Programowanie funkcyjne

## Zasady

1. Nasze klasy powinny być immutable czyli dajemy **final** (nie będzie więc dziedziczenia).
2. Pola składowe klasy są **inicjalizowane w konstruktorze.**
3. **Nie** tworzymy **seterów**.
4. Aby przeczytać wartości pól składowych tworzymy **getery.**
5. Metody są czyste. Nie modyfikują stanów innych obiektów.
6. Jeśli modyfikujemy jednak obiekt, to zwracamy już inny obiekt, jego kopię.
7. Unikamy nulli. Lepiej zwrócić obiekt klasy Optional<Klasa> niż null.
8. Metody są tak samo ważne jak klasy. Mogą być parametrami innych metod, a także mogą być przypisywane do zmiennych kokalnych.

## Co to jest interfejs funkcyjny?

To taki interfejs, który posiada **tylko jedną metodę**

Przykłady:

1. Metoda run() z interfejsu Runnable
2. Metoda compareTo() z interfejsu Comparable

Interfejsy funkcyjne najlepiej oznaczać adnotacją **@FunctionalInterface.**

Java 1.8 to kilkanaście nowych interfejsów, właśnie funkcyjnych

**Wyrażenie lambda – od Java 1.8**

Są to anonimowe metody, bo nie mają nazwy.

Mogą zastąpić każdy interfejs funkcyjny

(String s)->{sout ( c );};

Runnable r=()-> sout ( a);

Jeśli metoda coś zwraca, to i tak tego nie piszemy z prawej strony, bo domyślnie jest return.

Nie trzeba nawet pisać jak jest typ zmiennej z lewej. Java powinna się domyślić ;)

## Klasa Student i klasa Indeks

final public class Indeks {  
 private String indexNumber;  
  
 public Indeks(String indexNumber) {  
 this.indexNumber = indexNumber;  
 }  
  
 public String getIndexNumber() {  
 return indexNumber;  
 }  
}

final public class Student {  
 private String name;  
 private int age;  
 private Indeks index;  
  
 public Student(String name, int age,String indexNumber) {  
 this.name = name;  
 this.age = age;  
 this.index = new Indeks(indexNumber);  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public Indeks getIndex() {  
 return index;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 ", index=" + index +  
 '}';  
 }  
 public Student changeIndexNumber(String indexNumber){  
 return new Student("Paweł",39,indexNumber);  
 }  
}

## Interfejs Predicate i metoda test(T)

1. Pomaga przy sekwencyjnym przetwarzaniu danych.
2. Na podstawie obiektu zwraca nam Booleana.

Klasa App

public class App {  
 public static List<Student> createData(){   
 List<Student> result=new ArrayList<>(  
 Arrays.*asList*(  
 new Student("Paweł",38,"123"),  
 new Student("Jacek",34,"345"),  
 new Student("Kasia",26,"341"),  
 new Student("Tomasz",39,"3145")  
 )  
 );  
 return result;  
 }

}

public static void main(String[] args) {  
 List<Student> students=*createData*();  
 Predicate<Student> kasiaPredicate=new Predicate<Student>() {  
 @Override  
 public boolean test(Student student) {  
 return student.getName().equals("Kasia");  
 }  
 };  
 System.*out*.println(students);  
}

W metodzie main tworzymy listę studentów i predykat kasiaPredicate, który pomoże nam wyszukać studentów o imieniu Kasia. Ten zapis można zastąpić następującym wyrażeniem lambda

Predicate<Student> kasiaPredicate=student -> student.getName().equals("Kasia");

A więc będzie

public static void main(String[] args) {  
 List<Student> students=*createData*();  
 Predicate<Student> kasiaPredicate=student -> student.getName().equals("Kasia");  
 System.*out*.println(students);  
}

Napiszemy metodę, która dla listy studentów do niej przekazanej zwraca listę studentów spełniających test zawarty w predykacie.

public static List<Student> getStudents(List<Student> list,Predicate<Student> predicate){  
 List<Student> result=new ArrayList<>();  
 for (Student student:list  
 ) {  
 if (predicate.test(student))result.add(student);  
 }  
 return result;  
}

Obiekt spełnia test predykatu jeśli zostanie zaakceptowany przez metodę test(T)

Teraz wystarczy uruchomić tę metodę, aby otrzymać listę studentów, mających na imię Kasia.

System.*out*.println(*getStudents*(students,kasiaPredicate));

Jakie operacje można wykonywać na predykatach?

And, Or, negate

Predicate<Student> equals38andKasia=kasiaPredicate.and(equals38Predicate);  
System.*out*.println(*getStudents*(students,equals38andKasia));

Wybierze nam studentów, którzy spełniają jednocześnie dwa warunki

## Interfejs Consumer i metoda accept()

Na podstawie obiektu wykonuje jakąś operację dzięki metodzie accept() ale nic nie zwraca. Np. coś wyświetla.

Napiszemy teraz consumera, który na podstawie obiektu Student nic nie zwraca, ale wyświetla imię studenta.

Consumer<Student> studentNameConsumer=new Consumer<Student>() {  
 @Override  
 public void accept(Student student) {  
 System.*out*.println(student.getName());  
 }  
};

Można uprościć do wyrażenia lambda

Consumer<Student> print= student -> System.*out*.println(student.getName());

I wyświetlić (czyli skonsumować studentów)

public static void consumeStudents(List<Student>list,Consumer<Student>consumer){  
 for (Student student:list  
 ) {  
 consumer.accept(student);  
 }  
}

*consumeStudents*(students,print);

A jak zrobić skonsumowanie (wyświetlenie studentów po przefiltrowaniu?)

*consumeStudents*(*getStudents*(students,kasiaPredicate),print);

i wszystko jasne.

Jak się łaczy consumerów? Metodą andThen(C)

Consumer<Student> printName= student -> System.*out*.println(student.getName());  
Consumer<Student> printAge=student -> System.*out*.println(student.getAge());  
Consumer<Student> two=printName.andThen(printAge);  
*consumeStudents*(*getStudents*(students,kasiaPredicate),two);

Należy pamiętać, że jeśli nie powiedzie się wykonywanie pierwszego consumera, to i drugi będzie zatrzymany.

## Interfejs Supplier

Nie bierze żadnych argumentów, a zwraca obiekt czyli odwrotnie jak w przypadku Consumera.

Napiszemy teraz suppliera, aby zwracał nam całą listę studentów.

Supplier<List<Student>> supplyStudentList=()->*createData*();

Albo jako zapis funkcyjny

Supplier<List<Student>> supplyStudentList= App::*createData*;

Teraz użyjemy suppliera razem z consumerem i predicate.

*consumeStudents*(*getStudents*(supplyStudentList,kasiaPredicate),two);

oczywiście aby to zadziałało należy trochę zmienić metodę getStudents()

public static List<Student> getStudents(Supplier<List<Student>> supplier,Predicate<Student> predicate){  
 List<Student> result=new ArrayList<>();  
 List<Student> students=supplier.get();  
 for (Student student:students  
 ) {  
 if (predicate.test(student))result.add(student);  
 }  
 return result;  
}

## Interfejs Function i metoda apply()

Bierze jeden typ obiektu, modyfikuje i zwraca inny obiekt.

Zrobimy teraz Function, które będzie brało studenta i zwracało jego imię.

Function<Student,String> functionStudentName=student -> student.getName();

Albo prościej

Function<Student,String> functionStudentName= Student::getName;

Dodamy teraz funkcję do trzech pozostałych interfejsów.

Wcześniej musimy jednak zmienić consumera

Consumer<String> printString= string-> System.*out*.println(string);

*consumeStudents*(*getStudents*(supplyStudentList,kasiaPredicate),functionStudentName,printString);

public static void consumeStudents(List<Student>list,Function<Student,String> function,Consumer<String>consumer){  
 for (Student student:list  
 ) {  
 consumer.accept(function.apply(student));  
 }  
}

i teraz będzie działać

## Interfejs BiFunction

Bierze dwa obiekty, a zwraca trzy obiekty.

## Interfejs BinaryOperator

Na podstawie dwóch obiektów tworzy jeden.

## Warianty prymitywne interfejsów funkcyjnych

IntPredicate, na podstawie int zwraca Booleana

DoublePredicate, LongPredicate itd.

Podobnie jest z Consumerami i Supplier.

Jest też np. Klasa ToIntFunction = pobiera jakiś obiekt i zwraca inta. Są odpowiedniki dla innych typów prymitywnych.

## Method references

To zapis np.:

Function<Student,String> functionStudentName= Student::getName;

Gdzie App jest naszą klasą.

Można więc powiedzieć, że jest to skrócenie zapisu lambdy.

## Optional i metoda get()

Opakowuje inny obiekt i mówi o nim, że może być lub nie być.

Chodzi o to, by uniknąć zwracania nulli.

1 sposób, gdy nie jesteśmy pewni, czy nie będzie nullem

public Optional<Indeks> getIndex() {  
 return Optional.*ofNullable*(index);  
}

2 sposób, gdy jesteśmy pewni, że nie będzie nullem

public Optional<Indeks> getIndex() {  
 return Optional.*of*(index);  
}

3 sposób, gdy zwracamy pustą wartość

public Optional<Indeks> getIndex() {  
 return Optional.*empty*();  
}

Jak korzystać z Optional?

Optional<Indeks> indeks=supplyStudentList.get().get(0).getIndex();

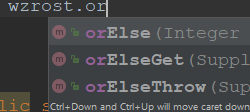
1 sposób

if (indeks.isPresent()){  
 System.*out*.println(indeks.get().getIndexNumber());  
}

jeśli wywołamy geta na pustym Optional dostaniemy wyjątek

2 sposób (tutaj będzie consumer)

indeks.ifPresent(i -> System.*out*.println(i));



3 sposób (filtrowanie)

indeks.filter(i->i.getIndexNumber().equals("123")).ifPresent(i-> System.*out*.println(i.getIndexNumber()));

4 sposób (mapowanie)

indeks.map(i->i.getIndexNumber()).filter(number->number.equals("123")).ifPresent(index-> System.*out*.println(index));

## Stream API

To zestaw metod do strumieniowego przetwarzania danych.

Po streamie możemy przejechać się tylko raz czyli inaczej niż w kolekcjach.

Przykład

supplyStudentList.get().stream().filter(kasiaPredicate).map(Student::getName).forEach(printString);

## Generowanie wartości dla strumieni

1 sposób (predefiniowane wartości)

Stream.*of*("A","B","C").forEach(printString);

2 sposób (kolekcje)

List<Student> studentList1=*createData*();  
studentList1.stream().filter(kasiaPredicate).forEach(printName);

3 sposób (kolekcje)

Stream.*generate*(supplyStudentList).forEach(print);

Tylko, ze trzeba napisać jeszcze consumera dla listy studentów

4 sposób (Supplier)

Stream.*generate*(()->Math.*random*()).limit(10).forEach(System.*out*::println);

5 sposób (iterate, wyświetlenie 20 liczb parzystych, począwszy od 0)

Stream.*iterate*(0,i->i+2).limit(20).forEach(System.*out*::println);

6 sposób (liczby podzielne przez 2)

IntStream.*rangeClosed*(5,100).filter(i->i%2==0).forEach(System.*out*::println);

## Filtr

Filter to operacja pośrednia na strumieniach. Filter przyjmuje jako argument Predicate.

public static Stream<Student> createDataStream(){  
 Student student1=new Student("Paweł",38,"123");  
 Student student2=new Student("Jacek",34,"345");  
 Student student3=new Student("Kasia",38,"341");  
 Student student4=new Student("Tomasz",39,"3145");  
 return Stream.*of*(student1,student2,student3,student4);  
}

oraz metoda main()

public static void main(String[] args) {  
 Predicate<Student> over30Predicate=student -> student.getAge()>30;  
 Consumer<String> println= System.*out*::println;  
 Function<Student,String> getStudentName=Student::getName;  
  
 *createDataStream*().filter(over30Predicate).map(getStudentName).forEach(println);  
}

Wyświetlamy więc imiona studentów powyżej 30 lat.

## Map

Map to operacja pośrednia na strumieniach. Map przyjmuje jako argument Function.

Map i Filter można stosować wiele razy i na przemian.

## ForEach

ForEach to metoda terminalna, która kończy używanie strumienia.

## FindFirst, AnyMatch,AllMatch,NoneMatch

Wszystkie trzy to metody terminalne, które po wykonaniu swoich zadań zamykają strumień.

Wyświetlimy imię pierwszego studenta, którego wiek przekracza 30 lat

*createDataStream*().filter(over30Predicate).map(getStudentName).findFirst().ifPresent(System.*out*::println);

**AnyMatch** – czy dowolny obiekt w strumieniu spełnia Predicate

Czy jest jakiś student o imieniu Paweł?

System.*out*.println(*createDataStream*().map(getStudentName).anyMatch(s -> s.equals("Paweł")));

AllMatch – czy wszystkie obiekty spełniają określony warunek

Czy wszystkie imiona są palindromami

System.*out*.println(*createDataStream*().map(getStudentName).allMatch(s->new StringBuilder(s).reverse().equals(s)));

Czy wszystkie wyrazy znajdujące się w pliku są palindromami?

Files.*readAllLines*(Paths.*get*("palindrom.txt")).stream().map(String::toLowerCase).allMatch(s->new StringBuilder(s).reverse().equals(s));

## Reduce

Metoda terminalna. Redukuje strumień do jednej wartości.

Można użyć do wyszukiwania min,max czy do łączenia stringów.

Sumowanie dziesięciu liczb losowych (użycie jako drugiego parametru new BinaryOperator i przekształcić na lambdę)

System.*out*.println(Stream.*generate*(Math::*random*).limit(10).reduce(0.0, (aDouble, aDouble2) -> aDouble+aDouble2));

Jak znaleźć najstarszego studenta?

*createDataStream*().map(s->s.getAge()).max(Comparator.*naturalOrder*()).ifPresent(System.*out*::println);

lub

*createDataStream*().map(Student::getAge).reduce(Integer::*max*).ifPresent(System.*out*::println);

## Collect

Metoda terminalna, jest to specjalny typ reduce, który pozwala nam np. na zebranie wszystkich elementów w listę.

Jak uzyskać listę wieku studentów?

System.*out*.println(*createDataStream*().map(Student::getAge).collect(Collectors.*toList*()));

Counting – zwraca liczbę elementów.

Jak połączyć wszystkie elementy ze strumienia do jednego stringa z separatorem „ , ” ?

*createDataStream*().map(Student::getAge).map(s->s.toString()).collect(Collectors.*joining*(","))

Jak utworzyć mapę, klucz = age, wartość = liczba studentów w danym wieku

Map<Integer,List<Student>> list=*createDataStream*().collect(Collectors.*groupingBy*(Student::getAge));  
System.*out*.println(list);

## Limit, skip, distinct,sorted, count

Limit – ograniczenie do liczby elementów

Skip – pomija określoną liczbę elementów

Distnict – bierzemy pod uwagę tylko różne obiekty (hashcode i equals)

Sorted() – sortowanie wg naturalnego porządku lub new Comparator w nawiasie

Count – oblicza ilość elementów w strumieniu (reduktor)

## Strumienie typów prymitywnych

Zaleta?

Szybciej pracuje się na strumieniach prymitywnych

IntStream intStream=*createDataStream*().map(Student::getAge).mapToInt(value -> value.intValue());

Może być jeszcze tylko Long i Double

Wada?

Od teraz wszędzie w parametrach trzeba podawać Integerowe odmiany interfejsów funkcyjnych.

## Modułowe SDK

Wszystkie moduły dziedziczą po module java.base