Java 8 Features

1. Lambda Expression / Anonymous Functions / Closures

2. Functional Interface

3. Default Methods

4. Predicate

5. Functions

6. Stream

1. Lambda Expression

Advantages

1. Less Code
2. Easy to implement anonymous Inner Classes
3. Pass Lambda Expression as parameter to other Methods
4. Functional Interface used to express Lambda Expression
5. Functional Interface
6. An interface having only one abstract method and the abstract method is called functional method
7. Example of Functional Interface:

Runnable which has run()

Comparator which has compareTo()

1. We can add multiple default methods in Functional Interface
2. @FunctionalInterface annotation

**Example 1: Without Lambda**

@FunctionalInterface

**public** **interface** A {

**void** myMethod();

}

@FunctionalInterface

**public** **interface** B **extends** A {

**void** myMethod();

}

**public** **class** C **implements** A {

@Override

**public** **void** myMethod() {

System.***out***.println("inside myMethod()");

}

}

**public** **class** TesterWithoutLambda {

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

A a = **new** C();

a.myMethod();

}

}

**Example 2: With Lambda without parameters**

@FunctionalInterface

**public** **interface** A {

**void** myMethod();

}

@FunctionalInterface

**public** **interface** B **extends** A {

**void** myMethod();

}

**public** **class** C **implements** A {

@Override

**public** **void** myMethod() {

System.***out***.println("inside myMethod()");

}

}

**public** **class** TesterWithLambdaWithoutParameter {

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

A a = () -> System.***out***.println("inside myMethod");

a.myMethod();

}

}

**Example 3: Addition With Lambda with parameters**

@FunctionalInterface

**public** **interface** Sum {

**void** add(**int** a, **int** b);

}

**public** **class** TesterWithLambdaWithParameter {

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

Sum sum = (a, b) -> System.out.println("Sum is : " + (a + b));

sum.add(100, 200);

}

}

**Example 4: Lambda Using Runnable Interface with External Class**

**public** **class** Tester {

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

Runnable runnable = **new** MyRunnableImpl();

Thread thread = **new** Thread(runnable);

thread.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread : " + (i + 1));

}

}

}

**public** **class** MyRunnableImpl **implements** Runnable {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread " + (i + 1));

}

}

}

**Example 5: Lambda Using Runnable Interface Without External Class**

**public** **class** Tester {

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

Runnable runnable = () -> {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread " + (i + 1));

}

};

Thread thread = **new** Thread(runnable);

thread.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread : " + (i + 1));

}

}

}

**Example 6: Anonymous Inner Class without Lambda**

**public** **class** Tester {

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

Thread thread = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread : " + (i + 1));

}

}

});

thread.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread : " + (i + 1));

}

}

}

**Example 7: Anonymous Inner Class with Lambda**

**public** **class** Tester {

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

Thread thread = **new** Thread(() -> {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread : " + (i + 1));

}

});

thread.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread : " + (i + 1));

}

}

}

1. Default Methods in Interface

**public** **interface** A {

**default** **void** m1() {

System.***out***.println("m1() inside A");

}

}

**public** **class** B **implements** A{

**public** **void** m1() {

System.***out***.println("m1() inside B");

}

}

**public** **class** Tester {

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

A a = **new** B();

a.m1();

}

}

Diamond Problem with default interface

1. Predicate
2. Predicate is a functional interface which can be expressed through Lambda Expression.
3. Predicate accepts single argument and returns boolean value

interface Predicate<T>{

public boolean test(T t);

}

**Example 1: Predicate GreaterThanTwenty**

**public** **class** GreaterThanTwenty {

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

Predicate<Integer> p1 = i -> (i > 20);

System.***out***.println(p1.test(10));

System.***out***.println(p1.test(20));

System.***out***.println(p1.test(30));

}

}

**Example 2: Predicate LengthOfString**

**public** **class** LengthOfString {

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

Predicate<String> p1 = s -> (s.length() > 5);

System.***out***.println(p1.test("delhi"));

System.***out***.println(p1.test("goa"));

System.***out***.println(p1.test("mumbai"));

}

}

**Example 3: Passing Predicate to Function/Method**

**public** **class** PredicateDemo {

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

PredicateDemo obj = **new** PredicateDemo();

**int**[] arr = { 0, 7, 10, 20, 30, 77 };

Predicate<Integer> p1 = i -> (i > 10);

obj.m1(p1, arr);

}

**public** **void** m1(Predicate<Integer> p, **int**[] arr) {

**for** (**int** value : arr) {

**if** (p.test(value))

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

}

}

}

**Example 4: Predicate Joins (and(), or() and negate())**

**public** **class** PredicateJoins {

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

PredicateJoins obj = **new** PredicateJoins();

**int**[] arr = { 0, 7, 10, 20, 30, 77 };

Predicate<Integer> p1 = i -> (i > 10);

Predicate<Integer> p2 = i -> (i % 2 == 0);

System.***out***.println("Greater than 10");

obj.m1(p1, arr);

System.***out***.println("Even Numbers");

obj.m1(p2, arr);

System.***out***.println("Not Greater than 10");

obj.m1(p1.negate(), arr);

System.***out***.println("Greater than 10 and Even");

obj.m1(p1.and(p2), arr);

System.***out***.println("Greater than 10 or Even");

obj.m1(p1.or(p2), arr);

}

**public** **void** m1(Predicate<Integer> p, **int**[] arr) {

**for** (**int** value : arr) {

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

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

}

}

}

}

5.Function

1. Function is a functional interface which can be expressed through Lambda Expression.

1. Function accepts single argument and returns any type value

3. Function Interface

interface Function(T,R){

R apply(T t);

}

**Example 1: Function**

**public** **class** FunctionDemo {

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

Function<String,Integer> f1 = s->s.length();

System.***out***.println(f1.apply("Pushkar"));

System.***out***.println(f1.apply("Indore"));

}

}

6.Method Referencing (method argument type must be same)

**Example 1: Method Referencing with double colon operator**

**Mapping a static method to Functional Interface method**

**public** **class** MethodReferencingDemo {

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

Runnable runnable = MethodReferencingDemo::*myMethod*;

Thread thread = **new** Thread(runnable);

thread.start();

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Main Thread " + i);

}

}

**public** **static** **void** myMethod() {

**for** (**int** i = 0; i < 10; i++) {

System.***out***.println("Child Thread " + i);

}

}

}

**Example 2: Method Referencing with double colon operator**

**Mapping an instance method to Functional Interface method**

**public** **interface** MyInterface {

**public** **void** myMethod(**int** i);

}

**public** **class** MethodReferencingDemo {

// This method parameter should be same as functional interface method

**public** **void** myMethod123(**int** i) {

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

}

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

MethodReferencingDemo obj = **new** MethodReferencingDemo();

// Express Functional Interface using Lambda

MyInterface f1 = i -> System.***out***.println(i);

f1.myMethod(10);

MyInterface f2 = obj::myMethod123;

f2.myMethod(20);

}

}

**Example 3: Method Referencing with double colon operator**

**Mapping a constructor to Functional Interface method**

**public** **class** MyClass {

**private** String s;

MyClass(String s) {

**this**.s = s;

System.***out***.println("Inside constructor " + s);

}

}

**public** **interface** MyInterface {

MyClass get(String s);

}

**public** **class** Tester {

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

// Express Functional Interface Using Lambda Expression

MyInterface f1 = s -> **new** MyClass(s);

f1.get("Using Lambdas");

// Using Constructor Mapping

MyInterface f2 = MyClass::**new**;

f2.get("Using Constructor Mapping");

}

}

7. Streams

1. Stream Configuration

* 1. Filter

public Stream filter(Predicate<T> p)

* 1. Map

public Stream map(Function f)

1. Processing
   1. collect()
   2. count()
   3. sorted()
   4. min()
   5. max()

**Example 1: Without Stream (More Code)**

**public** **class** FilterEvenNumbersWithoutStream {

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

List<Integer> list1 = **new** ArrayList<>();

**for** (**int** i = 0; i < 10; i++)

list1.add(i);

System.***out***.println(list1);

List<Integer> list2 = **new** ArrayList<>();

**for** (**int** i = 0; i < list1.size(); i++) {

**if** (list1.get(i) % 2 == 0) {

list2.add(list1.get(i));

}

}

System.***out***.println(list2);

}

}

**Example 2: With Stream and Filter**

**public** **class** FilterEvenNumbersWithStream {

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

List<Integer> list1 = **new** ArrayList<>();

**for** (**int** i = 0; i < 10; i++)

list1.add(i);

System.***out***.println(list1);

List<Integer> list2 = list1.stream().filter(i -> i % 2 == 0).collect(Collectors.*toList*());

System.***out***.println(list2);

}

}

**Example 3: With Stream and Map**

**public** **class** MapUppercaseToLowercase {

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

List<String> list1 = **new** ArrayList<>();

list1.add("PUSHKAR");

list1.add("VIKAS");

list1.add("BHARATH");

list1.add("JOHN");

System.***out***.println(list1);

List<String> list2 = list1.stream().map(s->s.toLowerCase()).collect(Collectors.*toList*());

System.***out***.println(list2);

}

}