



# Wstęp do programowania funkcyjnego w Javie



Wojciech Dec 2.04.2025



# Plan prezentacji

---

- Wprowadzenie do programowania funkcyjnego
- Wyrażenia lambda
- Interfejsy funkcyjne
- Strumienie (Stream API)
- Przykłady kodu
- Zadania

# Czym jest programowanie funkcyjne?

---

- Paradygmat programowania, w którym programy są konstruowane za pomocą funkcji.
- Funkcje są obywatelami pierwszej klasy (można je przekazywać jako argumenty, zwracać z funkcji, przypisywać do zmiennych).
- Dlaczego Java 8+?

# Zalety i wady programowania funkcyjnego:

---

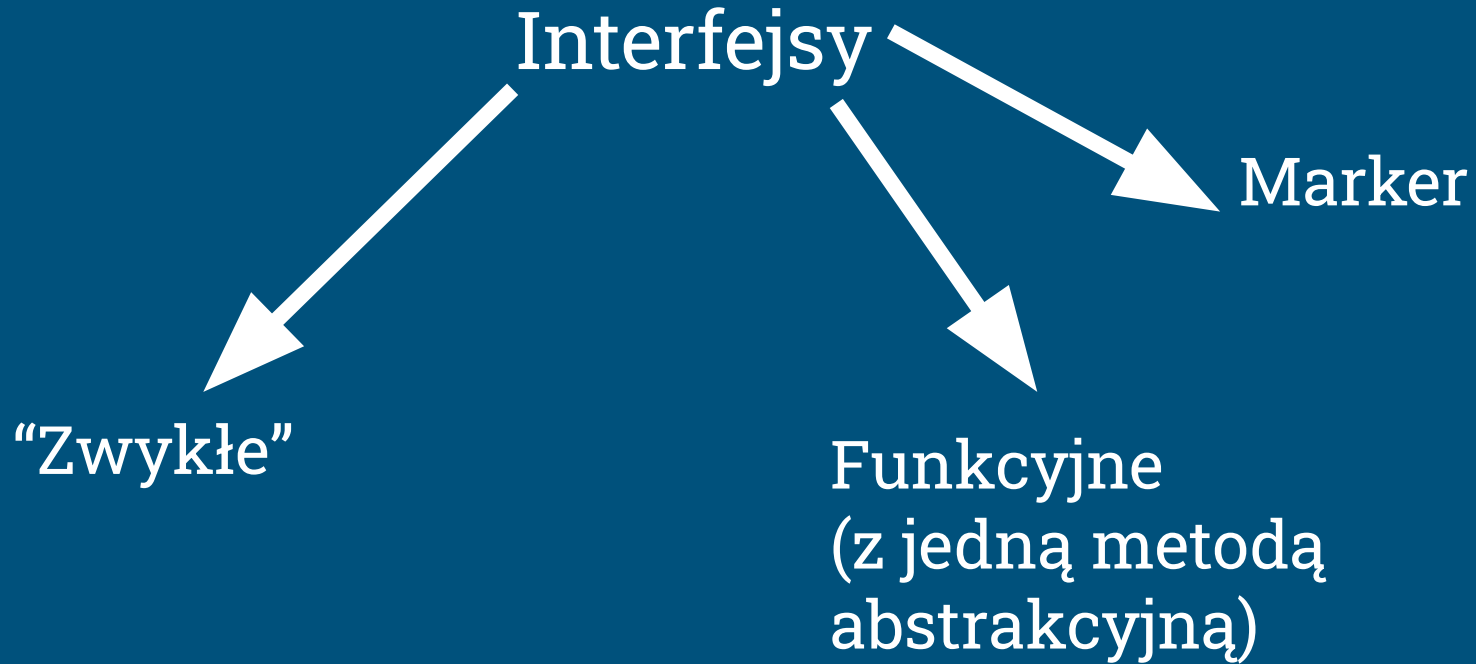
## Zalety

- Krótszy i bardziej czytelny kod (zamiast pętli for - filter/map/reduce)
- Łatwiejsze testowanie i debugowanie.

## Wady

- Mniejsza wydajność w niektórych przypadkach
- Trudniejsze testowanie i debugowanie w niektórych przypadkach

`.map().filter().flatMap()`



```
@FunctionalInterface
interface Calculator {
    // Metoda abstrakcyjna
    int calculate(int a, int b);

    // Metoda default
    default void logResult(int result) {
        System.out.println("Wynik: " + result);
    }

    // Metoda statyczna
    static Calculator getInstance() {
        return (a, b) -> a + b; // Domyślna implementacja
    }
}
```

```
public static void main(String[] args) {  
    Calculator adder = (a, b) -> a + b;  
    int result = adder.calculate(5, 3);  
    adder.logResult(result);  
  
    Calculator multiplier = Calculator.getInstance();  
    System.out.println(multiplier.calculate(2, 4));  
}
```

```
@FunctionalInterface
interface A {
    void show();
}

class B implements A {
    @Override
    public void show() {
        System.out.println("Cześć");
    }
}

public class Demo {
    public static void main(String[] args) {
        A obj = new B();
        obj.show();
    }
}
```



```

@FunctionalInterface
interface A {
    void show();
    String toString();
}

class B implements A {

    @Override
    public void show() {
        System.out.println("Cześć");
    }
}

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

        A obj = new B();
        obj.show();
    }
}

```

Dlaczego interfejs dalej  
jest funkcjonalny jeżeli  
dodana jest metoda  
String toString() ?

## Anonimowa Klasa Wewnętrzna

```
public class Demo {  
    public static void main(String[] args) {  
  
        A obj = new A() {  
            public void show()  
            {  
                System.out.println("Cześć");  
            }  
        };  
        obj.show();  
    }  
}
```

## Wyrażenie Lambda

```
public class Demo {  
    public static void main(String[] args) {  
        A obj = () -> System.out.println("Cześć");  
        obj.show();  
    }  
}
```

# Wyrażenia lambda

---

- W strumieniu: filter(\*lambda\*), mapToInt(\*lambda\*)
- anonimowe metody, które nie są częścią żadnej klasy lub interfejsu
- Zawierają zarówno definicję parametrów jak i ciało funkcji

```
(<lista parametrów>) -> {<ciało wyrażenia>}
```

# Dodatkowe mechanizmy związane z lambdaami

---

Referencje do metod (skrót dla lambda)

Jeśli lambda tylko wywołuje istniejącą metodę, można użyć ::

```
names.forEach(name -> System.out.println(name));  
// Lambda  
names.forEach(System.out::println);
```

# Do czego przydają się wyrażenia lambda?

---

- operacje na kolekcjach
- praca ze strumieniami
- zwiększenie czytelności kodu

```
list.sort(new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.compareTo(s2);  
    }  
});  
  
list.sort((s1, s2) -> s1.compareTo(s2));
```

# Interfejsy funkcyjne

---

Jest to interfejs z **jedną metodą abstrakcyjną** (np. Runnable, Comparator)

Przykłady z java.util.function:

- `Function<T, R>` //  $T \rightarrow R$  (np. `String`  $\rightarrow$  `Integer`)
- `Predicate<T>` //  $T \rightarrow \text{boolean}$
- `Consumer<T>` //  $T \rightarrow \text{void}$
- `Supplier<T>` //  $() \rightarrow T$

# Function<T, R>

---

zawiera metodę **apply**, która przyjmuje instancję klasy T zwracając instancję klasy R

```
Function<String, Integer> stringToLength = s -> s.length();  
int length = stringToLength.apply("Java"); // 4
```



# Consumer<T>

---

- zawiera metodę **accept**, która przyjmuje instancję klasy T
- Nie zwraca żadnej wartości
- Składnia:

```
Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());  
printUpperCase.accept("hello"); // wyświetli "HELLO"
```

# Predicate<T>

---

- zawiera metodę **test**, która przyjmuje instancję klasy T i zwraca flagę.
- Składnia:

```
Predicate<String> isLong = s -> s.length() > 3;  
boolean result = isLong.test("Lambda"); // true
```

# Supplier<T>

---

- zawiera metodę **get**, która nie przyjmuje żadnych parametrów i zwraca instancję klasy T.

```
Supplier<Double> randomValue = () -> Math.random();  
double value = randomValue.get(); // np. 0.42
```

# Przykład

---

```
public static void main(String[] args) {  
  
    List<String> n = Arrays.asList("Paweł", "Grzegorz", "Joanna", "Zofia");  
    Predicate<String> p = (s) -> s.startsWith("G");  
  
    for (String st : n) {  
        if (p.test(st)) {  
            System.out.println(st);  
        }  
    }  
}
```

# Kolekcje

---

- Struktury danych przechowujące wiele elementów.
- W Javie oparte na interfejsie `Collection<E>` z pakietu `java.util`
- `Collection<E>`

`List<E>` – uporządkowana, indeksowana (np. `ArrayList`, `LinkedList`).

`Set<E>` – brak duplikatów (np. `HashSet`, `TreeSet`).

`Queue<E>` – struktura FIFO (np. `PriorityQueue`).

# Stream API

---

- Sekwencja operacji:
  - usuwanie niektórych wartości
  - konwertowanie wartości
  - obliczenia
- Brak zmian w oryginalnym zbiorze danych

# Stream API

---

- Metoda stream()

```
Stream<BoardGame> gamesStream = games.stream();
```

# Sposoby tworzenia strumieni

---

```
Stream<Integer> stream1 = new LinkedList<Integer>().stream();
```

```
Stream<Integer> stream2 = Arrays.stream(new Integer[]{});
```

```
try (Stream<String> lines = new BufferedReader(new FileReader("file.txt")).lines()) {  
    // do something  
}
```



# Stream API

---

## Metody pośrednie:

- `filter()`
- `map()`
- `sorted()`

Zwracają nowy strumień

## Metody terminalne:

- `forEach()`
- `collect()`
- `reduce()`

Kończą strumień

# Przykład

```
public class BoardGame {  
    public final String name;  
    public final double rating;  
    public final BigDecimal price;  
    public final int minPlayers;  
    public final int maxPlayers;  
  
    public BoardGame(String name, double rating, BigDecimal price, int minPlayers, int maxPlayers) {  
        this.name = name;  
        this.rating = rating;  
        this.price = price;  
        this.minPlayers = minPlayers;  
        this.maxPlayers = maxPlayers;  
    }  
}
```

# Przykład

```
List<BoardGame> games = Arrays.asList(  
    new BoardGame("Terraforming Mars", 8.38, new BigDecimal("123.49"), 1, 5),  
    new BoardGame("Codenames", 7.82, new BigDecimal("64.95"), 2, 8),  
    new BoardGame("Puerto Rico", 8.07, new BigDecimal("149.99"), 2, 5),  
    new BoardGame("Terra Mystica", 8.26, new BigDecimal("252.99"), 2, 5),  
    new BoardGame("Scythe", 8.3, new BigDecimal("314.95"), 1, 5),  
    new BoardGame("Power Grid", 7.92, new BigDecimal("145"), 2, 6),  
    new BoardGame("7 Wonders Duel", 8.15, new BigDecimal("109.95"), 2, 2),  
    new BoardGame("Dominion: Intrigue", 7.77, new BigDecimal("159.95"), 2, 4),  
    new BoardGame("Patchwork", 7.77, new BigDecimal("75"), 2, 2),  
    new BoardGame("The Castles of Burgundy", 8.12, new BigDecimal("129.95"), 2, 4)  
);
```

# Porównanie kodu: zagnieżdżone warunki i strumienie

---

- Arrow Anti-Pattern

```
for (BoardGame game : games) {  
    if (game.maxPlayers > 4) {  
        if (game.rating > 8) {  
            if (new BigDecimal(150).compareTo(game.price) > 0) {  
                System.out.println(game.name.toUpperCase());  
            }  
        }  
    }  
}
```

- użycie strumieni

```
games.stream()  
    .filter(g -> g.maxPlayers > 4)  
    .filter(g -> g.rating > 8)  
    .filter(g -> new BigDecimal(150).compareTo(g.price) > 0)  
    .map(g -> g.name.toUpperCase())  
    .forEach(System.out::println);
```

# Filtrowanie

---

zwraca strumień zawierający tylko te elementy dla których filtr zwróci wartość true

`stream().filter()`

```
Stream<T> filter(Predicate<? Isuper T> predicate);
```

Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an **intermediate** operation.

Params: **mapper** – a **non-interfering, stateless** function to apply to each element

Returns: the new stream

# Przykład

---

Java ▾

```
List<Product> products = List.of(  
    new Product("Laptop", 2500.0),  
    new Product("Smartphone", 1200.0),  
    new Product("Mouse", 100.0)  
);  
  
Predicate<Product> isExpensive = product -> product.getPrice() > 1000.0;  
  
List<Product> expensiveProducts = products.stream()  
    .filter(isExpensive)  
    .collect(Collectors.toList());  
  
System.out.println(expensiveProducts);  
// [Product("Laptop", 2500.0), Product("Smartphone", 1200.0)]
```

# Mapowanie

---

- Zmiana typu elementu
- nowy element zawarty jest w nowym strumieniu

# Przykład - map() i interfejs Function

Java ▾

```
List<String> numbersAsStrings = List.of("1", "2", "3");

Function<String, Integer> stringToInt = s -> Integer.parseInt(s);

List<Integer> numbers = numbersAsStrings.stream()
    .map(stringToInt)
    .collect(Collectors.toList());

System.out.println(numbers); // [1, 2, 3]
```



# Redukcja

---

`T reduce(T identity, BinaryOperator<T> accumulator);`

- T identity jest opcjonalne
- Łączenie elementów w wynik: np. obliczając maksimum, minimum, sumę lub iloczyn.

# Przykład redukcji - sumowanie

---

Java ▾

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);  
  
int sum = numbers.stream()  
    .reduce(0, (a, b) -> a + b); // lub .reduce(0, Integer::sum)  
  
System.out.println(sum); // 15
```

# parallelStream

---

```
// Stream sekwencyjny (jednowątkowy)
names.stream().forEach(System.out::println);

// Stream równoległy (wielowątkowy)
names.parallelStream().forEach(System.out::println);
|
```

# Przykłady c.d

---

# Źródła

---

- Java dokumentacja
- Telusko
- Java Podstawy, Cay S. Horstmann
- SamouczekProgramisty.pl
- <https://java-programming.mooc.fi/>
- <https://www.geeksforgeeks.org>