**Java 8 New Features in Simple Way**

<https://capgemini.udemy.com/course/java-8-new-features-in-simple-way/>

1. **Lambda expressions**
2. **Functional interfaces**: Which can be used to call lambda expressions, Ex: I want to call lambda expression something or some type must be required that type is nothing but functional interface.
3. **Default methods in interfaces**: In java 1.8 version even, we can declare concreate methods also inside interface those methods are nothing but default methods.
4. **Static methods in interface**: Even we can declare static methods also inside interface.

These are two extra things are allowed inside interface form java 1.8 version onwards.

1. There are some **pre-defined functional interfaces** are there the standard pre-defined are:
2. **Predicate**
3. **Function**
4. **Consumer**
5. **Method reference and constructor reference by double colon (: :) operator**
6. **Stream API:** Using this we perform bulk operation of collection is very easy, manipulation the collection of data.
7. **Date and Time API:** This was introduced by joda.org these people wear introduced this APIs and it is also known as joda API

Why java changes in the java 1.8 feature.  
-> 1) To simplify programming  
 2) To utilize functional programming in java  
 3) To enable parallel programming in java   
So, my java program can run on multi-core processor overall performance by default is going to be improved these are the main important priorities of related to java 1.8 version.

1. **Lambda expressions**LISP is the first language which using the lambdas expression then various other languages like c, c++, c# .NET python etc. Then last finally java using a lambdas expression.

1) What are the benefits of using lambdas expression in java?  
 \* To enable the functional programming in java.  
 \* To write more readable, maintainable, and clean concise code.  
 \* To use APIs very easily and effectively.  
 \* To enable the parallel processing.

2) What is a lambda expression? How we can write lambdas expression.  
-> A lambda expression is an anonymous function; Anonymous function means it’s an empty  
 \* It is not having name, function without having any name  
 \* It is not having modifiers  
 \* Not having any return type  
Such type of anonymous function is nothing but lambdas expression.

We can write lambdas expression like this:  
**Ex 1**: For normal java function for lambdas function

Public void m1() { () -> {  
 s.o.pln(“Hello”); s.o.pln(“Hello”);  
 } }

**Ex 2**: How we can add sum for two int value  
 public void add (int a, int b) { (int a, in b) -> {   
 s.o.pln(a+b); s.o.pln(a+b);  
 } }

**Ex 3**: Using one int value or String value, write the lambda expression that return sum and String will provide length   
 the String.

Public int getLength (String s) { (String s) -> {  
 return s.length(); return s.length();  
 } }

3) Explain some more rules about the lambda’s expression?   
 **Ex 1**:  
 \* In the lambda expression body, if only one statement is there then curly braces are optional.  
\* If the body contains multiple statements, then we required to take curly braces.

Public void m1 () { () -> {s.o.pln (“Hello”); } () -> s.o.pln(“Hello”); s.o.pln(“Hello”); This is a single statement   
 } curly braces not required

**Ex 2**:  
 Sometimes based on context compiler will guess the types automatically this property is called **type inference**,

public void add (int a, int b) { (int a, int b) -> { s.o.pln(a+b); }   
 s.o.pln (a+b);   
 }  
 if the compiler will guess automatically the types based on the context then we can the type also

(a, b) -> {s.o.pln (a+b);

So, unnecessary things we are not required to write in the lambda expression so that more concise and readable the   
 code we can write using the lambda expression.

**Ex 3**: \* If I provide a String value, I have to get its length

Public int getLength (String s) { (String s) -> { return s.length();}  
 return s.length();   
 }   
 \* Based on the context if the compiler can guess the type automatically by type inference, then we can remove  
 the type also.

(s) -> return s.length();

\* Further simplification, anyway this method returns something based on situation compiler can be able to   
 guess, so that not required to return keyword.

(s) -> s.length();

\* My lambda expression contains on input value, if it is taking only one input value, only one parameter is  
 there then the parenthesis is also optional.

\* Finally, the most simplified lambda expression is:

s -> s.length();

4) What are the characteristics/properties of lambda expression?   
 \* Lambda expression can take any number of parameters  
 **ex**: () -> s.o.pln(“Hello”);  
 (a,b) -> s.o.pln(“a+b”);  
 s -> s.length();

\* If the multiple parameters are present then we separated with comma ,   
 **ex**: (a,b) -> s.o.pln(“a+b”);

\* If only one parameter is available then parenthesis   
 **ex**: (s) -> s.length(); s -> s.length();

\* Usually, we can specify the type of parameter if compiler expect the type based on context, then we can remove   
 type. [**type inference**]  
 **ex**: (int a, int b) -> s.o.pln(a+b); (a, b) -> s.o.pln(a+b);

\* Similar to method body, lambda expression body can contain any number of statements. If multiple statements are   
 there, then we should enclose within curly braces.  
 **ex**: () -> {  
 statement 1;  
 statement 2;  
 statement 3;  
 };  
 If the body contains only one statement the curly braces are optional  
 **ex**: () -> s.o.pln (“Hello”);

\* If lambda expression return something we are not required to use return keyword.  
 ex: s -> s.length();

These are the some of properties of lambdas expression.  
  
After writing the lambda expression, we have to call that lambda expression based on our requirement, lambda expression is just like a method, we can call that method based on our requirement, to call that lambda expression compulsory some concept must be required that concept is **Functional interface**.

1. **Functional interface:**Once we write the lambda expression, we invoke that lambda expression we required functional interface.

1) What is the Functional interface?  
-> If the interface contains **single abstract method (sam)**, which contains **exactly one abstract method** that interface is  
 called functional interface.   
 The purpose of functional interface is to invoke the lambda’s expression.

**Ex**: Runnable -> It contains only one method, that is run () method.  
 Callable -> It contains only one method, that is call () method.  
 ActionListener -> It contains only one method, that is actionPerformed () method.  
 Comparable -> It contains only one method, that is compareTo () method.

The rule is restriction is applicable only for abstract method only one abstract method, we can take any number of   
 default and static methods there is no problem.

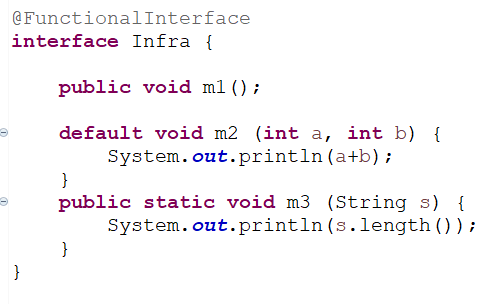
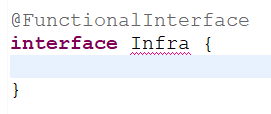
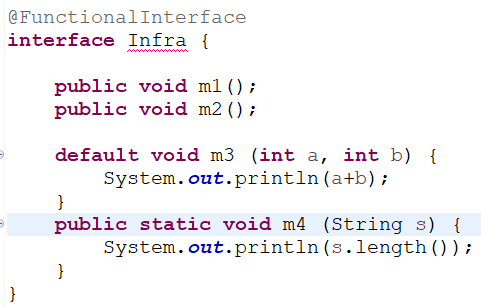
**Ex**: interface infra { interface infra {  
 public void m1 (); public void m1 ();  
 default void m2 () { public void m2 ();  
 } default void m3 () {   
 default void m3 () { }  
 } public static void m4 () {  
 public static void m4 () { }  
 }   
 public static void m5 () { This is not a functional interface, because, here having a two   
 } abstract methods here.  
 This is the functional interface we call it is as this normal interface. Not a functional interface.  
 restriction only for abstract   
 method.

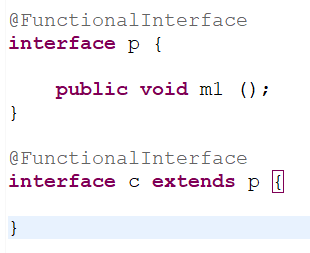
2) Why we use **@FunctionalInterface?  
 ->** To indicate that, this is the functional interface, java people introduced one special annotation in java 1.8 version is   
**@FunctionalInterface** If we use this annotation then convoying to the compiler saying. compiler intension is this is the functional interface, I want to declare this one is functional interface by mistake if I’m doing any mistake to indicate that we are using @**FunctionalInterface**.  
Weather we are using this annotation or not, if the interface contains a single abstract method is always functional interface only,   
but the advantages of this @**FunctionalInterface** if I’m doing any mistake compiler raise a error because our intension is this interface is functional interface.

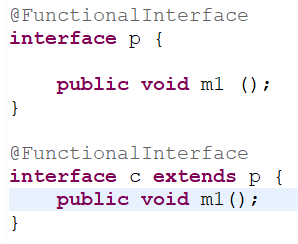
**Ex 1**: How we can use

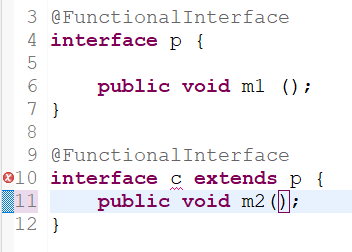
@FunctionalInterface @FunctionalInterface   
interface infra { interface infra {  
 public void m1 (); public void m1 ();  
 default void m2 () { public void m2 ();  
 } default void m3 () {   
 default void m3 () { }  
 } public static void m4 () {  
 public static void m4 () { }  
 } I’m declaring explicitly as @FunctionalInterface, when we   
 public static void m5 () { compile this code, we got a compile time error we are going to  
 } get. That error is   
Compiler will check how can abstract unexpected @FunctionalInterface annotation  
methods are there, only one abstract multiple non overriding abstract method present interface infra.   
method, then it perfect.  
So, if any person trying to add new abstract methods   
it cannot be allowed.

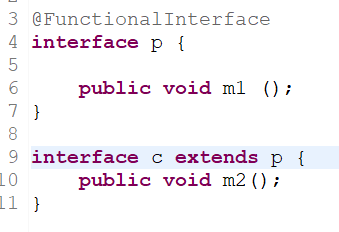
**Ex 2**:  
@FunctionalInterface   
interface infra {  
}

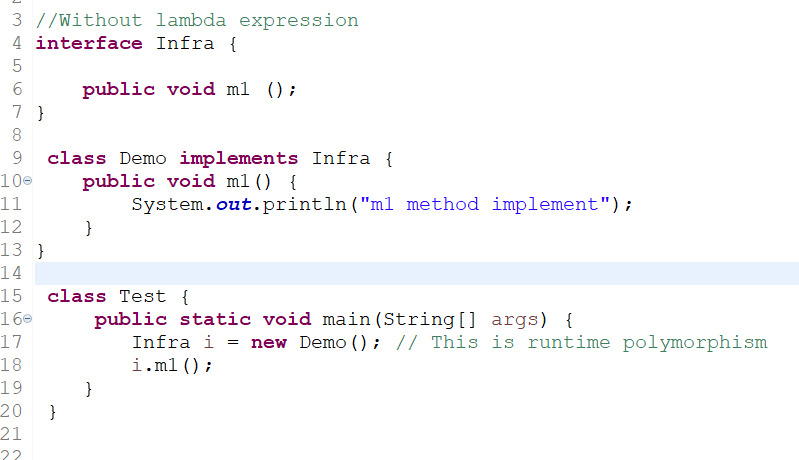
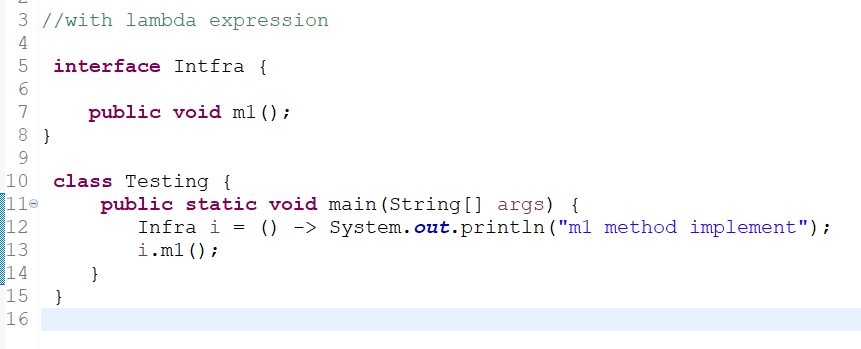
It will be invalid, because if the interface declared as a functional interface compulsory it should contain one abstract method.  
error is:  
unexpected @FunctionalInterface no abstract method found in interface infra.  


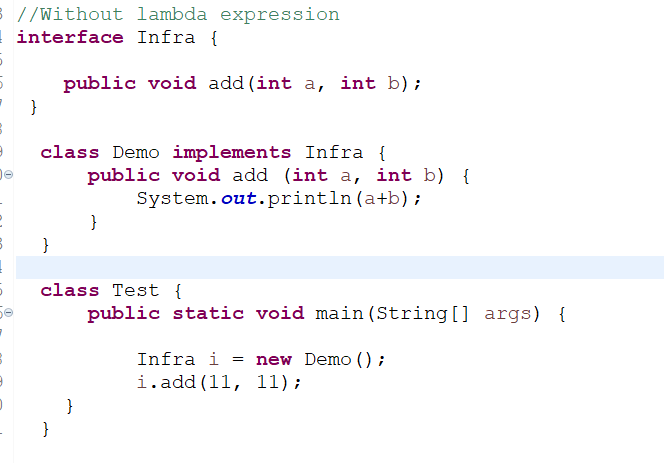
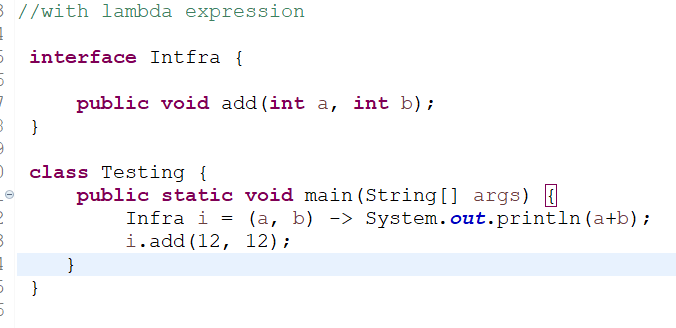
3) Functional interface with respect to interface?  
 **Case 1**: If an interface extends functional interface and child interface doesn’t contain any abstract method, then   
 child interface is always **Functional interface**. So, it is valid   
 **Ex**: 

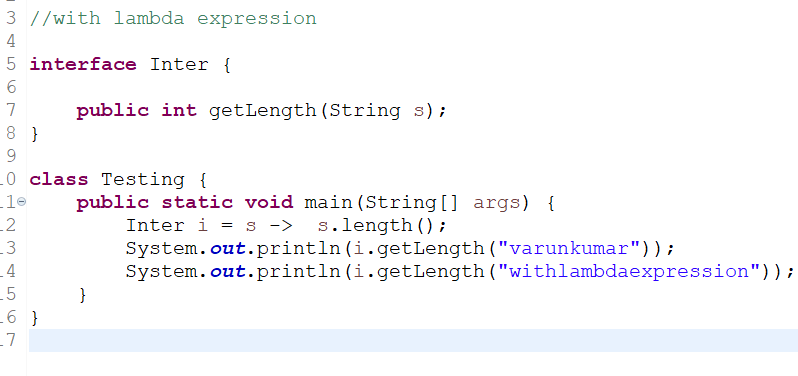
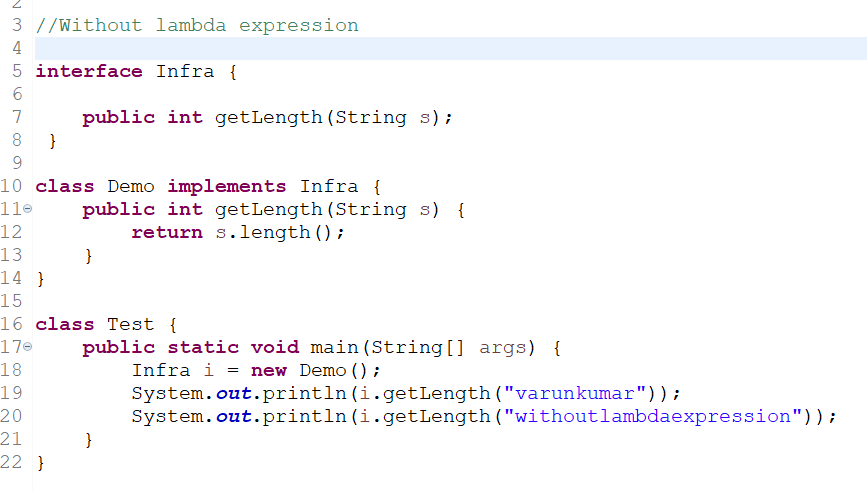
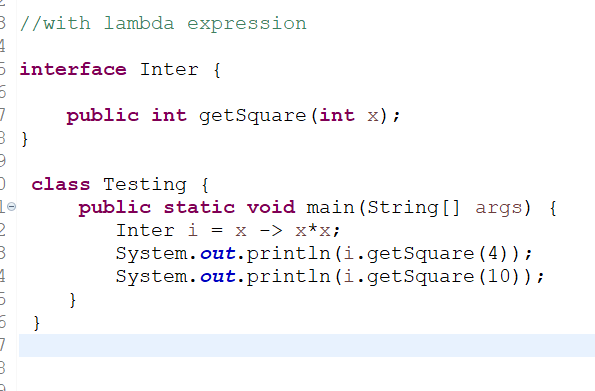
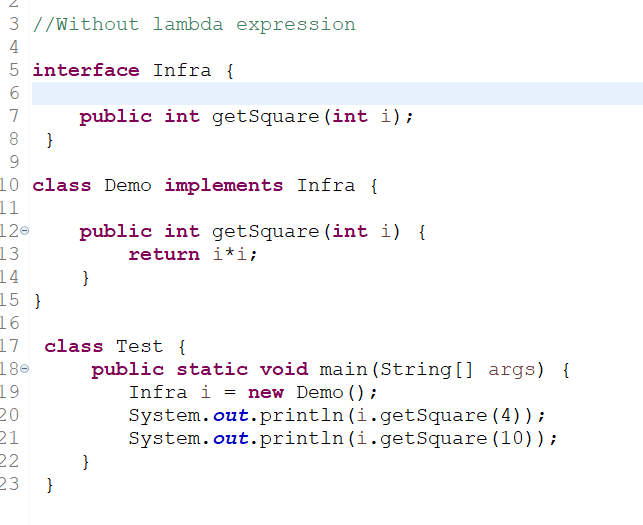
**Case 2**: In the child interface, we can define exactly same parent interface abstract method. There is not error  
 **Ex**: 

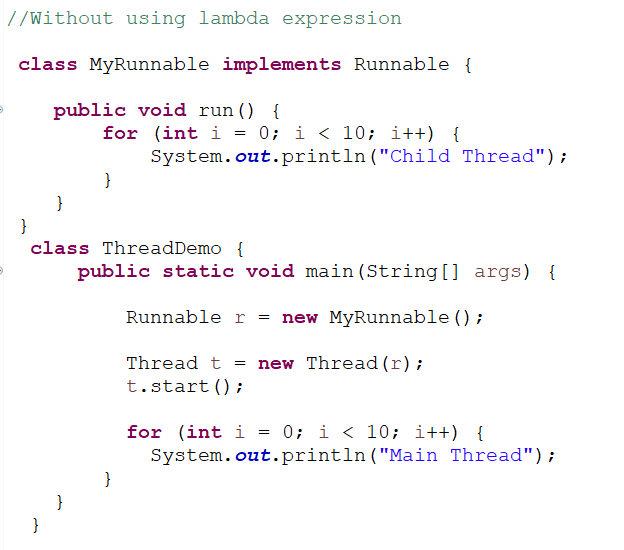
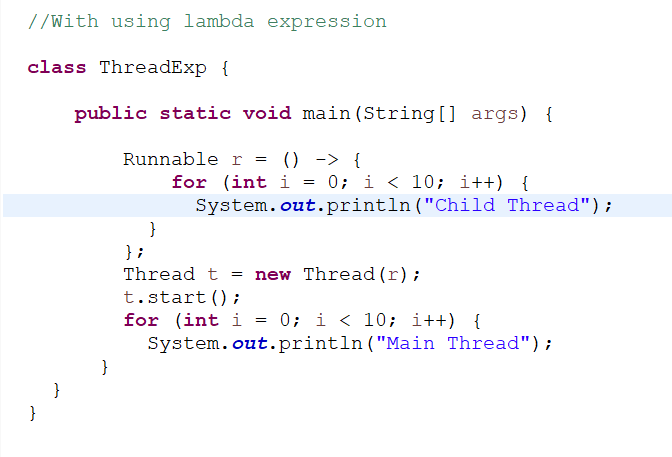
**Case 3**: In the child interface we can’t define any new abstract method, otherwise we will get compile time error.  
 **Ex**:  Unexpected @FunctionalInterface annotation.  
   
 Multiple non-overriding abstract methods found in   
  
 interface c.

**Case 4**: It is perfectly valid, because child interface contains two abstract methods, but still its valid, because of   
 I didn’t declare child interface as @FunctionalInterface.  
 **Ex**:    
default and static methods are no restriction in the parent or child interface, we can take any number.

4) Purpose of functional interface?  
 To invoke or call the lambda expression we should go for functional interface.

For the example above, in normal without lambda expression we required to write a separate class, we required to provide implementation separately.  
But if we use lambda expression, we are not required to write any separate class just we can total implementation  
Internally the name mapping is going to happen with the Intfra reference, once I write a lambdas expression to invoke its functionality, functional interface must be required. By using functional interface reference by allowed to call lambdas expression implementation.

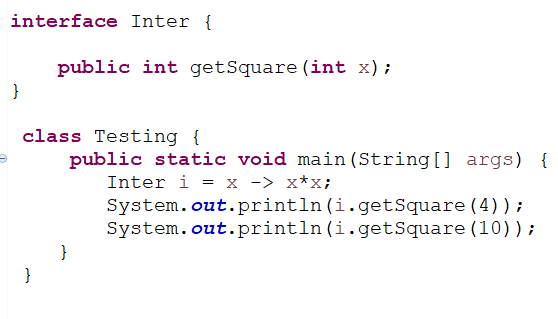
**Example 3**:  
**Example 4**:

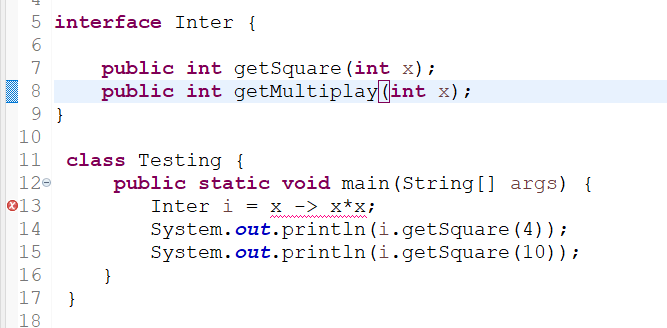
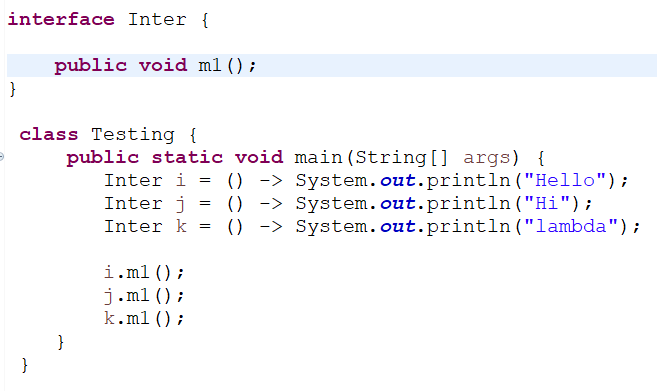
5) How can we use lambda expression with pre-defined functional interfaces. (More Important)  
 \* **Runnable**: Write a lambda expression for that implements runnable interface run () method.

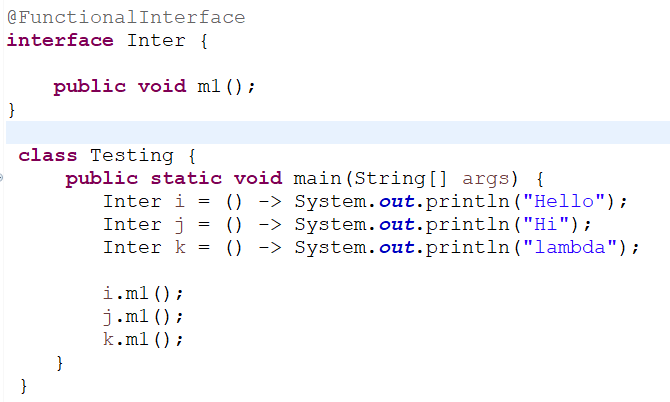
We can assign as a function; lambda expression is a function with a variables Runnable r = one function we are assigning as a value, such type of facility is there because of functional programming. Lambda expression concept came to enable and use functional programming features in java.

We don’t know exact out how we can use multi-threading?  
 -> In multi-threading, between the jobs there is no dependency, weather the jobs are all completed or not that’s what we required to check in multithreading.

6) Summary of the functional interface?  
 \* It should contain exactly one abstract method (SAM, single abstract method).  
 \* It can contain any number of static and default methods.  
 \* It acts a type for lambda expression  
 **Ex**: interface inter {}   
 inter I = () -> s.o.pln (“Hello”);   
 inter: So, inter is a type of lambda expression  
 () -> s.o.pln (“Hello”); : This is for lambda expression  
 \* To invoke a lambda expression functional interface must require.

**Case 1**: Why functional interface should contain only one abstract method?   
 \* Because the purpose of functional interface is to invoke lambda expression.  
 \* Lambda expression is mapped to some method of interface, so if interface contains multiple abstract method  
 then in the mapping there is a problem is going to come, that’s why the functional interface should contains   
 only one abstract method. More than on abstract method we can’t use lambda expression   
**Ex**:  
It got executed, because it has only one abstract method

For this scenario compiler will confuse for choosing the abstract method, that’s why in functional interface exactly one abstract method is available.  
Compilation error:  
incompatible type Inter is not a functional interface.  
multiple non-overriding abstract method in interface Inter.  
Showing like this compilation error.  
  
**Case 2**: What are the advantages of @FunctionalInterface annotation?  
\* All lambda expression which is internally mapped to m1 abstract method only.  
\* But by mistake tomorrow if a person came he is tyring to add another abstract method inside the interface, if any person adding the abstract method all these lambda expression won’t works immediately we get incompatible types error.   
\* So, I want to specify explicitly this interface is used for lambda’s expression, don’t add any new abstract method for that **@FunctionalInterface** concept came.

\* When seeing this annotation **@FunctionalInterface** everyone gets understand it is a functional interface it should contains only one abstract method not add any abstract method like we provide the information we have to use   
**@FunctionalInterface**This is the advantage of @FunctionalInterface  
It indicates as this interface is used for lambda expression purpose don’t add 2nd abstract method.

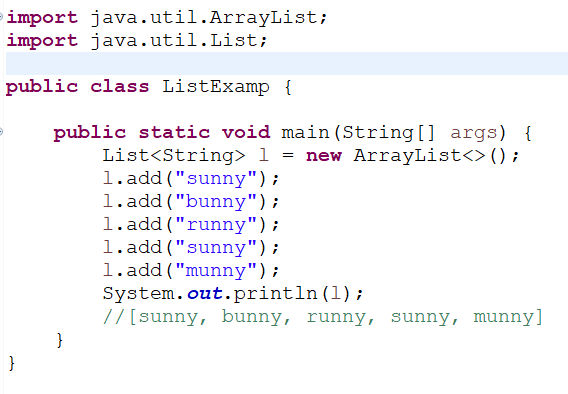
**How we can use lambda expressions for collection concepts:**

**Collection**:  
Collection is a group of objects represented as a single entity

**Types of collection**:  
1) List interface 2) Set interface 3) Map interface

1. **List interface**: When we use List?  
    \* If we want Insertion order  
    \* and if we want Duplicate objects, then we can choose a list.

We implement this list interface we are using a class as:   
1) Array List 2) Linked List and 3) Vector, vector child class is stack

**Ex**: 

1. **Set interface**: When we using set?

\* Duplicate element is not allowed.

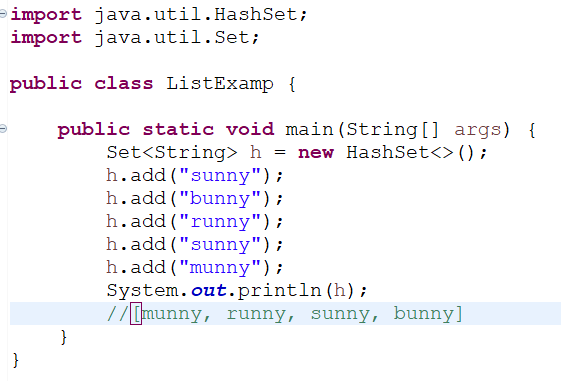
\* Not maintain an insertion order.

I don’t want about the insertion order in any order can save, not required for the insertion order but duplicate objects is don’t want this type of requirement happily we can go for set concept.

**Ex**: I want to send SMS message for all my colleagues, don’t worry about the order and duplicate number is not allowed.

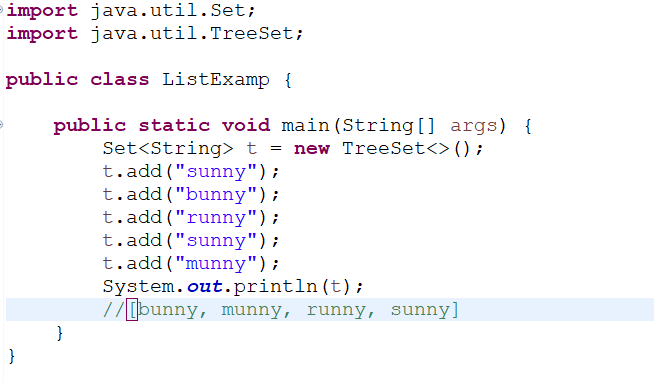
We implement this list interface we are using a class as:

1. **Hash set**: The object will be stored based on hash code we don’t know order.   
   it’s never worry about order in which order we want to get output we don’t know. But its not allowing duplicate entry.

**Ex**: 

1. **Tree set**: I want to be sorting for the set, we will go for tree set.

If we maintain the sorting order and not allowed the duplicate entry, then we choose a tree set

**Ex**: 

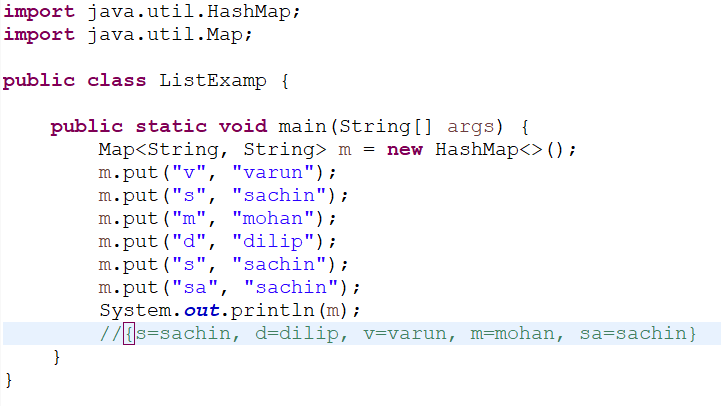
1. **Map Interface**: I want to represent a group of objects as key value pairs. Then we can go for Map concept.

Ex: I want to store a **Roll number -> Name** and **Mobile number -> Address.**

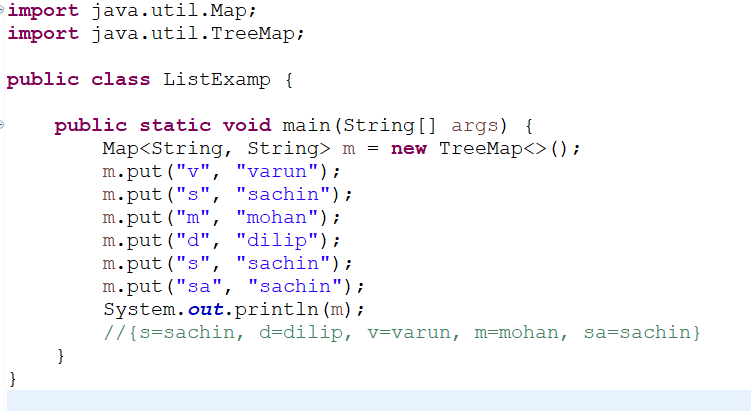
We implement this Map interface we are using a class as:

1. Hash Map 2) Tree Map  
     
   1) **Hash Map**:

\* It does not maintain an insertion order.  
 \* The key is stored as unique, does not allow duplicate  
 \* The value can access the duplicate.

**Ex**: 

2) **TreeMap**:   
 \* It has sorted an object ascending order.  
 \* The key is stored as unique, does not allow duplicate  
 \* The value can access the duplicate.

**Ex**: 

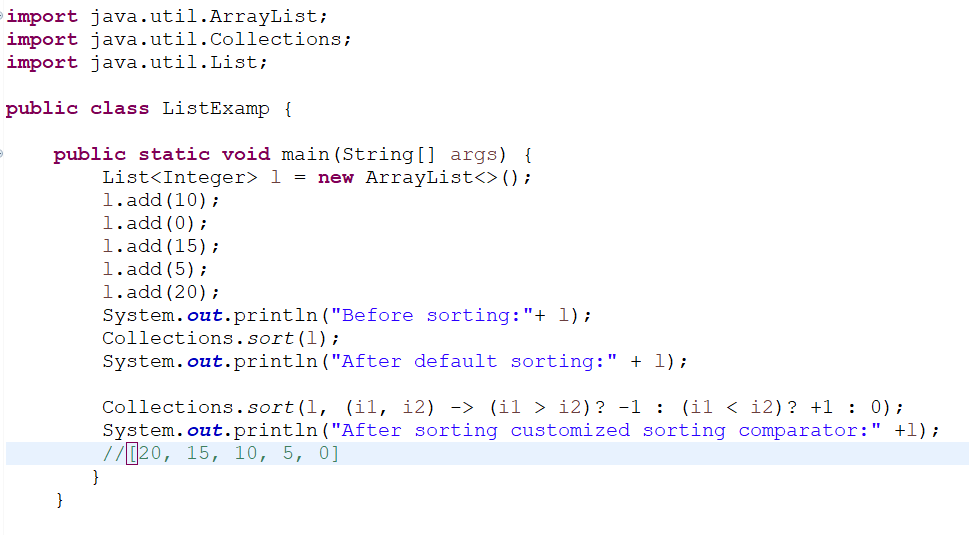
**Comparator Interface**:

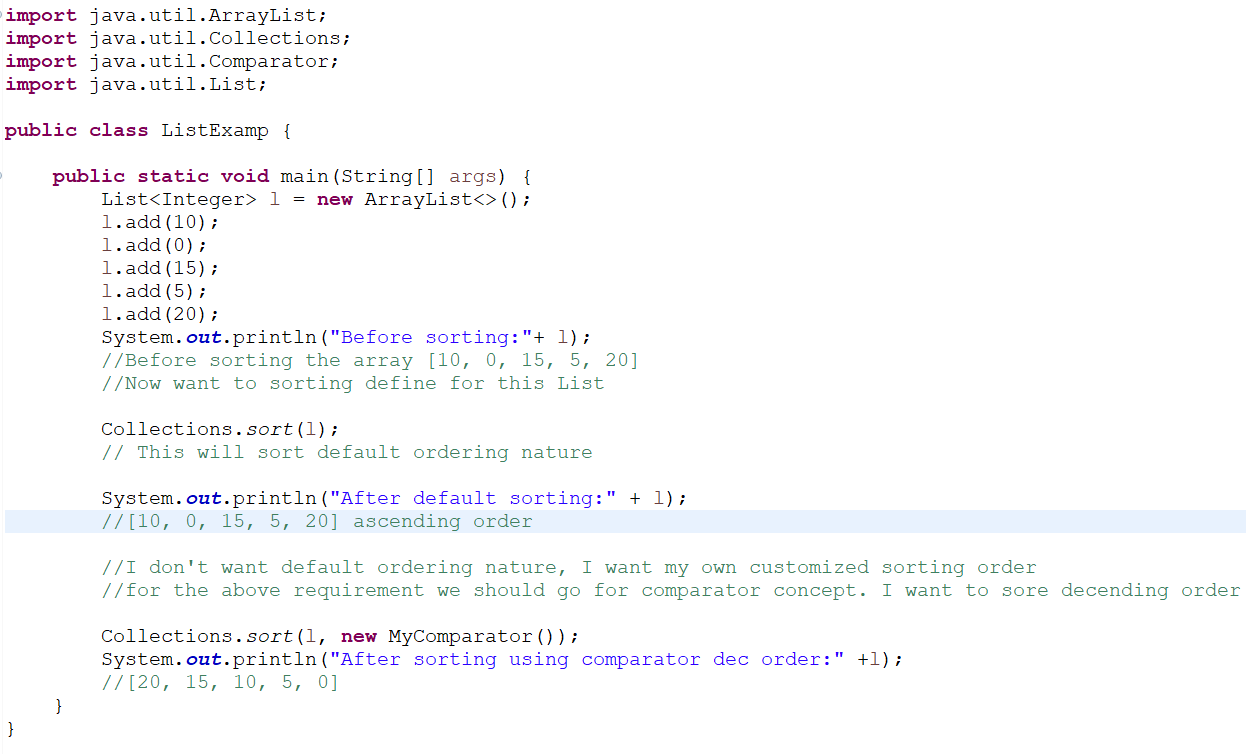
If wherever comparator is there happily our hand should go for lambda expression concept. Because the reason is comparator interface contain only one abstract method **compare ()**.

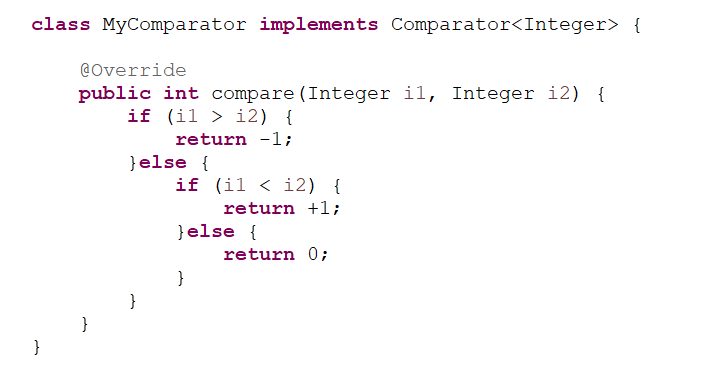
So, if any interface is containing single abstract method (SAM) is by considered as Functional interface. So wherever functional interface is there happily we can use lambdas expression concept.

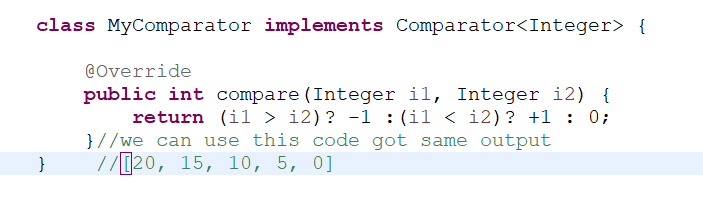
What is the use of comparator concept?  
-> This concept is to define our own sorting is nothing but customized sorting.   
 While defining sorting to place our objects in sorting order our JVM is always going to use comparator compare ()  
 method. If (obj1 = 10, obj2 = 20) JVM should decide internally wherever sorting is required the JVM is always to call   
 compare () method.   
 **Ex**: public int compare (obj1, obj2) {}  
 \* If obj1 as to come before obj2 according to our required sorting this comparator is going to return negative   
 value. (10, 20) [- ve]   
 \* If obj1 as to come after obj2 according to our required sorting this comparator is going to return positive   
 value. (20, 10) [+ ve]  
 \* if obj1 and obj2 are equal according to our required sorting this comparator is going to return zero (10, 10) [0].  
 \* For the List, Set and Map case wherever to define to sorting order internally we required to use comparator   
 JVM is always to call Comparator compare () method.   
 \* The return value of compare () methos is **int value**.

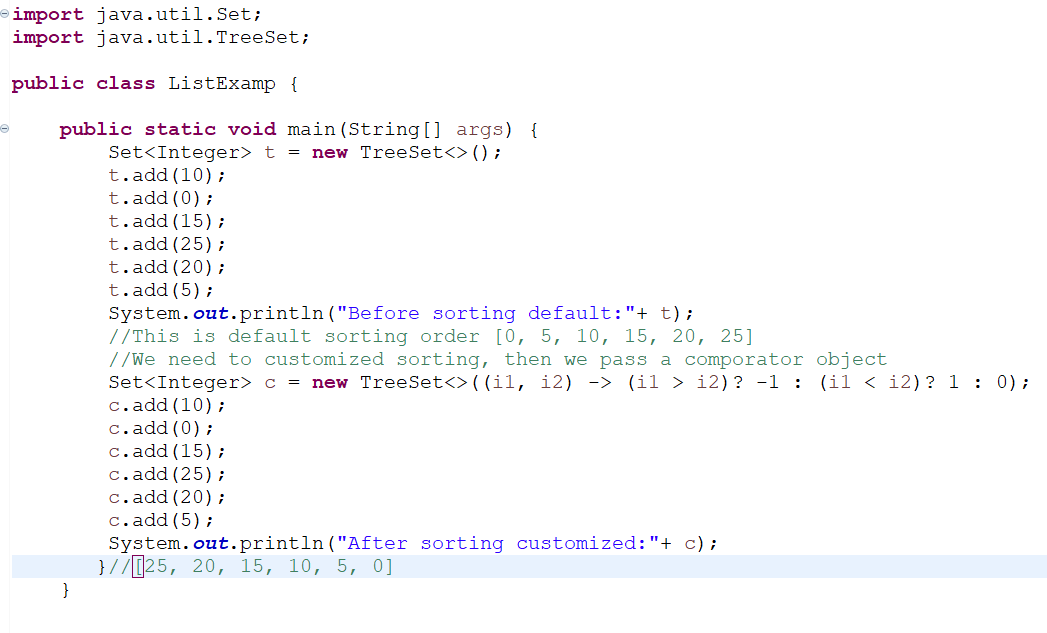
**Sorting the Array list by using comparator**.

Without lambda expression with lambda expression

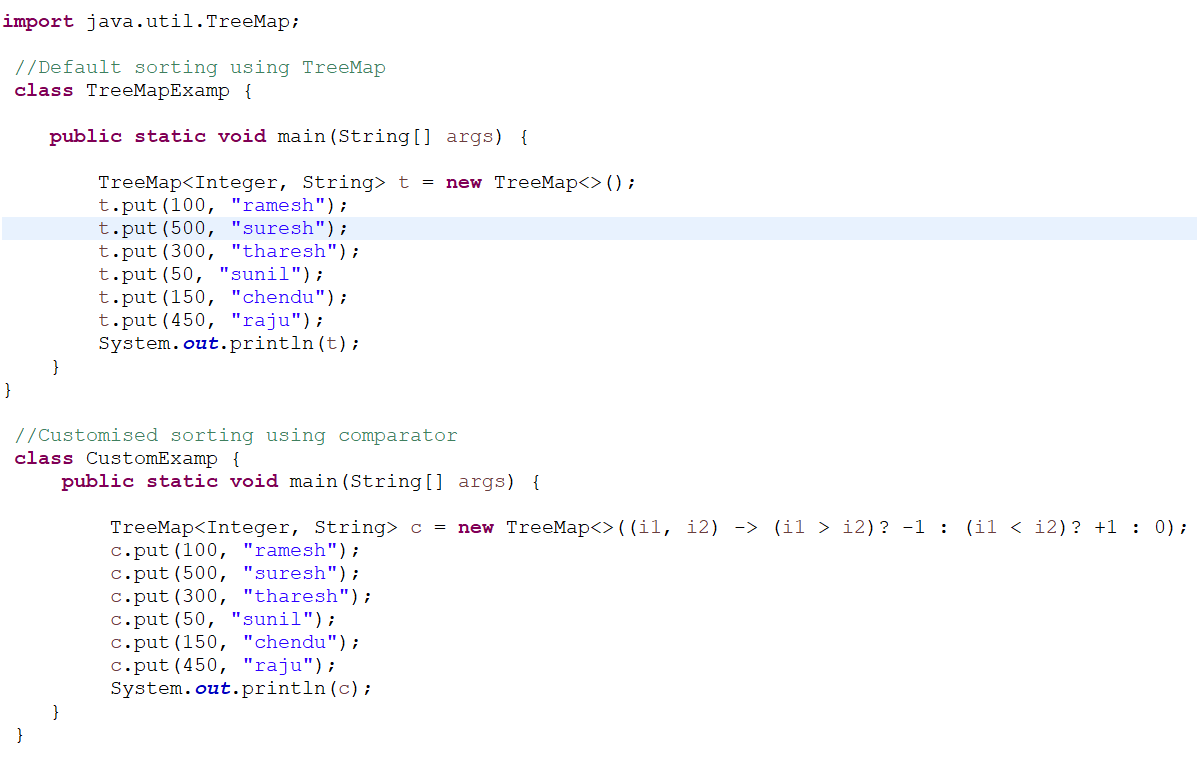




Using ternary operator, we can decrease the above code

**We want to use set concept in the case of set, if we want sorting order then we should go for tree set, if I create tree set object without argument.**   
**Set<Integer> t = new Treeset ();** this tree set is ment for default nature of sorting. Suppose taking  
**Set<Integer> t = new Treeset (Comparator c);** If pass the comparator object then it is ment for customized sorting order.  
**Ex**:

**We want to apply lambda expression concept for the Map, same lambda expression how we can define customised sorting for the tree map**

In the case Map if we want to be sorting then we go for TreeMap concept, tree map always meant for sorting order key values pair will be inserted according to sorting of key, sorting is always based on the key but not based on the value by default  


**Now using lambda expression then do a sorting, I want to ascending order to number  
**

**Anonymous inner classes and lambda:**

This lambda expression is related to very close relation with anonymous inner classes wherever anonymous inner class is there there may be a chance to replace with lambda expression. But not always there may be a possibility

Difference of anonymous inner classes and lambda expression?

anonymous inner classes are a nameless inner class is for default considered as anonymous inner classes. We can declare an inner class without using the name.

**Ex**: **Runnable r = new Runnable ();** This is a compile time error because in interface we cannot create an object.

This problem we are solving out we are using an anonymous inner class.

**Ex**: **Runnable r = new Runnable () {  
 public void run () {** compulsory we should provide implementation for run ()

**//we are implementing our logic here**

**}**

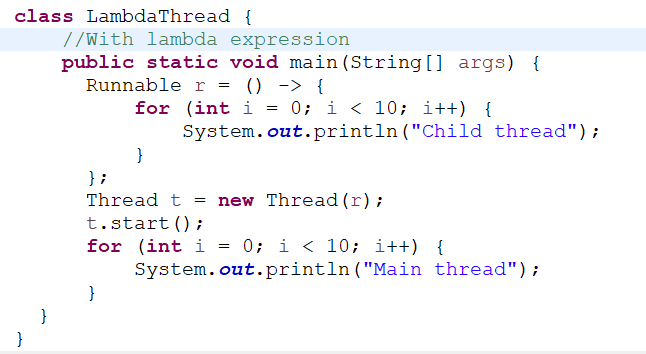
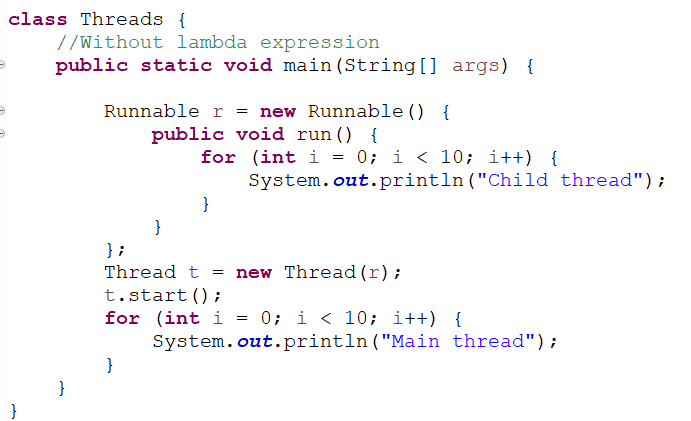
**};**

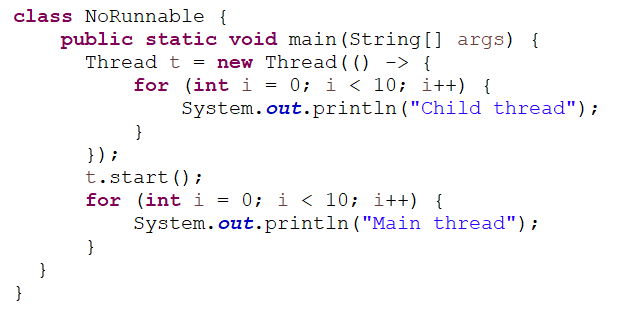
anonymous inner classes which implement Runnable interface for that implementation class creating an object. So, it is not a runnable object it is the object for its implementation class.

Now, wherever this symbol is there we can happily use the lambdas expression.

**Ex**:

Without lambda expression with lambda expression

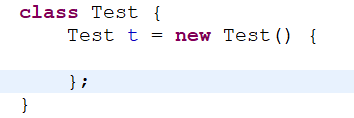
Without using a Runnable interface using lambda how can we write  
 **Ex**: 

**For every anonymous class can replace with lambda expression?**

-> No, for only sometimes, some anonymous function only we can replace with lambda expression, because lambda expressions and anonymous function both are not equal.  
Anonymous inner class function is more powerful than lambda expression.   
**For example**, where I cannot replace with anonymous function into lambda expression.

If interface contain only one abstract method, if the interface contains multiple methods, we cannot go for the lambda expression.

**Case 1**: Anonymous inner class can extends concreate class, lambda expression cannot extend the concreate class

**Ex**: 

**Case 2**: Anonymous inner class can extend abstract class   
**Ex**: **abstract** **class** Test {

}

Test t = **new** Test() {

};

**Case 3**: Anonymous inner class that implements an interface which contains multiple methods  
**Ex**: **interface** Test {

**public** **void** m1();

**public** **void** m2();

**public** **void** m3();

}

Test ***t*** = **new** Test ();

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

**public** **void** m2() {}

**public** **void** m3() {}

};

**Case 4**: Anonymous inner class that implements interface which contains only one abstract method.

For this anonymous inner class replaced with lambda expression

interface Test () { interface Test {

**public** void m1(); public void m1 ( );

} }

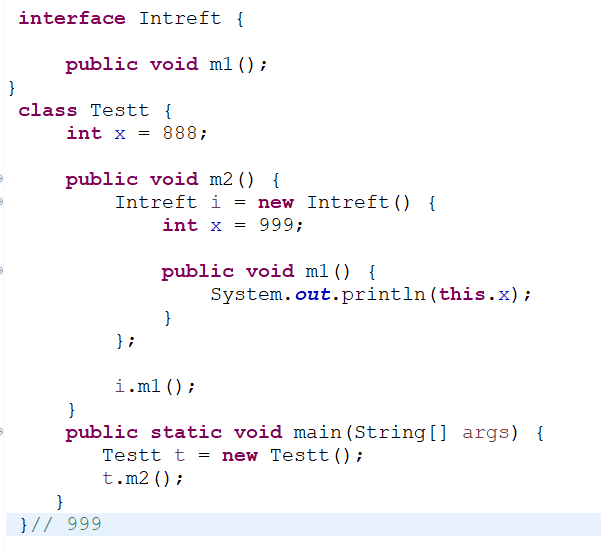
Test ***t*** = **new** Test() { Test t = () -> {

**public** **void** m1() { s.o.pln(“Hello”);

} }

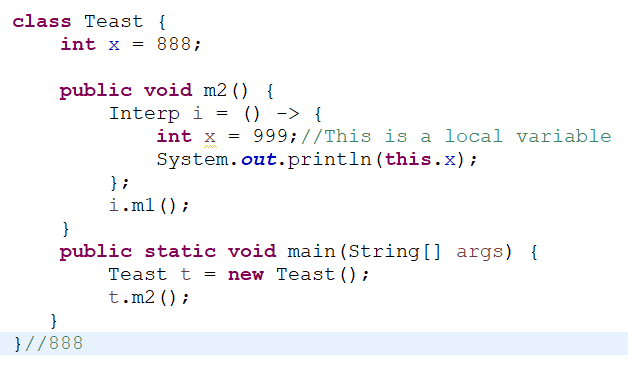
}; };

For using the above cases we are understanding, after we find out anonymous inner class and lambda’s expression is not same but compare to lambda’s expression anonymous inner class is most powerful.  
 **Anonymous inner classes ! = lambda expression**

**What is behaviour of this keywork to anonymous inner class and lambda expression:**

this keyword is always referred to current object.  
The behaviour of **this keyword** in anonymous class one way but in lambda expression another way. It is also making a significant difference between anonymous inner class and lambda expression.

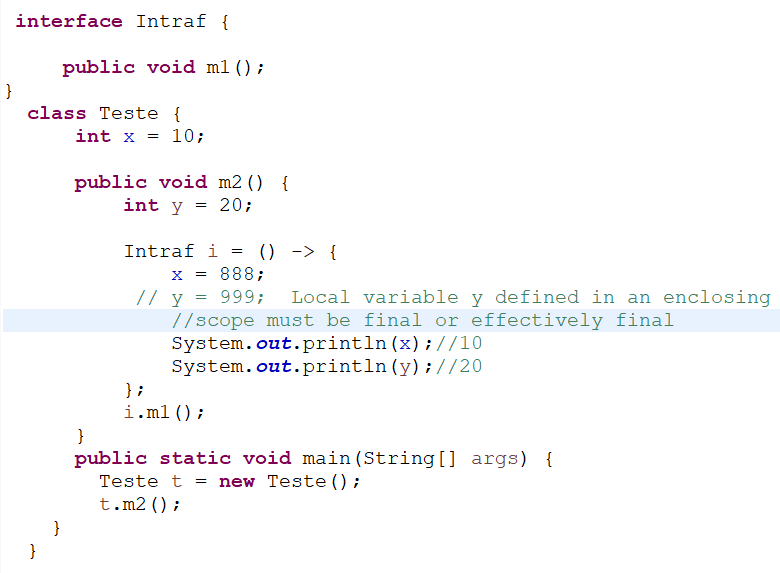
**Ex**: 1) For the **inside anonymous inner class** can we declare instance variable – yes, we can declare  
 2) Outer class contain **x** **instance variable** and inner class contains **instance variable x,** so inside anonymous inner   
 class this refers outer class object or inner class object.   
 -> Inside anonymous inner class **this keyword** always refers to current inner class variable only.

3) Inside **lambda expression** it is not possible to declare any instance variable, the variable which is declared inside   
 lambda expression is always access local variable only.  
 -> Inside lambda expression **this keyword** is always refers to outer class variables only.

**Difference between the anonymous inner class and lambda expression:**

**Anonymous inner class lambda expression**

\* It is a class without name \* It is a function without name (anonymous function)  
\* Anonymous inner class can extend abstract and concreate \* lambda expression cannot extend concreate and   
 class. abstract class.  
\* Anonymous inner class can implement an interface that \* lambda expression can implement an interface which   
 contains any number of abstract methods contains single abstract method (Functional interface).  
\* Inside anonymous inner class we can declare instance \* Inside lambda expression we can’t declare instance   
 variable. Variables. Whatever variables declared are considered   
 as local variables.  
\* Anonymous inner class can be instantiated, we can create \* lambda expression cannot be instantiated, we can’t   
 an object. Create an object.  
\* Inside anonymous inner class this keyword always refers \* Inside lambda expression this keyword is always refers to   
 current anonymous inner class object but not outer class current outer class object, that is called enclosing class  
 object. Object.  
\* Anonymous inner class is best choice if we want to handle \* lambda expression is best choice if we want to handle   
 multiple methods. Interface with single abstract method (SAM).  
\* For the anonymous inner class, at the time of compilation \* For the lambda expression at the time of compilation,  
 a separate class file will be generated. No separate class file will be generated.  
\* Memory will be allocated on demand whenever \* lambda expression will reside in permanent memory of   
 we are creating object. JVM (method area).   
 **So, the anonymous inner class and lambda is not equal**

**Important conclusion for lambda expression:**

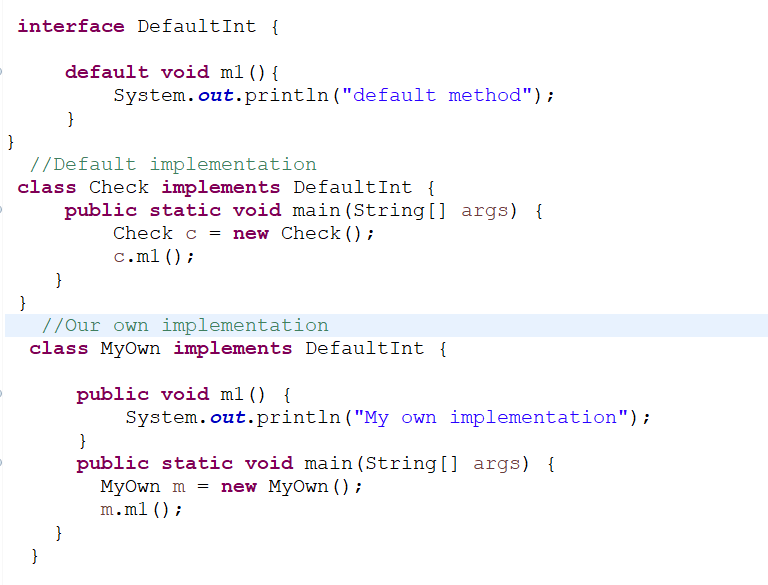
**Ex:**

\* From lambda expression we can access enclosing class variable and enclosing method variable directly there is no   
 problem.  
\* Inside lambda expression the local variable which are referenced or implicitly acts as final variable.   
\* The local variable referenced from lambda expression must be final or effectively final. Hence within lambda   
 expression or outside lambda expression we can’t change the local variable which are referenced from lambda   
 expression.

**Advantage of lambda expression**\* We can enable functional programming in java  
\* To reduce length of the code so that readability will be improved.  
\* We can resolve complexity of anonymous inner class until some extent.  
\* We can handle procedures/functions just like values.  
\* We can pass procedure/function as arguments.  
\* Easier to use updated Api’s and libraries.  
\* enable support for parallel processing.

**3) Default methods inside interface:**

\* From java 1.8 version onwards, we can declare concrete methods in interface in the form of default methods.   
\* With default keyword interface default methods by default available to the implementation class, implementation   
 directly can use the default methods, if the implementation class is not satisfied with default implementation, then  
 happily it is allowed to override its requirement. This is nothing but default methods.

**Ex**: 

**Default methods with respect to multiple inheritance:**

If the two interface contain the same default method and same signature, then implementation class there may be a chance of ambiguity problem.

How we can solve this problem.