***INTRODUCTION TO LAMBDA EXPRESSION***

* *Java 8 provided ability of Minimal code, functional programming along with Object oriented programming*
* *Java 8 introduced lambda expression, Streams, Date & Time API*

***Lambda Expression***

* *Anonymous function*
* *No name, no return type, no access modifier*
* *Used to implement methods of functional interface (Interface having only one abstract method but can multiple concrete methods)*

*Before using lambda function if we wanted to use abstract methods of functional interface then we should have created new class as below*

public class BasicProgramming {  
 public static void main(String args[]) {  
 Thread t1=new Thread(new Task());  
 }  
}

class Task implements Runnable{   
 @Override  
 public void run() {  
 System.*out*.println("Hello");  
 }  
}

*Below can be implemented using lambda expression below*

public class BasicProgramming {  
 public static void main(String args[]) {  
 Thread t1=new Thread(()->{  
 System.*out*.println("Hello");  
 });  
 }  
}

***Program Example***

package Streams;  
import java.util.stream.\*;  
  
public class BasicProgramming {  
 public static void main(String args[]) {  
 MathOperations sumoperation= (a, b)->a+b;  
 MathOperations suboperation= (a, b)->a-b;  
 int value1=sumoperation.operate(5,6);  
 int value2=suboperation.operate(5,6);  
 System.*out*.println(value1+" "+value2);  
 }  
***}***  
interface MathOperations{  
 int operate(int a,int b);  
}

*O/P-* ***11 -1***

***Predicate***

* *It is functional interface*
* *Used* ***checking*** *purpose*
* ***Boolean test(T t)****-> it can* ***accept an y data type*** *but* ***returns Boolean*** *(Abstract method)*

*Interface*

package java.util.function;  
  
import java.util.Objects;  
@FunctionalInterface  
public interface Predicate<T> {

boolean test(T t);

default Predicate<T> and(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) && other.test(t);  
 }  
  
 default Predicate<T> negate() {  
 return (t) -> !test(t);  
 }  
  
 default Predicate<T> or(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) || other.test(t);  
 }  
  
 static <T> Predicate<T> isEqual(Object targetRef) {  
 return (null == targetRef)  
 ? Objects::*isNull* : object -> targetRef.equals(object);  
 }  
  
 @SuppressWarnings("unchecked")  
 static <T> Predicate<T> not(Predicate<? super T> target) {  
 Objects.*requireNonNull*(target);  
 return (Predicate<T>)target.negate();  
 }  
}

***Program Example 1.1***

public class BasicProgramming {  
 public static void main(String args[]) {  
 Predicate<Integer> predicate = (x) -> x % 2 == 0;  
 boolean isEven = predicate.test(4);  
 System.*out*.println("4 isEven?="+isEven);  
 }  
}

***O/P-*** *4 isEven?=true*

***Program Example 1.2***

public static void main(String args[]) {  
   
 Predicate<String> startsWithAlphabetA = x -> x.toLowerCase() .startsWith("a");  
 Predicate<String> endsWithAlphabet = x -> x.toLowerCase() .endsWith("t");  
 boolean ankitConditionCheck = startsWithAlphabetA.and(endsWithAlphabet).test("Ankit");  
 System.*out*.println(ankitConditionCheck);  
 }  
}

***O/P-*** *true*

***Functions***

* *It is functional interface*
* *Used* ***to do some work/operations*** *purpose*
* ***R apply (T t)****-> it can* ***accept an y data type*** *but* ***returns any data type****(Abstract method)*

package java.util.function;  
  
import java.util.Objects;  
  
@FunctionalInterface  
public interface Function<T, R> {  
  
 R apply(T t);  
  
 default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {  
 Objects.*requireNonNull*(before);  
 return (V v) -> apply(before.apply(v));  
 }  
  
 default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  
 Objects.*requireNonNull*(after);  
 return (T t) -> after.apply(apply(t));  
 }  
  
 static <T> Function<T, T> identity() {  
 return t -> t;  
 }  
}

***Program Example***

public class BasicProgramming {  
 public static void main(String args[]) {  
 //Eg 1  
 Function<Integer,Integer> doubleIt= x->x\*2;  
 Function<Integer,Integer> TripleIt= y->y\*3;  
 System.*out*.println("Add then "+doubleIt.andThen(TripleIt).apply(100));  
 System.*out*.println("Compose just changing operands position from addthen "+TripleIt.compose(doubleIt).apply(100));  
 Function<Integer,Integer> jackpot= Function.*identity*();  
 Integer indentityApply = jackpot.apply(5);  
 System.*out*.println("Identity "+indentityApply);  
  
 //Eg 2  
 Function<String,String> replace= x->x.replace("A","K");  
 Function<String,String> lowerCase=x->x.toLowerCase();  
 System.*out*.println(replace.andThen(lowerCase).apply("AnKIt"));  
 }  
}

***o/p***

*Add then 600*

*Compose just changing operands position from addthen 600*

*Identity 5*

*Knkit*

***Consumer***

* *It is functional interface*
* *Used* ***consume but not return*** *purpose*
* ***void accept (T t)****-> it can* ***accept an y data type*** *but* ***doent returns*** *(Abstract method)*

package java.util.function;  
  
import java.util.Objects;

@FunctionalInterface  
public interface Consumer<T> {  
  
 void accept(T t);  
  
 default Consumer<T> andThen(Consumer<? super T> after) {  
 Objects.*requireNonNull*(after);  
 return (T t) -> { accept(t); after.accept(t); };  
 }  
}

***Program Example 1.1***

public class BasicProgramming {  
 public static void main(String args[]) {  
 Consumer<Integer> printValue= x->System.*out*.println(x+6);  
 Consumer<String> stringValue= x->System.*out*.println(x+"Enjoy");  
 printValue.accept(5);  
 stringValue.accept("Jack");

}  
}

**O/p**

*11*

*JackEnjoy*

***Program Example 1.2***

public class BasicProgramming {  
 public static void main(String args[]) {

List <Integer> list= Arrays.asList(1,2,3,4,7,8);  
 Consumer <List <Integer>> listValues= x-> {  
 for(Integer k: x)  
 {System.out.print(" "+k);}  
 };  
 listValues.accept(list);  
 }  
}

**O/p**

**1 2 3 4 7 8**

***Supplier***

* *It is functional interface*
* ***void get() -> it doesn’t*** ***accept an y data type*** *but* ***returns something*** *(Abstract method)*

package java.util.function;  
  
@FunctionalInterface  
public interface Supplier<T> {  
  
 T get();  
}

***Program Example***

public class BasicProgramming {  
 public static void main(String args[]) {

Supplier<String> supplierTest = () -> "hello";  
 System.*out*.println(supplierTest.get());  
 }  
}

***O/p***

***hello***

***Combination of predicate, function, consumer and supplier in program***

public class BasicProgramming {  
 public static void main(String args[]) {

Predicate<Integer> preCombine = x -> x % 2 == 0;  
 Supplier<Integer> supCombine= ()-> 100;  
 Function<Integer,Integer> funCombine= x->x\*x;  
 Consumer <Integer> conCombine= x -> System.*out*.println(x+2);  
 if(preCombine.test(supCombine.get())){  
 conCombine.accept(funCombine.apply(supCombine.get()));  
  
 }  
 }  
}

***O/p***

***10002***

***Bi-Predicate,Bi-Consumer, Bi-Function***

***Program Example***

public class BasicProgramming {  
 public static void main(String args[]) {

// Bi-predicate & Bi-Consumer  
int k = 12;  
int j = 14;  
BiPredicate<Integer, Integer> pred = (x, y) -> (x + y) % 2 == 0;  
boolean isEven = pred.test(k, j);  
BiConsumer<Integer, Integer> biCon = (x, y) -> System.*out*.println("Summation of " + x + " and " + y + "is Even?= " + (isEven == true ? "Yes" : "No"));  
biCon.accept(k, j);

}

***o/p= Summation of 12 and 14is Even?= Yes***

***Bi Function***

*Unary operator interface extends function interface and Binary operators interface extends Bifunction interface*

* ***BiFuntion<Integer,Integer,Integer> bifunction=(x,y)->x\*y\*100****; can be declared as* ***BinaryOperator<Integer> bifunction=(x,y)-> x\*y\*100;***
* ***Funtion<Integer,Integer> function=(x)->x\*100****; can be declared as*

***UnaryOperator<Integer> bifunction=(x,y)-> x\*y\*100;***

***Unary operator nterface***

package java.util.function;  
  
@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
 static <T> UnaryOperator<T> identity() {  
 return t -> t;  
 }  
}

***Program Example***

package Streams.Lambdas;  
  
import java.util.function.Function;  
import java.util.function.UnaryOperator;

public class BasicProgramming {  
 public static void main(String args[]) {

//Function declared as Unary Operator  
 UnaryOperator<String> replace= x->x.replace("A","K");  
 Function<String,String> lowerCase=x->x.toLowerCase();  
 System.*out*.println(replace.andThen(lowerCase).apply("AnKIt"));  
 }  
}

***O/P-*** ***knkit***

***Binary Iterface***

package java.util.function;  
  
import java.util.Objects;  
import java.util.Comparator;  
  
@FunctionalInterface  
public interface BinaryOperator<T> extends BiFunction<T,T,T> {

public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator) {  
 Objects.*requireNonNull*(comparator);  
 return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;  
 }  
  
 public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) {  
 Objects.*requireNonNull*(comparator);  
 return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;  
 }  
}

package Streams.Lambdas;  
  
import java.util.function.BinaryOperator;  
  
public class BinaryOperatorTest {  
 public static void main(String args[]) {  
  
 BinaryOperator<Integer> BinOp = (x, y) -> (x + y) \* 200;  
 System.*out*.println(BinOp.apply(4,5));  
  
  
 }  
}

***0/P- 1800***

***Method Reference and Constructor reference with Example***

package Streams.StreamWithLambda;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class MethodAndConstructorReference {  
 public static void main(String[] args) {  
 // 1. Method Reference  
 List<String> list1 = Arrays.*asList*("A", "B", "c");  
// list1.forEach(x -> System.out.println(x));  
 list1.forEach(System.*out*::println);  
  
 // 2. Construtor reference  
 List<String> list2 = Arrays.*asList*("A", "B", "c");  
// List<Phone> phoneList=list1.stream().map(x-> new Phone(x)).collect(Collectors.toList());  
 List<Phone> phoneList=list2.stream().map(Phone::new).collect(Collectors.*toList*());  
  
 }  
}  
  
class Phone{  
 private String name;  
 public Phone(String name) {  
 this.name = name;  
 }  
}