ONSJHJSA

[1] Lambda expression: Lambda expression is the extension of the Anonymous class.

* Anonymous class: By definition, an anonymous class is the inner class without any name. In other words, a nested class without any name is called Anonymous class.
* Anonymous class is the implementation of the interface. We create the object of the interface and implement the interface methods at the time of creating the object. This is called the anonymous class.
* Let’s create one interface and create the method “showMessage ()”.

package Java8Features.AnonymousClass;  
  
public interface AnonymousInterface {  
 void showMessage(String message);   
}

* Now, traditionally if the class implements the interface then, the class needs to override/define the unimplemented method showMessage().

package Java8Features.AnonymousClass;  
  
public class MainClass implements AnonymousInterface{  
  
 public static void main(String[] args) {  
 MainClass object = new MainClass();  
  
 object.showMessage("Test Message");  
 }  
  
 @Override  
 public void showMessage(String message) {  
 System.*out*.println("The provided message == " + message);  
 }  
}

* Here, we created a class “MainClass” and implement the AnonymousInterface. Now, We have to define the unimplemented showMessage() method in the MainClass.
* To call the method, we can create the object of the class and call the method.

1). Anonymous class implementation:

* Here, we created the object of the interface and it will ask us to define/override the method.
* We can call the same method by using the interface object.

package Java8Features.AnonymousClass;  
  
public class MainClass{  
  
 public static void main(String[] args) {  
  
 AnonymousInterface interfaceObject = new AnonymousInterface() {  
 @Override  
 public void showMessage(String message) {  
 System.*out*.println("Anonymous interface creation == " + message);  
 }  
 };  
   
 interfaceObject.showMessage("Interface message");  
  
// MainClass object = new MainClass();  
// object.showMessage("Test Message");  
 }

[1.1] Functional Interface: ***An Interface that contains exactly one abstract method is known as functional interface.*** It can have any number of default and/or static methods but can contain only one abstract method. It can also declare methods of object class. Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces.

* In our current example, the AnonymousInterface is the functional interface as it has ONLY ONE abstract method. Thus, we can use “LAMBDA EXPRESSION” to define that method.

public static void main(String[] args) {  
  
 //SOL 2: USING LAMBDA EXPRESSION:  
 AnonymousInterface anonymousLambda = (str) -> { System.*out*.println("This is lambda expression == " + str); };  
 anonymousLambda.showMessage("Lambda message");  
  
 //SOL 1: USING INNER ANONYMOUS CLASS  
 AnonymousInterface interfaceObject = new AnonymousInterface() {  
 @Override  
 public void showMessage(String message) {  
 System.*out*.println("Anonymous interface creation == " + message);  
 }  
 };  
 interfaceObject.showMessage("Interface message");  
}

* In the above example, we define the ONLY ABSTRACT METHOD of functional interface using lambda expression. As the functional Interface has the only one abstract method, we can eliminate the need of mentioning the method name. We can define the method body directly.

[1.1.1] DEFAULT VS STATIC METHOD:

* + Default methods **can be** overridden in implementing class, while static **cannot**.
  + Static method belongs **only** to Interface class, so you can only invoke static method on Interface class, not on class implementing this Interface.

public interface MyInterface {

default void defaultMethod(){

System.out.println("Default");

}

static void staticMethod(){

System.out.println("Static");

}

}

public class MyClass implements MyInterface {

public static void main(String[] args) {

MyClass.staticMethod(); //not valid - static method may be invoked on containing interface class only

MyInterface.staticMethod(); //valid

}

}

* + Both class and interface **can have** static methods with same names, and neither overrides other!

public class MyClass implements MyInterface {

public static void main(String[] args) {

//both are valid and have different behaviour

MyClass.staticMethod();

MyInterface.staticMethod();

}

static void staticMethod(){

System.out.println("another static..");

}

}

[1.2] MORE METHODS ON LAMBDA EXPRESSION:

* Create a function interface with one abstract method, int sum(int a, int b). We can define the method using different approach.

@FunctionalInterface  
public interface functionalInterfaceCalc {  
  
 int sum(int a, int b);  
}

APPROACH 1:

//APPROACH 1:   
functionalInterfaceCalc obj = (int a, int b) -> { return (a + b); };  
System.*out*.println("Sum of two numbers = " + obj.sum(4,5));

APPROACH 2:

//APPROACH 2:   
functionalInterfaceCalc obj1 = (a,b) -> (a+b);  
System.*out*.println("No need to metion the datatype and return value as well!!!" + obj1.sum(5,8));

* In Lambda expression, we don’t need to specify the data type as we have a single method with the fixed data type: sum(int a, int b) -> No need to specify the data type. It is optional.
* Also, the return type of the method is fixed. No need to specify the return type as well. “return” keyword is optional.

[1.3] SORTING OF COLLECTIONS:

* Let’s create a “Employee” class with 2 fields employeeName and EmpSalary. We’d like to sort based on the EmployeeName.
* As there are 2 fields to sort. We need to specify the based on which field, we want to sort. We can use “COLLECTIONS” API class’s “SORT” method which requires the LIST object and COMPARATOR INTERFACE and override COMPARE METHOD and COMPARE\_TO.

APPROACH 1:

public class SortingOfCollectionUsingLambda {  
 public static void main(String[] args) {  
  
 List<EmployeeCollection> employeeCollectionList = Arrays.*asList*(  
 new EmployeeCollection("Sam",1000),  
 new EmployeeCollection("Anj", 2000),  
 new EmployeeCollection("Brij", 3000),  
 new EmployeeCollection("Carry", 500)  
 );  
  
 //METHOD 1: USING COLLECTIONS API && COMPARATOR INTERFACE  
 Comparator<EmployeeCollection> comparatorEmp = new Comparator<EmployeeCollection>() {  
 @Override  
 public int compare(EmployeeCollection e1, EmployeeCollection e2) {  
 return e1.getEmpName().compareTo(e2.getEmpName());  
 }  
 };  
  
 Collections.*sort*(employeeCollectionList, comparatorEmp);  
  
 //METHOD 2: USING LAMBDA FUNCTION  
 Comparator<EmployeeCollection> comparator = (EmployeeCollection e1, EmployeeCollection e2) -> {  
 return e1.getEmpName().compareTo(e2.getEmpName());  
 };  
   
 Collections.*sort*(employeeCollectionList, comparator);  
   
   
 System.*out*.println("Sorted Collection!!!!");  
 System.*out*.println(employeeCollectionList);  
 }  
}

[1.3.1] Method Reference OR Double Colon operator “ :: ”

* Another way to sort the collection is by using the Method Reference function.
* //METHOD 3: Method reference (Double colon :: )  
  Collections.*sort*(employeeCollectionList, Comparator.*comparing*(EmployeeCollection::getEmpSalary));  
    
  System.*out*.println("Sorted Collection by Method Reference OR Double Colon operator !!!!");  
  System.*out*.println(employeeCollectionList);

Here, we need to use “CLASSNAME :: METHODNAME” to call the method reference. Use, Collections.sort(LIST\_TO\_SORT, CLASSNAME :: METHOD\_TO\_SORT\_VALUES)

[2.0] JAVA STREAMS -> PREDICATE:

* Predicate is the functional interface in JAVA which has one method “test” with a single argument and return “Boolean”.
* Predicate is used to test whether the given object satisfy the condition. In simple terms, if I want to get all the result greater than 18 then, predicate will return TRUE for all the values over 18 and FALSE for all the values less than 18.

@FunctionalInterface

**public** **interface** Predicate<T> {

**boolean** test(T t);

}

* EX: I have list of ages, I want to filter only ages above 18. Predicate is the functional interface so I need to define the one abstract method TEST. Approach 1, let’s define predicate as the anonymous class.

List<Integer> age = List.*of*(20,29,40,20,49,19,10,8,6);  
  
Predicate<Integer> predicate = new Predicate<Integer>() {  
 @Override  
 public boolean test(Integer age) {  
 if(age > 18)  
 return true;  
 else  
 return false;  
 }  
};

Here, I created the object of the functional interface. Thus, I need to define the abstract test method.

for(Integer ageNumber : age){  
 if(predicate.test(ageNumber))  
 System.*out*.println("Age passed == " + ageNumber);  
}

Now, I iterate through all the list and if the number pass the Predicate test then it should print.

* Approach 2: Using the lambda expression to define the abstract method of the predicate. In Lambda expression, we need to define the filtering criteria for which the test should pass. In other terms, only the IF condition for which the test will return true.

List<Integer> age = List.*of*(20,29,40,20,49,19,10,8,6);  
  
Predicate<Integer> predicate = (Integer ageNumber) -> ageNumber > 18;