Java-8 Features:

* Java- 8 came in March-2018.
* Latest version of java is 15 came in September-2020.
* We have latest version of java but still the people talk about java-8. Still most of the project are using java-8 only because in other dependent technology like spring, Spring-boot, start using java-8 features. It could be possible after one or two year people going to use java-9 or 10 or latest version.
* The popularity of java came down after 2013, because that time some other programming languages came like Python, R, and Scala. In these languages if we writes 10 line of code for any functionality then in java we need to write 100 line of code for the same functionality. That is why the people start moving towards these languages, especially with python. After that java people start thinking what features we are lacking in java in compare to these languages. Hence java people introduce java-8 with new features.
* Most of the Java-8 features talk about concise code (very less code). And they achieved these just because of enabling functional or procedural programming in java. Till java-7, java was only object oriented programming language but from java-8 onwards it starts supporting functional programming language also.
* Because of this functional programming only, java-8 reduces several line of code and makes the code concise. Hence from java-8 onwards we can say that java-8 supports functional or procedural programming along with object oriented programming. Lambda Expression is the main feature came in java-8 to enable functional programming.

|  |  |
| --- | --- |
| package com.eight.features;  public class LambdaExpression {    public static int squareIt(int n) {  return n\*n;  }  Public static void main(String[] args) {  System.*out*.println("square of 5:"+*squareIt*(5);  }  } | package com.eight.features;  import java.util.function.\*;  public class LambdaExpression {  public static void main(String[] args) {  Function<Integer , Integer> f = i->i\*i;  System.*out*.println  ("Square of 5:-"+f.apply(5));  }  } |
| In the above program we have created one method to calculate square and the same method we are calling in main method | But here using Lambda expression we have done in two lines only.   1. Imported one package java.util.function.\*; 2. Using Function keyword and lambda expression we have written logic to calculate square. 3. Finally using apply method we are passing the argument to calculate square. |

Hence we can say that using functional programing and Lambda expression we can reduce the line of code.

|  |  |
| --- | --- |
| package com.eight.features;  import java.util.function.\*;  public class LambdaExpression {  public static void main(String[] args) {  Predicate<Integer>p=a->a%2==0;  System.*out*.println("The Prdicate is: "+p.test(4));  System.*out*.println("The Prdicate is: "+p.test(5));  }} | Predicate<Integer>p=a->a%2==0;  It returns boolean-valued function.  It has one functional method test (obj) to predicates that the value passed in the test method value is true or false. |

|  |
| --- |
| [Open Declaration](eclipse-open:%E2%98%82=Java-8-Features/C:\/Program%20Files\/Java\/jre1.8.0_191\/lib\/rt.jar%3cjava.util.function(Predicate.class%E2%98%83Predicate)[java](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util).[function](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util.function).Predicate<[Integer](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util.function(Predicate.class%E2%98%83Predicate%E2%98%82java.lang.Integer)>  @[FunctionalInterface](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.lang(FunctionalInterface.class%E2%98%83FunctionalInterface)  Represents a predicate (boolean-valued function) of one argument.  This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test (Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-).  Since: 1.8 |
| [Open Declaration](eclipse-open:%E2%98%82=Java-8-Features/C:\/Program%20Files\/Java\/jre1.8.0_191\/lib\/rt.jar%3cjava.util.function(Function.class%E2%98%83Function)[java](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util).[function](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util.function).Function<[Integer](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util.function(Function.class%E2%98%83Function%E2%98%82java.lang.Integer), [Integer](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.util.function(Function.class%E2%98%83Function%E2%98%82java.lang.Integer)> @[FunctionalInterface](eclipse-javadoc:%E2%98%82=Java-8-Features/C:%5C/Program%20Files%5C/Java%5C/jre1.8.0_191%5C/lib%5C/rt.jar%3Cjava.lang(FunctionalInterface.class%E2%98%83FunctionalInterface)  Represents a function that accepts one argument and produces a result.  This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [apply (Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-).  Since:1.8 |

Hence from the above discussion we have fallowing features of java-8.

1. Lambda Expression.(To enable functional programming in java)
2. Functional Interface.
3. Default methods or Static methods inside interface.
4. Predefined functional Interfaces.

* Function.
* Predicate.
* Consumer.
* Supplier.

1. Double Colon Operator (::)

* Method Reference.
* Constructor Reference.

1. Streams API. (To enable parallel processing in java)
2. Date and Time API.
3. Optional Class.
4. Nashron JavaScript Engine.

Lambda Expressions:

This Lambda expression concept came from the mathematics calculus which is being used in several calculations. Lambda expressions used to solve several difficult problems in the easiest way. This concept came in 1930 in the mathematics. After this slowly the computer programmer also start using this concept in the programming for solving big- big problem.

The first programming language which has used the Lambda expression is LISP.

Note: This Lambda Expression concept is not new in Java, because this concept is added very late in java. Some other programming languages like (Python, LISP, C, C++, Ruby, and Scala) already have this Lambda expression concept. So we cannot say that Lambda Expression is the unique concept in java. As Java language was strict to object oriented language only that is why Lambda expression concept was not possible but now java supports functional programing also. Because of this functional programming support Lambda expression is possible in java.

Hence the main objective of Lambda expressionis to bring benefits of functional programming and to make the java code concise into java language.

What is Lambda Expression?

* **It is an anonymous function. The term anonymous means nameless, without return type, without modifier. In other word any function without name is called anonymous function.**

How to use and write Lambda Expression?

|  |  |
| --- | --- |
| Public void m1() {  System.out.println("Hello");  }  The equivalent code in lambda expression. | () -> System.out.println("hello");  So as per definition   1. Method name (m1) is not mentioned in the Lambda expression. 2. No return type (not even void) is mentioned. 3. No any modifier (public) is mentioned. 4. Since here we have only one line of code so curly brace is not required. |
| Public void m1(int a, int b){  System.*out*.println(a+b);  } | (int a, int b)->{System.out.println(a+b);} OR  (a,b)->System.out.println(a+b); |
| Public int m1(int n){  return (n\*n);  } | (int n)->{return n\*n;} // OR  in curly brace return keyword is required to write  (int n)->n\*n; // OR  return keyword is not required when curly brace is not there  (n)->n\*n; // OR  n->n\*n; //   * So parenthesis is optionalif only one parameter is there. * Curly brace is optionalif only one line of code is there. * Parameter type and return keyword is not required everything is optional. |
| publicint m1(String s){  returns.length();  } | s ->s.length();  Note:   * Without curly braces we cannot use return keyword. Compiler will consider returned value automatically. * Within the curly braces if we want to return some value compulsory we should use return statement. |
| n->return n\*n; // invalid | Because without curly braces we cannot use return keyword |
| n->{return n\*n;}; // valid |  |
| n->{return n\*n};// invalid | Semicolon is missing for java statement. |
| n->{n\*n}; // invalid | Because if Within the curly braces if we want to return some value compulsory we should use return statement. |
| n->n\*n; // valid |  |

How to call the Lambda Expression?

* We will use one special word FI (Functional Interface).
* If we want to invoke or call the lambda expression then must to use Functional Interface.
* An interface which contains only one abstract method is called Functional Interface.  @FunctionalInterface annotation is added so that we can mark an interface as functional interface. If we don’t use @FunctionalInterface still the interface will be treated as Functional Interface @FunctionalInterface
* In Java we already have so many functional interfaces. E.g.
* Runnable is an FI which have only one abstract method run ().
* Comparable is an FI which have only one abstract method compareTo ().
* Comparator is an FI which have only one abstract method compare ().
* ActionListener is an FI which have only one abstract method actionPerformed ().
* Callable is an FI which have only one abstract method call ().

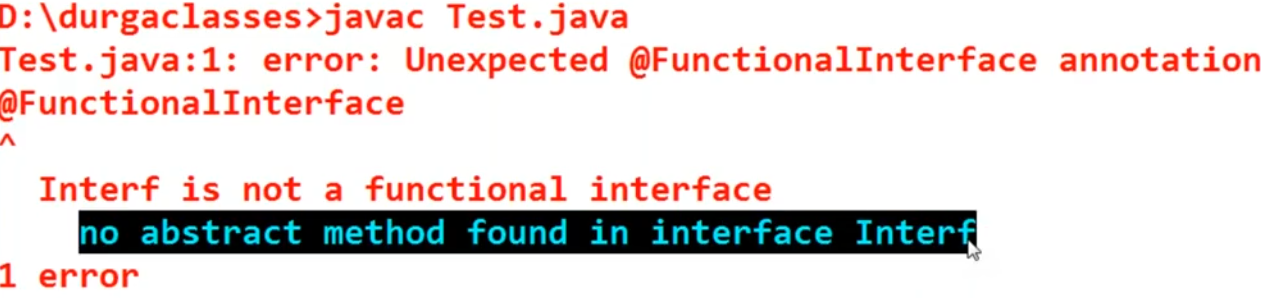
Note: Until java1.7, in interface every method is public and abstract by default whether we declare or not. And every variable is public static and final by default whether we declare or not. We cannot define any concrete method inside interface until java1.7.

But from java 1.8 onwards we can define concrete method also which is called default method. We use default keyword to define default method inside interface.

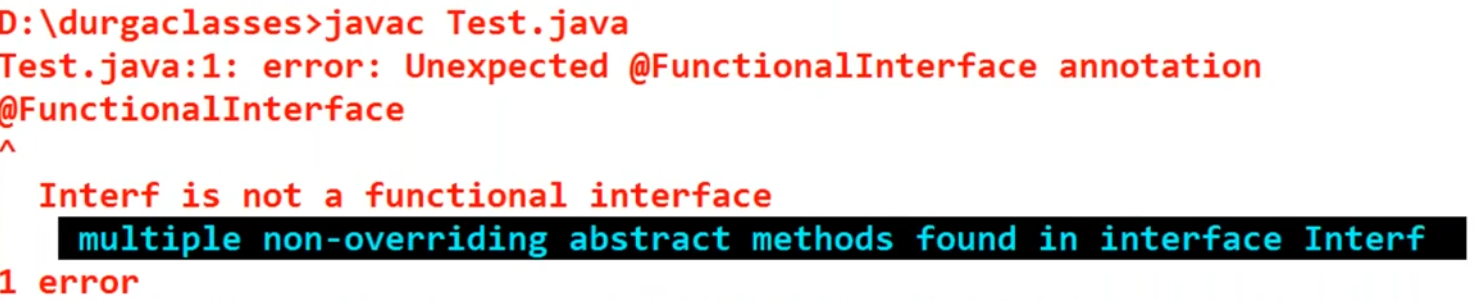
In Functional Interface we can have any number of default and static method but we cannot have more than one abstract method.

|  |  |
| --- | --- |
| @FunctionalInterface  Interface interf{  public void m1();  default void m2() {  // Code  }  Public static void m3() {  // Code  }  } | 1. Here we have created one functional interface and mark it using @FunctionalInterface. 2. Here we have exactly one abstract method m1 (). 3. Along with abstract method m1 () we have created one default method and static method inside the functional interface. 4. @FunctionalInterface is optional but recommended to use so that it can verify and validate that the interface is functional interface. |

Note-1: Let’s say if don’t have any abstract method inside functional interface then we will get fallowing exception.



Note-2: Let’s say if we declare more than one abstract method inside functional interface then we will get fallowing exception.



Note-3Let’s say we have one functional interface A and we have another functional interface B which extends A.

|  |  |
| --- | --- |
| @FunctionalInterface  interface A{  public void m1();  defaultvoid m2() {  // Code  }  publicstaticvoid m3() {  // Code  }  } | @FunctionalInterface  interface B extends A{    }  Here FI B extends FI A so whatever abstract method is available in A, automatically will be available in B also. So this is 100% valid FI. But if we declare any abstract method inside FI B then we will get compiler error saying.  Invalid '@FunctionalInterface' annotation;  EquivalentCode.B is not a functional interface |

Note-4: Let’s say if have declared the same abstract method in FI B what we have declared In FI A i.e. overriding the same abstract method in FI B. This case is valid and we will not get any compile time error.

|  |  |
| --- | --- |
| @FunctionalInterface  interface B extends A{  public void m1();  }  This case is Valid as we have override same abstract method. | @FunctionalInterface  interface B extends A{  publicvoid m2();  }  This case is invalid as we have declared one more abstract method. So including A there will be two abstract method which is not correct.  interface B extends A{  publicvoid m2();  }  This case is Valid as we this is not a FI. This normal Interface which can have multiple abstract method. |

Now linking the functional Interface with Lambda Expression.

|  |  |
| --- | --- |
| package com.eight.features;  interface intrf{  publicvoid m1();  }  Public class WithoutLambdaE implements intrf{  @override  Public void m1() {  System.*out*.println("Without Lambda Expression");  }  } | package com.eight.features;  public class Test {  public static void main(String[] args) {    WithoutLambdaE fi = new WithoutLambdaE ();  fi.m1();  intrf i = new WithoutLambdaE();  i.m1();  }  }  Output:  Without Lambda Expression  Without Lambda Expression |

The above example is without lambda expression. Here we have just

* Created one functional interface (intrf).
* Implement this interface in its implementation class (WithoutLambdaE).
* Finally created one object of WithoutLambdaE class and called its implemented method.

So the above example is a normal steps what we have done till java 1.7

Now implementing above example using functional interface and lambda expression.

|  |  |
| --- | --- |
| package com.eight.features;  interface intrf{  public void m1();  } | package com.eight.features;  public class Test {  public static void main(String[] args) {  intrf i2 = ()->System.*out*.println("With Lambda Expression");  i2.m1 (); // Calling the FI implemented method.  }  }  Output: With Lambda Expression |

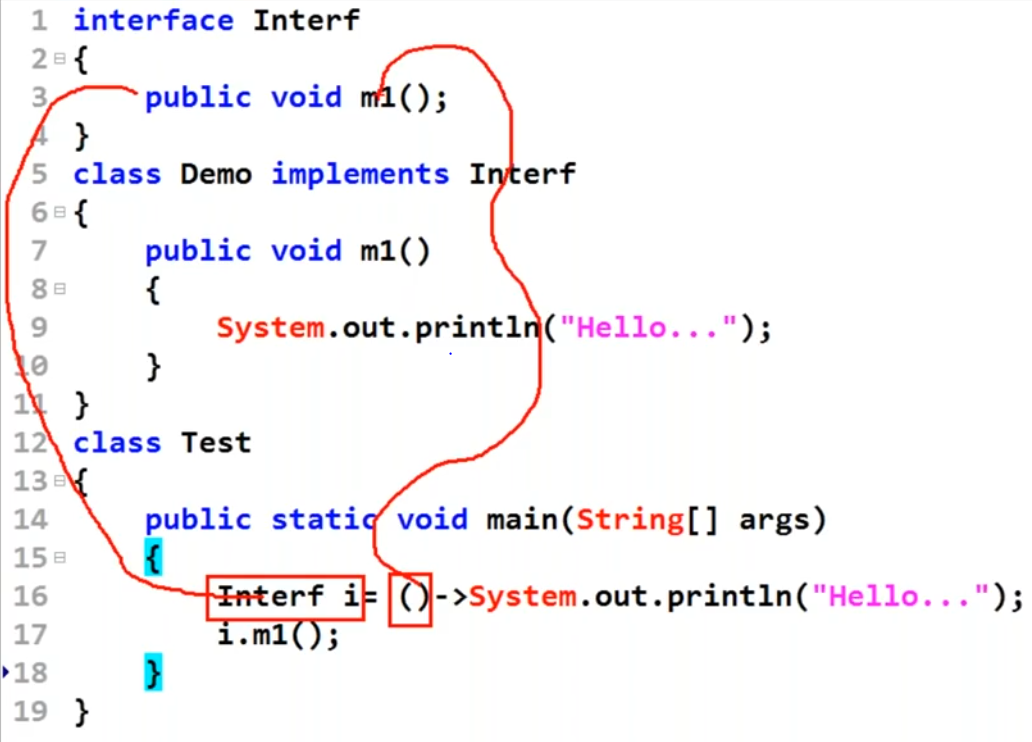
The above example is with lambda expression. Here we have just

* Created one functional interface (intrf).
* In the main method we have implemented FI abstract method m1 using Lambda Expression as shown above. Here in this case the method (m1) that we have implemented in interface’s implementation class WithoutLambdaE is replaced with

Intrf i2 = ()->System.out.println ("With Lambda Expression");

Here- to provide implementation for lambda expression FI reference [Intrf i2] is required.

* Here () is the argument of abstract method of FI m1 and [System.out.println ("With Lambda Expression");] is the implementation of m1. Since in FI we have only one abstract method so compiler automatically understand with FI reference (intrf) that the argument () and implementation is for FI abstract method m1 only.
* So every time when we do i2.m1(); it will print [With Lambda Expression]
* Hence using lambda expression we have reduced length of code.



This is the way to link Lambda expression with Functional Interface.

Example two: Having two argument:

Without Lambda Expression:

|  |  |
| --- | --- |
| @FunctionalInterface  interface TwoNumAdd{  public void AddNum(inta , intb);  }  class AddTwoNumbers implements TwoNumAdd{  @Override  publicvoid AddNum(int a, int b) {  System.*out*.println("Sum is :"+(a+b));  }  } | packagecom.eight.features;  public class Test {  public static void main(String[] args) {  AddTwoNumbers addn= new AddTwoNumbers();  OR  TwoNumAdd addn= new AddTwoNumbers();  addn.AddNum(50, 30);  }  } |

Here we have not used lambda expression so.

1. First we have created child class (implementation class) to define abstract method.
2. In the main method we created object of child class or parent reference of FI and using object of child class or parent reference of FI we called the child class method and passed two integer number to be added

Without Lambda Expression:

|  |  |
| --- | --- |
| @FunctionalInterface  interface TwoNumAdd{  public void AddNum(inta,intb);  } | packagecom.eight.features;  public class Test {  public static void main(String[] args) {  TwoNumAdd add=(int a,int b)->System.*out*.println("Sum of two is:-"+(a+b));  add.AddNum(20, 30);  add.AddNum(200, 300);  add.AddNum(2000, 3000);  }  } |
| public void AddNum(inta,intb) {  System.*out*.println("Sum is :"+(a+b)  } | Equivalent lambda expression of LHS method.  TwoNumAddadd=  (inta,intb)->System.*out*.println("Sum of two is:-"+(a+b));  OR  (a, b)->System.*out*.println("Sum of two is:-"+(a+b));  add.AddNum(20, 30); |

TwoNumAddadd= (a, b) ->System.*out*.println ("Sum of two is :-"+( a+b));

1. TwoNumAddadd 🡺 represents the reference of functional interface.
2. (a, b) ->System.*out*.println ("Sum of two is :-"+( a+b));🡺 Represents the implementation body of abstract method (AddNum) declared inside FI.
3. (a, b) 🡺 represents the argument of abstract method declared inside FI. Since in FI we have exactly one abstract class so whenever we use lambda expression the compiler automatically understands that this argument is for abstract class declared inside FI. Hence mentioning parameter type is optional.
4. add.AddNum (20, 30);🡺 Represents that what body we have defined using lambda expression is being called using FI reference.
5. So here so here we saw that we don’t need to create any child class to implement abstract method. We can implement FI abstract method using lambda expression as shown in the code. Hence we can reduce length of code and can increase the performance.
6. Hence FI is required to invoke lambda expression. In other words Lambda expression concepts applicable only for FI abstract method, without FI we cannot use lambda expression.
7. Lambda expression can be used everywhere, we just need to know how to use.
8. Lambda expression does not convert into .class file. At the time of compilation it gets converted into private method but never going to convert into .class file.

Lambda Expression in Multithreaded Environment: Without Lambda Expression:

|  |  |
| --- | --- |
| package lambda.ex.in.multithred.env;  public class MyRunnable implements Runnable {  @Override  publicvoid run() {  for(inti=0;i<5;i++) {  System.*out*.println("Child Thread");  }  }  }  Here we have created one implementation class MyRunnable to implement Runnable functional Interface abstract method run() | package lambda.ex.in.multithred.env;  publicclass TestThread {  public static void main(String[] args) {  MyRunnable r = new MyRunnable();  Thread t = new Thread(r);  t.start(); // Calls the run() method  for(int i=0;i<5;i++) {  System.*out*.println("Main thread");  }  }  }  Here in the main method we have created object of implementation class MyRunnable and then call the run method using t.start() |

With Lambda Expression:

|  |  |
| --- | --- |
| package lambda.ex.in.multithred.env;  public class TestThread {  public static void main(String[] args) {    Runnable r = ()->{  for(inti=0;i<5;i++) {  System.*out*.println("Child Thread using Lambda Expre");  }  };  Thread t = new Thread(r);  t.start();  for(inti=0;i<5;i++) {  System.*out*.println("Main thread");  }  }  } | Here we have completely replaced MyRunnable implement class and run () method with Lambda expression.  Here using main thread we created Runnable(r) interface reference and using this reference r we created Thread class object and using Thread class object we started lambda expression implementation. |

* Here We have one main thread using which we used to create Object of MyRunnable class and we creates object of Thread class and using Thread class object use start the child thread defined in the MyRunnable class. Hence after [t.start ()] there we have two threads (main thread and child thread).
* These two threads will execute simultaneously, hence the output will also be different, means sometime main class thread will execute first and sometime child class thread execute first.
* Here Runnable is Functional Interface because it contains only one abstract method: run (). Hence wherever FI is there happily we can apply Lambda expression. Hence for this Runnable Interface we don't need to create any Implementation class [MyRunnable class].In other word MyRunnable class implementation can be replaced with Lambda expression.

|  |
| --- |
| Runnable r = ()-> {for(int i=0; i<5; i++){  System.out.println("Child thread");  }  };  Ouput:  Main thread  Child Thread using LamdaExp  Child Thread using LamdaExp  Child Thread using LamdaExp  Child Thread using LamdaExp  Child Thread using LamdaExp  Main thread  Main thread  Main thread  Main thread |

Lambda Expression in collection:

Without Lambda Expression:

|  |  |
| --- | --- |
| package lambda.ex.in.multithred.env;  import java.util.Comparator;  publicclass MyComparator implements Comparator<Integer> {  @Override  publicint compare(Integer a1,Integer a2) {    return (a1>a2)?-1:(a1<a2)?1:0;  // OR  if(a1>a2) {  return -1;  }elseif(a1<a2) {  return -2;  }else {  return 0;  }  }  } | package lambda.ex.in.multithred.env;  import java.util.ArrayList;  import java.util.Collections;  importjava.util.Comparator;  publicclass TestCompare {  publicstaticvoid main(String[] args) {    ArrayList<Integer>al =new ArrayList<Integer>();  al.add(10);  al.add(5);  al.add(12);  al.add(3);  al.add(11);  al.add(15);  System.*out*.println(al);  Collections.*sort*(al);  System.*out*.println(al);  Collections.*sort*(al,new MyComparator());  }  }  Before Sorting :[10, 5, 12, 3, 11, 15]  After Sorting : [3, 5, 10, 11, 12, 15] |

Here for customized sorting we have created one class MyComparator which implements Comparator Interface. This Interface contains exactly one abstract method (compare) which will have to override and will write the logic for comparing two objects.

If obj1>obj2 then return -1: Means obj1 will come before obj2.

If obj 1<obj 2 then return +1: Means a1 will come after obj 2.

If obj 1==obj 2 then return 0: Means obj 1 and obj 2 will be equal.

Here in this example in the main method we created an ArrayList and to sort this ArrayList we have used Collections.sort (al, new MyComparator ()) method and in this method we have to passed object of ArrayList and MyCompare class object.

With Lambda Expression:

|  |  |
| --- | --- |
| package lambda.ex.in.multithred.env;  import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;  publicclass TestCompare {  publicstaticvoid main(String[] args) {    ArrayList<Integer>al =new ArrayList<Integer>();  al.add(10);  al.add(5);  al.add(12);  al.add(3);  al.add(11);  al.add(15);  System.*out*.println(al);  Collections.*sort*(al);  System.*out*.println(al);  // Lambda Expression without ternary operator  Comparator<Integer> c=(a1,a2)->{  if(a1>a2)  {  return -1;  }  elseif(a1<a2)  {  return -2;  }  else  {  return 0;  }  };  Collections.*sort*(al,c);  System.*out*.println(al);  // OR Lambda Expression and ternary operator  Comparator<Integer> c=(a1,a2)->a1>a2?-1:a1<a2?+1:0;  Collections.*sort*(al,c);  System.*out*.println(al);  System.*out*.println("Iterate List using forEach loop");  al.stream().forEach(System.*out*::println);  System.*out*.println("Even Number logic is”);  List<Integer>l2=  al.stream().filter(i->i%2==0).collect(Collectors.*toList*());  System.*out*.println("Even Number is:"+l2);  l2.stream().forEach(System.*out*::println);  }  } | If a1>a2 then return -1: Means a1 will come before a2.  If a1<a2 then return +1: Means a1 will come after a2.  If a1==a2 then return 0: Means a1 and a2 will be equal.  Output:  [15, 12, 11, 10, 5, 3]  Lambda[3, 5, 10, 11, 12, 15]  Iterate List using forEach loop  3  5  10  11  12  15  Even Number is:[10, 12]  10  12 |

The above example is about ArrayList having only one Integer type object, where we have applied customized sorting order. Here instead of sorting login in Comparator implementation class we have written the sorting / compare logic in the same main method using Lambda Expression.

Now let’s take an example where have an Employee class having empNo, empName. Then we have created one ArrayList object and added empNo and empName in this object.

|  |  |
| --- | --- |
| Employee.java:  package lambda.ex.in.multithred.env;  public class Employee {    public int empNo;  public String empName;  public Employee(intempNo, String empName){  this.empNo=empNo;  this.empName=empName;  }  // This is the default method of String to print object into string.  Public String toString() {  return empName+":"+empNo;  }  } | publicclass MyComparator implements Comparator<Employee> {  @Override  publicint compare(Employee e1, Employee e2) {  /\*String str1 = e1.empName.toString();  String str2 = e2.empName.toString();\*/  return(e2.empName.compareTo(e1.empName));  //return (e1.empNo<e2.empNo)?-1:(e1.empNo>e2.empNo)?1:0;  }  }  While using lambda expression, then we need not to write above class separately for sorting purpose. Same logic can be implemented in main class only. |
| packagecom.eight.features;  import java.util.ArrayList;  importjava.util.Collections;  importjava.util.Iterator;  importjava.util.List;  importjava.util.stream.Collector;  importjava.util.stream.Collectors;  importlambda.ex.in.multithred.env.Employee;  importlambda.ex.in.multithred.env.MyComparator;  publicclassSortEmpList {  public static void main(String[] args) {    ArrayList<Employee>el = new ArrayList<Employee>();  el.add(new Employee(1000,"Arun"));  el.add(new Employee(1005,"Bunty"));  el.add(new Employee(1002,"Tarun"));  el.add(new Employee(1003,"Junty"));  el.add(new Employee(1004,"Shilpi"));  el.add(new Employee(1011,"Priyanka"));  System.*out*.println(el);  //Collections.sort(el,newMyComparator());  //Collections.sort(el,(e1,e2)->(e1.empNo<e2.empNo)?-1:(e1.empNo>e2.empNo)?1:0);  Collections.*sort*(el,(e1,e2)->e1.empName.compareTo(e2.empName));  System.*out*.println("Dec"+el);  /\*Iterator it = el.iterator();  while(it.hasNext()) {  System.out.println(it.next());  }\*/  el.stream().forEach(System.*out*::println);  Listl2= el.stream().distinct().collect(Collectors.*toList*());  System.*out*.println("-----"+l2);  }  } | |

|  |
| --- |
| [Arun:1000, Bunty:1005, Tarun:1002, Junty:1003, Shilpi:1004, Priyanka:1011]  Dec[Arun:1000, Bunty:1005, Junty:1003, Priyanka:1011, Shilpi:1004, Tarun:1002]  Arun:1000  Bunty:1005  Junty:1003  Priyanka:1011  Shilpi:1004  Tarun:1002  -----[Arun:1000, Bunty:1005, Junty:1003, Priyanka:1011, Shilpi:1004, Tarun:1002] |

Note: al.stream ().forEach (System.*out*:: println);

The above line is used to iterate ArrayList object. Here we have used forEach introduced in java1.8

Note:

|  |  |
| --- | --- |
| (e1,e2)->(e1.empNo<e2.empNo)?-1:(e1.empNo>e2.empNo)?1:0 | Sorting ascending order of employee number |
| (e1,e2)->e1.empName.compareTo(e2.empName) | Sorting alphabetical order of employee name. |
| l2.stream().forEach(System.*out*::println); | Iterating list object without using Iterator and by using forEach. |
| List<Integer>l2=  al.stream().filter(i->i%2==0).collect(Collectors.*toList*()); | Getting the even value from the list and converting them in another list. |

.

Anonymous inner Class Vs Lambda Expression:

Note: Lambda expression is not the replacement of Anonymous inner classes, it is more powerful than Lambda Expression.

Q: If Employee class has employee id and salary then how to find second highest salary based on employee id:

A: First of all the sort the employee salary in descending order then the using second index of list we can get second highest salary.

|  |  |
| --- | --- |
| package com.employee.salary;  public class Employee {  public int empId;  public String name;  public double salary;  public Employee(int empId, String name, double salary) {  super();  this.empId = empId;  this.name = name;  this.salary = salary;  }  @Override  public String toString() {  return "Employee [salary=" + salary + "]";  } | package com.employee.salary;  import java.util.Comparator;  public class CompareSalary implements Comparator<Employee> {  @Override  public int compare(Employee e1, Employee e2) {  return (e1.salary > e2.salary) ? -1 : (e1.salary < e2.salary) ? 1 : 0;  //return (e1.empId > e2.empId) ? -1 : (e1.empId < e2.empId) ? 1 : 0;  //return (e1.name.compareTo(e2.name));  }  }  The above class object CompareSalary will be passed in sort method to get the decreasing order of salary. |

Main method

|  |  |
| --- | --- |
| package com.employee.salary;  import java.util.ArrayList;  import java.util.Collections;  import java.util.Iterator;  import java.util.List;  import java.util.stream.Collectors;  public class MainClass {  public static void main(String[] args) {    ArrayList<Employee> emp = new ArrayList<Employee>();  emp.add(new Employee(1000, "Arun", 12));  emp.add(new Employee(1001, "Arun", 13));  emp.add(new Employee(1002, "Arun", 14));  emp.add(new Employee(1003, "Arun", 15));  emp.add(new Employee(1003, "Arun", 16));  emp.add(new Employee(1003, "Arun", 16.5));  emp.add(new Employee(1003, "Arun", 17));  // emp.add(new Employee(1003, "Arun", 20));  // Collections.sort(emp, new CompareSalary());  // Logic to sort in decreasing order with Lambda  Collections.sort(emp, (e1, e2) -> (e1.salary > e2.salary) ? -1 : (e1.salary < e2.salary) ? 1 : 0); | System.out.println(emp);  System.out.println("Second biggest number1--" + emp.get(1).salary);  // Alternate Logic to get second largest salary from the list  double largest = emp.get(0).getSalary();  double secondLargest = (double) emp.get(1).getSalary();  for (int i = 0; i < emp.size(); i++) {  if (emp.get(i).getSalary() > largest) {  System.out.println("If condi");  secondLargest = largest;  largest = emp.get(i).getSalary();  }  if (emp.get(i).getSalary() > secondLargest && emp.get(i).getSalary() != largest) {  System.out.println("If condi2");  secondLargest = emp.get(i).getSalary();  }  }  System.out.print("Second biggest number " + secondLargest);  // return secondLargest;  }  }  Output:  [Employee [salary=17.0], Employee [salary=16.5], Employee [salary=16.0], Employee [salary=15.0], Employee [salary=14.0], Employee [salary=13.0], Employee [salary=12.0]]  Second biggest number1--16.5  Second biggest number 16.5 |

Note: while using Lambda to sort salary then in that case CompareSalary class is not required to implement at all.

Anonymous inner Class: The class which does not have any name and return type is called AIC.

|  |  |
| --- | --- |
| Thread t = new Thread();  Here in the above code we are just creating object of Thread class. | Thread t = new Thread(){  ………..  };  But here we are creating one inner class without name which extends Thread class and t is the object of that inner class. Here we can see that inner class does not have any name that is why it is called Anonymous inner class. |
| Runnable r = new Runnable();  Here the above line of code is invalid because we are trying to create object of Interface which is not possible. But🡺 | Runnable r = new Runnable(){  ………  };  But here we are creating one inner class without name which implements Runnable Interface and (r)is the object of inner implementation class. Here we can see that implementation inner class does not have any name that is why it is called Anonymous inner class. |

Anonymous inner class can be used wherever it is required for instant/immediate use, i.e. we can use AIC in between code also.

|  |  |
| --- | --- |
| Anonymous inner class  package lambda.ex.in.multithred.env;  public class TestInner{  public static void main(String[] args) {  Runnable r = new Runnable() {  @Override  Public void run() {  for (inti =0 ; i<5; i++) {  System.*out*.println("Child Clas");  }  }  };  Thread t = new Thread(r);  t.start();  for(inti=0;i<5;i++) {  System.*out*.println("Parent main Thread");  }  }  }  Here the advantage of AIC is that we don’t need to create any implementation class which implement Runnable Interface and creating child thread. We have just created one AIC in the same main method.As we can see that | Equivalent in Lambda expression  package lambda.ex.in.multithred.env;  publicclass TestThread {  publicstaticvoid main(String[] args) {  Runnable r = ()->{  for(inti=0;i<5;i++) {  System.*out*.println("Child Clas");  }  };  Thread t = new Thread(r);  t.start();  for(inti=0;i<5;i++) {  System.*out*.println("Main thread");  }  }  }  Since AIC also is like implementation class which implements Runnable interface which in-turn contain only one abstract method. Hence the AIC also can be written using Lambda Expression as mentioned in above example.  Note: If AIC implements an interface that contains single abstract method then only we can replace that anonymous inner class with lambda expression. |

Note: Anonymous Inner class is more powerful than Lambda Expression; Because Lambda expression can be used only when any Interface (FI) has exactly one abstract method. But on the other hand Anonymous Inner class can be used for the interface having more than one abstract method. And in case of AIC class does not use implement keyword for implementing interface abstract method, instead we just need to create reference object including body of the class as shown above in the example.

|  |  |
| --- | --- |
| Interface A{  M1();  M2();  } | Now the AIC for given Interface having two abstract method.  class AIC {  A a = new A() {  @Override  publicvoid M2() {  // TODO Auto-generated method stub  }  @Override  publicvoid M1() {  // TODO Auto-generated method stub  }  };  } |

Hence AIC is more powerful than Lambda Expression.

Default Methods: OR Virtual Extension Method: OR Defender Method:

* As we know that until java 1.7 i.e. before java 1.8, every method present inside interface is always public and abstract whether we are declaring or not.

|  |  |  |  |
| --- | --- | --- | --- |
| *void M1() OR* | *public void M1() OR* | *abstract void M1() OR* | *public abstract void M1()* |

* The above declared all four methods inside interface are same, there is no difference. Every method by default is public and abstract whether we declare or not.
* But from java 1.8 versions onwards apart from four methods as declared above we can declare default methods and static method inside the interface. And from 1.9 versions private methods are also allowed.
* Every variable present inside interface is always public static and final where it is java 1.7 or 1.8 or 1.9.
* Hence variable-vise there no enhancement inside interface but method-vise several method have been added inside interface in 1.8 and 1.9 version.
* Default Methods: is also known as Virtual Extension Method: OR Defender Method.
* This very important concept came in Java1.8 with this concept some other functionality like stream API is not possible.

Q: - Now the question arises that what is the need of default method?

* As we know that every method declared inside interface compulsory has to be implemented in its implementation classes. Let’s say we have 1 to 100 implementation class, and then every implementation class will have to implement all the methods declare inside the interface. Let’s say we have implemented all the method of Interface in all 1 to 100 implementation class. Now let’s say in future we have declared one more method in the interface then, it is mandatory to implement this newly added method in all 100 implementation class, otherwise we will get error saying implementation class is not abstract class and does not override abstract method, please implement Interface.
* Hence until java1.7 every method declared inside interface is final and we are not allowed to change. To solve this problem java people have introduced default method inside interface from java1.8 onwards. Using this method we can enhance Interface functionality without affecting its implementation classes. We don’t need to implement this default method in interface implementation classes.
* Hence instead of declaring normal method (public abstract method) inside interface if we declare default method inside interface then happily code will be compiled and will not get any error and it is not required to implement default method in implementation class.

Hence finally we can say that without affecting implementation classes we can enhance interface just by adding default method inside interface and whenever require we can implement this default method inside interface only. And if not require then we just left blank with it method body { }.

Or without effecting implementation class if we want to add new method to the interface then we should go for Default method.

Q: - Now the question comes, why the world default is used in defining default method?

It is not modifier that we have like (public, private and default). Like if we don’t declare any modifier then by-default method get default modifier. But here this is not the default modifier. The method is default because this default method is having default implementation.

Advantage of Default Method: Let’s say we have one default method inside interface then this default method can be used anywhere. We just need to implements that interface and we can call that default method in the implementation class.

Now let’s say we are not satisfied with the existing default method implemented inside interface then in that case we can override this default method anywhere according to our need. That is why this default method is called defender method because it can be override anywhere without affecting interface and implementation classes i.e. it defends the implementation classes. In other word if required we can use existing default method or we can override existing functionality and if don’t we can ignore. This way it defends implementation classes and interface.

|  |  |
| --- | --- |
| package com.defaulte.method;  interface Interf{  default void m1() {  System.*out*.println ("Default method...”);  }  }  Public class DefaulteMethodImpl implements Interf {  Public static void main(String[] args) {    DefaulteMethodImpl d = new DefaulteMethodImpl();  d.m1();  }  }  Output:  Default method...  Note: As here we can default method inside interface only and we can call in any of the method when required. | package com.defaulte.method;  interface Interf{  default void m1() {  System.*out*.println ("Default method...”);  }  }  Public class DefaulteMethodImpl implements Interf {  // Override Default method  Public void m1() {  System.*out*.println ("Override default method...");  }  Public static void main(String[] args) {  DefaulteMethodImpl d = new DefaulteMethodImpl();  d.m1();  }  }  Output:  Override default method...  Note: If we are not satisfied with the default method define inside interface then we can override and can write our own code according to our need. |

VIP Note: By-default this default method is available in the implementation class of interface. While overriding this default method in any implementation class the modifier always will be public, we cannot use default keyword while overriding otherwise we will get compile time error because in general class default key word is used with switch cases. Hence default key word can be used inside interface only not inside the class.

Note-2:

|  |
| --- |
| package com.defaulte.method;  interface Interf{  default void m1() {  System.*out*.println ("Default method...”); }  default int hashCode() {  return 10;  }  }  publicclass DefaulteMethodImpl implements Interf {  // Override Default method  Public void m1() {  System.*out*.println ("Override default method...");  }  Public static void main(String[] args) {  DefaulteMethodImpl d = new DefaulteMethodImpl();  d.m1();  }} |

|  |
| --- |
| The default method present inside interface is known as dummy method so that whenever require we can use this method according to our need by overriding.  Object class method cannot be declared as default inside interface otherwise we will get compile time error saying.  A default method cannot override a method from java.lang.Object.  Like here we tried to implement hashCode () method as default method which cannot be possible because hashCode () method is the method of Object class and it can be implement only in Object child class. Since by default all the class is the child of Object class hence here DefaulteMethodImpl will be treated as child of Object class so here in this class hashCode () can be implemented. Hence implementation of Object class is not required in interface and mentions it as default. That is why java people did not provide this feature in interface. |

Hence we can see that this default method is a very beautiful concept came java1.8.

Stream API- java -8: How streams work[#](https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/#how-streams-work)

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements:

List<String> myList =

Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList

.stream()

.filter(s -> s.startsWith("c"))

.map(String::toUpperCase)

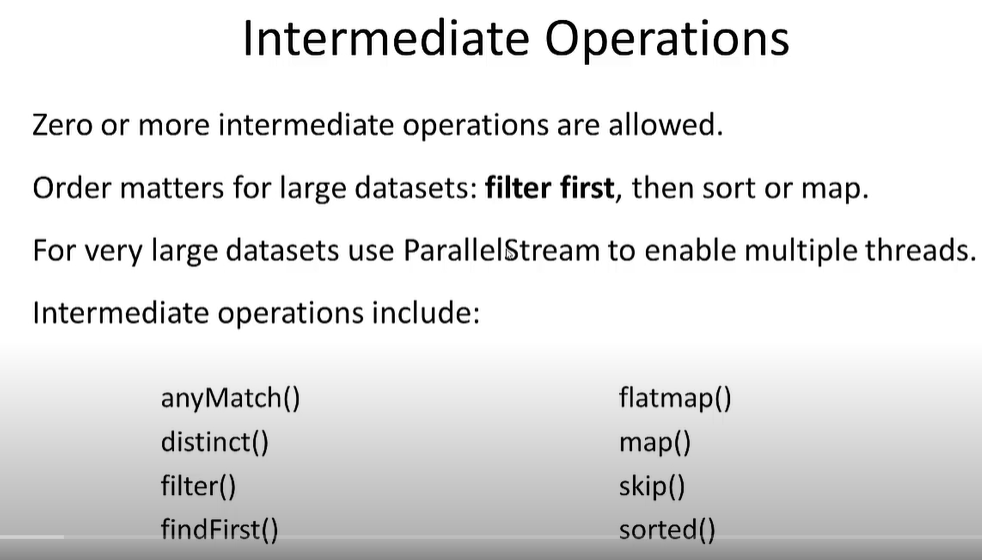
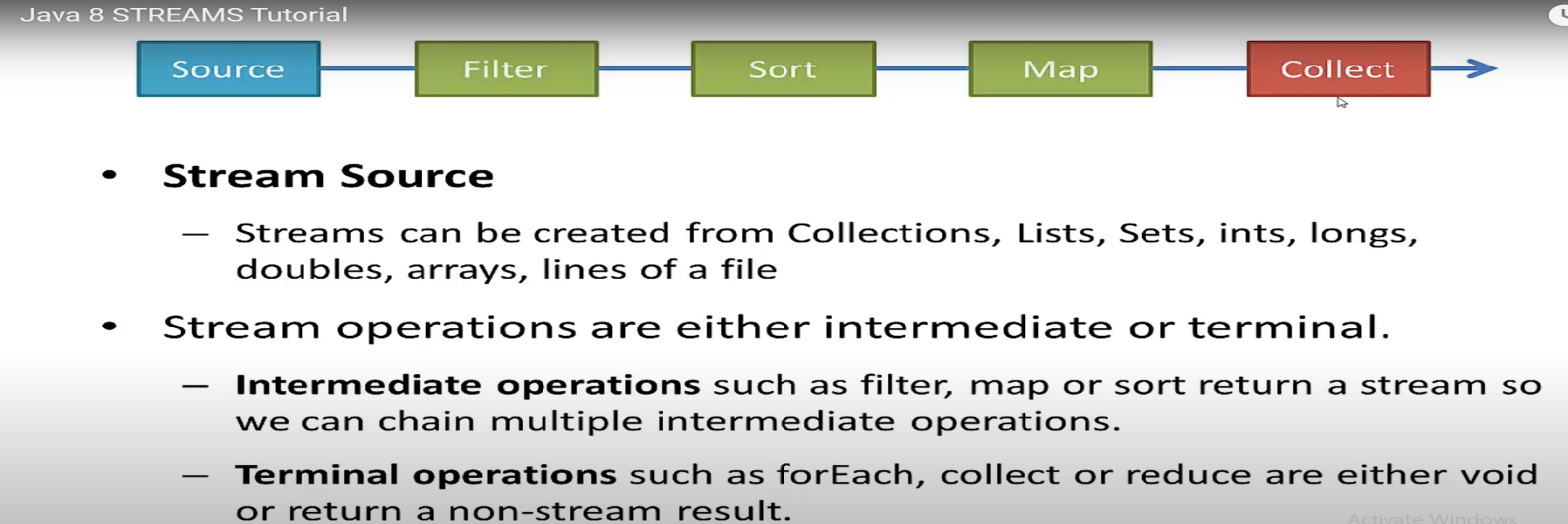
.sorted()

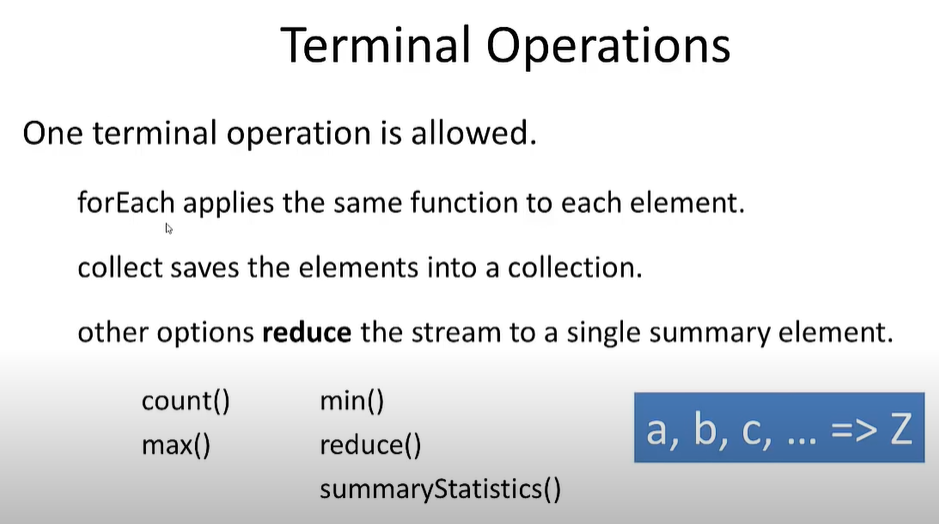
.forEach(System.out::println);

// C1

// C2

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example filter, map and sorted are intermediate operations whereas forEach is a terminal operation. For a full list of all available stream operations see the [Stream Javadoc](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html). Such a chain of stream operations as seen in the example above is also known as *operation pipeline*.





**Stream API in Java-8: Collection & Stream:**

If we want to represent a group of object as single entity then we should go for collection and if we want to process objects from the collection then should use Stream.

Q: To get the stream which method we have to use?

A: [Stream s = c.stream ();] so stream () is the method. C is the object of collection. So using stream () we can convert collection object into stream. And this method is available in java.util.stream package.

Case1- Let’s say we want to get a list of even number form the list containing even and odd both, let’s see how to do in Java 1.7 and java 1.8?

|  |  |
| --- | --- |
| Until Java- 1.7-  import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;  public class StreamsList {  public static void main(String[] args) {    public static void main(String[] args) {  List<Integer> l1 = new ArrayList<>();  List<Integer> l2 = new ArrayList<>();  l1.add(1);  l1.add(2);  l1.add(3);  l1.add(4);  l1.add(5);  l1.add(6);  l1.add(7);  l1.add(8);  l1.add(9);  l1.add(10);  for (Integer i : l1) {  if (i % 2 == 0) {  l2.add(i);  }  }  System.*out*.println(l2);  }  }  [2, 4, 6, 8, 10] | Until-Java - 1.8  import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;  public class StreamsList {  public static void main(String[] args) {  // TODO Auto-generated method stub  List<Integer> l1 = new ArrayList<>();  l1.add(6);  l1.add(7);  l1.add(8);  l1.add(9);  l1.add(10);  l1.add(1);  l1.add(2);  l1.add(3);  l1.add(4);  l1.add(5);    List<Integer> l2 = l1.stream().filter(i -> i % 2 == 0).collect(Collectors.*toList*());  System.*out*.println(l2);  System.out.println("Unsorted-" + l1);  List<Integer> sortedList = l1.stream().sorted().collect(Collectors.toList());  System.out.println("sortedList-" + sortedList);  }  }  [2, 4, 6, 8, 10]   1. First create ArrayList and add 10 number in the list(l1) 2. Convert this l1 ArrayList collection object into stream using stream() 3. Apply filter(use Predicate Boolean-valued FI to check even number)and this also return a string only 4. Finally collect this and convert this into list and store into l2 list |

Same filter can be applied for this case also. Let’s say we have a collection of mobile number of employees in List and we want to send SMS only for BSNL number only then in this case we can use stream-filter concept.

So wherever collection object has to be processed there we can use stream concept and we need to write very less code.

l1.stream().filter(i -> i >5).collect(Collectors.*toList*()); [ give the list of number greater than 5]

Filter is being used whenever we are checking any conditions then we need to use filter. Here we are just checking condition but not getting any value or object.

In filter if input is = 10 then output= from 0 to 10 or less than 10

Case2- Let’s say we have list of marks of student and we want add grace marks for each student’s marks. So here for every object if we want to perform some operation and we want some result object then we can go for map object.

So if for every object if after performing some operation (like adding, subtractions, sqrt, cube etc) and generating some new object then we should go for map.

Here we are performing operation on the object and expecting values in object then here we should go for map.

In filter if input is = 10 then output= 10 because operation is performed on every object and then collect the same.

|  |
| --- |
| package com.employee.salary;  import java.util.ArrayList;  import java.util.List;  import java.util.stream.Collectors;  public class StreamMap {  public static void main(String[] args) {  List<Integer> marks = new ArrayList<Integer>();  marks.add(100);  marks.add(200);  marks.add(300);  marks.add(400);  marks.add(500);  System.*out*.println(marks);  List<Integer> graceMarks = marks.stream().map(i -> i + 5).collect(Collectors.*toList*());  System.*out*.println(graceMarks);  }  }  [100, 200, 300, 400, 500]  [105, 205, 305, 405, 505] |
| List<String> elements = Stream.*of*("a", "b", "c").filter(element -> element.contains("b")).collect(Collectors.*toList*());  Output: elements-[b] |

Note: in JavaScript and python we don’t need to specify the data type, based on the value assigned to variable it dynamically assigned the data type. For example: [var x = 10; ] here in case of javaScript and python we dnot need to specify the data type and based on value assigned to x variable it will understood the data type. But in java we need to explicitly specify the data type based on the value assigned. But from java-10 onwords here in java also for local variable we can declare the variable like[ var x= 10; var name =Arun ;] and based on the value assigned to variable data type will be defined automatically.

|  |  |  |
| --- | --- | --- |
| stream() | sorted (): DSNO |  |
| filter(i->i% 2 ==0) | sorted((a, b) -> a.compareTo(b)) |  |
| map(i ->i+5) | sorted((a, b) ->-a.compareTo(b)) |  |
| collect() | sorted(Comparator) Custom SORTING |  |

Now let’s see another method in the Stream API count ():

If we want to know the number elements available in the stream then we will use count () method.

Let’s say if we want to count the number of even number or the number of student got the marks greater than 200.

long marksGtrThan200 = marks.stream().filter(i -> i >200).count();

long countOfEvenNumber = marks.stream().filter(i -> i %2==0).count();

Now let’s see another method in the Stream API sorted ():

If we want to sort in default natural sorting order then we need use sorted () method.

List<Integer> sortedList = l1.stream().sorted().collect(Collectors.toList());

Let’s if we want do our own custom sorting then in that case we will have to use Comparator Interface and internally it will compare() method to compare and sort the values.

|  |
| --- |
| Comparator<Integer> comp = (a, b) -> (a < b) ? 1 : (a > b) ? -1 : 0;  List<Integer> reverOrder = l1.stream().sorted(comp).collect(Collectors.*toList*()); |
| Or we can directly write condition inside sorted() method as shown below.  List<Integer> reverOrder = l1.stream().sorted((a, b) -> (a < b) ? 1 : (a > b) ? -1 : 0).collect(Collectors.*toList*()); |

So : sorted(Comparator) 🡺 customized sorting

Note: As we know that we want default natural sorting then we should go for Comparable(I) interface and as we know internally it uses compareTo() method to sort in natural order. Let’s say if we want to compare (i1,i2) in default natural sorting order then

|  |
| --- |
| List<Integer> naturalOrder = l1.stream().sorted((a, b) -> a.compareTo(b)).collect(Collectors.*toList*()); |

naturalOrder[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Now if want to reverse this order then we just need to put(-) minus (-a) and it will reverse the order so the above code will look like as given below

|  |
| --- |
| List<Integer> naturalOrder = l1.stream().sorted((a, b) -> -a.compareTo(b)).collect(Collectors.*toList*()); |

reverseOrder[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Note: What is the MIN element [MIN ()] and MAX element [MAX ()] present in the stream or list. Let’s say we have list [5, 3, 10, 15, 4] so here what is the MIN and MAX element? So Min= 3 & MAX = 15. How do we know that 3 is min and 15 is MAX?

Ans: We can tell the MIN and MAX element just by arranging them into ascending order [3,4,5,10,15]. So according to ascending 3 is Min and 15 is MAX.

Suppose my requirement is in descending order [15,10,5,4,3]. Now here what would be the MIN and MAX element? So here also the first element would be the MIN element and last element would be the MAX element.

So in the sorting First element always will considered as MIN element and last element is MAX element i.e. The MIN and MAX is not defined based on the value in sorting. In sorting first element is always MIN element and last element is MAX element whether order is ascending or descending.

So in descending order [15,10,5,4,3] also the Min element is 15 and MAX element is 3.

So whenever we talk about MIN and MAX element then compulsory we need to talk about sorting and compulsory we need to pass comparator object.

|  |
| --- |
| package com.employee.salary;  import java.util.ArrayList;  import java.util.List;  public class StreamMaxMin {  public static void main(String[] args) {  List<Integer> number = new ArrayList<Integer>();  number.add(5);  number.add(10);  number.add(3);  number.add(4);  number.add(15);  System.*out*.println("Number List" + number);  Integer minNumber = number.stream().min((a, b) -> a.compareTo(b)).get();  Integer maxNumber = number.stream().max((a, b) -> a.compareTo(b)).get();  System.*out*.println("Min Number" + minNumber);  System.*out*.println("Max Number" + maxNumber);  System.*out*.println("Now if we change the order or put -a then the Min & Max will be different");  Integer min\_Number = number.stream().min((a, b) -> -a.compareTo(b)).get();  Integer max\_Number = number.stream().max((a, b) -> -a.compareTo(b)).get();  System.*out*.println("Min Number" + min\_Number);  System.*out*.println("Max Number" + max\_Number);  System.*out*.println("Now if we change the order then the Min & Max will be different");  Integer min\_Cng\_order = number.stream().min((a, b) -> b.compareTo(a)).get();  Integer max\_chg\_order = number.stream().max((a, b) -> b.compareTo(a)).get();  System.*out*.println("Min min\_Cng\_order" + min\_Cng\_order);  System.*out*.println("Max max\_chg\_order" + max\_chg\_order);  }  }  Output:  Number List[5, 10, 3, 4, 15]  Min Number3  Max Number15  Now if we change the order or put -a then the Min & Max will be different  Min Number15  Max Number3  Now if we change the order then the Min & Max will be different  Min min\_Cng\_order15  Max max\_chg\_order3  So here in the above example, comparator object is getting passed inside min & max method to get min and max number and it is very easy to process the collection object using stream concept. |

Program: Write a program to sort the name (string) alphabetically in (A to z) and then (Z to A).

Program: Write a program to sort name in increasing order of their length.

Case3- Suppose I want to process each element present in the stream then usually we will do this using for loop or foreach loop until java 1.7. But from java 1.8 onwards we have got foreach() method inside which we can process each element of the stream.

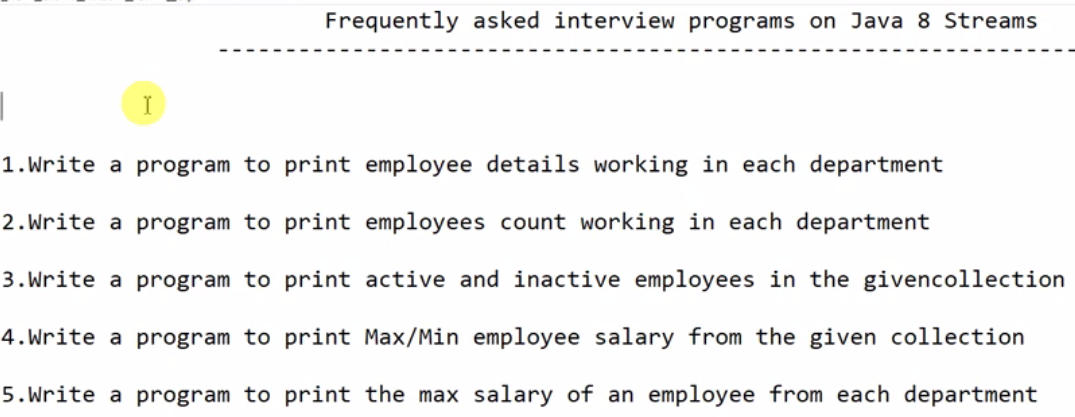
Stream().forEach(): This is not loop, this is method provided by Stream API to process each element present in the stream.

|  |  |
| --- | --- |
| For(Integer i1 : list){  System.out.println(i1)  } | Equivalent code in java 8 using foreach() method.  number.stream().forEach(System.*out*::println); |

Note: foreach(Consumer) :always expect a function inside foreach() method. Here println is a consumer from System. out object. Both above code will print the list value one by one.

|  |  |
| --- | --- |
| package com.employee.salary;  import java.util.ArrayList;  import java.util.List;  import java.util.function.Consumer;  public class ForEachMethod {  public static void main(String[] args) {  List<Integer> number = new ArrayList<Integer>();  number.add(5);  number.add(10);  number.add(3);  number.add(4);  number.add(15);  System.*out*.println("Number list:-" + number);  Consumer<Integer> c = i -> {  System.*out*.println("The square of each element is:" + i \* i);  };  number.stream().forEach(c);  // or directly we can write consumer inside foreach  number.stream().forEach(i -> {  System.*out*.println("The square of each element is:" + i \* i);  });  }  } | Output:  Number list:-[5, 10, 3, 4, 15]  The square of each element is:25  The square of each element is:100  The square of each element is:9  The square of each element is:16  The square of each element is:225 [[Open Declaration](eclipse-open:%E2%98%82=EmployeeTest/C:\/Program%20Files\/Java\/jdk1.8.0_211\/jre\/lib\/rt.jar%3cjava.util.function(Consumer.class%E2%98%83Consumer)](eclipse-open:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.function(Consumer.class%E2%98%83Consumer)[java](eclipse-javadoc:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util).[function](eclipse-javadoc:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.function).Consumer<[Integer](eclipse-javadoc:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.function(Consumer.class%E2%98%83Consumer%E2%98%82java.lang.Integer)> @[FunctionalInterface](eclipse-javadoc:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.lang(FunctionalInterface.class%E2%98%83FunctionalInterface) Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.  This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](eclipse-javadoc:%E2%98%82=EmployeeTest/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.function(Consumer.class%E2%98%83Consumer%E2%98%82%E2%98%82accept%E2%98%82Object). |

Stream().toArray():

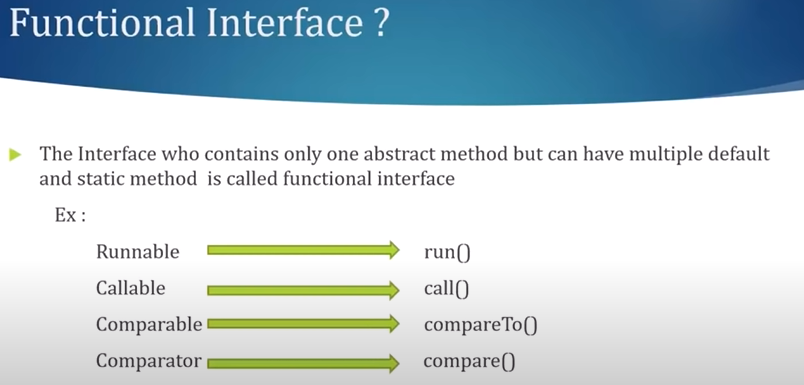


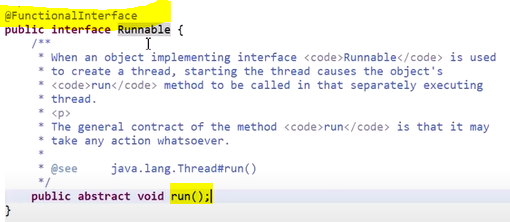
|  |
| --- |
| package com.java8.streams.grouping.concept;  public class PrintEmpDetailDeptWise {  // variable declaration  // parameterized constructor  // getters and setters  // override toString()  private int empId;  private String empName;  private int deptId;  private String status="Active";  private int empSalary;    public PrintEmpDetailDeptWise(int empId,String empName,int deptId,String status,int empSalary) {  this.empId=empId;  this.empName=empName;  this.deptId=deptId;  this.status=status;  this.empSalary=empSalary;  }  // Getters and setters  @Override  public String toString() {  return "PrintEmpDetailDeptWise [empId=" + empId + ", empName=" + empName + ", deptId=" + deptId + ", status="  + status + ", empSalary=" + empSalary + "]";  }  } |
| package com.java8.streams.grouping.concept;  import java.util.ArrayList;  import java.util.Comparator;  import java.util.List;  import java.util.Map;  import java.util.Optional;  import java.util.function.BinaryOperator;  import java.util.stream.Collectors;  public class TestGroupBy {  public static void main(String[] args) {  // Create an ArrayList of Employee type  // adding newly created object in the list.  // Converting listObject into sreams() and the apply the streams method  ArrayList<PrintEmpDetailDeptWise> empList = new ArrayList<PrintEmpDetailDeptWise>();  empList.add(new PrintEmpDetailDeptWise(101,"A",201,"Active",1000));  empList.add(new PrintEmpDetailDeptWise(102,"B",201,"Active",2000));  empList.add(new PrintEmpDetailDeptWise(103,"C",202,"Active",3000));  empList.add(new PrintEmpDetailDeptWise(104,"B",202,"Inactive",4000));  empList.add(new PrintEmpDetailDeptWise(105,"A",203,"Inactive",5000));  empList.add(new PrintEmpDetailDeptWise(106,"Arun",203,"Active",6000));  empList.add(new PrintEmpDetailDeptWise(107,"Farhan",204,"Active",7000));  empList.add(new PrintEmpDetailDeptWise(108,"Kiran",204,"Active",8000));  empList.add(new PrintEmpDetailDeptWise(109,"Suman",205,"Active",9000));  empList.add(new PrintEmpDetailDeptWise(110,"Parmit",205,"Active",9500));  // Print Employee Detail working in each Department  Map<Integer, List<PrintEmpDetailDeptWise>> empListByDeptWise = empList.stream().collect(Collectors.*groupingBy*(PrintEmpDetailDeptWise::getDeptId,Collectors.*toList*()));  // First converted the list object into streams()  // Java Stream collect() is mostly used to collect the stream elements to a collection: So using collect() collect all the elements of the stream  // After collecting streams-> group all the streams employee streams according to its deptid using Collectors.groupingBy  // Finally convert them into list using toList()  // This groupingBy returns map so assign this into map object whose key will be DeptId (integer) and value will be list of employee objects  // Note: groupingBy either return map or cuncurrnetMap object  /\*  The collect() method of Stream class can be used to accumulate elements of any Stream into a Collection. ...  The Collector class provides different methods like toList(), toSet(), toMap(), and toConcurrentMap() to collect the result of Stream into List, Set, Map, and ConcurrentMap in Java  \* \*/  **System.*out*.println("-----------------------Employee based on deptId-------------------------------------------");**  **empListByDeptWise.entrySet().forEach(entry-> {**  **System.*out*.println("Dept:-"+entry.getKey()+"-"+entry.getValue());**  **});**  **System.*out*.println("-------------------// Print employee count working in each department-------------------------------");**  **Map<Integer, Long> empCountInDept= empList.stream().collect(Collectors.*groupingBy*(PrintEmpDetailDeptWise::getDeptId, Collectors.*counting*()));**  **empCountInDept.entrySet().forEach(empCount->{**  **System.*out*.println("Dept:-"+empCount.getKey()+"--"+empCount.getValue());**  **});**  **System.*out*.println("-----------------------Employee based on name-------------------------------------------");**  **Map<String, List<PrintEmpDetailDeptWise>> empGroupByName = empList.stream().collect(Collectors.*groupingBy*(PrintEmpDetailDeptWise::getEmpName,Collectors.*toList*()));**  **empGroupByName.entrySet().forEach(entryName->{**  **System.*out*.println("EmpName:-"+ entryName.getKey()+"--"+entryName.getValue());**  **});**  **System.*out*.println("-------------------// Print employee having the same name-------------------------------");**  **List<PrintEmpDetailDeptWise> collect = empList.parallelStream().filter(entry->entry.getEmpName()=="A").collect(Collectors.*toList*());**  **System.*out*.println("SameNameEmp:-"+collect);**  **List<PrintEmpDetailDeptWise> equalSalary = empList.stream().filter(empSalary->empSalary.getEmpSalary()<2000).collect(Collectors.*toList*());**  **System.*out*.println("Employee having equalSalary:-"+equalSalary);**  **System.*out*.println("-------------------// Print employee based on status-------------------------------");**  **Map<String, List<PrintEmpDetailDeptWise>> empGroupByStatus = empList.stream().collect(Collectors.*groupingBy*(PrintEmpDetailDeptWise::getStatus,Collectors.*toList*()));**  **empGroupByStatus.entrySet().forEach(status->{**  **System.*out*.println("Status:-"+status.getKey()+"--"+status.getValue());**  **});**  **System.*out*.println("-------------------Employee Status count-------------------------------");**  **long ActiveEmp = empList.stream().filter(employee->"Active".equals(employee.getStatus())).count();**  **System.*out*.println("Active Employee:-"+ActiveEmp);**  **long InActiveEmp=empList.parallelStream().filter(inactive->"Inactive".equals(inactive.getStatus())).count();**  **System.*out*.println("InActive emp:-"+InActiveEmp);**  **System.*out*.println("-------------------Eamlpyee having max and min salary-------------------------------");**  **Optional<PrintEmpDetailDeptWise> max = empList.stream().max(Comparator.*comparing*(PrintEmpDetailDeptWise::getEmpSalary)).filter(salary->salary.getEmpSalary()>=6000);**  **Optional<PrintEmpDetailDeptWise> min = empList.stream().min(Comparator.*comparing*(PrintEmpDetailDeptWise::getEmpSalary));**  **System.*out*.println("MAX:"+max);**  **System.*out*.println("MIN:"+min);**  **System.*out*.println("-------------------Max Salary in each department-------------------------------");**  **Map<Integer, Optional<PrintEmpDetailDeptWise>> maxSalInEachDept = empList.stream().collect(Collectors.*groupingBy*(PrintEmpDetailDeptWise::getDeptId, Collectors.*reducing*(BinaryOperator.*maxBy*(Comparator.*comparing*(PrintEmpDetailDeptWise::getEmpSalary)))));**  **maxSalInEachDept.entrySet().forEach(entry->{**  **System.*out*.println("Dept:"+entry.getKey()+":::"+entry.getValue());**  **});**  **System.*out*.println("-------------------MIN Salary in each department-------------------------------");**  **Map<Integer, Optional<PrintEmpDetailDeptWise>> minSalInEachDept = empList.parallelStream().collect(Collectors.*groupingBy*(PrintEmpDetailDeptWise::getDeptId, Collectors.*reducing*(BinaryOperator.*minBy*(Comparator.*comparing*(PrintEmpDetailDeptWise::getEmpSalary)))));**  **minSalInEachDept.entrySet().forEach(entry->{**  **System.*out*.println("Dept::"+entry.getKey() +"::"+entry.getValue());**  **});**  **}**  **}** |
| Output:  -----------------------Employee based on deptId-------------------------------------------  Dept:-201-[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000], PrintEmpDetailDeptWise [empId=102, empName=B, deptId=201, status=Active, empSalary=2000]]  Dept:-202-[PrintEmpDetailDeptWise [empId=103, empName=C, deptId=202, status=Active, empSalary=3000], PrintEmpDetailDeptWise [empId=104, empName=B, deptId=202, status=Inactive, empSalary=4000]]  Dept:-203-[PrintEmpDetailDeptWise [empId=105, empName=A, deptId=203, status=Inactive, empSalary=5000], PrintEmpDetailDeptWise [empId=106, empName=Arun, deptId=203, status=Active, empSalary=6000]]  Dept:-204-[PrintEmpDetailDeptWise [empId=107, empName=Farhan, deptId=204, status=Active, empSalary=7000], PrintEmpDetailDeptWise [empId=108, empName=Kiran, deptId=204, status=Active, empSalary=8000]]  Dept:-205-[PrintEmpDetailDeptWise [empId=109, empName=Suman, deptId=205, status=Active, empSalary=9000], PrintEmpDetailDeptWise [empId=110, empName=Parmit, deptId=205, status=Active, empSalary=9500]]  -------------------// Print employee count working in each department-------------------------------  Dept:-201--2  Dept:-202--2  Dept:-203--2  Dept:-204--2  Dept:-205--2  -----------------------Employee based on name-------------------------------------------  EmpName:-Suman--[PrintEmpDetailDeptWise [empId=109, empName=Suman, deptId=205, status=Active, empSalary=9000]]  EmpName:-Farhan--[PrintEmpDetailDeptWise [empId=107, empName=Farhan, deptId=204, status=Active, empSalary=7000]]  EmpName:-A--[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000], PrintEmpDetailDeptWise [empId=105, empName=A, deptId=203, status=Inactive, empSalary=5000]]  EmpName:-B--[PrintEmpDetailDeptWise [empId=102, empName=B, deptId=201, status=Active, empSalary=2000], PrintEmpDetailDeptWise [empId=104, empName=B, deptId=202, status=Inactive, empSalary=4000]]  EmpName:-Kiran--[PrintEmpDetailDeptWise [empId=108, empName=Kiran, deptId=204, status=Active, empSalary=8000]]  EmpName:-C--[PrintEmpDetailDeptWise [empId=103, empName=C, deptId=202, status=Active, empSalary=3000]]  EmpName:-Arun--[PrintEmpDetailDeptWise [empId=106, empName=Arun, deptId=203, status=Active, empSalary=6000]]  EmpName:-Parmit--[PrintEmpDetailDeptWise [empId=110, empName=Parmit, deptId=205, status=Active, empSalary=9500]]  -------------------// Print employee having the same name----------------------------------------------------  SameNameEmp:-[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000], PrintEmpDetailDeptWise [empId=105, empName=A, deptId=203, status=Inactive, empSalary=5000]]  Employee having equalSalary:-[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000]]  -------------------// Print employee based on status-------------------------------  Status:-Active--[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000], PrintEmpDetailDeptWise [empId=102, empName=B, deptId=201, status=Active, empSalary=2000], PrintEmpDetailDeptWise [empId=103, empName=C, deptId=202, status=Active, empSalary=3000], PrintEmpDetailDeptWise [empId=106, empName=Arun, deptId=203, status=Active, empSalary=6000], PrintEmpDetailDeptWise [empId=107, empName=Farhan, deptId=204, status=Active, empSalary=7000], PrintEmpDetailDeptWise [empId=108, empName=Kiran, deptId=204, status=Active, empSalary=8000], PrintEmpDetailDeptWise [empId=109, empName=Suman, deptId=205, status=Active, empSalary=9000], PrintEmpDetailDeptWise [empId=110, empName=Parmit, deptId=205, status=Active, empSalary=9500]]  Status:-Inactive--[PrintEmpDetailDeptWise [empId=104, empName=B, deptId=202, status=Inactive, empSalary=4000], PrintEmpDetailDeptWise [empId=105, empName=A, deptId=203, status=Inactive, empSalary=5000]]  -------------------Employee Status count-------------------------------  Active Employee:-8  InActive emp:-2  -------------------Eamlpyee having max and min salary-------------------------------  MAX:Optional[PrintEmpDetailDeptWise [empId=110, empName=Parmit, deptId=205, status=Active, empSalary=9500]]  MIN:Optional[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000]]  -------------------Max Salary in each department-------------------------------  Dept:201:::Optional[PrintEmpDetailDeptWise [empId=102, empName=B, deptId=201, status=Active, empSalary=2000]]  Dept:202:::Optional[PrintEmpDetailDeptWise [empId=104, empName=B, deptId=202, status=Inactive, empSalary=4000]]  Dept:203:::Optional[PrintEmpDetailDeptWise [empId=106, empName=Arun, deptId=203, status=Active, empSalary=6000]]  Dept:204:::Optional[PrintEmpDetailDeptWise [empId=108, empName=Kiran, deptId=204, status=Active, empSalary=8000]]  Dept:205:::Optional[PrintEmpDetailDeptWise [empId=110, empName=Parmit, deptId=205, status=Active, empSalary=9500]]  -------------------MIN Salary in each department-------------------------------  Dept::201::Optional[PrintEmpDetailDeptWise [empId=101, empName=A, deptId=201, status=Active, empSalary=1000]]  Dept::202::Optional[PrintEmpDetailDeptWise [empId=103, empName=C, deptId=202, status=Active, empSalary=3000]]  Dept::203::Optional[PrintEmpDetailDeptWise [empId=105, empName=A, deptId=203, status=Inactive, empSalary=5000]]  Dept::204::Optional[PrintEmpDetailDeptWise [empId=107, empName=Farhan, deptId=204, status=Active, empSalary=7000]]  Dept::205::Optional[PrintEmpDetailDeptWise [empId=109, empName=Suman, deptId=205, status=Active, empSalary=9000]] |

**What is Lambda Expression?**



**Functional Interface ?**





Runnable interface has only one abstract method run ().

|  |  |
| --- | --- |
| **package** functional.interfac.demo;  @FunctionalInterface  **public** **interface** MyFunctionalInterface {    **void** m1();    **default** **void** m2() {  System.***out***.println ("My default method m2...”);  }  **default** **void** m3() {  System.***out***.println ("My Default Method m3...”);  }  **static** **void** m4() {  System.***out***.println ("My Static Method m4...”);  }  **static** **void** m5() {  System.***out***.println ("My Static Method m5...”);  }  } | We have created functional interface:  Where we have created exactly one abstract class.  We can create multiple default method and multiple static methods |

Example:

|  |
| --- |
| **package functional.interfac.demo;**  **interface Calculator{**  **void calculate();**  **}**  **public class CalculatorImpl implements Calculator{**  **// Old traditional way to implement interface**  **@Override**  **public void calculate() {**  **System.*out*.println("Old Traditional way to implement interface ");**  **}**    **public static void main(String[] args) {**  CalculatorImpl old = **new** CalculatorImpl();  old.calculate();  **// Comment the above code and now let's implement the interface using lambda expression**  **// Syntax**  **// () - > {body}**  **Calculator calculator= () -> {System.*out*.println ("Implement functional interface using lambda expression...”) ;};**  **calculator.calculate();**  }  }  **Output:**  Old Traditional way to implement interface  Implement functional interface using lambda expression... |
| Note: If only one statement is there then no need to put { } also. So the above statement can be written like  **Calculator calculator= () -> System.*out*.println ("Implement functional interface using lambda expression...”);** |

**Note**: Incase of old traditional way, we have to use implements keyword to implement Calculator’s calculate method. While in case of java 8, using lambda expression we don’t need to use implements keyword to implement calculate method.

Here we have just defined the functional interface method using lambda expression and assign it to Functional Interface reference (**calculator)**. And after that using Functional interface reference invoke the **calculate ()** method which in-turn execute the method body defined by using lambda expression.

**Lambda Expression with arguments and return type:**

|  |
| --- |
| **package** functional.interfac.demo;  **public** **class** LambdaExpWithArgumentAndReturnType {  @FunctionalInterface  **public** **interface** Calculator{    **int** calculate(**int** x, **int** y);  }  **public** **static** **void** main(String[] args) {    //void calculate(int x, int y); // To implement just remove void calculate and (x, y) -> { statements ..}  Calculator calculation= (**int** x, **int** y) -> {**return** x+y; };  System.***out***.println("Addition of two number: "+calculation.calculate(3,8));  }  } |
| **Output:**  Addition of two number: 11  Note: Since here we have only one line of statement so above lambda expression can be written as:  Calculator calculation= ( x, y) -> x+y; |

**Lambda Expression with conditional expressions:**

|  |
| --- |
| **package** functional.interfac.demo;  **public** **class** LambdaExpWithArgumentAndReturnType {  @FunctionalInterface  **public** **interface** Calculator{    **int** calculate(**int** x, **int** y);  }  **public** **static** **void** main(String[] args) {    //void calculate(int x, int y); // To implement just remove void calculate and (x, y) -> { statements ..}  //Calculator calculation= ( x, y) -> x+y;    Calculator calculation1= (x,y)->{    **if**(x>y) {  System.***out***.println("Greater number is-:"+x);  }**else**  System.***out***.println("Greater number is :"+y);  **return** x;  };  calculation1.calculate(13,8);  }  }  Output:  Greater number is-:13 |

Advantage: So the main advantage is it reduces the code length and we don’t need to write boiler plate code.

**Complex Example for Lambda Expression:**

|  |
| --- |
| 1. **Book.java class having three variables**   **public** **class** Book {  **public** **class** Book {  **private** **int** book\_id;  **private** String name;  **private** **int** pageNo;  **public** Book(**int** book\_id, String name, **int** pageNo) {  **super**();  **this**.book\_id = book\_id;  **this**.name = name;  **this**.pageNo = pageNo;  }  @Override  **public** String toString() {  **return** "Book [book\_id=" + book\_id + ", name=" + name + ", pageNo=" + pageNo + "]";  }  // Getters and Setters |
| 1. **BookListFromDAO.java class returning list of book from the database or service**   **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** BookListFromDAO {  **public** List<Book> bookList(){  List<Book> list= **new** ArrayList<Book>();  list.add(**new** Book(1, "Book1", 10));  list.add(**new** Book(2, "Book2", 20));  list.add(**new** Book(3, "Book3", 30));  list.add(**new** Book(3, "Hibernate", 30));  list.add(**new** Book(3, "Spring", 30));  list.add(**new** Book(3, "Applet", 30));  **return** list;  }} |

1. **Now let’s create the service class where we will write business logic to sort these books based on name.**

|  |
| --- |
| **import** java.util.Collections;  **import** java.util.Comparator;  **import** java.util.List;  **public** **class** BookService {  **public** List<Book> getBookList(){  List<Book> bookList= **new** BookListFromDAO().bookList();  Collections.*sort*(bookList, **new** MyComparator());  **return** bookList;  }  **public** **static** **void** main(String[] args) {  System.***out***.println(**new** BookService().getBookList());  }  }  **class** MyComparator **implements** Comparator<Book>{  **public** **int** compare(Book o1, Book o2) {  **return** (o1.getName().compareTo(o2.getName()));  }  }  **Output:**  [Book [book\_id=3, name=Applet, pageNo=30],  Book [book\_id=1, name=Book1, pageNo=10],  Book [book\_id=2, name=Book2, pageNo=20],  Book [book\_id=3, name=Book3, pageNo=30],  Book [book\_id=3, name=Hibernate, pageNo=30],  Book [book\_id=3, name=Spring, pageNo=30]]   * Here we have created one MyComparator **implements** Comparator<Book> class which is sorting the books based on its name. * Then we are passing book list and MyComparator class object in the Collections.*sort*(bookList, **new** MyComparator());   This is the traditional approach to sort the list. |
| **import** java.util.Collections;  **import** java.util.Comparator;  **import** java.util.List;  **public** **class** BookService {  **public** List<Book> getBookList(){  List<Book> bookList= **new** BookListFromDAO().bookList();  Collections.*sort*(bookList, **new Comparator<Book>() {**  **@Override**  **public int compare(Book o1, Book o2) {**  **return (o1.getName().compareTo(o2.getName()));**  **}**  **}**);  **return** bookList;  }  **public** **static** **void** main(String[] args) {  System.***out***.println(**new** BookService().getBookList());  }  }  This is another way where we have not written can class for comparator, we have just written the anonymous class inside the Collections.sort () method as we can see above. |

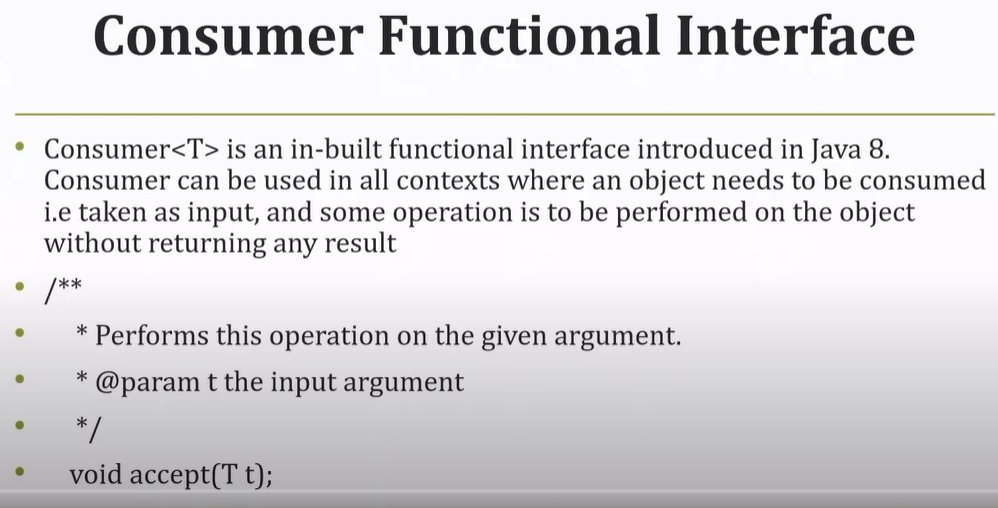
**Now let’s sort the above list of books using lambda expression:**

|  |
| --- |
| **public** List<Book> getBookList(){  List<Book> bookList= **new** BookListFromDAO().bookList();  Collections.*sort*(bookList, (Book o1, Book o2) -> (o1.getName().compareTo(o2.getName())));  **return** bookList;  } |
| **Note: How we have written Lambda expression:**  /\*class MyComparator implements Comparator<Book>{  public int compare(Book o1, Book o2) {  return (o1.getName().compareTo(o2.getName()));  } \*/    /\* Equivalent Lambda expression of above method  (Book o1, Book o2) -> {return (o1.getName().compareTo(o2.getName())) };  //Or  (o1, o2) -> (o1.getName().compareTo(o2.getName());  \*/ |
| **Complete Class:**  **public** **class** BookService {  **public** List<Book> getBookList(){  List<Book> bookList= **new** BookListFromDAO().bookList();  Collections.*sort*(bookList, **(Book o1, Book o2) -> (o2.getName().compareTo(o1.getName()))**);  **return** bookList;  }  **public** **static** **void** main(String[] args) {  System.***out***.println(**new** BookService().getBookList());  }  } |

Note: here Comparator Interface is nothing but a functional interface that is why we can represent its implementation in lambda expression.

**JAVA-8 – Consumer, Predicate, Supplier Example:**

**Consumer Functional Interface:**



**It is an in-built functional Interface in java-8.**

**When we can use Consumer Interface:**

Whenever we need an object as an input and based on this input we want to perform some operation and we don’t want to return anything.

So if we go inside this Consumer Interface then we will get only one method [void accept (T t)]. Here in this method based on the argument we perform some operation and return type is void.

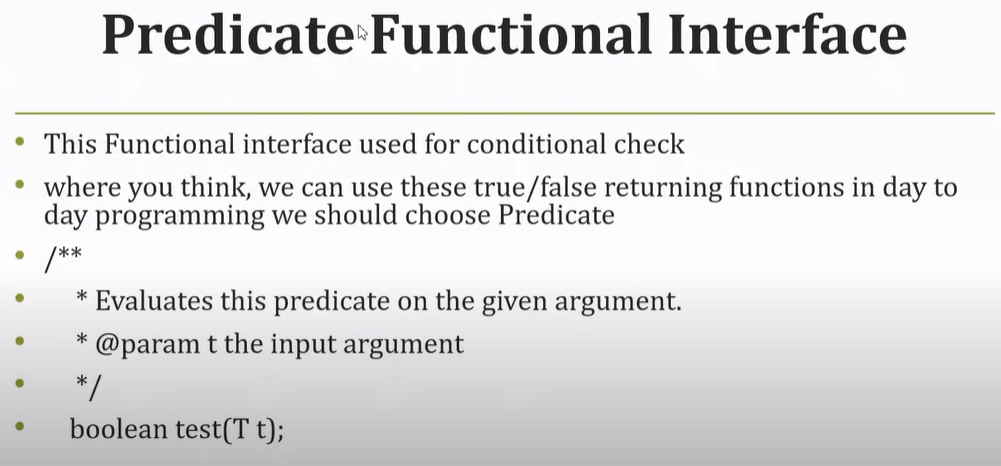
|  |
| --- |
| **Old way to implement Interface:**  **package** ConsumerInterface;  **import** java.util.function.Consumer;  **public** **class** ConsumerInterfaceDemo **implements** Consumer<Integer>{  @Override  **public** **void** accept(Integer t) {  System.***out***.println("The value passed in accept() :"+ t);  }  **public** **static** **void** main(String[] args) {  ConsumerInterfaceDemo treditinalWay = **new** ConsumerInterfaceDemo();  treditinalWay.accept(30);  }  }  **Output**:  The value passed in accept () :-30 |
| **Using Lambda Expression:**  Here in the below program we have written accept () implementation using lambda expression instead of using Old ways as we have done in above program. |
| **package** ConsumerInterface;  **import** java.util.function.Consumer;  **public** **class** ConsumerInterfaceUsingLambda {  /\* @Override  public void accept(Integer t) {  System.out.println("The value passed in accept() :"+ t);  } \*/  **public** **static** **void** main(String[] args) {  Consumer<Integer> consumer = (t) -> {System.***out***.println("The value passed in accept () :-"+t);};  consumer.accept(30);  }  }  **Output:**  The value passed in accept () :-30 |

In java stream api there is one method called forEach() and this forEach() always accept the Consumer Functional Interface

Program to print List using Consumer and forEach loop:

|  |
| --- |
| **package** ConsumerInterface;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.function.Consumer;  **public** **class** PrintingListUsigConsumerForEach {  **public** **static** **void** main(String[] args) {  Consumer<Integer> consumer = (t) -> {System.***out***.println("The value passed in accept () :-"+t);};  consumer.accept(30);    List<Integer> list = Arrays.*asList*(1,2,3,4,5);  list.stream().forEach(consumer);  }  } |
| **Output:**  The value passed in accept () :-30  The value passed in accept () :-1  The value passed in accept () :-2  The value passed in accept () :-3  The value passed in accept () :-4  The value passed in accept () :-5  **Note**: Here in the forEach loop it takes Consumer Interface reference as an argument and internally it calls accept () method for each element of the list and thus it print each element of list.    **The above program can be written in one line as given below.** |
| **public** **static** **void** main(String[] args) {  /\* Consumer<Integer> consumer = (t) -> {System.out.println("The value passed in accept () :-"+t);};  consumer.accept(30); \*/    List<Integer> list = Arrays.*asList*(1,2,3,4,5);  list.stream().forEach((t) -> {System.***out***.println("The value passed in accept () :-"+t);});  }  }  The value passed in accept () :-1  The value passed in accept () :-2  The value passed in accept () :-3  The value passed in accept () :-4  The value passed in accept () :-5 |

**Predicate Functional Interface:**



So whenever we have to perform some conditional statements and want to return Boolean then we have to use Predicate Functional Interface.

It contains one method [boolean test (T t)]: Here argument T is the generic type i.e. it can be object, int, String etc.

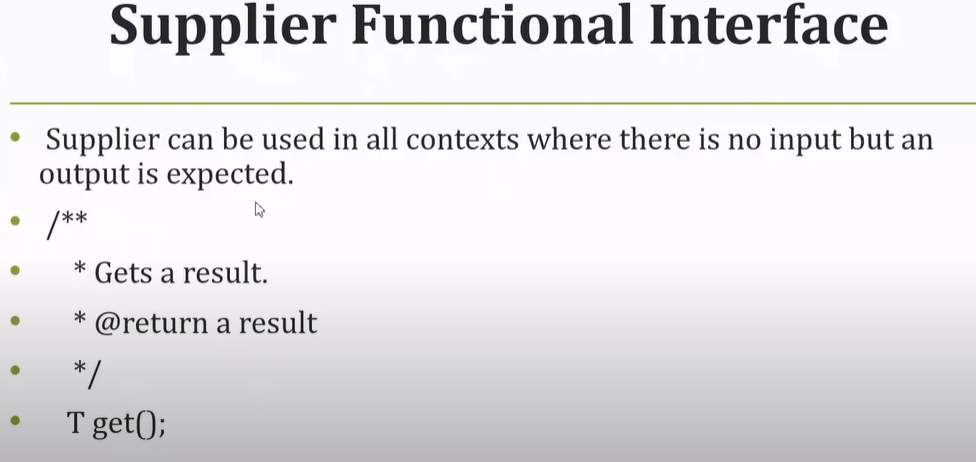
**Using Traditional way to implement Predicate Interface:**

|  |  |
| --- | --- |
| **package** PredicateInterface;  **import** java.util.function.Predicate;  **public** **class** PredicateInterfaceDemoUsingOldWay **implements** Predicate<Integer>{  @Override  **public** **boolean** test(Integer t) {  **if**(t % 2==0) {  **return** **true**;  }**else**  **return** **false**;  }  **public** **static** **void** main(String[] args) {  Predicate<Integer> oldWay = **new** PredicateInterfaceDemoUsingOldWay();  System.***out***.println(oldWay.test(20));  }  }  Output:  true | |
| **Using Lambda Expression:**  **package** PredicateInterface;  **import** java.util.function.Predicate;  **public** **class** PredicateInterfaceUsingLambda {  **public** **static** **void** main(String[] args) {  /\*  @Override  public boolean test(Integer t) {  if(t % 2==0) {  return true;  }else  return false;  } \*/  Predicate<Integer> predicate= (Integer t) -> {  **if**(t % 2 ==0)  **return** **true**;  **else**  **return** **false**;  };  System.***out***.println(predicate.test(201));  }  } | **Same Program using lambda in one liner:**  **package** PredicateInterface;  **import** java.util.function.Predicate;  **public** **class** PredicateLambdaInOneLine {  **public** **static** **void** main(String[] args) {    Predicate<Integer> predicate = (t) -> t % 2 ==0;  System.***out***.println(predicate.test(5));  }  }  **Output:**  **False** |

Note: In stream api we have one method filter() which internally use the Prdicate interface for conditinal check.

|  |
| --- |
| **package** PredicateInterface;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.function.Predicate;  **public** **class** PredicateLambdaInOneLine {  **public** **static** **void** main(String[] args) {    Predicate<Integer> predicate = (t) -> t % 2 ==0;  System.***out***.println(predicate.test(5));  List<Integer> list = Arrays.*asList*(1,2,3,4,5);  list.stream().filter(predicate).forEach(t -> System.***out***.println("The even value is: "+t));  }  }  **Output:**  false // For test(5)  The even value is: 2  The even value is: 4 |
| **In Short way:**  **package** PredicateInterface;  **import** java.util.Arrays;  **import** java.util.List;  **public** **class** PredicateAndFilter {  **public** **static** **void** main(String[] args) {  List<Integer> list = Arrays.*asList*(1,2,3,4,5);  list.stream().filter(t-> t% 2== 0).forEach(t-> System.***out***.println("The even vlaue : "+t));  }  }  **Output:**  The even vlaue : 2  The even vlaue : 4  Note: Here inside the filter it call the Predicate test() method for each element of the list for conditional check. |

**Supplier Functional Interface:**



Here in this interface we have one abstract method get () of generic return type T. So always this function will return the result.

When we can use this Supplier FI:

Let’s assume we have a list of element and we have to do some filter operation, but after filtering we are not getting any result but we want to return a dummy result.

|  |
| --- |
| **package** SupplierInterface;  **import** java.util.function.Supplier;  **public** **class** SupplierInterfaceImplUsingOldWay **implements** Supplier<String>{  @Override  **public** String get() {  // **TODO** Auto-generated method stub  **return** "This is supplier Functional Interface: ";  }  **public** **static** **void** main(String[] args) {  Supplier<String> oldWays = **new** SupplierInterfaceImplUsingOldWay();  System.***out***.println(oldWays.get());  }  }  Output:  This is supplier Functional Interface: |
| **Using Lambda Expression:**  **package** SupplierInterface;  **import** java.util.function.Supplier;  **public** **class** SupplierUsingLambda {  **public** **static** **void** main(String[] args) {  Supplier<String> supplier = () -> "This is supplier Inteface using lambda Exp";  System.***out***.println(supplier.get());  }  }  So when we call get() of Supplier FI then it execute the body of supplier and return the value. |

As we know that supplier FI can be used to return the default value if conditions does not satisfied, so let’s take an example of list having two element and if any of the element find then return the same else return the default supplier value:

|  |
| --- |
| **package** SupplierInterface;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.function.Supplier;  **public** **class** SupplierUsingLambda {  **public** **static** **void** main(String[] args) {  Supplier<String> supplier = () -> "This is supplier Inteface using lambda Exp";  //System.out.println(supplier.get());    List<String> list = Arrays.*asList*("A","B");  System.***out***.println(list.stream().findAny().orElseGet(supplier));  }  }  Output:  A  Note: Since here list contains value that is why it is returning the value from the list. Now lets make the list empty and then check. |
| **package** SupplierInterface;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.function.Supplier;  **public** **class** SupplierUsingLambda {  **public** **static** **void** main(String[] args) {  Supplier<String> supplier = () -> "This is supplier Inteface using lambda Exp";  //System.out.println(supplier.get());    List<String> list = Arrays.*asList*(); // Empty list  System.***out***.println(list.stream().findAny().orElseGet(supplier));  }  }  Output:  This is supplier Inteface using lambda Exp  Note: Here findAny() does not find any value in the list so it execute the orElseGet(supplier) and then it will print supplier default message. |
| **In short: another way:**  **public** **static** **void** main(String[] args) {  //Supplier<String> supplier = () -> "This is supplier Inteface using lambda Exp";  //System.out.println(supplier.get());  List<String> list = Arrays.*asList*();  System.***out***.println(list.stream().findAny().orElseGet(() -> "This is supplier Inteface using lambda Exp"));  }  Note: Please note the difference: |

So these three inbuilt functional interface can be used mostly in case of stream api to process the Collection objects.

**Stream in java**



|  |  |
| --- | --- |
|  | 1. We can achieve functional programming i.e. if we have functional interface then we can implement it with lambda expression. 2. Since we are going the implement FI with lambda expression so it reduces the code length and since Stream API support pipeline operation so we can write multiple operations simultaneously and thus code will be reduced. 3. Bulk Operation: If we have a collection with huge data and we want to perform some operation then we should go for stream API, because as compared to our traditional approach stream will give the better performance. |

Now in continuation let’s first discuss two methods of streams API. And we will discuss that how these internally works.

1. Filter : For conditional check
2. forEach : For iteration

|  |
| --- |
| **package** filter.and.forEach;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** FilterAndForEachForList {  **public** **static** **void** main(String[] args) {  List<String> studentList = **new** ArrayList<String>();  studentList.add("Ram");  studentList.add("Lakshman");  studentList.add("Bharat");  studentList.add("Shatrughan");  studentList.add("Sita");  studentList.add("Babita");  System.***out***.println("Iterating the list in traditional way");  **for**(String s : studentList) {  System.***out***.println(s);  }  System.***out***.println("Stream API forEach Loop");  studentList.stream().forEach(t-> System.***out***.println(t));  System.***out***.println("Filter the name length > 4 and then printing using forEach loop");  studentList.stream().filter(t-> t.length()>4).forEach(t-> System.***out***.println(t));  }  } |

Internally stream api forEach loop use the Consumer FI accept() method for each element to print. So this is the way we can iterate list using stream api.

Similarly stream api filter method use Predicate FI and this FI contains one test() method which is being used for conditional statement.

Output:

|  |  |  |
| --- | --- | --- |
| Iterating the list in traditional way  Ram  Lakshman  Bharat  Shatrughan  Sita  Babita | Stream API forEach Loop  Ram  Lakshman  Bharat  Shatrughan  Sita  Babita | Filter the name length > 4 and then printing using forEach loop  Lakshman  Bharat  Shatrughan  Babita |

**Internal working of stream forEach loop:**

|  |
| --- |
| System.***out***.println("Internal working of stream forEach loop");  Consumer<String> consumer =(String t)-> System.***out***.println(t);  **for**(String s : studentList) {  consumer.accept(s);  So internally stream forEach loop implements the traditional for each loop and inside this loop it calls the Consumer FI accept() for each element. |

**Iterating a map using stream API:**

|  |  |  |  |
| --- | --- | --- | --- |
| **package** filter.and.forEach;  **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** FilterAndForEachForMap {  **public** **static** **void** main(String[] args) {    Map<Integer ,String> entry = **new** HashMap<Integer, String>();  entry.put(1, "a");  entry.put(2, "b");  entry.put(3, "c");  entry.put(4, "d");  System.***out***.println("Iterating Map using Stream API");  entry.forEach((key, value)->System.***out***.println(key+":"+value)); // one way  entry.entrySet().stream().forEach(obj -> System.***out***.println(obj.getKey()+"="+obj.getValue())); // another way  entry.entrySet().stream().forEach(obj -> System.***out***.println(obj)); // another way  // fiter method  System.***out***.println();  entry.entrySet().stream().filter(key->key.getKey() % 2==0).forEach(obj->System.***out***.println(obj));  }  } | | | |
| Iterating Map using Stream API  1:a  2:b  3:c  4:d | 1=a  2=b  3=c  4=d | 1=a  2=b  3=c  4=d | // filter method  2=b  4=d |

Real time example to use stream and filter:

1. **Emoloyee.java:** Having EmpId, Salary, DeptName, EmpName, Created parameterized constructor, override toString method.
2. **EmployeeDao.java:** Created a list method which is fetching employee list from DB and returning list of employee.
3. **EmployeeService.**java: Here we are writing business logic on employee list, and fetching the taxable employee list based on their salary.

|  |
| --- |
| **package** filter.and.forEach;  **public** **class** Employee {  **private** **int** empId;  **private** **int** salary;  **private** String name;  **private** String detptName;    **public** Employee(**int** empId, **int** salary, String name, String detptName) {  **super**();  **this**.empId = empId;  **this**.salary = salary;  **this**.name = name;  **this**.detptName = detptName;  }  @Override  **public** String toString() {  **return** "Employee [empId=" + empId + ", salary=" + salary + ", name=" + name + ", detptName=" + detptName + "]";  }  } |
| **package** filter.and.forEach;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** EmployeeDao {  **public** **static** List<Employee> getEmployeeList(){  List<Employee> empList= **new** ArrayList<Employee>();  empList.add(**new** Employee(1, 100, "A", "Defence"));  empList.add(**new** Employee(2, 101, "B", "Railway"));  empList.add(**new** Employee(3, 102, "C", "IT"));  empList.add(**new** Employee(4, 103, "D", "HR"));  empList.add(**new** Employee(5, 105, "D", "Developer"));  **return** empList;  }  } |
| **package** filter.and.forEach;  **import** java.util.List;  **import** java.util.stream.Collectors;  **public** **class** EmployeeService {  **public** **static** List<Employee> getTaxUserList(String inputTax) {  /\* if (inputTax.equalsIgnoreCase("TAX")) {  return EmployeeDao.getEmployeeList().stream().filter(emp -> emp.getSalary() >= 103)  .collect(Collectors.toList());  } else {  return EmployeeDao.getEmployeeList().stream().filter(emp -> emp.getSalary() < 102)  .collect(Collectors.toList());  } \*/  // Turnory Operator  **return** (inputTax.equalsIgnoreCase("TAX")  ? EmployeeDao.*getEmployeeList*().stream().filter(emp-> emp.getSalary() >=103)  .collect(Collectors.*toList*())  :EmployeeDao.*getEmployeeList*().stream().filter(emp->emp.getSalary()<102)  .collect(Collectors.*toList*()));  }  **public** **static** **void** main(String[] args) {  // System.out.println(getTaxUserList("TAX"));  System.***out***.println(*getTaxUserList*("Non Tax"));//  }  } |
| [Employee [empId=1, salary=100, name=A, detptName=Defence], Employee [empId=2, salary=101, name=B, detptName=Railway]] // Non Tax |
| [Employee [empId=4, salary=103, name=D, detptName=HR], Employee [empId=5, salary=105, name=D, detptName=Developer]] // Tax |

# Java 8 Stream - How to Sort a List using lambda | Example |

|  |
| --- |
| **package** SortListUsingLambda;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.Comparator;  **import** java.util.List;  **public** **class** SortNameListUsingLambda {  **public** **static** List<String> nameList(){    List<String> nameList= **new** ArrayList<String>();  nameList.add("Arun");  nameList.add("Bunty");  nameList.add("Chotey");  nameList.add("Durga");  nameList.add("Imran");  nameList.add("Zafar");  Collections.*sort*(nameList); // Normal java method to sort list in ascending order  Collections.*reverse*(nameList); // Normal java method to sort list in reverse order  // sorting the list using stream api and sorted method of java 8  nameList.stream().sorted().forEach(name->System.***out***.println(name));  nameList.stream().sorted(Comparator.*reverseOrder*()).forEach(name->System.***out***.println(name));  // Just passed Comparator.*reverseOrder*() inside the sorted method to reverse list  **return** nameList;    }  **public** **static** **void** main(String[] args) {    System.***out***.println(*nameList*());  }  } |
| Zafar  Imran  Durga  Chotey  Bunty  Arun  [Arun, Bunty, Chotey, Durga, Imran, Zafar] |

**Note**: Since here we have taken the primitive type to be sorted that is why have passed the listName directly inside the method, but incase custom class where we have to sort the object based on its field or variable in that case we will use Comparator interface.

In the below example we have written anonymous method to sort (traditional way to sort object) and then we have used java 8 stream API lambda expression to sort the object for integer (salary) and String name wise.

|  |
| --- |
| **public** **static** **void** main(String[] args) {    System.***out***.println(*nameList*());    List<Employee> empList = EmployeeDao.*getEmployeeList*();  List<Employee> empList1 = EmployeeDao.*getEmployeeList*();  /\* Collections.sort(empList, new Comparator<Employee>() {  @Override  public int compare(Employee obj1, Employee obj2) {  Integer emp1= obj1.getSalary();  Integer emp2= obj2.getSalary();  return emp2.compareTo(emp1);  }  }); \*/    System.***out***.println("Sort by integer salary");  Collections.*sort*(empList, (Employee obj1, Employee obj2)-> (**int**)(obj2.getSalary() - obj1.getSalary()));  System.***out***.println(empList);    System.***out***.println("Sort by String Name");  Collections.*sort*(empList1,(Employee obj1, Employee obj2)->(obj1.getName().compareTo(obj2.getName())));  System.***out***.println(empList1);  }  } |
| Sort by integer salary  [Employee [empId=5, salary=105, name=D, detptName=Developer], Employee [empId=4, salary=103, name=D, detptName=HR], Employee [empId=3, salary=102, name=C, detptName=IT], Employee [empId=2, salary=101, name=B, detptName=Railway], Employee [empId=1, salary=100, name=A, detptName=Defence]]  Sort by String Name  [Employee [empId=1, salary=100, name=A, detptName=Defence], Employee [empId=2, salary=101, name=B, detptName=Railway], Employee [empId=3, salary=102, name=C, detptName=IT], Employee [empId=4, salary=103, name=D, detptName=HR], Employee [empId=5, salary=105, name=D, detptName=Developer]] |

Sorting the object using stream api:

|  |
| --- |
| System.***out***.println("Sort by String Name using stream api");  empList.stream().sorted((Employee obj1, Employee obj2)-> (obj1.getName().compareTo(obj2.getName()))).forEach(System.***out***::println);    System.***out***.println("Sort by integer salary using stream api");  empList1.stream().sorted((Employee obj1, Employee obj2)-> (obj1.getName().compareTo(obj2.getName()))).forEach(System.***out***::println);  System.***out***.println("Sort by integer salary using stream api and sorted method");  empList.stream().sorted(**Comparator.*comparing*(emp->emp.getSalary()**)).forEach(System.***out***::println);    System.***out***.println("Sort by integer salary using stream api and sorted method using method reference operator");  empList.stream().sorted(Comparator.*comparing*(**Employee::getSalary**)).forEach(System.***out***::println); |
| Sort by String Name using stream api  Employee [empId=1, salary=100, name=A, detptName=Defence]  Employee [empId=2, salary=101, name=B, detptName=Railway]  Employee [empId=3, salary=102, name=C, detptName=IT]  Employee [empId=5, salary=105, name=D, detptName=Developer]  Employee [empId=4, salary=103, name=D, detptName=HR]  Sort by integer salary using stream api  Employee [empId=1, salary=100, name=A, detptName=Defence]  Employee [empId=2, salary=101, name=B, detptName=Railway]  Employee [empId=3, salary=102, name=C, detptName=IT]  Employee [empId=4, salary=103, name=D, detptName=HR]  Employee [empId=5, salary=105, name=D, detptName=Developer] |

So here we have used several approach to sort the object -1 Using lambda expression then 2- optimized lambda expression and 3- using method reference.

# Java 8 Stream - How to Sort a Map using lambda | Example |

|  |
| --- |
| **package SortMapusingLambda;**  **import java.util.ArrayList;**  **import java.util.Collections;**  **import java.util.Comparator;**  **import java.util.HashMap;**  **import java.util.List;**  **import java.util.Map;**  **import java.util.Map.Entry;**  **public class SortMapUsingLambda {**  **public static void main(String[] args) {**  **Map<String , Integer> map = new HashMap<String, Integer>();**  **map.put("one", 1);**  **map.put("Three", 3);**  **map.put("Five", 5);**  **map.put("Six", 6);**  **map.put("Nine", 9);**  **System.out.println("map entries::"+map);**  **System.out.println("Convert Map into list");**  **List<Entry<String, Integer>> list = new ArrayList<Entry<String, Integer>>(map.entrySet());**  **System.out.println("list of Map::"+list);**    **System.out.println("Now sort the list having map entries");**      **// Old way to sort using Comparator Interface explicitly**  **Collections.sort(list, new Comparator<Entry<String, Integer>>() {**  **@Override**  **public int compare(Entry<String, Integer> o1, Entry<String, Integer> o2) {**  **return o2.getKey().compareTo(o1.getKey());**  **}**  **});**    **// sorting Using Lambda Expression with map key**  **Collections.sort(list, (Entry<String, Integer> o1, Entry<String, Integer> o2) -> o1.getKey().compareTo(o2.getKey()));**  **// Above lambda expression can be more optimized as given below**  **Collections.sort(list, (o1, o2) -> o1.getKey().compareTo(o2.getKey()));**    **// sorting Using Lambda Expression with map value**  **Collections.sort(list, (Entry<String, Integer> o1, Entry<String, Integer> o2) -> (int)(o1.getValue()- o2.getValue()))));**      **System.out.println(list);**  **System.out.println("Printing map sorted in the list using traditional for each loop");**  **for(Entry<String , Integer> entry : list) {**  **System.out.println(entry.getKey()+"::"+entry.getValue());**  **}**  **}**  **}** |

**Sorting a map using stream API:**

|  |  |
| --- | --- |
| **package** SortMapusingLambda;  **import** java.util.ArrayList;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Map.Entry;  **public** **class** SortMapUsingStreamAPI {  **public** **static** **void** main(String[] args) {    Map<String , Integer> map = **new** HashMap<String, Integer>();  map.put("one", 1);  map.put("Three", 3);  map.put("Five", 5);  map.put("Six", 6);  map.put("Nine", 9);  System.***out***.println("map entries::"+map);  System.***out***.println("Convert Map into list");  List<Entry<String, Integer>> list = **new** ArrayList<Entry<String, Integer>>(map.entrySet());  System.***out***.println("list of Map::"+list);    map.entrySet().stream().sorted(**Map.Entry.*comparingByKey*()**).forEach(entry-> System.***out***.println(entry));  // Or  System.***out***.println();  map.entrySet().stream().sorted(**Map.Entry.*comparingByValue*()**).forEach(System.***out***::println);  }  } | |
| map entries::{Five=5, Six=6, one=1, Nine=9, Three=3}  Convert Map into list  list of Map::[Five=5, Six=6, one=1, Nine=9, Three=3] | **Using - Map.Entry.*comparingByKey*()**  Five=5  Nine=9  Six=6  Three=3  one=1  **Using - Map.Entry.*comparingByValue*()**  one=1  Three=3  Five=5  Six=6  Nine=9 |

1- As here we can see that we have sort the method directly using stream () API and its sorted method just by comparing key or value.

2- We have print the map using forEach loop in two different ways. In the first forEach using lambda expression and in the second forEach using method reference.

**Sorting the complex Employee Map object:**

Let’s create Map<Employee, Integer>. Here we have take Employee class type as key and Integer as value.

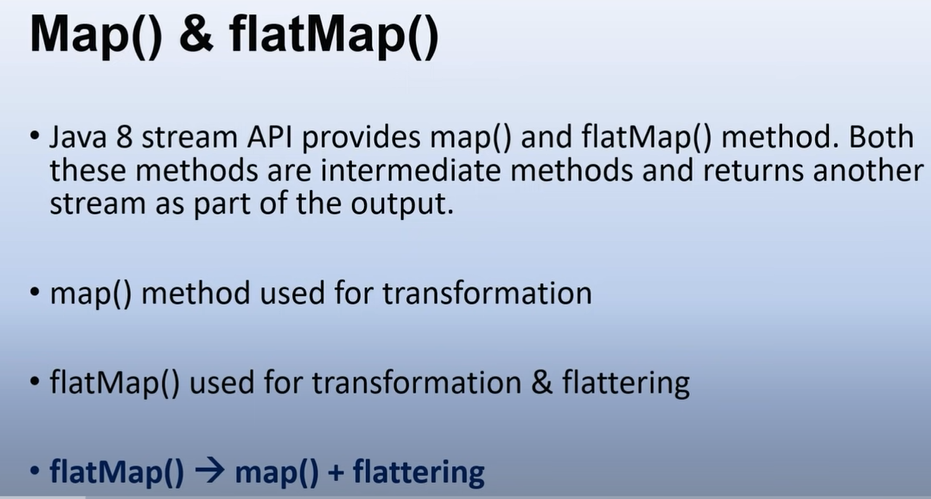
|  |
| --- |
| **package** SortMapusingLambda;  **import** java.util.ArrayList;  **import** java.util.Comparator;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** java.util.TreeMap;  **import** filter.and.forEach.Employee;  **public** **class** SortMapEMployeeUsingStream {  **public** **static** **void** main(String[] args) {  Map<Employee, Integer> employeeMap = **new** TreeMap<Employee, Integer>(**new** Comparator<Employee>() {  @Override  **public** **int** compare(Employee o1, Employee o2) {  // **TODO** Auto-generated method stub  **return** o2.getSalary() - o1.getSalary();  }  });  employeeMap.put(**new** Employee(1, 100, "A", "Defence"), 100);  employeeMap.put(**new** Employee(2, 101, "B", "Railway"), 200);  employeeMap.put(**new** Employee(3, 102, "C", "IT"), 300);  employeeMap.put(**new** Employee(4, 103, "D", "HR"), 400);  // Here we have create one TreeMap where taken employee object as key and value  // as integer.  // So first let's sort using traditional way, i.e. Creating Comparator interface  // and passing the object inside the TreeMap  System.***out***.println("Employee Map::" + employeeMap);  List<Entry<Employee, Integer>> empList = **new** ArrayList<>(employeeMap.entrySet());  System.***out***.println("Employee List::" + empList);  // Let's create another map object and sort the using lambda expression  Map<Employee, Integer> empMap2 = **new** TreeMap<Employee, Integer>(  (Employee o1, Employee o2) -> o1.getEmpId() - o2.getEmpId());  empMap2.put(**new** Employee(1, 100, "A", "Defence"), 100);  empMap2.put(**new** Employee(2, 101, "B", "Railway"), 200);  empMap2.put(**new** Employee(3, 102, "C", "IT"), 300);  empMap2.put(**new** Employee(4, 103, "D", "HR"), 400);  System.***out***.println("Sort Using Lambda Expression:-" + empMap2);  System.***out***.println();  System.***out***.println("Sorting the map using stream api and print using method reference");  employeeMap.entrySet().stream().sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getDetptName)))  .forEach(System.***out***::println);  System.***out***.println("\*\*\*\*\*\*\*Sorting the map using stream api and print using method reference\*\*\*\*\*");  empMap2.entrySet().stream().sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getName)))  .forEach(name -> System.***out***.println(name));  System.***out***.println("\*\*\*\*\*\*reversing the order ");  employeeMap.entrySet().stream()  .sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getDetptName).reversed()))  .forEach(System.***out***::println);  System.***out***.println("\*\*\*\*\*\*reversing the order\*\*\*\*\*");  empMap2.entrySet().stream().sorted(Map.Entry.*comparingByKey*(Comparator.*comparing*(Employee::getName).reversed())).forEach(name -> System.***out***.println(name));  }  } |

So we see the huge difference while sorting in traditional way and using stream api.

While using stream api we can reduce multiple line and in one line only we can sort the completed object

|  |
| --- |
| Employee Map::{Employee [empId=4, salary=103, name=D, detptName=HR]=400, Employee [empId=3, salary=102, name=C, detptName=IT]=300, Employee [empId=2, salary=101, name=B, detptName=Railway]=200, Employee [empId=1, salary=100, name=A, detptName=Defence]=100}  Employee List::  [Employee [empId=4, salary=103, name=D, detptName=HR]=400, Employee [empId=3, salary=102, name=C, detptName=IT]=300, Employee [empId=2, salary=101, name=B, detptName=Railway]=200, Employee [empId=1, salary=100, name=A, detptName=Defence]=100]  Sort Using Lambda Expression:-  {Employee [empId=1, salary=100, name=A, detptName=Defence]=100, Employee [empId=2, salary=101, name=B, detptName=Railway]=200, Employee [empId=3, salary=102, name=C, detptName=IT]=300, Employee [empId=4, salary=103, name=D, detptName=HR]=400}  Sorting the map using stream api and print using method reference  Employee [empId=1, salary=100, name=A, detptName=Defence]=100  Employee [empId=4, salary=103, name=D, detptName=HR]=400  Employee [empId=3, salary=102, name=C, detptName=IT]=300  Employee [empId=2, salary=101, name=B, detptName=Railway]=200  \*\*\*\*\*\*\*Sorting the map using stream api and print using method reference\*\*\*\*\*  Employee [empId=1, salary=100, name=A, detptName=Defence]=100  Employee [empId=2, salary=101, name=B, detptName=Railway]=200  Employee [empId=3, salary=102, name=C, detptName=IT]=300  Employee [empId=4, salary=103, name=D, detptName=HR]=400  \*\*\*\*\*\*reversing the order  Employee [empId=2, salary=101, name=B, detptName=Railway]=200  Employee [empId=3, salary=102, name=C, detptName=IT]=300  Employee [empId=4, salary=103, name=D, detptName=HR]=400  Employee [empId=1, salary=100, name=A, detptName=Defence]=100  \*\*\*\*\*\*reversing the order\*\*\*\*\*  Employee [empId=4, salary=103, name=D, detptName=HR]=400  Employee [empId=3, salary=102, name=C, detptName=IT]=300  Employee [empId=2, salary=101, name=B, detptName=Railway]=200  Employee [empId=1, salary=100, name=A, detptName=Defence]=100 |

# Java 8 Streams | map () & flatMap() Example |

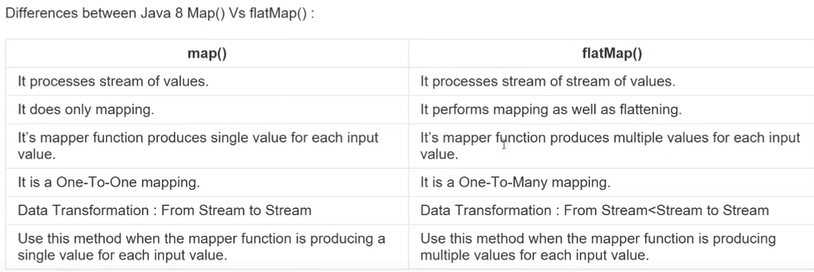


|  |  |
| --- | --- |
|  | Since map is used for data manipulation or data transformation so it takes stream data as input and return result as stream only.  Second point is the method prototype for map. |
|  | Since flatMap is used for data transformation as well as flattering data so it takes stream of streams as input and then it convert the stream of streams into a single stream(which is nothing but data flattering) and it return an another stream as output. |
|  | 1. We are converting a stream lower case albhabates into upper case so this is nothing byt data transformation. 2. In the second we have a stream of streams which we are converting into one stream so this is nothing but flattering. |

Hands On-

1. Created one POJO class with following fields [id, name, email, PhoneList]
2. Created a DAO layer class where we have hard coded and returns the list of customer

|  |
| --- |
| **package MapAndFlatMap;**  **import java.util.List;**  **import java.util.stream.Collectors;**  **public class MapVsFlatMapMainTest {**  **public static void main(String[] args) {**    **List<Customer> customerList = CustomerDao.*getCustomerList*();**  **System.*out*.println ("List of customers.....");**  **customerList.stream().map(customer->customer.getName()).collect(Collectors.*toList*()).forEach(System.*out*::println);**  **// customer->customer.getEmail() >> One to one Mapping**  **List<String> emailList = customerList.stream().map(customer->customer.getEmail()).collect(Collectors.*toList*());**  **System.*out*.println("Email List::"+emailList);**  **List<List<String>> listOfEmailList = customerList.stream().map(customer->customer.getPhones()).collect(Collectors.*toList*());**  **System.*out*.println("listOfEmailList::"+listOfEmailList);**  **System.*out*.println("Let get the flat map of the phone list....");**  **// customer-> customer.getPhones() ->> one to many mapping**  **List<String> flatMapListofPhones = customerList.stream().flatMap(customer->**  **customer.getPhones().stream()).collect(Collectors.*toList*());**  **System.*out*.println("flatMapListofPhones::"+flatMapListofPhones);**  **}**  **}** |
| List of customers.....  A  B  C  D  Email List::[A@Gmail.com, B@Gmail.com, C@Gmail.com, [D@Gmail.com](mailto:D@Gmail.com)]  listOfEmailList::[[123456, 123123], [5555666, 44446666], [77775555, 7777999], [222222, 45623133]]  Let get the flat map of the phone list....  flatMapListofPhones::[123456, 123123, 5555666, 44446666, 77775555, 7777999, 222222, 45623133] |



# Java 8 Streams | Optional Usage and Best Practices |

1. Optional class introduced in java-8
2. As we know that java is purely object oriented language and if any object contains value then it will be executed but if contains null then it throws null pointer exception

|  |  |
| --- | --- |
|  | **Here we can see that object, customer1 contains the id and name both values and whenever we try to get id or name using its getter method then we will get the data.**  **But in object customet2 contains only id and name value is null. So whenever we try to get name value using its getter method then it will give null value and if we try to perform any operation like name.toUpperCase () then we will get null pointer exception.**  **So here to avoid this null pointer exception we can check using if-else condition and if it contains null then we can skip that operation and else part will be performed.**  **Note: Since here we two values in the customer object so it is easy to check using if-else condition but what if we have 100 object then in this case we have to write 100 if and else condition which is not looking a good practice.**  **To avoid this kind of practices or to avoid unpredictable null pointer exception java-8 introduces one class called Optional.** |

**public final class Optional<T> extends** [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) **:**

**A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.**

**Additional methods that depend on the presence or absence of a contained value are provided, such as**[**orElse()**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#orElse-T-)**(return a default value if value not present) and**[**ifPresent()**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ifPresent-java.util.function.Consumer-)**(execute a block of code if the value is present).**

|  |
| --- |
| **package** MapAndFlatMap;  **import** java.util.Arrays;  **import** java.util.Optional;  **public** **class** OptionalMainTest {  **public** **static** **void** main(String[] args) {  Customer customer = **new** Customer(1, "A", "arun@gmail.com", Arrays.*asList*("123456","789456"));  Customer customer1= **new** Customer(2, "B", **null**, Arrays.*asList*("123456","789456"));  // empty:- Creates an empty object and does not any argument  // of:- :- Creates an Optional object with the object values otherwise throws null pointer exception if value  of object is null so if we 100 ensure then we should use this method  // ofNullable:- So this method returns an object with value or returns an empty object: so this is combination  of [empty + of] method  Optional<Object> emptyObject = Optional.*empty*();  System.***out***.println("Empty Object:- "+emptyObject);  Optional<String> optionalOfObject = Optional.*of*(customer.getEmail());  System.***out***.println("optionalOfObject::-"+optionalOfObject);  Optional<String> ofNullable = Optional.*ofNullable*(customer.getEmail());  System.***out***.println("ofNullable::"+ofNullable);  System.***out***.println(ofNullable.get());  Optional<String> ofNullable1 = Optional.*ofNullable*(customer1.getEmail());  System.***out***.println("ofNullable1::"+ofNullable1);  // System.out.println(ofNullable1.get()); since value of email is null that is why we will get "NoSuchElementException: No value present, so to avoid exception we have check first using isPresent method    **if**(ofNullable1.isPresent()) {  System.***out***.println(ofNullable1.get());  }  // ifPresent() will returns false if value is not present and if block will not be executed  // returns the default value if object does not contains any value. and for that we can use **orElse()** method  System.***out***.println (ofNullable1.orElse ("default@gmail.com"));  **// orElse: This is equal to if and else block. When if block is false then execute else block with default value.**  }  } |
| **Output:**  Empty Object:- Optional.empty  optionalOfObject::-Optional[arun@gmail.com]  ofNullable::Optional[arun@gmail.com]  arun@gmail.com  ofNullable1::Optional.empty  default@gmail.com |

**Optional Class using with stream() API**

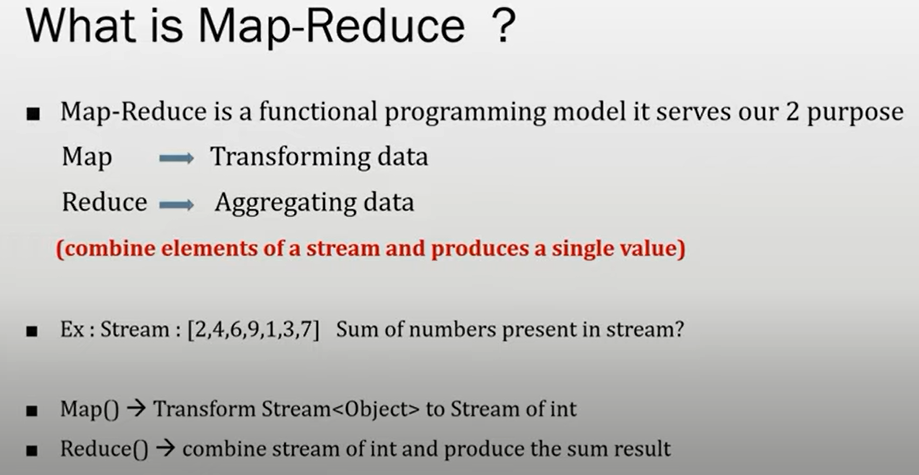
|  |
| --- |
| **package** MapAndFlatMap;  **import** java.util.List;  **public** **class** OptionalMainTestWithStreamAPI {  **public** **static** Customer getCustomerByEmailId(String email) {  List<Customer> customerList = CustomerDao.*getCustomerList*();  **return** customerList.stream()  .filter(customer->customer.getEmail().equals(email))  .findAny()  //.findAny().get(); // it just simply throws  //.orElse(new Customer()); // It will just simply return blank object without exception  .orElseThrow(()-> **new** IllegalArgumentException("No email found"));  }  **public** **static** **void** main(String[] args) {  *getCustomerByEmailId*("A1@Gmail.com");  }  }  **Note: here based on the argument passed we are trying to find out email id if found then return the same else throw custom exception.** |
| **package** MapAndFlatMap;  **public** **class** OptionalMainTestWithStreamAPI {  **public** **static** Customer getCustomerByEmailId(String email) {  List<Customer> customerList = CustomerDao.*getCustomerList*();  Customer customer2 = customerList.stream()  .filter(customer->customer.getEmail().equals(email))  // .findAny()  .findAny().get(); // it just simply throws  //.orElse(new Customer()); // It will just simply return blank object without exception  // .orElseThrow(()-> new IllegalArgumentException("No email found"));  System.***out***.println(customer2.getEmail());    **return** customer2;  }  **public** **static** **void** main(String[] args) {  *getCustomerByEmailId*("A@Gmail.com");  }  } |
| **Output:** A@Gmail.com |

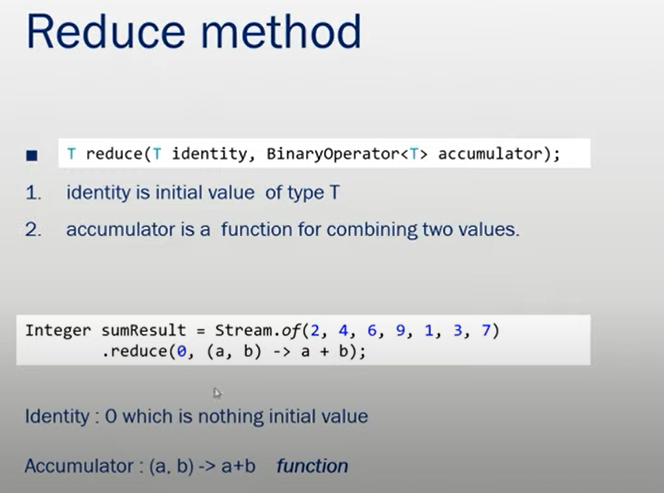
So as per above example and definition Optional class does a lot in comparison to simple not null check.

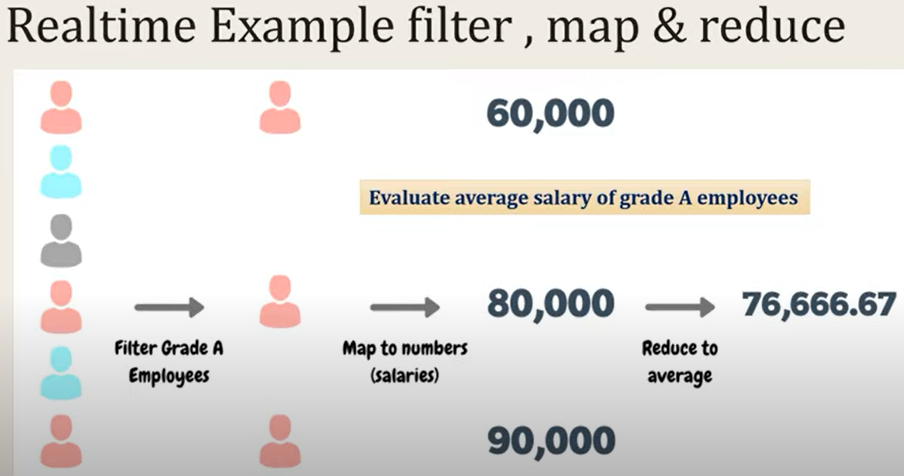
1. We can simply return empty object
2. We can return the object with its value
3. We can return the object with its value and if object contains null then we can return the default value
4. We can return the object with its value and if object contains null then we can throw custom exception also
5. ifPresent() can used to check the null condition
6. in case we have 100 object and

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| **static <T>**[**Optional**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)**<T>** | [**empty**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#empty--)**()**  **Returns an empty Optional instance.** |
| **static <T>**[**Optional**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)**<T>** | [**of**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#of-T-)**(T value)**  **Returns an Optional with the specified present non-null value.** |
| **static <T>**[**Optional**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)**<T>** | [**ofNullable**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ofNullable-T-)**(T value)**  **Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.** |
| **boolean** | [**isPresent**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#isPresent--)**()**  **Return true if there is a value present, otherwise false.** |

# Java 8 Streams | map() and reduce() Example |







|  |  |
| --- | --- |
| **import** java.util.Arrays;  **import** java.util.List;  **public** **class** MapAndReduseAggrigate {  **public** **static** **void** main(String[] args) {  List<Integer> numberlist = Arrays.*asList*(1,2,3,4);  **int** sum=0;  // Old ways to add numbers  **for**(**int** i=0;i<=numberlist.size();i++) {  sum=sum+i;  }  System.***out***.println("Sum Using For loop:="+sum);  **int** sum1=0;  **for**(**int** i : numberlist) {  sum1=sum1+i;  }  System.***out***.println("Sum Using Foreach loop:="+sum1);    **int** sumUsingLambda = numberlist.stream().mapToInt(num->num).sum();  System.***out***.println("Sum Using stream and map function="+sumUsingLambda);    Integer sumUsingReduce = numberlist.stream().reduce(0, (a,b)->(a+b));  System.***out***.println("Sum Using using reduce:="+sumUsingReduce);    Integer sumUsingReduce1 = numberlist.stream().reduce(1, (a,b)->(a\*b));  System.***out***.println("multiply Using reduce:="+sumUsingReduce1);    // finding greater number using reduce  Integer maxValue = numberlist.stream().reduce(0,(a,b)->a>b?a:b);  System.***out***.println("Max-Value using reduce::"+maxValue);  Optional<Integer> maxUsingMethodReference = numberlist.stream().reduce(Integer::*max*);  System.***out***.println("maxUsingMethodReference::"+maxUsingMethodReference);  }  } | |
| Output:  Sum Using For loop:=10  Sum Using Foreach loop:=10  Sum Using stream and map function=10  Sum Using reduce function:=10  multiply Using reduce:=24  Max-Value using reduce::4  maxUsingMethodReference::Optional[4] | 1. so here using(for loop) the traditional way we have added the integer value 2. Added the same number using for each loop 3. Added the same number using map[mapToInt 🡺using sum function added all numbers 4. Added the same number using reduce function 5. Multiply the same number using reduce 6. Find the max number using reduce 7. Find the max using reduce method reference and it returns Optional object with value   **Note: So basically identity and accumulator plays a very imp role, so we just need know accumulator syntax:** |

<U> U reduce(U identity, BiFunction<U, ? **super** T, U> accumulator, BinaryOperator<U> combiner);

**Find the greater string using reduce:**

|  |
| --- |
| **package MapAndReduceAggrigate;**  **import java.util.Arrays;**  **import java.util.List;**  **public class FindGreaterStingUsingReduce {**  **public static void main(String[] args) {**  **List<String> wordslist= Arrays.*asList*("a","bb","ccc","dddd");**    **String greaterWord = wordslist.stream().reduce((word1,word2)->word1.length()>word2.length()?word1:word2).get();**  **System.*out*.println("greaterWord::"+greaterWord);**  **}**  **}** |
| **Output: greaterWord::dddd** |

**Complex Example:**

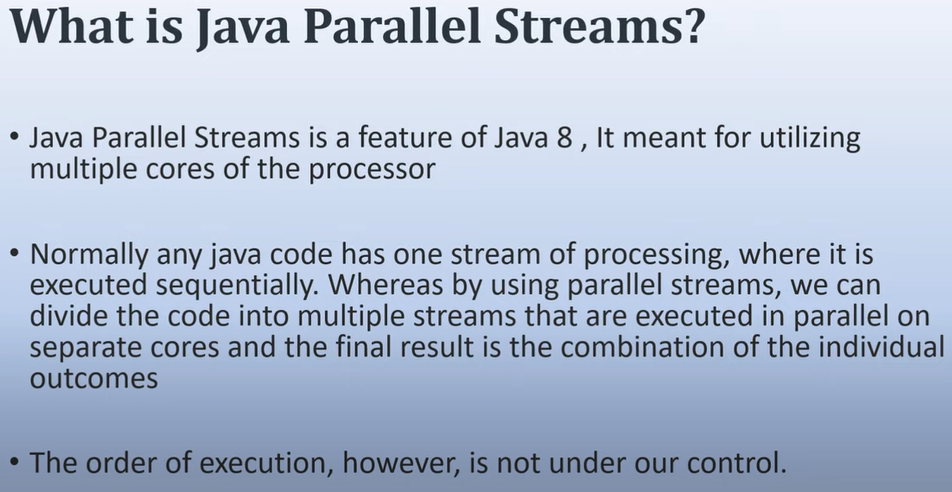
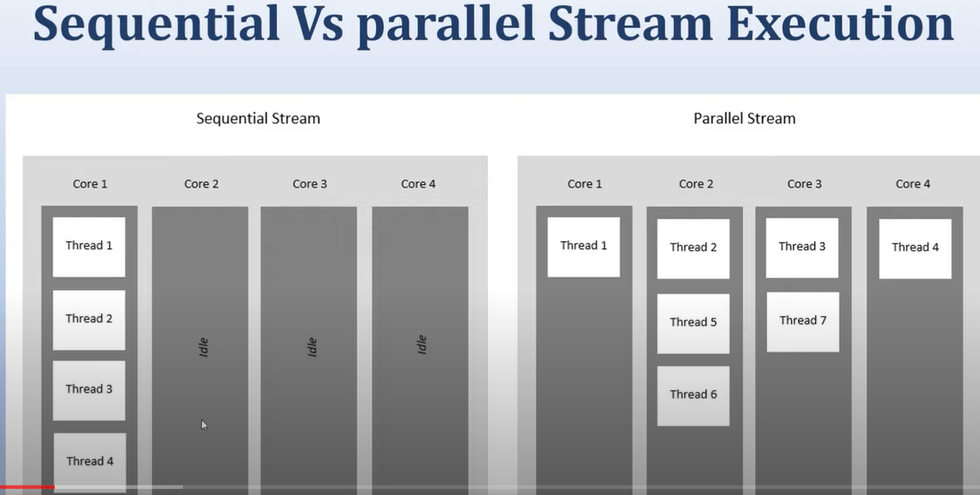
Let’s say we have a list of employees having id, name, grade, salary.

1. Find out all the employees whose dept is IT and find their salary average
2. Find out all the employees whose grade is IT and find their salary sum

|  |
| --- |
| **package filter.and.forEach;**  **public class Employee {**  **private int empId;**  **private int salary;**  **private String name;**  **private String detptName;**    **public Employee(int empId, int salary, String name, String detptName) {**  **super();**  **this.empId = empId;**  **this.salary = salary;**  **this.name = name;**  **this.detptName = detptName;**  **}**    **@Override**  **public String toString() {**  **return "Employee [empId=" + empId + ", salary=" + salary + ", name=" + name + ", detptName=" + detptName + "]";**  **}**  **}** |
| **package filter.and.forEach;**  **import java.util.ArrayList;**  **import java.util.List;**  **public class EmployeeDao {**  **public static List<Employee> getEmployeeList(){**  **List<Employee> empList= new ArrayList<Employee>();**  **empList.add(new Employee(1, 100, "A", "Defence"));**  **empList.add(new Employee(2, 101, "B", "Railway"));**  **empList.add(new Employee(3, 102, "C", "IT"));**  **empList.add(new Employee(4, 103, "D", "IT"));**  **empList.add(new Employee(5, 105, "D", "Developer"));**  **return empList;**  **}**  **}** |
| **package MapAndReduceAggrigate;**  **import filter.and.forEach.EmployeeDao;**  **public class FindAvgAndSumOfSalaryUsingReduce {**  **public static void main(String[] args) {**  **double asDouble = EmployeeDao.*getEmployeeList*().stream()**  **.filter(dept->dept.getDetptName().equalsIgnoreCase("IT"))**  **.map(emp->emp.getSalary())**  **.mapToInt(i->i)**  **// .average().getAsDouble();**  **.sum();**  **System.*out*.println(asDouble);**  **}**  **}** |
| **Output:** 205.0 |

So we can say that map and reduce method is very-2 strong feature in java-8

# Java 8 Parallel Streams | Parallel data processing and performance Example |

As here in case of sequential stream the task are getting executed by multiple threads sequentially one by one, while in case of parallel stream tasks are being executed by multiple parallel. When multiple treads try to execute, then we can’t predict their order of execution.

Here in sequential execution the other core are sitting ideal i.e. not able to utilize the existing resource, while in case of Parallel stream all the core processor are being utilized properly and hence increase the performances.

In single stream the order of execution is certain and we can expect the proper output.

**Hands On:**

|  |
| --- |
| **package StreamVsParallelStream;**  **import java.util.Arrays;**  **import java.util.List;**  **public class StreamVsParallelStreamExample {**  **public static void main(String[] args) {**  **long start=0;**  **long end=0;**  **long start1=0;**  **long end1=0;**  **List<Integer> integerList = Arrays.*asList*(1,2,3,4,5);**  **start=System.*currentTimeMillis*();**  **integerList.stream().forEach(i->System.*out*.println(i));**  **end=System.*currentTimeMillis*();**  **System.*out*.println("Plane stream took time:-"+(end-start));**  **System.*out*.println("===========parallel stream=================");**  **start1=System.*currentTimeMillis*();**  **integerList.parallelStream().forEach(i->System.*out*.println(i));**  **end1=System.*currentTimeMillis*();**  **System.*out*.println("Parallel stream took time:-"+(end1-start1));**  **}** |

Output:

|  |  |
| --- | --- |
| 1  2  3  4  5  Plane stream took time:-22  ======================================  3  5  4  2  1  Parallel stream took time:-8 | As here we can see that in the plane stream it runs sequentially and here we are getting the correct output, while in case of parallel stream it runs parallel and here we cannot give the guarantee of the output.  The plane stream or sequential stream takes more time to complete it execution in compare to parallel stream.  So parallel stream performance is better than sequential stream. |

|  |
| --- |
| **IntStream.*range*(1, 100).parallel().forEach(i->System.*out*.println(i));**  Here we have created stream of integer ranging 1 to 100 and then printing parallel using forEach loop. |

**Let’s print the thread executing sequential stream and parallel stream:**

|  |
| --- |
| // Sequential String thread name:  IntStream.*range*(1, 5).forEach(t->{  System.***out***.println("Thread Name::"+Thread.*currentThread*().getName()+":"+t);  });  // Thread name of Parallel stream  IntStream.*range*(1, 5).parallel().forEach(t->{  System.***out***.println("ParallelStreamThreadName::"+Thread.*currentThread*().getName()+":"+t);  }); |
| Thread Name::main:1  Thread Name::main:2  Thread Name::main:3  Thread Name::main:4  ParallelStreamThreadName::ForkJoinPool.commonPool-worker-5:4  ParallelStreamThreadName::ForkJoinPool.commonPool-worker-7:2  ParallelStreamThreadName::ForkJoinPool.commonPool-worker-3:1  ParallelStreamThreadName::main:3 |

So here we can see that in case of sequential stream all the number getting printed by main thread and we are getting correct output.

While in case of parallel stream two threads are being used to print the numbers 1- ForkJoinPool.commonPool-worker 2- main. And here we cannot give guarantee for output and here we can see the thread switching, whichever getting the lock then start executing the process.

**Complex Example: Let’s create an Employee class have id, name, and salary fields. Let’s create 1000 employee using for loop. Now let’s see the difference of performance using sequential and parallel stream. Here we will use filter and**

|  |
| --- |
| **public** **class** Employee {  **private** **int** id;  **private** String name;  **private** **double** salary;  **private** String email;  **public** Employee(**int** id, String name, **double** salary, String email) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.salary = salary;  **this**.email = email;  } |

Now let’s create EmployeeDAO class where we will fetch the record from DB

**EmployeeDAO.java: Created 1000 employee object as given below.**

|  |
| --- |
| **package StreamVsParallelStream;**  **import java.util.ArrayList;**  **import java.util.List;**  **import java.util.Random;**  **public class EmployeeDAO {**  **public static List<Employee> getEmpList(){**    **List<Employee> empList= new ArrayList<Employee>();**  **for(int i=1;i<=1000;i++) {**  **empList.add(new Employee(i, "Emp-"+i, new Random().nextInt(1000\*10), "email"+1+"@gmail.com"));**  **}**  **return empList;**  **}**  **}** |

Now in the mail class extract the employee list and get the each employee salary and calculate the average of salary using stream api and the ParallelStream () api and check the performance.

|  |
| --- |
| **List<Employee> empList = EmployeeDAO.*getEmpList*();**  **start=System.*currentTimeMillis*();**  **double salWithStream = empList.stream().map(emp->emp.getSalary()).mapToDouble(i->i).average().getAsDouble();**  **end=System.*currentTimeMillis*();**  **System.*out*.println("Time to Execute Stream::"+(end-start));**  **start=System.*currentTimeMillis*();**  **double salWithParallelStream = empList.parallelStream().map(Employee::getSalary).mapToDouble(i->i)**  **.average().getAsDouble();**  **end=System.*currentTimeMillis*();**  **System.*out*.println("Time to Execute Parallel Stream::"+(end-start));** |
| Output:  Time to Execute Stream::66  Time to Execute Parallel Stream::4  **Note**: So we can observer the huge difference of performance between stream and parallel stream. |

**Java-8 HashMap enhancement:**

|  |
| --- |
| **package** Java8EnhanceHashmap;  **public** **class** Employee {  **private** **int** id;  **private** String name;  **private** **double** salary;  **private** String email;    **public** Employee(**int** id, String name, **double** salary, String email) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.salary = salary;  **this**.email = email;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  // result = prime \* result + ((email == null) ? 0 : email.hashCode());  // result = prime \* result + id;  // result = prime \* result + ((name == null) ? 0 : name.hashCode());  // long temp;  // temp = Double.doubleToLongBits(salary);  // result = prime \* result + (int) (temp ^ (temp >>> 32));  **return** 100;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  Employee other = (Employee) obj;  **if** (email == **null**) {  **if** (other.email != **null**)  **return** **false**;  } **else** **if** (!email.equals(other.email))  **return** **false**;  **if** (id != other.id)  **return** **false**;  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;  **if** (Double.*doubleToLongBits*(salary) != Double.*doubleToLongBits*(other.salary))  **return** **false**;  **return** **true**;  }  @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + ", salary=" + salary + ", email=" + email + "]";  }  } |

**Employee.java – with fields, override hashCode (), equals() and toString() method**

**Department.java – with fields, override hashCode (), equals() and toString() method**

|  |
| --- |
| **package** Java8EnhanceHashmap;  **public** **class** Department {  **private** **int** id;  **private** String name;  **private** String deptLocation;    **public** Department(**int** id, String name, String deptLocation) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.deptLocation = deptLocation;  }  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + ((deptLocation == **null**) ? 0 : deptLocation.hashCode());  result = prime \* result + id;  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if** (obj == **null**)  **return** **false**;  **if** (getClass() != obj.getClass())  **return** **false**;  Department other = (Department) obj;  **if** (deptLocation == **null**) {  **if** (other.deptLocation != **null**)  **return** **false**;  } **else** **if** (!deptLocation.equals(other.deptLocation))  **return** **false**;  **if** (id != other.id)  **return** **false**;  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;  **return** **true**;  }  } |

Now main method:

|  |
| --- |
| **package** Java8EnhanceHashmap;  **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** HashMapMainTest {  **public** **static** **void** main(String[] args) {  Employee e1 = **new** Employee(1, "A", 101, "a@gmail.com");  Employee e2 = **new** Employee(2, "B", 102, "b@gmail.com");  Employee e3 = **new** Employee(3, "C", 103, "c@gmail.com");  Department d1 = **new** Department(101, "D1", "location1");  Department d2 = **new** Department(102, "D2", "location2");  Department d3 = **new** Department(103, "D3", "location3");  Map<Employee, Department> empDeptMap= **new** HashMap<Employee, Department>();  empDeptMap.put(e1, d1);  empDeptMap.put(e2, d2);  empDeptMap.put(e3, d3);    System.***out***.println(empDeptMap.get(e1));  System.***out***.println(empDeptMap.get(**new** Employee(1, "A", 101, "a@gmail.com")));// Will get null if don’t override hashCode method.  System.***out***.println(empDeptMap.get(e2));  }  }  // Here in the hashCode method for every value in the hash map we will get same has code and index number because in the HashMap code it is returning  // only same 100 integer value for every key. In this case all the hash map value will stored at the same index in the form linked list and thus a  // collision state will be created. Now in this case whenever we try to access the any value based on the key then again first of a hash code will be calculated  // and then again we will get the same index. Now it is assumed that every bucket will have one value and thus the time complexity in this condition is O(1)  // Now at the same index we will get multiple node having value and pointer to next node and now the time complexicity will O(n). Now in this case each and every  // node will be compared to get the exact value for that key. Hence the performance issue will created during the access. And this happens we don't override hashCode method properly.  // To avoid such situation, in java-8 there has been done some enhancement to improve the performance.  // Java-8 HashMap element use balanced tree instead of linked-list after a certain threshold is reached while storing the values. This improves the worst case performance from O (n) to O (log n).  // In other word in the case of collision state after a certain threshold is reached, the linked-list gets converted into balanced tree and this the reason complexicity changes from O(n) to O(log n). |

**Note**: Since every wrapper class like Integer, String by default implement hashCode () and equals () method. So we don’t need to override explicitly if key and value are taken as wrapper classes (like Integer or string) because in this case (wrapper class) when we try to store the HashMap entry into bucket then hashCode get calculated and we get the index and thus the HashMap value Is stored in the buckets.

But in case of custom class object it does not implement hashCode () and equals () method then it used the default hashCode and equal method provide by Object class and we cannot give the assured about the uniqueness of the hascode and its result during access using get() method.

So it is recommended to override hashCode and equals method for custom class.

|  |  |
| --- | --- |
|  |  |

Here Object means their content.

