**Functional Interfaces in Java 8 – 2022**

**Functional programming** is a declarative programming paradigm which treats computation as mathematical evaluation without changing the state. A **functional interface** is an interface that contains only one abstract method.

**Consumer<Input> 🡺 takes an argument and returns nothing. Consumer accepts (consumes)**

**Consumer<Input> 🡺 void accept(Input input)**

**@FunctionalInterface**

**public interface Consumer<T> {**

**void accept(T t);**

**}**

**Consumer<String> print = x -> System.out.println(x);**

**print.accept("java"); // java**

**OUTPUT**

arpit  
ARPIT

**Consumer<String> firstC = x -> System.out.println(x.toLowerCase());**

**Consumer<String> secondC = y -> System.out.println(y.toUpperCase());**

**Consumer<String> result = firstC.andThen(secondC);**

**result.accept("Arpit");**

**public void onlyConsume**(String val) {  
 System.***out***.println(val);  
}

**public void** check() {  
 Consumer consumer = x -> **onlyConsume**(x.toString());  
 consumer.accept("abcd");  
}

**public class** Check1 {  
  
 **public static void** show(String value) {  
 System.***out***.println(value);  
 }  
  
 **public void** show1(String value) {  
 System.***out***.println(value);  
 }  
  
 **public static void** main(String[] args) {  
 Consumer<String> consumer = Check1::*show*;  
 consumer.accept(**"Apple"**);  
  
 Consumer<String> consumer1 = t -> **new** Check1().show1(t);  
 consumer1.accept(**"orange"**);  
 }  
}

**Class implementing Consumer<I>**

public class **ConsumerImpl implements Consumer<String>** {  
  
 @Override  
 **public void accept(String t) {  
 System.*out*.println(t);  
 }**}

public class Test {  
 public static void main(String[] args) {  
 **Consumer<String> consumer =**

**s -> System.*out*.println(s);  
 consumer.accept("hello");**  
 Consumer1 consumer1 = new Consumer1();  
 **consumer1.getConsumer().accept("hello");**  
 }  
}

public class Consumer1 {  
  
 **public Consumer<String> getConsumer() {  
 return s -> System.*out*.println(s);  
 }**  
}

## **Supplier<O> 🡺 takes no arguments and returns a result. Supplier<O> 🡺 <O> get()**

**@FunctionalInterface**

**public interface Supplier<Output> {**

**T get();**

**}**

Supplier<LocalDateTime> s = () -> LocalDateTime.now();

**LocalDateTime time = s.get();**

**Class implementing Supplier<O>**

public class SupplierImpl implements Supplier<String> {  
 @Override  
 public String get() {  
 return "Hello World";   
 }  
}

**How to use 🡺 Inside main method**

SupplierImpl supplier = new SupplierImpl();  
System.*out*.println(supplier.get());  
  
Supplier1 supplier1 = new Supplier1();  
System.*out*.println(supplier1.getSupplier().get());

public class Supplier1 {  
  
 public Supplier<String> getSupplier() {  
 return () -> "World of Functional Programming!";  
 }  
}

## **BiConsumer<Input1, Input2> 🡺** takes two arguments and returns nothing.

**@FunctionalInterface BiConsumer<Input, Input> 🡺 void accept(input1, input2)**

**public interface BiConsumer<T, U> {**

**void accept(T t, U u);**

**}**

BiConsumer<Integer, Integer> addTwo = (x, y) -> System.out.println(x + y);

addTwo.accept(1, 2);

BiConsumer<String,String> biConsumer = (t,v) -> System.***out***.println(**"Complete Name: "**+(t+**" "**+v));  
biConsumer.accept(**"John"**, **"Abraham"**);

**public void onlyConsume**(**int** x, **int** y) {  
 System.***out***.println(x+y);  
}  
  
**public void** check() {  
 BiConsumer<Integer,Integer> **biConsumer** = (x,y) -> onlyConsume(x,y);  
 biConsumer.accept(10,20);  
}

## **Predicate<Input> 🡺 accepts an argument and returns boolean. It used to apply in a filter for a collection of objects.**

**@FunctionalInterface Predicate<Input> 🡺 boolean test(input)**

**public interface Predicate<T> {**

**boolean test(T t);**

**}**

**public boolean** isTrue(String value) {  
 **return true**;  
}

**public void** check() {  
 Predicate p = x -> isTrue(x.toString());  
 **boolean** value = p.test("some value");  
 System.***out***.println(value);  
}

Predicate<Integer> pr = a -> (a > 18);

System.out.println(pr.test(10)); // false

**public static** Boolean checkAge(**int** age) {  
 **if** (age > 17)  
 **return true**;  
 **else return false**;  
}

Predicate<Integer> p = Check1::*checkAge*;  
**boolean** flag = p.test(17);  
System.***out***.println(flag); //false

Predicate<Integer> noGreaterThan5 = x -> x > 5;

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> collect = list.stream().filter(noGreaterThan5).collect(Collectors.toList());

## **BiPredicate<Input1,Input2>** 🡺 which accepts two arguments and returns a boolean

**@FunctionalInterface BiPredicate<Input, Input> 🡺 boolean test(input1, input2)**

**public interface BiPredicate<Input1, Input2> {**

**boolean test(Input input1, Input2 input2);**

**}**

BiPredicate<String, Integer> filter = (x, y) -> {

return x.length() == y;

};

boolean result = filter.test("mkyong", 6);

System.out.println(result); // true

BiPredicate<String, Integer> bp = (name, age) -> {  
 **return** name.equals(**"Ram"**) & age < 60;  
};  
**boolean** flag = bp.test(**"Ram"**, 24);  
System.***out***.println(flag);

## **Function<Input,Output>** 🡺  **takes an argument (object of type I) and returns an object (object of type O)**

**@FunctionalInterface Function<Input, Output> 🡺  <Output> apply(input)**

**public interface Function<I, O> {**

**O apply(I i);**

**}**

**public** String getValue(**int** x) {  
 **return** "some value";  
}  
  
**public void** check() {  
 Function fn = x -> getValue(Integer.*parseInt*(x.toString()));  
 String val = (String) fn.apply(23);  
 System.***out***.println(val);  
}

Function<String, Integer> func = x -> x.length();

Integer apply = func.apply("mkyong"); // 6

System.out.println(apply);

Function<String, Integer> func = x -> x.length();

Function<Integer, Integer> func2 = x -> x \* 2;

Integer result = func.andThen(func2).apply("mkyong"); // 12

System.out.println(result);

**Class implementing Function<Input,Output>**

public class FunctionImpl implements Function<String, String> {  
 @Override  
 public String apply(String t) {  
 return t + " World of Functions";  
 }  
}

public class Function1 {  
 public Function<String,String> getFunction() {  
 return val -> val + " World of Functions!";  
 }  
}

**How to use – Inside main method**

String value = new FunctionImpl().apply("Hello");  
System.*out*.println("Value: " + value);  
  
Function1 function1 = new Function1();  
String value2 = function1.getFunction().apply("Hello");  
System.*out*.println("Value2: " + value2);

## **BiFunction<Input1, Input2, Output> 🡺** takes two arguments and returns an object.

**BiFunction 🡺  <Output> apply(input1, input2)**

**@FunctionalInterface**

**public interface BiFunction<Input1, Input2, Output> {**

**R apply(Input1 i1, Input2 i2);**

**}**

BiFunction<Integer, Integer, Integer> func = (x1, x2) -> x1 + x2;

Integer result = func.apply(2, 3);

System.out.println(result); // 5

## **UnaryOperator<T>** 🡺 functional interface and it extends Function. takes one argument, and returns a result of the same type of its arguments.

@FunctionalInterface

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

}

UnaryOperator<Integer> func2 = x -> x \* 2;

Integer result2 = func2.apply(2);

## **BinaryOperator<T>** 🡺 functional interface and it extends BiFunction. takes two arguments of the same type and returns a result of the same type of its arguments.

@FunctionalInterface

public interface BinaryOperator<T> extends BiFunction<T,T,T> {

}

BinaryOperator<Integer> func2 = (x1, x2) -> x1 + x2;

Integer result2 = func.apply(2, 3);

Comparator<Integer> comparator = (a, b) -> (a.compareTo(b));  
 *// Using maxBy()* BinaryOperator<Integer> opMax = BinaryOperator.*maxBy*(comparator);  
 System.***out***.println(**"Max: "** + opMax.apply(5, 6));*//Prints 6* System.***out***.println(**"Max: "** + opMax.apply(9, 6));*//Prints 9  
  
 // Using minBy()* BinaryOperator<Integer> opMin = BinaryOperator.*minBy*(comparator);  
 System.***out***.println(**"Min: "** + opMin.apply(5, 6));*//Prints 5* System.***out***.println(**"Min: "** + opMin.apply(9, 6));*//Prints 6*