Mirosław J. Kubiak

# Java

## ZADANIA Z PROGRAMOWANIA

Przykładowe

**funkcyjne**

rozwiązania

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

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

ISBN: 978-83-283-8668-6



Printed in Poland.

# Spis treści

Rozdział 3.

# Funkcje w Javie

*W tym rozdziale omawiam mechanizmy znane z paradygmatu programowania funkcyjnego na przykładzie analizy rozwiązanych zadań.*

Od wersji 8 Java zyskała wiele mechanizmów znanych z paradygmatu programowania funkcyjnego. Poznamy teraz niezbędne podstawy i nowości, które weszły wraz z tą nową wersją. Ich rozbudowana postać pojawi się w dalszych rozdziałach tej książki.

## Czyste funkcje

Metody[1] to sposób reprezentacji (do pewnego stopnia) funkcji w języku Java. Aby metody stały się funkcyjne, muszą spełniać następujące zasady obowiązujące **czyste funkcje**:

- Nie mogą modyfikować niczego poza funkcją. Żadne zmiany wewnątrz funkcji nie mogą wyciekać na zewnątrz.
- Nie mogą modyfikować żadnych argumentów.
- Nie mogą rzucać wyjątków lub błędów.
- Muszą zawsze zwracać tę samą wartość.
- Funkcja po wywołaniu z tymi samymi argumentami **musi** zawsze zwrócić ten sam wynik [5].

[1] Zob. rozdział 5. w PI.

Do tej pory korzystaliśmy z funkcji w Javie, być może nie mając tej świadomości. Poniżej znajduje się metoda `dodaj(int liczba1, int liczba2)`, która doskonale pełni rolę funkcji czystej.

Jej postać jest następująca:

```
int dodaj(int liczba1, int liczba2)
    {
        int suma = liczba1 + liczba2;
        return suma;
    }
```

# Funkcje wyższego rzędu

Funkcja wyższego rzędu to szczególny rodzaj funkcji, przyjmujący funkcje jako argumenty i zwracający funkcje.

Zilustrujemy to przykładem zadania 3.1.

**Zadanie**

**3.1** Napisz program, który ilustruje zastosowanie funkcji wyższego rzędu w sortowaniu kolekcji i w którym zastosowano metodę `Collections.sort()`.


Do sortowania użyj interfejsu `Comparator`. Interfejs ten służy do porządkowania obiektów klas zdefiniowanych przez użytkownika. Obiekt porównawczy może porównywać dwa obiekty dwóch różnych klas. Poniższa funkcja porównuje `obj1` z `obj2`.

```
public int compare(Object obj1, Object obj2):
```
[2]

Cała funkcja znajduje się w programie z listingu 3.1.

**Listing 3.1.** *Przykładowe rozwiązanie*

```
// Zadanie 3.1.

package com.mycompany.zadanie_31;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class NewClass
{
```

[2] Zob. w [3] i dalsze zadania w tym rozdziale.

```
    public static void main(String[] args)
    {
        List<String> lista = new ArrayList<>();

       lista.add("Ewelina");
       lista.add("Robert");
       lista.add("Adam");
       lista.add("Małgorzata");

       System.out.println("Lista przed sortowaniem.");

       System.out.println(lista);

       Collections.sort(lista, (String a, String b) -> // Funkcja wyższego rzędu.
       {
           return a.compareTo(b); // Metoda compareTo() porównuje dwa obiekty.
       });

       System.out.println("Lista po sortowaniu.");

       System.out.println(lista);
    }
}
```

Funkcja wyższego rzędu `Collection.sort()` przyjmuje dwa parametry:

```
Collections.sort(lista, (String a, String b)
```

Pierwszy parametr to lista o nazwie `lista`, a drugi to wyrażenie lambda (funkcja):

```
(String a, String b) ->
       {
           return a.compareTo(b); // Metoda compareTo() porównuje dwa obiekty.
       }
```

Parametr lambda sprawia, że funkcja `Collections.sort()` staje się funkcją wyższego rzędu. Metoda `CompareTo()` porównuje ze sobą dwa łańcuchy (dwa obiekty).

Rezultat działania programu można zobaczyć na rysunku 3.1.

**Rysunek 3.1.** *Efekt działania programu Zadanie 3.1*

```
Lista przed sortowaniem.
[Ewelina, Robert, Adam, Małgorzata]

Lista po sortowaniu.
[Adam, Ewelina, Małgorzata, Robert]
```

**Zadanie 3.2** Napisz program, który listę zamieszczoną w zadaniu 3.1 sortuje w odwrotnej kolejności.

**Listing 3.2.** *Przykładowe rozwiązanie*

```
// Zadanie 3.2.

package com.mycompany.zadanie_32;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class NewClass
{
    public static void main(String[] args)
    {
        List<String> lista = new ArrayList<>();

       lista.add("Ewelina");
       lista.add("Robert");
       lista.add("Adam");
       lista.add("Małgorzata");

       System.out.println("Lista przed sortowaniem.");

       System.out.println(lista);

       Collections.sort(lista, (String a, String b) -> // Funkcja wyższego rzędu.
       {
           return b.compareTo(a); // Metoda compareTo() porównuje dwa obiekty.
       });

       System.out.println("Lista po odwrotnym sortowaniu.");

       System.out.println(lista);
    }
}
```

Rezultat działania programu można zobaczyć na rysunku 3.2.

**Rysunek 3.2.** *Efekt działania programu Zadanie 3.2*

```
Lista przed sortowaniem.
[Ewelina, Robert, Adam, Małgorzata]

Lista po odwrotnym sortowaniu.
[Robert, Małgorzata, Ewelina, Adam]
```

# Przykłady zadań wprowadzających do dalszych rozdziałów

Przez przykłady rozwiązanych zadań znajdujących się poniżej zostaną zasygnalizowane następujące zagadnienia:

- klasa wewnętrzna,
- klasa anonimowa,
- wyrażenia lambda,
- referencje do metod,
- strumienie.

Dokładna analiza tych rozwiązanych zadań pozwoli nam łatwiej poruszać się po zagadnieniach zawartych w następnych rozdziałach.

**Zadanie 3.3**

Dana jest lista składająca się z siedmiu różnych imion. Napisz program, który sortuje te imiona w porządku rosnącym.


Skorzystaj z właściwości metody `asList()` klasy `java.util.Arrays`. Metoda ta zwraca listę o stałym rozmiarze, popartą określoną tablicą.

**Listing 3.3.** *Przykładowe rozwiązanie*

```
// Zadanie 3.3.

package com.mycompany.zadanie_33;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class NewClass
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        Collections.sort(imiona); // Sortowanie kolekcji według naturalnego porządku.

        System.out.println(imiona);
    }
}
```


Uwaga

Lista `imiona`

```
List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
        "Bartosz", "Jakub", "Sławomira", "Magdalena");
```

będzie punktem wyjścia do naszych rozważań w najbliższych zadaniach z tego rozdziału.

W programie skorzystaliśmy z klasy `Collections` i z fabrycznej metody `sort()`.

Rezultat działania programu można zobaczyć na rysunku 3.3.

**Rysunek 3.3.** *Efekt działania programu Zadanie 3.3*

[Adam, Bartosz, Ewa, Jakub, Katarzyna, Magdalena, Sławomira]

**Zadanie**

**3.4** Napisz program, który sortuje powyższą listę imion, ale w odwrotnej kolejności. Sposób sortowania określ w osobnej klasie wewnętrznej.

**Listing 3.4.** *Przykładowe rozwiązanie*

```
// Zadanie 3.4.

package com.mycompany.zadanie_34;

import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class przykładKomparatora
{
    static class Komparator implements Comparator<String> // Klasa Komparator.
    {
        @Override // Adnotacja – sprawdza, czy dana metoda przesłania metodę klasy
                  // nadrzędnej.
        public int compare(String o1, String o2)
        {
            return o2.compareTo(o1);
        }
    }
}

public class Main
{
```

```
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        System.out.println("Lista przed sortowaniem.");

        System.out.println(imiona);

        Collections.sort(imiona, new przykładKomparatora.Komparator());
                                                    // Odwrócone sortowanie kolekcji.

        System.out.println("Lista po odwrotnym sortowaniu.");

        System.out.println(imiona);
    }
}
```

Przeanalizujmy poniższe linijki kodu:

```
class przykładKomparatora
{
    static class Komparator implements Comparator<String> // Klasa Komparator.
    {
        @Override // Adnotacja – sprawdza, czy dana metoda przesłania metodę klasy
                  // nadrzędnej.
        public int compare(String o1, String o2)
        {
            return o2.compareTo(o1);
        }
    }
}
```

W powyższym kodzie, we wnętrzu klasy `przykładKomparatora`, zdefiniowaliśmy klasę `Komparator`, tworząc tak zwaną **klasę wewnętrzną**[3], która definiuje warunek sortowania i pozwala nam posortować listę `imiona` w porządku odwrotnym.

Rezultat działania programu można zobaczyć na rysunku 3.4.

**Rysunek 3.4.** *Efekt działania programu Zadanie 3.4*

```
Lista przed sortowaniem.
[Adam, Ewa, Katarzyna, Bartosz, Jakub, Sławomira, Magdalena]
Lista po odwrotnym sortowaniu.
[Sławomira, Magdalena, Katarzyna, Jakub, Ewa, Bartosz, Adam]
```

[3] To jest klasa znajdująca się wewnątrz innej klasy.

**Zadanie**

**3.5** Napisz powyższy program w uproszczonej postaci, korzystając z **klasy anonimowej**[4]. Program robi to samo, co we wcześniejszym zadaniu.

**Listing 3.5.** *Przykładowe rozwiązanie*

```
// Zadanie 3.5.

package com.mycompany.zadanie_35;

import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        Collections.sort(imiona, new Comparator<String>() // Klasa anonimowa.
        {
            @Override
            public int compare(String o1, String o2)
            {
                return o2.compareTo(o1);
            }

        }); // Odwrócone sortowanie kolekcji.

        System.out.println(imiona);
    }
}
```

Rezultat działania programu można zobaczyć na rysunku 3.5.

**Rysunek 3.5.** *Efekt działania programu Zadanie 3.5*

```
[Sławomira, Magdalena, Katarzyna, Jakub, Ewa, Bartosz, Adam]
```

[4] Klasa anonimowa (ang. *nested class*) to **wewnętrzna klasa**, która nie ma nazwy.

**Zadanie**

**3.6** Napisz powyższy program w uproszczonej postaci, korzystając z **wyrażenia lambda**[5]. Program robi to samo, co wcześniej.

**Listing 3.6.** *Przykładowe rozwiązanie*

```
// Zadanie 3.6.

package com.mycompany.zadanie_36;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Main
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        Collections.sort(imiona, (String o1, String o2) ->
        {return o2.compareTo(o1);}); // Wyrażenie lambda.

        System.out.println(imiona);
    }
}
```

Wyrażenie lambda ma postać:

```
Collections.sort(imiona, (String o1, String o2) ->
        {return o2.compareTo(o1);}); // Wyrażenie lambda.
```

Rezultat działania programu można zobaczyć na rysunku 3.6.

**Rysunek 3.6.** *Efekt działania programu Zadanie 3.6*

```
[Sławomira, Magdalena, Katarzyna, Jakub, Ewa, Bartosz, Adam]
```

**Zadanie**

**3.7** Napisz powyższy program w uproszczonej postaci, korzystając z **referencji do metody**[6]. Program robi to samo, co wcześniej, z tą różnicą, że teraz sortowanie listy odbywa się według naturalnego porządku.

[5] Zob. rozdział 5. w PF.

[6] Zob. rozdział 6. w PF.

**Listing 3.7.** *Przykładowe rozwiązanie*

```
// Zadanie 3.7.

package com.mycompany.zadanie_37;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Main
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        Collections.sort(imiona, String::compareTo); // :: Referencja do metody.

        System.out.println("Sortowanie listy według naturalnego
        ↪porządku.");

        System.out.println(imiona);
    }
}
```

W programie wykorzystano referencje do metod statycznych :

```
Collections.sort(imiona, String::compareTo); // :: Referencja do metody.
```

Ogólna postać tej referencji jest następująca:

```
NazwaKlasy::nazwaMetody
```

Podwójny dwukropek zapewnia referencję do metody `compareTo` egzemplarza klasy `String` i jest to referencja typu `Collections`. Wynikiem będzie wydrukowanie każdego elementu posortowanej listy `imiona`.

Rezultat działania programu można zobaczyć na rysunku 3.7.

**Rysunek 3.7.** *Efekt działania programu Zadanie 3.7*

```
Sortowanie listy według naturalnego porządku.
[Adam, Bartosz, Ewa, Jakub, Katarzyna, Magdalena, Sławomira]
```

# Strumienie

Od wersji 8 w Javie pojawiła się biblioteka **strumieni**[7] (ang. *streams*). Umożliwia ona operacje na kolekcjach z wykorzystaniem tak zwanej wewnętrznej iteracji. Strumień to ciąg elementów, jeden po drugim, na których możemy wykonywać różne operacje. Strumień nie modyfikuje źródła danych. W dalszej części tego rozdziału nadal będziemy operować na znanej nam liście `imiona`.

**Zadanie**

**3.8** Napisz program, który korzystając z właściwości rozszerzonej pętli `for`, wyświetla wszystkie elementy listy `imiona`.

**Listing 3.8.** *Przykładowe rozwiązanie*

```
// Zadanie 3.8.

package com.mycompany.zadanie_38;

import java.util.Arrays;
import java.util.List;

public class NewClass
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        for (String im : imiona)
        {
            System.out.println(im);
        }
    }
}
```

Rozszerzona pętla `for` znajduje się poniżej:

```
for (String im : imiona)
        {
            System.out.println(im);
        }
```

Rezultat działania programu można zobaczyć na rysunku 3.8.

[7] Zob. rozdział 6. w PF.

**Rysunek 3.8.** *Efekt działania programu Zadanie 3.8*

```
Adam
Ewa
Katarzyna
Bartosz
Jakub
Sławomira
Magdalena
```

**Zadanie**

**3.9** Napisz program, w którym dzięki właściwościom strumieni zamieniono rozszerzoną pętlę `for` z poprzedniego zadania na strumień `stream`. Należy wyświetlić wszystkie elementy listy `imiona`, korzystając z wyrażenia lambda i referencji do metody. Sama lista `imiona` pozostaje bez zmian.

**Listing 3.9.** *Przykładowe rozwiązanie*

```
// Zadanie 3.9.

package com.mycompany.zadanie_39;

import java.util.Arrays;
import java.util.List;

public class NewClass
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        System.out.println("Wyświetlenie zawartości listy z wykorzystaniem
        ↪wyrażenia lambda.");

        imiona.stream().forEach(s -> System.out.println(s)); // Wyrażenie
                                                             // lambda.

        System.out.println("");

        System.out.println("Wyświetlenie zawartości listy
        ↪z wykorzystaniem referencji do metody.");

        imiona.stream().forEach(System.out::println); // Referencja do metody.
    }
}
```

Dzięki właściwościom strumieni rozszerzona pętla `for` może zostać zamieniona na strumień `stream`, z wykorzystaniem wyrażenia lambda:

```
imiona.stream().forEach(s -> System.out.println(s)); // Wyrażenie lambda.
```

lub referencji do metody:

```
imiona.stream().forEach(System.out::println); // Referencja do metody.
```

Rezultat działania programu można zobaczyć na rysunku 3.9.

**Rysunek 3.9.** *Efekt działania programu Zadanie 3.9*

```
Wyświetlenie zawartości listy z wykorzystaniem wyrażenia lambda.
Adam
Ewa
Katarzyna
Bartosz
Jakub
Sławomira
Magdalena

Wyświetlenie zawartości listy z wykorzystaniem referencji do metody.
Adam
Ewa
Katarzyna
Bartosz
Jakub
Sławomira
Magdalena
```

**Zadanie**

**3.10** Napisz program, który z listy `imiona` wyświetla tylko te imiona, których długość jest większa od czterech znaków.

**Listing 3.10.** *Przykładowe rozwiązanie*

```
// Zadanie 3.10.
package com.mycompany.zadanie_310;

import java.util.Arrays;
import java.util.List;

public class NewClass
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
```

```
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        imiona.stream()
                .filter(s -> s.length() > 4)
                .forEach(System.out::println);
    }
}
```

W programie skorzystano z właściwości metody `filter()`[8] i logicznego warunku `(s -> s.length() > 4)`:

```
filter(s -> s.length() > 4).
```

Metoda `length()` zwraca długość łańcucha (czyli długość naszych imion).

Rezultat działania programu można zobaczyć na rysunku 3.10.

**Rysunek 3.10.** *Efekt działania programu Zadanie 3.10*

```
Katarzyna
Bartosz
Jakub
Sławomira
Magdalena
```

**Zadanie**

**3.11** Napisz program liczący, z ilu imion, których długość jest większa od czterech znaków, składa się lista `imiona`.


Skorzystaj z metod: `peek()` i `count()`.

**Listing 3.11.** *Przykładowe rozwiązanie*

```
// Zadanie 3.11.

package com.mycompany.zadanie_311;

import java.util.Arrays;
import java.util.List;

public class NewClass
{
    public static void main(String[] args)
    {
```

[8] Operacja `filter()` (po polsku *filtr*) zwraca strumień zawierający **tylko** te elementy, dla których zwrócił on wartość `true`. W naszym przypadku zwracane są imiona, których długość jest większa niż cztery znaki.

```
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        long count = imiona.stream()
                .filter(s -> s.length() > 4)
                .peek(System.out::println)
                .count();

        System.out.println("Lista składa się z " + count + " elementów.");
    }
}
```

Metoda `peek()` pozwala na przeprowadzenie operacji na każdym elemencie w strumieniu i zwraca strumień z tymi samymi elementami. Instrukcja: `peek(System.out::println)`

wydrukuje te elementy listy `imiona`, które spełniają zadany warunek. Po jej zakończeniu metoda pozwala dalej na wykonywanie operacji na strumieniu. Natomiast metoda `count()` tylko zwraca liczbę elementów w strumieniu.

Rezultat działania programu można zobaczyć na rysunku 3.11.

**Rysunek 3.11.** *Efekt działania programu Zadanie 3.11*

```
Katarzyna
Bartosz
Jakub
Sławomira
Magdalena
Lista składa się z 5 elementów.
```

Dzięki właściwościom strumieni możemy równie łatwo przekształcić naszą listę w inną listę. Ilustruje to zadanie 3.12.

**Zadanie**

**3.12** Napisz program, który z listy `imiona` tworzy nową listę, `imiona1`, składającą się **tylko** z dużych liter. Dodatkowo nowa lista składa się z imion, których długość jest większa niż cztery litery.

**Listing 3.12.** *Przykładowe rozwiązanie*

```
// Zadanie 3.12.

package com.mycompany.zadanie_312;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class NewClass
{
    public static void main(String[] args)
    {
        List<String> imiona = Arrays.asList("Adam", "Ewa", "Katarzyna",
                "Bartosz", "Jakub", "Sławomira", "Magdalena");

        List<String> imiona1 = imiona.stream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());

        imiona1.stream()
                .filter(s -> s.length() > 4)
                .forEach(System.out::println);
    }
}
```

W programie skorzystano z właściwości operacji `map()` i `collect()` na strumieniu.

Metoda `map()` powoduje, że każdy z elementów może zostać zamieniony na nowy obiekt w nowym strumieniu.

Metoda `String::toUpperCase` zamienia wszystkie znaki w ciągu znaków na duże litery, np. łańcuch `"abc"` zostanie przekonwertowany na łańcuch `"ABC"`.

Metoda `collect()` umożliwia utworzenie nowego obiektu na podstawie istniejących już elementów strumienia.

Klasa `Collectors.toList()` konwertuje wszystkie znaki w metodzie `toList()` klasy `Collectors`. Zwraca interfejs kolektora, który gromadzi dane wejściowe na nowej liście.

Rezultat działania programu można zobaczyć na rysunku 3.12.

**Rysunek 3.12.** *Efekt działania programu Zadanie 3.12*

```
KATARZYNA
BARTOSZ
JAKUB
SŁAWOMIRA
MAGDALENA
```


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

## POZNAJ PODSTAWY PROGRAMOWANIA FUNKCYJNEGO W JAVIE

Java jest obiektowym, bazującym na klasach, współbieżnym językiem programowania. Niezwykłą popularność zawdzięcza on między innymi niezależności od platformy, łatwości pisania w nim programów i klarownemu kodowi. Nic więc dziwnego, że Javę pokochało wielu programistów. To do nich jest skierowana dwuczęściowa publikacja *Java. Zadania z programowania*, której autorem jest Mirosław J. Kubiak.

Jej druga część, zatytułowana *Przykładowe funkcyjne rozwiązania*, jest przeznaczona dla odbiorcy, któremu nieobce są podstawy Javy i który chciałby przyswoić elementy programowania funkcyjnego w tym języku. Ideę paradygmatu funkcyjnego Javy autor omawia na wybranych, czytelnych przykładach. Co ciekawe, wszystkim zawartym tu zadaniom — o różnym stopniu trudności — towarzyszą rozwiązania, których skrupulatne prześledzenie pozwoli Ci w krótkim czasie zapoznać się z podstawami programowania funkcyjnego w Javie. W książce znalazła się niemal setka typowych zadań zilustrowanych nie tylko listingami programów dotyczącymi wybranych zagadnień, lecz także licznymi wskazówkami. Wieńczący całość dodatek zawiera szczegółowe podpowiedzi dotyczące kompilacji dowolnego programu w środowisku Apache NetBeans IDE.

- Rozszerzona pętla for i kolekcje
- Funkcje
- Rekurencja i rekurencja ogonowa
- Wyrażenie lambda i interfejsy funkcyjne
- Strumienie sekwencyjne i równoległe
- Pakiet java.util.function
- Wielowątkowość i równoległość w Javie

**Naucz się programowania funkcyjnego w Javie — na konkretnych przykładach!**

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl



ISBN 978-83-283-8668-6

9 788328 386686



Cena: 49,00 zł