

Podręcznik jest zgodny z podstawą programową kształcenia w zawodzie technik informatyk 312 [01]

Wydanie II

Podręcznik do nauki zawodu

# TECHNIK INFORMATYK

PROGRAMOWANIE  
STRUKTURALNE  
I OBIEKTOWE

Zawiera CD



 **Helion**  
EDUKACJA

Tomasz Rudny

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

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

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

Redaktor prowadzący: Marcin Borecki  
Projekt okładki: Maciej Pasek

Fotografia na okładce została wykorzystana za zgodą Shutterstock.

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

Drogi Czytelniku!  
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres  
<http://helion.pl/user/opinie?prstk2>  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-246-3385-2

Copyright © Helion 2012

Printed in Poland.

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

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

# Spis treści

<b>Wstęp</b> .....	13
--------------------	----

## **Część I Wstęp do programowania**

<b>Rozdział 1.</b> Co to jest algorytm? .....	17
<b>1.1.</b> Wstęp .....	17
<b>1.2.</b> Definicja algorytmu .....	18
<b>1.3.</b> Algorytmy w szkole i w życiu .....	19
<b>1.4.</b> Algorytmy a programy komputerowe .....	22
<b>1.5.</b> Zapis algorytmów .....	22
<b>Rozdział 2.</b> Przykłady algorytmów .....	28
<b>2.1.</b> Sortowanie liczb .....	28
<b>2.2.</b> Wyszukiwanie .....	31
<b>2.3.</b> Schemat Hornera .....	32
<b>2.4.</b> Znajdowanie miejsc zerowych funkcji .....	34
<b>Rozdział 3.</b> Podstawowe pojęcia .....	36
<b>3.1.</b> Jak komputery wykonują obliczenia? .....	36
<b>3.2.</b> Język programowania .....	39
<b>3.3.</b> Kompilacja i konsolidacja .....	40
<b>3.4.</b> Biblioteki .....	42
<b>Rozdział 4.</b> Narzędzia programistyczne .....	44
<b>4.1.</b> Edytor .....	44
<b>4.2.</b> Debugger .....	46
<b>4.3.</b> Zintegrowane środowisko programistyczne (IDE) .....	47

## **Część II Programowanie strukturalne w Pascalu**

<b>Rozdział 5.</b> Środowisko języka Pascal .....	51
<b>5.1.</b> Turbo Pascal .....	52
<b>5.2.</b> Dev-Pascal i Hugo — alternatywne IDE dla systemu Windows .....	53
<b>5.3.</b> Free Pascal Compiler .....	55
<b>5.4.</b> Programowanie w Pascalu w systemie Linux .....	56

<b>Rozdział 6.</b> Podstawy programowania w Pascalu .....	60
<b>6.1.</b> Najprostszy program w Pascalu .....	60
<b>6.2.</b> Struktura programu w Pascalu .....	62
<b>6.3.</b> Słowa kluczowe języka .....	64
<b>6.4.</b> Komunikaty o błędach .....	65
<b>Rozdział 7.</b> Typy danych i zmienne .....	68
<b>7.1.</b> Pojęcie typu danych .....	68
<b>7.2.</b> Podstawowe typy danych .....	69
<b>7.3.</b> Zmienne .....	73
<b>7.4.</b> Deklaracje zmiennych i przypisanie wartości .....	74
<b>Rozdział 8.</b> Operatory i wyrażenia .....	77
<b>8.1.</b> Operatory arytmetyczne .....	77
<b>8.2.</b> Operatory porównania i operatory logiczne .....	80
<b>Rozdział 9.</b> Instrukcje warunkowe i iteracyjne .....	84
<b>9.1.</b> Podstawowa instrukcja warunkowa .....	84
<b>9.2.</b> Instrukcja wyboru .....	89
<b>9.3.</b> Instrukcja iteracyjna for .....	91
<b>9.4.</b> Inne instrukcje iteracyjne .....	94
<b>Rozdział 10.</b> Procedury i funkcje .....	97
<b>10.1.</b> Procedury i funkcje standardowe .....	98
<b>10.2.</b> Procedury i funkcje .....	100
<b>10.2.1.</b> Definicja procedury .....	101
<b>10.2.2.</b> Definicja funkcji .....	104
<b>10.3.</b> Przekazywanie parametrów i odbieranie wyników .....	106
<b>10.4.</b> Wzorcowa struktura programu .....	108
<b>Rozdział 11.</b> Tablice .....	112
<b>11.1.</b> Definicja tablicy w Pascalu .....	112
<b>11.2.</b> Wykorzystanie tablic w programach .....	113
<b>11.3.</b> Tablice wielowymiarowe .....	118
<b>Rozdział 12.</b> Rekurencja .....	120
<b>12.1.</b> Co to jest rekurencja? .....	120
<b>12.2.</b> Kiedy korzystać z rekurencji? .....	123
<b>12.3.</b> Wady rekurencji .....	127

<b>Rozdział 13.</b> Typy strukturalne .....	130
<b>13.1.</b> Definiowanie nowych typów danych w Pascalu .....	130
<b>13.2.</b> Rekordy .....	131
<b>13.3.</b> Tablice jako parametry podprogramów .....	133
<b>Rozdział 14.</b> Operacje na plikach .....	135
<b>14.1.</b> Dostęp do plików .....	135
<b>14.2.</b> Swobodny dostęp do pliku .....	140
<b>Rozdział 15.</b> Elementy zaawansowanego programowania w Pascalu .....	144
<b>15.1.</b> Dyrektywy kompilatora .....	144
<b>15.2.</b> Wbudowany asembler .....	147
<b>15.3.</b> Optymalizacja programów .....	148
<b>15.4.</b> Grafika BGI w Pascalu .....	149
<b>15.5.</b> Dynamiczny przydział pamięci .....	152
<b>Rozdział 16.</b> Przykład podsumowujący: baza danych .....	157
<b>16.1.</b> Omówienie programu .....	157
<b>16.2.</b> Kod programu .....	158
 <b>Część III Programowanie obiektowe w C++</b>	
<b>Rozdział 17.</b> Środowisko języka C++ .....	167
<b>17.1.</b> Turbo C++ .....	167
<b>17.2.</b> Dev C++ .....	168
<b>17.3.</b> Microsoft Visual Studio .....	169
<b>17.4.</b> Tworzenie programów w C++ dla systemu Linux .....	171
<b>Rozdział 18.</b> Składnia języka C++ .....	173
<b>18.1.</b> Słowa kluczowe .....	173
<b>18.2.</b> Typy danych .....	174
<b>18.3.</b> Operatory i wyrażenia .....	175
<b>18.4.</b> Instrukcje .....	178
<b>18.5.</b> Funkcje .....	181
<b>18.6.</b> Struktura programu .....	182
<b>Rozdział 19.</b> Podobieństwa i różnice pomiędzy Pascalem i C++ .....	186
<b>19.1.</b> Struktura programu .....	186

<b>19.2.</b> Specyfika instrukcji warunkowych . . . . .	187
<b>19.3.</b> Pułapki pętli w C++ . . . . .	190
<b>19.4.</b> Znaki specjalne . . . . .	193
<b>Rozdział 20.</b> Tablice i wskaźniki . . . . .	195
<b>20.1.</b> Tablice w języku C++ . . . . .	195
<b>20.2.</b> Tablice wielowymiarowe . . . . .	200
<b>20.3.</b> Tablice znaków (char) jako typ napisowy w C++ . . . . .	202
<b>20.4.</b> Wskaźniki w C++ . . . . .	204
<b>20.5.</b> Równoważność pomiędzy wskaźnikiem a tablicą . . . . .	207
<b>Rozdział 21.</b> Struktury i unie . . . . .	210
<b>21.1.</b> Struktury . . . . .	210
<b>21.2.</b> Unie . . . . .	213
<b>21.3.</b> Funkcje wewnętrzne struktur . . . . .	214
<b>Rozdział 22.</b> Operacje wejścia-wyjścia w C++ . . . . .	217
<b>22.1.</b> Strumienie wejścia-wyjścia . . . . .	217
<b>22.2.</b> Funkcje wejścia-wyjścia w stylu C . . . . .	220
<b>22.3.</b> Funkcje operujące na plikach . . . . .	223
<b>Rozdział 23.</b> Dynamiczne struktury danych . . . . .	227
<b>23.1.</b> Tablice dynamiczne . . . . .	227
<b>23.2.</b> Implementacja listy jednokierunkowej w C++ . . . . .	229
<b>23.3.</b> Drzewa binarne . . . . .	231
<b>23.4.</b> Inne struktury dynamiczne . . . . .	235
<b>Rozdział 24.</b> Wprowadzenie do programowania obiektowego . . . . .	237
<b>24.1.</b> Obiektowe postrzeganie świata . . . . .	237
<b>24.2.</b> Klasy i obiekty . . . . .	239
<b>24.3.</b> Przykłady modelowania obiektowego . . . . .	240
<b>24.4.</b> Hermetyzacja danych . . . . .	242
<b>24.5.</b> Konstruktory i destruktory . . . . .	245
<b>24.6.</b> Klasy wewnętrzne (zagnieżdżone) . . . . .	249
<b>Rozdział 25.</b> Przeciążanie funkcji i operatorów . . . . .	252
<b>25.1.</b> Przeciążanie funkcji . . . . .	252
<b>25.2.</b> Domyślne wartości parametrów . . . . .	255
<b>25.3.</b> Przeciążanie operatorów . . . . .	257

<b>Rozdział 26.</b> Funkcje i klasy zaprzyjaźnione .....	261
<b>26.1.</b> Jak definiować niektóre operatory? .....	261
<b>26.2.</b> Funkcje zaprzyjaźnione .....	262
<b>26.3.</b> Klasy zaprzyjaźnione .....	265
<b>Rozdział 27.</b> Dziedziczenie i polimorfizm .....	269
<b>27.1.</b> Dziedziczenie proste .....	269
<b>27.2.</b> Dziedziczenie wielobazowe .....	273
<b>27.3.</b> Polimorfizm .....	274
<b>Rozdział 28.</b> Przykład podsumowujący: sortowanie plików .....	279
<b>28.1.</b> Omówienie programu .....	279
<b>28.2.</b> Kod programu .....	281
 <b>Część IV Programowanie w języku Java</b>	
<b>Rozdział 29.</b> Podstawowe pojęcia .....	289
<b>29.1.</b> Koncepcja języka Java .....	289
<b>29.2.</b> Tworzenie i uruchamianie programów w Javie .....	291
<b>29.3.</b> Automatyczna obsługa pamięci w Javie .....	295
<b>Rozdział 30.</b> Java a C++ — podobieństwa i różnice .....	296
<b>30.1.</b> Java jako język obiektowy .....	296
<b>30.2.</b> Obiekty, referencje, porównywanie obiektów .....	298
<b>30.3.</b> Standardowe typy danych .....	301
<b>30.4.</b> Tablice .....	302
<b>30.5.</b> Strumień wejścia-wyjścia .....	303
<b>Rozdział 31.</b> Definicja i wykorzystanie klas w Javie .....	306
<b>31.1.</b> Definicja klasy w Javie .....	306
<b>31.2.</b> Kolekcje i ich zastosowanie .....	308
<b>31.3.</b> Wybrane klasy z biblioteki standardowej .....	310
<b>Rozdział 32.</b> Dziedziczenie w Javie. Interfejsy .....	314
<b>32.1.</b> Dziedziczenie proste .....	314
<b>32.2.</b> Polimorfizm w Javie .....	317
<b>32.3.</b> Interfejsy .....	319
<b>Rozdział 33.</b> Mechanizm wyjątków .....	323
<b>33.1.</b> Tradycyjna obsługa błędów .....	323



<b>33.2.</b> Wyjątki i ich obsługa	326
<b>33.3.</b> Hierarchia wyjątków	329
<b>Rozdział 34.</b> Tworzenie graficznego interfejsu użytkownika	331
<b>34.1.</b> Podstawy tworzenia aplikacji okienkowych w Javie	331
<b>34.2.</b> Dostępne kontrolki	334
<b>34.3.</b> Układ elementów w oknie	336
<b>34.4.</b> Obsługa zdarzeń	340
<b>34.5.</b> Rysowanie w oknie	344
<b>Rozdział 35.</b> Komponenty lekkie i ciężkie	347
<b>35.1.</b> Tworzenie aplikacji okienkowych w Swingu	347
<b>35.2.</b> Modyfikacja wyglądu okien i kontrolki	350
<b>Rozdział 36.</b> Aplety	352
<b>36.1.</b> Co to jest aplet Javy?	352
<b>36.2.</b> Jak pisać aplety?	354
<b>Rozdział 37.</b> Wstęp do programowania współbieżnego	360
<b>37.1.</b> Równoległe wykonanie programu	361
<b>37.2.</b> Tworzenie wątków	362
<b>37.3.</b> Potencjalne zagrożenia	363
<b>Rozdział 38.</b> Synchronizacja wątków	366
<b>38.1.</b> Metody synchronizowane	366
<b>38.2.</b> Synchronizowane bloki kodu	367
<b>38.3.</b> Komunikacja pomiędzy wątkami	369
<b>Rozdział 39.</b> Komunikacja sieciowa	373
<b>39.1.</b> Podstawy programowania sieciowego	373
<b>39.2.</b> Prosty serwer	375
<b>39.3.</b> Prosty klient	378
<b>Rozdział 40.</b> Przykład podsumowujący: gra sieciowa	381
<b>40.1.</b> Omówienie programu	381
<b>40.2.</b> Kod programu	383
<b>Bibliografia</b>	389
<b>Skorowidz</b>	390



## Rozdziały, które znajdują się na płycie CD

### Część V Programowanie w środowisku graficznym

<b>Rozdział 41.</b> Elementy składowe interfejsu użytkownika .....	395
<b>41.1.</b> Środowisko Dev-C++ .....	396
<b>41.2.</b> Dokumentacja MSDN .....	398
<b>41.3.</b> Okno i jego elementy składowe .....	399
<b>41.4.</b> Tworzenie nowego okna .....	402
<b>Rozdział 42.</b> Projektowanie aplikacji graficznej .....	406
<b>42.1.</b> Dodawanie kontrolki do okna aplikacji .....	406
<b>42.2.</b> Rodzaje kontrolki w WinAPI .....	409
<b>42.3.</b> Przykładowy projekt okna .....	412
<b>Rozdział 43.</b> Komunikaty i zdarzenia .....	416
<b>43.1.</b> Wstęp do programowania zdarzeniowego .....	416
<b>43.2.</b> Komunikaty i ich obsługa .....	417
<b>43.3.</b> Przykłady programów z obsługą komunikatów .....	419
<b>Rozdział 44.</b> Rysowanie na ekranie. Tworzenie animacji .....	422
<b>44.1.</b> Rysowanie na ekranie .....	422
<b>44.2.</b> Biblioteka graficzna WinAPI .....	428
<b>44.3.</b> Tworzenie animacji .....	430
<b>Rozdział 45.</b> Wykorzystanie fontów .....	434
<b>45.1.</b> Wypisywanie tekstów w oknie .....	434
<b>45.2.</b> Rodzaje fontów .....	436
<b>45.3.</b> Zmiana fontu tekstu .....	437
<b>Rozdział 46.</b> Tworzenie aplikacji SDI .....	441
<b>46.1.</b> Elementy składowe aplikacji SDI .....	441
<b>46.2.</b> Dodawanie menu użytkownika .....	442
<b>46.3.</b> Obsługa komunikatów menu .....	443
<b>Rozdział 47.</b> Tworzenie aplikacji MDI .....	449
<b>47.1.</b> Aplikacje MDI — podstawowe pojęcia .....	449
<b>47.2.</b> Zasady tworzenia aplikacji MDI .....	450
<b>47.3.</b> Przykładowa aplikacja MDI .....	451

<b>Rozdział 48.</b> Systemowe okna dialogowe .....	460
<b>48.1.</b> Okno wyboru pliku .....	460
<b>48.2.</b> Okno wyboru koloru .....	465
<b>48.3.</b> Okno wyboru fontu .....	466
<b>Rozdział 49.</b> Operacje na plikach .....	468
<b>49.1.</b> Zapis danych do pliku .....	468
<b>49.2.</b> Odczyt danych z pliku .....	472
<b>Rozdział 50.</b> Przykład podsumowujący: baza danych z interfejsem graficznym .....	474
<b>50.1.</b> Omówienie programu .....	475
<b>50.2.</b> Kod programu .....	476
 <b>Część VI Profesjonalne tworzenie oprogramowania</b>	
<b>Rozdział 51.</b> Projektowanie oprogramowania .....	487
<b>51.1.</b> Co to jest projekt informatyczny? .....	487
<b>51.2.</b> Zalecana zawartość dokumentacji projektowej .....	488
<b>51.3.</b> Modelowanie obiektowe .....	489
<b>Rozdział 52.</b> Optymalizacja kodu .....	492
<b>52.1.</b> Efektywne korzystanie z instrukcji iteracyjnych .....	492
<b>52.2.</b> Optymalny zapis instrukcji warunkowych .....	493
<b>52.3.</b> Obliczenia .....	494
<b>Rozdział 53.</b> Testowanie oprogramowania .....	496
<b>53.1.</b> Rola testowania .....	496
<b>53.2.</b> Zasady przeprowadzania testów .....	497
<b>53.3.</b> Testowanie w Eclipse za pomocą JUnit .....	498
<b>Rozdział 54.</b> Tworzenie dokumentacji programu .....	500
<b>54.1.</b> Treść dokumentacji technicznej .....	500
<b>54.2.</b> Narzędzia do automatycznego tworzenia dokumentacji .....	502
<b>Rozdział 55.</b> Narzędzia pracy grupowej .....	505
<b>55.1.</b> Systemy wersjonowania kodu .....	505
<b>55.2.</b> Instalacja wtyczki Subclipse .....	506
<b>55.3.</b> Podstawowa praca z repozytorium SVN .....	508
<b>55.4.</b> Rozwiązywanie konfliktów .....	510

---

<b>Dodatek A</b> Algorytmy i systemy liczbowe .....	513
<b>A.1.</b> Eliminacja Gaussa .....	513
<b>A.2.</b> Systemy pozycyjne .....	515
<b>A.3.</b> Funkcja printf .....	517
<b>Dodatek B</b> Programowanie w języku Java .....	521
<b>B.1.</b> Zastosowanie zarządcy rozkładu BorderLayout .....	521
<b>B.2.</b> Jak działa mechanizm prawidłowego zamykania okien w Swingu? .....	526
<b>B.3.</b> Tworzenie wątków poprzez rozszerzanie klasy Thread .....	528
<b>B.4.</b> Elementy pakietu Swing .....	529
<b>B.5.</b> Czy Java jest wolniejsza niż języki kompilowane do kodu maszynowego? ..	533



# 38

## Synchronizacja wątków

- W jaki sposób można zapewnić synchronizację wątków?
- Do czego służą metody `wait()` i `notify()`?

W poprzednim rozdziale widzieliśmy, jak nieprzewidywalne mogą być rezultaty uruchomienia programu wielowątkowego. To pokazuje wyraźnie, że w pewnych okolicznościach pożądane byłoby wymuszenie ograniczenia dostępu wątków do wspólnych danych — w tamtym przypadku do konsoli, na której wątki piszą. Java udostępnia mechanizmy synchronizacji, czyli zapewnienia dostępu do obiektu przez tylko jeden wątek na raz.

### 38.1. Metody synchronizowane

Metody synchronizowane to takie, które mogą być wywołane tylko przez jeden wątek na raz. Aby uczynić metodę synchronizowaną, dodajemy do jej definicji słowo kluczowe `synchronized`, np.

```
public void synchronized metoda();
```

Jeśli jeden wątek wywoła metodę synchronizowaną obiektu, żaden inny wątek nie będzie mógł wywołać tej ani żadnej innej metody synchronizowanej tego obiektu — będzie musiał poczekać, aż pierwszy wątek zakończy wykonywanie metody synchronizowanej. Bardzo ważne jest zrozumienie, że synchronizacja dotyczy konkretnego obiektu. Innymi słowy, wątek, wywołując metodę synchronizowaną, blokuje dla siebie obiekt. W niczym nie przeszkadza to innym wątkom wywoływać metody synchronizowane innych obiektów!

- wątek 1. wywołuje: `obiekt1.metoda1();`,
- wątek 2. nie może wywołać: `obiekt1.metoda1();`,
- ale może wywołać: `obiekt2.metoda1();`,

nawet jeśli `obiekt1` i `obiekt2` to obiekty tej samej klasy. (Zakładamy oczywiście, że `metoda1()` to metoda synchronizowana).

## 38.2. Synchronizowane bloki kodu

Inną metodą synchronizacji wątków w Javie jest zastosowanie synchronizowanych bloków kodu. Dowolny fragment kodu może być objęty klamrami `synchronized { }` i dzięki temu dostęp do niego zostanie ograniczony tylko do jednego wątku naraz. Metoda synchronizowana w naturalny sposób wiąże się z obiektem, na którym jest wywoływana. Synchronizowane bloki kodu muszą jawnie wskazywać, jakiego obiektu dotyczą:

```
synchronized (Obiekt, na którym synchronizujemy) {
    /* Kod, który może wykonać tylko jeden wątek na raz */
}
```

Ta początkowo dość dziwna konstrukcja staje się bardziej zrozumiała, gdy uświadomimy sobie, że Java blokuje wątkom dostęp do konkretnego obiektu. Oznacza to, że dowolny obiekt może być swoistą blokadą synchronizującą dla wątków. Dlatego właśnie w sekcji `synchronized` konieczne jest podanie obiektu, na którym chcemy synchronizować.

### UWAGA

W programowaniu współbieżnym mówimy o **monitorach** (lub *semaforach*), czyli mechanizmach gwarantujących wyłączny dostęp dla jednego wątku. W Javie rolę monitora pełni obiekt, na którym synchronizujemy.

Jaki obiekt wybrać jako barierę synchronizacji? Zazwyczaj najlepiej, gdy jest to obiekt będący wspólnym zasobem, do którego dostęp uzyskują wątki. W przykładzie programu, w którym wątki piszą na konsoli, może to być jakikolwiek obiekt wykorzystywany do pisania.

W stosunku do przykładu 37.2 wprowadzono pewne różnice — wątki nie piszą bezpośrednio na ekranie, ale poprzez wspólny obiekt o nazwie `konsola`. Ponadto do tego obiektu przeniesiono algorytm pisania po znaku i usypiania na pewien losowy czas.

Warto jednak sprawdzić, że to nie przeniesienie kodu piszącego do nowego obiektu, ale dodanie synchronizacji wprowadza porządek — tekst wypisywany jest całymi słowami (choć słowa mogą się pojawiać w przypadkowej kolejności).

### Przykład 38.1 Plik `Watek.java`

```
package podrecznik.synchronizacja;

public class Watek implements Runnable {
    private String slowo;
    private Thread watek;
    private int znak = 0;
```

```

private Konsola kon;

public Watek(String slowo, String id, Konsola kon) {
    this.slowo = new String(slowo);
    this.kon = kon;
    watek = new Thread(this, id);
    watek.start();
}

public void run() {
    kon.pisz(slowo);
}
}

```

### Plik Konsola.java

```

package podrecznik.synchronizacja;

public class Konsola {
    public synchronized void pisz(String s) {
        short znak = 0;
        while (znak < s.length()) {
            System.out.print(s.charAt(znak++));
            try {
                Thread.sleep((int) (Math.random() * 200));
            } catch (InterruptedException e) { }
        }
    }
}

```

### Plik KlasaGlowna.java

```

package podrecznik.synchronizacja;

public class KlasaGlowna {

    public static void main(String[] args) {
        String zdanie[] = { "Na ", "ten ", "czas ", "Wojski ", "chwycił ",
            "róg ", "długi, ", "czętkowany, ", "kręty " };
        Konsola kon = new Konsola();

        for (int i = 0; i < 9; i++) {
            new Watek(zdanie[i], "" + i, kon);
        }
    }
}

```



## 38.3. Komunikacja pomiędzy wątkami

Rozważmy przykład: dwa wątki operują na stosie. Jeden umieszcza elementy na stosie, drugi je ze stosu zdejmuje<sup>36</sup>. Oczywiście, nie można pozwolić, by oba wątki uzyskiwały dostęp do stosu jednocześnie, dlatego metody `push()` i `pop()` muszą być synchronizowane. Co jednak wtedy, gdy drugi wątek chce pobrać element ze stosu, ale na nim nie ma elementów? Musi poczekać, aż pierwszy wątek (producent) doda element na stos. Tu pojawia się problem — aktualnie to wątek konsumenta ma dostęp do obiektu stosu, zatem wątek producenta nie może niczego na nim umieścić!

Jak rozwiązać ten problem? Byłoby dobrze, gdyby wątek mógł „usnąć” i zwolnić synchronizowany obiekt (monitor) dla innego wątku, a potem „obudzić się”, gdy jest już możliwe wykonanie jego zadania. I rzeczywiście, w Javie jest taki mechanizm — są to metody `wait()` i `notify()`. Działa on następująco:

1. Wątek wykonujący kod (lub metodę) synchronizowany, który czeka na jakieś zdarzenie, wywołuje metodę `wait()`. W ten sposób zwalnia dostęp do obiektu.
2. Gdy inny wątek kończy wykonywanie kodu synchronizowanego, wywołuje metodę `notify()`, aby powiadomić (obudzić) inne wątki, że zakończył działanie, co być może oznacza dla tych wątków możliwość wykonania ich zadania.

Ten schemat został zilustrowany w przykładzie poniżej. Wątki `producent` i `konsument` w losowych odstępach czasu dodają i usuwają elementy ze stosu. Stos jest zaimplementowany jako zwykła tablica. Zmienna typu całkowitego wskazuje aktualną liczbę elementów na stosie. Dla ułatwienia dodano dwie metody, `pełny()` i `pusty()`, informujące o aktualnym stanie stosu. Jeśli stos jest pusty, wątek konsumenta „usypia”, wywołując metodę `wait()`. Podobnie zachowuje się wątek producenta, jeśli stos jest pełny. Każdy z wątków informuje drugi o wyjściu z sekcji synchronizowanej, wywołując metodę `notify()`.

### Przykład 38.2

#### Plik `Stos.java`

```
package podrecznik.synchronizacja;

public class Stos {
    private Integer tablica[];
    private volatile int pozycja;
    private final int maxStos = 10;

    public Stos() {
        tablica = new Integer[maxStos];
        pozycja = -1;
    }
}
```

<sup>36</sup> Takie zadania noszą nazwę problemów producentów i konsumentów.

```
public synchronized Integer pop() {
    if (pusty()) {
        try {
            System.out.println("Konsument czeka na elementy stosu.");
            wait();
        } catch (InterruptedException e) { } // Nic nie rób po wybudzeniu
    }
    Integer element = tablica[pozycja];
    tablica[pozycja] = null;
    pozycja--;
    System.out.println("Zdjęto element. Aktualna pozycja stosu: " +
    pozycja);
    notify(); // Powiadom wątek producenta
    return element;
}

public synchronized void push(Integer i) {
    if (pełny()) {
        try {
            System.out.println("Producent czeka, aż się zwolni miejsce na
            stosie.");
            wait();
        } catch (InterruptedException e) { }
    }
    tablica[++pozycja] = i;
    System.out.println("Dodano element. Aktualna pozycja stosu: " +
    pozycja);
    notify();
}

private boolean pusty() {
    if (pozycja < 0)
        return true;
    return false;
}

private boolean pełny() {
    if (pozycja == maxStos - 1)
        return true;
    return false;
}
}
```

## Plik Producent.java

```
package podrecznik.synchronizacja;

public class Producent extends Thread {
    private Stos stos;

    public Producent(String id, Stos stos) {
        super(id);
        this.stos = stos;
        start();
    }

    public void run() {
        while (true) {
            System.out.println(getName() + ": dodaje na stos...");
            stos.push(new Integer(1));
            try {
                sleep((int) (Math.random()*200));
            } catch (InterruptedException e) { }
        }
    }
}
```

## Plik Konsument.java

```
package podrecznik.synchronizacja;

public class Konsument extends Thread {
    private Stos stos;

    public Konsument(String id, Stos stos) {
        super(id);
        this.stos = stos;
        start();
    }

    public void run() {
        while (true) {
            System.out.println(getName() + ": zdejmuję ze stosu...");
            stos.pop();
            try {
                sleep((int) (Math.random()*200));
            } catch (InterruptedException e) { }
        }
    }
}
```

**Plik Test.java**

```

package podrecznik.synchronizacja;

public class Test {

    public static void main(String[] args) throws InterruptedException {
        Stos stos = new Stos();
        new Producent("producent", stos);
        new Konsument("konsument", stos);
        System.out.println("Koniec pracy wątku głównego");
    }

}

```

W naszym przykładzie wątki będą pracować w pętli nieskończonej (`while (true)`). Aby zatrzymać działanie programu, należy zakończyć proces w systemie. W środowisku Eclipse można to uczynić, klikając czerwony kwadracik w prawym dolnym rogu ekranu. Jako ćwiczenie pozostawiamy zmianę pętli, tak by np. zatrzymywała się po 10 sekundach (sprawdzanie czasu systemowego `System.currentTimeMillis()`).

**PYTANIA KONTROLNE**

1. Jakie są dwie metody synchronizacji wątków w Javie?
2. Wątek A wywołuje synchronizowaną metodę `m1()` obiektu `Ob1`. Czy wątek B może wywołać w tym czasie metodę synchronizowaną `m2()` obiektu `Ob1`? A obiektu `Ob2` tej samej klasy?
3. Do czego służą metody `wait()` i `notify()`?

**ĆWICZENIA**

1. Zmodyfikuj program z przykładu 38.2 tak, aby jednocześnie działało wielu konsumentów.
2. Zmodyfikuj program z przykładu 38.2 tak, aby jednocześnie działało wielu producentów.

# 39

## Komunikacja sieciowa

- Jak nawiązać połączenie sieciowe?
- W jaki sposób można czytać i pisać przez sieć?
- Jaką strukturę ma schemat budowy aplikacji serwera i klienta?

Wraz z upowszechnieniem się internetu coraz więcej programów przesyła informacje w sieci. Komunikatory przesyłają i odbierają wiadomości pisane przez rozmawiające ze sobą osoby, przeglądarki internetowe pobierają treść stron WWW, programy użytkowe automatycznie aktualizują swoje dane itd. W jaki sposób nawiązać połączenie pomiędzy dwoma programami i przesyłać dane? Jak dowiemy się z tego rozdziału, należy skorzystać ze specjalnego obiektu, tzw. gniazda (ang. *socket*)<sup>37</sup>.

### 39.1. Podstawy programowania sieciowego

Programy korzystające z sieci używają gniazd (ang. *socket*). Gniazdo to zakończenie dwukierunkowego łącza, podobnie jak gniazdko w ścianie jest zakończeniem np. linii telefonicznej. Za pomocą gniazda programy określają, z kim chcą się komunikować.

#### DEFINICJA

**Gniazdo** — zakończenie abstrakcyjnego łącza komunikacji sieciowej pomiędzy dwoma programami. Gniazdo jest opisane przez adres IP oraz numer portu.

Aby określić adresata komunikacji w sieci, konieczne jest podanie *adresu IP oraz numeru portu*. Adres IP składa się z czterech liczb z przedziału 0 – 255 rozdzielonych kropkami,

<sup>37</sup> Oczywiście programy sieciowe można pisać w dowolnym języku. W podręczniku temat ten omówiono na przykładzie Javy.

np. 194.178.29.1. Każdy komputer, nawet niepodłączony do internetu, ma swój lokalny adres 127.0.0.1. Dzięki temu możemy testować programy sieciowe, nie mając dostępu do sieci komputerowej. Z kolei numer portu pozwala na korzystanie z sieci przez wiele programów jednocześnie — każdy program korzysta z innego portu i dzięki temu system operacyjny potrafi dostarczyć właściwym programom właściwe paczki danych, mimo że do wszystkich odnosi się ten sam adres IP. To rozwiązanie przypomina trochę strukturę bloku mieszkalnego — ten sam numer ulicy, ale różne numery mieszkań.

### UWAGA

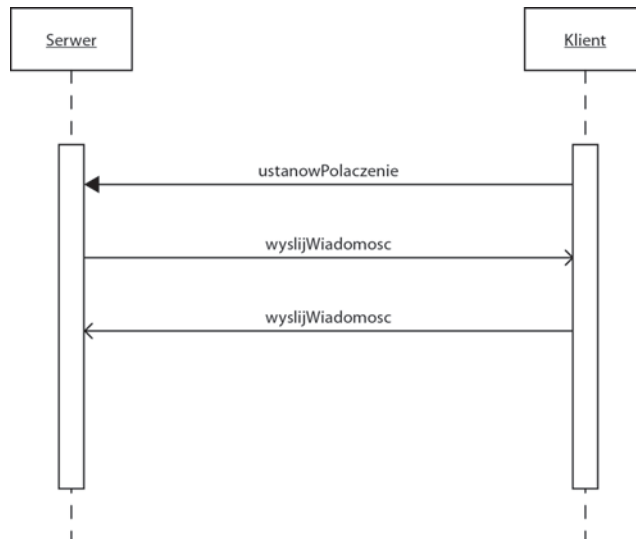
Jeśli kod programu wygląda poprawnie, a mimo to program sieciowy nie chce działać, warto spróbować zmienić używany numer portu. Być może jakiś inny program już korzysta z tego portu, uniemożliwiając tym samym jego wykorzystanie przez nasz proces. Przyczyną może być również włączona zapora sieciowa.

Z punktu widzenia komunikacji sieciowej wyróżniamy dwa rodzaje programów — *serwery* i *klientów*. *Serwer* zazwyczaj udostępnia gniazdo do komunikacji i czeka, nasłuchując, czy ktoś chce się z nim połączyć. Jeśli jakiś inny program ustanowi połączenie z serwerem, ten przejdzie w tryb obsługi tego połączenia. Z kolei *klient* to właśnie ten program, który łączy się z serwerem, a następnie komunikuje się z nim („rozmawia”). Zasadnicza różnica polega na tym, że serwer nie wie, z kim będzie się komunikował — udostępnia gniazdo i czeka, z kolei klient musi znać adres i port serwera, z którym chce się połączyć.

Mechanizm ten przedstawiono symbolicznie na rysunku 39.1. Oś pionowa wyobraża czas, jej zwrot jest skierowany w dół. Wąskie, pionowe prostokąty pod obiektami **Serwer** i **Klient** wyobrażają wykonywanie ich kodu. Poziome strzałki to przesyłanie komunikatów. Najpierw klient zgłasza się do serwera, prosząc o ustanowienie połączenia. W odpowiedzi serwer konfiguruje połączenie i rozpoczyna się (zazwyczaj) naprzemienne przesyłanie wiadomości.

**Rysunek 39.1.**

Schemat komunikacji serwer-klient



## 39.2. Prosty serwer

Jak zbudować prostą aplikację serwerową? Schemat postępowania jest zawsze taki sam:

1. Utwórz gniazdo serwera.
2. Czekaj na połączenia od klientów.
3. Gdy klient się podłączy, utwórz nowe gniazdo do komunikacji z klientem.
4. Otwórz strumienie do pisania i czytania z gniazda klienckiego.
5. Czytaj i pisz za pośrednictwem gniazda klienckiego.
6. Zamknij strumienie i gniazda.

Utworzenie gniazda serwerowego (do nasłuchiwania i czekania na klientów) jest wykonywane za pomocą konstruktora klasy `ServerSocket`:

```
gniazdoSerwera = new ServerSocket(port);
```

gdzie zmienna `port` typu `int` przechowuje numer portu, np. 5555. Jak widać, serwer nie potrzebuje znać swojego adresu IP. Następnie wywoływana jest metoda `accept` obiektu gniazda, która zawiesza swoje działanie do czasu zgłoszenia się klienta.

```
gniazdoObslugiKlienta = gniazdoSerwera.accept();
```

Metoda ta jest zwykle wywoływana bezpośrednio po utworzeniu gniazda, ale powrót z niej następuje dopiero po nawiązaniu połączenia. Zwracaną wartością jest referencja do nowego obiektu typu `Socket`. To nowe gniazdo służy do komunikacji z nowo połączonym klientem. Dlaczego serwer otwiera nowe gniazdo? Chodzi o to, by na tym pierwszym w dalszym ciągu oczekiwać (nasłuchiwać) na kolejnych klientach.

### UWAGA

W profesjonalnych programach zwykle tworzy się osobne wątki do obsługi każdego klienta, tak aby wątek główny programu mógł stale nasłuchiwać na nowych klientach. W omówionym tutaj prostym schemacie kolejni dołączający klienci będą czekać na gnieździe, aż wątek główny serwera skończy obsługiwać pierwszego klienta.

Kolejny, czwarty krok naszego schematu to otwarcie strumieni do pisania i czytania. Gniazdo jest interfejsem umożliwiającym przesyłanie danych przez sieć, ale — podobnie jak plik czy konsola — wymaga dostępu za pośrednictwem strumieni. Standardowo strumienie te otwiera się następująco:

```
BufferedReader in = new BufferedReader(new InputStreamReader(gniazdo.  
getInputStream()));  
PrintWriter out = new PrintWriter(gniazdo.getOutputStream(), true);
```

gdzie `gniazdo` oznacza gniazdo wykorzystywane do obsługi klienta. Mając do dyspozycji strumienie `in` i `out`, możemy z nich korzystać tak jak z innych strumieni i nie martwić się już o to, że odnoszą się do gniazda:

```
out.println("Witaj, kliencie!");  
in.readLine();
```



Na końcu pracy programu konieczne jest jeszcze tylko zamknięcie strumieni i gniazd:

```
in.close();
out.close();
gniazdo.close();
```

Poniższy przykład zawiera pełny kod programu serwera. Warto zwrócić uwagę na kolejne kroki nawiązywania połączenia i przesyłania komunikatów. Serwer wysyła wiadomości do klienta, dopóki ten odpowiada (pętla `while(line != null)`). Wszelkie błędy zostały obsługane przez mechanizm wyjątków.

### Przykład 39.1

```
package podrecznik;

import java.io.*;
import java.net.*;

public class Serwer {
    private ServerSocket gniazdoSerwera = null;
    private Socket gniazdoObslugiKlienta = null;
    private BufferedReader in = null;
    private PrintWriter out = null;
    private String line = "";
    private int port = 5555;

    public Serwer() {
        sluchaj();
        ustanowPolaczenie();
        rozmawiaj();
        // Przyjmij jeszcze trzech klientów
        for (int i = 0; i < 3; i++) {
            ustanowPolaczenie();
            rozmawiaj();
        }
    }

    public void ustanowPolaczenie() {
        try {
            gniazdoObslugiKlienta = gniazdoSerwera.accept();
            System.out.println("[Serwer] Zaakceptowano połączenie.");
        } catch (IOException e) {
            System.err.println("[Serwer] Metoda accept zawiodła.");
            System.exit(-1);
        }

        try {
            in = new BufferedReader(new InputStreamReader(gniazdoObslugiKlienta.getInputStream()));
```

```
        out = new PrintWriter(gniazdoObslugiKlienta.getOutputStream(),
true);
    } catch (IOException e) {
        System.err.println("[Serwer] Nie można utworzyć strumieni.");
        System.exit(-1);
    }
}

public void rozmawiaj() {
    while (line != null) {
        try {
            line = in.readLine();
            if (line != null) {
                System.out.println("[Serwer] Otrzymano wiadomość: " +
line);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                out.println("dziękuję, kliencie.");
            }
        } catch (IOException e) {
            System.err.println("[Serwer] Błąd odczytu z gniazda -
prawdopodobnie klient rozłączył się.");
            break;
        }
        line = "";
    }

    public void sluchaj() {
        try {
            System.out.println("[Serwer] Serwer rozpoczyna nasłuchiwanie na
porcie " + port);
            gniazdoSerwera = new ServerSocket(port);
        } catch (IOException e) {
            System.err.println("[Serwer] Nie można nasłuchiwać na porcie: "
+ port);
            System.exit(-1);
        }
    }

    public void zwolnijZasoby() {
        try {
```

```

        in.close();
        out.close();
        gniazdoSerwera.close();
    } catch (IOException e) {
        System.err.println("[Serwer] Błąd metody close.");
        System.exit(-1);
    }
}

public static void main(String[] args) {
    Serwer serwer = new Serwer();
    serwer.zwolnijZasoby();
}
}

```

## 39.3. Prosty klient

Po poznaniu schematu działania serwera łatwo będzie nam zrozumieć działanie aplikacji klienckiej. Jej działanie można opisać następującymi krokami:

1. Utwórz gniazdo serwera.
2. Otwórz strumienie do pisania i czytania z gniazda.
3. Czytaj i pisz za pośrednictwem gniazda.
4. Zamknij strumienie i gniazdo.

Nawiązanie połączenia polega na utworzeniu obiektu gniazda. Tym razem podajemy również adres IP serwera, do którego chcemy się podłączyć:

```
gniazdo = new Socket(ip, port);
```

Jeśli wszystko pójdzie zgodnie z planem, możemy — podobnie jak w przypadku serwera — otworzyć strumienie do czytania i pisania, a następnie rozpocząć komunikację. Reszta kodu klienta niewiele już różni się od kodu serwera. Cały kod przedstawiono w przykładzie poniżej.

### Przykład 39.2

```

package podrecznik;

import java.io.*;
import java.net.*;

public class Klient {
    private Socket gniazdo = null;
    private PrintWriter out = null;
    private BufferedReader in = null;
    private int port = 5555;
    private String ip = "127.0.0.1";
}

```

```
public Klient() {
    ustanowPolaczenie();
    wyslij("Witaj, serwerze!");
    try {
        String line = in.readLine();
        System.out.println("[Klient] Otrzymano wiadomość: " + line);
        out.println("Dziękuję, serwerze!");
    } catch (IOException e) {
        System.out.println("Read failed");
        System.exit(1);
    }
}

public void zwolnijZasoby() {
    try {
        in.close();
        out.close();
        gniazdo.close();
    } catch (IOException e) {
        System.err.println("[Klient] Błąd metody close.");
    }
}

private void wyslij(String s) {
    out.println(s);
}

public void ustanowPolaczenie() {
    try {
        gniazdo = new Socket(ip, port);
        out = new PrintWriter(gniazdo.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(gniazdo.
getInputStream()));
        System.out.println("[Klient] Podłączono do serwera " + ip + ": "
+ port);
    } catch (UnknownHostException e) {
        System.err.println("[Klient] Nie można połączyć z: " + ip);
        System.exit(1);
    } catch (IOException e) {
        System.err.println("[Klient] Błąd wejścia-wyjścia");
        System.exit(1);
    }
}
```

```
public static void main(String[] args) {  
    Klient klient = new Klient();  
    klient.zwolnijZasoby();  
}  
}
```

### PYTANIA KONTROLNE

1. Co to jest gniazdo i do czego służy?
2. Wymień kolejne kroki w typowym schemacie działania serwera.
3. Wyjaśnij, czym różni się działanie serwera od działania klienta.
4. Do czego wykorzystuje się strumienie w programach sieciowych?

### ĆWICZENIA

1. Jeśli masz dostęp do sieci, uruchom program serwera i klienta na dwóch różnych komputerach. Wskazówka: konieczne będzie wpisanie w kodzie poprawnego adresu IP.
2. Zmodyfikuj program klienta tak, aby przesyłał do serwera napisy wprowadzane przez użytkownika na konsoli.

# 40

## Przykład podsumowujący: gra sieciowa

- Przykład podsumowujący poznane elementy programowania w Javie, w szczególności komunikację sieciową i graficzny interfejs użytkownika.

W celu utrwalenia poznanych technik programowania w języku Java przedstawimy program umożliwiający grę w kółko i krzyżyk w sieci. Napišemy dwa programy — serwera i klienta. Ich rola w komunikacji sieciowej będzie inna, ale gdy połączenie zostanie nawiązane, oba będą w identyczny sposób umożliwiać grę.

### 40.1. Omówienie programu

Struktura obu programów — serwera i klienta — jest oparta na klasie bazowej `JFrame`. W konstruktorze tworzymy okienko aplikacji i dodajemy do niego dziewięć przycisków, które będą reprezentować pola planszy. Każdemu z nich przypisujemy jako słuchacza zdarzeń klasę okna aplikacji. Dodatkowo, aby móc rozróżniać zdarzenia pochodzące od przycisków, korzystamy z metody `setActionCommand(String)`, która dodaje zmienną napisową do każdego komunikatu (`Event`).

```
przyciski = new JButton[9];
for (int i = 0; i < 9; i++) {
    przyciski[i] = new JButton("");
    przyciski[i].setActionCommand("" + i); //Dodanie napisu (numer) do komunikatów
    przyciski[i].addActionListener(this);
    getContentPane().add(przyciski[i]);
}
```

Napis ten można wydobyć z klasy komunikatu o zdarzeniu za pomocą metody `getActionCommand` klasy `Event`. W ten sposób identyfikujemy przycisk, który został naciśnięty. Tę czynność wykonujemy w kodzie metody `actionPerformed`.

```
int j = Integer.parseInt(arg0.getActionCommand());
```

W procedurze obsługi zdarzeń sprawdzamy, czy aktualnie jest nasz ruch (służy do tego zmienna logiczna `mojRuch`). Jeśli nie, ignorujemy kliknięcia przycisków. Jeśli natomiast jest nasz ruch, zmieniamy tekst odpowiedniego przycisku na `O` oraz przesyłamy współrzędną pola (`ruch`) do klienta.

Komunikacja sieciowa jest zaimplementowana zgodnie z poznanym wcześniej schematem: najpierw metoda `sluchaj()` tworzy gniazdo serwera, następnie czeka na podłączenie klienta, wreszcie otwiera strumień (buforowane) do czytania i pisania. Potem wywoływana jest metoda `odbieraj()`. Metoda ta w pętli próbuje czytać linię tekstu ze strumienia powiązanego z gniazdem. Gdy taka się pojawi, jest interpretowana i plansza zostaje odświeżona — pojawia się krzyżyk w odpowiednim miejscu. Zaktualizowana zostaje też zmienna `mojRuch`, ponieważ nadejście informacji o ruchu wykonanym przez przeciwnika jest równoznaczne z tym, że teraz nasza kolej.

```
while ((line = in.readLine()) != null) {
    int j = Integer.parseInt(line);
    System.out.println("[Serwer] <== " + j);
    if (j >= 0 && j < 9) {
        przyciski[j].setText("X");
    }
    mojRuch = true;
}
```

Oczywiście wszystkie wymagające tego fragmenty kodu są otoczone blokami `try-catch` w celu obsługi błędów. Dodatkowo dla lepszego zrozumienia w wielu miejscach program wypisuje na konsoli informacje pomocnicze.

W metodzie `odbieraj()` dziwić może pętla `while`. Jak to możliwe, że program, wchodząc w tę pętlę, dalej reaguje na kliknięcia myszą? Nie zaimplementowano przecież jawnie wielowątkowości. A jednak program działa. Aby lepiej zrozumieć tę kwestię, musimy przypomnieć sobie sposób działania strumieni buforowanych — wywołanie metody `readLine()` jest *blokowane* (zawieszane) do czasu, aż strumień będzie gotowy (pojawi się ciąg znaków zakończony `\n`). Przez ten czas program może obsługiwać komunikaty zdarzeń, np. kliknięcia przycisków. I tak jest w naszym programie.

Program klienta jest bardzo podobny do serwera. Jedyna różnica polega na sposobie nawiązywania połączenia.



## 40.2. Kod programu

### Przykład 40.1

#### Plik Serwer.java

```
package podrecznik;

import java.awt.GridLayout;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class Serwer extends JFrame implements ActionListener {
    protected JButton przyciski[];
    protected boolean krzyzyk = true;

    private ServerSocket gniazdoSerwera = null;
    private Socket gniazdoObslugiKlienta = null;
    private BufferedReader in = null;
    private PrintWriter out = null;
    private String line = "";
    private int port = 4567;

    // Serwer zaczyna
    private boolean mojRuch = true;

    public Serwer() {
        super("OXO Serwer");
        setLayout(new GridLayout(3, 3));
        przyciski = new JButton[9];
        for (int i = 0; i < 9; i++) {
            przyciski[i] = new JButton("");
            przyciski[i].setActionCommand("" + i);
            przyciski[i].addActionListener(this);
            getContentPane().add(przyciski[i]);
        }
        setSize(300, 300);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
    }
}
```

```
        }

        zwolnijZasoby();
    }
});

setVisible(true);
sluchaj();
odbieraj();
}

public void actionPerformed(ActionEvent arg0) {
    if (!mojRuch)
        return;
    int j = Integer.parseInt(arg0.getActionCommand());
    if (j >= 0) {
        if ("".equals(przyciski[j].getText())) {
            przyciski[j].setText("0");
            // Wyślij informację o wykonanym ruchu do klienta
            System.out.println("[Serwer] ==> " + j);

            repaint();
            out.println(j);
        }
    }
    mojRuch = false;
    this.setTitle("Oczekiwanie na ruch przeciwnika");
}

public void odbieraj() {
    try {
        while ((line = in.readLine()) != null) {
            int j = Integer.parseInt(line);
            System.out.println("[Serwer] <== " + j);
            if (j >= 0 && j < 9) {
                przyciski[j].setText("X");
            }
            mojRuch = true;
            this.setTitle("Wykonaj ruch");
        }
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
```

```
public void sluchaj() {
    try {
        System.out.println("[Serwer] Serwer rozpoczyna nasłuchiwanie na
porcie "+ port);
        gniazdoSerwera = new ServerSocket(port);
    } catch (IOException e) {
        System.err.println("[Serwer] Nie można nasłuchiwać na porcie: "
+ port);
        System.exit(-1);
    }

    try {
        gniazdoObslugiKlienta = gniazdoSerwera.accept();
        System.out.println("[Serwer] Zaakceptowano połączenie.");
    } catch (IOException e) {
        System.err.println("[Serwer] Metoda accept zawiodła.");
        System.exit(-1);
    }

    try {
        in = new BufferedReader(new InputStreamReader(
            gniazdoObslugiKlienta.getInputStream()));
        out = new PrintWriter(gniazdoObslugiKlienta.getOutputStream(),
true);
    } catch (IOException e) {
        System.err.println("[Serwer] Nie można utworzyć strumieni.");
        System.exit(-1);
    }
}

public void zwolnijZasoby() {
    try {
        in.close();
        out.close();
        gniazdoSerwera.close();
        gniazdoObslugiKlienta.close();
    } catch (IOException e) {
        System.err.println("[Serwer] Błąd metody close.");
        System.exit(-1);
    }
}

public static void main(String[] args) {
    new Serwer();
}
}
```

**Plik Klient.java**

```

package podrecznik;

import java.awt.GridLayout;
import java.awt.event.*;
import java.io.*;
import java.net.*;

import javax.swing.JButton;
import javax.swing.JFrame;

public class Klient extends JFrame implements ActionListener {
    private Socket gniazdo = null;
    private PrintWriter out = null;
    private BufferedReader in = null;
    private int port = 4567;
    private String ip = "127.0.0.1";
    private String line = "";

    // Zaczyna serwer
    private boolean mojRuch = false;

    private JButton przyciski[];

    public Klient() {
        super("OXO Klient");
        setLayout(new GridLayout(3, 3));
        przyciski = new JButton[9];
        for (int i = 0; i < 9; i++) {
            przyciski[i] = new JButton("");
            przyciski[i].setActionCommand("" + i);
            przyciski[i].addActionListener(this);
            getContentPane().add(przyciski[i]);
        }
        setSize(300, 300);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
                try {
                    in.reset();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
        zwolnijZasoby();
    }
}

```

```
});

setVisible(true);

ustanowPolaczenie();
odbieraj();
}

public void actionPerformed(ActionEvent arg0) {
    if (!mojRuch)
        return;
    int j = Integer.parseInt(arg0.getActionCommand());
    if (j >= 0) {
        if (przyciski[j].getText().equals("")) {
            przyciski[j].setText("X");
            // Wyślij informację o wykonanym ruchu do serwera
            System.out.println("[Klient] ==> " + j);

            repaint();
            out.println(j);
        }
    }
    mojRuch = false;
    this.setTitle("Oczekiwanie na ruch przeciwnika");
}

public void odbieraj() {
    try {
        while ((line = in.readLine()) != null) {
            int j = Integer.parseInt(line);
            System.out.println("[Klient] <== " + j);
            if (j >= 0 && j < 9) {
                przyciski[j].setText("O");
            }
            mojRuch = true;
            this.setTitle("Wykonaj ruch");
        }
    } catch (IOException e1) {
        e1.printStackTrace();
    }
}

public void zwolnijZasoby() {
    try {
```

```

        in.close();
        out.close();
        gniazdo.close();
    } catch (IOException e) {
        System.err.println("[Klient] Błąd metody close.");
    }
}

public void ustanowPolaczenie() {
    try {
        gniazdo = new Socket(ip, port);
        out = new PrintWriter(gniazdo.getOutputStream(), true);
        in = new BufferedReader(new InputStreamReader(
            gniazdo.getInputStream()));
        System.out.println("[Klient] Podłączono do serwera " + ip + ": "
+ port);

        } catch (UnknownHostException e) {
            System.err.println("[Klient] Nie można połączyć z: " + ip);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("[Klient] Błąd wejścia-wyjścia");
            System.exit(1);
        }
    }

    public static void main(String[] args) {
        new Klient();
    }
}

```

## ĆWICZENIA

1. Napisz funkcję przeglądającą planszę w poszukiwaniu linii złożonej z krzyżyków lub kółek. Funkcja ta powinna przekazać informację o zakończeniu gry i o zwycięzcy (np. wyświetlić na konsoli).
2. Dodaj możliwość rozmów pomiędzy graczami.
3. Napisz algorytm logiki gry w kółko i krzyżyk, tak aby móc grać z przeciwnikiem komputerowym. Wybór najlepszego ruchu powinien uwzględniać następujące czynniki:
  - a) Czy przeciwnik może wygrać w następnym ruchu? Jeśli tak, zablokuj go.
  - b) Czy ja mogę wygrać w następnym ruchu? Jeśli tak, wykonaj taki ruch.

# Skorowidz

- .
- .A, 42
- .DLL, 42
- .LIB, 42
- .O, 42
- .OBJ, 42
- .SO, 42
- A**
- algorytmy, 18, 19
  - bisekcji, 31, 32
  - liniowe, 31
  - połowienia, 31, 32
  - schemat Hornera, 33
  - sortowanie liczb, 28, 29, 30
  - wyszukiwanie, 31, 32
  - zapis, 22, 23, 24
  - znajdowanie miejsc zerowych funkcji, 34
- assembler, 147
- AVL, drzewo, 235
- B**
- B-drzewo, 235
- biblioteki, 42
  - dynamiczne, 42, 43
  - statyczne, 42
- binarny, system, *Patrz* system dwójkowy
- błąd
  - kompilacji, 40
  - składniowy, 39
  - uruchomienia, 41
- boxing, 302
- breakpoint, *Patrz* punkt przerwania
- C**
- C++, 167
  - &, operator, 204
  - \*, operator, 204
  - ::, operator, 273
  - a Java, 296
  - a Pascal, 186, 187
  - bool, typ danych, 175
  - char, typ danych, 174
  - cin, obiekt, 217, 220
  - const, słowo kluczowe, 253
  - cout, obiekt, 217, 218
  - delete, operator, 228
  - destruktor, 248
  - do ... while, pętla, 180, 181
  - double, typ danych, 175
  - drzewa binarne, 231, 232, 233, 234, 235
  - dynamiczna alokacja pamięci, 228
  - dziedziczenie proste, 269, 270, 271, 272
  - dziedziczenie wielobazowe, 273
  - enkapsulacja, 244
  - feof, funkcja, 224
  - float, typ danych, 175
  - fopen, funkcja, 224
  - for, pętla, 179, 180, 190, 191
  - fprintf, funkcja, 223
  - fscanf, funkcja, 223
  - funkcje, 181, 182
  - funkcje zaprzyjaźnione, 262, 263, 265
  - hermetyzacja danych, 242, 244, 245
  - if, instrukcja, 179
  - include, dyrektywa, 182
  - instrukcje, 178
  - int, typ danych, 174
  - islower, funkcja, 224
  - klasa, 239, 240
  - klasy wewnętrzne, 249
  - klasy zaprzyjaźnione, 265, 266
  - komentarze, 178
  - konstruktor, 245, 246
  - konstruktor kopiujący, 253
  - Linux, 171
  - lista jednokierunkowa, 229, 230, 231
  - long long int, typ danych, 175
  - long long, typ danych, 175
  - long, typ danych, 174
  - main, funkcja, 182
  - napisy, 202, 203
  - new, operator, 228
  - obiekt, 239, 240
  - operacje wejścia-wyjścia, 217, 218, 219, 220, 221, 222, 223
  - operatory, 175, 176, 177
  - parametry, domyślne wartości, 255, 256, 257
  - pliki, 223, 224
  - pobrania adresu, operator, 204
  - pola bitowe, 213, 214
  - polimorfizm, 274, 275
  - printf, funkcja, 220, 221, 222
  - private, słowo kluczowe, 243
  - przeciążanie funkcji, 252, 253, 254
  - przeciążanie operatorów, 257, 259
  - przestrzenie nazw, 183
  - public, słowo kluczowe, 242
  - return, instrukcja, 181
  - rozszerzenia plików źródłowych, 182
  - scanf, funkcja, 223
  - short int, typ danych, 174
  - short, typ danych, 174
  - signed char, typ danych, 174
  - signed int, typ danych, 174
  - signed long long int, typ danych, 175
  - signed long long, typ danych, 175
  - signed long, typ danych, 174
  - signed short int, typ danych, 174
  - signed short, typ danych, 174
  - signed, typ danych, 174
  - słowa kluczowe, 173
  - sprintf, funkcja, 224
  - sscanf, funkcja, 224
  - std::cin, instrukcja, 178
  - std::cout, instrukcja, 178
  - struktura programu, 182
  - struktury, 210, 211, 212, 214, 215
  - switch, instrukcja, 189
  - tablice, 195, 196, 197, 198, 207, 208
  - tablice dynamiczne, 227, 228
  - tablice wielowymiarowe, 200, 201
  - tablice znaków, 202, 203
  - this, 259
  - typy danych, 174
  - unie, 213
  - unsigned char, typ danych, 174
  - unsigned int, typ danych, 174
  - unsigned long long int, typ danych, 174
  - unsigned long long, typ danych, 174
  - unsigned long, typ danych, 174
  - unsigned short int, typ danych, 174
  - unsigned short, typ danych, 174
  - unsigned, typ danych, 174
  - void, typ, 181
  - while, pętla, 180
  - wskaźniki, 204, 205, 206, 207, 208
  - wyluskania, operator, 204
  - wyrażenia, 175
  - znaki specjalne, 193, 194
- Cezara, szyfr, 224
- ciąg Fibonacciego, 123
- compilation error, *Patrz* błąd kompilacji
- cykl von Neumanna, 37
- D**
- debuger, 46, 47
- deklaracja zmiennej, 74
- Delphi, 57
- destruktor, 248
- Dev C++, 168, 169
- Dev-Pascal, 53, 54
- diagramy, 24
- drzewo AVL, 235
- drzewo binarne, 231, 232, 233, 235
  - korzeń, 232
  - liść, 232
  - węzeł, 232
- dwójkowy, system, 38

- dziedziczenie  
 proste, 269, 270  
 wielobazowe, 273
- E**  
 Eclipse IDE, 48, 171, 172, 291  
 edytory, 44, 45  
 enkapsulacja, 244
- F**  
 Fibonacciego, ciąg, 123  
 formatowanie kodu, 45  
 FPC, 55, 56  
 Free Pascal Compiler, *Patrz* FPC
- G**  
 gniazdo, 373  
 GNU GPL, 54  
 graf skierowany, 235
- H**  
 heksadecymalny, system, *Patrz* system szesnastkowy  
 hermetyzacja danych, 242, 244, 245  
 Hornera, schemat, 32, 33  
 Hugo, 55
- I**  
 IDE, 47, 48  
 implementacja, 22  
 inicjalizacja zmiennej, 75  
 instancja, 239, 308  
 instrukcja iteracyjna, 84, 91, 92, 93, 94, 95  
 instrukcja warunkowa, 84, 85, 86, 88, 179, 187, 189  
   zagnieżdżanie, 88  
 instrukcja wyboru, 89, 90, 189  
 interpreter, 41
- J**  
 Java, 289, 290, 291  
   a C++, 296  
   abs, metoda, 311  
   actionPerformed, metoda, 340  
   aplety, 352, 353, 354, 355  
   aplikacje okienkowe, 331, 332, 333, 334, 335, 336, 337, 347, 348, 350  
   ArithmeticException, wyjątek, 330  
   AWT, pakiet, 331, 332  
   biblioteka standardowa, 310  
   boolean, typ danych, 301  
   BorderLayout, 338  
   Button, kontrolka, 334  
   byte, typ danych, 301  
   Canvas, kontrolka, 334  
   CardLayout, 339  
   char, typ danych, 301  
   charAt, metoda, 300  
   Checkbox, kontrolka, 334  
   Choice, kontrolka, 334  
   destroy, metoda, 354  
   double, typ danych, 301  
   dziedziczenie proste, 314, 315, 316  
   equals, metoda, 300  
   exp, metoda, 311  
   extends, słowo kluczowe, 314  
   final, 307  
   float, typ danych, 301  
   FlowLayout, 337  
   Garbage Collector, 295  
   gc, metoda, 304  
   Graphics, klasa, 344  
   GridBagLayout, 338  
   GridLayout, 337, 338  
   hasMoreTokens, metoda, 311  
   hierarchia wyjątków, 329  
   import, instrukcja, 297  
   in, strumień, 304  
   indexOf, metoda, 300  
   init, metoda, 354  
   int, typ danych, 301  
   interfejsy, 319, 320  
   IOException, wyjątek, 329  
   java.util, pakiet, 308  
   klasy, 300, 306, 307  
   klient, 378, 379, 380  
   kolekcje, 308  
   kompilacja bez IDE, 294  
   kontrolki, 334  
   Label, kontrolka, 334  
   LinkedList, klasa, 308  
   List, kontrolka, 334  
   lista, 308, 309  
   log, metoda, 311  
   long, typ danych, 301  
   LookAndFeel, klasa, 350  
   Math, klasa, 310, 311  
   max, metoda, 311  
   min, metoda, 311  
   MouseEvent, klasa, 340  
   MouseListener, interfejs, 342  
   nadklasa, 316  
   native, 307  
   nextToken, metoda, 311  
   notify, metoda, 369  
   NullPointerException, wyjątek, 330  
   obiekt, 298  
   Object, klasa, 300  
   obsługa błędów, 323, 325, 326, 328  
   obsługa pamięci, 295  
   obsługa zdarzeń, 340, 342  
   out, strumień, 304  
   package, deklaracja, 297  
   paint, metoda, 344  
   pakiety, 297  
   panel szklany, 348  
   panel treści, 348  
   panel warstwowy, 348  
   polimorfizm, 317, 319  
   println, metoda, 303, 304  
   random, metoda, 311  
   referencja, 298  
   run, metoda, 362  
   Runnable, interfejs, 362  
   rysowanie, 344  
   Scanner, klasa, 312  
   Scrollbar, kontrolka, 334  
   serwer, 375, 376, 377, 378  
   short, typ danych, 301  
   Stack, klasa, 309  
   start, metoda, 354  
   static, 307  
   stop, metoda, 354  
   stos, 309  
   String, klasa, 300  
   StringTokenizer, klasa, 311  
   strumień wejścia-wyjścia, 303, 304, 305  
   substring, metoda, 300  
   super, słowo kluczowe, 316  
   Swing, pakiet, 347, 348, 350  
   synchronizacja wątków, 366, 367, 369  
   synchronized, słowo kluczowe, 366  
   tablice, 302, 303  
   TextArea, kontrolka, 334  
   TextField, kontrolka, 334  
   Thread, klasa, 362  
   toDegrees, metoda, 311  
   toLowerCase, metoda, 300  
   toRadians, metoda, 311  
   typ napisowy, 300  
   typy danych, 301  
   valueOf, metoda, 307  
   wait, metoda, 369  
   wątki, 361, 362, 363, 369  
   wersje, 291  
   WindowListener, interfejs, 343  
   wyjątki, 326, 328  
   zarządca układu, 336, 337, 339, 340
- JDK, 291  
 język programowania, 39  
 języki interpretowane, 41  
 JRE, 291  
 JVM, 290
- K**  
 keyword, *Patrz* słowo kluczowe  
 klasa, 240  
   bazowa, 270  
   pochodna, 270, 271  
 klient, 374, 378, 379, 380  
 kod  
   bajtowy, 290  
   formatowanie, 45, 109  
   maszynowy, 39  
   źródłowy, 40  
 komentarz, 63  
 kompilacja, 40  
 kompilator, 40  
   dyrektywy, 144, 145  
 komputer, architektura, 36, 37  
 komunikacja sieciowa, 373, 374  
 konsolidacja, 41  
 konstruktor, 245  
   kopiujący, 253  
 Kylix, 57
- L**  
 Lazarus, 57, 58  
 liczba zmiennopezycyjna, 72



liczby pseudolosowe, 99  
 licznik pętli, 93  
 linking, *Patrz* konsolidacja  
 lista jednokierunkowa, 153

**M**

maszyna wirtualna, 290  
 metoda siecznych, 34, 35  
 metoda wirtualna, 275  
 Microsoft Visual Studio, 169, 170,  
 171  
 modelowanie obiektowe, 240  
 monitor, 367

**N**

notacja "O", 31  
 notacja węgierska, 62

**O**

obiekt, 238, 240  
 operator dwuargumentowy, 77

**P**

pamięć operacyjna, 18  
 Pascal  
   :=, operator, 74  
   a C++, 186, 187  
   append, procedura, 137  
   assembler, 147  
   begin, słowo kluczowe, 61  
   błędy, 65, 66  
   boolean, typ danych, 70, 72  
   byte, typ danych, 70  
   char, typ danych, 70  
   chr, funkcja, 70  
   Circle, funkcja, 150  
   close, procedura, 138  
   Cos, funkcja, 99  
   CRT, moduł, 64  
   Dec, procedura, 94  
   deklaracja zmiennej, 74  
   dispose, procedura, 153  
   Division by zero, 66  
   double, typ danych, 70  
   dynamiczny przydział pamięci,  
   152, 153  
   dyrektywy kompilatora, 144, 145  
   end, słowo kluczowe, 61  
   eof, funkcja, 139  
   Exp, funkcja, 99  
   file, typ danych, 135  
   FloodFill, funkcja, 150  
   for, instrukcja, 91, 92, 93  
   Frac, funkcja, 99  
   funkcje, 98, 99, 104  
   grafika BGI, 149, 150  
   Graph, moduł, 149  
   if, instrukcja, 84  
   if-then-else, 87  
   Inc, procedura, 94  
   InitGraph, funkcja, 149  
   Int, funkcja, 99  
   integer, typ danych, 70, 71  
   komentarz, 63, 75  
   Line, funkcja, 150  
   LineTo, funkcja, 150

Linux, 56, 57  
 longint, typ danych, 70, 71  
 MoveTo, funkcja, 150  
 new, procedura, 153  
 operatory arytmetyczne, 77, 78,  
 79  
 operatory logiczne, 80, 82  
 operatory porównania, 80, 81  
 optymalizacja programu, 148  
 ord, funkcja, 71  
 OutTextXY, funkcja, 150  
 pliki, 135, 136, 137, 138, 140,  
 141  
 priorytety operatorów, 83  
 procedury, 61, 98, 101  
 program, instrukcja, 61  
 przekazywanie parametrów przez  
 wartość, 106  
 przekazywanie parametrów przez  
 zmienną, 106  
 przesłanianie zmiennych, 108  
 random, funkcja, 99  
 read, procedura, 140  
 ReadLn, procedura, 64  
 real, typ danych, 70, 72  
 Rectangle, funkcja, 150  
 repeat ... until, instrukcja, 94, 95  
 reset, procedura, 137  
 rewrite, procedura, 137  
 Round, funkcja, 99  
 seek, procedura, 141  
 SetColor, funkcja, 150  
 shortint, typ danych, 70  
 Sin, funkcja, 99  
 single, typ danych, 70  
 słowa kluczowe, 64, 65  
 Sqrt, funkcja, 99  
 struktura programu, 62, 108, 109  
 tablice, 112, 113, 114, 115, 116,  
 152  
 tablice wielowymiarowe, 118  
 typy danych, 69  
 typy okrojone, 113  
 uses, słowo kluczowe, 63, 64  
 while, instrukcja, 94  
 word, typ danych, 70  
 Write, procedura, 61, 80  
 WriteLn, procedura, 61, 80  
 wskaźniki, 153  
 zmienne, 73  
 pętla, 84  
 licznik, 93  
 pliki  
   binarne, 136  
   dostęp sekwencyjny, 140  
   dostęp swobodny, 141  
   tekstowe, 136  
   wykonywalne, 41, 42  
 podprogramy, 97, 98  
 programowanie, 17  
   obiektywne, 238  
   strukturalne, 238  
 programy  
   równoległe, 361  
   sekwencyjne, 360  
 pseudokod, 23

pseudolosowe, liczby, 99  
 punkt przerwania, 47

**R**

RAM, 18  
 rekordy, 130, 131, 132  
 rekurencja, 120, 121, 122, 127, 128  
   kiedy korzystać, 123  
   wady, 127  
 runtime error, *Patrz* błąd  
 uruchomienia

**S**

schemat Hornera, 32, 33  
 schematy blokowe, 24, 25  
 semafor, 367  
 serwer, 374, 375, 376, 377, 378  
 sieciowa, komunikacja, 373, 374  
 siecznych, metoda, 34, 35  
 silna kontrola typów, 69  
 składowe chronione, 271  
 słowo kluczowe, 40  
 sortowanie, 28, 29  
   przez wybór, 29, 30  
   w miejscu, 29  
 Stroustrup, Bjarne, 167  
 struktura danych, 18  
 syntax errors, *Patrz* błąd składniowy  
 system binarny, *Patrz* system  
 dwójkowy  
 system dwójkowy, 38  
 system heksadecymalny, *Patrz* system  
 szesnastkowy  
 system szesnastkowy, 39  
 szyfr Cezara, 224

**T**

Turbo C++, 167, 168  
 Turbo Pascal, 52  
   skrótów klawiszowe, 53  
 typ danych, 68, 69

**U**

UML, 24  
 unboxing, 302

**W**

warunek stopu, 122  
 Wirth, Niklaus, 18  
 wskaźnik pliku, 140, 141  
 wskaźniki, 204  
 wyciek pamięci, 235  
 wykonywalny, plik, 41, 42  
 wyrażenia logiczne, 80, 81, 82  
 wyrażenie arytmetyczne, 77  
 wyszukiwanie, 31

**Z**

zintegrowane środowisko  
 programistyczne, *Patrz* IDE  
 zmienne, 73  
   globalne, 101  
   lokalne, 101  
 zmiennopozycyjna, liczba, 72

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW  
w działający bankomat!

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

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

GRUPA WYDAWNICZA

 **Helion SA**

Podręcznik do nauki zawodu

# TECHNIK INFORMATYK

Wydanie II

## PROGRAMOWANIE STRUKTURALNE I OBIEKTOWE

**Technik informatyk nie jest zwykłym użytkownikiem komputerów. Jeśli uczeń wybiera szkołę o takim profilu, z czasem staje się prawdziwym komputerowym ekspertem.**

Programowanie komputerów osobistych stanowi esencję informatyki. Jest twórcze i wymiennie wpływa na działanie sprzętu, na którym pracujemy – w tym na jego użyteczność oraz efektywność. Bez odpowiedniego oprogramowania niemożliwe jest funkcjonowanie stron internetowych, programów biurowych i bankowych czy nawet sprzętu AGD. Umiejętność programowania w dzisiejszym świecie jest wręcz nieodzowna w zawodzie inżyniera, technika, webmastera czy naukowca.

*Programowanie strukturalne i obiektowe. Podręcznik do nauki zawodu technik informatyk w przystępny sposób wyjaśnia wszystkie niezbędne terminy, tak aby nawet osoba, która nigdy nie miała styczności z programowaniem, mogła je bez przeszkód zrozumieć. Podręcznik został podzielony na części stanowiące trzy oddzielne kursy programowania: w językach Pascal, C++ oraz Java. W czasie realizacji materiału uczniowie poznają najważniejsze elementy tych języków. Każdą część zamyka rozbudowany i ciekawy przykład podsumowujący.*

Do książki została dołączona płyta CD zawierająca ważne i interesujące informacje dodatkowe, które ze względów praktycznych nie mogły ukazać się w wersji drukowanej. Znajduje się tam kurs wprowadzający do programowania w Windows za pomocą WinAPI, omówienie zagadnień pracy grupowej, projektowania oprogramowania i optymalizacji kodu, a także wybrane tematy uzupełniające, m.in. wiadomości dotyczące reprezentacji pozycyjnej liczb.

<http://edukacja.helion.pl>

**helion.pl**  
księgarnia  
internetowa

Nr katalogowy: 8861

Księgarnia internetowa:  
<http://helion.pl>

Zamówienia telefoniczne:  
**0 801 339900**  
**0 601 339900**

**Helion**  
EDUKACJA

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

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



ISBN 978-83-246-3385-2



Informatyka w najlepszym wydaniu