Wprowadzenie do programowania funkcyjnego w Javie: Wyrażenia lambda i strumienie

Wojciech Dec 2.04.2025

Plan prezentacji

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

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 programowania funkcyjnego:

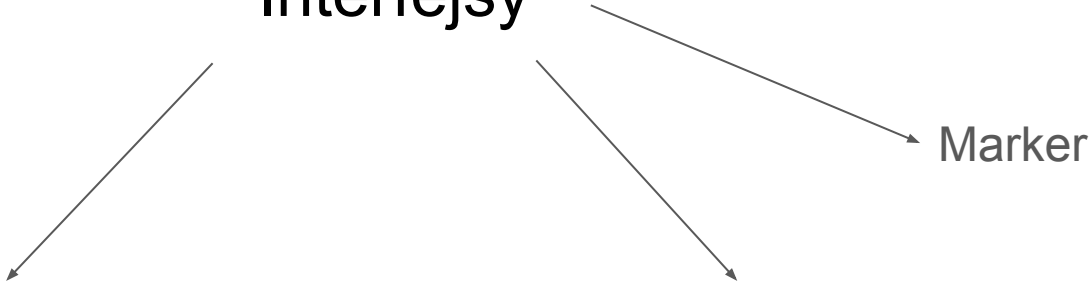
- Krótszy i bardziej czytelny kod (zamiast pętli for - filter/map/reduce)
- Łatwiejsze testowanie i debugowanie.
- Lepsza współbieżność

Do czego przydają się wyrażenia lambda?

- Wyrażenia lambda są bardzo pomocne przy operacji na kolekcjach. Są niezastąpione także przy pracy ze strumieniami. Pozwalają także na pisanie w Javie w sposób “funkcyjny”.

- Oczywistą zaletą wyrażeń lambda jest ich zwięzłość. Kod zajmuje o wiele mniej miejsca, staje się przez to bardziej czytelny.

Interfejsy



Marker

“Zwykłe”

Funkcyjne
(z jedną metodą abstrakcyjną)

```java
@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();
    }
}
```

```java
@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() ?

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

        A obj = new A() {
            public void show()
            {
                System.out.println("Cześć");
            }
        };
        obj.show();
    }
}
```

Anonimowa Klasa Wewnętrzna

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

        A obj = () -> System.out.println("Cześć");
        obj.show();
    }
}
```

Wyrażenie Lambda

Wyrażenia lambda

- Wartości strumienia są przetwarzane przez metody związane z strumieniami, np. filter(), mapToInt()
- Te metody otrzymują funkcję jako parametr
- Wyrażenia lambda to inaczej anonimowe metody, które nie są częścią żadnej klasy lub interfejsu
- Zawierają zarówno definicję parametrów jak i ciało funkcji

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

Interfejsy funkcyjne

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

Przykłady z java.util.function:

- Function<T, R> // T → R (np. String → Integer)
- Predicate<T>   // T → boolean
- Consumer<T>   // T → void
- Supplier<T>   // () → T

Function<T, R>

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

```
Function<Integer, Long> multiline = x -> {
    if (x != null && x % 2 == 0) {
        return (long) x * x;
    }
    else {
        return 123L;
    }
};
```

Consumer<T>

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

```java
Consumer<Integer> consumer = (value) -> System.out.println(value);
```

Predicate<T>

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

```java
Predicate predicate = (value) -> value != null;
```

Supplier<T>

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

```
Supplier<String> someString = () -> "some return value";
```

Przykład 1

```java
// Demonstrate Predicate Interface
import java.util.*;
import java.util.function.Predicate;

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

        List<String> n = Arrays.asList(
            "Paweł", "Grzegorz", "Joanna", "Rafał", "Zofia");

        // declare the predicate type as string and use
        // lambda expression to create object
        Predicate<String> p = (s) -> s.startsWith("G");

        // Iterate through the list
        for (String st : n) {

            // call the test method
            if (p.test(st))
                System.out.println(st);
        }
    }
}
```

Kolekcja

- 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. LinkedList, PriorityQueue).

Stream API

- Strumień jest sposobem na przejście przez kolekcję, gdzie programista określa operację do przeprowadzenia dla każdej wartości
- Sekwencja operacji może zawierać na przykład:
  - usuwanie niektórych wartości
  - konwertowanie wartości
  - obliczenia
- Strumień nie zmienia wartości w oryginalnej kolekcji, tylko je przetwarza

Stream API

Aby utworzyć strumień trzeba użyć metody stream()

Jest to metoda domyślna zaimplementowana w interfejsie Collection.

Pozwala ona na utworzenie strumienia na podstawie danych znajdujących się w danej kolekcji.

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

Sposoby tworzenia strumieni

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

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

```java
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()
```

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

każdy z elementów może zostać zmieniony do innego typu, nowy obiekt 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
- Używa się tej metody powszechnie do agregowania lub łączenia elementów w jeden wynik, np. obliczając maksimum, minimum, sumę lub iloczyn.

Przykład redukcji - sumowanie

Java

```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
```

Aplikacja

Źródła

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