telefonu stworzymy klasę. Klasa będzie miała pola, w których będziemy zapisywać potrzebne informacje. Na listingu klasy 14.1 przedstawiłem, od czego zaczniemy definiowanie klasy.

### **Listing klasy 14.1**

```
1 Class CellPhone
2     //Deklarujemy pola
3     Private String manufacturer
4     Private String modelNumber
5     Private Real retailPrice
6
7     // To jeszcze nie jest koniec klasy!
8 End Class
```

Zwróć uwagę, że w linii 1. nadałem klasie nazwę `CellPhone`. W tej książce nazwy klas zawsze będę rozpoczynał od dużej litery. Nie jest to konieczne, ale wielu programistów stosuje właśnie taki zapis, aby odróżnić nazwy klas od nazw zmiennych.

W liniach 3., 4. i 5. deklaruję trzy pola. W linii 3. deklaruję pole typu `String` o nazwie `manufacturer`, w linii 4. deklaruję pole typu `String` o nazwie `modelNumber`, a w linii 5. deklaruję pole typu `Real` o nazwie `retailPrice`. Zwróć uwagę, że każda deklaracja rozpoczyna się od słowa `Private`. Kiedy przed deklaracją pola użyjemy słowa `Private`, oznacza to, że do danego pola nie będzie się można odwołać bezpośrednio w kodzie

umieszczonym poza obiektem. W większości języków programowania słowo **Private** jest nazywane **modyfikatorem dostępu**. Określa ono, w jaki sposób będzie można się odwoływać do danego pola lub metody.

Dzięki modyfikatorowi dostępu **Private** klasa może ukryć dane przed kodem umieszczonym na zewnątrz obiektu. W ten sposób dane są chronione przed przypadkowym uszkodzeniem. Tworzenie pól jako prywatnych i odwoływanie się do nich za pomocą metod to bardzo popularna technika, wykorzystywana w wielu językach obiektowych. Następnie dodamy do klasy następujące metody, dzięki którym kod umieszczony na zewnątrz klasy będzie mógł uzyskać dostęp do pól:

- **setManufacturer** — moduł, który będzie zapisywał wartość w polu **manufacturer**;
- **setModelNumber** — moduł, który będzie zapisywał wartość w polu **modelNumber**;
- **setRetailPrice** — moduł, który będzie zapisywał wartość w polu **retailPrice**.

Na listingu klasy 14.2 przedstawiłem, jak będzie wyglądała klasa **CellPhone** po dodaniu do niej metod.

### **Listing klasy 14.2**

```

1 Class CellPhone
2 //Deklarujemy pola
3 Private String manufacturer
4 Private String modelNumber
5 Private Real retailPrice
6
7 //Definiujemy metody
8 Public Module setManufacturer (String manufact)
9     Set manufacturer = manufact
10 End Module
11
12 Public Module setModelNumber (String modNum)
13     Set modelNumber = modNum
14 End Module
15
16 Public Module setRetailPrice (Real retail)
17     Set retailPrice = retail
18 End Module
19
20 //To jeszcze nie jest koniec klasy!
21 End Class

```

W linach od 8. do 10. pojawia się metoda **setManufacturer**. Wygląda ona jak zwykła definicja modułu z tą różnicą, że w jej nagłówku występuje słowo **Public**. W większości języków obiektowych jest ono modyfikatorem dostępu. W przypadku metody słowo **Public** wskazuje, że daną metodę będzie można wywołać na zewnątrz klasy.

Metoda **setManufacturer** ma parametr typu **String** o nazwie **manufact**. Podczas wywołania metody należy do niej przekazać jako argument łańcuch znakowy. W linii 9. do pola **manufacturer** przypisuję wartość przekazaną przez parametr **manufact**.

W liniach od 12. do 14. pojawia się metoda `setModelNumber`, która ma parametr typu `String` o nazwie `modNum`. Wywołując tę metodę, należy do niej przekazać jako argument łańcuch znakowy. W linii 13. wartość przekazaną przez parametr `modNum` przypisuję do pola `modelNumber`.

Metoda `setRetailPrice` ma parametr typu `Real` o nazwie `retail`. Wywołując tę metodę, należy do niej przekazać jako argument liczbę rzeczywistą. W linii 17. wartość przekazaną przez parametr `retail` zapisuję w polu `retailPrice`.

Ponieważ pola `manufacturer`, `modelNumber` i `retailPrice` są prywatne, musiałem zdefiniować metody `setManufacturer`, `setModelNumber` i `setRetailPrice`, dzięki którym kod umieszczony na zewnątrz klasy `CellPhone` będzie mógł zapisać w polach wartości. Aby kod na zewnątrz klasy mógł pobrać wartości zapisane w polach, musimy zdefiniować kolejne metody. Utworzę więc w tym celu metody o nazwach `getManufacturer`, `getModelNumber` i `getRetailPrice`. Metoda `getManufacturer` będzie zwracała wartość zapisaną w polu `manufacturer`, metoda `getModelNumber` będzie zwracała wartość zapisaną w polu `modelNumber`, a metoda `getRetailPrice` będzie zwracała wartość zapisaną w polu `retailPrice`.

Na listingu klasy 14.3 przedstawiłem, jak będzie wyglądała klasa `CellPhone` po dodaniu do niej tych metod. Nowe metody znajdują się w liniach od 20. do 30.

### **Listing klasy 14.3**



```

1 Class CellPhone
2 //Deklarujemy pola
3 Private String manufacturer
4 Private String modelNumber
5 Private Real retailPrice
6
7 //Definiujemy metody
8 Public Module setManufacturer(String manufact)
9     Set manufacturer = manufact
10 End Module
11
12 Public Module setModelNumber(String modNum)
13     Set modelNumber = modNum
14 End Module
15
16 Public Module setRetailPrice(Real retail)
17     Set retailPrice = retail
18 End Module
19
20 Public Function String getManufacturer()
21     Return manufacturer
22 End Function
23
24 Public Function String getModelNumber()
25     Return modelNumber
26 End Function
27
28 Public Function Real getRetailPrice()
29     Return retailPrice
30 End Function
31 End Class

```

Metoda `getManufacturer` pojawia się w liniach od 20. do 22. Zwróć uwagę, że metoda ta ma postać funkcji, a nie modułu. Po wywołaniu polecenie w linii 21. zwróci wartość zapisaną w polu `manufacturer`.

W liniach od 24. do 26. pojawia się metoda `getModelNumber`, a w liniach od 28. do 30. — metoda `getRetailPrice`. Obie są także funkcjami. Metoda `getModelNumber` zwraca wartość zapisaną w polu `modelNumber`, a metoda `getRetailPrice` zwraca wartość zapisaną w polu `retailPrice`.

Na listingu klasy 14.3 przedstawiłem cały kod klasy, ale nie stanowi ona kompletnego programu. Jest po prostu planem, według którego będziemy tworzyć obiekty. Abyśmy mogli użyć klasy, musimy napisać program, w którym utworzymy na jej podstawie obiekt. Przedstawiłem go na listingu 14.1.

### **Listing 14.1**



```

1 Module main()
2 //Deklarujemy zmienną, za pomocą której
3 //będziemy się mogli odwoływać do obiektu klasy CellPhone
4 Declare CellPhone myPhone
5
6 //Za pomocą poniższego polecenia tworzymy obiekt,
7 //a jako plan posłuży nam klasa CellPhone
8 //Zmienna myPhone odwołuje się do obiektu
9 Set myPhone = New CellPhone()
10
11 //W polach obiektu zapisujemy wartości
12 Call myPhone.setManufacturer("Motorola")
13 Call myPhone.setModelNumber("MOTO G5")
14 Call myPhone.setRetailPrice(599.99)
15
16 //Wyświetlamy wartości zapisane w polach
17 Display "Producent telefonu: ", myPhone.getManufacturer()
18 Display "Model telefonu: ", myPhone.getModelNumber()
19 Display "Cena detaliczna telefonu: ", myPhone.getRetailPrice(), " zł"
20 End Module

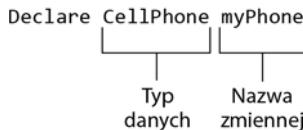
```

#### **Wynik działania programu**

Producent telefonu: Motorola  
Model telefonu: MOTO G5  
Cena detaliczna telefonu: 599.99 zł

Polecenie w linii 4. to deklaracja zmiennej. Deklaruję za jego pomocą zmienną o nazwie `myPhone`. Polecenie to wygląda identycznie jak każda inna deklaracja zmiennej z tą różnicą, że typem obiektu jest klasa `CellPhone`. Ilustruje to rysunek 14.6. Kiedy deklarujemy zmienną, a jako jej typ wskazujemy nazwę klasy, tworzymy tym samym specjalną **zmienną obiektową**, która będzie się odnosiła do obiektu umieszczonego w pamięci komputera. Zmienna `myPhone`, którą zadeklarowałem w linii 4., będzie się odnosiła do obiektu utworzonego na podstawie klasy `CellPhone`.

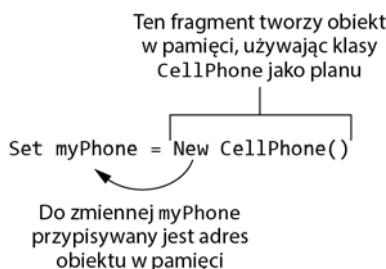
W wielu językach obiektowych sam fakt zadeklarowania zmiennej obiektowej nie tworzy jeszcze obiektu w pamięci komputera. Tworzona jest jedynie zmienna, za pomocą której będzie się można odwoływać do danego typu obiektu. Następnym krokiem jest utworzenie samego obiektu. Robimy to za pomocą operacji przypisania widocznej w linii 9.:



Rysunek 14.6. Deklaracja zmiennej obiektowej

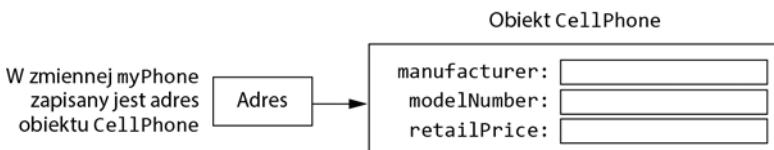
```
Set myPhone = New CellPhone()
```

Zwróć uwagę, że po prawej stronie operatora = znajduje się słowo New. W wielu językach programowania tworzy ono obiekt w pamięci komputera. Po słowie New pojawia się nazwa klasy (w tym przypadku CellPhone), a następnie para nawiasów. W ten sposób określamy klasę, która będzie służyła jako plan budowy danego obiektu. Po utworzeniu obiektu do zmiennej myPhone przypisujemy za pomocą operatora = jego adres w pamięci komputera. Na rysunku 14.7 przedstawiłem operacje, które wykonuje to polecenie.



Rysunek 14.7. Tworzenie obiektu i przypisywanie jego adresu do zmiennej obiektowej

Kiedy do zmiennej obiektowej zostanie przypisany adres obiektu, mówimy, że zmienna ta odnosi się (stanowi referencję) do obiektu. Jak widać na rysunku 14.8, po wykonaniu instrukcji zmienna myPhone będzie odnosiła się do obiektu typu CellPhone.



Rysunek 14.8. Zmienna myPhone odnosi się do obiektu CellPhone



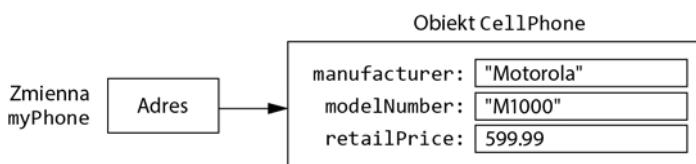
**UWAGA:** W niektórych językach (na przykład w C++) zadeklarowanie zmiennej obiektowej tworzy także obiekt w pamięci komputera. W takich językach nie trzeba używać słowa New, jak zrobiliśmy to w linii 9. na listingu 14.1.

W linii 12. pojawia się następujące polecenie:

```
Call myPhone.setManufacturer("Motorola")
```

Za pomocą tego polecenia wywołuję metodę `myPhone.setManufacturer`. Stosuję tutaj **operator kropki**. Po lewej stronie operatora kropki znajduje się nazwa zmiennej obiektowej, a po jego prawej stronie znajduje się nazwa metody, którą mamy zamiar wywołać. W momencie uruchomienia tego polecenia wywołana zostanie metoda `setManufacturer` obiektu, do którego odnosi się zmienna `myPhone`, a do metody zostanie przekazany łańcuch znakowy "Motorola". W wyniku tej operacji do pola `manufacturer` zostanie przypisany łańcuch znakowy "Motorola".

W linii 13. wywołuję metodę `myPhone.setModelNumber` i przekazuję do niej jako argument łańcuch znakowy "MOTO G5". Po wykonaniu tego polecenia w polu `modelNumber` obiektu `myPhone` zostanie zapisany łańcuch znakowy "MOTO G5". W linii 14. wywołuję metodę `myPhone.setRetailPrice` i przekazuję do niej jako argument wartość 599.99. W wyniku tej operacji do pola `retailPrice` zostanie przypisana wartość 599.99. Na rysunku 14.9 przedstawiłem stan obiektu po wywołaniu poleceń w liniach od 12. do 14.



Rysunek 14.9. Stan obiektu, do którego odnosi się zmienna `myPhone`

Za pomocą poleceń w liniach od 17. do 19. wyświetlам wartości przypisane do pól. Oto polecenie w linii 17.:

```
Display "Producent telefonu: ", myPhone.getManufacturer()
```

Za pomocą tego polecenia wywołuję metodę `myPhone.getManufacturer`, która zwraca łańcuch znakowy "Motorola". Na ekranie wyświetli się więc następujący tekst:

```
Producent telefonu: Motorola
```

Następnie wykona się polecenie w linii 18.:

```
Display "Model telefonu: ", myPhone.getModelNumber()
```

Za pomocą tego polecenia wywołuję metodę `myPhone.getModelNumber`, która zwraca łańcuch znakowy "MOTO G5". Na ekranie wyświetli się więc następujący tekst:

```
Model telefonu: MOTO G5
```

Następnie wykona się polecenie w linii 19.:

```
Display "Cena detaliczna telefonu: ", myPhone.getRetailPrice(), " zł"
```

Za pomocą tego polecenia wywołuję metodę `myPhone.getRetailPrice`, która zwraca liczbę 599.99. Na ekranie wyświetli się więc następujący tekst:

```
Cena detaliczna telefonu: 599.99 zł
```

## Akcesory i mutatory

Jak już wspomniałem, bardzo często w klasie tworzy się prywatne pola, do których można się odwoływać za pomocą publicznych metod. Dzięki takiemu rozwiązaniu obiekt panuje nad tym, jakie wartości będą przypisywane do pól. Metodę, za pomocą której odczytywana jest wartość z pola, nazywa się **akcesorem**. Metoda, za pomocą której przypisuje się wartość do pola, nazywa się **mutatorem**. W przypadku klasy `CellPhone` akcesorami są metody `getManufacturer`, `getModelNumber` i `getRetailPrice`, a mutatorami są metody `setManufacturer`, `setModelNumber` i `setRetailPrice`.



**UWAGA:** Mutatory nazywa się w żargonie setterami, a akcesory — getterami.

## Konstruktory

Konstruktor to metoda, która jest wywoływana automatycznie w momencie utworzenia obiektu. Konstruktorów używa się przeważnie, aby zainicjalizować pola wartościami początkowymi. Metody te nazywają się konstruktorami, ponieważ pomagają w budowaniu obiektu.

W wielu językach programowania konstruktor ma taką samą nazwę jak klasa, w której jest umieszczony. Takiej konwencji będę się trzymał w tej książce. Przykładowo jeśli będziemy chcieli utworzyć konstruktor w klasie `CellPhone`, utworzymy moduł o nazwie `CellPhone`. Na listingu klasy 14.4 przedstawiłem definicję klasy po dodaniu do niej konstruktora. Konstruktor pojawia się w liniach od 8. do 13.



**UWAGA:** W Visual Basicu konstruktory nazywają się `New`.

### Listing klasy 14.4



```

1 Class CellPhone
2 //Deklarujemy pola
3 Private String manufacturer
4 Private String modelNumber
5 Private Real retailPrice
6
7 //Konstruktor
8 Public Module CellPhone(String manufact,
9                         String modNum, Real retail)
10    Set manufacturer = manufact
11    Set modelNumber = modNum
12    Set retailPrice = retail
13 End Module
14
15 //Mutatory
16 Public Module setManufacturer(String manufact)
17     Set manufacturer = manufact
18 End Module
19

```

```

20   Public Module setModelNumber(String modNum)
21     Set modelNumber = modNum
22   End Module
23
24   Public Module setRetailPrice(String retail)
25     Set retailPrice = retail
26   End Module
27
28 // Akcesory
29   Public Function String getManufacturer()
30     Return manufacturer
31   End Function
32
33   Public Function String getModelNumber()
34     Return modelNumber
35   End Function
36
37   Public Function Real getRetailPrice()
38     Return retailPrice
39   End Function
40 End Class

```

Konstruktor przyjmuje trzy argumenty przekazywane do parametrów `manufact`, `modNum` i `retail`. W liniach od 10. do 12. wartości parametrów przypisuję do pól `manufacturer`, `modelNumber` i `retailPrice`.

Na listingu 14.2 tworzę obiekt typu `CellPhone` i za pomocą konstruktora inicjalizuję jego pola. Zwróć uwagę, że w linii 9. po nazwie klasy umieściłem w nawiasach wartości "Motorola", "MOTO G5" i 599.99. Wartości te zostaną przekazane w konstruktorze do parametrów `manufact`, `modNum` i `retail`. W kodzie konstruktora przypisuję je do pól `manufacturer`, `modelNumber` i `retailPrice`.

### **Listing 14.2**



```

1 Module main()
2   // Deklarujemy zmienną, za pomocą której
3   // będziemy się mogli odwoływać do obiektu klasy CellPhone
4   Declare CellPhone myPhone
5
6   // Za pomocą poniższego polecenia tworzymy obiekt typu CellPhone
7   // i inicjalizujemy jego pola wartościami
8   // przekazanymi do konstruktora
9   Set myPhone = New CellPhone("Motorola", "MOTO G5", 599.99)
10
11  // Wyświetlamy wartości zapisane w polach
12  Display "Producent telefonu: ", myPhone.getManufacturer()
13  Display "Model telefonu: ", myPhone.getModelNumber()
14  Display "Cena detaliczna telefonu: ", myPhone.getRetailPrice(), " zł"
15 End Module

```

### **Wynik działania programu**

Producent telefonu: Motorola  
 Model telefonu: MOTO G5  
 Cena detaliczna telefonu: 599.99 zł

Na listingu 14.3 przedstawiłem kolejny przykład wykorzystania klasy `CellPhone`. Program prosi użytkownika o wprowadzenie informacji dotyczących telefonu, a następnie tworzy obiekt, w którym zapisuje wprowadzone dane.

### **Listing 14.3**

```

1 Module main()
2   //Zmienne, w których zapiszemy dane wprowadzone przez użytkownika
3   Declare String manufacturer, model
4   Declare Real retail
5
6   //Deklarujemy zmienną, za pomocą której
7   //będziemy się mogli odwoływać do obiektu klasy CellPhone
8   Declare CellPhone phone
9
10  //Pobieramy od użytkownika dane dotyczące telefonu
11  Display "Wprowadź producenta telefonu."
12  Input manufacturer
13  Display "Wprowadź model telefonu."
14  Input model
15  Display "Wprowadź cenę detaliczną telefonu."
16  Input retail
17
18  //Tworzymy obiekt typu CellPhone i inicjalizujemy go
19  //danymi wprowadzonymi przez użytkownika
20  Set phone = New CellPhone(manufacturer, model, retail)
21
22  //Wyświetlamy wartości zapisane w polach
23  Display "Oto wprowadzone przez Ciebie wartości:"
24  Display "Producent telefonu: ", myPhone.getManufacturer()
25  Display "Model telefonu: ", myPhone.getModelNumber()
26  Display "Cena detaliczna telefonu: ", myPhone.getRetailPrice(), " zł"
27 End Module

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź producenta telefonu.

**Samsung [Enter]**

Wprowadź model telefonu.

**GALAXY J5 [Enter]**

Wprowadź cenę detaliczną telefonu.

**479.99 [Enter]**

Oto wprowadzone przez Ciebie wartości:

Producent telefonu: Samsung

Model telefonu: GALAXY J5

Cena detaliczna telefonu: 479.99 zł

## **Konstruktory domyślne**

W większości języków programowania w momencie tworzenia obiektu wywoływany jest jego konstruktor. Ale co się stanie, jeśli w danej klasie nie zadeklarujemy konstruktora? W takim przypadku większość języków programowania utworzy go automatycznie podczas kompilowania programu. Tak wygenerowany konstruktor nazywamy **konstruktorem domyślnym**. To, jakie operacje przeprowadza konstruktor domyślny, zależy od języka. Zazwyczaj konstruktor domyślny przypisuje wartości początkowe do pól obiektu.



## Punkt kontrolny

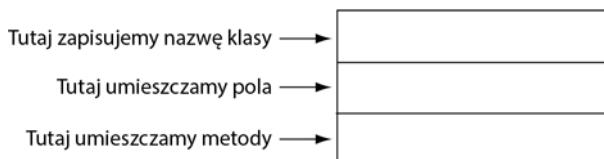
- 14.5. Słyszysz, jak ktoś mówi: „Plan to projekt domu. Cieśla może wykorzystać plan do budowy domu. Jeśli chce, cieśla może wybudować na podstawie planu wiele takich samych domów”. Pomyśl o tym w kontekście klas i obiektów. Czy plan reprezentuje klasę czy obiekt?
- 14.6. Aby wyjaśnić różnicę między klasą i obiektem, posłużyłem się w tym rozdziale analogią z wykrawaczem do ciastek i ciastkami, które powstają po użyciu wykrawacza. Co jest w tym przypadku odpowiednikiem wykrawacza, a co jest odpowiednikiem ciastka?
- 14.7. Co to modyfikator dostępu?
- 14.8. Którego modyfikatora dostępu używa się zazwyczaj w stosunku do pól klasy?
- 14.9. Gdy zmienna obiektowa odnosi się do obiektu, to co tak naprawdę jest w niej zapisane?
- 14.10. Jak działa słowo kluczowe New?
- 14.11. Co to jest akcesor? Co to jest mutator?
- 14.12. Co to jest konstruktor?
- 14.13. Co to jest konstruktor domyślny?

14.3

## Projektowanie klas za pomocą języka UML

**WYJAŚNIENIE:** Język UML (Unified Modeling Language) jest ustandaryzowanym sposobem opisu systemów obiektowych za pomocą diagramów.

Podczas projektowania klasy pomocne okazują się diagramy UML. UML to zunifikowany język modelowania. Zapewnia on szereg diagramów, za pomocą których można graficznie opisać system obiektowy. Na rysunku 14.10 przedstawiłem ogólną postać diagramu klasy w języku UML. Zwróć uwagę, że prostokąt podzielony jest na trzy części. W górnej części zapisujemy nazwę klasy, w środkowej umieszczałyśmy listę pól danej klasy, a w dolnej listę metod danej klasy.



**Rysunek 14.10.** Ogólna postać diagramu klasy w języku UML

Zgodnie z tymi wytycznymi na rysunku 14.11 przestawiłem uproszczony diagram UML klasy CellPhone.

| CellPhone                                                                                                                             |
|---------------------------------------------------------------------------------------------------------------------------------------|
| manufacturer<br>modelNumber<br>retailPrice                                                                                            |
| CellPhone()<br>setManufacturer()<br>setModelNumber()<br>setRetailPrice()<br>getManufacturer()<br>getModelNumber()<br>getRetailPrice() |

Rysunek 14.11. Uproszczony diagram UML klasy CellPhone

## Zapis typów danych i parametrów metod

Na diagramie UML zamieszczonym na rysunku 14.11 widnieją tylko podstawowe informacje na temat klasy CellPhone. Nie widać na nim szczegółów, takich jak typy danych i parametry metod. Aby dodać typ danych pola, umieszcza się po jego nazwie dwukropki i typ danych. Przykładowo pole manufacturer klasy CellPhone to łańcuch znakowy. Można je zapisać w diagramie UML w taki sposób:

```
manufacturer : String
```

Tak samo zapisujemy typ danych zwracany przez metodę. Po nazwie metody wstawiamy dwukropki i typ zwracanych danych. Metoda getRetailPrice klasy CellPhone zwraca liczbę typu Real, więc możemy ją zapisać na diagramie UML w taki sposób:

```
getRetailPrice() : Real
```

Parametry i ich typy danych umieszcza się w nawisach. Przykładowo metoda setManufacturer klasy CellPhone przyjmuje argument manufact w postaci łańcucha znakowego, więc zapiszmy ją w diagramie UML w taki sposób:

```
setManufacturer(manufact : String)
```

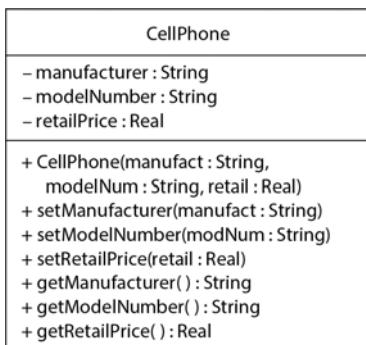
Na rysunku 14.12 przedstawiłem diagram UML klasy CellPhone, do którego dodałem typy danych i parametry.

| CellPhone                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| manufacturer : String<br>modelNumber : String<br>retailPrice : Real                                                                                                                                                                                               |
| CellPhone(manufact : String,<br>modelNum : String, retail : Real)<br>setManufacturer(manufact : String)<br>setModelNumber(modNum : String)<br>setRetailPrice(retail : Real)<br>getManufacturer() : String<br>getModelNumber() : String<br>getRetailPrice() : Real |

Rysunek 14.12. Diagram UML klasy CellPhone zawierający typy danych i parametry

## Zapis modyfikatorów dostępu

Diagramy UML przedstawione na rysunkach 14.11 i 14.12 zawierają listę pól i metod klasy CellPhone, ale w żaden sposób nie informują o tym, czy dane pole lub metoda są prywatne czy publiczne. Aby zaznaczyć, że dana metoda lub pole są prywatne, oznaczamy je na diagramie UML znakiem -, a żeby zaznaczyć, że dane pole lub metoda są publiczne oznaczamy je znakiem +. Na rysunku 14.13 przedstawiłem diagram UML, w którym dodałem modyfikatory dostępu.



**Rysunek 14.13.** Diagram UML klasy CellPhone zawierający modyfikatory dostępu



### Punkt kontrolny

- 14.14. Typowy diagram UML składa się z trzech części. Jakie informacje umieszczone są w każdej z tych części?
- 14.15. Założmy, że pewna klasa ma pole o nazwie `description`. Pole to jest typu `String`. Jak oznaczysz typ tego pola na diagramie UML?
- 14.16. Za pomocą których symboli oznaczamy, że dane pole lub metoda są prywatne lub publiczne?

**14.4**

## Wyznaczanie klas i ich zakresu obowiązków w zadaniu

**WYJAŚNIENIE:** Podczas projektowania programu obiektowego jednym z pierwszych zadań jest wyznaczenie klas, które będą potrzebne w programie, i określenie ich zakresu obowiązków.

Dotychczas omówiłem podstawy dotyczące tego, jak zapisuje się klasy, tworzy na ich podstawie obiekty i wykonuje określone zadania. Pomimo że wiedza ta jest konieczna do stworzenia aplikacji obiektowej, pracę należy rozpocząć od czegoś innego. Pierwszym

krokiem jest przeanalizowanie zadania, które masz zamiar rozwiązać, i wyznaczenie klas, których będziesz potrzebować. Sekcje „W centrum uwagi” przeprowadzą Cię przez ten proces i pomogą w wyznaczeniu klas i określaniu ich zakresu obowiązków.



## **W centrum uwagi**

### Wyznaczanie klas

Właściciel warsztatu samochodowego Joe's Automotive Shop poprosił Cię o zaprojektowanie programu, za pomocą którego będzie mógł wyceniać klientom prace serwisowe. Postanowiłeś zaprojektować program obiektowy. Jednym z pierwszych zadań jest określenie klas, których będziemy potrzebować w programie. W wielu przypadkach oznacza to wyodrębnienie z zadania obiektów rzeczywistych, na podstawie których utworzymy w programie klasy.

Na przestrzeni lat programiści opracowali wiele technik, za pomocą których można określić klasy potrzebne w danym zadaniu programistycznym. Jedna z prostszych i najbardziej popularnych technik polega na wykonaniu następujących kroków:

1. Stwórz pisemny opis modelu dziedziny.
2. Zaznacz wszystkie rzeczowniki, zaimki i wyrażenia występujące w opisie. Każde z tych słów to potencjalny kandydat na klasę.
3. Zawęż listę tych słów do najbardziej kluczowych w danym zadaniu.

Przyjrzyjmy się bliżej każdemu z tych kroków.

### **Tworzenie pisemnego opisu modelu dziedziny**

**Model dziedziny** to zbiór rzeczywistych obiektów, podmiotów i głównych zdarzeń związanych z danym zadaniem. Jeśli jesteś w stanie zrozumieć, na czym polega zadanie, które masz zamiar rozwiązać, możesz stworzyć opis modelu dziedziny samodzielnie. Jeśli nie do końca rozumiesz zadanie, opis powinien stworzyć dla Ciebie ktoś bardziej doświadczony.

Opis dziedziny modelu powinien zawierać co najmniej jeden z poniższych elementów:

- obiekty rzeczywiste, takie jak pojazdy, urządzenia czy produkty;
- role, jakie pełnią ludzie, na przykład menadżer, pracownik, klient, nauczyciel, uczni;
- wyniki procesów biznesowych, na przykład zamówienie złożone przez klienta (w naszym przypadku wycena prac serwisowych);
- elementy podlegające ewidencji, na przykład historia zakupów klienta, historia wypłat.

Oto opis stworzony przez właściciela serwisu samochodowego Joe's Automotive Shop:

Warsztat samochodowy Joe's Automotive Shop serwisuje samochody niemieckie, w szczególności samochody takich marek jak Mercedes, Porsche i BMW. Gdy klient dostarczy samochód do warsztatu, menadżer zapisuje jego imię i nazwisko, adres oraz numer telefonu. Menadżer określa markę, model i rok

produkci samochodu i przedstawia klientowi wycenę. Wycena zawiera szacunkowy koszt części, szacunkowy koszt robocizny, podatek oraz całkowity koszt usługi.

### Znalezienie rzeczowników

Następnym krokiem jest zaznaczenie w opisie modelu dziedziny wszystkich rzeczowników i wyrażeń (jeśli w opisie występują zaimki, to także je zaznaczamy). Przyjrzymy się ponownie opisowi modelu dziedziny sporządzonemu przez właściciela warsztatu. Tym razem wszystkie rzeczowniki i wyrażenia zaznaczyłem pogrubieniem.

**Warsztat samochodowy Joe's Automotive Shop** serwisuje samochody niemieckie, w szczególności samochody takich marek jak Mercedes, Porsche i BMW. Gdy klient dostarczy samochód do warsztatu, menadżer zapisuje jego imię i nazwisko, adres oraz numer telefonu. Menadżer określa markę, model i rok produkcji samochodu i przedstawia klientowi wycenę. Wycena zawiera szacunkowy koszt części, szacunkowy koszt robocizny, podatek oraz całkowity koszt usługi.

Zwróć uwagę, że niektóre rzeczowniki się powtarzają. Oto lista wszystkich rzeczowników i wyrażeń (bez powtórzeń):

|                        |                                            |
|------------------------|--------------------------------------------|
| adres                  | Porsche                                    |
| BMW                    | rok produkcji                              |
| całkowity koszt usługi | samochody                                  |
| imię i nazwisko        | samochody niemieckie                       |
| klient                 | samochód                                   |
| marka                  | szacunkowy koszt części                    |
| menadżer               | szacunkowy koszt robocizny                 |
| Mercedes               | warsztat                                   |
| model                  | warsztat samochodowy Joe's Automotive Shop |
| numer telefonu         | wycena                                     |
| podatek                |                                            |

### Zawężenie listy rzeczowników

Rzeczowniki, które umieściliśmy na liście, to tylko potencjalni kandydaci, na podstawie których możemy utworzyć klasy. Być może nie będziemy musieli tworzyć klasy dla każdego z tych rzeczowników i wyrażeń. Kolejnym krokiem jest zawężenie listy to klas, których będziemy potrzebować w danym zadaniu. Przyjrzymy się więc, jak możemy zawieźć listę potencjalnych klas.

#### 1. Niektóre rzeczowniki i wyrażenia mają takie samo znaczenie

W tym przypadku dwa elementy oznaczają to samo:

- **samochody i samochody niemieckie**  
Oba elementy odnoszą się do ogólnie pojętego samochodu.
- **warsztat samochodowy Joe's Automotive Shop i warsztat**  
Oba elementy oznaczają firmę Joe's Automotive Shop.

Do opisania każdego z tych elementów wystarczy nam jedna klasa. Wykreślimy więc z listy **samochody niemieckie** i pozostawimy **samochody**. Analogicznie wykreślimy z listy **warsztat samochodowy Joe's Automotive** i pozostawimy **warsztat**. Lista potencjalnych klas wygląda teraz tak:

|                        |                                                   |
|------------------------|---------------------------------------------------|
| adres                  | Porsche                                           |
| BMW                    | rok produkcji                                     |
| całkowity koszt usługi | samochody                                         |
| imię i nazwisko        | <b>samochody niemieckie</b>                       |
| klient                 | samochód                                          |
| marka                  | szacunkowy koszt części                           |
| menadżer               | szacunkowy koszt robocizny                        |
| Mercedes               | warsztat                                          |
| model                  | <b>warsztat samochodowy Joe's Automotive Shop</b> |
| numer telefonu         | wycena                                            |
| podatek                |                                                   |

Ponieważ zarówno **samochody**, jak i **samochody niemieckie** oznaczają w przypadku tego zadania to samo, wykreśliśmy z listy **samochody niemieckie**. Analogicznie **warsztat samochodowy Joe's Automotive** i **warsztat** oznaczają to samo, więc wykreśliśmy z listy **warsztat samochodowy Joe's Automotive**.

- Niektóre rzeczowniki to elementy, których nie będziemy potrzebować, aby rozwiązać zadanie.

Przyjrzyjmy się ponownie opisowi modelu dziedziny, aby przypomnieć sobie, jakie zadanie musi wykonywać aplikacja. W tym przypadku możemy wykreślić z listy dwie klasy, których nie będziemy potrzebować:

- Możemy wykreślić **warsztat**, ponieważ w aplikacji zajmujemy się jedynie wyceną usługi. Aplikacja nie będzie przetwarzala danych dotyczących samego warsztatu. Jeśli w opisie byłaby informacja o obliczaniu sumarycznych kosztów wszystkich prac serwisowych, wtedy jak najbardziej sensowne byłoby utworzenie klasy reprezentującej warsztat.
- Nie będziemy także potrzebować klasy **menadżer**, ponieważ opis nie wskazuje na to, abyśmy musieli przetwarzać jakiekolwiek informacje dotyczące samego menadżera. Gdyby w opisie była informacja, że w warsztacie pracuje kilku menadżerów i należy zapamiętać, który menadżer wyceniał dane prace serwisowe, wtedy utworzenie klasy dla menadżera miałoby sens.

Na tym etapie lista potencjalnych klas prezentuje się następująco:

|                        |                                                   |
|------------------------|---------------------------------------------------|
| adres                  | Porsche                                           |
| BMW                    | rok produkcji                                     |
| całkowity koszt usługi | samochody                                         |
| imię i nazwisko        | <b>samochody niemieckie</b>                       |
| klient                 | samochód                                          |
| marka                  | szacunkowy koszt części                           |
| <b>menadżer</b>        | szacunkowy koszt robocizny                        |
| Mercedes               | <b>warsztat samochodowy</b>                       |
| model                  | <b>warsztat samochodowy Joe's Automotive Shop</b> |
| numer telefonu         | wycena                                            |
| podatek                |                                                   |

W opisie modelu nie było informacji o konieczności przetwarzania danych dotyczących **warsztatu** i **menadżera**, więc wykreśliliśmy te elementy z listy.

- Niektóre rzeczowniki reprezentują obiekty, a nie klasy.

Na liście możemy także wykreślić słowa **Mercedes**, **Porsche** i **BMW**, ponieważ reprezentują one konkretne samochody i możemy je uważać za instancje klasy **samochody**. Możemy także usunąć słowo **samochód**. Z opisu wynika, że reprezentuje ono konkretny samochód, który klient dostarcza do warsztatu — jest on więc także przykładem instancji klasy **samochody**. Na tym etapie lista potencjalnych klas wygląda następująco:

|                        |                                                   |
|------------------------|---------------------------------------------------|
| adres                  | <b>Porsche</b>                                    |
| <b>BMW</b>             | rok produkcji                                     |
| całkowity koszt usługi | <b>samochody</b>                                  |
| imię i nazwisko        | <b>samochody niemieckie</b>                       |
| klient                 | <b>samochód</b>                                   |
| marka                  | <b>szacunkowy koszt części</b>                    |
| <b>menadżer</b>        | <b>szacunkowy koszt robocizny</b>                 |
| <b>Mercedes</b>        | <b>warsztat</b>                                   |
| model                  | <b>warsztat samochodowy Joe's Automotive Shop</b> |
| numer telefonu         | wycena                                            |
| podatek                |                                                   |

Wykreśliliśmy z listy elementy **Mercedes**, **Porsche**, **BMW** i **samochód**, ponieważ są to instancje klasy **samochody**. Oznacza to, że elementy te należy traktować jako obiekty, a nie jako klasy.



**WSKAZÓWKA:** Niektórzy projektanci programów obiektowych sprawdzają, czy rzeczownik występuje w liczbie pojedynczej czy mnogiej. Liczba mnoga wskazuje, że może to być klasa, a liczba pojedyncza — że może to być obiekt.

- Niektóre rzeczowniki reprezentują elementy, które możemy zapisać w zwykłych zmiennych, a nie w klasie.

Pamiętaj, że klasa zawiera pola i metody. Pola przechowują w obiekcie danej klasy wartości określające stan tego obiektu. Metody to działania, które można wykonać na obiekcie danej klasy. Jeśli rzeczownik reprezentuje coś, dla czego nie możemy określić pól lub metod, należy wykreślić go z listy. Aby określić, czy dla danego rzeczownika możemy wskazać pola i metody, zadaj sobie następujące pytania:

- Czy aby określić stan takiego obiektu będę potrzebować kilku wartości?
- Czy dany obiekt wykonuje jakieś operacje?

Jeśli odpowiedź na oba te pytania jest przecząca oznacza to, że dany rzeczownik reprezentuje wartość, którą można zapisać w zwykłej zmiennej. Jeżeli sprawdzimy w ten sposób wszystkie elementy pozostałe na liście, okaże się, że następujące elementy nie są klasami: **imię i nazwisko**, **adres**, **numer telefonu**, **marka**, **model**, **rok produkcji**, **szacunkowy koszt robocizny**, **szacunkowy koszt**

**części, podatek i całkowity koszt usługi.** Są to po prostu łańcuchy znakowe albo wartości liczbowe, które możemy zapisać w zmiennych. Teraz lista potencjalnych klas wygląda następująco:

|                               |                                            |
|-------------------------------|--------------------------------------------|
| adres                         | Porsche                                    |
| BMW                           | rok produkcji                              |
| <u>całkowity koszt usługi</u> | samochody                                  |
| imię i nazwisko               | samochody niemieckie                       |
| klient                        | samochód                                   |
| marka                         | szacunkowy koszt części                    |
| menadżer                      | szacunkowy koszt robocizny                 |
| Mercedes                      | warsztat                                   |
| model                         | warsztat samochodowy Joe's Automotive Shop |
| numer telefonu                | wycena                                     |
| podatek                       |                                            |

Wykreśliśmy z listy elementy **imię i nazwisko, adres, numer telefonu, marka, model, rok produkcji, szacunkowy koszt robocizny, szacunkowy koszt części, podatek i całkowity koszt usługi**, ponieważ reprezentują one proste wartości, które można zapisać w zwykłych zmiennych.

Jak widzisz, wykreśliśmy z listy wszystkie elementy z wyjątkiem elementów **samochody, klient i wycena**. Oznacza to, że w aplikacji będziemy potrzebować klas, które będą reprezentowały samochody, klientów i wycenę. W następnej sekcji „W centrum uwagi” stworzymy klasy Car (samochód), Customer (klient) i ServiceQuota (wycena).

## W centrum uwagi

### Określanie zakresu obowiązków klasy

W poprzedniej sekcji „W centrum uwagi” przyjrzaliśmy się opisowi modelu dziedziny w programie do sporządzania wyceny prac serwisowych. Określiliśmy klasy, których będziemy potrzebować: Car, Customer i ServiceQuote. Kolejnym krokiem będzie określenie zakresu obowiązków klasy. Obowiązki klasy to:

- rzeczy, która klasa musi pamiętać;
- operacje, które klasa musi wykonywać.

Kiedy określisz, które rzeczy klasa musi pamiętać, otrzymasz listę wartości, które należy zapisać w polach. Analogicznie jeśli określisz, które operacje musi wykonywać klasa, otrzymasz listę jej metod.

Warto sobie zadać pytanie: co musi pamiętać i robić klasa w programie? Pierwszym miejscem, w którym należy szukać odpowiedzi na to pytanie, jest opis modelu dziedziny. Znajdziesz tam kilka informacji dotyczących tego, co musi pamiętać i robić dana klasa. Niektóre rzeczy nie będą jednak wymienione w opisie wprost, więc często trzeba wysilić szare komórki. Zastosujmy tę metodologię w przypadku klas, które udało nam się określić.

## Klasa Customer

Pomyślmy, jakie informacje będzie musiała zapamiętać klasa `Customer`. W opisie modelu dziedziny wymienione są następujące elementy:

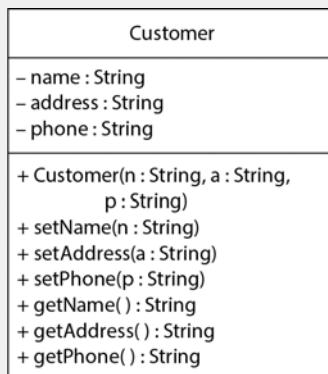
- imię i nazwisko klienta;
- adres klienta;
- numer telefonu klienta.

Do reprezentowania tych wartości możemy wykorzystać pola typu `String`. Klasa `Customer` może potencjalnie pamiętać wiele innych rzeczy. Bardzo często popełnianym błędem na tym etapie jest określenie zbyt wielu wartości. W przypadku niektórych programów klasa `Customer` może zapamiętywać adres e-mail klienta. W tym przypadku opis modelu dziedziny nie wskazuje na to, abyśmy musieli zapamiętywać adres e-mail klienta, więc nie powinniśmy umieszczać go w klasie.

Określmy teraz metody klasy. Jakie operacje musi wykonywać klasa `Customer` w projektowanym programie? Najbardziej oczywiste operacje to:

- tworzenie instancji klasy `Customer`;
- pobieranie i ustawianie imienia i nazwiska klienta;
- pobieranie i ustawianie adresu klienta;
- pobieranie i ustawianie numeru telefonu klienta.

Na podstawie tej listy widzimy, że klasa będzie musiała mieć konstruktor i kilka akcesorów i mutatorów — dla poszczególnych pól. Na rysunku 14.14 przedstawiłem diagram UML klasy `Customer`. Na listingu klasy 14.5 zamieściłem pseudokod definicji klasy.



Rysunek 14.14. Diagram UML klasy `Customer`

### Listing klasy 14.5

```

1 Class Customer
2   //Pola
3   Private String name
4   Private String address
5   Private String phone
6
7   //Konstruktor
  
```

```

8  Public Module Customer(String n, String a,
9      String p)
10     Set name = n
11     Set address = a
12     Set phone = p
13 End Module
14
15 //Mutatory
16 Public Module setName(String n)
17     Set name = n
18 End Module
19
20 Public Module setAddress(String a)
21     Set address = a
22 End Module
23
24 Public Module setPhone(String p)
25     Set phone = p
26 End Module
27
28 //Akcesory
29 Public Function String getName()
30     Return name
31 End Function
32
33 Public Function String getAddress()
34     Return address
35 End Function
36
37 Public Function String getPhone()
38     Return phone
39 End Function
40 End Class

```

## Klasa Car

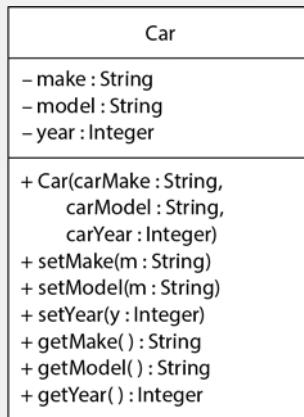
Jakie informacje będzie musiała zapamiętać klasa `Car`? Poniższa lista to właściwości samochodu, które wymienione są w opisie modelu dziedziny:

- marka samochodu;
- model samochodu;
- rok produkcji samochodu.

Określmy teraz metody klasy. Jakie operacje musi wykonywać klasa `Car` w projektowanym programie? Ponownie najbardziej oczywistymi operacjami są metody, które znajdziemy w większości klas (konstruktor, akcesory i mutatory). W tym przypadku będą to następujące operacje:

- tworzenie instancji klasy `Car`;
- pobieranie i ustawianie marki samochodu;
- pobieranie i ustawianie modelu samochodu;
- pobieranie i ustawianie roku produkcji samochodu.

Na rysunku 14.15 przedstawiłem diagram UML klasy `Car`. Na listingu klasy 14.6 zamieściłem pseudokod definicji klasy.

**Rysunek 14.15.** Diagram UML klasy Car**Listing klasy 14.6**

```

1 Class Car
2   //Pola
3   Private String make
4   Private String model
5   Private Integer year
6
7   //Konstruktor
8   Public Module Car(String carMake,
9     String carModel, Integer carYear)
10  Set make = carMake
11  Set model = carModel
12  Set year = carYear
13 End Module
14
15 //Mutatory
16 Public Module setMake(String m)
17   Set make = m
18 End Module
19
20 Public Module setModel(String m)
21   Set model = m
22 End Module
23
24 Public Module setYear(Integer y)
25   Set year = y
26 End Module
27
28 //Akcesory
29 Public Function String getMake()
30   Return make
31 End Function
32
33 Public Function String getModel()
34   Return model
35 End Function
36
37 Public Function Integer getYear()
  
```

```

38      Return year
39  End Function
40 End Class

```

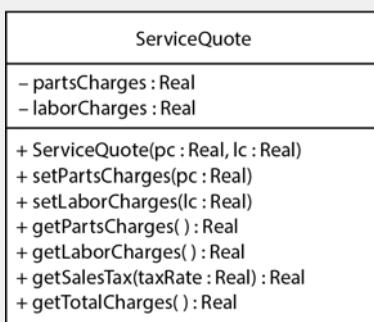
### Klasa ServiceQuote

Jakie informacje będzie musiała zapamiętać klasa ServiceQuote? W opisie modelu dziedziny znajdziemy następujące elementy:

- szacunkowy koszt części;
- szacunkowy koszt robocizny;
- wartość podatku;
- całkowity koszt usługi.

Po przemyśleniu okazuje się, że dwa z tych elementów możemy tak naprawdę obliczyć: wartość podatku i całkowity koszt usługi. Wartości te uzależnione są od szacunkowego kosztu części i robocizny. Zamiast zapisywać te wartości w polach, utworzymy odpowiednie metody, które będą obliczały i zwracały te wartości. Za chwilę wyjaśnię, dlaczego postanowiłem zastosować takie podejście.

Pozostałe metody, których będziemy potrzebować, to konstruktor, akcesory i mutatory dla pól, w których zapisany jest szacunkowy koszt części i robocizny. Na rysunku 14.16 przedstawiłem diagram UML klasy Car. Na listingu klasy 14.7 zamieściłem pseudokod definicji klasy.



**Rysunek 14.16.** Diagram UML klasy ServiceQuote

### Listing klasy 14.7

```

1 Class ServiceQuote
2 //Pola
3 Private Real partsCharges
4 Private Real laborCharges
5
6 //Konstruktor
7 Public Module ServiceQuote(Real pc, Real lc)
8     Set partsCharges = pc
9     Set laborCharges = lc
10 End Module

```

```

11
12 // Mutatory
13 Public Module setPartsCharges(Real pc)
14     Set partsCharges = pc
15 End Module
16
17 Public Module setLaborCharges(Real lc)
18     Set laborCharges = lc
19 End Module
20
21 // Akcesory
22 Public Function Real getPartsCharges()
23     Return partsCharges
24 End Function
25
26 Public Function Real getLaborCharges()
27     Return laborCharges
28 End Function
29
30 Public Function Real getSalesTax(Real taxRate)
31     // Podatek doliczamy tylko do kosztów części
32     Return partsCharges * taxRate
33 End Function
34
35 Public Function Real getTotalCharges(Real taxRate)
36     Return partsCharges + laborCharges + getSalesTax(taxRate)
37 End Function
38 End Class

```

Zwróć uwagę, że metoda `getSalesTax` w liniach od 30. do 33. przyjmuje jako argument stawkę podatku w postaci liczby typu `Real`. Metoda zwraca w linii 32. obliczoną wartość podatku.

Metoda `getTotalCharges` umieszczona w liniach od 35. do 37. zwraca sumę kosztów usługi. Zwracana wartość jest wynikiem obliczeń — w linii 36. pojawia się polecenie `partsCharges + laborCharges + getSalesTax(taxRate)`. Zauważ, że w wyrażeniu tym korzystam z jednej z metod klasy: `getSalesTax`.

### Unikanie nieaktualnych danych

Metody `getPartsCharges` i `getLaborCharges` w klasie `ServiceQuote` zwracają wartości zapisane w polach, ale metody `getSalesTax` i `getTotalCharges` zwracają wartości będące wynikiem obliczeń. Być może zastanawiasz się, dlaczego wartość podatku i całkowity koszt usługi nie są również zapisane w polach. To dlatego, że pola te mogą potencjalnie zawierać **nieaktualne** dane. Kiedy wartość jednego z pól jest zależna od innych danych i w momencie, gdy dane te ulegną zmianie, nie zaktualizujemy wartości pola, zapisane w nim dane staną się nieaktualne. Gdybyśmy zapisali w polach wartość podatku i całkowity koszt usługi, wartości te stałyby się nieaktualne w chwili, kiedy zmianie ulegnie wartość zapisana w polu `partsCharges` lub `laborCharges`.

Projektując klasę, upewnij się, że nie zapisujesz w polach danych, które są wynikiem obliczeń i potencjalnie mogą stać się nieaktualne. Zamiast tego umieść w klasie metodę, która będzie zwracała wynik obliczeń.



## Punkt kontrolny

- 14.17. Co to jest opis modelu dziedziny?
- 14.18. Na czym polega technika określania klas potrzebnych w zadaniu, którą zaprezentowałem w tym podrozdziale?
- 14.19. Co to jest zakres obowiązków klasy?
- 14.20. Kiedy dane mogą stać się nieaktualne?

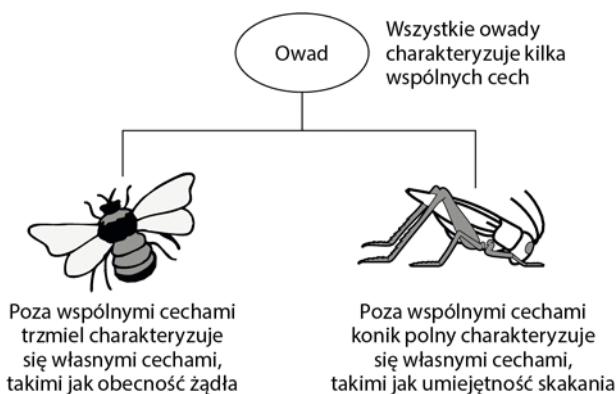
14.5

## Dziedziczenie

**WYJAŚNIENIE:** Dzięki dziedziczeniu jedna klasa może być rozwinięciem innej klasy. Nowa klasa dziedziczy składowe po klasie bazowej.

### Uogólnienie i uszczegółowienie

W prawdziwym świecie można znaleźć wiele przykładów obiektów, które są szczególnymi przypadkami innych obiektów. Przykładowo określenie *owad* wskazuje na bardzo ogólną formę zwierzęcia, które możemy opisać za pomocą pewnych cech. Ponieważ zarówno konik polny, jak i trzmiel są owadami, można je opisać za pomocą pewnych wspólnych cech. Ale każdy z tych owadów charakteryzuje się także swoimi własnymi cechami odróżniającymi go od innych owadów. Przykładowo konik polny potrafi skakać, a trzmiel wyposażony jest w żądło. Konik polny i trzmiel są więc szczególnymi przypadkami owadów. Ilustruje to rysunek 14.17.



Rysunek 14.17. Trzmiel i konik polny to szczególne przypadki owadów

## Dziedziczenie i relacja typu „jest”

Kiedy jeden z obiektów jest szczególnym przypadkiem innego obiektu, zachodzi między nimi relacja typu „jest”. Przykładowo konik polny jest owadem. Oto kilka innych przykładów relacji typu „jest”:

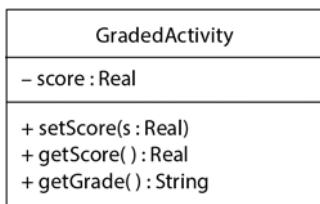
- pudel jest psem;
- samochód jest pojazdem;
- kwiat jest rośliną;
- prostokąt jest kształtem;
- piłkarz jest sportowcem.

Kiedy między dwoma obiektami zachodzi relacja typu „jest”, oznacza to, że obiekt będącym szczególnym przypadkiem drugiego obiektu ma wszystkie jego cechy, ale także kilka innych cech, które sprawiają, że jest on szczególnym przypadkiem. W przypadku programowania obiektowego odpowiednikiem relacji „jest” jest dziedziczenie klas. Dzięki dziedziczeniu można rozwinąć daną klasę, tworząc nową klasę, będącą jej szczególnym przypadkiem.

Z dziedziczeniem wiążą się pojęcia klasy bazowej i klasy pochodnej. **Klasa pochodna** to szczególny przypadek **klasy bazowej**. Możesz sobie wyobrazić, że klasa pochodna jest rozwinięciem klasy bazowej. Klasa pochodna dziedziczy po klasie bazowej pola i metody — bez konieczności ich przepisywania. Ponadto do klasy pochodnej można dodawać nowe pola i metody, dzięki którym stanie się ona szczególnym przypadkiem klasy bazowej.

Spójrzmy więc, jak możemy wykorzystać dziedziczenie w programie. Nauczyciele zadają uczniom zadania, z których uczniowie otrzymują oceny. Uczeń uzyskuje z danego zadania określony wynik (np. 70, 85, 90 itp.), na podstawie którego wyznaczana jest ocena: A, B, C, D lub F.

Na rysunku 14.18 przedstawiłem diagram UML klasy `GradedActivity`, która przechowuje wynik z zadania. Metoda `setScore` służy do ustawiania wyniku liczbowego, a metoda `getScore` zwraca wynik liczbowy. Metoda `getGrade` zwraca ocenę literową odpowiadającą danemu wynikowi liczbowemu. Na listingu klasy 14.8 przedstawiłem pseudokod klasy. Na listingu 14.4 pokazałem, jak wykorzystałem tę klasę.



**Rysunek 14.18.** Diagram UML klasy `GradedActivity`

**Listing klasy 14.8**

```

1 Class GradedActivity
2 // Pole score zawiera wynik liczbowy
3 Private Real score
4
5 // Mutator
6 Public Module setScore(Real s)
7     Set score = s
8 End Module
9
10 // Akcesor
11 Public Function Real getScore()
12     Return score
13 End Function
14
15 // Metoda getGrade
16 Public Function String getGrade()
17     // Zmienna lokalna, w której zapiszę ocenę
18     Declare String grade
19
20     // Wyznaczamy ocenę
21     If score >= 90 Then
22         Set grade = "A"
23     Else If score >= 80 Then
24         Set grade = "B"
25     Else If score >= 70 Then
26         Set grade = "C"
27     Else If score >= 60 Then
28         Set grade = "D"
29     Else
30         Set grade = "F"
31     End If
32
33     // Zwracam ocenę
34     Return grade
35 End Function
36 End Class

```

**Listing 14.4**

```

1 Module main()
2 // Zmienna, w której zapiszę wynik
3 Declare Real testScore
4
5 // Zmienna obiektowa
6 // odnosząca się do obiektu typu GradedActivity
7 Declare GradedActivity test
8
9 // Tworzę obiekt typu GradedActivity
10 Set test = New GradedActivity()
11
12 // Pobieram od użytkownika wynik
13 Display "Wprowadź wynik."
14 Input testScore
15
16 // Zapisuję wynik wewnątrz obiektu
17 test.setScore(testScore)

```

```

18
19 // Wyświetlam ocenę
20 Display "Uczeń otrzymał ocenę ",
21 test.getGrade()
22 End Module

```

**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik.

**89 [Enter]**

Uczeń otrzymał ocenę B

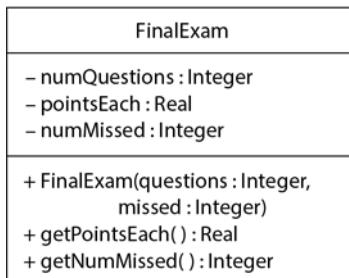
**Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź wynik.

**75 [Enter]**

Uczeń otrzymał ocenę C

Klasa `GradedActivity` reprezentuje ogólne zadanie, z którego uczniów może otrzymać ocenę. Zadania mogą być jednak różnego typu: testy, egzaminy końcowe, sprawozdania z ćwiczeń laboratoryjnych, wypracowania itp. Ponieważ w przypadku każdego z tych zadań wynik liczbowy może być obliczany w inny sposób, można stworzyć dla każdego z nich osobną klasę pochodną. Przykładowo klasa `FinalExam` może być klasą pochodną klasy `GradedActivity`. Na rysunku 14.19 przedstawiłem diagram UML takiej klasy, a na listingu klasy 14.9 jej pseudokod. Klasa ta ma pola, w których zapisana jest liczba pytań na egzaminie końcowym (`numQuestions`), liczba punktów za każde pytanie (`pointsEach`) oraz liczba pytań, na które uczeń odpowiedział błędnie (`numMissed`).

**Rysunek 14.19.** Diagram UML klasy `FinalExam`**Listing klasy 14.9**

```

1 Class FinalExam Extends GradedActivity
2 //Pola
3 Private Integer numQuestions
4 Private Real pointsEach
5 Private Integer numMissed
6
7 //Konstruktor ustawia
8 //liczbę pytań na egzaminie
9 //i liczbę pytań, na które uczeń odpowiedział błędnie
10 Public Module FinalExam(Integer questions,
                           Integer missed)
11
12 //Zmienna lokalna, w której zapiszę wynik liczbowy z egzaminu
13 Declare Real numericScore
14

```

```

15 // Ustawiamy pola numQuestions i numMissed
16 Set numQuestions = questions
17 Set numMissed = missed
18
19 // Obliczamy liczbę punktów za każde pytanie
20 // i wynik liczbowy z egzaminu
21 Set pointsEach = 100.0 / questions
22 Set numericScore = 100.0 - (missed * pointsEach)
23
24 // Aby ustawić wynik liczbowy,
25 // wywołujemy dziedziczoną metodę setScore.
26 Call setScore(numericScore)
27 End Module
28
29 // Akcesory
30 Public Function Real getPointsEach()
31     Return pointsEach
32 End Function
33
34 Public Function Integer getNumMissed()
35     Return numMissed
36 End Function
37 End Class

```

Zwróć uwagę, że w pierwszej linii klasy FinalExam znajduje się słowo kluczowe Extends, które wskazuje, że klasa ta dziedziczy po innej klasie (bazowej). Po słowie Extends znajduje się nazwa klasy bazowej. W tym przypadku FinalExam jest nazwą nowej klasy, a GradedActivity to klasa bazowa, po której klasa FinalExam dziedziczy.

Jeśli chcielibyśmy opisać relację między tymi dwiema klasami, moglibyśmy powiedzieć, że egzamin końcowy (FinalExam) jest zadaniem podlegającym ocenie (GradedActivity). Ponieważ klasa FinalExam jest rozwinięciem klasy GradedActivity, dziedziczy ona później wszystkie publiczne składowe. Oto lista składowych klasy FinalExam:

#### Pola:

- numQuestions — zadeklarowane w klasie FinalExam
- pointsEach — zadeklarowane w klasie FinalExam
- numMissed — zadeklarowane w klasie FinalExam

#### Metody:

- Konstruktor — zadeklarowana w klasie FinalExam
- getPointsEach — zadeklarowana w klasie FinalExam
- getNumMissed — zadeklarowana w klasie FinalExam
- setScore — dziedziczone po klasie GradedActivity
- getScore — dziedziczone po klasie GradedActivity
- getGrade — dziedziczone po klasie GradedActivity

Zwróć uwagę, że wśród składowych klasy FinalExam nie ma pola score klasy GradedActivity. To dlatego, że pole score jest polem prywatnym. W większości języków składowe prywatne klasy bazowej są niedostępne dla klasy pochodnej, czyli innymi słowy, klasa pochodna ich nie dziedziczy. Kiedy tworzymy obiekt klasy pochodnej, składowe prywatne istnieją w pamięci komputera, ale mogą się do nich odwoływać tylko metody klasy bazowej. Są to więc faktycznie składowe prywatne klasy bazowej.

Aby zrozumieć, jak działa dziedziczenie w naszym przykładzie, przyjrzyjmy się bliżej konstruktorowi klasy FinalExam w liniach od 10. do 27. Konstruktor przyjmuje dwa argumenty: liczbę pytań egzaminacyjnych i liczbę pytań, na które uczeń odpowiedział błędnie. W liniach 16. i 17. przypisuję przekazane wartości do pól numQuestions i numMissed. Następnie w liniach 21. i 22. obliczam liczbę punktów za każde pytanie i wynik. W linii 26. pojawia się ostatnie polecenie konstruktora:

```
Call setScore(numericScore)
```

Wywołuję tutaj metodę setScore, która została odziedziczona po klasie GradedActivity. Mimo że konstruktor klasy FinalExam nie może się odwołać bezpośrednio do pola score (jest ono polem prywatnym dla klasy GradedActivity), może wywołać metodę setScore, która przypisze do pola score określzoną wartość.

Na pseudokodzie z listingu 14.5 przedstawiłem przykład wykorzystania klasy FinalExam.

### **Listing 14.5**



```

1 Module main()
2   //Zmienne, w których zapiszę dane wprowadzone przez użytkownika
3   Declare Integer questions, missed
4
5   //Zmienna obiektowa, za pomocą której będę się odnosił do obiektu typu FinalExam
6   Declare FinalExam exam
7
8   //Proszę użytkownika o wprowadzenie
9   //liczby pytań na egzaminie
10  Display "Wprowadź liczbę pytań egzaminacyjnych."
11  Input questions
12
13  //Proszę użytkownika o wprowadzenie liczby pytań,
14  //na które uczeń odpowiedział błędnie
15  Display "Wprowadź liczbę pytań, na które uczeń odpowiedział",
16  "błędnie."
17  Input missed
18
19  //Tworzymy obiekt typu FinalExam
20  Set exam = New FinalExam(questions, missed)
21
22  //Wyświetlamy wynik egzaminu
23  Display "Za każde pytanie można uzyskać ",
24  exam.getPointsEach(), " punktów."
25  Display "Wynik z egzaminu wynosi ", exam.getScore()
26  Display "Uczeń otrzymał ocenę ", exam.getGrade()
27 End Module

```

#### **Wynik działania programu (pogrubione linie to dane wprowadzone przez użytkownika)**

Wprowadź liczbę pytań egzaminacyjnych.

**20 [Enter]**

Wprowadź liczbę pytań, na które uczeń odpowiedział błędnie.

**3 [Enter]**

Za każde pytanie można uzyskać 5 punktów.

Wynik z egzaminu wynosi 85

Uczeń otrzymał ocenę B

W linii 20. tworzę instancję klasy `FinalExam` i przypisuję ją do zmiennej `exam` za pomocą następującego polecenia:

```
Set exam = New FinalExam(question, missed)
```

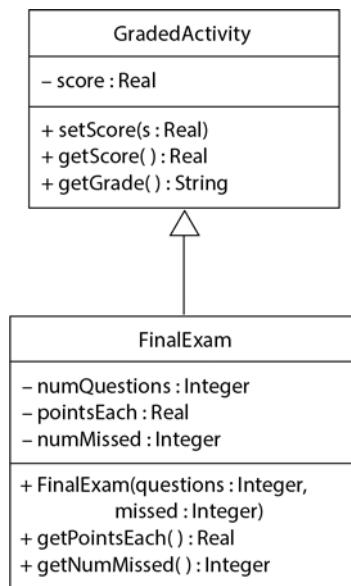
Kiedy obiekt typu `FinalExam` zostanie utworzony w pamięci komputera, będzie miał nie tylko składowe zadeklarowane w klasie `FinalExam`, ale także wszystkie nieprywatne składowe zadeklarowane w klasie `GradedActivity`. Zauważ, że w przedstawionych poniżej liniach kodu odwołuję się bezpośrednio do metod `getScore` i `getGrade` klasy `GradedActivity`:

```
Display " Wynik z egzaminu wynosi ", exam.getScore()
Display " Uczeń otrzymała ocenę ", exam.getGrade()
```

Kiedy klasa pochodna rozwija klasę bazową, wszystkie składowe publiczne klasy bazowej stają się publicznymi składowymi klasy pochodnej. W przypadku tego programu metody `getScore` i `getGrade` można wywołać względem obiektu `exam`, ponieważ są to publiczne składowe obiektu klasy bazowej.

## Oznaczanie dziedziczenia na diagramach UML

Dziedziczenie oznaczamy na diagramach UML za pomocą strzałki łączącej obie klasy. Strzałka skierowana jest na klasę bazową. Na rysunku 14.20 widoczny jest diagram UML przedstawiający relację między klasami `GradedActivity` i `FinalExam`.



**Rysunek 14.20.** Diagram UML ilustrujący dziedziczenie

## Dziedziczenie nie działa w drugą stronę

W przypadku dziedziczenia klasa pochodna dziedziczy po klasie bazowej, a nie odwrotnie. Oznacza to, że względem klasy bazowej nie można wywoływać metod klasy pochodnej. Przykładowo jeśli utworzymy obiekt `GradedActivity`, nie możemy względem niego wywołać metod `getPointsEach` i `getNumMissed`, ponieważ są one częścią klasy `FinalExam`.



### Punkt kontrolny

- 14.21. W tym podroziale wyjaśnilem, czym są klasa bazowa i klasa pochodna. Która z nich jest szczególnym przypadkiem drugiej?
- 14.22. Co oznacza, że między dwoma obiektami zachodzi relacja typu „jest”?
- 14.23. Które składowe dziedziczy klasa pochodna po klasie bazowej?
- 14.24. Spójrz na poniższy pseudokod. Która linia jest pierwszą linią definicji klasy? Jaką nazwę ma klasa bazowa? Jaką nazwę ma klasa pochodna?

```
Class Canary Extends Bird
```

**14.6**

## Polimorfizm

**WYJAŚNIENIE:** Dzięki polimorfizmowi można tworzyć w dziedziczących po sobie klasach metody o takich samych nazwach. W zależności od typu obiektu, względem którego wywołujemy metodę, zostanie wywołana odpowiednia wersja metody.

**Polimorfizm** oznacza umiejętność przybierania różnych form. To bardzo ważne zagadnienie w przypadku programowania obiektowego. W tym podroziale omówię dwa aspekty polimorfizmu:

1. Możliwość zdefiniowania metody o takiej samej nazwie zarówno w klasie bazowej, jak i w klasie pochodnej. Kiedy klasa pochodna ma metodę o takiej samej nazwie jak klasa bazowa, mówimy, że klasa pochodna zastępuje (ang. *override*) metodę klasy bazowej.
2. Możliwość zadeklarowania zmiennej typu klasa bazowej i przypisywania do niej zarówno obiektów typu klasa bazowej, jak i obiektów typu klasa pochodnej.

Najlepiej będzie, jeśli omówię polimorfizm na pewnym przykładzie. Na listingu klasy 14.10 znajduje się pseudokod klasy o nazwie `Animal`.

### Listing klasy 14.10



```
1 Class Animal
2 // Metoda showSpecies
3 Public Module showSpecies()
4     Display "Jestem zwierzakiem."
5 End Module
```

```

6
7 // Metoda makeSound
8 Public Module makeSound()
9     Display "Grrrrr."
10 End Module
11 End Class

```

Klasa ma dwie metody: `showSpecies` i `makeSound`. Oto przykładowy pseudokod, za pomocą którego tworzęinstancję klasy `Animal` i wywouję jej metody:

```

Declare Animal myAnimal
Set myAnimal = New Animal()
Call myAnimal.showSpecies()
Call myAnimal.makeSound()

```

Gdyby był to kod prawdziwego programu, na ekranie wyświetliły się następujący tekst:

```

Jestem zwierzakiem.
Grrrrr.

```

Spójrzmy teraz na listing klasy 14.11, na którym przedstawiłem pseudokod klasy `Dog`. Klasa `Dog` jest klasą pochodną klasy `Animal`.

### **Listing klasy 14.11**



```

1 Class Dog Extends Animal
2 // Metoda showSpecies
3 Public Module showSpecies()
4     Display "Jestem psem."
5 End Module
6
7 // Metoda makeSound
8 Public Module makeSound()
9     Display "Hau! Hau!"
10 End Module
11 End Class

```

Mimo że klasa `Dog` dziedziczy po klasie `Animal` metody `showSpecies` i `makeSound`, będą one działały niezbyt precyzyjnie dla klasy `Dog`. Klasa `Dog` ma więc własne metody `showSpecies` i `makeSound`, które wyświetlają komunikaty odpowiednie dla psa. Mówimy w takim przypadku, że metody `showSpecies` i `makeSound` klasy `Dog` zastępują metody `showSpecies` i `makeSound` klasy `Animal`. Oto przykładowy pseudokod, za pomocą którego tworzęinstancję klasy `Dog` i wywouję jej metody:

```

Declare Dog myDog
Set myDog = New Dog()
Call myDog.showSpecies()
Call myDog.makeSound()

```

Gdyby był to kod prawdziwego programu, na ekranie wyświetliły się następujący tekst:

```

Jestem psem.
Hau! Hau!

```

Na listingu klasy 14.12 przedstawiłem pseudokod klasy `Cat`, która jest także klasą pochodną klasy `Animal`.

### **Listing klasy 14.12**

```

1 Class Cat Extends Animal
2 // Metoda showSpecies
3 Public Module showSpecies()
4     Display "Jestem kotem."
5 End Module
6
7 // Metoda makeSound
8 Public Module makeSound()
9     Display "Miau."
10 End Module
11 End Class

```



Klasa `Cat` także ma metody `showSpecies` i `makeSound`. Oto przykładowy pseudokod, za pomocą którego tworzę instancję klasy `Cat` i wywołuję jej metody:

```

Declare Cat myCat
Set myCat = New Cat()
Call myCat.showSpecies()
Call myCat.makeSound()

```

Gdyby był to kod prawdziwego programu, na ekranie wyświetliłby się następujący tekst:

```

Jestem kotem.
Miau.

```

Ponieważ między dziedziczącymi po sobie klasami występuje relacja „jest”, obiekt klasy `Dog` jest nie tylko obiektem `Dog` — jest także obiektem `Animal` (pies jest zwierzęciem). Dzięki tej relacji możemy za pomocą zmiennej typu `Animal` odnosić się do obiektu typu `Dog`. Spójrz na następujący pseudokod:

```

Declare Animal myAnimal
Set myAnimal = New Dog()
Call myAnimal.showSpecies()
Call myAnimal.makeSound()

```

W pierwszym wierszu deklaruję zmienną `myAnimal` typu `Animal`. W drugim wierszu tworzę obiekt typu `Dog` i zapisuję jego adres w zmiennej `myAnimal`. Wykonanie tego polecenia jest dopuszczalne w wielu językach programowania, ponieważ obiekt typu `Dog` jest także obiektem typu `Animal`. W trzecim i czwartym wierszu wywołuję metody `showSpecies` i `makeSound` obiektu `myAnimal`. Gdyby był to kod prawdziwego programu, w większości języków programowania na ekranie wyświetliłby się następujący tekst:

```

Jestem psem.
Hau! Hau!

```

Analogicznie do zmiennej typu `Animal` możemy przypisać obiekt typu `Cat`:

```

Declare Animal myAnimal
Set myAnimal = New Cat()
Call myAnimal.showSpecies()
Call myAnimal.makeSound()

```

Gdyby był to kod prawdziwego programu, w większości języków programowania na ekranie wyświetliłby się następujący tekst:

Jestem kotem.

Miau.

Ta właściwość polimorfizmu daje podczas projektowania programów bardzo dużą elastyczność. Spójrz na przykład na taki moduł:

```
Module showAnimalInfo(Animal creature)
    Call creature.showSpecies()
    Call creature.makeSound()
End Module
```

Moduł ten wyświetla informacje dotyczące zwierzęcia. Ponieważ parametr modułu jest typu `Animal`, możemy do niego podczas wywołania przekazać dowolny obiekt typu `Animal`. Moduł wywołuje po prostu metody `showSpecies` i `makeSound` przekazanego do niego obiektu.

Moduł `showAnimalInfo` obsługuje obiekty typu `Animal`. A co w przypadku, gdybyśmy chcieli wyświetlić także informacje zapisane w obiektach typu `Dog` czy `Cat`? Musielibyśmy utworzyć nowe moduły dla każdego z tych typów? Dzięki polimorfizmowi można odpowiedzieć na to pytanie przecząco. Poza obiektami typu `Animal` do modułu `showAnimalInfo` można także przekazywać obiekty typu `Dog` i `Cat`. Zademonstrowałem to na listingu 14.6.

### **Listing 14.6**



```
1 Module main()
2 // Deklarujemy trzy zmienne obiektowe
3 Declare Animal myAnimal
4 Declare Dog myDog
5 Declare Cat myCat
6
7 // Tworzymy obiekty typu Animal:
8 // Dog i Cat
9 Set myAnimal = New Animal()
10 Set myDog = New Dog()
11 Set myCat = New Cat()
12
13 // Wyświetlamy informacje dotyczące zwierzęcia
14 Display "Oto informacje dotyczące zwierzęcia:"
15 showAnimalInfo(myAnimal)
16 Display
17
18 // Wyświetlamy informacje dotyczące psa
19 Display "Oto informacje dotyczące psa:"
20 showAnimalInfo(myDog)
21 Display
22
23 // Wyświetlamy informacje dotyczące kota
24 Display "Oto informacje dotyczące kota:"
25 showAnimalInfo(myCat)
26 End Module
27
28 // Moduł showAnimalInfo przyjmuje jako argument
29 // obiekt typu Animal i wyświetla
30 // dotyczące go informacje
```

```

31 Module showAnimalInfo(Animal creature)
32   Call creature.showSpecies()
33   Call creature.makeSound()
34 End Module

```

### **Wynik działania programu**

Oto informacje dotyczące zwierzęcia:  
 Jestem zwierzakiem.  
 Grrrrr.

Oto informacje dotyczące psa:  
 Jestem psem.  
 Hau! Hau!

Oto informacje dotyczące kota:  
 Jestem kotem.  
 Miau.

Chociaż przedstawiony przeze mnie przykład jest bardzo prosty, polimorfizm ma wiele praktycznych zastosowań. Założymy, że projektujemy dla uczelni program, który będzie przetwarzał dużo danych dotyczących studentów. Moglibyśmy stworzyć w takim programie klasę o nazwie **Student**. Jedną z metod klasy **Student** byłaby metoda o nazwie **getFee**, która zwracałaby wysokość opłaty za semestr nauki.

Moglibyśmy utworzyć także klasę **BiologyStudent**, będącą klasą pochodną klasy **Student** (student biologii jest studentem). Ponieważ student biologii uczęszcza dodatkowo na zajęcia z biologii, opłata za semestr studiów jest w jego przypadku wyższa niż opłata dla zwykłego studenta. Klasa **BiologyStudent** miałaby więc własną metodę **getFee**, która zwracałaby wysokość opłat za semestr nauki dla studenta biologii.



## **Punkt kontrolny**

14.25. Spójrz na następujące definicje klas:

```

Class Vegetable
  Public Module message()
    Display "Jestem warzywem."
  End Module
End Class
Class Potato Extends Vegetable
  Public Module message()
    Display "Jestem ziemniakiem."
  End Module
End Class

```

Co wyświetli się na ekranie po wykonaniu poniższego pseudokodu?

```

Declare Vegetable v
Declare Potato p
Set v = New Potato()
Set p = New Potato()
Call v.message()
Call p.message()

```

**14.7**

## Rzut oka na języki Java, Python i C++

W niniejszym podrozdziale omówię sposoby implementowania w językach programowania Java, Python i C++ poszczególnych zagadnień zawartych w tym rozdziale. Jeśli chcesz lepiej poznać te języki programowania i przyjrzeć się większej liczbie przykładowych programów w każdym z nich, możesz pobrać pozycje *Java Language Companion*, *Python Language Companion* oraz *C++ Language Companion* ze strony wydawnictwa: <ftp://ftp.helion.pl/przyklady/plkpro5.zip>.

### **Java**

#### Klasy i obiekty

##### **Deklaracje klas w języku Java**

Przedstawiona na listingu 14.13 klasa CellPhone jest wersją klasy z listingu 14.3 napisaną w języku Java. Zauważ, że każda z deklaracji pól (wiersze od 4. do 6.) zaczyna się od słowa kluczowego private. Jest to tak zwany specyfikator dostępu, który definiuje pola prywatne dla danej klasy. Żaden kod znajdujący się poza tą klasą nie ma bezpośredniego dostępu do prywatnych pól tej klasy. Zauważ też, że każdy z nagłówków metod zaczyna się od specyfikatora dostępu public. To sprawia, że metody te są publiczne, a zatem jakikolwiek kod spoza tej klasy może je wywoływać.

**Listing klasy 14.13 (CellPhone.java)**

```

1  public class CellPhone
2  {
3      //Deklarujemy pola
4      private String manufacturer;
5      private String modelNumber;
6      private double retailPrice;
7
8      //Definiujemy metody
9      public void setManufacturer(String manufact)
10     {
11         manufacturer = manufact;
12     }
13
14     public void setModelNumber(String modNum)
15     {
16         modelNumber = modNum;
17     }
18
19     public void setRetailPrice(double retail)
20     {
21         retailPrice = retail;
22     }
23
24     public String getManufacturer()
25     {
26         return manufacturer;
27     }
28 }
```

```

27 }
28
29     public String getModelNumber()
30     {
31         return modelNumber;
32     }
33
34     public double getRetailPrice()
35     {
36         return retailPrice;
37     }
38 }
```

Zauważ, że żadna z deklaracji pól czy metod nie zawiera słowa kluczowego `static`. Wszystkie deklaracje pól i metod w tej klasie są **niestatyczne**. Niestatyczne pole klasy należy do określonego obiektu. Na przykład klasa `CellPhone` ma trzy pola: `manufacturer`, `modelNumber` i `retailPrice`. Są to pola niestatyczne, a zatem nie istnieją w pamięci, dopóki nie zostanie utworzona instancja klasy `CellPhone`. Po utworzeniu tej instancji tworzone są jej pola. Ponieważ niestatyczne pola należą do określonej instancji klasy, są one zwykle określane jako **pola instancji**.

Omówmy teraz różnicę między metodami statycznymi i niestatycznymi. Metody statyczne, takie jak metoda `main`, którą dotychczas pisaliśmy w każdym programie, są metodami ogólnego przeznaczenia, które można po prostu wywołać w programie. Są zdefiniowane w klasie, ale aby z nich skorzystać, wcale nie musimy tworzyć instancji klasy. Natomiast metody niestatyczne są przeznaczone do pracy na instancji klasy, w której są zdefiniowane. Z tego powodu metody niestatyczne są zwykle nazywane **metodami instancji**. Aby wywołać metodę instancji, musi istnieć instancja klasy, w której deklarowana jest ta metoda.

Na przykład wszystkie metody zadeklarowane w klasie `CellPhone` są metodami instancji. Mają one działać na danych należących do instancji klasy `CellPhone`. Zanim będziemy mogli wywołać któryś z tych metod, musimy utworzyć instancję klasy `CellPhone` (innymi słowy, musimy stworzyć obiekt typu `CellPhone`). Następnie możemy wywołać metodę `setManufacturer`, aby wprowadzić nazwę producenta dla tego obiektu. Możemy wywołać metodę `setModelNumber` w celu ustawienia numeru modelu dla tego obiektu. Możemy również wywołać metodę `setRetailPrice`, aby ustawić cenę detaliczną dla tego obiektu. Podobnie możemy też wywołać metody `getManufacturer`, `getModelNumber` i `getRetail`, aby uzyskać nazwę producenta, numer modelu i cenę detaliczną danego obiektu.

Poniższy kod demonstruje sposób tworzenia instancji klasy `CellPhone` i użycia metod klasy do przechowywania danych w obiekcie:

```

// Deklarujemy zmienną referencyjną CellPhone
CellPhone myPhone;

// Tworzymy obiekt CellPhone
myPhone = new CellPhone();
```

```
// W polach obiektu zapisujemy wartości
myPhone.setManufacturer("Motorola");
myPhone.setModelNumber("MOTO G5");
myPhone.setRetailPrice(599.99);

// Wyświetlamy wartości zapisane w polach obiektu
System.out.println(myPhone.getManufacturer());
System.out.println(myPhone.getModelNumber());
System.out.println(myPhone.getRetailPrice());
```

### Konstruktory w języku Java

Konstruktor klasy w Javie to metoda mająca taką samą nazwę jak klasa. Na listingu klasy 14.14 znajduje się wersja klasy CellPhone, która została wyposażona w konstruktor. Jest to wersja programu w pseudokodzie z listingu klasy 14.4 napisana w języku Java. Konstruktor pojawia się wierszach od 9. do 14.

#### **Listing klasy 14.14 (CellPhone.java)**

```
1  public class CellPhone
2  {
3      //Deklarujemy pola
4      private String manufacturer;
5      private String modelNumber;
6      private double retailPrice;
7
8      //Konstruktor
9      public CellPhone(String manufact, String modNum, double retail)
10     {
11         manufacturer = manufact;
12         modelNumber = modNum;
13         retailPrice = retail;
14     }
15
16     //Mutatory
17     public void setManufacturer(String manufact)
18     {
19         manufacturer = manufact;
20     }
21
22     public void setModelNumber(String modNum)
23     {
24         modelNumber = modNum;
25     }
26
27     public void setRetailPrice(double retail)
28     {
29         retailPrice = retail;
30     }
31
32     //Akcesory
33     public String getManufacturer()
34     {
35         return manufacturer;
36     }
37
38     public String getModelNumber()
39     {
```

```

40     return modelNumber;
41 }
42
43 public double getRetailPrice()
44 {
45     return retailPrice;
46 }
47 }
```

Poniższy kod demonstruje sposób tworzenia instancji klasy poprzez przekazanie argumentów do konstruktora. Kod ten tworzy instancję klasy `CellPhone`, przekazując do konstruktora argumenty "Motorola", "MOTO G5" i 599.99:

```
CellPhone myPhone;
myPhone = new CellPhone("Motorola", "MOTO G5", 599.99);
```

### Dziedziczenie w języku Java

W Javie używamy słowa kluczowego `extends`, aby zaznaczyć, że klasa jest podklassą innej klasy. Na przykład spójrz na następujące deklaracje klas:

```

public class ClassA
{
    Deklarujemy pola i metody
}
public class ClassB extends ClassA
{
    Deklarujemy pola i metody
}
```

Nagłówek klasy `ClassB` wskazuje, że klasa `ClassB` rozszerza klasę `ClassA`. Tak więc klasa `ClassA` jest nadklassą, a klasa `ClassB` jest podklassą. Klasa `ClassB` dziedziczy wszystkich publicznych członków klasy `ClassA`.

### Polimorfizm w języku Java

Wcześniej w tym rozdziale przedstawiłem program demonstrujący polimorfizm, który jako nadklasy używał pseudokodu klasy `Animal` z listingu klasy 14.10. Wersja tej klasy dla języka Java została pokazana na listingu klasy 14.15.

#### **Listing klasy 14.15 (Animal.java)**

```

1 public class Animal
2 {
3     //Metoda showSpecies
4     public void showSpecies()
5     {
6         System.out.println("Jestem zwierzakiem.");
7     }
8
9     //Metoda makeSound
10    public void makeSound()
11    {
```

```

12     System.out.println("Grrrrr");
13 }
14 }
```

Pseudokod dla klasy Dog, który rozszerza klasę Animal, został pokazany na listingu klasy 14.11. Wersję klasy Dog napisaną w Javie przedstawiam listingu klasy 14.16.

#### **Listing klasy 14.16 (Dog.java)**

```

1 public class Dog extends Animal
2 {
3     // Metoda showSpecies
4     public void showSpecies()
5     {
6         System.out.println("Jestem psem.");
7     }
8
9     // Metoda makeSound
10    public void makeSound()
11    {
12        System.out.println("Hau! Hau!");
13    }
14 }
```

Pseudokod dla klasy Cat, który rozszerza klasę Animal, został pokazany na listingu klasy 14.12. Wersję klasy Cat napisaną w Javie prezentuję listingu klasy 14.17.

#### **Listing klasy 14.17 (Cat.java)**

```

1 public class Cat extends Animal
2 {
3     // Metoda showSpecies
4     public void showSpecies()
5     {
6         System.out.println("Jestem kotem.");
7     }
8
9     // Metoda makeSound
10    public void makeSound()
11    {
12        System.out.println("Miau.");
13    }
14 }
```

Na listingu 14.7 został przedstawiony program demonstrujący polimorficzne zachowanie klas, o którym mówiliśmy już wcześniej w tej książce. Jest to wersja pseudokodowego programu z listingu 14.6 napisana w języku Java.

**Listing 14.7 (PolymorphismDemo.java)**

```

1  public class PolymorphismDemo
2  {
3      public static void main(String[] args)
4      {
5          // Deklarujemy trzy zmienne obiektowe
6          Animal myAnimal;
7          Dog myDog;
8          Cat myCat;
9
10         // Tworzymy obiekty typu Animal:
11         // Dog i Cat
12         myAnimal = new Animal();
13         myDog = new Dog();
14         myCat = new Cat();
15
16         // Wyświetlamy informacje dotyczące zwierzęcia
17         System.out.println("Oto informacje dotyczące zwierzęcia:");
18         showAnimalInfo(myAnimal);
19         System.out.println();
20
21         // Wyświetlamy informacje dotyczące psa
22         System.out.println("Oto informacje dotyczące psa:");
23         showAnimalInfo(myDog);
24         System.out.println();
25
26         // Wyświetlamy informacje dotyczące kota
27         System.out.println("Oto informacje dotyczące kota:");
28         showAnimalInfo(myCat);
29     }
30
31     // Moduł showAnimalInfo przyjmuje jako argument
32     // obiekt typu Animal i wyświetla
33     // dotyczącego informacje
34     public static void showAnimalInfo(Animal creature)
35     {
36         creature.showSpecies();
37         creature.makeSound();
38     }
39 }
```

**Wynik działania programu**

Oto informacje dotyczące zwierzęcia:  
Jestem zwierzakiem.

Grrrrr

Oto informacje dotyczące psa:  
Jestem psem.  
Hau! Hau!

Oto informacje dotyczące kota:  
Jestem kotem.  
Miau.

## Python

### Klasy i obiekty

#### Deklaracje klas w języku Python

Oto ogólny format deklaracji klasy w Pythonie:

```
class NazwaKlasy:
    Definicje metod znajdują tutaj...
```

Pierwszy wiersz deklaracji klasy rozpoczyna się od słowa kluczowego `class`, po którym następuje nazwa klasy, a następnie dwukropki. W kolejnych wierszach umieszczane są definicje metod klasy. Schemat ich zapisu jest podobny do zwykłych definicji funkcji. Definicje metod muszą być zapisane z wcięciem, ponieważ należą do danej klasy.

Podstawową różnicą, którą można zauważyci między deklaracjami klas w Pythonie a deklaracjami klas pseudokodowych w tej książce, jest brak deklaracji pól w klasie Pythona. Dzieje się tak dlatego, że pola obiektu są tworzone przez instrukcje przypisania, które pojawiają się w metodach danej klasy.

Kolejną różnicą, którą można zauważyci, jest brak specyfikatorów dostępu, takich jak `Private` i `Public`. W Pythonie pole lub metodę można ukryć, rozpoczynając zapis ich nazw od dwóch znaków podkreśleń. Jest to podobne do definiowania pola lub metody jako prywatnych.

Poniższy program został napisany w Pythonie i zawiera klasę `CellPhone`, podobną do klasy zawartej w pseudokodzie z listingu klasy 14.3. Ma on również metodę `main`, za pomocą której demonstrujemy działanie klasy, podobnie jak to pokazano w pseudokodzie z listingu 14.3.

#### **Listing 14.8**

```
1 class CellPhone:
2     def set_manufacturer(self, manufact):
3         self.__manufacturer = manufact
4
5     def set_model_number(self, model):
6         self.__model_number = model
7
8     def set_retail_price(self, retail):
9         self.__retail_price = retail
10
11    def get_manufacturer(self):
12        return self.__manufacturer
13
14    def get_model_number(self):
15        return self.__model_number
16
17    def get_retail_price(self):
18        return self.__retail_price
19
20 def main():
21     # Tworzymy obiekt CellPhone, który
22     # zapisujemy w zmiennej phone
```

```

23     phone = CellPhone()
24
25     # W polach obiektu zapisujemy wartości
26     phone.set_manufacturer("Motorola")
27     phone.set_model_number("MOTO G5")
28     phone.set_retail_price(599.99)
29
30     # Wyświetlamy wartości zapisane w polach
31     print('Producent telefonu:', phone.get_manufacturer())
32     print('Model telefonu:', phone.get_model_number())
33     print('Cena detaliczna telefonu:', phone.get_retail_price())
34
35 # Wywołujemy funkcję main
36 main()

```

### **Wynik działania programu**

Producent telefonu: Motorola  
 Model telefonu: MOTO G5  
 Cena detaliczna telefonu: 599.99 zł

Zauważ, że każda metoda ma parametr o nazwie `self`. Jest on wymagany w każdej metodzie należącej do klasy. Metoda działa na atrybutach danych określonego obiektu. Kiedy metoda jest wykonywana, musi mieć sposób na poznanie atrybutów danych obiektu, z którymi ma pracować. W takim właśnie celu pojawia się parametr `self`. Podczas wywoływanego metody Python automatycznie powiąże parametr `self` z konkretnym obiektem, na którym ma działać dana metoda.

Spójrzmy teraz na metodę `set_manufacturer`, znajdująca się wierszach 2. i 3. Oprócz parametru `self` ma ona parametr `manufact`. Instrukcja znajdująca się wierszu 3. przypisuje parametr `manufact` do `self .__manufacturer`. Czym jednak jest `self .__manufacturer`?

Przeanalizujmy to szczegółowo:

- element `self` odnosi się do określonego obiektu `CellPhone` znajdującego się w pamięci;
- element `manufacturer` jest nazwą pola; dwa podkreślenia na początku nazwy pola sprawiają, że staje się on prywatny, czyli niewidoczny dla kodu spoza klasy `CellPhone`.

Zatem instrukcja w wierszu 3. przypisuje wartość parametru `manufact` do pola `__manufacturer` obiektu `CellPhone`.

Natomiast metoda `set_model_number`, w wierszach 5. i 6., jest podobna. Pobiera ona parametr `model` i przypisuje go do pola `__model_number` obiektu.

Także metoda `set_retail_price`, w wierszach 8. i 9., działa na podobnej zasadzie. Przyjmuje parametr `retail` i przypisuje go do pola `__retail_price` obiektu.

Metoda `get_manufacturer`, w wierszach 11. i 12., zwraca wartość pola `__manufacturer` obiektu. Metoda `get_model_number`, w wierszach 14. i 15., zwraca wartość pola `__model_number` obiektu. Metoda `get_retail_price`, w wierszach 17. i 18., zwraca pole `__retail_price` obiektu.

Wewnątrz funkcji `main` wiersz 23. tworzy w pamięci instancję klasy `CellPhone` i przypisuje ją do zmiennej `phone`. W takim przypadku mówimy, że do obiektu odwołuje się zmienna `phone`. (Zwrót uwagę, że Python nie wymaga stosowania słowa kluczowego `New`, jak już wspomniałem wcześniej w tym rozdziale). Następnie wiersze od 26. do 28. wywołują po kolei metody obiektu `set_manufacturer`, `set_model_number` i `set_retail_price`, przekazując do każdej z nich odpowiednie argumenty.

Przypomnijmy, że w klasie `CellPhone` metody `set_manufacturer`, `set_model_number` i `set_retail_price` mają dwa parametry. Jednakże gdy wywołujemy te metody w wierszach od 26. do 28., przekazujemy im tylko jeden argument. Pierwszym parametrem w każdej z tych metod jest `self`. Kiedy wywołujemy daną metodę, nie przekazujemy argumentu dla parametru `self`, ponieważ Python automatycznie przekazuje do pierwszego parametru metodę referencję na obiekt wywołującego. Dzięki temu parametr `self` automatycznie będzie wskazywał obiekt, w którym ma działać dana metoda.

A oznacza to, że:

- w wierszu 26. argument "Motorola" jest przekazywany do parametru `manufact` metody `set_manufacturer`;
- w wierszu 27. argument "MOTO G5" jest przekazywany do parametru `model` metody `set_model_number`;
- w wierszu 28. argument 599,99 jest przekazywany do parametru `retail` metody `set_retail_price`.

Wiersze od 31. do 33. wywołują funkcję `print`, aby wyświetlić wartości pól obiektu.

### Konstruktory w języku Python

W Pythonie klasy mogą mieć metodę o nazwie `__init__`, która jest wykonywana automatycznie, gdy tworzona jest w pamięci instancja klasy. Metoda `__init__` jest powszechnie znana jako **metoda inicjalizująca**, ponieważ inicjalizuje atrybuty danych obiektu. (Nazwa metody zaczyna się od dwóch znaków podkreślenia, po których następuje słowo `init`, a potem dwa kolejne znaki podkreślenia).

Program przedstawiony na listingu 14.9 pokazuje wersję klasy `CellPhone`, która otrzymała metodę `__init__`. Jest to wersja pseudokodu z listingu klasy 14.4 połączonego z pseudokodem z listingu 14.2 napisana w języku Python.

#### **Listing 14.9**

```

1 class CellPhone:
2     def __init__(self, manufact, model, retail):
3         self.__manufacturer = manufact
4         self.__model_number = model
5         self.__retail_price = retail
6
7     def set_manufacturer(self, manufact):
8         self.__manufacturer = manufact
9
10    def set_model_number(self, model):
11        self.__model_number = model
12
13    def set_retail_price(self, retail):
```

```

14         self.__retail_price = retail
15
16     def get_manufacturer(self):
17         return self.__manufacturer
18
19     def get_model_number(self):
20         return self.__model_number
21
22     def get_retail_price(self):
23         return self.__retail_price
24
25 def main():
26     # Tworzymy obiekt CellPhone i inicjalizujemy jego
27     # pola wartościami przekazanymi do metody __init__
28     phone = CellPhone("Motorola", "MOTO G5", 599.99)
29
30     # Wyświetlamy wartości zapisane w polach
31     print('Producent telefonu:', phone.get_manufacturer())
32     print('Model telefonu:', phone.get_model_number())
33     print('Cena detaliczna telefonu:', phone.get_retail_price())
34
35 # Wywołujemy funkcję main
36 main()

```

### **Wynik działania programu**

Producent telefonu: Motorola  
 Model telefonu: MOTO G5  
 Cena detaliczna telefonu: 599.99 zł

Instrukcja w wierszu 28. tworzy obiekt `CellPhone` w pamięci i przypisuje go do zmiennej `phone`. Można tu zauważyć, że wartości "Motorola", "MOTO G5" i "599.99" pojawiają się w nawiasach po nazwie klasy. Wartości te są przekazywane jako argumenty dla metody `__init__` tej klasy.

### **Dziedziczenie w języku Python**

Załóżmy, że chcemy zadeklarować dwie klasy: `ClassA` i `ClassB`. Ponadto chcemy, aby klasa `ClassB` była podklassą klasy `ClassA`. Ogólny format deklaracji w Pythonie jest taki:

```

class ClassA:
    Deklaracje pól i metod
class ClassB(ClassA):
    Deklaracje pól i metod

```

W nagłówku klasy `ClassB` napisaliśmy w nawiasach `ClassA`. Oznacza to, że klasa `ClassB` rozszerza klasę `ClassA`. Tak więc klasa `ClassA` jest nadklassą, a klasa `ClassB` jest podklassą. Klasa `ClassB` dziedziczy wszystkie publiczne składowe klasy `ClassA`.

### **Polimorfizm w języku Python**

W tym rozdziale przedstawiłem już zasady polimorfizmu, wykorzystując do tego klasę `Animal` (listing klasy 14.10) jako nadklassę oraz klasę `Dog` (listing klasy 14.11) i klasę `Cat` (listing klasy 14.12) jako podklasy klasy `Animal`. Wersje tych klas napisane

w Pythonie pokazane są na listingu 14.10. Funkcja `main` i funkcje `show_animal_info` są odpowiednikiem programu pseudokodowego z listingu 14.6 napisanym w języku Python.

### **Listing 14.10 (polymorphism.py)**

```

1  class Animal:
2      def show_species(self):
3          print("Jestem zwierzakiem.")
4
5      def make_sound(self):
6          print('Grrrrr')
7
8  class Dog(Animal):
9      def show_species(self):
10         print("Jestem psem.")
11
12     def make_sound(self):
13         print('Hau! Hau!')
14
15 class Cat(Animal):
16     def show_species(self):
17         print('Jestem kotem.')
18
19     def make_sound(self):
20         print('Miau.')
21
22 # Funkcja main
23
24 def main():
25     # Tworzymy obiekt Animal obiektu Dog
26     # i obiektu Cat.
27     my_animal = Animal()
28     my_dog = Dog()
29     my_cat = Cat()
30
31     # Wyświetlamy informacje dotyczące zwierzęcia
32     print('Oto informacje dotyczące zwierzęcia:')
33     show_animal_info(my_animal)
34     print()
35
36     # Wyświetlamy informacje dotyczące psa
37     print('Oto informacje dotyczące psa:')
38     show_animal_info(my_dog)
39     print()
40
41     # Wyświetlamy informacje dotyczące kota
42     print('Oto informacje dotyczące kota:')
43     show_animal_info(my_cat)
44
45 # Funkcja show_animal_info przyjmuje obiekt
46 # Animal jako argument i wyświetla
47 # informacje na jego temat
48
49 def show_animal_info(creature):
50     creature.show_species()
51     creature.make_sound()
52

```

```
53 # Wywołujemy funkcję main
54 main()
```

### **Wynik działania programu**

Oto informacje dotyczące zwierzęcia:  
Jestem zwierzakiem.  
Grrrrr.

Oto informacje dotyczące psa:  
Jestem psem.  
Hau! Hau!

Oto informacje dotyczące kota:  
Jestem kotem.  
Miau.

## **C ++**

### Klasy i obiekty

#### **Deklaracje klas w języku C++**

Oto ogólny format deklaracji klasy w C++:

```
class NazwaKlasy
{
    Deklaracje pól i definicje funkcji składowych można znaleźć tutaj...
};
```

Jak można zauważyc, deklaracja klasy w języku C++ kończy się zawsze średnikiem.

Klasa CellPhone pokazana na listingu 14.11 jest wersją programu pseudokodowego z listingu klasy 14.3 przygotowaną w C++. Zwróć uwagę, że w wierszu 7. znajduje się tekst `private`:, natomiast w wierszu 9. znajduje się tekst `public`:. Są to **specyfikatory dostępu**, które kontrolują, w jaki sposób kod spoza klasy może uzyskać dostęp do poszczególnych pól i funkcji składowych klasy. Wszystkie deklaracje pola pojawiające się po parametrze `private`: w wierszu 7. są prywatne. Dostęp do nich może uzyskać tylko kod znajdujący się wewnątrz klasy. Wszystkie funkcje składowe po specyfikatorze dostępu `public`: w wierszu 13. są publiczne i mogą być wywoływanie przez kod spoza klasy.

Program znajdujący się na listingu 14.11 zawiera także funkcję `main`, która demonstruje działanie klasy i jest podobna do zaprezentowanej wcześniej funkcji w programie pseudokodowym z listingu 14.3.

#### **Listing 14.11 (CellPhoneDemo.cpp)**

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class CellPhone
```

```
6 {
7     private:
8         //Deklarujemy pola
9         string manufacturer;
10        string modelNumber;
11        double retailPrice;
12
13    public:
14        //Funkcje składowe
15        void setManufacturer(string manufact)
16        {
17            manufacturer = manufact;
18        }
19
20        void setModelNumber(string modNum)
21        {
22            modelNumber = modNum;
23        }
24
25        void setRetailPrice(double retail)
26        {
27            retailPrice = retail;
28        }
29
30        string getManufacturer()
31        {
32            return manufacturer;
33        }
34
35        string getModelNumber()
36        {
37            return modelNumber;
38        }
39
40        double getRetailPrice()
41        {
42            return retailPrice;
43        }
44    };
45
46 int main()
47 {
48     //Deklarujemy zmienne, w której
49     //znajdzie się obiekt CellPhone
50     CellPhone myPhone;
51
52     //Zapisujemy wartości w polach obiektu
53     myPhone.setManufacturer("Motorola");
54     myPhone.setModelNumber("MOTO G5");
55     myPhone.setRetailPrice(599.99);
56
57     //Wyświetlamy wartości zapisane w polach
58     cout << "Producent telefonu:"
59         << myPhone.getManufacturer() << endl;
60     cout << "Model telefonu:"
61         << myPhone.getModelNumber() << endl;
62     cout << "Cena detaliczna telefonu:"
63         << myPhone.getRetailPrice() << " zł" << endl;
64 }
```

```

65     return 0;
66 }
```

### **Wynik działania programu**

Producent telefonu: Motorola

Model telefonu: MOTO G5

Cena detaliczna telefonu: 599.99 zł

W funkcji `main`, w wierszu 50., tworzona jest w pamięci instancja klasy `CellPhone` i przypisywana jest do zmiennej `myPhone`. Mówimy, że zmienna `myPhone` odwołuje się do obiektu. (Zauważ, że język C++ nie wymaga słowa kluczowego `New`, o czym pisalem już wcześniej w tym rozdziale). W wierszach od 53. do 55. wywoływane są funkcje składowe obiektu `setManufacturer`, `setModelNumber` oraz `setRetailPrice`, a każdej z nich przekazywane są odpowiednie argumenty.

### **Konstruktory w języku C++**

Konstruktor klasy w C++ jest funkcją składową, która ma taką samą nazwę jak klasa. Program z listingu 14.12 prezentuje wersję klasy `CellPhone`, która została wyposażona w konstruktor. Jest to wersja programu pseudokodowego z listingu klasy 14.4 połączonego z pseudokodem z listingu 14.2 napisana w języku C++. Konstruktor pojawia się w wierszach od 9. do 14.

#### **Listing 14.12 (ConstructorDemo.cpp)**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class CellPhone
6 {
7 private:
8     //Deklarujemy pola
9     string manufacturer;
10    string modelNumber;
11    double retailPrice;
12
13 public:
14     //Konstruktor
15     CellPhone(string manufact, string modNum, double retail)
16     {
17         manufacturer = manufact;
18         modelNumber = modNum;
19         retailPrice = retail;
20     }
21
22     //Funkcje składowe
23     void setManufacturer(string manufact)
24     {
25         manufacturer = manufact;
26     }
27
28     void setModelNumber(string modNum)
29     {
```

```

30     modelNumber = modNum;
31 }
32
33 void setRetailPrice(double retail)
34 {
35     retailPrice = retail;
36 }
37
38 string getManufacturer()
39 {
40     return manufacturer;
41 }
42
43 string getModelNumber()
44 {
45     return modelNumber;
46 }
47
48 double getRetailPrice()
49 {
50     return retailPrice;
51 }
52 };
53
54 int main()
55 {
56     // Tworzymy obiekt CellPhone i inicjalizujemy jego
57     // pola wartościami przekazanymi do konstruktora
58     CellPhone myPhone("Motorola", "MOTO G5", 599.99);
59
60     // Wyświetlamy wartości zapisane w polach
61     cout << "Producent telefonu: "
62         << myPhone.getManufacturer() << endl;
63     cout << "Model telefonu: "
64         << myPhone.getModelNumber() << endl;
65     cout << "Cena detaliczna telefonu: "
66         << myPhone.getRetailPrice() << endl;
67
68     return 0;
69 }
```

### **Wynik działania programu**

Producent telefonu: Motorola  
 Model telefonu: MOTO G5  
 Cena detaliczna telefonu: 599.99 zł

### **Dziedziczenie w języku C++**

Załóżmy, że chcemy zadeklarować dwie klasy: ClassA i ClassB. Ponadto chcemy, aby klasa ClassB była podklassą klasy ClassA. Ogólny format deklaracji klasy w C++ to:

```

class ClassA
{
    Deklaracje elementów składowych
};

class ClassB: public ClassA
{
    Deklaracje elementów składowych
};
```

Nagłówek klasy ClassB kończy się następującą klauzulą:

```
: public ClassA
```

Informuje nas to, że klasa ClassB rozszerza klasę ClassA. Tak więc ClassA jest nadklasą, a klasa ClassB jest podklasą. Klasa ClassB dziedziczy wszystkie publiczne składowe klasy ClassA.

### Polimorfizm w języku C++

W tym rozdziale przedstawiłem już zasady polimorfizmu, wykorzystując do tego klasę Animal (listing klasy 14.10) jako nadkласę oraz klasę Dog (listing klasy 14.11) i klasę Cat (listing klasy 14.12) jako podklasy klasy Animal. Wersje tych klas napisane w C++ pokazane są na listingu 14.13. Funkcja main i funkcje showAnimalInfo są napisaną w C++ wersją programu pseudokodowego z listingu 14.6.

Zauważ, że w nagłówkach funkcji showSpecies i makeSound pojawia się słowo kluczowe virtual (wiersze 9., 15., 25., 31., 40. i 46.). Sprawia ono, że kompilator oczekuje ponownego zdefiniowania danej funkcji w podklasie.

Zwróć też uwagę na fakt, że w wierszu 82. funkcja showAnimalInfo przyjmuje przez referencję obiekt typu Animal. W C++ zachowanie polimorficzne jest możliwe tylko wtedy, gdy obiekt jest przekazywany przez referencję.

#### **Listing 14.13 (Polymorphism.cpp)**

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Animal
6 {
7 public:
8     //Funkcja showSpecies
9     virtual void showSpecies()
10    {
11        cout << "Jestem zwierzakiem." << endl;
12    }
13
14    //Funkcja makeSound
15    virtual void makeSound()
16    {
17        cout << "Grrrrr" << endl;
18    }
19 };
20
21 class Dog : public Animal
22 {
23 public:
24     //Funkcja showSpecies
25     virtual void showSpecies()
26    {
27        cout << "Jestem psem." << endl;
28    }
29

```

```
30 //Funkcja makeSound
31 virtual void makeSound()
32 {
33     cout << "Hau! Hau!" << endl;
34 }
35 };
36
37 class Cat : public Animal
38 {
39 //Funkcja showSpecies
40 virtual void showSpecies()
41 {
42     cout << "Jestem kotem." << endl;
43 }
44
45 //Funkcja makeSound
46 virtual void makeSound()
47 {
48     cout << "Miau." << endl;
49 }
50 };
51
52 //Prototyp funkcji
53 void showAnimalInfo(Animal &creature);
54
55 int main()
56 {
57     //Deklarujemy trzy zmienne klasy
58     Animal myAnimal;
59     Dog myDog;
60     Cat myCat;
61
62     //Wyświetlamy informacje dotyczące zwierzęcia
63     cout << "Oto informacje dotyczące zwierzęcia:" << endl;
64     showAnimalInfo(myAnimal);
65     cout << endl;
66
67     //Wyświetlamy informacje dotyczące psa
68     cout << "Oto informacje dotyczące psa:" << endl;
69     showAnimalInfo(myDog);
70     cout << endl;
71
72     //Wyświetlamy informacje dotyczące kota
73     cout << "Oto informacje dotyczące kota:" << endl;
74     showAnimalInfo(myCat);
75
76     return 0;
77 }
78
79 //Funkcja showAnimalInfo przyjmuje obiekt
80 //typu Animal jako argument i wyświetla
81 //informacje na jego temat
82 void showAnimalInfo(Animal &creature)
83 {
84     creature.showSpecies();
85     creature.makeSound();
86 }
```

**Wynik działania programu**

Oto informacje dotyczące zwierzęcia:  
 Jestem zwierzakiem.  
 Grrrrr.

Oto informacje dotyczące psa:  
 Jestem psem.  
 Hau! Hau!

Oto informacje dotyczące kota:  
 Jestem kotem.  
 Miau.

**Pytania kontrolne****Test jednokrotnego wyboru**

1. Programowanie \_\_\_\_\_ polega na tworzeniu modułów i funkcji, które są oddzielone od przetwarzanych przez nie danych.
  - a) modularne
  - b) proceduralne
  - c) funkcyjne
  - d) obiektowe
2. Programowanie \_\_\_\_\_ skupia się na tworzeniu obiektów.
  - a) obiektocentryczne
  - b) obiektywne
  - c) proceduralne
  - d) obiektywne
3. \_\_\_\_\_ to składowa klasy, w której zapisane są dane.
  - a) metoda
  - b) instancja
  - c) pole
  - d) konstruktor
4. \_\_\_\_\_ informuje, czy kod znajdujący się na zewnątrz klasy będzie miał dostęp do danej składowej.
  - a) deklaracja pola
  - b) słowo kluczowe New
  - c) modyfikator dostępu
  - d) konstruktor
5. Pola klasy oznacza się zazwyczaj modyfikatorem dostępu \_\_\_\_\_.
  - a) Private
  - b) Public
  - c) ReadOnly
  - d) Hidden
6. Zmienna \_\_\_\_\_ to zmienna, za pomocą której można się odnosić do obiektu znajdującego się w pamięci komputera.

- a) pamięciowa
  - b) proceduralna
  - c) obiektowa
  - d) dynamiczna
7. W wielu językach programowania obiekt w pamięci tworzy się za pomocą słowa kluczowego \_\_\_\_\_.  
a) Create  
b) New  
c) Instantiate  
d) Declare
8. \_\_\_\_\_ zwraca wartość pola w danej klasie, ale nie może zmienić tej wartości.  
a) apporter  
b) konstruktor  
c) mutator  
d) akcesor
9. \_\_\_\_\_ zapisuje lub modyfikuje wartość pola w danej klasie.  
a) modyfikator  
b) konstruktor  
c) mutator  
d) akcesor
10. \_\_\_\_\_ jest wywoływany automatycznie w momencie tworzenia obiektu.  
a) akcesor  
b) konstruktor  
c) setter  
d) mutator
11. Za pomocą \_\_\_\_\_, korzystając z diagramów, można graficznie opisać system obiektowy.  
a) języka UML  
b) schematów blokowych  
c) pseudokodu  
d) obiektowego systemu hierarchicznego
12. Dane \_\_\_\_\_ to dane, które powstają w momencie, gdy jedna wartość zależna o drugiej wartości nie zostanie zaktualizowana, kiedy zmianie ulegnie druga wartość.  
a) nieadekwatne  
b) nieaktualne  
c) asynchroniczne  
d) przestarzałe
13. Zakres obowiązków klasy to \_\_\_\_\_.  
a) tworzenie obiektów na jej podstawie  
b) rzeczy, które klasa musi zapamiętać  
c) operacje, które klasa musi wykonywać  
d) zarówno b., jak i c.
14. W przypadku dziedziczenia klasę, po której dziedziczy inna klasa, nazywamy \_\_\_\_\_.  
\_\_\_\_\_.

- a) klasą pochodną
  - b) klasą bazową
  - c) klasą zależną
  - d) klasą podzieloną
15. W przypadku dziedziczenia klasę, która jest szczególnym przypadkiem innej klasy, nazywamy \_\_\_\_\_.  
 a) klasą bazową  
 b) klasą główną  
 c) klasą pochodną  
 d) klasą nadzczną
16. Dzięki \_\_\_\_\_ za pomocą zmiennej typu klasy bazowej można odnosić się do obiektu typu klasy pochodnej.  
 a) polimorfizmowi  
 b) dziedziczeniu  
 c) uogólnieniu  
 d) specjalizacji

### Prawda czy fałsz?

1. Programowanie proceduralne polega w głównej mierze na tworzeniu obiektów.
2. Wielokrotne wykorzystanie obiektów to jedna z cech, które przyczyniły się do popularyzacji programowania obiektowego.
3. W przypadku programowania obiektowego wszystkie pola deklaruje się najczęściej jako publiczne.
4. Jedna z technik określania klas potrzebnych w danym programie polega na sporządzeniu listy wszystkich czasowników występujących w opisie modelu dziedziny.
5. Klasa bazowa dziedziczy po klasie pochodnej pola i metody.
6. Dzięki polimorfizmowi można za pomocą zmiennej typu klasy bazowej odnosić się zarówno do obiektu typu klasy bazowej, jak i klasy pochodnej.

### Krótką odpowiedź

1. Co to jest hermetyzacja?
2. Dlaczego dane zapisane w obiekcie ukrywa się zazwyczaj przed kodem znajdującym się na zewnątrz klasy?
3. Wyjaśnij różnicę między klasą a instancją klasy.
4. Do czego służy słowo kluczowe `New` w większości języków programowania?
5. Za pomocą poniższego polecenia pseudokodu wywołuję metodę obiektu. Jak nazywa się ta metoda? Jak nazywa się zmienna, która odnosi się do tego obiektu?  
`Call wallet.getDollar()`
6. Co to są dane nieaktualne?
7. Co dziedziczy po klasie bazowej klasa pochodna?

8. Spójrz na poniższy pseudokod, będący pierwszą linią definicji klasy. Jaką nazwę ma w tym przypadku klasa bazowa? Jaką nazwę ma klasa pochodna?

```
Call Tiger Extends Felis
```

### **Warsztat projektanta algorytmów**

- Załóżmy, że zmienna obiektowa myCar odnosi się do pewnego obiektu, który ma metodę o nazwie go. Metoda go nie przyjmuje żadnego argumentu. Zapisz za pomocą pseudokodu polecenie, za pomocą którego wywołasz metodę go obiektu myCar.
- Spójrz na poniższy fragment definicji klasy i wykonaj zadania wymienione pod definicją.

```
Class Book
    Private String title
    Private String author
    Private String publisher
    Private Integer copiesSold
End Class
```

- a) Zdefiniuj konstruktor klasy. Powinien on przyjmować argumenty odpowiadające każdemu polu klasy.
- b) Zdefiniuj akcesory i mutatory dla każdego pola.
- c) Narysuj diagram UML klasy zawierający zdefiniowane metody.

3. Spójrz na poniższy opis modelu dziedziny:

Bank oferuje swoim klientom następujące typy rachunków: rachunek oszczędnościowy, rachunek bieżący i rachunek inwestycyjny. Klient może wpłacać środki na rachunek (zwiększać saldo), wypłacać środki z rachunku (zmniejszać saldo) i uzyskiwać na rachunku odsetki. Każdy z rachunków charakteryzuje się innym oprocentowaniem.

Załóżmy, że tworzysz program, który będzie służył do obliczania odsetek uzyskanych na rachunku bankowym.

- a) Stwórz listę potencjalnych klas.
  - b) Zawęź listę klas tylko do tej klasy (lub klas), która będzie potrzebna w zadaniu.
  - c) Określ zakres obowiązków klasy (lub klas).
4. Zapisz za pomocą pseudokodu pierwszą linię definicji klasy Poodle. Powinna ona być rozwinięciem klasy Dog.
5. Spójrz na poniższą definicję klasy:

```
Class Plant
    Public Module message()
        Display "Jestem rośliną."
    End Module
End Class
Class Tree Extends Plant
    Public Module message()
        Display "Jestem drzewem."
    End Module
End Class
```

Co wyświetli się po wykonaniu poniższego pseudokodu?

```
Declare Plant p
Set p = New Tree()
Call p.message()
```

## Ćwiczenia programistyczne

### 1. Klasa Pet

Zaprojektuj klasę o nazwie `Pet`, która będzie miała następujące pola:

- `name` — będzie tu zapisane imię zwierzęcia;
- `type` — będzie tu zapisany rodzaj zwierzęcia (mogą to być na przykład wartości "Kot", "Pies" albo "Ptak");
- `age` — będzie tu zapisany wiek zwierzęcia.

Klasa `Pet` powinna mieć także następujące metody:

- `setName` — zapisuje wartość w polu `name`;
- `setType` — zapisuje wartość w polu `type`;
- `setAge` — zapisuje wartość w polu `age`;
- `getName` — zwraca wartość w polu `name`;
- `getType` — zwraca wartość w polu `type`;
- `getAge` — zwraca wartość w polu `age`.

Gdy zaprojektujesz klasę, zaprojektuj program, w którym utworzysz obiekt tej klasy i poprosisz użytkownika o wprowadzenie imienia, rodzaju i wieku jego ulubionego zwierzaka. Wprowadzone dane zapisz w obiekcie. Następnie pobierz za pomocą akcesorów imię, rodzaj i wiek zwierzaka i wyświetl je na ekranie.

### 2. Klasa Car

Zaprojektuj klasę o nazwie `Car`, która będzie miała następujące pola:

- `yearModel` — pole typu `Integer` zawierające rok produkcji samochodu;
- `make` — pole typu `String` zawierające markę samochodu;
- `speed` — pole typu `Integer` zawierające bieżącą prędkość samochodu.

Ponadto klasa powinna mieć konstruktor i inne metody:

- Konstruktor — powinien przyjmować jako argumenty rok produkcji i markę samochodu. Wartości te należy zapisać w polach `yearModel` i `make`. Konstruktor powinien także przypisać do pola `speed` wartość 0.
- Akcesory — zaprojektuj odpowiednie metody, za pomocą których będzie można pobierać wartości zapisane w polach `yearModel`, `make` i `speed`.
- `accelerate` — metoda ta po każdym wywołaniu powinna dodawać do pola `speed` wartość 5.
- `brake` — metoda ta po każdym wywołaniu powinna odejmować od pola `speed` wartość 5.

Następnie zaprojektuj program, w którym utworzysz obiekt typu `Car` i pięciokrotnie wywołasz metodę `accelerate`. Po każdym wywołaniu metody `accelerate` pobierz i wyświetl na ekranie bieżącą prędkość samochodu.

Następnie pięciokrotnie wywołaj metodę `brake`. Po każdym wywołaniu metody `brake` pobierz i wyświetl na ekranie bieżącą prędkość samochodu.

### 3. Klasa PersonalInformation

Zaprojektuj klasę, w której będą zapisane następujące dane: imię i nazwisko, adres, wiek oraz numer telefonu. Stwórz odpowiednie akcesory i mutatory. Następnie zaprojektuj program, w którym utworzysz trzy instancje klasy. W jednej instancji zapisz swoje dane osobowe, a w pozostałych dwóch instancjach zapisz dane osobowe swoich przyjaciół lub członków rodziny.

### 4. Klasy Employee i ProductionWorker

Zaprojektuj klasę `Employee` wyposażoną w pola, w których można zapisać następujące informacje:

- nazwisko pracownika;
- identyfikator pracownika.

Następnie zaprojektuj klasę o nazwie `ProductionWorker`, która jest rozwinięciem klasy `Employee`. Klasa `ProductionWorker` powinna zawierać następujące pola:

- numer zmiany (liczba całkowita, np. 1, 2 lub 3);
- stawka godzinowa.

Dzień roboczy jest podzielony na dwie zmiany: dzienną i nocną. Pole „numer zmiany” będzie zawierało liczbę całkowitą oznaczającą, na której zmiana pracuje dany pracownik. Zmiana dzienna odpowiada liczbie 1, a zmiana nocna — liczbie 2. W każdej z klas zaprojektuj odpowiednie akcesory i mutatory.

Kiedy zaprojektujesz obie klasy, zaprojektuj program, w którym utworzysz obiekt klasy `ProductionWorker` i poprosisz użytkownika o wprowadzenie danych dla każdego pola. Zapisz wprowadzone wartości w obiekcie, a następnie, korzystając z metod, pobierz je i wyświetl na ekranie.

### 5. Klasa Essay

Zaprojektuj klasę `Essay` będącą rozwinięciem klasy `GradedActivity`, którą zaprezentowałem w tym rozdziale. Klasa `Essay` powinna określać ocenę ucznia z wypracowania. Wynik z wypracowania wynosi maksymalnie 100 punktów, a jest określany w następujący sposób:

- gramatyka: maksymalnie 30 punktów;
- pisownia: maksymalnie 20 punktów;
- długość: maksymalnie 20 punktów;
- treść: maksymalnie 30 punktów.

Kiedy zaprojektujesz klasę, zaprojektuj program, za pomocą którego poprosisz użytkownika o wprowadzenie wyniku uzyskanego przez ucznia za gramatykę, pisownię, długość i treść wypracowania. Utwórz obiekt typu `Essay` i zapisz w nim wprowadzone dane. Za pomocą metod obiektu oblicz wynik i ocenę, a następnie wyświetl te informacje na ekranie.

### 6. Klasa Patient

Zaprojektuj klasę o nazwie `Patient`, która zawiera pola dla następujących danych:

- imię, drugie imię, nazwisko;
- ulica, kod pocztowy, miejscowość, województwo;

- numer telefonu;
- nazwisko i numer telefonu osoby, którą należy powiadomić w razie pogorszenia się stanu zdrowia pacjenta.

Klasa Patient powinna mieć konstruktor przyjmujący argument dla każdego z wymienionych wyżej pól. Powinna też dla każdego pola udostępniać metody akcesora i mutatora.

Następnie napisz klasę o nazwie Procedure, reprezentującą procedurę medyczną zastosowaną u pacjenta. Klasa Procedure powinna zawierać pola dla następujących danych:

- nazwa procedury;
- data przeprowadzenia procedury;
- nazwisko lekarza, który przeprowadził procedurę;
- wysokość opłaty za przeprowadzoną procedurę.

Klasa Procedure powinna mieć konstruktor przyjmujący argument dla każdego z wymienionych wyżej pól. Powinna też dla każdego pola udostępniać metody akcesora i mutatora.

Następnie zaprojektuj program, który utworzy instancję klasy Patient zainicjalizowaną przykładowymi danymi, a potem utworzy trzy instancje klasy Procedure, również zainicjalizowane następującymi danymi:

| Procedura nr 1:                    | Procedura nr 2:         | Procedura nr 3:                  |
|------------------------------------|-------------------------|----------------------------------|
| Nazwa procedury:<br>Badanie ogólne | Nazwa procedury:<br>RTG | Nazwa procedury:<br>Badanie krwi |
| Data: dzisiejsza data              | Data: dzisiejsza data   | Data: dzisiejsza data            |
| Lekarz: Dr Irvine                  | Lekarz: Dr Jamison      | Lekarz: Dr Smith                 |
| Opłata: 250,00                     | Opłata: 500,00          | Opłata: 200,00                   |

Program powinien wyświetlać dane pacjenta oraz informacje o wszystkich trzech procedurach i całkowity ich koszt.

## TEMATYKA

- |                                                                         |                                                    |
|-------------------------------------------------------------------------|----------------------------------------------------|
| 15.1 Graficzny interfejs użytkownika                                    | 15.3 Tworzenie procedury obsługi zdarzenia         |
| 15.2 Projektowanie interfejsu użytkownika w programie wyposażonym w GUI | 15.4 Projektowanie aplikacji na urządzenia mobilne |
|                                                                         | 15.5 Rzut oka na języki Java, Python i C++         |

## 15.1

## Graficzny interfejs użytkownika

**WYJAŚNIENIE:** Graficzny interfejs użytkownika pozwala komunikować się z systemem operacyjnym i programami za pomocą elementów graficznych, takich jak ikony, przyciski czy okna dialogowe.

**Interfejs użytkownika** to ten element, za pomocą którego użytkownik komunikuje się z komputerem. Jednym ze składników interfejsu użytkownika są urządzenia, takie jak klawiatura czy monitor. Innym jest to, w jaki sposób komputer przyjmuje od użytkownika polecenia. Przez bardzo wiele lat jedynym sposobem komunikacji użytkownika z komputerem był **interfejs wiersza poleceń**, taki jak na rysunku 15.1. Interfejs ten zazwyczaj wyświetla znak zachęty — po nim użytkownik może wprowadzić polecenie, które następnie wykona komputer.



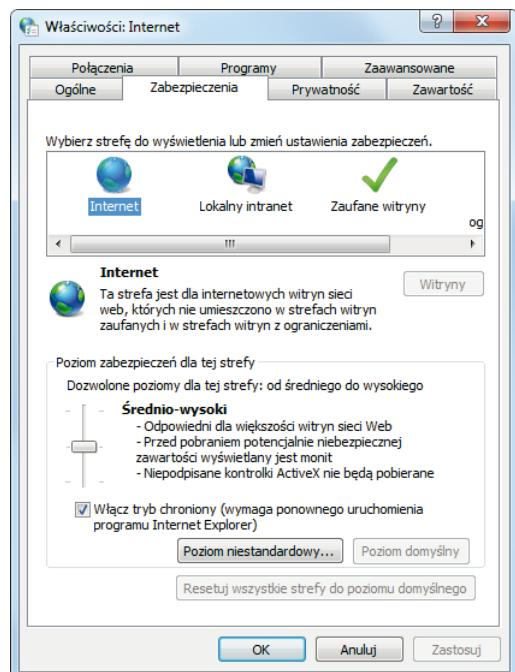
```
C:\>cd MojeProgramy
C:\MojeProgramy>dir
Wolumin w stacji C to 0S
Numer seryjny woluminu: 7A82-96DC
Katalog: C:\MojeProgramy
2018-01-29  07:52    <DIR>
2018-01-29  07:52    <DIR>
2018-01-29  07:53               68 Payroll.java
                           1 plików   68 bajtów
                           2 katalogów  69 866 708 992 bajtów wolnych
C:\MojeProgramy>
```

Rysunek 15.1. Interfejs wiersza poleceń (dzięki uprzejmości Microsoft Corporation)

Wielu użytkownikom komputerów, zwłaszcza początkującym, interfejs wiersza poleceń wydaje się zbyt skomplikowany. Wynika to z faktu, że aby z niego korzystać, należy poznać wiele poleceń, z których każde ma inną składnię — podobnie jak w językach programowania. Jeśli podczas wprowadzania polecenia użytkownik popełni błąd, polecenie nie zadziała.

W latach osiemdziesiątych XX wieku w komercyjnych systemach operacyjnych pojawił się nowy typ interfejsu — **graficzny interfejs użytkownika** (ang. *graphical user interface* — GUI). Dzięki graficznemu interfejsowi użytkownika można się komunikować z komputerem za pomocą elementów graficznych wyświetlanych na ekranie. GUI przyczynił się także do spopularyzowania myszki komputerowej jako urządzenia wskazującego. Zamiast wprowadzać polecenia za pomocą klawiatury, użytkownik może wskazać dany element na ekranie i aktywować go naciśnięciem przycisku myszy.

W przypadku GUI za interakcję z użytkownikiem w dużej mierze odpowiadają **okna dialogowe**, czyli okienka, w których wyświetlają się informacje i które umożliwiają użytkownikowi wykonanie określonych operacji. Na rysunku 15.2 przedstawiłem przykładowe okno dialogowe, dzięki któremu użytkownik systemu operacyjnego Windows może zmodyfikować opcje internetowe. Zamiast wprowadzać tajemnicze polecenia, użytkownik korzysta z elementów graficznych, takich jak ikony, przyciski czy suwaki.



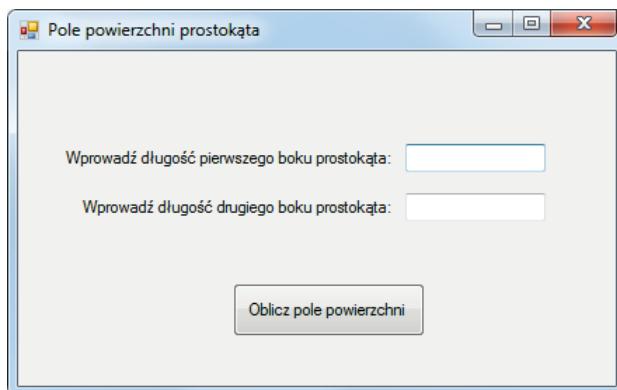
Rysunek 15.2. Okno dialogowe (dzięki uprzejmości Microsoft Corporation)

Jeśli będziesz tworzyć oprogramowanie dla systemów operacyjnych wyposażonych w GUI, takich jak Windows, Mac OS X lub Linux, możesz w swoich programach umieścić graficzny interfejs użytkownika, czyli wykorzystać takie popularne elementy jak okna dialogowe, ikony, przyciski itp.

## Programy z GUI to programy sterowane zdarzeniami

W środowiskach tekstowych, czyli na przykład w interfejsie wiersza poleceń, to program decyduje o tym, w jakiej kolejności wykonają się polecenia. Wyobraź sobie program obliczający pole powierzchni prostokąta. Najpierw program poprosi użytkownika o wprowadzenie długości pierwszego boku prostokąta. Użytkownik wprowadza wartość, po czym program prosi go o wprowadzenie długości drugiego boku prostokąta. Użytkownik wprowadza wartość, a program oblicza pole powierzchni prostokąta. Użytkownik nie ma wpływu na to, w jakiej kolejności będzie wprowadzał dane.

W przypadku środowiska wyposażonego w GUI to użytkownik wybiera kolejność operacji. Przykładowo na rysunku 15.3 pokazałem wyposażony w GUI program obliczający pole powierzchni prostokąta. Użytkownik może w tym przypadku wprowadzić długości boków w dowolnej kolejności. Jeśli popełni błąd, może w każdej chwili usunąć wprowadzone dane i wpisać poprawne. Kiedy użytkownik chce, aby program obliczył pole powierzchni prostokąta, kliką przycisk *Oblicz pole powierzchni*. Ponieważ programy wyposażone w GUI muszą reagować na działania wykonywane przez użytkownika, nazywa się je **programami sterowanymi zdarzeniami**. Użytkownik, klikając przycisk, powoduje wystąpienie w programie określonego zdarzenia, a program musi to zdarzenie obsłużyć.



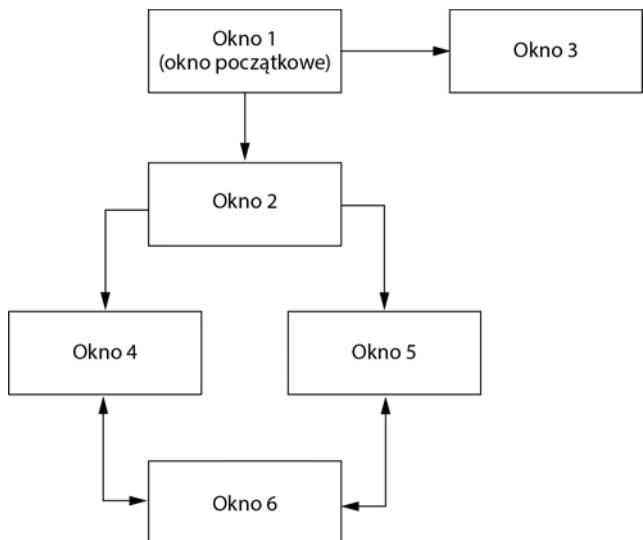
**Rysunek 15.3.** Program wyposażony w GUI (dzięki uprzejmości Microsoft Corporation)

## Tworzenie programu z GUI

Tworzenie programu wyposażonego w GUI w znacznym stopniu jest podobne do tworzenia programów tekstowych, które do tej pory przedstawiłem w książce. Przykładowo musisz określić, jakie zadania będzie wykonywał program, i kroki, dzięki którym je wykona.

Ponadto musisz zaprojektować okno programu składające się z różnych elementów GUI — będzie ono stanowiło interfejs użytkownika. Musisz także określić, w jaki sposób program będzie przechodził z jednego okna do drugiego. Niektórym programistom pomagają w tym **diagramy interfejsu użytkownika**. Na rysunku 15.4

przedstawiłem przykład takiego diagramu. Każdy prostokąt odpowiada innemu oknu dialogowemu. Kiedy jakieś działanie w danym oknie dialogowym skutkuje otwarciem innego okna, pomiędzy tymi oknami umieszczona jest strzałka. Na przykładowym diagramie jedna ze strzałek skierowana jest od Okna 1 do Okna 2. Oznacza to, że wskutek pewnego działania wykonanego w Oknie 1 następuje otwarcie Okna 2. Kiedy strzałka jest po obu stronach zakończona grotem, znaczy to, że oba okna mogą się wzajemnie otwierać.



**Rysunek 15.4.** Diagram interfejsu użytkownika



## Punkt kontrolny

- 15.1. Do czego służy interfejs użytkownika?
- 15.2. W jaki sposób działa interfejs wiersza poleceń?
- 15.3. Co decyduje o kolejności operacji w przypadku programu stworzonego w środowisku tekstowym, takim jak interfejs wiersza poleceń?
- 15.4. Co to są programy sterowane zdarzeniami?
- 15.5. Co to jest schemat blokowy interfejsu użytkownika?

**15.2**

## Projektowanie interfejsu użytkownika do programu wyposażonego w GUI

**WYJAŚNIENIE:** Tworząc program wyposażony w GUI, należy zaprojektować wszystkie okna programu i umieścić w nich komponenty graficzne.

GUI programu składa się z jednego lub większej liczby okien, które wyświetlają się w czasie działania programu. Jednym z zadań, które należy wykonać, tworząc program wyposażony w GUI, jest zaprojektowanie okien oraz elementów graficznych, z których się one składają.

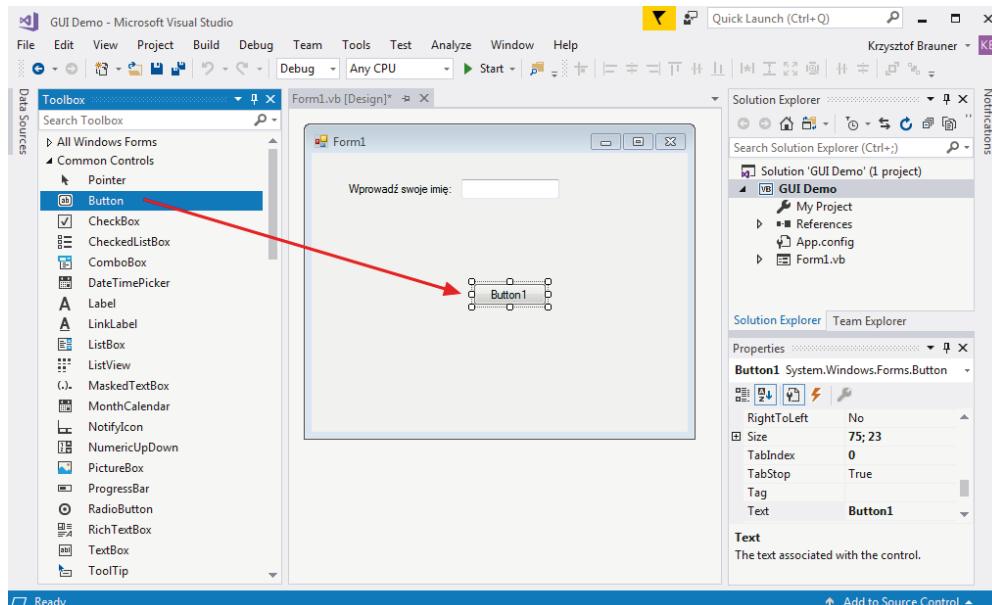
We wczesnej fazie rozwoju oprogramowania wyposażonego w graficzny interfejs użytkownika tworzenie okien było zadaniem bardzo złożonym i czasochłonnym. Programista musiał napisać kod odpowiedzialny za tworzenie okien oraz elementów graficznych takich jak ikonki i przyciski, a także ustawiać ich kolor, położenie, rozmiar i wiele innych właściwości. Nawet najprostszy program, który wyświetla po prostu napis *Witaj, świecie!*, wymagał od programisty napisania setki linii kodu, a czasem i więcej. Ponadto programista nie widział tak naprawdę, jak prezentuje się interfejs użytkownika, dopóki nie skompilował programu i go nie uruchomił.

Obecnie dostępnych jest wiele zintegrowanych środowisk programistycznych (ang. *integrated development environment* — IDE), za pomocą których można tworzyć okna i umieszczać w nich elementy w sposób graficzny, bez konieczności pisania kodu. Przykładowo w pakiecie Microsoft Visual Studio można tworzyć GUI dla programów pisanych w językach Visual Basic, C++ czy C#. Istnieje także wiele innych środowisk programistycznych.

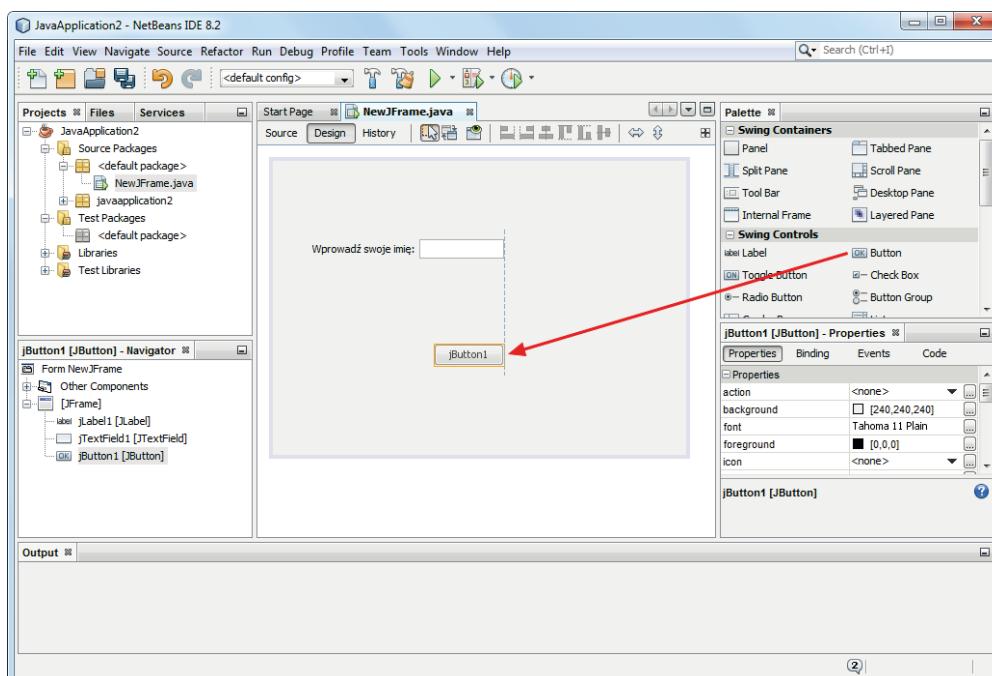
W większości IDE dostępny jest edytor okna, za pomocą którego można tworzyć nowe okna, i przybornik z elementami, które można umieścić w oknie. Okno tworzy się poprzez przeciągnięcie danego elementu z przybornika do okna. Zilustrowałem to na rysunkach 15.5 i 15.6. Na rysunku 15.5 widać edytor w środowisku Visual Studio, a na rysunku 15.6 — w środowisku NetBeans. Gdy tworzysz w ten sposób okno programu, IDE automatycznie generuje kod programu niezbędny do wyświetlenia okna.

### Komponenty

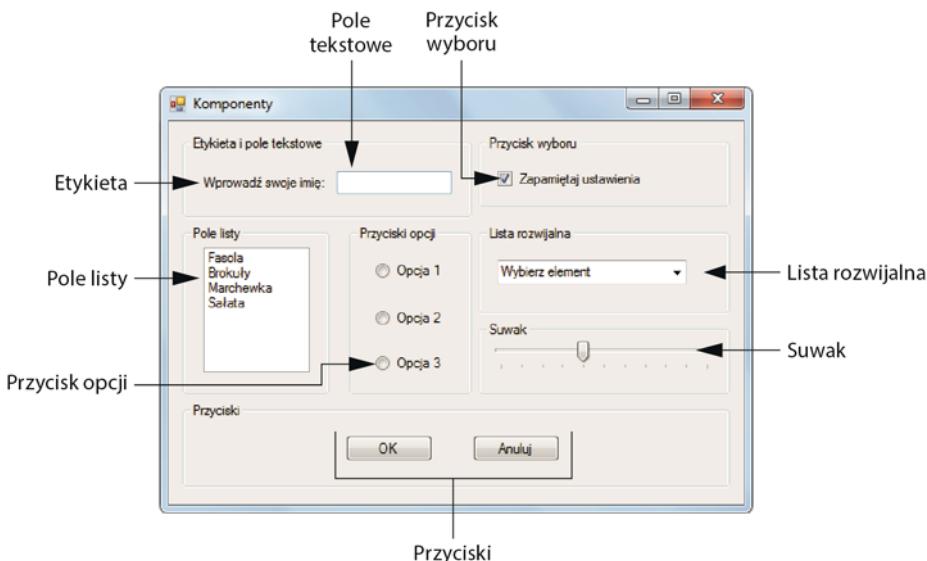
Elementy, z których składa się GUI, to **komponenty**. Najpopularniejszymi komponentami są przyciski, etykiety, pola tekstowe, przyciski wyboru czy przyciski opcji. Na rysunku 15.7 przedstawiłem przykładowe okno, w którym umieściłem kilka komponentów. W tabeli 15.1 opisałem każdy z widocznych w oknie komponentów.



**Rysunek 15.5.** Tworzenie okna w środowisku Visual Studio (dzięki uprzejmości Microsoft Corporation)



**Rysunek 15.6.** Tworzenie okna w środowisku NetBeans (dzięki uprzejmości Microsoft Corporation)



**Rysunek 15.7.** Różne komponenty umieszczone w oknie GUI (dzięki uprzejmości Microsoft Corporation)

**Tabela 15.1.** Popularne komponenty GUI

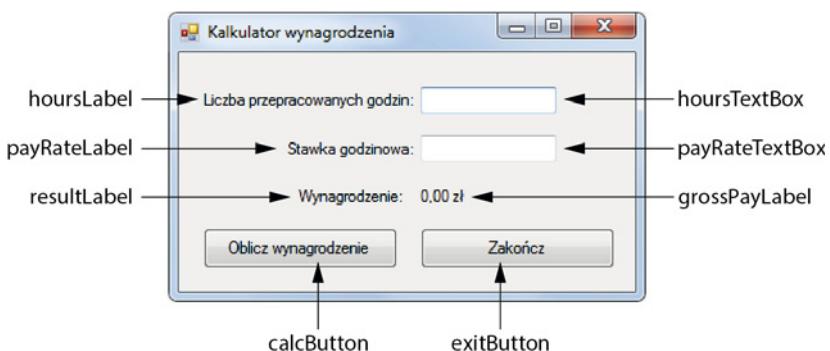
| Komponent                          | Opis                                                                                                                                                                        |
|------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Przycisk (ang. button)             | W momencie kliknięcia generuje w programie zdarzenie.                                                                                                                       |
| Etykieta (ang. label)              | Obszar, na którym można wyświetlić dowolny tekst.                                                                                                                           |
| Pole tekstowe (ang. text box)      | Pole, w którym użytkownik może wprowadzić dane wejściowe za pomocą klawiatury.                                                                                              |
| Przycisk wyboru (ang. check box)   | Mały kwadrat, który można zaznaczyć lub odznaczyć.                                                                                                                          |
| Przycisk opcji (ang. radio button) | Mały okrąg, który można zaznaczyć lub odznaczyć. Zazwyczaj umieszcza się kilka takich komponentów, spośród których można zaznaczyć tylko jeden.                             |
| Lista rozwijalna (ang. combo box)  | Komponent ten (będący połączeniem pola listy i pola tekstowego) wyświetla listę wartości, spośród których może wybierać użytkownik. W polu testowym można wprowadzić tekst. |
| Pole listy (ang. list box)         | Lista wartości, spośród których może wybierać użytkownik.                                                                                                                   |
| Suwak (ang. slider)                | Komponent, za pomocą którego użytkownik może ustawić wartość, przesuwając suwak.                                                                                            |



**UWAGA:** Komponenty GUI nazywane są także **kontrolkami** lub **widżetami**.

## Nazwy komponentów

W większości IDE po umieszczeniu komponentu w oknie musisz nadać mu unikatową nazwę. Za pomocą tej nazwy będziemy się odwoływać do danego komponentu w kodzie programu, podobnie jak robimy to w przypadku zmiennych. Przykładowo na rysunku 15.8 widoczne jest okno programu do obliczania wynagrodzenia pracownika. Na rysunku zaznaczylem nazwy, które programista nadal poszczególnym komponentom. Zauważ, że nazwa każdego komponentu wskazuje także jego przeznaczenie: pole tekstowe, w którym użytkownik będzie wpisywał liczbę przepracowanych godzin, nazywa się `hoursTextBox`, a przycisk za pomocą którego uruchamia się obliczenia, nazywa się `calcButton`. W większości języków programowania zasady nazewnictwa komponentów są takie same jak zasady nazewnictwa zmiennych.



Rysunek 15.8. Komponenty i ich nazwy (dzięki uprzejmości Microsoft Corporation)



**UWAGA:** Jeśli przyjrzyisz się kodowi wygenerowanemu przez IDE, zauważysz, że każdy komponent umieszczony w oknie jest obiektem, a nazwa tego obiektu jest taka sama jak nazwa komponentu.

## Właściwości

Do większości komponentów GUI przypisane są **właściwości**, które decydują o tym, jak dany komponent wygląda na ekranie. Zazwyczaj komponent ma właściwości odpowiadające za jego kolor, rozmiar i położenie. Do właściwości można przypisać określoną wartość — podobnie jak ma to miejsce w przypadku zmiennych. Po przypisaniu wartości do właściwości zmieni się wygląd komponentu.

Przyjrzyjmy się przykładowi w języku Visual Basic. Założymy, że w oknie umieściliśmy przycisk i chcemy, aby wyświetlił się na nim tekst *Pokaż wynik*. W Visual Basicu służy do tego celu właściwość o nazwie `Text`, więc kiedy przypiszemy do tej właściwości wartość *Pokaż wynik*, na przycisku pojawi się właśnie ten tekst. Widoczne jest to na rysunku 15.9.

**Pokaż wynik**

**Rysunek 15.9.** Przycisk, w którym do właściwości Text przypisano wartość Pokaż wynik

Większość komponentów w Visual Basicu ma także właściwość BackColor, która odpowiada za kolor komponentu, a także właściwość o nazwie ForeColor, która odpowiada za kolor wyświetlanego na komponencie tekstu. Przykładowo jeśli chcesz zmienić kolor tekstu, który wyświetla się na przycisku, na niebieski, wystarczy, że przypiszesz do właściwości ForeColor wartość Blue.

W większości IDE można ustawać właściwości komponentów podczas tworzenia danego okna. Zazwyczaj IDE udostępnia Okno właściwości, w którym możesz określić wszystkie dostępne właściwości poszczególnych komponentów.

## Tworzenie okna — podsumowanie

Teraz, gdy już wiesz, jak tworzy się okna za pomocą IDE, przyjrzymy się poszczególnym krokom, dzięki którym stworzysz nowe okno.

### 1. Naszkicuj okno.

Zanim przystąpisz do konstruowania okna w IDE, naszkicuj je najpierw na kartce papieru. Dzięki temu określisz, których komponentów będziesz potrzebować. Na tym etapie pomocne będzie także sporządzenie listy potrzebnych komponentów.

### 2. Umieść w oknie odpowiednie komponenty i nadaj im nazwy.

Gdy już naszkicujesz okno i określisz listę potrzebnych komponentów, możesz przystąpić do tworzenia okna w IDE. Po umieszczeniu w oknie nowego komponentu należy mu nadać unikatową i wskazującą jego przeznaczenie nazwę.

### 3. Ustaw odpowiednio właściwości komponentów.

Właściwości komponentu mają wpływ na jego wygląd, czyli na przykład kolor, rozmiar, położenie lub tekst wyświetlany na komponencie. Jeśli chcesz, aby komponent miał określony wygląd, odpowiednio ustaw jego właściwości. W większości IDE możesz wykorzystać do tego celu okno właściwości.

## W centrum uwagi

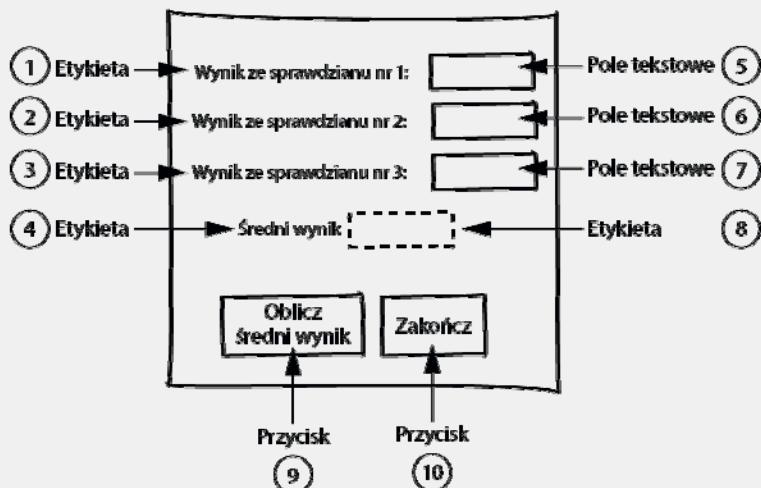
### Projektowanie okna

Kasia uczy w szkole biologii. W rozdziale 4. prześledziliśmy proces tworzenia programu, za pomocą którego uczniowie Kasi mogą obliczać średni wynik z trzech sprawdzianów. Program prosił ucznia o wprowadzenie wyniku z każdego sprawdzianu, a następnie wyświetlał średni wynik. Kasia poprosiła Cię, abyś stworzył tym razem program wyposażony w GUI, który będzie wykonywał tę samą operację. Kasia chce, aby w oknie



programu znajdowały się trzy pola tekstowe, w których będzie można wpisać wyniki ze sprawdzianów, i przycisk, po którego kliknięciu wyświetli się średni wynik.

Na samym początku musimy naszkicować okno programu — przedstawiłem je na rysunku 15.10. Na szkicu widoczne są także typy komponentów (ponumerowanie ich ułatwia nam stworzenie listy potrzebnych komponentów).



Rysunek 15.10. Szkic okna

Na podstawie szkicu możemy sporządzić listę potrzebnych komponentów. Opracowując listę, dodamy także krótki opis każdego komponentu i nazwę, jaką mu nadamy podczas tworzenia okna w IDE.

| Numer komponentu na szkicu | Typ komponentu | Opis komponentu                                                   | Nazwa komponentu |
|----------------------------|----------------|-------------------------------------------------------------------|------------------|
| 1                          | Etykieta       | Wskazuje użytkownikowi, aby wprowadził wynik ze sprawdzianu nr 1. | test1Label       |
| 2                          | Etykieta       | Wskazuje użytkownikowi, aby wprowadził wynik ze sprawdzianu nr 2. | test2Label       |
| 3                          | Etykieta       | Wskazuje użytkownikowi, aby wprowadził wynik ze sprawdzianu nr 3. | test3Label       |
| 4                          | Etykieta       | Wskazuje, gdzie wyświetli się średni wynik.                       | resultLabel      |
| 5                          | Pole tekstowe  | Tutaj użytkownik wprowadza wynik ze sprawdzianu nr 1.             | test1TextBox     |

| Numer komponentu na szkicu | Typ komponentu | Opis komponentu                                                                                       | Nazwa komponentu |
|----------------------------|----------------|-------------------------------------------------------------------------------------------------------|------------------|
| 6                          | Pole tekstowe  | Tutaj użytkownik wprowadza wynik ze sprawdzianu nr 2.                                                 | test2TextBox     |
| 7                          | Pole tekstowe  | Tutaj użytkownik wprowadza wynik ze sprawdzianu nr 3.                                                 | test3TextBox     |
| 8                          | Etykieta       | W tej etykiecie program wyświetli średni wynik ze sprawdzianów.                                       | averageLabel     |
| 9                          | Przycisk       | Po naciśnięciu tego przycisku program obliczy średni wynik i wyświetli go w komponencie averageLabel. | calcButton       |
| 10                         | Przycisk       | Po naciśnięciu tego przycisku program zakończy działanie.                                             | exitButton       |

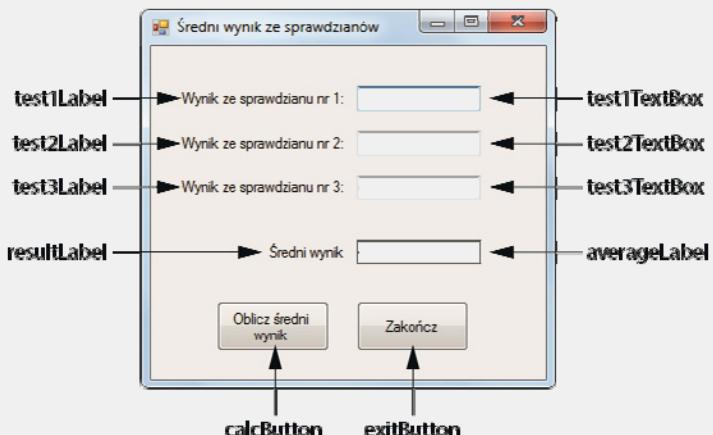
Gdy już mamy szkic okna i listę potrzebnych komponentów, możemy przystąpić do tworzenia okna w IDE. Po umieszczeniu komponentu w oknie należy odpowiednio ustawić jego właściwości. W przypadku języka Visual Basic ustaw właściwości komponentów w następujący sposób:

- Do właściwości Text komponentu test1Label przypisz tekst *Wynik ze sprawdzianu nr 1:*.
- Do właściwości Text komponentu test2Label przypisz tekst *Wynik ze sprawdzianu nr 2:*.
- Do właściwości Text komponentu test3Label przypisz tekst *Wynik ze sprawdzianu nr 3:*.
- Do właściwości Text komponentu resultLabel przypisz tekst *Średni wynik.*
- Do właściwości Text komponentu calcButton przypisz tekst *Oblicz średni wynik.*
- Do właściwości Text komponentu exitButton przypisz tekst *Zakończ.*
- Do właściwości BorderStyle komponentu averageLabel przypisz wartość *FixedSingle*. Dzięki temu etykieta zostanie otoczona ramką, podobnie jak na szkicu.



**WSKAZÓWKA:** Mimo że wymienione powyżej właściwości dotyczą Visual Basica, w wielu innych językach występują bardzo podobne właściwości. Ich nazwy mogą się jednak różnić.

Na rysunku 15.11 przedstawiłem, jak powinno wyglądać okno programu. Zaznaczylem na nim także nazwy poszczególnych komponentów.



Rysunek 15.11. Ukończone okno (dzięki uprzejmości Microsoft Corporation)

W kolejnej sekcji „W centrum uwagi” będziemy kontynuować pracę nad programem i dopiszemy pseudokod, dzięki któremu program będzie reagował na działania użytkownika.



## Punkt kontrolny

- 15.6. Dlaczego we wczesnej fazie rozwoju oprogramowania wyposażonego w GUI tworzenie interfejsu było tak złożonym i czasochłonnym zadaniem?
- 15.7. W jaki sposób podczas tworzenia GUI za pomocą IDE umieszcza się w oknie komponenty takie jak przycisk?
- 15.8. Co to jest komponent?
- 15.9. Dlaczego komponentom trzeba nadawać nazwy?
- 15.10. Do czego służą właściwości komponentów?

### 15.3

## Tworzenie procedury obsługi zdarzenia

**WYJAŚNIENIE:** Jeśli chcesz, aby w momencie wystąpienia jakiegoś zdarzenia program wyposażony w GUI wykonał określone zadanie, musisz stworzyć kod zwany procedurą obsługi zdarzenia.

Kiedy już stworzysz GUI, możesz przystąpić do pisania kodu, który będzie reagował na zdarzenia. Jak wspomniałem, **zdarzenie** ma miejsce w programie na przykład wtedy, gdy użytkownik kliknie przycisk. Jednym z etapów tworzenia aplikacji wyposażonej w GUI jest napisanie procedur obsługi zdarzeń. Procedura obsługi zdarzenia to

moduł, który zostanie wywołany w momencie, kiedy w programie wystąpi określone zdarzenie. Jeśli chcesz, aby po wystąpieniu zdarzenia program wykonał jakieś operacje, musisz stworzyć procedurę obsługi zdarzenia. W pseudokodzie procedurę obsługi zdarzenia będziemy zapisywać w następujący sposób:

```
Module NazwaKomponentu_NazwaZdarzenia()
    Instrukcje umieszczone w tym miejscu wykonają się wtedy,
    gdy w programie wystąpi określone zdarzenie
End Module
```

W takiej ogólnej postaci *NazwaKomponentu* oznacza nazwę komponentu, który generuje zdarzenie, a *NazwaZdarzenia* oznacza nazwę zdarzenia, które ma zostać obsłużone. Założymy, że w oknie znajduje się przycisk o nazwie *showResultButton* i chcemy stworzyć obsługę zdarzenia występującego po jego naciśnięciu. Procedura obsługi zdarzenia będzie wtedy wyglądała tak:

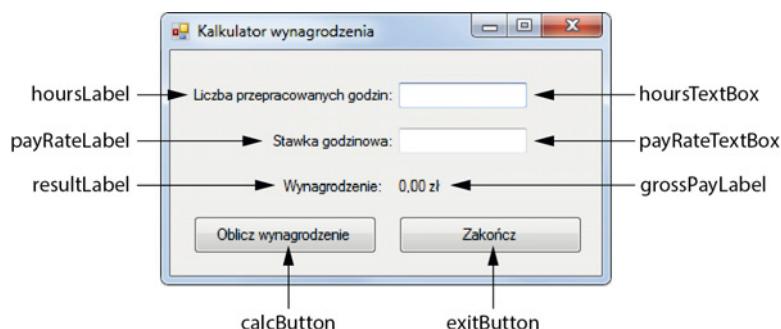
```
Module showResultButton_Click()
    instrukcja
    instrukcja
    itd.
End Module
```

Każde zdarzenie generowane przez program wyposażony w GUI ma swoją predefiniowaną nazwę. W tym przypadku było to zdarzenie o nazwie *Click*, które występuje, gdy użytkownik kliknie dany komponent. W programie dostępnych jest znacznie więcej zdarzeń. Przykładowo w momencie przesunięcia kurSORA na komponent, generuje on zdarzenie o nazwie *MouseEnter*, a gdy kurSOR znajdzie się poza komponentem, generowane jest zdarzenie o nazwie *MouseLeave*.



**UWAGA:** Jeśli w momencie wystąpienia zdarzenia okaże się, że w programie nie ma zdefiniowanej procedury obsługującej to zdarzenie, zostanie ono zignorowane.

Spójrzmy teraz, jak za pomocą pseudokodu możemy utworzyć procedurę obsługi zdarzenia. Wczesniej w tym rozdziale zaprezentowałem okno GUI programu do obliczania wynagrodzenia pracownika. Dla wygody zaprezentuję je ponownie na rysunku 15.12. Na rysunku widać także nazwy poszczególnych komponentów.



Rysunek 15.12. Okno GUI (dzięki uprzejmości Microsoft Corporation)

Chcemy, aby program po uruchomieniu reagował na dwa zdarzenia: gdy użytkownik kliknie przycisk `calcButton` i gdy użytkownik kliknie przycisk `exitButton`. Gdy użytkownik kliknie przycisk `calcButton`, program powinien obliczyć wynagrodzenie pracownika i wyświetlić je w komponencie `grossPayLabel`. Gdy użytkownik kliknie przycisk `exitButton`, program powinien zakończyć działanie.

Aby obsłużyć zdarzenie generowane po kliknięciu przycisku `calcButton`, napiszemy następującą procedurę obsługi zdarzenia:

```

1 Module calcButton_Click()
2   // Zmienne lokalne, w których zapiszemy liczbę przepracowanych godzin,
3   // stawkę godzinową i wynagrodzenie
4   Declare Real hours, payRate, grossPay
5
6   // Pobieramy liczbę przepracowanych godzin z komponentu
7   // hoursTextBox
8   Set hours = stringToReal(hoursTextBox.Text)
9
10  // Pobieramy stawkę godzinową z komponentu
11  // payRateTextBox
12  Set payRate = stringToReal(payRateTextBox.Text)
13
14  // Obliczamy wynagrodzenie.
15  Set grossPay = hours * payRate
16
17  // Wyświetlamy wynagrodzenie w komponencie
18  // grossPayLabel
19  Set grossPayLabel.Text = realToString(grossPay)
20 End Module

```

Przyjrzymy się bliżej poszczególnym instrukcjom w procedurze obsługi zdarzenia:

- W linii 4. deklaruję trzy zmienne lokalne: `hours`, `payRate` i `grossPay`.
- W linii 8. pobieram wartość wprowadzoną w komponencie `hoursTextBox` i przypisuję ją do zmiennej `hours`. W tej linii dzieje się kilka rzeczy, więc warto ją omówić dokładniej.

Gdy użytkownik wprowadzi wartość w polu tekstowym, zostanie ona przypisana do właściwości `Text`. W pseudokodzie odnosimy się właściwości `Text` za pomocą kropki. Przykładowo do właściwości `Text` komponentu `hoursTextBox` odnosimy się w taki sposób: `hoursTextBox.Text`.

W wielu językach nie można przypisać wartości zwracanej przez właściwość `Text` bezpośrednio do zmiennej typu liczbowego. Przykładowo gdybyśmy zapisali poleceńie w linii 8. w poniższy sposób, w programie wystąpiłby błąd:

```
Set hours = hoursTextBox.Text
```

Stanie się tak dlatego, że właściwość `Text` przyjmuje wartość w postaci ciągu znaków, a ciągu znaków nie da się przypisać do zmiennej typu liczbowego. Musimy więc zamienić wartość przypisaną do właściwości `Text` na liczbę rzeczywistą. Możemy to zrobić za pomocą funkcji `stringToReal`:

```
Set hours = stringToReal(hoursTextBox.Text)
```

Funkcję `stringToReal` omówiłem w rozdziale 6.

- W linii 12. pobieram wartość wprowadzoną w komponencie payRateTextBox, zamieniam ją na liczbę rzeczywistą i przypisuję do zmiennej payRate.
- W linii 15. mnożę hours przez payRate, a wynik przypisuję do zmiennej grossPay.
- W linii 19. wyświetlам wynagrodzenie. Robię to, przypisując do właściwości Text komponentu grossPayLabel wartość zapisaną w zmiennej grossPay. Zwróć uwagę, że w przypadku zmiennej grossPay do zamiany wartości liczbowej na ciąg znaków użyłem funkcji `realToString`. Jest to konieczne, ponieważ w wielu językach wystąpi błąd, jeśli spróbujemy przypisać liczbę rzeczywistą bezpośrednio do właściwości Text. Gdy przypiszemy już wartość do właściwości Text komponentu, wyświetli się ona na etykietce.

Aby obsłużyć zdarzenie mające miejsce w momencie kliknięcia przycisku exitButton, posłużymy się następującą procedurą obsługi zdarzenia:

```
1 Module exitButton_Click()
2   Close
3 End Module
```

Procedura wywołuje jedynie polecenie `Close`. W przypadku pseudokodu polecenie `Close` zamyka otwarte okno. Jeśli zamykane okno jest jedynym otwartym oknem w programie, polecenie zakończy także działanie programu.

## Moduł Init

Często się zdarza, że pewne czynności trzeba wykonać już podczas rozpoczętania pracy aplikacji GUI. Na przykład może zachodzić konieczność wyświetlenia początkowych danych w etykietach. W naszym pseudokodzie można wykonać te czynności za pomocą modułu `Init`. Jeśli aplikacja GUI ma już moduł `Init`, to jest on wywoływany automatycznie przy uruchomieniu aplikacji. Na przykład możemy chcieć dodać następujący moduł `Init` do kalkulatora wynagrodzeń:

```
Module Init()
  Set grossPayLabel.Text = "0.00 zł"
End Module
```

Moduł ten inicjalizuje właściwość `Text` komponentu `grossPayLabel` ciągiem znaków `"0,00 zł"`. W rezultacie po uruchomieniu aplikacji etykieta wyświetli tekst `0.00 zł`.

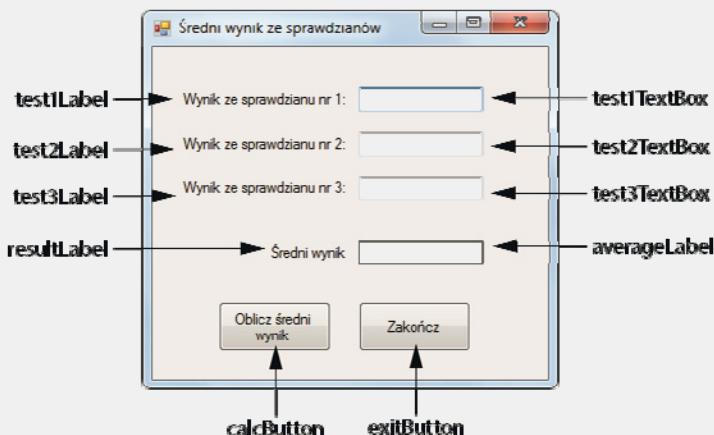
### W centrum uwagi

#### Projektowanie procedury obsługi zdarzenia

W poprzedniej sekcji „W centrum uwagi” zaprojektowaliśmy okno programu do obliczania średniego wyniku ze sprawdzianów, widoczne na rysunku 15.13.

Teraz zajmiemy się projektowaniem procedur obsługi zdarzenia. Kiedy użytkownik kliknie przycisk `calcButton`, program powinien obliczyć średnią wartość trzech liczb i wyświetlić ją w komponencie `averageLabel`. Natomiast kiedy użytkownik kliknie





**Rysunek 15.13.** Okno programu do obliczania średniego wyniku ze sprawdzianów (dzięki uprzejmości Microsoft Corporation)

przycisk `exitButton`, program powinien zakończyć działanie. Na listingu 15.1 zamieściłem pseudokod obu procedur obsługi zdarzenia, a także moduł `Init`, który ustala początkowy tekst w komponentie `averageLabel`.

### Listing 15.1

```

1 Module Init()
2     Set averageLabel.Text = "0"
3 End Module
4
5 Module calcButton_Click()
6     //Deklarujemy zmienne, w których zapiszemy
7     //wyniki ze sprawdzianów i średni wynik
8     Declare Real test1, test2, test3, average
9
10    //Pobieramy wynik z pierwszego sprawdzianu
11    Set test1 = stringToReal(test1TextBox.Text)
12
13    //Pobieramy wynik z drugiego sprawdzianu
14    Set test2 = stringToReal(test2TextBox.Text)
15
16    //Pobieramy wynik z trzeciego sprawdzianu
17    Set test3 = stringToReal(test3TextBox.Text)
18
19    //Obliczamy średni wynik
20    Set average = (test1 + test2 + test3) / 3
21
22    //Wyświetlamy średni wynik ze sprawdzianów
23    //w etykiecie averageLabel
24    Set averageLabel.Text = realToString(average)
25 End Module
26
27 Module exitButton_Click()
28     Close
29 End Module

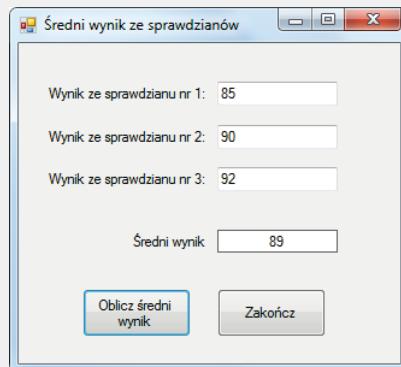
```

Moduł `Init` inicjalizuje właściwość `Text` komponentu `averageLabel` wartością 0. Powoduje to początkowe wyświetlenie w etykiecie tekstu 0. Oto opis działania poszczególnych instrukcji w module `calcButton_Click`:

- W linii 8. deklaruję zmienne lokalne, w których zapiszę wyniki ze sprawdzianów i ich średnią wartość.
- W linii 11. pobieram wartość wprowadzoną w polu tekstowym `test1TextBox`, następnie zamieniam ją na liczbę rzeczywistą i przypisuję do zmiennej `test1`.
- W linii 14. pobieram wartość wprowadzoną w polu tekstowym `test2TextBox`, następnie zamieniam ją na liczbę rzeczywistą i przypisuję do zmiennej `test2`.
- W linii 17. pobieram wartość wprowadzoną w polu tekstowym `test3TextBox`, następnie zamieniam ją na liczbę rzeczywistą i przypisuję do zmiennej `test3`.
- W linii 20. obliczam średni wynik z trzech sprawdzianów i zapisuję wynik w zmiennej `average`.
- W linii 24. zamieniam na ciąg znaków wartość zapisaną w zmiennej `average` i przypisuję ją do właściwości `Text` komponentu `averageLabel`. Dzięki temu wartość wyświetli się na etykiecie.

Moduł `exitButton_Click` wywołuje polecenie `Close`, które spowoduje zamknięcie okna i tym samym zakończenie programu.

Na rysunku 15.14 pokazałem okno programu po tym, jak użytkownik wprowadzi wartości i kliknie przycisk `calcButton`.



**Rysunek 15.14.** Okno z wyświetlonym średnim wynikiem (dzięki uprzejmości Microsoft Corporation)



## Punkt kontrolny

- 15.11. Co to jest zdarzenie?
- 15.12. Co to jest procedura obsługi zdarzenia?
- 15.13. Spójrz na poniższy pseudokod i odpowiedz na umieszczone pod nim pytania:

```

Module showValuesButton_Click()
    instrukcja
    instrukcja
    itd.
End Module

```

- a) Na które zdarzenie reaguje ten moduł?
- b) Jak nazywa się komponent, który generuje to zdarzenie?
- 15.14. W naszym pseudokodzie wprowadziliśmy moduł o nazwie Init. Jaki jest cel stosowania tego modułu? Kiedy się go używa?

## 15.4.

## Projektowanie aplikacji na urządzenia mobilne

**WYJAŚNIENIE:** Mniejszy rozmiar ekranu i specjalne możliwości sprzętowe zazwyczaj wymagają innego podejścia ze strony programistów tworzących aplikacje na urządzenia mobilne.

W aplikacjach opracowanych na urządzenia mobilne, takie jak na przykład smartfony i tablety, stosuje się interfejsy GUI. Jednak ekranы urządzeń mobilnych są znacznie mniejsze niż w komputerach stacjonarnych i w laptopach. W związku z tym GUI dla aplikacji mobilnej jest zwykle inaczej zaprojektowane niż w przypadku aplikacji przeznaczonej na komputery stacjonarne lub laptopy. Ponadto użytkownicy urządzeń mobilnych przeważnie wchodzą w interakcje z aplikacjami za pośrednictwem ekranu dotykowego, a nie myszki czy fizycznej klawiatury. Są to ważne kwestie, o których programiści muszą pamiętać przy tworzeniu aplikacji na urządzenia mobilne. Oto kilka wskazówek, których muszą przestrzegać programiści takich aplikacji:

- **Prezentuj mniej informacji naraz** — aplikacje mobilne muszą wyświetlać mniej informacji na ekranie niż tradycyjne aplikacje oglądane na ekranach komputerów stacjonarnych i laptopów. Treści, które z łatwością zmieszcza się w jednym oknie aplikacji komputera stacjonarnego, czasami muszą być rozzielone na wiele ekranów w aplikacji mobilnej.
- **Większe rozmiary czcionek ułatwiają czytanie** — aby zmniejszyć zmęczenie oczu i ułatwić czytanie na ekranach mobilnych, zazwyczaj wymagane jest użycie większych rozmiarów czcionki.
- **Rozmieść komponenty w sposób dopasowany do mniejszych rozmiarów ekranu** — często użytkownik urządzenia mobilnego musi przewijać ekran w aplikacji, aby zobaczyć całą ich zawartość. Z tego powodu ważniejsze i najczęściej używane komponenty powinny znajdować się w górnej części ekranu.
- **Używaj komponentów, które dobrze współpracują z ekranami dotykowymi** — aplikacje mobilne muszą korzystać z komponentów GUI, które działają dobrze w środowisku dotykowym. Komponenty te należy rozmieścić w taki sposób, aby użytkownik mógł z nich łatwo korzystać. Na przykład na ekranie dotykowym trudno jest kliknąć przyciski umieszczone blisko krawędzi ekranu.

Ponadto w środowisku dotykowym listy wyboru sprawdzają się lepiej niż listy rozwijane.

## Wykorzystanie unikalnych możliwości sprzętowych urządzeń mobilnych

Urządzenia mobilne mają zazwyczaj funkcje, które nie są dostępne na laptopach i na komputerach stacjonarnych. Na przykład typowe smartfony mogą wykonywać następujące czynności:

- wykonywanie i odbieranie połączeń telefonicznych;
- wysyłanie i odbieranie wiadomości SMS;
- wykrywanie lokalizacji urządzenia;
- wykrywanie fizycznej orientacji urządzenia w przestrzeni 3D;
- określanie, czy urządzenie się porusza.

Takie działania są możliwe dzięki wyspecjalizowanym komponentom sprzętowym, które są na wyposażeniu większości urządzeń mobilnych. Program lub aplikacja działająca na urządzeniu mobilnym może wchodzić w interakcje z tymi komponentami sprzętowymi. Program może uzyskiwać dane z komponentów i w wielu przypadkach powodować ich określone działanie. W tej sekcji przyjrzymy się niektórym algorytmom pseudokodowym i schematom blokowym, które wchodzą w interakcje ze specjalistycznymi komponentami sprzętowymi, jakie zazwyczaj znajdują się w urządzeniach mobilnych.



**UWAGA:** W naszym pseudokodzie użyjemy ogólnych technik programowania do interakcji z symulowanym urządzeniem mobilnym. Używana składnia się zmienia, gdy zaczniesz uczyć się programowania dla określonego typu urządzenia mobilnego. Większość logiki będzie jednak taka sama.

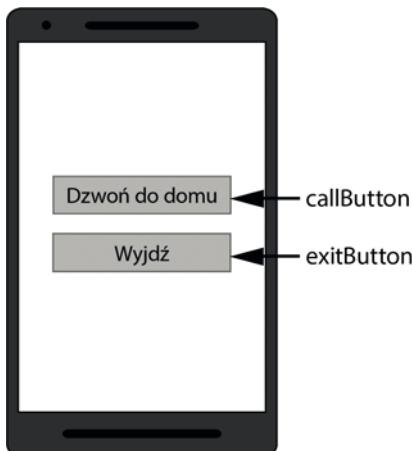
## Wykonywanie połączenia telefonicznego

Podstawowym elementem sprzętowym smartfonu jest ten, który wykonuje i odbiera połączenia telefoniczne. W naszych algorytmach będziemy odwoływać się do niego jako do komponentu PhoneCall. Nasz komponent PhoneCall ma metodę o nazwie makeCall, która wykonuje połączenie z określonym numerem telefonu. (Jak zapewne pamiętasz z rozdziału 14., metoda jest modelem lub funkcją, którą obiekt może wykonać). Oto przykład, jak wywołujemy metodę makeCall:

```
PhoneCall.makeCall("605-555-212")
```

Instrukcja ta sprawia, że urządzenie wykona połączenie telefoniczne z numerem 605-555-212. Zauważ, że numer telefonu jest ciągiem znaków i jest przekazywany jako argument do metody.

Spójrzmy na projekt bardzo prostej aplikacji, która korzysta z tej metody. Na rysunku 15.15 przedstawiam symulowany smartfon wyświetlający GUI aplikacji. W tym GUI znajduje się komponent Button o nazwie callButton oraz inny komponent Button,



**Rysunek 15.15.** Ekran dla aplikacji mobilnej Dzwoń do domu

o nazwie `exitButton`. Kiedy użytkownik naciśnie komponent `callButton`, aplikacja wybierze predefiniowany numer telefonu i zainicjuje połączenie telefoniczne. Natomiast w przypadku, gdy użytkownik naciśnie komponent `exitButton`, aplikacja zostanie zamknięta. Na listingu 15.2 prezentuję pseudokod dla metod obsługi zdarzeń tej aplikacji.

### **Listing 15.2**

```

1 Module callButton_Click()
2     PhoneCall.makeCall("919-555-1212")
3 End Module
4
5 Module exitButton_Click()
6     Close
7 End Module

```

W module `callButton_Click` instrukcja w wierszu 2. wywołuje metodę `makeCall` komponentu `PhoneCall`, przekazując jej jako argument ciąg znaków "919-555-1212". To spowoduje, że urządzenie wybierze określony numer telefonu. Natomiast w module `exitButton_Click` instrukcja `Close` z wiersza 6. spowoduje, że aplikacja zakończy działanie.

## **Wysyłanie wiadomości SMS**

Kolejnym ważnym komponentem smartfonu jest ten, który wysyła i odbiera wiadomości SMS. W naszych algorytmach będziemy odwoływać się do niego jako do komponentu `TextMessage`. Podobnie jak `PhoneCall`, komponent `TextMessage` również jest komponentem niewizualnym. Komponent `TextMessage` ma metodę o nazwie `sendText`, która wysyła wiadomość SMS na określony numer telefonu. Oto przykład wywołania metody `sendText`:

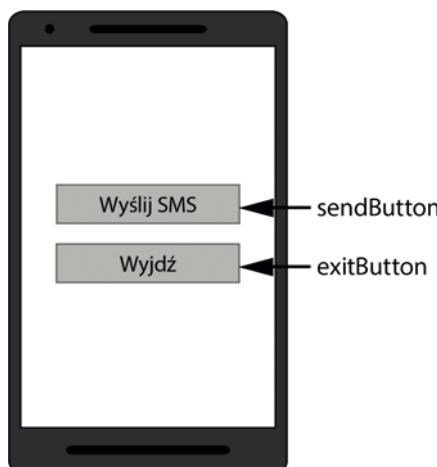
```

Declare String phoneNumber = "919-555-1212"
Declare String message = "O której godzinie przyjdiesz?"
TextMessage.sendText(phoneNumber, message)

```

Zauważ, że metodzie `sendText` przekazujemy dwa argumenty typu `string`: pierwszy to numer telefonu, na który wiadomość ma zostać wysłana, a drugi to tekst, który chcemy wysłać. Wynikiem działania tej instrukcji będzie wysłanie wiadomości tekstowej *O której godzinie przyjdiesz?* na numer telefonu 919-555-1212.

Spójrzmy na projekt prostej aplikacji, która korzysta z tej metody. Na rysunku 15.16 przedstawiam GUI aplikacji na symulowanym ekranie smartfonu. Znajduje się tam komponent `Button` o nazwie `sendButton` oraz inny komponent `Button`, o nazwie `exitButton`. Kiedy użytkownik naciśnie komponent `sendButton`, to aplikacja wyśle wiadomość SMS na wcześniej zdefiniowany numer telefonu. Natomiast w przypadku, gdy użytkownik naciśnie komponent `exitButton`, aplikacja zostanie zamknięta. Na listingu 15.3 możesz zobaczyć pseudokod metod obsługi zdarzeń aplikacji.



**Rysunek 15.16.** Ekran dla aplikacji mobilnej Wyślij SMS

### Listing 15.3

```

1 Module sendButton_Click()
2     Declare String phoneNumber = "919-555-1212"
3     Declare String message = "Jestem w drodze do domu."
4     TextMessage.sendText(phoneNumber, message)
5 End Module
6
7 Module exitButton_Click()
8     Close
9 End Module

```

W module `sendButton_Click` instrukcja w wierszu 2. deklaruje zmienną typu `String` o nazwie `phoneNumber` i inicjalizuje ją ciągiem znaków "919-555-1212". W wierszu 3. deklarowana jest zmienna typu `String` o nazwie `message`, zainicjalizowana ciągiem

znaków "Jestem w drodze do domu". W wierszu 4. wywoływana jest metoda `sendText` komponentu `TextMessage` i jako argumenty przekazywane są jej zmienne `phoneNumber` i `message`. Powoduje to, że urządzenie wysyła wiadomość SMS *Jestem w drodze do domu* na numer telefonu 919-555-1212. W module `exitButton_Click` znajdująca się w wierszu 8. instrukcja `Close` kończy działanie aplikacji.

## Odbieranie połączeń telefonicznych

W pseudokodzie symulowanego urządzenia mobilnego, w momencie, gdy urządzenie odbiera przychodzące połączenie telefoniczne, komponent `PhoneCall` generuje zdarzenie `IncomingCall`. Jeśli chcemy, aby aplikacja wykonywała pewne działania, gdy urządzenie odbiera przychodzące połączenie telefoniczne, musimy napisać procedurę obsługi zdarzenia w następującym ogólnym formacie:

```
Module PhoneCall_IncomingCall()
    instrukcja
    instrukcja
    itd.
End Module
```

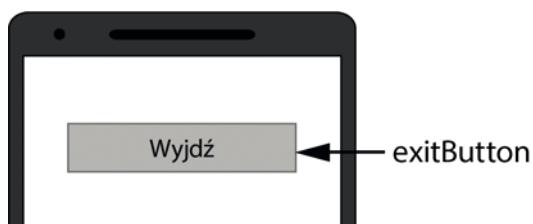
Instrukcje zapisane w module `PhoneCall_IncomingCall` będą wykonywane za każdym razem, gdy urządzenie otrzyma przychodzące połączenie telefoniczne.

Komponent `PhoneCall` ma również właściwość o nazwie `IncomingNumber`, która przechowuje numer telefonu (jako ciąg znaków) ostatniego połączenia przychodzącego. Oto jak można uzyskać numer telefonu z tej właściwości:

```
Declare String phoneNumber
phoneNumber = PhoneCall.IncomingNumber
```

Po wykonaniu tych instrukcji zmienna `phoneNumber` będzie przechowywać numer telefonu ostatniego przychodzącego połączenia.

Spójrzmy zatem na przykładowy projekt aplikacji, który odpowiada na przychodzące połączenie telefoniczne. Założymy, że przez pewien czas nie masz możliwości odbierania telefonu i chcesz, żeby za każdym razem, gdy przyjdzie połączenie, telefon automatycznie wysyłał do dzwoniącego SMS z informacją, że nie możesz odebrać. Na rysunku 15.17 pokazałem GUI aplikacji na symulowanym ekranie smartfonu. Ma on jeden komponent `Button` o nazwie `exitButton`. Na listingu 15.4 możesz zobaczyć pseudokod dla metod obsługi zdarzeń aplikacji.



**Rysunek 15.17** Ekran dla aplikacji mobilnej automatycznie odpowiadającej na połączenie

**Listing 15.4**

```

1 Module PhoneCall_IncomingCall()
2   Declare String phoneNumber
3   Declare String message = "Oddzwonię do Ciebie później."
4
5   phoneNumber = PhoneCall.IncomingNumber
6   TextMessage.sendText(phoneNumber, message)
7 End Module
8
9 Module exitButton_Click()
10  Close
11 End Module

```

W skrócie możemy opisać tę aplikację w następujący sposób: po uruchomieniu oczekuje na przychodzące połączenia telefoniczne, a w momencie, gdy pojawi się połączenie przychodzące, wysyła do osoby dzwoniącej SMS z tekstem *Oddzwonię do Ciebie później*. Użytkownik może zakończyć działanie aplikacji, naciskając przycisk *Exit*, aby ją zamknąć.

## Odbieranie wiadomości SMS

W pseudokodzie symulowanego urządzenia mobilnego komponent *TextMessage* generuje zdarzenie *IncomingText* w momencie, gdy urządzenie odbiera przychodząca wiadomość SMS. Jeśli chcemy, aby aplikacja wykonywała pewne czynności, kiedy urządzenie odbierze przychodząca wiadomość SMS, musimy napisać procedurę obsługi zdarzenia w następującym ogólnym formacie:

```

Module TextMessage_IncomingText()
  instrukcja
  instrukcja
  itd.
End Module

```

Instrukcje zapisane w module *TextMessage\_IncomingText* będą wykonywane za każdym razem, gdy urządzenie otrzyma przychodząca wiadomość SMS.

Komponent *TextMessage* ma również właściwość o nazwie *IncomingNumber*, która przechowuje numer telefonu (w postaci ciągu znaków), z którego nadeszła ostatnia wiadomość SMS. Oto jak uzyskać numer telefonu z tej właściwości:

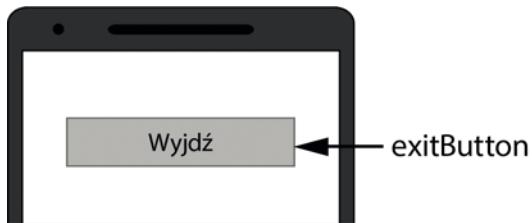
```

Declare String phoneNumber
phoneNumber = TextMessage.IncomingNumber

```

Po wykonaniu tych instrukcji zmienna *phoneNumber* będzie przechowywać numer telefonu, z którego wysłano ostatnią przychodząca wiadomość SMS.

Spójrzmy zatem na przykładowy projekt aplikacji, który odpowiada na przychodząca wiadomość tekstową. Założmy, że prowadzisz samochód i chcesz, aby telefon za każdym razem, gdy przychodzi wiadomość SMS, automatycznie wysyłał do nadawcy SMS, w którym przepraszasz, że nie możesz teraz odpowiedzieć. Na rysunku 15.18 możesz



Rysunek 15.18. Ekran dla aplikacji mobilnej automatycznie odpowiadającej na wiadomość

zobaczyć GUI aplikacji na symulowanym ekranie smartfonu. Ma on jeden komponent Button o nazwie exitButton. Na listingu 15.5 prezentuję pseudokod dla metod obsługi zdarzeń aplikacji.

### Listing 15.5

```

1 Module TextMessage_IncomingMessage()
2   Declare String phoneNumber
3   Declare String message = "Prowadzę samochód, więc odpowiem później."
4   phoneNumber = TextMessage.IncomingNumber
5   TextMessage.sendText(phoneNumber, message)
6 End Module
7
8 Module exitButton_Click()
9   Close
10 End Module

```

Po uruchomieniu programu aplikacja czeka na przychodzące wiadomości SMS. Gdy nadaje się wiadomość, aplikacja odsyła SMS z tekstem *Prowadzę samochód, więc odpowiem później*. Użytkownik może zakończyć działanie aplikacji, naciskając przycisk Exit.

## Wykrywanie lokalizacji urządzenia

Większość urządzeń mobilnych jest wyposażona w *czujnik lokalizacji*. Jest to komponent, który może określić szerokość i długość geograficzną, a także najbliższy adres. W pseudokodzie będziemy określać ten komponent jako *Location*. Nasz komponent Location ma następujące trzy metody, za pomocą których możemy uzyskać parametry określające lokalizację urządzenia:

- *getLatitude()* — zwraca szerokość geograficzną lokalizacji urządzenia jako liczbę typu Real.
- *getLongitude()* — zwraca długość geograficzną lokalizacji urządzenia jako liczbę typu Real.
- *getAddress()* — zwraca ciąg znaków, który zawiera adres znajdujący się najbliżej lokalizacji urządzenia.

Komponent *Location* generuje również zdarzenie *LocationChanged* za każdym razem, gdy zmienia się lokalizacja urządzenia. Jeśli chcemy, aby aplikacja wykonywała jakąś czynność po zmianie lokalizacji urządzenia, musimy napisać procedurę obsługi zdarzenia w następującym formacie ogólnym:

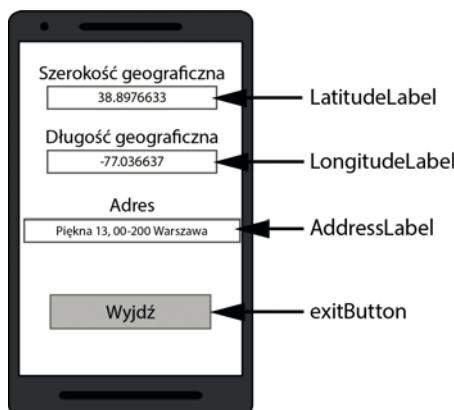
```

Module Location_LocationChanged()
    instrukcja
    instrukcja
    itd.
End Module

```

Instrukcje znajdujące się w module `Location_LocationChanged` będą wykonywane za każdym razem, gdy zmieni się lokalizacja urządzenia.

Spójrzmy na przykładowy projekt aplikacji, który korzysta z czujnika lokalizacji. GUI aplikacji prezentuję na rysunku 15.19, na symulowanym ekranie smartfonu. Po uruchomieniu aplikacja wyświetla w etykietach aktualną szerokość i długość geograficzną oraz najbliższy adres ulicy. Ilekroć zmienia się położenie urządzenia, aplikacja pobiera nową szerokość i długość geograficzną oraz adres ulicy i aktualizuje zawartość etykiet, wpisując do nich nowe informacje. Program z listingu 15.6 pokazuje pseudokod takiej aplikacji.



Rysunek 15.19. Ekran dla aplikacji mobilnej korzystającej z czujnika lokalizacji

### Listing 15.6

```

1 Module Init()
2     // Inicjalizowanie etykiet danymi
3     // aktualnej lokalizacji urządzenia
4     LatitudeLabel.Text = realToString(Location.getLatitude())
5     LongitudeLabel.Text = realToString(Location.getLongitude())
6     AddressLabel.Text = Location.getAddress()
7 End Module
8
9 Module Location_LocationChanged()
10    // Aktualizowanie etykiet po zmianie
11    // lokalizacji urządzenia
12    LatitudeLabel.Text = realToString(Location.getLatitude())
13    LongitudeLabel.Text = realToString(Location.getLongitude())
14    AddressLabel.Text = Location.getAddress()
15 End Module
16
17 Module exitButton_Click()
18     Close
19 End Module

```

Przyjrzyjmy się bliżej powyższemu pseudokodowi. Moduł `Init`, pokazany w wierszach od 1. do 7., jest wykonywany po uruchomieniu aplikacji. Wykonuje on następujące czynności, aby wyświetlić początkową lokalizację urządzenia:

1. W wierszu 4. pobiera aktualną szerokość geograficzną lokalizacji urządzenia, przekształca ją w ciąg znaków i przypisuje go do właściwości `Text` komponentu `LatitudeLabel`. Spowoduje to wyświetlenie szerokości geograficznej w etykiecie.
2. W wierszu 5. pobiera aktualną długość geograficzną lokalizacji urządzenia, przekształca ją w ciąg znaków i przypisuje go do właściwości `Text` komponentu `LongitudeLabel`. Spowoduje to wyświetlenie długości geograficznej w etykiecie.
3. W wierszu 6. pobiera adres znajdujący się najbliżej aktualnej lokalizacji urządzenia i przypisuje go do właściwości `Text` komponentu `AddressLabel`. Wykonanie tej instrukcji spowoduje, że w etykiecie wyświetli się adres.

Moduł `Location_LocationChanged` jest zapisany w wierszach od 9. do 15. Moduł ten jest funkcją obsługi zdarzenia, która wykonywana jest za każdym razem, gdy zmienia się lokalizacja urządzenia. Instrukcje w wierszach od 12. do 14. pobierają szerokość i długość geograficzną lokalizacji urządzenia oraz najbliższy adres, a następnie wyświetlają je w komponentach `Label`.

Natomiast moduł `exitButton_Clicked`, pokazany w wierszach od 17. do 19., zamienia aplikację, gdy użytkownik kliknie przycisk `Wyjdź`.



## Punkt kontrolny

- 15.15. Czy urządzenia mobilne umożliwiają wyświetlenie większej czy mniejszej ilości informacji na ekranie?
- 15.16. Czy ekrany urządzeń przenośnych zazwyczaj wymagają większej czy mniejszej czerwonki w porównaniu do ekranów urządzeń stacjonarnych?
- 15.17. Wymień co najmniej trzy unikalne możliwości, jakimi zwykle charakteryzują się urządzenia mobilne w porównaniu z komputerami stacjonarnymi.
- 15.18. Co to jest komponent niewizualny? Podaj dwa przykłady, które omówiliśmy w tym podrоздziale.
- 15.19. Założmy, że piszesz pseudokod aplikacji, która wykonuje jakąś akcję za każdym razem, gdy smartfon odbiera połączenie przychodzące. Skąd aplikacja ma informacje, że przychodzi połączenie telefoniczne?
- 15.20. Założmy, że piszesz pseudokod aplikacji, która wykonuje jakąś akcję za każdym razem, gdy smartfon odbiera przychodząca wiadomość tekstową. Skąd aplikacja będzie wiedziała, że nadeszła nowa wiadomość?
- 15.21. Założmy, że piszesz pseudokod aplikacji mobilnej, która wykonuje jakąś akcję za każdym razem, gdy zmienia się lokalizacja urządzenia. Skąd aplikacja będzie wiedziała, że zmieniła się lokalizacja urządzenia?

**15.5**

## Rzut oka na języki Java, Python i C++

### Java

W tym rozdziale omówiłem już sposoby tworzenia komponentów GUI w zintegrowanym środowisku programistycznym (IDE) oraz sposoby tworzenia metod obsługi zdań z uwzględnieniem odpowiednich interakcji z użytkownikiem. Java nie jest wyposażona w IDE, ale firma Oracle oferuje bezpłatne IDE, a mianowicie NetBeans. Można pobrać je za darmo ze strony <http://www.netbeans.org/downloads>. Pamiętaj jednak, aby pobrać wersję Java SE. Po pobraniu i zainstalowaniu NetBeans zapoznaj się z rozdziałem 15. podręcznika *Java Language Companion*, który jest dostępny na stronie internetowej wydawcy niniejszego podręcznika (<ftp://ftp.helion.pl/przyklady/pkpro5.zip>). *Java Language Companion* zawiera samouczek krok po kroku, który prowadzi użytkownika przez proces tworzenia prostej aplikacji z interfejsem GUI w języku Java w środowisku NetBeans.

### Python

Python nie ma wbudowanych funkcji programowania GUI, jest jednak wyposażony w moduł o nazwie Tkinter, który pozwala tworzyć proste programy GUI. Nazwa „Tkinter” jest skrótem od „Tk interface”. Interfejs został tak nazwany, ponieważ umożliwia programistom Pythona korzystanie z biblioteki GUI o nazwie Tk. Wiele innych języków programowania również korzysta z biblioteki Tk.

Rozdział 15. w książce *Python Language Companion*, która jest dostępna na stronie internetowej wydawcy niniejszej książki (<ftp://ftp.helion.pl/przyklady/pkpro5.zip>), zawiera krótkie wprowadzenie do programowania GUI za pomocą języka Python i biblioteki Tkinter.

### C++

C++ nie ma wbudowanych funkcji programowania GUI. Istnieje jednak wiele bibliotek zewnętrznych i narzędzi programistycznych, które można wykorzystać do pisania aplikacji GUI w tym języku. Na przykład bardzo popularne jest środowisko Microsoft Visual C++. Możemy użyć IDE Microsoft Visual Studio do tworzenia aplikacji w języku Visual C++.

Visual Studio można bezpłatnie pobrać ze strony [www.visualstudio.com/downloads](http://www.visualstudio.com/downloads). Pobierz wersję Community Edition. Po pobraniu i zainstalowaniu programu Visual Studio zapoznaj się z rozdziałem 15. książki C++ *Language Companion*, która jest dostępna na stronie internetowej wydawcy niniejszej książki (<ftp://ftp.helion.pl/przyklady/pkpro5.zip>). W książce C++ *Language Companion* znajduje się samouczek krok po kroku, który prowadzi użytkownika przez proces tworzenia prostej aplikacji z interfejsem GUI w programie Visual C++.

## Pytania kontrolne

### Test jednokrotnego wyboru

1. Za pomocą \_\_\_\_\_ użytkownik komunikuje się z komputerem.
  - a) procesora
  - b) interfejsu użytkownika
  - c) systemu sterowania
  - d) systemu interaktywnego
2. Zanim popularyzowane zostały graficzne interfejsy użytkownika, z programów korzystano za pomocą interfejsu \_\_\_\_\_.
  - a) wiersza poleceń
  - b) zdalnego terminala
  - c) sensorycznego
  - d) sterowanego zdarzeniami
3. \_\_\_\_\_ to małe okno, w którym można wyświetlić informacje i które umożliwia użytkownikowi wykonanie określonych operacji.
  - a) menu
  - b) okno zatwierdzające
  - c) ekran rozruchowy
  - d) okno dialogowe
4. Program \_\_\_\_\_ jest przykładem programu sterowanego zdarzeniami.
  - a) wiersza poleceń
  - b) tekstowy
  - c) GUI
  - d) proceduralny
5. \_\_\_\_\_ to element będący częścią graficznego interfejsu użytkownika.
  - a) gadżet
  - b) komponent
  - c) narzędzie
  - d) obiekt graficzny
6. Dzięki \_\_\_\_\_ komponentu można wpływać na to, jaki kolor, rozmiar i położenie będą miały elementy, z których składa się GUI.
  - a) właściwościom
  - b) atrybutom
  - c) metodom
  - d) procedurom obsługi zdarzenia
7. \_\_\_\_\_ to działanie mające miejsce w programie, takie jak kliknięcie przycisku.
  - a) procedura obsługi zdarzenia
  - b) anomalia
  - c) zdarzenie
  - d) wyjątek
8. \_\_\_\_\_ to moduł, który jest wywoływany automatycznie w momencie wystąpienia w programie określonego zdarzenia.
  - a) procedura obsługi zdarzenia
  - b) moduł automatyczny

- c) moduł rozruchowy
  - d) wyjątek
9. W naszym pseudokodzie moduł \_\_\_\_\_ wykonywany jest automatycznie po uruchomieniu aplikacji GUI.
- a. StartUp
  - b. Init
  - c. Auto
  - d. Main
10. Komponent \_\_\_\_\_, który omówiliśmy w tym rozdziale, jest przykładem komponentu niewizualnego.
- a. Button
  - b. Label
  - c. TextMessage
  - d. Text box

### Prawda czy fałsz?

1. Wielu użytkowników komputerów, zwłaszcza początkujący, uważa, że interfejs wiersza poleceń jest zbyt skomplikowany.
2. Tworzenie programów wyposażonych w GUI jest obecnie bardzo trudne i czasochłonne, ponieważ należy napisać cały kod programu odpowiedzialnego za wygląd okna.
3. Do właściwości Text komponentu przypisywany jest zazwyczaj ciąg znaków.
4. Każde ze zdarzeń generowanych przez program wyposażony w GUI ma predefiniowaną nazwę.
5. Diagram interfejsu użytkownika ilustruje, w jaki sposób program w wyniku działań użytkownika przechodzi z jednego okna do innego.
6. W naszym pseudokodzie moduł Init automatycznie uruchamia się po uruchomieniu aplikacji.
7. Podczas tworzenia graficznego interfejsu użytkownika dla urządzeń mobilnych i komputerów stacjonarnych zwykle używasz mniejszej czcionki dla urządzeń mobilnych niż dla komputerów stacjonarnych.

### Krótką odpowiedź

1. Co odpowiada za kolejność wykonywania operacji w programie działającym w środowisku tekstowym, takim jak interfejs wiersza poleceń?
2. Za pomocą czego można dostosować wygląd komponentu?
3. Wyjaśnij, w jaki sposób można zmienić kolor komponentu.
4. Dlaczego komponentom trzeba nadawać nazwy?
5. Co się stanie, gdy w momencie wystąpienia zdarzenia w programie nie będzie zdefiniowanej procedury jego obsługi?
6. Co to jest komponent niewizualny? Podaj przykład takiego komponentu, o którym mówiliśmy już w tym rozdziale.

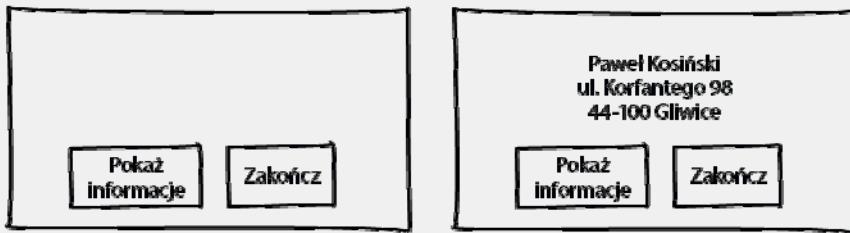
## Warsztat projektanta algorytmów

- Zaprojektuj procedurę obsługi zdarzenia, która wywołana zostanie w momencie kliknięcia komponentu o nazwie `showNameButton`. Procedura powinna wykonać następujące operacje:
  - zapisać Twoje imię w etykiecie o nazwie `firstNameLabel`;
  - zapisać Twoje drugie imię w etykiecie o nazwie `middleNameLabel`;
  - zapisać Twoje nazwisko w etykiecie o nazwie `lastNameLabel`.
 Pamiętaj, że aby przypisać wartość do etykiety, musisz zapisać ją we właściwości `Text`.
- Zaprojektuj procedurę obsługi zdarzenia, która wywołana zostanie w momencie kliknięcia komponentu o nazwie `calcAvailableCreditButton`. W procedurze wykonaj następujące operacje:
  - zadeklaruj zmienne typu `Real` o nazwach `maxCredit`, `usedCredit` i `availableCredit`;
  - pobierz wartość zapisaną w polu tekstowym o nazwie `maxCreditTextBox` i zapisz ją w zmiennej `maxCredit`;
  - pobierz wartość zapisaną w polu tekstowym o nazwie `usedCreditTextBox` i zapisz ją w zmiennej `usedCredit`;
  - odejmij od wartości zapisanej w zmiennej `maxCredit` wartość zapisaną w zmiennej `usedCredit` i przypisz wynik do zmiennej `availableCredit`;
  - przypisz wartość zapisaną w zmiennej `availableCredit` do etykiety o nazwie `availableCreditLabel`.
- GUI dla aplikacji mobilnej zawiera komponent typu `Label` o nazwie `phoneNumberLabel`. Napisz procedurę obsługi zdarzenia, która będzie wykonywana, gdy urządzenie odbierze przychodzące połączenie telefoniczne. Procedura obsługi tego zdarzenia powinna wyświetlić numer telefonu połączenia przychodzącego w komponencie `phoneNumberLabel`.
- W GUI aplikacji mobilnej znajduje się komponent typu `Label` o nazwie `addressLabel`. Napisz moduł `Init`, który będzie pobierał adres znajdujący się najbliżej lokalizacji urządzenia, a następnie wyświetli go w komponencie `addressLabel`.

## Ćwiczenia programistyczne

### 1. Dane osobowe

Zaprojektuj program wyposażony w GUI, który po naciśnięciu przycisku wyświetli Twoje dane osobowe. Okno programu powinno wyglądać podobnie jak to na szkicu z lewej strony rysunku 15.20. Kiedy użytkownik naciśnie przycisk *Pokaż informacje*, program powinien wyświetlić Twoje imię i nazwisko oraz adres zamieszkania — tak jak na szkicu z prawej strony rysunku 15.20.



**Rysunek 15.20.** Program wyświetlający dane osobowe

## 2. Tłumacz z łaciny

Spójrz na poniższą listę łacińskich słów i ich znaczeń w języku polskim:

| Język łaciński | Język polski |
|----------------|--------------|
| sinister       | lewo         |
| dexter         | prawo        |
| medium         | środek       |

Zaprojektuj program wyposażony w GUI, który będzie tłumaczył słowa z języka łacińskiego na język polski. W oknie powinny być umieszczone trzy przyciski, a na każdym z nich powinien widnieć jeden z wyrazów łacińskich. Kiedy użytkownik kliknie dany przycisk, program powinien wyświetlić na innym komponencie (etykiecie) tłumaczenie słowa na język polski.

## 3. Zużycie paliwa

Zaprojektuj program wyposażony w GUI, który będzie obliczał zużycie paliwa. W oknie programu powinny się znajdować pola tekstowe, w których użytkownik będzie mógł wprowadzić pojemność baku samochodu (w litrach) oraz liczbę kilometrów, jaką może przejechać samochód na pełnym baku. Po kliknięciu przycisku *Oblicz zużycie paliwa* program powinien wyświetlić liczbę kilometrów, którą może przejechać samochód na 1 litrze paliwa. Skorzystaj z następującego wzoru:

$$KNL = \text{przejechane kilometry} : \text{pojemność baku}$$

## 4. Zamiana stopni Celsjusza na stopnie Fahrenheita

Zaprojektuj program wyposażony w GUI, który będzie służył do zamiany temperatury wyrażonej w stopniach Celsjusza na stopnie Fahrenheita.

Użytkownik powinien mieć możliwość wprowadzenia wartości temperatury w stopniach Celsjusza i kliknięcia przycisku. Program powinien wyświetlić temperaturę w stopniach Fahrenheita. Skorzystaj z następującego wzoru:

$$F = \frac{9}{5}C + 32$$

We wzorze  $F$  oznacza temperaturę w stopniach Fahrenheita, a  $C$  — temperaturę w stopniach Celsjusza.

## 5. Podatek od nieruchomości

Hrabstwo pobiera podatek od nieruchomości naliczany na podstawie jej szacowanej wartości, wynoszącej 60% faktycznej wartości. Przykładowo jeżeli akr ziemi kosztuje 10000 dolarów, jego szacowana wartość wyniesie 6000 dolarów. Podatek od nieruchomości wynosi 64 centy od każdych 100 dolarów wartości szacowanej. Podatek od akra nieruchomości o wartości szacowanej równej 6000 dolarów będzie więc wynosił 38,40 dolara. Zaprojektuj program wyposażony w GUI, który po wprowadzeniu przez użytkownika rzeczywistej wartości nieruchomości wyświetli wartość szacowaną i kwotę podatku od nieruchomości.

## 6. Aplikacja mobilna do wysyłania wiadomości lokalizacyjnych

Czasami istnieje potrzeba przekazania swojej lokalizacji innej osobie, a można to zrobić za pomocą wiadomości SMS. Zaprojektuj aplikację mobilną, która:

- pozwala użytkownikowi wprowadzić numer telefonu osoby, do której zostanie wysłana wiadomość SMS;
- udostępnia przycisk, który wysyła wiadomość SMS na ostatnio podany numer telefonu, przy czym wiadomość ta powinna zawierać adres znajdujący się najbliżej aktualnej lokalizacji urządzenia.

Pamiętaj, że użytkownik nie powinien wprowadzać numeru telefonu za każdym razem, gdy wysłana jest wiadomość SMS. Wiadomość powinna zostać wysłana na ostatnio wprowadzony numer telefonu. Jeśli użytkownik spróbuje wysłać wiadomość przed wprowadzeniem numeru telefonu, aplikacja powinna wyświetlić komunikat o błędzie.

## 7. Śledzenie tras

Zaprojektuj aplikację mobilną, która będzie zapisywać w pliku długość i szerokość geograficzną lokalizacji urządzenia za każdym razem, gdy lokalizacja się zmieni. Aby zapisać te parametry do pliku w postaci rekordów, wykorzystaj techniki opisane w rozdziale 10.



## Tablica kodów ASCII/Unicode

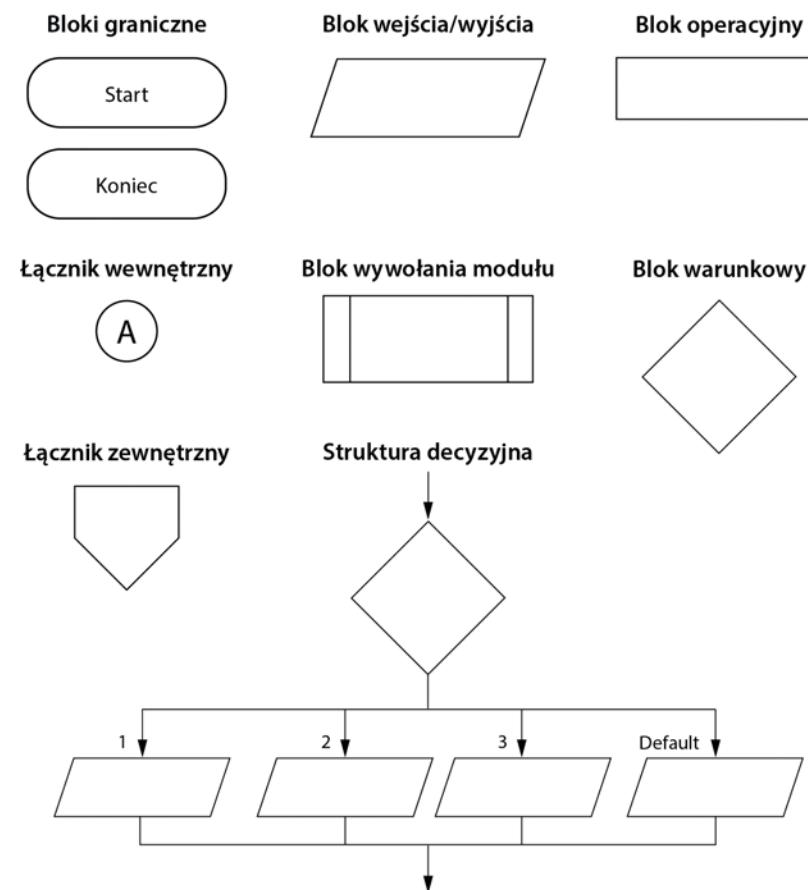
Poniższa tabelka zawiera listę znaków w kodowaniu ASCII (*American Standard Code for Information Interchange*), które wyglądają dokładnie tak samo jak pierwszych 127 znaków w kodowaniu Unicode. Jest to tzw. *Latin Subset of Unicode*, czyli podzbiór zestawu Unicode zawierający znaki łacińskie. W kolumnie *Kod* znajduje się kod znaku, a w kolumnie *Znak* — odpowiadający temu kodowi znak. Przykładowo kod 65 odpowiada literze A. Zwróć uwagę, że pierwszych 31 znaków i kod o numerze 127 odpowiadają znakom sterującym, których nie da się wyświetlić.

| Kod | Znak      | Kod | Znak   | Kod | Znak | Kod | Znak | Kod | Znak |
|-----|-----------|-----|--------|-----|------|-----|------|-----|------|
| 0   | NUL       | 26  | SUB    | 52  | 4    | 78  | N    | 104 | h    |
| 1   | SOH       | 27  | Escape | 53  | 5    | 79  | 0    | 105 | i    |
| 2   | STX       | 28  | FS     | 54  | 6    | 80  | P    | 106 | j    |
| 3   | ETX       | 29  | GS     | 55  | 7    | 81  | Q    | 107 | k    |
| 4   | EOT       | 30  | RS     | 56  | 8    | 82  | R    | 108 | l    |
| 5   | ENQ       | 31  | US     | 57  | 9    | 83  | S    | 109 | m    |
| 6   | ACK       | 32  | Space  | 58  | :    | 84  | T    | 110 | n    |
| 7   | BEL       | 33  | !      | 59  | ;    | 85  | U    | 111 | o    |
| 8   | Backspace | 34  | "      | 60  | <    | 86  | V    | 112 | p    |
| 9   | HTab      | 35  | #      | 61  | =    | 87  | W    | 113 | q    |
| 10  | Line Feed | 36  | \$     | 62  | >    | 88  | X    | 114 | r    |
| 11  | VTab      | 37  | %      | 63  | ?    | 89  | Y    | 115 | s    |
| 12  | Form Feed | 38  | &      | 64  | @    | 90  | Z    | 116 | t    |
| 13  | CR        | 39  | '      | 65  | A    | 91  | [    | 117 | u    |
| 14  | SO        | 40  | (      | 66  | B    | 92  | \    | 118 | v    |
| 15  | SI        | 41  | )      | 67  | C    | 93  | ]    | 119 | w    |
| 16  | DLE       | 42  | *      | 68  | D    | 94  | ^    | 120 | x    |
| 17  | DC1       | 43  | +      | 69  | E    | 95  | _    | 121 | y    |
| 18  | DC2       | 44  | ,      | 70  | F    | 96  | -    | 122 | z    |
| 19  | DC3       | 45  | -      | 71  | G    | 97  | a    | 123 | {    |
| 20  | DC4       | 46  | .      | 72  | H    | 98  | b    | 124 |      |
| 21  | NAK       | 47  | /      | 73  | I    | 99  | c    | 125 | }    |
| 22  | SYN       | 48  | 0      | 74  | J    | 100 | d    | 126 | ~    |
| 23  | ETB       | 49  | 1      | 75  | K    | 101 | e    | 127 | DEL  |
| 24  | CAN       | 50  | 2      | 76  | L    | 102 | f    |     |      |
| 25  | EM        | 51  | 3      | 77  | M    | 103 | g    |     |      |



## Symbole na schematach blokowych

Oto symbole występujące na schematach blokowych zamieszczonych w tej książce:



**782** Dodatek B. Symbole na schematach blokowych



# Przewodnik po pseudokodzie

Ten dodatek pełni rolę przewodnika, w którym zwięzle opisałem polecenia i operatory pseudokodu, jakich używam w tej książce. Nie zawarłem w nim jednak opisu funkcji bibliotecznych — możesz go odnaleźć w treści książki, korzystając ze skorowidza.

| Typy danych | Opis                                          |
|-------------|-----------------------------------------------|
| Integer     | Służy do zapisywania liczb całkowitych        |
| Real        | Służy do zapisywania liczb z częścią ułamkową |
| String      | Służy do zapisywania ciągów znaków            |
| Character   | Służy do zapisywania pojedynczych znaków      |

## Zmienne

Aby zadeklarować zmienną, używamy polecenia `Declare`. Oto jego ogólna postać:

`Declare TypDanych NazwaZmiennej`

*TypDanych* wskazuje typ danych zmiennej, a *NazwaZmiennej* to nazwa, za pomocą której będziemy się odwoływać do zmiennej. Oto kilka przykładów:

```
Declare Integer distance  
Declare Real grossPay  
Declare String name  
Declare Character letter
```

Opcjonalnie możesz także zainicjalizować zmienną określona wartością. Oto przykład:

```
Declare Real price = 49.95
```

## Stałe nazwane

Stałą nazwaną tworzymy za pomocą polecenia `Constant`. Oto jego ogólna postać:

```
Constant TypDanych Nazwa = Wartość
```

*TypDanych* wskazuje typ danych stałej nazwanej, *Nazwa* to nazwa stałej, a *Wartość* to wartość, która zostanie przypisana do stałej. Oto przykład:

```
Constant Real INTEREST_RATE = 0.072
```

## Tablice

Oto ogólna postać deklaracji tablicy:

```
Declare TypDanych NazwaTablicy[Rozmiar]
```

*TypDanych* oznacza typ danych, które będą zapisane w tablicy, *NazwaTablicy* oznacza nazwę tablicy, a *Rozmiar* wskazuje liczbę elementów, które mogą zostać zapisane w tablicy. Oto przykład:

```
Declare Integer units[10]
```

Za pomocą tego polecenia zadeklarowałem tablicę elementów typu Integer. Tablica nazywa się *units* i składa się z 10 elementów.

## Tablice dwuwymiarowe

Oto ogólna postać deklaracji tablicy dwuwymiarowej:

```
Declare TypDanych NazwaTablicy[Wiersze][Kolumny]
```

*TypDanych* oznacza typ danych, które będą zapisane w tablicy, *NazwaTablicy* oznacza nazwę tablicy, *Wiersze* wskazuje liczbę wierszy, a *Kolumny* wskazuje liczbę kolumn, z których będzie się składała tablica. Oto przykład:

```
Declare Integer values[10][20]
```

Za pomocą tego polecenia zadeklarowałem tablicę elementów typu Integer. Tablica nazywa się *values* i składa się z 10 wierszy i 20 kolumn.

## Wyświetlanie danych wyjściowych

Aby wyświetlić dane, używamy polecenia *Display*. Wyświetla ono na ekranie pojedynczą linię. Oto ogólna postać polecenia, po którego wykonaniu wyświetli się na ekranie jeden element:

```
Display Element
```

Aby wyświetlić kilka elementów, musimy oddzielić je przecinkami:

```
Display Element, Element, Element, ...
```

Oto kilka przykładów:

```
Display "Witaj, świecie!"  
Display grossPay  
Display "Mam na imię ", name
```

Za pomocą słowa *Tab* możemy wstawić w linii znaki tabulacji. Oto przykład:

```
Display amount1, Tab, amount2, Tab, amount3
```

## Odczytywanie danych wejściowych

Dane wprowadzone za pomocą klawiatury możemy odczytać za pomocą polecenia Input. Oto jego ogólna postać:

Input *NazwaZmiennej*

*NazwaZmiennej* oznacza zmienną, w której zostanie zapisana wprowadzona wartość. Oto przykład:

Input hours

## Komentarze

W tej książce komentarze rozpoczynamy od podwójnych ukośników (//). Wszystko, co znajduje się w danej linii za ukośnikami, jest traktowane jako komentarz. Oto przykład:

// Pobieramy liczbę przepracowanych godzin

## Operatory matematyczne

| Symbol | Operator    | Opis                                                        |
|--------|-------------|-------------------------------------------------------------|
| +      | Dodawanie   | Dodaje do siebie dwie liczby                                |
| -      | Odejmowanie | Odejmuje jedną liczbę od drugiej                            |
| *      | Mnożenie    | Mnoży przez siebie dwie liczby                              |
| /      | Dzielenie   | Dzieli jedną liczbę przez drugą i zwraca iloraz             |
| MOD    | Modulo      | Dzieli jedną liczbę przez drugą i zwraca resztę z dzielenia |
| ^      | Potęgowanie | Podnosi liczbę do potęgi                                    |

## Operatory relacji

| Operator | Znaczenie              |
|----------|------------------------|
| >        | Większe niż            |
| <        | Mniejsze niż           |
| >=       | Większe niż lub równe  |
| <=       | Mniejsze niż lub równe |
| ==       | Równe                  |
| !=       | Różne od               |

## Operatory logiczne

| Operator | Znaczenie                                                                                                                                                                                                                                                                                            |
|----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AND      | Operator AND łączy dwa wyrażenia boolowskie w jedno złożone wyrażenie. Aby takie złożone wyrażenie zwracało prawdę, oba podwyrażenia także muszą zwracać prawdę.                                                                                                                                     |
| OR       | Operator OR łączy dwa wyrażenia boolowskie w jedno złożone wyrażenie. Aby takie złożone wyrażenie zwracało prawdę, jedno z dwóch podwyrażeń musi zwracać prawdę. Wystarczy, że tylko jedno, którekolwiek z wyrażeń będzie zwracało prawdę.                                                           |
| NOT      | Operator NOT jest operatorem jednoargumentowym, co znaczy, że wymaga tylko jednego operandu. Operand musi być wyrażeniem boolowskim. Operator NOT neguje wartość wyrażenia — jeżeli wyrażenie zwraca prawdę, operator NOT zwróci fałsz, a jeżeli wyrażenie zwraca fałsz, operator NOT zwróci prawdę. |

## Instrukcja If-Then

Ogólna postać instrukcji If-Then:

```
If warunek Then
    instrukcja
    instrukcja
    itd.
End If
```

**Te instrukcje zostaną wykonane warunkowo.**

W takiej ogólnej postaci *warunek* oznacza dowolne wyrażenie boolowskie. Jeżeli *warunek* jest spełniony, uruchomią się *instrukcje* zawarte pomiędzy klauzulami If i End If. W przeciwnym przypadku *instrukcje* zostaną pominięte. Oto przykład:

```
If sales > 50000 Then
    Set bonus = 500.0
    Set commissionRate = 0.12
    Display "Wykonałeś plan!"
End If
```

## Instrukcja If-Then-Else

Oto ogólna postać instrukcji If-Then-Else:

```
If warunek Then
    instrukcja
    instrukcja
    itd.
Else
    instrukcja
    instrukcja
    itd.
End If
```

**Te instrukcje wykonają się, gdy warunek zostanie spełniony**

**Te instrukcje wykonają się, gdy warunek nie zostanie spełniony**

Warunek oznacza tutaj dowolne wyrażenie boolowskie. Jeśli wyrażenie to zwróci prawdę, wykonają się instrukcje zawarte między klauzulami If i Else. W przeciwnym przypadku wykonają się instrukcje zawarte między klauzulami Else i End If. Oto przykład:

```
If temperature < 40 Then
    Display "Mamy dzisiaj żadną pogodę."
Else
    Display "Ależ dzisiaj chłodno!"
End If
```

## Instrukcja Select Case

Oto ogólna postać instrukcji Select Case:

```
Select wyrażenie ← Tutaj jest wyrażenie lub zmienna.
Case wartość_1:
    instrukcja
    instrukcja
    itd. } Te instrukcje wykonają się wtedy,
          gdy wyrażenie będzie równe wartość_1.

Case wartość_2:
    instrukcja
    instrukcja
    itd. } Te instrukcje wykonają się wtedy,
          gdy wyrażenie będzie równe wartość_2.

tutaj możesz wstawić tyle klauzul Case, ile będziesz potrzebował

Case wartość_N:
    instrukcja
    instrukcja
    itd. } Te instrukcje wykonają się wtedy,
          gdy wyrażenie będzie równe wartość_N.

Default:
    instrukcja
    instrukcja
    itd. } Te instrukcje wykonają się wtedy, gdy wyrażenie nie będzie
          równe żadnej z wartości wymienionych wcześniej.

End Select ← Tutaj instrukcja się kończy.
```

Pierwsza linia instrukcji zaczyna się od słowa Select, a następnie pojawia się *wyrażenie*. Wewnątrz instrukcji znajduje się jeden lub więcej bloków instrukcji zaczynających się od słowa Case. Zwróć uwagę, że za słowem Case umieszczona jest wartość.

Kiedy uruchomisz się instrukcję Select Case, *wyrażenie* zostanie porównane z wartością znajdującej się przy instrukcji Case — z góry na dół. Jeśli okaże się, że jedna z tych wartości jest taka sama jak wartość zwracana przez *wyrażenie*, program przejdzie do tej instrukcji, zacznie uruchamiać instrukcje umieszczone bezpośrednio pod nią, a następnie opuści strukturę. Jeżeli okaże się, że żadna z wartości nie jest równa wartości zwracanej przez *wyrażenie*, program przejdzie do instrukcji Default i zacznie wykonywać instrukcje znajdujące się pod nią. Oto przykład:

```
Select month
Case 1:
    Display "Styczeń"
Case 2:
    Display "Luty"
Case 3:
    Display "Marzec"
Default:
    Display "Błąd: nieprawidłowy miesiąc"
End Select
```

## Pętla While

Oto ogólna postać pętli While:

```
While warunek
    instrukcja
    instrukcja
    itd.
End While
```

**Te instrukcje stanowią ciało pętli.  
Będą one wykonywane dopóty, dopóki spełniony będzie warunek.**

Warunek oznacza dowolne wyrażenie boolowskie, a instrukcje zawarte między klauzulami While i End While nazywamy ciałem pętli. Kiedy uruchomisz się pętlą, sprawdzony zostanie warunek. Jeżeli jest on spełniony, zostaną uruchomione instrukcje składające się na ciało pętli, a następnie pętla uruchomisz się ponownie. Jeżeli warunek nie jest spełniony, program opuści pętlę. Na poniższym pseudokodzie przedstawiłem przykładową pętlę While:

```
Set count = 0
While count < 10
    Display count
    Set count = count + 1
End While
```

## Pętla Do-While

Oto ogólna postać pętli Do-While:

```
Do
    instrukcja
    instrukcja
    itd.
While warunek
```

**Te instrukcje stanowią ciało pętli. W każdym przypadku zostaną wykonane co najmniej jeden raz, a jeżeli warunek jest spełniony, zostaną wykonane ponownie.**

Instrukcje znajdujące się pomiędzy klauzulami Do i While stanowią ciało pętli. Warunek, który pojawi się po klauzuli While, jest wyrażeniem boolowskim. Po uruchomieniu pętli program wykona instrukcje zawarte w ciele pętli, a następnie sprawdzi warunek. Jeżeli warunek jest spełniony, pętla uruchomisz się ponownie i program ponownie wykona instrukcje umieszczone w ciele pętli. Jeżeli warunek nie jest spełniony, program wyjdzie z pętli. Oto przykład:

```
Set count = 10
Do
    Display count
    Set count = count - 1
    While count > 0
```

## Pętla Do-Until

Oto ogólna postać pętli Do-Until:

```
Do
    instrukcja
    instrukcja
    itd.
Until warunek
```

**Te instrukcje stanowią ciało pętli. W każdym przypadku zostaną wykonane co najmniej jeden raz i będą wykonywane powtórnie, aż do momentu, gdy warunek zostanie spełniony.**

Instrukcje znajdujące się pomiędzy klauzulami Do i Until stanowią ciało pętli. Warunek, który pojawia się po klauzuli Until, jest wyrażeniem boolowskim. Po uruchomieniu pętli program wykona instrukcje zawarte w ciele pętli, a następnie sprawdzi warunek. Jeżeli warunek jest spełniony, program wyjdzie z pętli. Jeżeli warunek nie jest spełniony, pętla uruchomi się ponownie i program ponownie wykona instrukcje umieszczone w ciele pętli. Oto przykład:

```
Set count = 10
Do
    Display count
    Set count = count - 1
Until count == 0
```

## Pętla For

Oto ogólna postać pętli For:

```
For zmienialicznikowa = wartośćPoczątkowa To wartośćMaksymalna
    instrukcja
    instrukcja } Te instrukcje stanowią ciało pętli.
    instrukcja
    itd.
End For
```

Zmienialicznikowa to nazwa zmiennej, która będzie pełniła funkcję zmiennej licznikowej, wartośćPoczątkowa to wartość, jaką zmienna ta zostanie zainicjalizowana, a wartośćMaksymalna to maksymalna wartość, jaką może przyjąć zmienna licznikowa. Kiedy uruchomi się pętla, program wykona następujące operacje:

1. Do zmiennej zmienialicznikowa zostanie przypisana wartośćPoczątkowa.
2. Wartość przypisana do zmiennej zmienialicznikowa zostanie porównana z wartością wartośćMaksymalna. Jeżeli zmienialicznikowa jest większa niż wartośćMaksymalna, program opuści pętlę. W przeciwnym wypadku:
  - a) wykonane zostaną instrukcje umieszczone w ciele pętli;
  - b) zmienialicznikowa zostanie zwiększeniowana;
  - c) pętla powróci do kroku 2.

Oto przykład:

```
For counter = 1 To 10
    Display "Witaj, świecie!"
```

End For

## Pętla For Each

Oto ogólna postać pętli For Each:

```
For Each zmienna In tablica
    instrukcja
    instrukcja
    instrukcja
    itd.
End For
```

Zmienna oznacza nazwę zmiennej, a tablica to nazwa tablicy. Pętla taka będzie przetwarzała po kolejni każdy element danej tablicy. Podczas każdej iteracji do zmiennej zmienna zostanie przypisana wartość równa kolejnemu elementowi tablicy. Przykładowo w pierwszej iteracji do zmiennej zmienna zostanie przypisana wartość elementu tablica[0], w drugiej iteracji w zmiennej tej znajdzie się element tablica[1] itd. Proces ten będzie się powtarzał, aż pętla dojdzie do ostatniego elementu tablicy. Oto przykładowy pseudokod:

```
Constant Integer SIZE = 5
Declare Integer numbers[SIZE] = 5, 10, 15, 20, 25
Declare Integer num
For Each num In numbers
    Display num
End For
```

## Definiowanie modułu

Aby stworzyć moduł, należy napisać jego definicję, która składa się z dwóch elementów: nagłówka i ciała. Nagłówek określa punkt początkowy modułu, a ciało to lista instrukcji wchodzących w skład modułu. Oto ogólny schemat modułu, zapisany za pomocą pseudokodu:

```
Module NazwaModułu()
    instrukcja
    instrukcja
    itd.
End Module
```

**Te instrukcje to ciało modułu.**

## Wywoływanie modułu

Do wywoływania modułu używamy polecenia Call. Oto jego ogólna postać:

```
Call NazwaModułu()
```

*NazwaModułu* to nazwa modułu, który mamy zamiar wywołać. Oto przykład wywołania modułu o nazwie showMessage:

```
Call showMessage()
```

## Parametry

Jeśli chcesz, aby funkcja lub moduł były w stanie odebrać przekazane do nich argumenty, musisz wyposażyć je w jeden lub kilka parametrów. Parametr deklarujemy w nawiasach podczas definiowania modułu. Oto przykład modułu z parametrem typu Integer, zapisany za pomocą pseudokodu:

```
Module doubleNumber(Integer value)
    Declare Integer result
    Set result = value * 2
    Display result
End Module
```

Podczas wywołania tego modułu należy przekazać do niego wartość typu Integer. Aby przekazać do modułu argument przez referencję, w deklaracji parametru należy użyć słowa Ref:

```
Module setToZero(Integer Ref value)
    value = 0
End Module
```

## Definiowane funkcji

Definiowanie funkcji wygląda podobnie jak definiowanie modułu. Oto ogólna postać definicji funkcji:

```
Function TypDanych NazwaFunkcji(ListaParametrów)
    instrukcja } W funkcji musi się znajdować polecenie Return. Dzięki niemu wskazana
    instrukcja } wartość zostanie przesłana do tej części programu, w której nastąpiło
    itd.         } wywołanie funkcji.

    Return wartość
End Function
```

W pierwszej linii znajduje się nagłówek funkcji składający się ze słowa Function, a potem pojawiają się następujące elementy:

- *TypDanych* wskazuje, jakiego typu wartości zwraca funkcja.
- *NazwaFunkcji* wskazuje, jaką nazwę ma funkcja.
- Opcjonalnie możesz wskazać w nawiasach listę parametrów. Jeżeli funkcja nie przyjmuje żadnych argumentów, pozostaw puste nawiasy.

Po linii z nagłówkiem funkcji pojawia się co najmniej jedna instrukcja. Instrukcje te składają się na ciało funkcji i zostaną wykonane po jej wywołaniu. Jedną z tych instrukcji musi być Return, która wygląda następująco:

```
Return wartość
```

Wartość umieszczona po słowie Return to wartość, którą zwróci funkcja w miejscu jej wywołania. Oto przykład:

```
Function Integer sum(Integer num1, Integer num2)
    Declare Integer result
    Set result = num1 + num2
    Return result
End Function
```

## Otwieranie pliku i zapisywanie w nim danych

W pseudokodzie najpierw deklarujemy obiekt reprezentujący plik, a potem otwieramy plik za pomocą polecenia Open. Oto przykład:

```
Declare OutputFile customerFile
Open customerFile "customer.dat"
```

Następnie możemy zapisać dane w pliku za pomocą polecenia Write:

```
Write customerFile "Konrad Grzesiak"
```

Kiedy program zakończy przetwarzać plik, należy taki plik zamknąć za pomocą polecenia **Close**. Oto przykład:

```
Close customerFile
```

## Otwieranie pliku i odczytywanie z niego danych

W pseudokodzie najpierw deklarujemy obiekt reprezentujący plik, a potem otwieramy plik za pomocą polecenia **Open**. Oto przykład:

```
Declare inputFile inventoryFile
Open inventoryFile "inventory.dat"
```

Następnie możemy odczytać dane z pliku za pomocą polecenia **Read**:

```
Read inventoryFile itemName
```

Polecenie to odczyta z pliku jeden element i zapisze go w zmiennej **itemName**. Kiedy program zakończy przetwarzać plik, należy taki plik zamknąć za pomocą polecenia **Close**. Oto przykład:

```
Close inventoryFile
```

## Wykrywanie końca pliku

W pseudokodzie do wykrywania końca pliku używamy funkcji **eof**. Ma ona następującą postać:

```
eof(nazwaObiektuReprezentującegoPlik)
```

Funkcja **eof** przyjmuje jako argument obiekt reprezentujący plik i kiedy program odczyta już wszystkie dane zapisane w pliku, zwraca wartość **True**. Gdy w pliku znajdują się jeszcze jakieś dane, funkcja **eof** zwraca wartość **False**. Oto przykładowy pseudokod:

```
// Deklarujemy plik wejściowy
Declare inputFile salesFile

// Deklarujemy zmienną, w której zapiszemy wartość sprzedazy
// odczytaną z pliku
Declare Real sales

// Otwieramy plik sales.dat
Open salesFile "sales.dat"

// Odczytujemy wszystkie wartości z pliku
// i wyświetlamy je
While NOT eof(salesFile)
    Read salesFile sales
    Display currencyFormat(sales)
End While

// Zamkamy plik
Close salesFile
```

## Usuwanie pliku

Aby usunąć plik, korzystamy z polecenia Delete. Oto jego ogólna postać:

```
Delete NazwaPliku
```

*NazwaPliku* oznacza nazwę pliku znajdującego się na dysku komputera. Oto przykład:

```
Delete "customers.dat"
```

## Zmienianie nazwy pliku

Aby zmienić nazwę pliku, korzystamy z polecenia Rename. Oto jego ogólna postać:

```
Rename AktualnaNazwaPliku, NowaNazwaPliku
```

*AktualnaNazwaPliku* oznacza nazwę pliku znajdującego się na dysku komputera, a *NowaNazwaPliku* to nazwa, którą będzie miał plik po zmianie. Oto przykład:

```
Rename "temp.dat", "customers.dat"
```

## Definiowanie klasy

Oto ogólna postać deklaracji klasy:

```
Class NazwaKlasy
    deklaracje pól i definicje metod...
End Class
```

Pierwsza linia rozpoczyna się od słowa Class, a po nim jest nazwa klasy. Następnie deklarujemy pola klasy i definiujemy metody klasy. Definicję klasy kończą słowa End Class. Oto przykład definicji klasy CellPhone, którą omówilem w rozdziale 14.:

```
Class CellPhone
    // Deklarujemy pola
    Private String manufacturer
    Private String modelNumber
    Private Real retailPrice

    // Definiujemy metody
    Public Module setManufacturer(String manufact)
        Set manufacturer = manufact
    End Module

    Public Module setModelNumber(String modNum)
        Set modelNumber = modNum
    End Module

    Public Module setRetailPrice(Real retail)
        Set retailPrice = retail
    End Module

    Public Function String getManufacturer()
        Return manufacturer
    End Function
```

```

Public Function String getModelNumber()
    Return modelNumber
End Function

Public Function Real getRetailPrice()
    Return retailPrice
End Function
End Class

```

## Tworzenie instancji klasy

Aby utworzyć instancję klasy (czyli obiekt), deklarujemy zmienną, za pomocą której będziemy się do tego obiektu odnosić, a następnie używamy operatora `New`, aby utworzyć instancję klasy. Oto przykład, w którym tworzę instancję przedstawionej wcześniej klasy `CellPhone`:

```

Declare CellPhone myPhone
Set myPhone = New CellPhone()

```

W pierwszym poleceniu deklaruję zmienną o nazwie `myPhone`. W drugim poleceniu tworzę instancję klasy `CellPhone` i przypisuję jej adres w pamięci komputera do zmiennej `myPhone`.

## Moduł Init w aplikacji GUI

Jeżeli w aplikacji GUI znajduje się moduł `Init`, to zostanie on automatycznie wykonyany podczas uruchamiania tej aplikacji. Moduł `Init` jest zapisywany w następującym ogólnym formacie:

```

Module Init()
    Wyświetlane tutaj instrukcje są wykonywane,
    kiedy aplikacja zacznie działać.
End Module

```

## Komponenty GUI

Kilka komponentów GUI wymieniłem w tabeli 15.1. Jednak w naszych przykładach używane są następujące komponenty:

- `Button` — komponent, który wywołuje operację po kliknięciu. Udostępnia on właściwość `Text`, która definiuje tekst wyświetlany na przycisku.
- `Label` — obszar, w którym można wyświetlać tekst. Komponent `Label` ma właściwość `Text`, która definiuje tekst wyświetlany przez ten komponent.
- `TextBox` — obszar, w którym użytkownik może wpisać z klawiatury pojedynczy wiersz tekstu. Komponent `TextBox` ma właściwość `Text`, która przechowuje dane wprowadzone do komponentu przez użytkownika.

## Komponenty urządzeń mobilnych

W rozdziale 15. omówiłem następujące komponenty, które są szczególnymi elementami składowymi urządzeń mobilnych:

- `PhoneCall` — niewizualny komponent odpowiedzialny za wykonywanie i odbieranie połączeń telefonicznych.
- `TextMessage` — niewizualny komponent odpowiedzialny za wysyłanie i odbieranie wiadomości SMS.
- `Location` — niewizualny komponent, który może podawać aktualną szerokość i długość geograficzną lokalizacji urządzenia oraz najbliższy adres.

## Nawiązywanie połączenia telefonicznego na urządzeniu mobilnym

Aby wykonać połączenie telefoniczne na urządzeniu mobilnym, wywołujemy metodę `makeCall` komponentu `PhoneCall`. Metoda ta przyjmuje argument w postaci ciągu znaków, który zawiera numer telefonu do wywołania. Oto przykład:

```
PhoneCall.makeCall("605-555-212")
```

## Wysyłanie wiadomości tekstowej z urządzenia mobilnego

Aby wysłać wiadomość tekstową z urządzenia mobilnego, wywołujemy metodę `sendText` komponentu `TextMessage`. Metoda ta pobiera dwa argumenty: (1) ciąg znaków zawierający numer telefonu, na który wiadomość ma być wysłana, i (2) ciąg znaków zawierający samą wiadomość. Oto ogólny format takiej instrukcji:

```
TextMessage.sendText(NumerTelefonu, Wiadomość)
```

## Uzyskanie aktualnej lokalizacji urządzenia mobilnego

Komponent `Location` udostępnia następujące metody, których można użyć, aby uzyskać lokalizację urządzenia:

- `getLatitude()` — zwraca szerokość geograficzną lokalizacji urządzenia jako liczbę typu `Real`.
- `getLongitude()` — zwraca długość geograficzną lokalizacji urządzenia jako liczbę typu `Real`.
- `getAddress()` — zwraca串 znaków, który zawiera adres znajdujący się najbliżej lokalizacji urządzenia.

## Obsługa zdarzeń GUI

Nasze pseudokodowe procedury obsługi zdarzeń są napisane w poniższym, ogólnym formacie:

```
Module NazwaKomponentu_NazwaZdarzenia()
    Wyświetlane tutaj instrukcje są
    wykonywane po wystąpieniu zdarzenia.
End Module
```

W ogólnym formacie *NazwaKomponentu* jest nazwą komponentu, który wygenerował zdarzenie, a *NazwaZdarzenia* jest nazwą zdarzenia, które miało miejsce. W tej książce omówię następujące zdarzenia GUI:

- **Click** — występuje po kliknięciu komponentu Button.
- **IncomingCall** — występuje w smartfonie, gdy urządzenie odbiera przychodzące połączenie telefoniczne.
- **Incoming\_Text** — występuje w smartfonie, gdy urządzenie odbiera przychodząca wiadomość SMS.
- **LocationChanged** — występuje w urządzeniu mobilnym po zmianie jego lokalizacji.

Załóżmy, że aplikacja GUI ma komponent Button o nazwie calculateButton. Procedura obsługi zdarzenia Click dla tego komponentu zostanie zapisana w następującym, ogólnym formacie:

```
Module calculateButton_Click()
    Wyświetlane tutaj instrukcje są
    wykonywane po wystąpieniu zdarzenia.
End Module
```

W aplikacji mobilnej procedura obsługi zdarzenia IncomingCall zostanie zapisana w następującym, ogólnym formacie:

```
Module PhoneCall_IncomingCall()
    Wyświetlane tutaj instrukcje są
    wykonywane po wystąpieniu zdarzenia.
End Module
```



## Zamiana liczb dziesiętnych na postać binarną

Aby zamienić liczbę w systemie dziesiętnym na liczbę w systemie binarnym (dwójkowym), możesz narysować prostą tabelkę, a następnie wykonać kilka operacji odejmowania. Oto kroki, dzięki którym zamienisz na postać binarną liczbę 162:

1. Narysuj tabelkę zawierającą dwa wiersze. W górnym wierszu wstaw liczby równe kolejnym potęgom liczby 2, aż dojdiesz do liczby, którą masz zamiar zamienić, jednak ostatnia potęga nie może być większa od tej liczby. Kolejne potęgi liczby 2 to:  $2^0 = 1$ ,  $2^1 = 2$ ,  $2^2 = 4$ ,  $2^3 = 8$  itd. Liczby powinny być umieszczone w kolejności od prawej strony do lewej.

Aby zamienić na postać binarną liczbę 162, stworzymy taką oto tabelkę (wartości w górnym wierszu kończą się na liczbie 128, ponieważ kolejna potęga liczby 2 jest równa 256, czyli jest to liczba większa od 162):

|     |    |    |    |   |   |   |   |
|-----|----|----|----|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
|     |    |    |    |   |   |   |   |

2. Odszukaj w tabelce największą wartość (w tym przypadku 128) i w dolnym wierszu przypisz do niej 1. Następnie odejmij tę wartość od liczby, którą masz zamiar zamienić na postać binarną. W tym przypadku otrzymamy wynik równy 34. Będzie to nasza nowa liczba. Tabelka wygląda na tym etapie tak:

|     |    |    |    |   |   |   |   |
|-----|----|----|----|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 1   |    |    |    |   |   |   |   |

$$\begin{array}{r} 162 \\ -128 \\ \hline 34 \end{array}$$

3. Odszukaj w tabelce najwyższą wartość, która jest nie większa niż liczba otrzymana w kroku 2. W tym przypadku szukamy największej wartości nie większej niż 34. Będzie to liczba 32. W dolnym wierszu zapisz przy tej liczbie 1. Następnie

**798 Dodatek D. Zamiana liczb dziesiętnych na postać binarną**

odejmij tę liczbę od liczby, nad którą w tej chwili pracujemy. W tym przypadku wynik będzie równy 2. Będzie to nasza nowa liczba. Tabelka wygląda na tym etapie tak:

|     |    |    |    |   |   |   |   |
|-----|----|----|----|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 1   |    | 1  |    |   |   |   |   |

$$\begin{array}{r} 162 \\ -128 \\ \hline 34 \\ -32 \\ \hline 2 \end{array}$$

4. Odszukaj w tabelce najwyższą wartość, która jest nie większa niż liczba otrzymana w kroku 3. W tym przypadku szukamy największej wartości nie większej niż 2. Będzie to liczba 2. W dolnym wierszu zapisz przy tej liczbie 1. Następnie odejmij tę liczbę od liczby, nad którą w tej chwili pracujemy. W tym przypadku wynik będzie równy 0. Będzie to nasza nowa liczba. Tabelka wygląda na tym etapie tak:

|     |    |    |    |   |   |   |   |
|-----|----|----|----|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 1   |    | 1  |    |   |   | 1 |   |

$$\begin{array}{r} 162 \\ -128 \\ \hline 34 \\ -32 \\ \hline 2 \\ -2 \\ \hline 0 \end{array}$$

5. Kiedy w wyniku odejmowania otrzymamy 0, oznacza to, że zakończyliśmy operację zamiany liczby dziesiętnej na liczbę binarną. Wpisz teraz 0 we wszystkich pustych komórkach tabelki. Powinna ona wyglądać teraz tak:

|     |    |    |    |   |   |   |   |
|-----|----|----|----|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 1   | 0  | 1  | 0  | 0 | 0 | 1 | 0 |

W drugim wierszu widoczny jest binarny odpowiednik liczby 162: 10100010.

# Odpowiedzi do pytań z punktów kontrolnych

## Rozdział 1.

- 1.1. Program komputerowy to zbiór instrukcji, które musi uruchomić komputer, aby wykonać określone zadanie.
- 1.2. Sprzęt to wszystkie urządzenia fizyczne (komponenty), z których zbudowany jest komputer.
- 1.3. Procesor, pamięć główna, nośniki danych, urządzenia wejściowe i urządzenia wyjściowe.
- 1.4. Procesor.
- 1.5. Pamięć główna.
- 1.6. Nośnik danych.
- 1.7. Urządzenie wejściowe.
- 1.8. Urządzenie wyjściowe.
- 1.9. Jeden bajt.
- 1.10. Bit.
- 1.11. System binarny.
- 1.12. Kodowanie ASCII to zestaw 128 kodów, które reprezentują litery w języku angielskim, znaki przestankowe, a także kilka innych znaków. Za ich pomocą można zapisać w pamięci komputera znaki. ASCII jest skrótem nazwy *American Standard Code for Information Interchange*.
- 1.13. Unicode.
- 1.14. Dane cyfrowe to dane zapisane za pomocą systemu binarnego. Urządzenie cyfrowe to urządzenie, które wykorzystuje podczas działania dane binarne.
- 1.15. W języku maszynowym.
- 1.16. Do pamięci głównej.
- 1.17. Dochodzi do tzw. cyklu rozkazowego.

- 1.18. Język asemblera jest alternatywą dla języka maszynowego. Zamiast z liczb w systemie binarnym korzysta się w nim z kilkuliterowych skrótów zwanych mnemonikami.
- 1.19. Języki wysokiego poziomu.
- 1.20. Składnia.
- 1.21. Kompilator.
- 1.22. Interpreter.
- 1.23. Błąd składniowy.
- 1.24. System operacyjny.
- 1.25. Programy narzędziowe.
- 1.26. Oprogramowanie użytkowe.

## Rozdział 2.

- 2.1. To osoba, grupa osób lub przedsiębiorstwo, które zlecają stworzenie danego programu.
- 2.2. To pojedyncza funkcja, którą będzie wykonywał program.
- 2.3. Algorytm to lista ścisłe określonych operacji, które należy wykonać, aby zrealizować określone zadanie.
- 2.4. Pseudokod to nieformalny język, który nie ma żadnej składni i nie służy do tego, aby go kompilować i uruchamiać. Programiści używają go do tworzenia modeli lub „makiet” programów.
- 2.5. Schemat blokowy to diagram, w którym za pomocą symboli graficznych przedstawia się kolejne operacje, jakie musi wykonać program.
- 2.6. Owale to bloki graniczne. Równoległy boki to bloki wejścia/wyjścia. Prostokąty to bloki operacyjne.
- 2.7. Przyjmuje dane wejściowe, przetwarza je i generuje dane wyjściowe.
- 2.8. Tabelka IPO prezentuje dane wejściowe i wyjściowe oraz przetwarzanie danych, jakie ma mieć miejsce w projektowanym programie.
- 2.9. Strukturą sekwencyjną nazywamy grupę instrukcji, które będą uruchamiane w takiej kolejności, w jakiej zostały zapisane w programie.
- 2.10. Ciąg znaków to dane w postaci sekwencji znaków. Literał ciągu znaków to ciąg znaków, który występuje w kodzie programu.
- 2.11. Znakami cudzysłówów.
- 2.12. Zmienna to miejsce w pamięci komputera, które jest reprezentowane przez określoną nazwę.
- 2.13. • Nazwa zmiennej musi się składać z jednego słowa (nie może zawierać znaków spacji).
  - W większości języków w nazwie nie mogą występować znaki przestankowe. Dobrą praktyką jest wykorzystywanie w nazwach zmiennych tylko liter i liczb.
  - W większości języków pierwszym znakiem w nazwie zmiennej nie może być cyfra.

2.14. camelCase.

- 2.15. • Program wstrzyma działanie i będzie czekał, aż użytkownik wprowadzi na klawiaturze dowolną wartość i naciśnie klawisz *Enter*.
- Po naciśnięciu klawisza *Enter* wartość wprowadzona przez użytkownika zostanie zapisana w zmiennej *temperature*.

2.16. Użytkownik to potencjalna osoba, która będzie używała danego programu i dostarczała do niego dane wejściowe.

2.17. Komunikat to wiadomość, która wskazuje użytkownikowi, że musi wprowadzić jakąś wartość.

2.18. Należy:

1. Wyświetlić na ekranie komunikat.
2. Odczytać wprowadzoną na klawiaturze wartość.

2.19. Program „przyjazny dla użytkownika” to program, który jest łatwy w obsłudze.

2.20. Polecenie przypisania zapisuje w zmiennej określonej wartości.

2.21. Zostanie zastąpiona nową wartością.

2.22. 1. Operacje umieszczone w nawiasach.

2. Operacje podnoszenia do potęgi.
3. Operacje mnożenia, dzielenia i modulo, od lewej do prawej strony.
4. Operacje dodawania i odejmowania, od lewej do prawej strony.

2.23. Operator potęgowania podnosi liczbę do danej potęgi.

2.24. Operator modulo zwraca resztę z dzielenia.

2.25. Nazwę zmiennej i typ danych.

2.26. Tak. Deklarację zmiennej należy umieścić w kodzie przed miejscem, w którym będziemy się do niej odwoływać.

2.27. Inicjalizacja zmiennej oznacza przypisanie do niej wartości w momencie jej deklarowania.

2.28. Tak. Niezainicjalizowane zmienne są źródłem wielu błędów. Kiedy użyjemy takiej zmiennej na przykład do obliczeń, w programie pojawi się błąd.

2.29. Niezainicjalizowana zmienna to zmienna, która została zadeklarowana, ale nie została do niej przypisana żadna wartość.

2.30. Dokumentacja zewnętrzna jest zazwyczaj przeznaczona dla użytkownika końcowego. Składają się na nią takie kwestie jak opis funkcji programu czy samouczki służące do zaznajomienia użytkownika z programem.

2.31. Dokumentacja wewnętrzna ma formę komentarzy w kodzie programu. Komentarze to krótkie informacje tekstowe, umieszczone w różnych miejscach kodu, informujące o tym, jak dany fragment kodu działa.

2.32. Programiści umieszczają zazwyczaj w kodzie komentarze blokowe i liniowe. Komentarze blokowe zajmują wiele linii kodu i służą do szczegółowego opisania fragmentu kodu. Komentarze liniowe zajmują tylko jedną linię i służą do objaśnienia krótkiego fragmentu kodu.

## Rozdział 3.

- 3.1. Moduł to zbiór instrukcji, który służy do wykonania określonego zadania w programie.
- 3.3. Metoda „dziel i zwyciężaj” polega na podzieleniu większego zadania na kilka mniejszych, które jest znacznie łatwiej wykonać.
- 3.3. Jeżeli określona operacja jest wykonywana w kilku miejscach programu, można ją ująć w osobnym module i uruchamiać moduł wtedy, gdy będzie taka konieczność.
- 3.4. Można stworzyć moduły, które będą wykonywały pewne wspólne dla wielu programów operacje. Raz utworzony moduł można następnie dołączać do wielu różnych programów.
- 3.5. W sytuacji, gdy tworzony program jest podzielony na moduły wykonujące poszczególne zadania, do pracy nad każdym z modułów można oddelegować innego programistę.
- 3.6. W większości języków programowania definicja modułu składa się z dwóch elementów: nagłówka i ciała. Nagłówek określa punkt początkowy modułu, a ciało to lista instrukcji wchodzących w skład modułu.
- 3.7. Wywołanie modułu oznacza wykonanie znajdujących się w nim poleceń.
- 3.8. Kiedy moduł się skończy, komputer przeskakuje pod zapisany wcześniej adres powrotny i kontynuuje wykonywanie programu.
- 3.9.
  - Zadanie główne, jakie ma wykonać program, dzielimy na mniejsze podzadania.
  - Każde z podzadań analizujemy i sprawdzamy, czy nie da się go podzielić na jeszcze mniejsze podzadania. Powtarzamy ten krok dopóty, dopóki jesteśmy w stanie dzielić zadanie na kolejne podzadania.
  - Kiedy wszystkie podzadania zostały określone, przystępujemy do pisania kodu każdego z nich.
- 3.10. Zmienna lokalna to zmienna zadeklarowana wewnątrz modułu. Należy ona tylko i wyłącznie do modułu, w którym została zadeklarowana, i mogą z niej korzystać tylko instrukcje wewnątrz tego modułu.
- 3.11. Zasięg widoczności zmiennej to ta część programu, w której można się do tej zmiennej odwoływać.
- 3.12. Nie można. Kompilator lub interpreter nie wiedziałby, której ma użyć, gdy nastąpi odwołanie do jednej z nich.
- 3.13. Można.
- 3.14. Argumenty.
- 3.15. Parametry.
- 3.16. Tak. Jeśli spróbujesz przekazać do modułu argument innego typu niż typ parametru, program zgłosi błąd.
- 3.17. Zasięg widoczności parametru obejmuje zazwyczaj cały moduł, w którym parametr został zdefiniowany.
- 3.18. Przekazywanie argumentu przez wartość oznacza, że do parametru trafi kopią przekazanej wartości. Jeśli wewnątrz modułu zmienisz wartość zapisaną

w parametrze, nie będzie to miało wpływu na wartość argumentu. Przekazywanie argumentu przez referencję oznacza, że argument zostanie przekazany do specjalnego typu parametru zwanego zmienną typu referencyjnego. Kiedy użyjemy jako parametru zmiennej typu referencyjnego, moduł będzie mógł modyfikować wartość argumentu zdefiniowanego na zewnątrz tego modułu.

- 3.19. Zasięg widoczności zmiennej globalnej obejmuje cały program.
- 3.20. • Zmienne globalne utrudniają debugowanie programu. Wartość przypisaną do zmiennej globalnej może zmienić każde polecenie w programie. Jeśli okaże się, że w zmiennej globalnej jest nie taka wartość, jakiej się spodziewaliśmy, będziemy musieli prześledzić każdą instrukcję, która może modyfikować tę wartość, aby znaleźć miejsce, w którym jest błąd. Może to być kłopotliwe, zwłaszcza jeżeli program składa się z kilku tysięcy linii kodu.
- Moduły, które korzystają ze zmiennych globalnych, są od nich zależne. Jeśli zechcesz taki moduł wykorzystać w innym programie, najprawdopodobniej będziesz go musiał przeprojektować, aby nie odwoływał się do tej zmiennej globalnej.
  - Zmienne globalne sprawiają, że program jest mniej zrozumiały. Zmienną globalną może zmodyfikować każde polecenie w programie. Jeśli będziesz chciał zrozumieć fragment kodu, w którym pojawia się odwołanie do zmiennej globalnej, będziesz też musiał sprawdzić inne miejsca w programie, w których następuje odwołanie do tej zmiennej.
- 3.21. Stała globalna to taka stała nazwana, do której ma dostęp każdy moduł w programie. W programie powinno się korzystać ze stałych globalnych. Ponieważ nie da się zmienić w programie wartości przypisanej do stałej, nie musisz się martwić zagrożeniami wynikającymi z korzystania ze zmiennych globalnych.

## Rozdział 4.

- 4.1. Struktura sterująca to konstrukcja logiczna, której celem jest sterowanie kolejnością, w jakiej będzie uruchamiana grupa instrukcji.
- 4.2. Struktura warunkowa wykonuje pewne instrukcje tylko wtedy, gdy spełniony zostanie określony warunek.
- 4.3. W strukturze warunkowej pojedynczego wyboru możemy wybrać jedną ścieżkę alternatywną — jeśli warunek jest spełniony, program zostanie skierowany na ścieżkę alternatywną.
- 4.4. Wyrażenie boolowskie to wyrażenie, które zwraca wynik typu prawda lub fałsz.
- 4.5. Za pomocą operatorów relacji można sprawdzić, czy jedna wartość jest większa, mniejsza, większa lub równa, mniejsza lub równa, równa albo różna od drugiej wartości.
- 4.6. 

```
If y == 20 Then
    Set x = 0
End If
```
- 4.7. 

```
If sales >= 10000 Then
    Set commission = 0.2
End If
```

- 4.8. W strukturze warunkowej podwójnego wyboru istnieją dwie ścieżki, którymi może podążyć program. Pierwsza z nich dotyczy sytuacji, gdy warunek zostanie spełniony, a druga — gdy warunek nie zostanie spełniony.
- 4.9. If-Then-Else.
- 4.10. Kiedy warunek nie zostanie spełniony.
- 4.11. z nie jest mniejsze a.
- 4.12. Boston  
Nowy Jork
- 4.13. W strukturze warunkowej podwójnego wyboru istnieją dwie ścieżki, którymi może podążyć program. Pierwsza z nich dotyczy sytuacji, gdy warunek zostanie spełniony, a druga — gdy warunek nie zostanie spełniony.
- 4.14. If-Then-Else.
- 4.15. Kiedy warunek nie zostanie spełniony.
- 4.16.
- ```
if number == 1 Then
    Display "Jeden"
Else If number == 2 Then
    Display "Dwa"
Else If number == 3 Then
    Display "Trzy"
Else
    Display "Brak danych"
End If
```
- 4.17. Za pomocą struktury warunkowej wielokrotnego wyboru można sprawdzić wartość wyrażenia lub zmiennej i w zależności od tej wartości wykonać w programie określone instrukcje.
- 4.18. Za pomocą polecenia Select Case.
- 4.19. Zmienną lub wyrażenie.
- 4.20. W takim przypadku można wykorzystać polecenie If-Then-Else If lub zagnieżdzoną strukturę warunkową.
- 4.21. Złożone wyrażenie boolowskie to wyrażenie, które powstaje w wyniku zastosowania operatora logicznego względem dwóch podwyrażeń boolowskich.
- 4.22. F  
P  
F  
F  
P  
P  
F  
F  
P

4.23. P

F  
P  
P  
P

4.24. W przypadku operatora AND: jeżeli wyrażenie po lewej stronie operatora AND zwraca fałsz, wtedy wyrażenie po prawej stronie operatora AND nie zostanie w ogóle sprawdzone.

W przypadku operatora OR: jeżeli wyrażenie po lewej stronie operatora OR zwraca prawdę, wtedy wyrażenie po prawej stronie operatora OR nie zostanie w ogóle sprawdzone.

4.25. If speed >= 0 AND speed <= 200 Then  
Display "Liczba jest poprawna"  
End If

4.26. If speed < 0 OR speed > 200 Then  
Display "Liczba nie jest poprawna"  
End If

4.27. True lub False.

4.28. Flaga to zmienna, która informuje, czy spełniony jest określony warunek.

## Rozdział 5.

5.1. Zadaniem struktury cyklicznej jest wielokrotne wykonywanie wybranego fragmentu kodu.

5.2. W pętlach warunkowych do kontrolowania, ile razy wykonają się określone instrukcje, używa się warunku typu prawda – fałsz.

5.3. W pętlach licznikowych instrukcje wykonują się określoną liczbę razy.

5.4. Iteracja to każdorazowe uruchomienie instrukcji zawartych w ciele pętli.

5.5. Pętla, w której warunek sprawdzany jest na początku, w pewnych przypadkach może nie wykonać żadnej iteracji. Pętla, w której warunek sprawdzany jest na końcu, wykoną co najmniej jedną iterację.

5.6. Przed.

5.7. Po.

5.8. Pętli nieskończonej nie da się zatrzymać — będzie się wykonywała aż do momentu, gdy zamkniesz program.

5.9. Pętla Do-While wykonuje iteracje wtedy, gdy warunek jest spełniony. Kiedy warunek nie jest spełniony, pętla kończy działanie. Pętla Do-Until wykonuje iteracje aż do momentu, gdy warunek zostanie spełniony. Kiedy warunek zostanie spełniony, pętla kończy działanie.

5.10. Zmienna licznikowa służy do zliczania iteracji w pętli.

5.11. Inicjalizacja, sprawdzanie warunku i inkrementacja.

5.12. Inkrementacja oznacza zwiększenie, a dekrementacja — zmniejszenie wartości.

5.13. 6

5.14. 1

2  
3  
4  
5

5.15. 0

100  
200  
300  
400  
500

5.16. 1

2  
3  
4  
5  
6  
7  
8

5.17. 1

3  
5  
7

5.18. 5

4  
3  
2  
1

- 5.19. • z pętli, za pomocą której będą odczytywane kolejne wartości;  
• ze zmiennej, w której sumowane będą odczytane wartości.

5.20. Akumulator to zmienna, w której sumowane są wartości.

5.21. Tak, akumulator należy inicjalizować wartością 0. Dlatego, że wartości są dodawane do akumulatora w pętli. Jeśli w momencie rozpoczęcia pętli w akumulatorze będzie inna wartość niż 0, błędna będzie także suma wartości obliczona w pętli.

5.22. 15

5.23. 5

5.24. Wartownik to specjalna wartość, która wskazuje koniec listy elementów.

5.25. Wartość wartownika musi być na tyle unikatowa, aby nie mogła być pomyłona z innymi wartościami znajdującymi się na liście elementów.

## Rozdział 6.

6.1. Kiedy moduł zakończy działanie, program wraca do miejsca, w którym moduł został wywołany, i kontynuuje wykonywanie kolejnych instrukcji. Kiedy funkcja zakończy działanie, zwraca do miejsca w programie, w którym została wywołana, pewną wartość.

6.2. Funkcje biblioteczne to gotowe funkcje będące częścią danego języka programowania.

6.3. „Czarną skrzynką” nazywamy mechanizm, w którym przekazuje się pewne dane, następnie mechanizm ten wykonuje na tych danych jakieś operacje i zwraca dane

wyjściowe. Funkcję biblioteczną możemy traktować jak „czarną skrzynkę”, ponieważ nie znamy kodu, z którego składa się dana funkcja — przyjmuje ona po prostu dane wyjściowe, przetwarza je i zwraca dane wyjściowe.

- 6.4. Polecenie to przypisze do zmiennej `x` losową liczbę z przedziału 1 – 100.
- 6.5. Polecenie to wyświetli losową liczbę z przedziału 1 – 20.
- 6.6. Polecenie `Return` służy do wskazania wartości, jaką zwróci funkcja, gdy zakończy działanie. Wywołanie polecenia `Return` kończy działanie funkcji i zwraca wartość zapisaną w zmiennej `result` do tego miejsca w programie, w którym została ona wywołana.
- 6.7. a. `doSomething`  
b. `Integer`  
c. 10
- 6.8. Funkcja boolowska to funkcja, która zwraca wynik typu prawda lub fałsz.

## Rozdział 7.

- 7.1. Określenie „garbage in, garbage out” oznacza, że jeśli dostarczymy do programu błędne dane wejściowe, w wyniku otrzymamy błędne dane wyjściowe.
- 7.2. Walidacja danych wejściowych to proces polegający na przeanalizowaniu danych wejściowych. Jeśli dane wejściowe są błędne, program powinien je odrzucić i poprosić użytkownika o wprowadzenie prawidłowych danych.
- 7.3. Odczytujemy dane wejściowe, a następnie pojawia się pętla, w której warunek jest sprawdzany na początku. Jeśli dane są niepoprawne, wykonywane są instrukcje zawarte w ciele pętli — pętla wyświetla wtedy komunikat błędu, aby użytkownik wiedział, że dane są błędne, i program ponownie czeka na wprowadzenie poprawnych danych. Pętla będzie działała dopóty, dopóki dane wejściowe będą niepoprawne.
- 7.4. Odczyt wstępny polega na odczytaniu danych wejściowych jeszcze przed pętlą walidacji danych. Jego zadaniem jest pobranie od użytkownika początkowych danych, które będziemy walidować w pętli.
- 7.5. Żadnej.

## Rozdział 8.

- 8.1. Nie można. Wszystkie elementy tablicy muszą być tego samego typu.
- 8.2. Rozmiar tablicy określa liczbę elementów, które można w niej zapisać.
- 8.3. Nie.
- 8.4. Elementy to poszczególne komórki w tablicy.
- 8.5. Indeks to unikatowy identyfikator liczbowy przypisany do każdego elementu tablicy.
- 8.6. 0.

- 8.7. a. numbers  
b. 7  
c. Real  
d. 6
- 8.8. Większość języków programowania przeprowadza sprawdzanie zakresu indeksu, dzięki czemu odwołanie się w programie do nieistniejącego elementu tablicy jest niemożliwe.
- 8.9. Błąd off-by-one polega na tym, że pętla wykonuje o jedną iterację za dużo lub za mało.
- 8.10. Algorytmy wyszukiwania służą do odnajdywania określonej wartości w większym zbiorze danych — takim jak tablica.
- 8.11. Od pierwszego elementu tablicy.
- 8.12. Pętla w algorytmie przeszukiwania sekwencyjnego służy do pobierania wartości kolejnych elementów tablicy i porównywania ich z wartością szukaną. Gdy szukana wartość nie zostanie odnaleziona w tablicy, pętla zakończy działanie.
- 8.13. Algorytm sprawdzi w takim przypadku wszystkie elementy tablicy.
- 8.14. Możemy skorzystać z funkcji `contains`. Gdy dany ciąg znaków zawiera inny ciąg znaków, funkcja `contains` zwraca prawdę, a w przeciwnym przypadku zwraca fałsz.
- 8.15. Aby obliczyć sumę wartości wszystkich elementów tablicy, wykorzystamy w pętli zmienną pełniącą rolę akumulatora. Pętla pobiera po kolei elementy tablicy i dodaje je do akumulatora.
- 8.16. Pierwszym etapem obliczania średniej wartości elementów tablicy jest obliczenie sumy wszystkich wartości. Można w tym celu użyć algorytmu sumującego wszystkie elementy tablicy. Drugim etapem jest podzielenie obliczonej sumy przez liczbę elementów tablicy.
- 8.17. Tworzymy zmienną, w której zapiszemy wartość największego elementu (w przykładach zamieszczonych w tej książce nazwałem tę zmienną `highest`). Następnie do zmiennej tej przypisujemy wartość elementu tablicy o indeksie 0. W kolejnym kroku korzystamy z pętli, aby prześledzić pozostałe elementy tablicy, począwszy od elementu o indeksie 1. W każdej iteracji pętli porównujemy wartość elementu tablicy ze zmienną `highest`. Jeżeli element tablicy jest większy niż zmienna `highest`, wtedy przypisujemy do tej zmiennej wartość tego elementu. Kiedy pętla zakończy działanie, w zmiennej `highest` będzie się znajdowała wartość największego elementu tablicy.
- 8.18. Tworzymy zmienną, w której zapiszemy wartość najmniejszego elementu (w przykładach zamieszczonych w tej książce nazwałem tę zmienną `lowest`). Następnie do zmiennej tej przypisujemy wartość elementu tablicy o indeksie 0. W kolejnym kroku korzystamy z pętli, aby prześledzić pozostałe elementy tablicy, począwszy od elementu o indeksie 1. W każdej iteracji pętli porównujemy wartość elementu tablicy ze zmienną `lowest`. Jeżeli element tablicy jest mniejszy niż zmienna `lowest`, wtedy przypisujemy do tej zmiennej wartość tego elementu. Kiedy pętla zakończy działanie, w zmiennej `lowest` będzie się znajdowała wartość najmniejszego elementu tablicy.
- 8.19. Aby skopiować zawartość tablicy do innej tablicy, należy po kolej skopiować poszczególne elementy jednej tablicy do drugiej. Najłatwiej jest to zrobić przy użyciu pętli.

- 8.20. W tablicach równoległych elementy są ze sobą powiązane za pomocą tego samego indeksu.
- 8.21. W elemencie creditScore[82].
- 8.22. Z 88 wierszy i 100 kolumn.
- 8.23. Set points[87][99] = 100
- 8.24. Constant Integer ROWS = 3  
 Constant Integer COLS = 5  
 Declare Integer table[ROWS][COLS] = 12, 24, 32, 21, 42,  
                                 14, 67, 87, 65, 90,  
                                 19, 1, 24, 12, 8
- 8.25. Declare Integer row  
 Declare Integer col  
 For row = 0 To ROWS - 1  
     For col = 0 to COLS - 1  
         Set info[row][col] = 99  
     End For  
 End For
- 8.26. Constant Integer RACKS = 50  
 Constant Integer SHELVES = 10  
 Constant Integer BOOKS = 25  
 Declare String books[RACKS][SHELVES][BOOKS]

## Rozdział 9.

- 9.1. Algorytm sortowania bąbelkowego.
- 9.2. Algorytm sortowania przez wstawianie.
- 9.3. Algorytm sortowania przez wybieranie.
- 9.4. W algorytmie wyszukiwania sekwencyjnego sprawdzamy za pomocą pętli po kolejnym elementy tablicy, porównując go z wartością szukaną. Algorytm wyszukiwania binarnego wymaga, aby elementy tablicy były uporządkowane w kolejności rosnącej. Sprawdza on najpierw środkowy element tablicy. Jeżeli wartość środkowego elementu jest większa od szukanej wartości, wtedy algorytm sprawdza środkowy element pierwszej połówki tablicy. Jeśli wartość środkowego elementu tablicy jest mniejsza od szukanej wartości, wówczas algorytm sprawdza środkowy element drugiej połówki tablicy. Po każdym porównaniu wartości środkowego elementu obszar poszukiwań zauważany jest do połówki pozostały do przeszukania obszaru tablicy. Proces ten powtarza się do momentu, gdy wartość zostanie odnaleziona lub zabraknie elementu do porównania. Algorytm wyszukiwania binarnego jest wydajniejszy od algorytmu wyszukiwania sekwencyjnego.
- 9.5. 500.
- 9.6. 10.

## Rozdział 10.

- 10.1. Na dysku komputera.
- 10.2. Plik wyjściowy to plik, w którym zapisujemy dane. Nazwa ta wynika stąd, że program zapisuje w pliku dane wyjściowe.
- 10.3. Plik wejściowy to plik, z którego odczytujemy dane. Nazwa ta wynika stąd, że program pobiera z pliku dane wejściowe.
- 10.4.1. Otwarcie pliku.
  2. Przetworzenie pliku.
  3. Zamknięcie pliku.
- 10.5. Pliki tekstowe i binarne. Plik tekstowy zawiera dane tekstowe zapisane za pomocą kodowania takiego jak Unicode. Nawet jeśli plik będzie zawierał liczby, zostaną one zapisane w pliku jako zbiór znaków. Dzięki temu plik taki można otworzyć i podejrzeć w edytorze tekstowym, takim jak na przykład Notatnik. Plik binarny zawiera dane, które nie zostały zapisane za pomocą kodowania znaków. Dlatego pliku binarnego nie da się podejrzeć w edytorze tekstowym.
- 10.6. Dostęp sekwencyjny i dostęp swobodny. Gdy korzystamy z dostępu sekwencyjnego, dane będącymi musieli odczytywać po kolej — od początku pliku do jego końca. W przypadku dostępu swobodnego możemy przeskoczyć do konkretnego miejsca w pliku, bez potrzeby odczytywania danych zapisanych wcześniej.
- 10.7. Nazwę pliku i nazwę obiektu reprezentującego plik. Nazwa pliku to nazwa, która wskazuje plik zapisany na dysku komputera. Nazwa obiektu reprezentującego plik jest podobna do nazwy zmiennej — za jej pomocą odwołujemy się do danego pliku w kodzie programu.
- 10.8. Zawartość pliku zostanie wykasowana.
- 10.9. Po otwarciu pliku utworzona zostanie łączność między programem a plikiem i będzie można się do niego odwoływać za pomocą obiektu reprezentującego plik.
- 10.10. Zamknięcie pliku odłącza plik od programu.
- 10.11. Separator to predefiniowany znak lub ciąg znaków, które wskazują koniec pewnego fragmentu danych. W wielu językach programowania separator jest wstawiany po każdym elemencie zapisanym w pliku.
- 10.12. Jest to specjalny znak lub ciąg znaków zwany znacznikiem end-of-file.
- 10.13. Wskaźnik ten wskazuje miejsce w pliku, od którego rozpocznie się kolejny odczyt danych. Zaraz po otwarciu pliku wewnętrzny wskaźnik pliku jest ustawiany na samym początku pliku.
- 10.14. Plik należy otworzyć w trybie dołączania. Podczas zapisywania danych w trybie dołączania nowe dane zostaną zapisane na samym końcu pliku.
- 10.15. 

```
Declare Integer counter
Declare OutputFile myFile
Open myFile "myfile.dat"
For counter = 1 To 10
    Write myFile, counter
End For
Close myFile
```
- 10.16. Funkcja eof informuje o tym, w którym miejscu kończą się dane zapisane w pliku.

- 10.17. Nie można. Operacja taka spowoduje wystąpienie błędu.
- 10.18. Oznacza to, że program odczytał całą zawartość pliku, do którego odwołuje się obiekt `myFile`.
- 10.19. b.
- 10.20. Rekord to kompletny zbiór danych opisujących konkretny element, a pole to jedna z właściwości tego elementu.
- 10.21. Kopiujemy do pliku tymczasowego całą zawartość pliku pierwotnego, ale gdy dojdziemy do rekordu modyfikowanego, to zamiast jego pierwotnej wersji zapisujemy wersję zmodyfikowaną. Następnie kopujemy pozostałe rekordy zawarte w pliku pierwotnym.
- 10.22. Kopiujemy do pliku tymczasowego wszystkie rekordy z pliku pierwotnego, z wyjątkiem rekordu usuwanego. Następnie musimy zastąpić plik pierwotny plikiem tymczasowym — usuwamy więc plik pierwotny i zmieniamy nazwę pliku tymczasowego na taką, jaką miał plik pierwotny.

## Rozdział 11.

- 11.1. Program sterowany za pomocą menu wyświetla na ekranie listę operacji, jakie może wykonać, i umożliwia użytkownikowi wybranie jednej z nich. Listę operacji wyświetlonych na ekranie nazywamy menu.
- 11.2. Dzięki temu użytkownik może wybrać interesującą go operację, wprowadzając na klawiaturze odpowiednią liczbę lub znak.
- 11.3. W takiej sytuacji korzystamy ze struktury warunkowej — np. zagnieżdzonego polecenia `If-Then-Else` lub polecenia `If-Then-Else If`. Może to być także struktura decyzyjna.
- 11.4. Jeśli nie użyjemy pętli, program zakończy działanie po wykonaniu wybranej przez użytkownika operacji. Takie zachowanie programu jest dla użytkownika niewygodne, ponieważ aby wykonać inną operację, musiałby ponownie uruchomić program.
- 11.5. Operacja zakończenia programu powinna się znajdować w menu. Gdy użytkownik wybierze tę operację, program wyjdzie z pętli i tym samym zakończy się działanie programu.
- 11.6. W przypadku menu jednopoziomowego wszystkie operacje możemy umieścić w pojedynczym menu. Gdy użytkownik wybierze jedną z operacji w menu, program ją natychmiast wykona, a następnie ponownie wyświetli menu.
- 11.7. W przypadku menu wielopoziomowego program zaraz po uruchomieniu wyświetla menu główne, na którym są widoczne tylko wybrane elementy, a gdy użytkownik wybierze którąś z pozycji, program wyświetla mniejsze podmenu.
- 11.8. Ponieważ użytkownicy takiego programu mieliby kłopot z przeglądaniem menu zawierającego tak dużą liczbę operacji.

## Rozdział 12.

- 12.1. n
- 12.2. Set str[0] = "B"
- 12.3. If isDigit(str[0]) Then  
    delete(str, 0, 0)  
End If
- 12.4. If isUpperCase(str[0]) Then  
    Set str[0] = "0"  
End If
- 12.5. insert(str, 0, "Witaj, ")
- 12.6. delete(city, 0, 2)

## Rozdział 13.

- 13.1. Algorytm rekurencyjny polega na wielokrotnym wywoływaniu modułu. Podczas każdego wywołania komputer musi wykonać kilka dodatkowych operacji: zaalokować pamięć na zmienne parametry i zmienne lokalne i zapisać adres, do którego ma wrócić program po wykonaniu modułu. Operacje te nazywamy narzutem. W algorytmie iteracyjnym z wykorzystaniem pętli narzut taki nie występuje.
- 13.2. Jest to przypadek, w którym dane zadanie da się rozwiązać bez użycia rekurencji.
- 13.3. Jest to przypadek, w którym dane zadanie da się rozwiązać, korzystając z rekurencji.
- 13.4. Moduł przestaje wywoływać sam siebie w momencie, gdy dojdzie do przypadku bazowego.
- 13.5. W rekurencji bezpośredniej moduł wywołuje sam siebie. W przypadku rekurencji pośredniej moduł A wywołuje moduł B, który z kolei wywołuje moduł A.

## Rozdział 14.

- 14.1. Obiekt to taki byt, w którym umieszczone są zarówno dane, jak i procedury.
- 14.2. Hermetyzacja polega na ukrywaniu danych przed kodem znajdującym się na zewnątrz obiektu.
- 14.3. Kiedy wewnętrzne dane obiektu są niewidoczne na zewnątrz niego, dzięki czemu są chronione przed przypadkowym uszkodzeniem. Ponadto kod umieszczony na zewnątrz obiektu nie musi znać formatu i struktury danych zapisanych w obiekcie.
- 14.4. Metody publiczne to metody, które można wywołać na zewnątrz obiektu. Metod prywatnych nie da się wywołać na zewnątrz obiektu — można ich używać tylko wewnątrz niego.
- 14.5. Plan reprezentuje klasę.
- 14.6. Wykrawacz to klasa, a ciastka to obiekty.
- 14.7. Modyfikator dostępu wskazuje, czy kod umieszczony na zewnątrz obiektu będzie miał dostęp do składowych klasy.

- 14.8. **Private.**
- 14.9. Adres obiektu w pamięci, do którego się ona odnosi.
- 14.10. Słowo kluczowe **New** tworzy obiekt w pamięci komputera.
- 14.11. Akcesor to metoda, która służy do pobierania wartości danego pola. Mutator to metoda, która służy do ustawiania wartości danego pola.
- 14.12. Konstruktor to metoda, za pomocą której ustawiamy wartości początkowe pól.  
Jest on uruchamiany automatycznie w momencie tworzenia obiektu.
- 14.13. Jeśli w klasie nie zdefiniujemy konstruktora, w większości języków programowania zostanie on utworzony automatycznie podczas kompilowania programu.  
Wygenerowany w ten sposób konstruktor nazywamy konstruktorem domyślnym.  
Konstruktor domyślny przypisuje zazwyczaj wartości początkowe do pól obiektu.
- 14.14. W górnej części znajduje się nazwa klasy. W środkowej części znajduje się lista pól klasy, a w dolnej części — lista metod klasy.
- 14.15. Wstawiając za nazwą pola dwukropki i słowo **String**. W taki sposób: **description : String**.
- 14.16. Metody prywatne oznaczamy znakiem **-**, a metody publiczne znakiem **+**.
- 14.17. Model dziedziny to zbiór rzeczywistych obiektów, podmiotów i głównych zdarzeń związanych z danym zadaniem.
- 14.18. Technika ta polega na sporządzeniu listy rzeczowników, zaimków i wyrażeń występujących w opisie modelu dziedziny. Następnie zawężamy listę słów przez usunięcie z niej powtórzeń, elementów, które nie są potrzebne do rozwiązania zadania, elementów, które reprezentują obiekty, a nie klasy, oraz elementów, które można zapisać w zwykłych zmiennych.
- 14.19. Zakres obowiązków klasy to rzeczy, które klasa musi pamiętać, i operacje, które musi wykonywać.
- 14.20. Kiedy wartość jednego z pól jest zależna od innych danych i w momencie, gdy dane te ulegną zmianie, nie zaktualizujemy wartości pola, to zapisane w nim dane stana się nieaktualne.
- 14.21. Klasa pochodna jest szczególnym przypadkiem klasy bazowej.
- 14.22. Kiedy między dwoma obiektami zachodzi relacja typu „jest”, oznacza to, że jeden z obiektów ma wszystkie cechy drugiego oraz kilka własnych cech, które czynią go szczególnym przypadkiem drugiego obiektu.
- 14.23. Klasa pochodna dziedziczy wszystkie składowe z wyjątkiem prywatnych.
- 14.24. **Bird** jest klasą pochodną, a **Canary** jest klasą bazową.
- 14.25. Jestem ziemniakiem.  
Jestem ziemniakiem.

## Rozdział 15.

- 15.1. Interfejs użytkownika służy do komunikowania się użytkownika z komputerem.
- 15.2. Interfejs wiersza poleceń wyświetla zazwyczaj znak zachęty, po nim użytkownik może wprowadzić polecenie, które następnie wykona komputer.
- 15.3. Program.
- 15.4. Program sterowany zdarzeniami to program, który reaguje na zdarzenia generowane na przykład w momencie kliknięcia przycisku.
- 15.5. Diagram interfejsu użytkownika ilustruje, jak program w wyniku działań użytkownika przechodzi z jednego okna do innego.
- 15.6. Ponieważ programista musiał napisać kod odpowiedzialny za tworzenie okien oraz elementów graficznych takich jak ikonki i przyciski, a także ustawać ich kolor, położenie, rozmiar i wiele innych właściwości. Nawet najprostszy program wyposażony w GUI, który wyświetla jedynie napis *Witaj, świecie!*, wymagał od programisty napisania setek linii kodu, albo i więcej. Ponadto programista nie widział tak naprawdę, jak prezentuje się interfejs użytkownika, dopóki nie skompilował programu i go nie uruchomił.
- 15.7. Przeciągając je z przybornika do okna edytora.
- 15.8. Jest to element będący częścią graficznego interfejsu użytkownika.
- 15.9. Za pomocą nazwy komponentu możemy się do niego odwoływać w kodzie programu, podobnie jak ma to miejsce w przypadku nazwy zmiennej.
- 15.10. Właściwości komponentu wpływają na jego wygląd na ekranie.
- 15.11. Zdarzenie ma miejsce w programie na przykład wtedy, gdy użytkownik kliknie przycisk.
- 15.12. Jest to moduł, który jest wywoływany w momencie wystąpienia określonego zdarzenia.
- 15.13. a. Moduł reaguje na zdarzenie `Click`.  
b. Komponent nazywa się `showValuesButton`.
- 15.14. Jest wykonywany automatycznie po uruchomieniu aplikacji GUI (lub aplikacji mobilnej).
- 15.15. Mniej.
- 15.16. Większy.
- 15.17. Oto pięć unikalnych cech, jakimi zwykle wyróżniają się urządzenia mobilne w porównaniu z komputerami stacjonarnymi:
  - wykonywanie i odbieranie połączeń telefonicznych;
  - wysyłanie i odbieranie wiadomości SMS;
  - określanie lokalizacji urządzenia;
  - wykrywanie fizycznej orientacji urządzenia w przestrzeni;
  - określanie, kiedy urządzenie się porusza.
- 15.18 Komponenty niewizualne to obiekty, które są niewidoczne w GUI, ale pozwalają programom wykonywać specjalne działania. Przykłady, które omówiliśmy, to komponenty `PhoneCall`, `TextMessage` i `Location`.
- 15.19. Generowane jest zdarzenie `IncomingCall`.
- 15.20. Generowane jest zdarzenie `IncomingText`.
- 15.21. Generowane jest zdarzenie `LocationChanged`.

# Skorowidz

## A

akcesor, 695  
akumulator, 289  
algorytm, 53  
przeszukiwania sekwencyjnego, 400  
rekurencyjny, 664, 671  
sortowania, 453  
wyszukiwania binarnego, 479  
wyszukiwania, 400, 407, 409  
argument, 110, 148  
arkusz kalkulacyjny, 504  
ASCII, 33, 779  
asembler, 38, 45

## B

bajt, 30  
binarny system liczbowy, 31  
bit, 30  
blok, 170  
    graniczny, 56, 781  
    operacyjny, 56, 781  
    warunkowy, 781  
wejścia/wyjścia, 56, 781  
wywołania modułu, 781  
błąd  
    logiczny, 52  
    niezgodności typów, 342  
    off-by-one, 396  
    składniowy, 42  
bufor, 508

## C

camelCase, 65  
chmura, 28  
ciało  
    funkcji, 324  
    metody, 167  
    pętli, 254  
ciasteczko, 504  
ciąg  
    Fibonacciego, 665  
    znaków, 62  
CPU, 25  
cykl rozkazowy, 37

## D

dane  
    cyfrowe, 34  
    ukrywanie, 684  
    wejściowe, 29, 51, 100, 330  
    wyjściowe, 29, 51, 101, 330  
debugowanie, 52  
deklarowanie, 85  
dekrementowanie, 281  
diagram interfejsu użytkownika, 749  
dokumentacja  
    wewnętrzna, 94  
    zewnętrzna, 93  
dostęp  
    sekwencyjny, 506  
    swobodny, 506  
dysk  
    SSD, 28  
    twardy, 28  
dziedziczenie, 711  
dzielenie całkowite, 90

## E

edytor graficzny, 504  
element, 437  
ENIAC, 26  
EOF, 510

## F

flaga, 229  
funkcja, 133, 315  
    Ackermann, 681  
    append, 345, 628  
    biblioteczna, 316  
    contains, 348, 628  
    definiowanie, 791  
    discount, 330  
    eof, 792  
    float, 112  
    formatująca, 344  
    int, 112  
    isDigit, 631  
    isInteger, 349, 628  
    isLetter, 631

funkcja  
 isLower, 631  
 isReal, 349, 628  
 isUpper, 631  
 isWhiteSpace, 631  
 konwertowania typów danych, 342  
 length, 345, 628  
 logiczna, 354  
 matematyczna, 339

abs, 341  
 cos, 341  
 pow, 340  
 round, 341  
 sin, 341  
 sqrt, 339  
 tan, 341  
 random, 317  
 return, 327  
 sprawdzająca znaki, 631  
 stringToInteger, 348, 628  
 stringToReal, 348, 628  
 substring, 347, 628  
 ToInteger, 342  
 toLower, 346, 628  
 toReal, 342  
 toUpper, 346, 628  
 walidacyjna, 370

**G**

garbage in, 365  
 garbage out, 365  
 głębokość rekurencji, 659  
 GUI, 578, 748, 751

**H**

hermetyzacja, 684

**I**

IDE, 43, 751  
 inicjalizacja, 88, 272  
 inkrementacja, 272, 278  
 instancja, 687  
 instrukcja  
     Display, 66  
     If-Then, 786  
     If-Then-Else, 786  
     przypisania, 69  
     Return, 791  
     Select Case, 787

interfejs  
 graficzny interfejs użytkownika, 748  
 użytkownika, 747  
 wiersza poleceń, 747  
 interpreter, 41, 45  
 IPO, 59, 330  
 iteracja, 256

**J**

język programowania  
 Ada, 40  
 Asembler, 38  
 BASIC, 40  
 C#, 40  
 C, 40  
 C++, 40, 114, 175, 238, 304, 354, 378, 441,  
     493, 564, 619, 647, 677, 734, 773  
 COBOL, 39  
 FORTRAN, 40  
 Java, 40, 101, 167, 230, 299, 350, 376, 434,  
     485, 550, 616, 642, 674, 723, 773  
 JavaScript, 40  
 maszynowy, 36  
 niskiego poziomu, 39  
 Pascal, 40  
 Python, 40, 110, 170, 235, 302, 352, 377, 437,  
     489, 554, 618, 645, 675, 729, 773  
 Ruby, 40  
 UML, 698  
 Visual Basic, 40  
 wysokiego poziomu, 39

**K**

klasa, 687  
 bazowa, 712  
 definiowanie, 793  
 pochodna, 712  
 składowa, 689  
 wyznaczanie, 701  
 kod źródłowy, 41  
 komentarz, 94  
     blokowy, 94  
     liniowy, 94  
 kompilator, 41  
 komponent, 751  
     Button, 794  
     etykieta, 753  
     Label, 794  
     lista rozwijana, 753

Location, 795  
 PhoneCall, 795  
 pole listy, 753  
 pole tekstowe, 753  
 przycisk, 753  
     opcji, 753  
     wyboru, 753  
 suwak, 753  
 TextBox, 794  
 TextMessage, 795  
 komputer, 25  
 komputer budowa, 25  
     centralna jednostka obliczeniowa, 25  
     nośnik danych, 25, 28  
     pamięć główna, 25  
     urządzenie  
         wejściowe, 25  
         wyjściowe, 25  
 komunikat, 68  
 konkatenacja, 346  
 konserwacja oprogramowania, 133  
 konstruktor, 695  
 konstruktor domyślny, 697  
 kontrolka, 753  
 krok, 278

**L**

licznik, 272  
 lista, 437  
     inicjalizacyjna, 395  
     parametrów, 152  
     rozkazów procesora, 36  
 literal  
     ciągu znaków, 62  
     liczbowy, 89

**L**

łącznik  
     wewnętrzny, 781  
     zewnętrzny, 781

**M**

menu, 577  
     główne, 610  
     jednopoziomowe, 610  
     podmenu, 610  
     wielopoziomowe, 610  
 metoda, 102, 133, 167, 684  
     getAddress(), 795  
     getLatitude(), 795

getLongitude(), 795  
 inicjalizująca, 731  
 instancji, 724  
 isalnum(), 646, 648  
 isalpha(), 646, 648  
 isdigit(), 646, 648  
 islower(), 646, 648  
 isspace(), 646, 648  
 isupper(), 646, 648  
 prywatna, 686  
 publiczna, 686  
 mikroprocesor, 26  
 mnemonika, 38  
 model dziedziny, 701  
 modularyzacja  
     kodu, 330  
     pętla, 260  
     programu, 587  
 modularyzowanie  
     C++, 175  
     Java, 167  
     Python, 170  
 moduł, 131  
     ciało, 135  
     definicja, 135  
     definiowanie, 790  
     delete, 636  
     główny, 136  
     init, 761, 794  
     insert, 636  
     nagłówek, 135  
     rekurencyjny, 657  
     sterowanie, 138  
     swap, 458  
     wywoływanie, 136, 790  
 modyfikator dostępu, 690  
 mutator, 695

**N**

nagłówek  
     funkcji, 170, 324  
     klasy, 101  
     metody, 102, 167  
 napęd USB, 28  
 narzędzie do tworzenia oprogramowania, 45  
 narzut, 660  
 nośnik optyczny, 28

**O**

obiekt, 684  
 obsługa błędu, 368  
 odczyt pustych danych, 374  
 odczyt wstępny, 367  
 okno dialogowe, 748  
 operanda, 72  
 operator, 39  
     arytmetyczny, 108, 113, 121  
     konkatenacji, 108  
     kropka, 694  
     logiczny, 221  
         AND, 221, 222  
         NOT, 221  
         OR, 221, 222  
     matematyczny, 72  
         \*, 72  
         /, 72  
         ^, 72  
         +, 72  
         MOD, 72  
         modulo, 81  
         potęgowanie, 81  
 relacji, 190  
     !=, 191  
     <, 191  
     <=, 191  
     ==, 191  
     >, 191  
     >=, 191  
     wstawiania do strumienia, 115  
 oprogramowanie, 23  
     systemowe, 44  
      użytkowe, 45

**P**

pamięć  
     flash, 28  
     główna, 26  
     nieulotna, 27  
     o dostępie swobodnym, 26  
     tylko do odczytu, 27  
     ulotna, 26  
 parametr, 148, 790  
 pendrive, 28  
 pętla, 252, 516, 591  
     For Each, 399, 789  
     licznikowa, 252, 271  
     for, 273, 789  
     nieskończona, 260  
     walidacji danych wejściowych, 368

warunkowa, 252  
 Do-Until, 268, 788  
 Do-While, 262  
 While, 253, 788  
 zagnieżdziona, 295  
 piksel, 34  
 plik  
     binarny, 506  
     rozszerzenie, 506  
     specyfikacja, 529  
     tekstowy, 505  
     wejściowy, 504  
     wewnętrzny wskaźnik, 513  
     wyjściowy, 504  
 pobieranie danych wejściowych, 98  
 procedura, 133  
 podprogram, 133  
 pole, 525  
     instancji, 724  
     klasy, 169  
 polecenie, 41  
     Close, 792  
     Constant, 783  
     Declare, 783  
     Delete, 793  
     Display, 784  
     Input, 785  
     Open, 792  
     Rename, 793  
     Write, 791  
 polimorfizm, 718  
 procedura, 133, 683  
 procesor tekstowy, 503  
 program  
     narzędziowy, 45  
     sterowany zdarzeniami, 587, 749  
 programowanie  
     defensywne, 374  
     obiektowe, 684  
 prototyp funkcji, 176  
 próbka, 34  
 przekazywanie argumentu przez  
     referencję, 157, 159  
     wartość, 156  
 przetwarzanie, 100, 330  
     danych wejściowych, 98  
     znaków, 629  
 przecięcie, 90  
 przypadek  
     bazowy, 660  
     rekurencyjny, 661  
 pseudokod, 54

**R**

refaktoryzacja, 160  
rekord, 525  
  dodawanie, 530  
  modyfikowanie, 535  
  odczytywanie, 528  
  usuwanie, 540  
  wyszukiwanie, 533  
  zapisywanie, 526  
zarządzanie, 530  
rekurencja, 673  
  bezpośrednia, 664  
  pośrednia, 664

**S**

schemat  
  blokowy, 55  
  hierarchiczny, 140  
  rozmieszczenia znaków, 549  
  strukturalny, 140  
separator, 510  
separator sterowania, 543  
short-circuit evaluation, 223  
składnia, 40  
słowo  
  klucz, 39  
  zarezerwowane, 39  
sortowanie  
  bąbelkowe, 454  
  przez wstawianie, 454, 473  
  przez wybieranie, 454, 468  
  tablicy, 464  
  w kolejności malejącej, 466  
sprawdzanie warunku, 272  
sprzęt, 24  
stała  
  globalna, 163  
  nazwana, 91  
struktura  
  cykliczna, 252  
  decyzyjna, 215, 781  
  If-Then-Else If, 213  
    podwójnego wyboru, 198  
    pojedynczego wyboru, 189  
  sekwencyjna, 62, 187  
  sterująca, 62, 187  
  warunkowa, 188, 208, 213, 578  
  wielokrotnego wyboru, 215  
suma bieżąca, 289

**symbol**

łączników wewnętrznych, 56  
łączników zewnętrznych, 56  
system operacyjny, 44

**S**

śledzenie ręczne, 92

**T**

tablica, 386  
  dwuwymiarowa, 424  
element, 387  
indeks, 387  
jednowymiarowa, 424  
kopiowanie, 410  
przekazywanie, 411  
przetwarzanie elementów, 405  
rozmiar, 386  
równoległa, 420  
sumowanie wartości elementów, 405  
trójwymiarowa, 432  
uśrednianie wartości elementów, 407  
wyszukiwanie elementu o najmniejszej  
  wartości, 409  
wyszukiwanie elementu o największej  
  wartości, 407  
technika  
  uściślanie stopniowe, 139  
  uzupełnienia do 2, 34  
tryb  
  dołączania, 514  
  pliku, 507  
  postfiksowy, 299, 305  
  prefiksowy, 299, 305  
typ danych, 85, 105, 117  
  bool, 117  
  byte, 105  
  char, 117  
  character, 783  
  double, 105, 117  
  float, 105, 117  
  int, 105, 117  
  Integer, 783  
  long, 105, 117  
  real, 783  
  short, 105, 117  
  string, 105, 117, 783

**U**

Unicode, 33, 779  
 urządzenie  
   cyfrowe, 34  
   wejściowe, 29  
   wyjściowe, 29  
 użytkownik końcowy, 64

**W**

validacja danych, 580  
 walidacja danych wejściowych, 366  
 wartość kroku, 304  
 wartownik, 292  
 widżet, 753  
 wielokrotne wykorzystanie kodu, 133  
 właściwości, 754  
 wykonanie warunkowe, 189  
 wyrażenie  
   boolowskie, 190, 227  
   inicjalizujące, 301, 307  
   inkrementujące, 302, 307  
   testujące, 302, 307  
 $x!=y$ , 192  
 $x<=y$ , 192  
 $x<y$ , 192  
 $x==y$ , 192  
 $x>=y$ , 192  
 $x>y$ , 192

**Z**

zagłędzanie, 208  
 zapis zmiennoprzecinkowy, 34  
 zdarzenie, 758  
   Click, 796  
   Incoming\_Text, 796  
   IncomingCall, 796  
   LocationChanged, 796  
 zmienna, 63  
   boolowska, 229  
   globalna, 162  
   integer, 229  
   licznikowa, 272, 276  
   lokalna, 145, 171  
   niezainicjalizowana, 88  
   obiektowa, 692  
   real, 229  
   referencyjna, 177  
   sterująca, 544  
   string, 229, 631  
   typu referencyjnego, 157  
   zasięg widoczności, 146  
   zdublowanie nazwy, 146  
 znacznik end-of-file, 510  
 znak  
   modyfikacji, 556  
   wiersza, 556  
 zwracanie danych wyjściowych, 99

# PROGRAM PARTNERSKI

— GRUPY 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  
**Helion**

## KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE  
ZGODNIE Z METODĄ

## BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

**WWW.HELIONSZKOLENIA.PL**