**Functional Interfaces in Java 8 – 2022**

A **functional interface** is an interface that contains only one abstract method.

**Consumer 🡺 takes an argument and returns nothing. Consumer accepts**

**@FunctionalInterface Consumer 🡺 accept()**

**public interface Consumer<T> {**

**void accept(T t);**

**}**

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

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

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");

arpit  
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"**);  
  
 }  
}

## **BiConsumer 🡺** takes two arguments and returns nothing.

@FunctionalInterface **BiConsumer 🡺 accept()**

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

@FunctionalInterface **Predicate** 🡺 test()

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<T,U>** 🡺 which accepts two arguments and returns a boolean

@FunctionalInterface

public interface BiPredicate<T, U> {

boolean test(T t, U u);

}

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<T,R>** 🡺  takes an argument (object of type T) and returns an object (object of type R)

@FunctionalInterface **Function** 🡺   **apply**()

public interface Function<T, R> {

R apply(T t);

}

**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);

## **BiFunction<T, U, R> 🡺** takes two arguments and returns an object.

@FunctionalInterface

public interface BiFunction<T, U, R> {

R apply(T t, U u);

}

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*

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

@FunctionalInterface

public interface Supplier<T> {

T get();

}

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

LocalDateTime time = s.get();

**What is Functional Programming**

**Functional programming** is a [programming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) that treats [computation](https://en.wikipedia.org/wiki/Computation) as the evaluation of [mathematical functions](https://en.wikipedia.org/wiki/Function_(mathematics)) and avoids changing-[state](https://en.wikipedia.org/wiki/Program_state) and [mutable](https://en.wikipedia.org/wiki/Immutable_object) data. It is a [declarative programming](https://en.wikipedia.org/wiki/Declarative_programming) paradigm, which means programming is done with [expressions](https://en.wikipedia.org/wiki/Expression_(computer_science))[[1]](https://en.wikipedia.org/wiki/Functional_programming#cite_note-expression_style-1) or declarations[[2]](https://en.wikipedia.org/wiki/Functional_programming#cite_note-declaration_style-2) instead of [statements](https://en.wikipedia.org/wiki/Statement_(computer_science)).

**Functional programming** is a declarative programming paradigm which treats computation as mathematical evaluation without changing the state.