**Lambda Document (v1)**

**Created by**

**Vineet Semwal (vineetsemwal82@gmail.com)**

**Functional Interface**

1. Interface with exactly one method
2. Optionally marked with @FunctionalInterface
3. Functional interfaces are used for cases where we need to pass around functionality

**@FunctionalInterface**

interface IAdder{

int add(int a, int b);

}

**Lambda Expression**

Lambda expressions are basically instances of functional interfaces

lambda expressions are added in Java 8

1)Expression

IAdder<Integer> adder=(a,b)->a+b;

-> is used to separate arguments and body of expression

LHS of -> denotes arguments to function

RHS of ->denotes body of expression

1. Block ( set of statements)

IAdder<Integer> adder=(a,b)->{  
 **int** c= a+b;  
 **return** c;  
};

**Builtin Functional Interfaces**

1. Supplier
2. Consumer
3. Predicate
4. Function

**Supplier**

Represents function that takes no argument and return a result of Type T

@FunctionalInterface  
**public interface** Supplier<T> {  
T get();  
}

Supplier<Student>**supplier**=()->**new** Student();

Student student=supplier.get();

**Consumer**

Represents function that takes one argument and does NOT return anything

@FunctionalInterface

public interface Consumer<T> {

void accept(T t);

}

How to use it

Consumer<String> up=(arg)->{  
 String upperCase=arg.toUpperCase();  
 System.***out***.println(upperCase);  
};

up.accept(“hello”);

**BiConsumer**

Represents function that takes two arguments and does NOT return anything

@FunctionalInterface  
**public interface** BiConsumer<T, U> {  
**void** accept(T t, U u);  
}

BiConsumer<String,Integer>con=(input,times)->{  
 String result=**""**;  
 **for** (**int** i=0;i<times;i++){  
 result=result+input;  
 }  
 System.***out***.println(result);  
};  
con.accept(**"hello"**,3);

**Predicate**

Represents a function that takes an argument and returns true or false

Predicate<Integer>isEven=arg->arg%2==0;  
**boolean** result=isEven.test(11);  
System.***out***.println(result);

**BiPredicate**

Represents a function that takes two argument and returns true or false

BiPredicate<String,Integer> isLengthExpected= (input,length)->input.length()==length;  
**boolean** result=isLengthExpected.test(**"hello"**,5);  
System.***out***.println(result);

**Function**

Represents a function that takes one argument and returns the result

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

}

Function<String,Integer>length=arg->arg.length();  
**int** result=length.apply(**"hello"**);

**BiFunction**

Represents a function that takes two arguments and returns the result

@FunctionalInterface  
**public interface** BiFunction<T, U, R> {  
R apply(T t, U u);

}

BiFunction<String,Integer,String>concat=(input,times)->{  
 String result=**""**;  
 **for**(**int** i=0;i<times;i++){  
 result=result+input;  
 }  
 **return** result;  
};  
String result=concat.apply(**"hello"**,3);

**UnaryOperator**

Represents function that takes one argument and return result of same type

@FunctionalInterface

public interface UnaryOperator<T> extends Function<T, T> {

static <T> UnaryOperator<T> identity() {

return t -> t;

}

}

UnaryOperator<Integer>twicer=(input)->input\*2;

**int** result=twicer.apply(10);

It is same as

Function<Integer,Integer>twicer=(input)->input\*2;  
**int** result=twicer.apply(10);

**BinaryOperator**

Represents function that takes two arguments and return result of same type

BinaryOperator<Integer>power=(input,times)->{  
 **int** result=1;  
 **for** (**int** i=0;i<times;i++){  
 result=result\*input;  
 }  
 **return** result;  
};  
**int** result=power.apply(10,3);

It is same as

BiFunction<Integer,Integer,Integer>power=(input,times)->{  
 **int** result=1;  
 **for** (**int** i=0;i<times;i++){  
 result=result\*input;  
 }  
 **return** result;  
};  
**int** result=power.apply(10,3);

**Method Reference**

1. A method reference provides a way to refer to a method without executing it
2. It relates to lambda expressions because return type is compatible functional interface

Consumer<String>consumer=(input)->System.***out***.println(input)

Or

Consumer<String>consumer=System.***out***::println

<class or instance name> :: <method name>

Double colon specifies method reference

**Method Reference Types**

1. Reference to Static method using classname
2. Reference to Instance method using instance
3. Reference to constructor using syntax Classname::new

**Method reference using classname**

BinaryOperator<Integer>operator=Adder::add;

int result=operator.apply(1,2);

public class Adder{  
**public static int** add(**int** a,**int** b){  
 **return** a+b;  
}

}

**Method reference using instance**

Adder adder=new Adder();

BinaryOperator<Integer>operator=adder::add;

int result=operator.apply(1,2);

public class Adder{  
**public int** add(**int** a,**int** b){  
 **return** a+b;  
}

}

**Constructor Reference**

Reference to constructor using syntax Classname::new

**class** Student{  
 String **name**;  
 **public** Stud(String arg){  
 **this**.**name**=arg;  
 }

**void** doSomething(){  
 Function<String,Student>function=Student::**new**;  
 Student result=function.apply(**"scooby"**);  
 }

Get instance of Arraylist

Supplier<ArrayList>supplier= ArrayList::new;

ArrayList list=supplier.get();

}