Functional programming:

**package** java8features;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

**public** **class** Test1 {

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

Function<Integer, Integer> function = i -> i \* i;

System.***out***.println("The square of 4 is : " + function.apply(4));

System.***out***.println("The square of 5 is : " + function.apply(5));

Predicate<Integer> predicate = i -> i%2==0;

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

}

}

LAMBDA EXPRESSION:

@FunctionalInterface

**public** **interface** Demo {

**void** doStuff();

}

**public** **class** Test {

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

Demo demo = () -> {

System.***out***.println("The method is running!!");

};

demo.doStuff();

}

}

Example of Runnable interface:

**public** **class** ImplementingRunnableByLambda {

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

Runnable runnable = () -> {

**for** (**int** i = 1; i < 10; i++) {

System.***out***.println(i + "===" + Thread.*currentThread*().getName());

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

};

Runnable runnable2 = () -> {

**for** (**char** ch = 'a'; ch < 'z'; ch++) {

System.***out***.println(ch + "===" + Thread.*currentThread*().getName());

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

};

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

Thread thread2 = **new** Thread(runnable2);

thread.start();

thread2.start();

}

}

FOREACH ()🡪

**public** **class** ForEachRef {

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

List<String> valueList = Arrays.*asList*("C", "Java", "Python", "DevOps");

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

System.***out***.println("The values are : " + valueList.get(i));

}

System.***out***.println("--------------------------------------");

**for** (String string : valueList) {

System.***out***.println("The values are : " + string);

}

System.***out***.println("--------------------------------------");

valueList.forEach(val -> System.***out***.println("The values are :" + val));

}

}

**Method reference**

-- using the same implementation for a interface method from an existing logic present in a method. "::". Used specially for functional interface.

Types: static method reference or reference to static method

non static method or reference to non static method

constructor method reference or reference to constructor.

Method ref is internally using Lambda expression

Method reference(Static method)🡪

@FunctionalInterface

**interface** Reference {

**public** **void** stuff();

}

**public** **class** Demo {

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

System.***out***.println("Welcome to java 8 static ");

}

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

Reference reference = Demo:: *m1*;

reference.stuff();

}

}

Non-Static method reference:

@FunctionalInterface

**interface** Reference {

**public** **void** stuff();

}

**public** **class** Demo {

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

System.***out***.println("Welcome to java 8 static ");

}

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

Reference reference = **new** Demo():: m1;

reference.stuff();

}

}

Default constructor ref:

@FunctionalInterface

**interface** Reference {

**public** **void** stuff();

}

**public** **class** Demo {

**public** Demo() {

System.***out***.println("Parameterized constructor reference");

}

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

Reference reference = Demo::**new**;

reference.stuff();

}

}

Param constructor ref:

@FunctionalInterface

**interface** Reference {

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

}

**public** **class** Demo {

**public** Demo(**int** i) {

System.***out***.println("Parameterized constructor reference");

}

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

Reference reference = Demo::**new**;

reference.stuff(10);

}

}