Programowanie funkcyjne

Programowanie funkcyjne jest możliwe dzięki interfejsom funkcjonalnym. Posiadają one dokładnie jedną metodę abstrakcyjną. Najbardziej przydatne takie interfejsy znajdują się w pakiecie **java.io.function.** Najczęściej używane interfejsy to **Predicate**, **Consumer** oraz **Supplier.** Przed programowaniem funkcyjnym programiści korzystali często z biblioteki **Guava**.

Interfejs Predicate<T> i metoda test(T)

Interfejs **Predicate<T>** pomaga przy sekwencyjnym przetwarzaniu danych. Na podstawie przekazanego do metody **test(T)** obiektu zwraca booleana. Inaczej mówiąc, Predicate służy do sprawdzenia czy przekazany obiekt spełnia jakiś warunek.

Wykorzystamy interfejs **Predicate** aby wyświetlić studentów powyżej 30 na podanej liście studentów.

https://github.com/idzikpro/JavaBasics/blob/master/src/main/java/pl/idzikpro/function/PredicateMain.java

```
List<Student> studentList = Arrays.asList(
        new Student("Michał",32),
        new Student("Kasia",31),
        new Student("Jacek",30)
);
Predicate<Student> predicateAgeOver30 = new Predicate<Student>() {
    @Override
    public boolean test(Student student) {
        return student.getAge()>30;
    }
};
for (Student student:studentList
     ) {
    if (predicateAgeOver30.test(student)){
        System.out.println(student);
    }
}
```

Widzimy utworzoną listę studentów oraz predicate predicateAgeOver30, który jak sama nazwa zmiennej mówi będzie służył do sprawdzania wieku studentów. Wykorzystując pętle foreach możemy w końcu znaleźć studentów, których wiek jest większy o 30.

Predicate można uprościć korzystając z wyrażenia lambda do następującej postaci.

```
Predicate<Student> predicateAgeOver30 = student -> student.getAge()>30;
```

Na predykatach można stosować operacje **and, or, negate**. Posługujemy się nimi podobnie jak we wzorcu projektowym Builder.

Interfejs Consumer<T> i metoda accept(T)

Interfejs **Consumer<T>** wykonuje jakąś operację na obiekcie T ale nic nie zwraca. Operacja wykonywana jest w metodzie accept(T) ale nic nie zwraca. Często stosujemy go do wyświetlania obiektów.

Wykorzystamy interfejs **Consumer** do wyświetlenie nazwisk studentów.

https://github.com/idzikpro/JavaBasics/blob/master/src/main/java/pl/idzikpro/function/ConsumerMain.java

```
List<Student> studentList = Arrays.asList(
        new Student("Michał",32),
        new Student("Kasia",31),
        new Student("Jacek",30)
);
Consumer<Student> consumerName= student -> System.out.println(student.getName());
for (Student student:studentList
) {
    consumerName.accept(student);
}
```

Wywołania Consumera można łączyć stosując metodę **andThen()**. Robimy to w taki sposób jak we wzorcu Builder. Należy pamiętać, że jeśli nie powiedzie się wykonywanie pierwszego consumera, to i drugi będzie zatrzymany.

Interfejs Supplier <T> i metoda T get()

Metoda get nie pobiera żadnych argumentów, a zwraca obiekt typu T czyli odwrotnie jak w przypadku Consumera.

Wykorzystamy **Supplier** do tego, aby zwrócił nam „losowego" studenta.

https://github.com/idzikpro/JavaBasics/blob/master/src/main/java/pl/idzikpro/function/SupplierMain.java

```
List<String> stringList= Arrays.asList("Kasia","Jacek","Monika");
Random random=new Random();
Supplier<Student> supplierStudent= () -> new Student(
        stringList.get(random.nextInt(3)),
        random.nextInt(40)+20);
System.out.println(supplierStudent.get());
```

Interfejs Function<T,R> i metoda R apply(T)

Metoda apply pobiera obiekt typu T, a zwraca obiekt typu R.

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

https://github.com/idzikpro/JavaBasics/blob/master/src/main/java/pl/idzikpro/function/FunctionMain.java

```
Student student=new Student("Kasia",22);
Function<Student,String> functionStudentName= student1 -> student1.getName();
System.out.println(functionStudentName.apply(student));
```

Interfejs BiFunction<T,U,R> i metoda R apply(T)

Metoda apply pobiera dwa typy T i U, a zwraca obiekt klasy R.

Wykorzystamy interfejs **BiFunction** do utworzenia obiektu Student na podstawie imienia (String) oraz wieku (Integer).

https://github.com/idzikpro/JavaBasics/blob/master/src/main/java/pl/idzikpro/function/BiFunctionMain.java

```
BiFunction<String,Integer,Student> biFunction= (s, integer) -> new Student(s,integer);
System.out.println(biFunction.apply("Leszek",32));
```

Interfejs BinaryOperator<T> i metoda T apply(T,T)

Metoda apply(T,T) na podstawie dwóch obiektów typu T tworzy jeden T.

Wykorzystamy **BinaryOperator** do dodawania dwóch liczb.

https://github.com/idzikpro/JavaBasics/blob/master/src/main/java/pl/idzikpro/function/BinaryOperatorMain.java

```
BinaryOperator<Integer> binaryOperator= (integer1, integer2) -> integer1+integer2;
System.out.println(binaryOperator.apply(12,13));
```

Warianty prymitywne interfejsów funkcyjnych

Mamy w Java jeszcze interfejsy funkcyjne, które są odpowiednikami typów prymitywnych. Na przykład **IntPredicate** - na podstawie int zwraca Booleana. Podobnie jest **DoublePredicate**, **LongPredicate** itd. Podobnie jest z Consumer i Supplier.

Jest też np. Klasa **ToIntFunction.** Pobiera jakiś obiekt i zwraca inta. Są też oczywiście odpowiedniki dla innych typów prymitywnych.

Method references

Lambda to skrócony zapis **funkcji anonimowej**. A **method reference** to skrócony zapis wyrażenia lambda np.:

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

Zastosowanie:

1.Interfejsy funkcyjne wykorzystuje się w pracy na strumieniach : Stream API.

2.Biblioteka **vavr** rozszerza stosowanie programowania funkcyjnego.