Laboratorium 4

Materiały:

ftp://ftp.wsiz.rzeszow.pl/WSPOLNY/LGajecki/SzkolenieTechniczne1

https://gitlab.wsiz.pl/lgajecki/st1

Wyrażenia Lambda. Operacje na kolekcjach

Wyrażenia lambda – służą do zapisania krótkiej, nienazwanej funkcji w miejscu jej wywołania. Stosujemy je kiedy kiedy nie chcemy tworzyć nowych definicji funkcji, które byłyby wywołane przypuszczalnie tylko raz. Stosujemy je często -przypisując jako obiekt interfejsu - do określenia sposobu działania jakiegoś algorytmu.

```
(parametr1, parametr2) -> wyrażenie_zwracane
```

Parametry nie muszą zawierać określenia ich typu (ale mogą -jak w deklaracji funkcji). Głównym założeniem jest prostota zapisu.
wyrażenie_zwracane -to wyrażenie, które można umieścić po słowie return w funkcji. Powyższe wyrażenie możemy zapisać jako funkcję:

```
typ_zwracany fun1(typ1 parametr1, typ2 parametr2)
{
    return wyrażenie_zwracane;
};
```

Inne wersje wyrażenia lambda:

```
parametr1 -> wyrażenie_zwracane

(Typ1 parametr1, Typ2 parametr2) -> wyrażenie_zwracane

(parametr1, parametr2) -> { blok_kodu;
                    return wyrażenie_zwracane;}
```

Mając interfejs:

```
@FunctionalInterface
public interface Operation{
    double operation(double a, double b);
}
```

Przykładowe wyrażenie lambda (odpowiadające funkcji suma) i jego wywołanie:

```
Operation operacja = (a, b) -> a + b;
int wynik = operacja(3, 100);
```

Przykład 1. *Napiszmy funkcję wywołującą operację określoną przez wyrażenie lambda spełniające interfejs Operation*

```
public double calc(double a,double b,Operation op){
    return op.operation(a,b);
}
```

Wywołanie:

```
Operation op= (a,b) -> a+b;
double result = calc(3, 4, op);
System.out.println("Wynik :" +result);
```

Zapisując prościej wyrażenie lambda:

```
double result2 = calc(3, 4, (a,b)->a+b);
```

Odpowiada to klasie implementującej interfejs Operation:

```
class Add implements Operation{

    @Override
    public double operation(double a, double b) {
        return a+b;
    }
}
```

Wywołanie:

```
double result3 = calc(3, 4, new Add());
```

Odpowiada to wywołaniu klasy anonimowej, implementującej Operation:

```
double result4 = calc(3, 4, new Operation(){
    @Override
    public double operation(double a, double b){
        return a+b;
    }
});
```

Możemy zamienić interfejs Operation na interffejs wbudowany Javy (z pakietu java.util.function)

```
public double calc2(double a,double b,DoubleBinaryOperator op){
    return op.applyAsDouble(a,b);
}
```

Wywołanie:

```
double result5 = calc2(3, 4, (a,b)->a+b);
```

Przykład 2. *Na podstawie [1]: Napiszemy funkcję -dla serwisu społecznościowego , która dla osób z kolekcji , wypisze te osoby, które spełniają warunek dotyczący płci i wieku (kobiety w wieku 18-25 lat).*

a) Funkcja może mieć w sobie zaimplementowany ten warunek . Jednak aby funkcja była jak najbardziej uniwersalna, to można ten warunek przekazać z zewnątrz np. wyrażeniem lambda

```
public static void wypiszOsoby(
    List<Osoba> lista, SprawdzOsoba tester) {
    for (Osoba o : lista) {
        if (tester.test(o)) {
            o.wypisz();
        }
    }
}
```

Interfejs:

```
public interface SprawdzOsoba {
    boolean test(Osoba o);
}
```

Wywołanie:

```
wypiszOsoby(lista_osob, (Osoba o) -> o.getPlec() == Osoba.Plec.K
            && o.wiek() >= 18
            && o.wiek() <= 25);
```

b) Zastosujmy wbudowany interfejs Predicate<T>, który posiada metodę: boolean test(T object)

```
    public static void wypiszOsobyPredicate(
        List<Osoba> lista, Predicate<Osoba> tester) {
        for (Osoba o : lista) {
            if (tester.test(o)) {
                o.wypisz();
            }
        }
    }
```

Wywołanie:

```
wypiszOsobyPredicate(lista_osob, (Osoba o) -> o.getPlec() == Osoba.Plec.K
            && o.wiek() >= 18
            && o.wiek() <= 25);
```

c) Skorzystajmy z kolejnego interfejsu celem określenia akcji do wykonania dla każdego obiektu – interfejs Consumer<T> z metodą void accept(T o)

```
    public static void przetwarzajOsoby(
        List<Osoba> lista,
        Predicate<Osoba> tester,
        Consumer<Osoba> block) {
            for (Osoba o : lista) {
                if (tester.test(o)) {
                    block.accept(o);
                }
            }
    }
```

Wówczas działanie wpiszemy wyrażeniem lambda:

```
    przetwarzajOsoby(lista_osob, o -> o.getPlec() == Osoba.Plec.K
            && o.wiek() >= 18
            && o.wiek() <= 25,
        o -> o.wypisz());
```

d) Określmy jeszcze przekształcenie danych pochodzących z kolekcji na typ potrzebny do akcj block. Interfejs Function<X, Y> z metodą: Y mapper.apply(X o);

```
    public static void przetwarzajOsobyzFunkcja(
        List<Osoba> lista,
        Predicate<Osoba> tester,
        Function<Osoba, String> mapper,
        Consumer<String> block) {
            for (Osoba o : lista) {
                if (tester.test(o)) {
                    String dane=mapper.apply(o);
                    block.accept(dane);
                }
            }
    }
```

Wywołanie:

```
        przetwarzajOsobyzFunkcja(lista_osob, o -> o.getPlec() == Osoba.Plec.K
            && o.wiek() >= 18
```

```
            && o.wiek() <= 25,
        o -> o.getNazwisko(),
        nazw -> System.out.println(nazw));
```

e) Zastosujmy typy generyczne intensywniej (zamiana listy na interfejs Iterable<T>, którego implementuje ArrayList,TreeSet,...)

```
    public static <X, Y> void przetwarzajElementy(
        Iterable<X> lista,
        Predicate<X> tester,
        Function<X, Y> mapper,
        Consumer<Y> block) {
            for (X o : lista) {
                if (tester.test(o)) {
                    Y dane=mapper.apply(o);
                    block.accept(dane);
                }
            }
    }
```

Wywołanie:

```
        przetwarzajElementy(lista_osob, o -> o.getPlec() == Osoba.Plec.K
            && o.wiek() >= 18
            && o.wiek() <= 25,
        o -> o.getNazwisko(),
        nazw -> System.out.println(nazw));
        System.out.println();
```

f) Wykorzystajmy operacje agregujące (operacje strumienia)

```
        lista_osob
            .stream()
            .filter(
                o -> o.getPlec() == Osoba.Plec.K
                    && o.wiek() >= 18
                    && o.wiek() <= 25)
            .map(o -> o.getNazwisko())
            .forEach(nazw -> System.out.println(nazw));
```

Uzyskujemy strumień danych (Stream) z kolekcji. W strumieniu mamy np. operacje pośrednie (filter, map,sorted,distinct), jak i operacje finalne (reduce, find). Api metod strumienia spełnia założenia FluentApi (m.in. łatwość zapisu, łańcuch operacji) [4]

Przykład 3. *Utwórz listę liczb całkowitych, wybierz elementy większe niż 7 i posortuj je.*

Liczby większe niż 7. Metoda collect służy do zamiany strumienia w listę/

```
        ArrayList<Integer> lista1=new ArrayList<>(Arrays.asList(1, 7, 5, 2, 150,
10, 100));
        List<Integer> wynik_lista=lista1.stream()
                .filter( el -> el >=7)
                .collect(Collectors.toList());
```

Dodatkowo posortowane

```
        lista1.stream()
                .filter( el -> el >=7)
                .sorted((a,b)-> - a.compareTo(b))
                .forEach(el -> System.out.print(el + ", "));
        System.out.println();
```

Przykład 4. *Z listy osób wypisz nazwiska i imiona kobiet.*
Wykorzystamy operację map -do zamiany typu i postaci danych, w forEach wykorzystamy referencję do metody

```
        lista_osob
            .stream()
            .filter(
                o -> o.getNazwisko().startsWith("K"))
            .map(o -> o.getImie()+" "+o.getNazwisko())
            .forEach(System.out::println);
```


1. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
2. https://www.baeldung.com/java-8-lambda-expressions-tips
3. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html
4. http://mw.home.amu.edu.pl/zajecia/PRA2020/PRA03ENG.html
5. https://www.baeldung.com/java-8-streams-introduction

Zadania:

1. Napisz dowolny interfejs i do obiektu interfejsu przypisz wyrażenie lambda, a później je wywołaj.

2. Kolekcja zawierająca liczby - wypisz elementy parzyste

3. Kolekcja zawierająca liczby – wypisz najmniejszą wartość spośród elementów parzystych

4. Kolekcja zawierająca liczby - wypisz elementy z podanego zakresu

5. Kolekcja osoby - wypisz osoby urodzone po podanym roku

6. Kolekcja osoby - wypisz osoby, których imię zaczyna się od podanego ciągu (np.2 litery), posortuj w kolejności alfabetycznej po nazwisku

7. Jak -wyżej -ile jest takich osób

8*. Napisz funkcję - algorytm, który jako dane wejściowe przyjmuje dwie kolekcje, zwraca kolekcję wynikową. Dla elementu o indeksie i kolekcji 1 i kolekcji 2 generuje element o indeksie i kolekcji wynikowej. Sposób wyliczenia wyniku określony jest przez wyrażenie lambda / obiekt interfejsu.