

Mirosław J. Kubiak



# Java

ZADANIA Z PROGRAMOWANIA

Przykładowe

**funkcyjne**

rozwiązania

Helion 

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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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](mailto: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

Copyright © Helion S.A. 2022

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

# Spis treści

	<b>Od autora</b>	<b>5</b>
<b>Rozdział 1.</b>	<b>Programowanie funkcyjne — wprowadzenie</b>	<b>9</b>
	Wstęp	9
	Co to jest paradygmat programowania?	9
	Co to jest programowanie funkcyjne?	10
<b>Rozdział 2.</b>	<b>Rozszerzona pętla for i kolekcje</b>	<b>15</b>
	Rozszerzona pętla for	15
	Kolekcje	18
	Wybrane rodzaje kolekcji	20
	Lista	20
	Zbiór	29
	Mapa	37
	Klasy Vector i Stack	39
<b>Rozdział 3.</b>	<b>Funkcje w Javie</b>	<b>45</b>
	Czyste funkcje	45
	Funkcje wyższego rzędu	46
	Przykłady zadań wprowadzających do dalszych rozdziałów	49
	Strumienie	55
<b>Rozdział 4.</b>	<b>Rekurencja i rekurencja ogonowa</b>	<b>61</b>
	Rekurencja	61
	Rekurencja ogonowa	74
<b>Rozdział 5.</b>	<b>Wyrażenia lambda i interfejsy funkcyjne</b>	<b>83</b>
	Wyrażenie lambda	83
	Interfejs funkcyjny	84

<b>Rozdział 6. Strumienie w Javie</b>	<b>91</b>
Wprowadzenie	91
Przykład porównujący oba paradygmaty: imperatywny i funkcyjny	92
Operacje na strumieniu	94
Proste strumienie	95
Strumienie numeryczne	96
Wykonywanie operacji na strumieniu	101
Metody range() i rangeClosed()	101
Metoda sum()	105
Metoda reduce()	109
Metody split() i count()	110
Rekurencja i rekurencja ogonowa z użyciem strumienia	118
Firma	122
<b>Rozdział 7. Strumienie równoległe w Javie</b>	<b>125</b>
Wprowadzenie	125
Strumienie sekwencyjne vs. równoległe	125
Strumienie równoległe	127
<b>Rozdział 8. Pakiet java.util.function</b>	<b>139</b>
Function	140
Consumer	141
Supplier	143
Predicate	144
<b>Rozdział 9. Wielowątkowość i równoległość w Javie</b>	<b>147</b>
Współbieżność i równoległość	147
Pule wątków	150
Synchronizacja wątków raz jeszcze	153
Użycie blokad	156
Współdziałanie między wątkami	159
Programowanie równoległe	166
<b>Dodatek</b>	<b>175</b>
Tworzenie nowego projektu	175
Uruchomienie naszego programu	178
Wzorzec kodu programu dla programowania obiektowego	181
<b>Bibliografia</b>	<b>183</b>
Bibliografia	183
Darmowe zasoby Internetu	183
Zbiory zadań autora z programowania	183

## 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<sup>1</sup> 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].

---

<sup>1</sup> 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()`.



Wskazówka

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
{
```

<sup>2</sup> 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.



Wskazówka

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 poniŹsze 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 powyŹszym kodzie, we wnętrzu klasy przykŁadKomparatora, zdefiniowaliśmy klasę Komparator, tworząc tak zwaną **klasę wewnętrzną**<sup>3</sup>, 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

<p>Lista przed sortowaniem. [Adam, Ewa, Katarzyna, Bartosz, Jakub, Sławomira, Magdalena]</p> <p>Lista po odwrotnym sortowaniu. [Sławomira, Magdalena, Katarzyna, Jakub, Ewa, Bartosz, Adam]</p>
---

<sup>3</sup> To jest klasa znajdująca się wewnątrz innej klasy.

**Zadanie**

**3.5** Napisz powyższy program w uproszczonej postaci, korzystając z **klasy anonimowej**<sup>4</sup>. 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]

<sup>4</sup> 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**<sup>5</sup>. 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**<sup>6</sup>. Program robi to samo, co wcześniej, z tą różnicą, że teraz sortowanie listy odbywa się według naturalnego porządku.

<sup>5</sup> Zob. rozdział 5. w PF.

<sup>6</sup> 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**<sup>7</sup> (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`, wyświetli 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.

<sup>7</sup> 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 `imi ona`, korzystając z wyrażenia `lambda` i referencji do metody. Sama lista `imi ona` 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()`<sup>8</sup> 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.



Wskazówka

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)
    {

```

<sup>8</sup> 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()` umożliwi 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 obiektywnym, 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](http://helion.pl)

 **HELION SA**  
ul. Kościuszki 1c  
44-100 Gliwice  
tel.: 32 230 98 63  
[helion@helion.pl](mailto:helion@helion.pl)

Sprawdź nasze szkolenia!



AKADEMIA IT & BUSINESS

[HELIONSZKOLENIA.PL](http://HELIONSZKOLENIA.PL)

**KOD KORZYŚCI**  
Sięgnij po więcej! ▶



ISBN 978-83-283-8668-6



9 788328 386686

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 49,00 zł