

Какво представлява функцията ? - Представлява релация(връзка) между входни данни, които имат само 1 изход. Или накратко казано всяко едно уравнение или операция, която трябва да извършим , и то има само 1 изход се нарича функция. Ако има повече от два изхода, като например квадратното уравнение с x_1 , x_2 , това няма да са функции. Колкото и променливи да вкарваме във функцията, резултатът винаги е един единствен. Ако имам повече от 1 такъв това означава, че не използваме правилния начин.

Java Advanced - Functional Programming - януари 2017 - Ве

What is Function?

- Mathematic function

$$f(x) = x^2$$

A **function** is a special relationship where **each** input has a **single** output.


Lambda expressions - Състоят се от 3 основни части - параметри, тяло на функцията където се случват действията с параметрите и третата част е '->'

Java Advanced - Functional Programming - януари 2017 - Венцислав Иванов

Lambda Expressions

SOFTWARE UNIVERSITY FOUNDATION

- A lambda expression is an **unnamed function** with parameters and a body


(parameters) -> {body}

Lambda Expressions

- A lambda expression is an **unnamed function** with parameters and a body

- Lambda syntax

```
(parameters) -> {body}
```

- Use the lambda operator ->
 - Read as "goes to"

Чете се по следния начин:

- Параметрите отиват към тялото, където се извършват някакви действия с тях и в следствие на това се получава резултат.

От примерната картинка отгоре, ако заместим parameters с x , а body-то го заместим с x(на втора), ламбда изразът се чете, че x отива в x(на втора).

Ако имаме само един параметър в (parameters) не е задължително да слагаме още едни () , но ако са повече от 1 параметрите тогава е задължително. Примерно (x) -> {x + 1}, тук x е един единствен параметър и не е нужно да се поставят още едни () , но ако параметрите са повече от 1 , примерно ((x,y)) -> { x + y }; тук вече е необходимо да има тези допълнителни скоби.

Видове ламбди:

- Имплицитни - приемат само един параметър и извършват върху него някакво действие. Те са т.нар. void методи, т.е. не връщат резултат. В примера на снимката се вижда, че подаваме като параметър променливата msg и тя се изписва на конзолата. Резултат не се връща в този пример.

- **Implicit lambda expression**

```
msg -> System.out.println(msg);
```

– Експлицитни ламбда изрази - почти същите са като имплицитните , но с малко по-дълъг запис.

- **Explicit lambda expression**

```
(String msg) -> { System.out.println(msg); }
```

- Ламбда изрази без никакви входни данни. Те просто осъществяват някакво действие.

- **Zero parameters**

```
() -> { System.out.println("hi"); }
```

- Ламбда изрази с много и различни параметри. Задължително е когато се подават много параметри , да се подават и

техните типове, както е показано на следващата картинка.

More parameters

```
(int x, int y) -> { return x + y; }
```

7

Всеки един ламбда израз, можем спокойно да напишем като свой метод. Всяка една ламбда не зависи колко е сложна представлява метод. Компиляторът, когато види ламбда израз и създава метод.

Data in, Data out method - представлява метод, който приема параметър и директно ни връща някакъв резултат(output). Ако пипаме данни, които са странични за метода, тогава той не е Data in, Data out. Ако обаче имаме входни данни, които само обработваме по някакъв независимо колко дълъг начин. Само тях обработваме и не пипаме странични данни, това означава че този метод може да бъде заменен то ламбда израз. Така че следващия път когато пишем метод можем да си зададем въпроса дали е Data in, Data out и ако е.. би могъл спокойно да се замени от ламбда. Особено за методи, които използваме точно на едно място в нашия код, тоест трябва ни точно веднъж... такъв метод е перфектен за ламбда.

Functions(Функции).

Java Advanced - Functional Programming - януари 2017 - Венцислав Иванов

SOFTWARE UNIVERSITY FOUNDATION

Java Functions

Initialization of function

Lambda Expressions

```
Function<Integer, String> func = n -> n.toString();
```

Input type Output type Name Input parameter Return expression

- Input and output type can be different type
- Input and output type must be from type which we declare them

Функцията приема в себе си Input type и Output type(вторият параметър). Тялото на функцията представлява ламбда израз както се вижда на картинката.

Важни неща относно функцията са, че:

- Input-a и Output-a могат да бъдат от различни типове. Важно е само да декларираме какви са те.

-Друго важно е да връщаме резултат точно декларирания тип или иначе програмата гърми.

Ето как би изглеждал методът ако е написан като метод:

- In Java **Function<T, R>** is a interface that accepts a parameter of type **T** and returns type **R**

```
int increment(int number) {  
    return number + 1;  
}
```

Методът на горната картинка върши абсолютно същото като този на следващата:

```
Function<Integer, Integer> increment = number -> number + 1;
```

Ако обаче в кода ни се наложи примерно да напишем 10 пъти `number -> number + 1`, то тогава е по-добре да си използваме метода на по-горната снимка. Ако пък от друга страна в цялата ни програма ни се налага само веднъж да пресметнем нещо, то тогава би било по-удачно да използваме ламбдата `number -> number + 1` както е на картинката.

Функцията от горната снимка се използва с метода `.apply()`;

- We use function with **.apply()**

```
Function<Integer, Integer> increment = number -> number + 1;  
int a = increment.apply(5);  
int b = increment.apply(a);
```

Други видове функции(Специфични видове функции):

- `Consumer<T>` - представлява метод, който приема даден параметър(един или много) и не връща нищо(`void`), тоест той е `void` метод.

Java Advanced - Functional Programming - януари 2017 - Венцислав Иванов



Consumer<T>

- In Java **Consumer<T>** is a void interface:

```
void print(String message) {  
    System.out.println(message);  
}
```



Този метод като ламбда би изглеждал:

- Instead of writing the method we can do:

```
Consumer<String> print =  
    message -> System.out.print(message);
```

Как се работи с този метод:

- Then we use it like that:

```
print.accept("pesho");  
print.accept(String.valueOf(5));
```

-Supplier<T> - представлява метод, който не приема никакви входни параметри.

Java Advanced - Functional Programming - януари 2017 - Венцислав Иванов



Supplier<T>

- In Java **Supplier<T>** takes no parameters:

```
void genRandomInt()  
    Random rnd = new Random();  
    return rnd.nextInt(51);
```



- Instead of writing the method we can do:

```
Supplier<Integer> genRandomInt =  
    () -> new Random().nextInt(51);
```

- Then we use it like that:

```
int rnd = generateRandomInt.get();
```

-Predicate<T> - метод , който като резултат връща булева стойност(true or false).

Java Advanced - Functional Programming - януари 2017 - Венцислав Иванов



Predicate<T>

- In Java **Predicate<T>** evaluates a condition:

```
boolean isEven(int number) {  
    return number % 2 == 0;  
}
```



- Instead of writing the method we can do:

```
Predicate<Integer> isEven = number -> number % 2 == 0;
```

- Then we use it like that:

```
System.out.println(isEven.test(6)); //true
```

-UnaryOperator<T> - метод, който връща същата по тип стойност като тази, която приема.

UnaryOperator<T>

- `UnaryOperator<T>` is a function with same type of input/output

```
String toUpper(String str) {  
    return str.toUpperCase()  
}
```



- Instead of writing the method we can do:

```
UnaryOperator<String> toUpper = (x) -> x.toUpperCase();
```

- Then we use it like that:

```
System.out.println(isEven.test(6)); //true
```

21

BiFunctions:

- `BiFunction<T, U, R>` - представлява функция, която приема 2 параметъра (T, U), а R е резултатът, който тази функция връща. Тук типът на параметрите е абсолютно разнообразен.

Java Advanced - Functional Programming - януари 2017 - Венцислав Иванов

BiFunctions

- `BiFunction <T, U, R>`

```
BiFunction <Integer, Integer, String> sum =  
    (x, y) -> "The sum of " + x + " and " + y + " is " + (x + y);
```

- `BiConsumer<T, U>`

```
BiConsumer<Integer, Integer> sum = (x, y) -> {  
    System.out.println((x, y) -> "The sum of " + x + " and " + y  
+ " is " + (x + y);
```

- `BiPredicate <T, U>`

```
BiPredicate<Integer, Integer> bi = (x, y) -> x == y;  
System.out.println(bi.test(2, 3));  
//False
```

- `BiConsumer<T, U>` - приема два параметъра и не връща нищо като резултат.

- `BiPredicate<T, U>` - приема два параметъра и връща boolean (булева стойност - true or false) като резултат.

Special functions (Специални функции) :

- `IntFunction<R>` - приема като параметър стринг и връща цяло число (integer).

- `IntToDoubleFunction` - приема като параметър цяло число (integer) и връща число с плаваща запетая (double).

Special Functions

IntFunction <R>

```
IntFunction<String> i = (x) -> Integer.toString(x);
```

IntToDoubleFunction

```
IntToDoubleFunction i = (x) -> { return Math.sin(x); };
System.out.println(i.applyAsDouble(2));
// 0.90929742682
```

Някои от тези функции са 5-6 пъти по-бавни от един метод. Затова, когато ние пишем функции по-умният вариант е да преценяваме - ако една ламбда е една и съща, но трябва да я използваме над 2 пъти, то по-добрият вариант е да си направим метод. Ако се налага само веднъж да се пише, тогава ламбдата е по-добрият вариант.

Една от основните цели на ламбдата е кодът да става по-четим и по-кратък. Затова е общоприето, че в ламбдите параметрите се кръщават по обикновен начин (означава, че е по-добре параметъра да се казва примерно x, отколкото word).

Passing Functions to Method

Passing Functions to Method

We can pass Function<T,R> to methods:

```
int operation(int number, Function<Integer, Integer> function) {
    return function.apply(number);
}
```

We can use the method like that:

```
int a = 5;
int b = operation(a, number -> number * 5);
int c = operation(a, number -> number - 3);
int d = operation(b, number -> number % 2);
//b = 25
//c = 2
//d = 1
```

Както вече се спомена в началото, функциите могат да имат само един единствен изход, който задаваме в самата му декларация. Примерно в горната картинка, методът `int operation(..)` връща като резултат цяло число. Същото се отнася и за функцията, при чието деклариране ние посочваме какъв тип ще е резултатът. В случая на следващата картинка функцията ще приема като параметър цяло число и като резултат ще връща цяло число. Типът на резултатът, който ще се връща е обграден в зелено.

■ We can pass `Function<T,R>` to methods:

```
int operation(int number, Function<Integer, Integer> function) {  
    return function.apply(number);  
}
```

Горният метод `int operation...` може да се извика по начина показан в следващата снимка.

■ We can use the method like that:

```
int a = 5;  
int b = operation(a, number -> number * 5);  
int c = operation(a, number -> number - 3);  
int d = operation(b, number -> number % 2);  
//b = 25  
//c = 2  
//d = 1
```

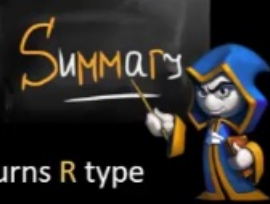
Както се вижда от картинката не е нужно да дефинираме нова функция , и после да я предаваме като аргумент на `operation` метода, а можем директно в него като параметър да пуснем лямбдата.

По начина, по който е дефиниран методът `operation` на по-горната снимка, ни позволява да подаваме като аргумент на методът лямбда израз или функция без да се налага и тя също да се дефинира някъде.

Следващата снимка е обобщителен слайд:

Summary

- Lambda expressions are anonymous methods
- `Consumer<T>` is a void function
- `Supplier<T>` gets no parameters
- `Predicate<T>` evaluates a condition
- `Function<T,R>` is a function that returns `R` type
- `UnaryOperator<T>` is a function that return same type
- `BiFunction<T, U, R>` is a function that accept more parameters
- Function can be pass like variable to methods



SOFTWARE UNIVERSITY
FOUNDATION