Programowanie funkcyjne jest jedną z metodyk (zasad i wytycznych) programowania.

## 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 lokalnych.

@Getter  
@AllArgsConstructor

@ToString  
**final public class** Student {  
 **private** String **name**;  
 **private int age**;  
  
}

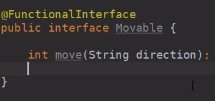
# Co to jest interfejs funkcyjny?

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 ;)

# Interfejs Predicate i metoda test(T)

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

W metodzie main tworzymy listę studentów i predykat kasiaPredicate, który pomoże nam wyszukać studentów o imieniu Kasia

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

Ten zapis można zastąpić następującym wyrażeniem lambda

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

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(**"Studenci"**);  
System.***out***.println(students);  
System.***out***.printf(**"Studenci z predykatem"**);  
System.***out***.println(*getStudents*(students,kasiaPredicate));

Jakie operacje można wykonywać na predykatach?

And (łączenie ze sobą), Or (sumowanie), negate (negowanie)

Predicate<Student> equals20OrMoreandKasia=kasiaPredicate**.and**(kasiaPredicate);  
System.***out***.println(*getStudents*(students,equals20OrMoreandKasia));

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 to uprościć do wyrażenia lambda

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

Możemy teraz wyświetlić (czyli skonsumować studentów ;)

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

Main wygląda następująco

**public static void** main(String[] args) {  
 List<Student> students = *createData*();  
 Consumer<Student> studentNameConsumer= student -> System.***out***.println(student.getName());  
 *consumeStudents*(students,studentNameConsumer);  
 }

A jak zrobić skonsumowanie (wyświetlenie studentów po przefiltrowaniu?)

List<Student> students = *createData*();  
Predicate<Student> kasiaPredicate = student -> **"Kasia"**.equals(student.getName());  
Consumer<Student> studentNameConsumer= student -> System.***out***.println(student.getName());  
***consumeStudents*(*getStudents*(students,kasiaPredicate),studentNameConsumer);**

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> printNameAndAge=printName**.andThen**(printAge);  
*consumeStudents*(*getStudents*(students,kasiaPredicate),printNameAndAge);

# Interfejs Supplier i metoda get()

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

Napiszemy teraz Suppliera, aby zwracał nam całą listę studentów.

Supplier<List<Student>> listSupplier=**new** Supplier<List<Student>>() {  
 @Override  
 **public** List<Student> get() {  
 **return** *createData*();  
 }  
};

Co można skrócić do wyrażenia lambda

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

Albo nawet jako zapis funkcyjny

Supplier<List<Student>> listSupplier= AppSupplier::*createData*;

Teraz użyjemy suppliera razem z consumerem i predicate. Należy jednak zmienić trochę metodę getStudents()

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

main

**public static void** main(String[] args) {  
 List<Student> students = *createData*();  
 Predicate<Student> kasiaPredicate = student -> **"Kasia"**.equals(student.getName());  
 Consumer<Student> printName = student -> System.***out***.println(student.getName());  
 Consumer<Student> printAge = student -> System.***out***.println(student.getAge());  
 Consumer<Student> printNameAndAge = printName.andThen(printAge);  
 **Supplier<List<Student>> listSupplier= AppSupplier::*createData*;**  
 *consumeStudents*(*getStudents*(**listSupplier**, kasiaPredicate), printNameAndAge);  
}

# 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> getStudentName=**new** Function<Student, String>() {  
 @Override  
 **public** String apply(Student student) {  
 **return** student.getName();  
 }  
};

Zapiszemy jako wyrażenie lambda.

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

Albo jako method reference

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

I użyjemy w programie, aby wyświetlić studentów zgodnie z predykatem

**public static void** main(String[] args) {  
 List<Student> students = *createData*();  
 Predicate<Student> kasiaPredicate = student -> **"Kasia"**.equals(student.getName());  
 Consumer<String> print = string -> System.***out***.println(string);  
 Supplier<List<Student>> listSupplier= AppSupplier::*createData*;  
 **Function<Student,String> getStudentName= Student::getName;** *consumeStudents*(*getStudents*(listSupplier,kasiaPredicate),getStudentName,print);  
}

Jednak trzeba zmienić trochę metodę consume()

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

## Interfejs BiFunction

Bierze dwa obiekty, a zwraca jeden obiekt.

## 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 **Supplie**r.

Jest też np. Klasa **ToIntFunction** = pobiera jakiś obiekt i zwraca inta lub **IntFunction** działa odwrotnie. Są odpowiedniki dla innych typów prymitywnych.

# Method references

To zapis np.:

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

Gdzie Student jest naszą klasą.

Można więc powiedzieć, że jest to **skrócenie zapisu lambdy**.

Supplier<List<Student>> listSupplier= AppSupplier::*createData*

Gdzie App jest naszą klasą.

# Optional i metoda get()

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

Chodzi o to, by uniknąć zwracania nulli.

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

Optional<String> name=Optional.*ofNullable*(student.getName());

2 sposób, gdy jesteśmy pewni, że nie będzie nullem

Optional<String> name=Optional.*of(*student.getName());

3 sposób, gdy zwracamy pustą wartość

Optional<String> name=Optional.empty();

**Jak korzystać z Optional?**

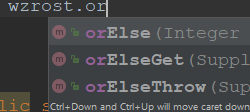
**1 sposób**

Optional<String> name=Optional.*ofNullable*(student.getName());  
**if** (name.isPresent()){  
 System.***out***.println(name.get());  
}

**jeśli wywołamy geta na pustym Optional dostaniemy wyjątek**

2 sposób (tutaj będzie consumer)

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



orElse – jeśli jest pusty, zwróć coś innego, orElseGet, to samo ale supplier

3 sposób (filtrowanie)

name.filter(i->i.equals(**"Kasia"**)).ifPresent(i-> System.***out***.println(i));

4 sposób (mapowanie)

name.map(i->i+**"a"**).filter(i->i.length()>0).ifPresent(i-> System.***out***.println(i));