**Usage of Double Colon (::) in Java 8**

It is used as a method reference by replacing lambda expression. **Double colon operator works well in all places whenever we implement functional interface methods**.

1. A static method (**ClassName::methodName**)
2. An instance method of a particular object (**instanceRef::methName**)
3. A super method of a particular object (**super::methName**)
4. An instance method of an arbitrary object of a particular type (**ClassName::methodName**)
5. A class constructor reference (ClassName::new)
6. An array constructor reference (**TypeName[]::new**)

**Using Object::new**

**public class** Animal {  
 **private** String **name**;  
  
 **public** Animal(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** String getType() {  
 **return "Mamal"**;  
 }  
}

**public interface** Zoo {  
 Animal getAnimal(String name);  
}

In the Test class main method, you can write using lambda like this.

**Using Double Colon (::)**

Zoo zoo = Animal::**new**;  
Animal animal = zoo.getAnimal(**"Hati"**);  
System.***out***.println(animal.getType());

**Zoo zoo = (value) -> new Animal(value);  
Animal animal = zoo.getAnimal("Hati");  
System.out.println(animal.getType());**

🡺

**Another example**

**public static void** m1() {  
 System.***out***.println(**"Method called "**);  
}

You can also write like this.

Runnable r1 = () -> *m1*();  
Runnable r2 = Test::*m1*;

**Another Example**

**public int** getSquare(**int** n) {  
 **return** n\*n;  
}

Function<Integer,Integer> fnVal = **new** Test()::getSquare;  
**int** val = fnVal.apply(5);  
System.***out***.println(val);

List<Employee> empList = List.*of*(new Employee("John", 23),  
 new Employee("Vidya", 27), new Employee("Romano",17));  
**List<Employee> filterList = empList.stream().filter(e -> Test1.*isGreaterThan20*(e))  
 .collect(Collectors.*toList*());**  
**filterList.forEach(e -> System.*out*.println(e));**

The above can be written as

List<Employee> empList = List.*of*(new Employee("John", 23),  
 new Employee("Vidya", 27), new Employee("Romano",17));  
**List<Employee> filterList = empList.stream().filter(Test1::*isGreaterThan20*)  
 .collect(Collectors.*toList*());**  
**filterList.forEach(System.*out*::println);**

**Interface with Single Abstract Method**

To understand lambda expression usage, follow the rules

**parameter -> expression 🡸 parameter can be blank like () -> m2()**

**(parameter1, parameter2) -> expression**

**(parameter1, parameter2) -> { code block } or body or any method**

@FunctionalInterface  
**public interface** Interface2 {  
 **void** doItWithParam(String s);  
}

@FunctionalInterface  
**public interface** Interface1 {  
 **void doIt**();  
}

How to define above interface methods using lambda expression.

We know that in case of Runnable interface, we define like this 🡺 **Runnable r = () -> m1();**

Similarly, we define the above functional interfaces like below.

**public class** Test1 {  
  
 **public void** m1() {  
 System.***out***.println("m1() method called ...");  
 }  
  
 **public void** m2(String s) {  
 System.***out***.println("m2() called with param : "+s);  
 }  
  
 **public void** check() {  
 **Interface1 i1 = () -> m1();  
 i1.doIt();**  
 **Interface2 i2 = (s) -> m2(s); 🡸 In case of parameter  
 i2.doItWithParam("abcd");**  
 }  
  
 **public static void** main(String[] args) {  
 **new** Test1().check();  
 }  
}

**public interface** Interface3 {  
 String passAndGet(String value);  
}

**public** String getUpperCase(String s) {  
 **return** s.toUpperCase();  
}  
  
**public void** check() {  
 Interface3 i3 = (value) -> getUpperCase(value);  
 String value = i3.passAndGet("abcd");  
 System.***out***.println(value);  
}

How to use lambda expressions for the implemented Functional Interface classes.

**public void** check1() {  
 **Interface1 i11 = () -> new InterfaceImpl1().doIt();**  
 i11.doIt();  
  
 **Interface1 i22 = () -> new SomeArbitraryClazz().perform();**  
 i11.doIt();  
  
 Interface2 i2 = (s) -> **new** InterfaceImpl2().doItWithParam(s);  
 i2.doItWithParam("abcd11");  
}

The above looks peculiar but correct.

**public class** SomeArbitraryClazz {  
  
 **public void** perform() {  
 System.***out***.println("Performing ...");  
 }  
}

It means while using lambda expression, follow the fundamental like the below.

**(with or without param) -> expression with logic or body or method or any instance method**

**public class** Test1 {

**public static int** getSquare(**int** value) {  
 **return** value\*value;  
 }

}

**Function<Integer,Integer> fn = (val) -> Test1.getSquare(val);**  
 **Function<Integer,Integer> fn = Test1::*getSquare*;**  
 **int** result = fn.apply(5);  
 System.***out***.println("Result: "+result);

**public interface** Interface3 {  
 **int** getValue(**int** value);  
}

Interface3 i3 = Test1::*getSquare*;  
result = i3.getValue(25);  
System.***out***.println("New Result: "+result);

In case of Class Instance

**public class** Fruit {  
  
 **private** String name;  
  
 **public** Fruit(String s) {  
 **this**.name = s;  
 }  
  
 **public** String getFruitName(String s) {  
 **return** s.toUpperCase();  
 }  
}

**public interface** IFruit {  
 Fruit getFruit(String s);  
}

**public void** check5() {  
// **IFruit fruit = (value) -> new Fruit(value);** **IFruit fruit = Fruit::new;** Fruit actualFruit = fruit.getFruit("WaterMelon");  
 System.***out***.println(actualFruit.getFruitName("WaterMelon"));  
 }

In case of Class instance, you need to consider the return type also.