* What is Function ?
* Mathematic function
  + ***f*(x) = x2**

name = f , input = x , output =  **x2**

* A function is a special relationship where each input has a single output.
* A lambda expression is an unnamed function with parameters and a body

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

* Use the lambda operator “ -> “ read as “ goes to “
* Parameters can be enclosed in parentheses “ ( ) “
* The body holds the expression or statement and can be enclosed in braces { }
* Examples and different ways to define lambda expressions
  + Implicit lambda expression

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

* + Explicit lambda expression

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

* + Zero parameters

() -> { System.out.println(“Hello !”); }

* + More parameters

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

( if we have more parameters, we must define their type. )

* Initialization of function

Name Lambda expressions

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

Input type output type Input return expression

Parameter

* + - Input and output can be different type
    - Input and output type must be from type which we declare them
* Function<T,R>
  + 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;

}

* + We use function with .apply()
    - Function <Integer, Integer> increment = number -> number +1;

Int a = increment.apply(5);

Int b = increment.apply(a);

Other Type Functions

Special Functions

* Consumer<T>
  + In java Consumer<T> is a void interface:
    - void print(String message) {

System.out.println(message)

}

* + Instead of writing the method we can do:
    - Consumer<String> print = message -> System.out.print(message);
  + Then we use it like that:

print.accept(“pesho”);

print.accept(String.valueOf(5));

* Supplier<T>
  + In java Supplier<T> takes no parameters:
    - void genRandomInt(String message) {

Random rnd = new Random();

Return rnd.nextInt(51);

}

* Predicate<T>
  + In java Predicate <T> evaluates a condition:
    - boolean isEven(int num) {

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
* 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

* 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**

**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(b, number -> number - 3);

// b = 25;

// c = 22;