Wykład 13

Programowanie Funkcyjne

Funkcje

Wyrażenia lambda

Strumienie

Programowanie Funkcyjne

Programowanie funkcyjne to paradygmat programowania, który skupia się na wykorzystaniu funkcji. Z tym, że jest to podobne do czysto matematycznej postaci tego stwierdzenia: "*Każdej liczbie z jednego zbioru jest przyporządkowywana dokładnie jedna liczba z jakiegoś innego zbioru*".
Dotychczasowo w nie obiektowych językach programowania z funkcji, nawet dla tych samych argumentów, mogliśmy otrzymać różne wyniki. (np. zwracanie sumy liczby losowej oraz tej będącej argumentem)

Funkcje

Jak to zostało zaimplementowane w javie?

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello");
    }
});
t.start();
```

Tworzony jest nowy wątek (*Thread*) z implementacją interfejsu *Runnable*. Interfejs *Runnable* jest funkcjonalnym interfejsem, który ma tylko jedną metodę, *run()*, która musi być zaimplementowana. Jest on tworzony z pomoca **klasy anonimowej**.

> `Klasa anonimowa` — to specjalny typ klasy zagnieżdżonej, która **nie ma nazwy**. Klasy anonimowe **są używane do tworzenia obiektów na miejscu, gdy potrzebne są tylko raz**. Zwykle są używane podczas deklarowania i inicjalizacji klasy w jednym miejscu

Co się po kolei dzieje?

1. Tworzysz nowy obiekt *Runnable* za pomocą klasy anonimowej. Implementujesz metodę `run()` tej klasy anonimowej, aby wyświetlić "*Hello*".
2. Następnie tworzysz nowy obiekt *Thread*, przekazując do niego twój obiekt *Runnable*.
3. Na końcu uruchamiasz wątek za pomocą metody `start()`. Metoda `start()` w klasie Thread uruchamia nowy wątek i wywołuje metodę `run()` obiektu Runnable.

```
Thread t = new Thread(()->System.out.println("Hello"));
t.start();
```

Jest to równoważny zapis z tym zaprenzetowanym i omówionym powyżej. Jest to tzw. ***Wyrażenie lambda***

`Wyrażenia lambda` — w Javie są metodami, które można przypisać do zmiennej, wywołać lub przekazać jako argument do innej metody. Można je stosować w miejscach, gdzie oczekiwany jest argument typu będącego interfejsem funkcyjnym.

Wyrażenia lambda można używać tylko jako instancje interfejsu funkcyjnego, który posiada jedną metodę!

```
Runnable r = ()->System.out.println("Hello");
Thread t = new Thread(r);
t.start();
```

zapis trzeci jest połączeniem dwóch poprzednich.

Przykłady definicji funkcji:

```
() -> {System.out.println("Hello");}    // java.lang.Runnable
x -> {System.out.println(x);}           // java.util.function.Consumer<T>
() -> 7                                 // java.util.function.Supplier<T>
```

Te nawiasy mozna sobie wyobrazić jako `void funkcja()` czyli funkcja nie przyjmująca żadnych argumentów oraz nic nie zwracająca ale coś robiąca(wyspecyfikowane w nawiasach klamrowych).

Dla tych zdegenrowanych przypadków istnieją typy w javie które je określają.

1. `java.lang.Runnable` — **ponieważ nie przyjmuje żadnych argumentów i nie zwraca wyniku**. *Runnable* jest zaprojektowany do wykonywania działań wyrażonych w kodzie, które mają być wykonywane w wątku.
2. `java.util.function.Consumer<T>` — *Consumer* to operacja, która **przyjmuje jeden argument wejściowy i nie zwraca wyniku**. W tym przypadku *x* jest argumentem wejściowym.
3. `java.util.function.Supplier<T>` — ponieważ **nie przyjmuje żadnych argumentów, ale zwraca wynik**. *Supplier* reprezentuje dostawcę wyników.

Poprawne użycie w kodzie:

```
Runnable r = ()->{System.out.println("Hello");};
Consumer<Integer> cons = x -> {System.out.println(x);};
Supplier<Integer> sup = () -> 7;
```

Tworzenie faktycznych funkcji:

```
Function<Integer, Integer> inc = x -> x+1;
```

`inc` — jest funkcją
`x -> x+1` — wyrażenie lambda

```
BiFunction<Integer, Integer, Integer> sum = (x,y) -> x+y;
```

Można tworzyć też wielo argumentowe funkcje(dwa pierwsze to argumenty, ostatni to wynik)

Nie ma funkcji przyjmującej więcej niż dwa argumenty!

Ale za to każdą funkcję wieloargumentową można zapisać jako złożenie funkcji jednoargumentowych:

```
Function<Integer, Function<Integer, Integer>> sum1 = x->y->x+y;
```

Funkcja może być też argumentem innej funkcji:

```
BiFunction<Function<Integer, Integer>, Integer, Integer> fx = (f, x) -> f.apply(x);
```

Interfejsy `Function` i `BiFunction` posiadają metodę `apply()` i służy ona do wyliczenia wartości funkcji.

> W programowaniu funkcyjnym **funkcje powinny działać zawsze tak samo** i nie powinny powodować żadnych "efektów ubocznych".

Podstawowe zasady:

- wszystkie zmienne są stałe (*final*)
- nie korzystamy zmiennych globalnych
- funkcje mogą być zarówno argumentami innych funkcji oraz mogą być zwracane jako wynik innych funkcji

java.util.function

Wszystkie rzeczy związane z funkcjami są w pakiecie `java.util.function`

Implementacja interfejsu *Function*:

```
Function<T, R>
 R apply(T t);
 Function<T, V> andThen(Function<? super R, ? extends V> after);
 Function<V, R> compose(Function<? super V, ? extends T> before);
```

`T` - typ argumentu wejściowego funkcji
`R` - typ wyniku funkcji

Funkcje `andThen` oraz `compose` służą do składania funkcji (np.f(x) = x+1, g(x) =x^2 f(g(x)) = x^2 + 1)

Implementacja interfejsu *Consumer*:

```
Consumer<T>
 void accept(T t);
 Consumer<T> andThen(Consumer<? super T> after);
```

> Odpowiednikiem `apply()` w Consumerze jest `accept()` — pobiera argument, nic nie zwraca

metoda *accept()* wykonuje funkcję, natomiast *andThen()* zwracaja
funkcję będącą złożeniem dwóch funkcji.

Implementacja interfejsu *Supplier*:

```
Supplier<R>
 R get();
```

> Odpowiednikiem `apply()` w Consumerze jest `get()` — zwraca *R*

Inne **(wybrane)** interfejsy:

Rachunek predykatów (Zdania logiczne)

```
Predicate<T>
 boolean test(T t);
 Predicate<T> and(Predicate<? super T>, other);
 Predicate<T> or(Predicate<? super T>, other);
 Predicate<T> negate();

```

`Predicate` to funkcja, której wynik jest typu boolean

`boolean test(T t)` — Jest używana do testowania, **czy dany argument spełnia pewien warunek**.

`Predicate<T> and(Predicate<? super T> other)` — Ta metoda przyjmuje inny predykat jako argument i zwraca nowy predykat, który jest koniunkcją (AND) obu predykatów. **Nowy predykat zwraca *true*, tylko gdy oba predykaty zwracają *true***.

`Predicate<T> or(Predicate<? super T> other)` — Ta metoda działa podobnie do metody `and`, ale zwraca nowy predykat, który jest alternatywą (OR) obu predykatów. Nowy predykat **zwraca *true*, gdy którykolwiek z predykatów zwraca *true*.**

`Predicate<T> negate()` — Ta metoda zwraca predykat, który jest negacją (NOT) bieżącego predykatu. Nowy predykat **zwraca *true*, gdy bieżący predykat zwraca *false*, i na odwrót**.

```
public interface UnaryOperator<T> extends Function<T,T>
```

`UnaryOperator<T>` — to interfejs funkcyjny w Javie, który rozszerza interfejs *Function<T,T>*. Jest używany, gdy mamy operację jednoargumentową, która zwraca wartość tego samego typu, co jej argument.

Przykładem może być operacja negacji dla liczb całkowitych, gdzie argumentem i wynikiem jest liczba całkowita. Inny przykład to operacja zmiany znaków na łańcuchach znaków, gdzie argumentem i wynikiem jest łańcuch znaków.

Strumienie

Są to inne strumienie niż zaprezentowane wcześniej!

Strumienie te znajdują si w `java.util.stream.Stream` oraz reprezentują strumień danych (obiektów) i są odpowiednikiem kolekcji (java.util.Collection) w programowaniu obiektowym.

Przykład:

```
Stream<String> objectStream = Stream.of("Ala", "Ola");
```

Strumień można też utworzyć z tablicy lub kolekcji:

```
String[] array = {"Ala", "Ola"};
Stream<String> arrayStream = Arrays.stream(array);
            // Analogicznie Collections.stream(array);
```

> W interfejsie *Collection* istnieje metoda `stream()` lub `parallelStream()` przekształcająca kolekcję w strumień.

`parallelStream()` — w Javie jest częścią interfejsu *Collection* i służy do tworzenia strumieni równoległych. Strumień równoległy **pozwala na wykorzystanie wielordzeniowości procesora poprzez równoległe wykonanie operacji strumieniowych** na wielu rdzeniach procesora.(JVM sam optymalizuje i wykorzystuje)
Kolejność wypisywania przy używaniu *parallelStream-ów* może być inna niż oryginalnie!

Przetwarzanie danych w strumieniach opiera się na przekształcaniu strumieni I stosowaniu odpowiednich funkcji

```
Stream.of(1, 2, 3) // tworzymy strumień zawierający 1 2 3
        .map(num -> num * num) // nowy strumień, w którym element
                              // zastępujemy jego drugą potęgą (1 4 9)
        .forEach(System.out::println); // dla każdego elementu strumienia
                                      // wywołujemy funkcję która
                                    // wypisuje element na ekran
```

`map()` — Na każdemu elemencie strumienia na którym jes wywołana, wykonujemy funkcję podaną w argumencie (jako wyrażenie lambda)

`forEach()` — Dla kazdego elementu strumienia wywołaj następującą funkcję podaną jako argument

> `System.out::println` – "*wskaźnik*" do funkcji. Równoważne:

```
num -> System.out.println(num)
```

Kolejny przykład:

```
List<Integer> list1 = Arrays.asList(1, 2, 3);
List<Integer> list2 = Arrays.asList(4, 5, 6);

Stream.of(list1, list2)                 // Stream<List<Integer>>
    .flatMap(List::stream)              // Stream<Integer>
    .filter(num -> num % 2 == 0)        // zostają liczby parzyste
    .forEach(System.out::println);      // 2 4 6
```

mamy dwie listy i tworzymy strumień zawierający te dwie listy.

`flatMap(List::stream)` — *List::stream* tworzy nam dwa osobne strumienie z pierwszej i drugiej listy. A *flatMap()* **łączy nam je w jeden strumień**

`filter()` — Pozostawia w strumieniu **tylko liczby które spełniają podane wyrażenie lambda**

Kolejny przykłady:

```
String sentence = Stream.of("Hello", "world")
            .collect(Collectors.joining(" "));
System.out.println(sentence); // Hello world
```

`collect(Collectors.joining(" "))` — Łączy wszystkie elementy strumienia w jeden ciąg, używając spacji jako separatora. W tym przypadku, tworzy ciąg "*Hello world*".

```
Integer sum = Stream.of(1, 2, 3)
            .reduce(0, Integer::sum);
System.out.println(sum); // 6
```

`reduce(0, Integer::sum)` — Redukuje strumień do pojedynczej wartości, (w tym przypadku) sumując wszystkie elementy. Zaczyna od wartości początkowej *0*, a następnie dodaje do niej każdy element strumienia. W tym przypadku, wynikiem jest suma *1 + 2 + 3*, czyli *6*.
Jeśli byśmy chcieli mnożenie możemy zastosować wyrażenie lambda `(x,y) -> x*y`