Predicate and function are almost same, the only difference is predicate always returns Boolean value, while Function returns any type of object (String, User, Employee, ArrayList.. etc)

|  |  |
| --- | --- |
| Predicate | Function |
| To implement conditional checks, we should go for Predicate | To perform certain operations and to return some result we should got for functions |
| Predicate can take any type of parameter, which represents input argument | Function can take 2 type parameters. First represents input type and second is return type |
| Predicate defines an abstract method called test | Function defines an abastract method called apply |
| Always return boolean (true or false) | Returns any type of object |
| java.util.function.Predicate<T> | R java.util.function.Function<T, R> |
| @FunctionalInterface  **public** **interface** Predicate<T> {  **boolean** test(T t);  } | @FunctionalInterface  **public** **interface** Function<T, R> {  /\*\*  \* Applies this function to the given argument.  \*  \* **@param** t the function argument  \* **@return** the function result  \*/  R apply(T t);  …..  } |

Find length of string using Function Functional Interface

Find square of a number

**package** com.mahendra.section7;

**import** java.util.function.Function;

**public** **class** FunctionExampleOne {

**public** **static** **void** main(String[] args) {

Function<String, Integer> lengthOfStringFunction = s -> s.length();

Integer length = lengthOfStringFunction.apply("Hello World!");

System.***out***.println(length);

Function<Integer, Integer> squareOfFunction = i -> i \* i;

System.***out***.println(squareOfFunction.apply(25));

Function<String, String> trimString = s -> s == **null** ? s : s.trim();

System.***out***.println(trimString.apply(" Hello World! "));

Function<String, String> removeSpaces = s -> s == **null** ? s : s.replaceAll(" ", "");

System.***out***.println(removeSpaces.apply(" Hello World! "));

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

==

**package** com.mahendra.data;

**public** **class** Student {

**private** String name;

**private** **int** score;

**public** Student() {

}

**public** Student(String name, **int** score) {

**this**.name = name;

**this**.score = score;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getScore() {

**return** score;

}

**public** **void** setScore(**int** score) {

**this**.score = score;

}

}

**package** com.mahendra.data;

**public** **class** Employee {

**private** String name;

**private** **double** salary;

**public** Employee() {

}

**public** Employee(String name, **double** salary) {

**this**.name = name;

**this**.salary = salary;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **double** getSalary() {

**return** salary;

}

**public** **void** setSalary(**double** salary) {

**this**.salary = salary;

}

**public** String toString() {

**return** name + ":" + salary;

}

}

==

**package** com.mahendra.data;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Data {

**public** **static** List<Student> getStudentList() {

List<Student> studentList = **new** ArrayList<Student>();

studentList.add(**new** Student("Sunny", 100));

studentList.add(**new** Student("Bunny", 65));

studentList.add(**new** Student("Chinny", 55));

studentList.add(**new** Student("Vinny", 45));

studentList.add(**new** Student("Pinny", 25));

**return** studentList;

}

**public** **static** List<Employee> getEmployeeList() {

List<Employee> employeeList = **new** ArrayList<Employee>();

employeeList.add(**new** Employee("Mark", 1000));

employeeList.add(**new** Employee("Joan", 2000));

employeeList.add(**new** Employee("David", 3000));

employeeList.add(**new** Employee("Pet", 4000));

employeeList.add(**new** Employee("Jack", 5000));

**return** employeeList;

}

}

==

**package** com.mahendra.section7;

**import** java.util.List;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

**import** com.mahendra.data.Data;

**import** com.mahendra.data.Student;

**public** **class** FunctionExampleTwo {

**public** **static** **void** main(String[] args) {

Predicate<Student> gradeA = s -> s.getScore() >= 80;

Predicate<Student> gradeB = s -> s.getScore() >= 60 && s.getScore() < 80;

Predicate<Student> gradeC = s -> s.getScore() >= 50 && s.getScore() < 60;

Predicate<Student> gradeD = s -> s.getScore() >= 35 && s.getScore() < 50;

Function<Student, String> f = s -> {

**if** (gradeA.test(s)) {

**return** "A Grade";

} **else** **if** (gradeB.test(s)) {

**return** "B Grade";

} **else** **if** (gradeC.test(s)) {

**return** "C Grade";

} **else** **if** (gradeD.test(s)) {

**return** "D Grade";

} **else** {

**return** "E Grade";

}

};

List<Student> studentList = Data.*getStudentList*();

**for** (Student s : studentList) {

System.***out***.println("Student Name:" + s.getName());

System.***out***.println("Student Score:" + s.getScore());

System.***out***.println("Student Grade:" + f.apply(s));

System.***out***.println();

}

}

}

**package** com.mahendra.section7;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

**import** com.mahendra.data.Data;

**import** com.mahendra.data.Employee;

**public** **class** FunctionExampleThree {

**public** **static** **void** main(String[] args) {

List<Employee> employeeList = Data.*getEmployeeList*();

System.***out***.println(employeeList);

Function<List<Employee>, Double> salariesFunction = eList -> {

**double** total = 0;

**for** (Employee e : eList) {

total = total + e.getSalary();

}

**return** total;

};

System.***out***.println("The total salaries of employees for this month:" + salariesFunction.apply(employeeList));

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Predicate<Employee> p = e -> e.getSalary() < 3500;

Function<Employee, Employee> f = e -> {

e.setSalary(e.getSalary() + ((3 \* e.getSalary() / 100)));

**return** e;

};

List<Employee> l2 = **new** ArrayList<Employee>();

**for** (Employee e : employeeList) {

**if** (p.test(e)) {

l2.add(f.apply(e));

}

}

System.***out***.println("Employees with incremented salary:");

System.***out***.println(l2);

}

}

==

What is function composition?

It all has to do with creating small reusable functions that you can combine to compose new functions. We have and or etc for predicate, similarly we have compose , andThen in functions.

f1.andThen.f2 => f1 will be applied followed by f2.

f1.compose.f2 => f2 will be applied followed by f1.

Now, how can we achieve this using compose and andThen?

Let's first define two simple functions - times2 and squared.

Function<Integer, Integer> times2 = e -> e \* 2;

Function<Integer, Integer> squared = e -> e \* e;

Next, let's combine them, using compose and andThen.

times2.compose(squared).apply(4);

// Returns 32

times2.andThen(squared).apply(4);

// Returns 64

As you can see, the difference between compose and andThen is the order they execute the functions. While the compose function executes the caller last and the parameter first, the andThen executes the caller first and the parameter last.

**identity()**

A function that returns input value as output is identity function

**public** **class** FunctionIdentity {

**public** **static** **void** main(String[] args) {

Function<String, String> fId = Function.*identity*();

String output = fId.apply("Test case");

System.***out***.println(output);

}

}

==

<https://stackoverflow.com/questions/28032827/java-8-lambdas-function-identity-or-t-t>

I have a question regarding the usage of the Function.identity() method.

Imagine the following code:

Arrays.asList("a", "b", "c")

.stream()

.map(Function.identity()) // <- This,

.map(str -> str) // <- is the same as this.

.collect(Collectors.toMap(

Function.identity(), // <-- And this,

str -> str)); // <-- is the same as this.

Is there any reason why you should use Function.identity() instead of str->str (or vice versa). I think that the second option is more readable (a matter of taste of course). But, is there any "real" reason why one should be preferred?

==

As of the current JRE implementation, Function.identity() will always return the same instance while each occurrence of identifier -> identifier will not only create its own instance but even have a distinct implementation class. For more details, see [here](https://stackoverflow.com/a/27524543/2711488).

The reason is that the compiler generates a synthetic method holding the trivial body of that lambda expression (in the case of x->x, equivalent to return identifier;) and tell the runtime to create an implementation of the functional interface calling this method. So the runtime sees only different target methods and the current implementation does not analyze the methods to find out whether certain methods are equivalent.

So using Function.identity() instead of x -> x might save some memory but that shouldn’t drive your decision if you really think that x -> x is more readable than Function.identity().

You may also consider that when compiling with debug information enabled, the synthetic method will have a line debug attribute pointing to the source code line(s) holding the lambda expression, therefore you have a chance of finding the source of a particular Function instance while debugging. In contrast, when encountering the instance returned by Function.identity() during debugging an operation, you won’t know who has called that method and passed the instance to the operation.

==

From the [JDK source](http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/function/Function.java#l97):

static <T> Function<T, T> identity() {

return t -> t;

}

So, no, as long as it is syntactically correct.

In your example there is no big difference between str -> str and Function.identity() since internally it is simply t->t.

But sometimes we can't use Function.identity because we can't use a Function. Take a look here:

List<Integer> list = new ArrayList<>();

list.add(1);

list.add(2);

this will compile fine

int[] arrayOK = list.stream().mapToInt(i -> i).toArray();

but if you try to compile

int[] arrayProblem = list.stream().mapToInt(Function.identity()).toArray();

you will get compilation error since mapToInt expects ToIntFunction, which is not related to Function. Also ToIntFunction doesn't have identity() method.