**Why Lambdas?**

It enables functional programming.

Readable and concise coding

Easier-to use API and libraries

Enable support for parallel processing

**Prerequisites**: JDK 1.8

**Functional Programming:**

Why function programming?

We have function which must except action, based on that it should perform execution.

Example :

**public** **void** greet(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) {

\_\_\_\_\_\_\_\_\_\_\_\_();

}

Functions as values:

Inline value :

String name = “foo”;

As same can we assign block of code

aBlockOfCode = {

------------------------

--------------------- }

aBlockOfCode = **public** **void** perform() {

// **TODO** Auto-generated method stub

System.***out***.println("Hello world from perform");

}

Remove “public” it always public

Remove “perform” hence name of the function is aBlockOfCode

Remove “void” function will not return any values. Compiler identifies return type

aBlockOfCode = () {

// **TODO** Auto-generated method stub

System.***out***.println("Hello world from perform");

}

We need to add one new symbol to make it lambda expression (->)

aBlockOfCode = ()-> {

// **TODO** Auto-generated method stub

System.***out***.println("Hello world from perform");

}

If it is one line remove “{“ brasses

aBlockOfCode = () -> System.***out***.println("Hello world from perform");

greetingFunction = () -> System.***out***.println("Hello world from perform");

few more example:

doubleNumberFuction = **public** **int** doubleValue(**int** b) {

**return** b\*2;

}

doubleNumberFuction = (int b) -> b\*2;

addFunction = (int a, int b) -> a+b;

safeDividingFunction = (int a, int b) -> {

if(b==0) return 0;

return a/b;

}

stringLengthCountFunction = (String str) -> str.lenght();

**Return type of the lambda function is Interface**

**Class Greeting{**

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

// **TODO** Auto-generated method stub

Greeter greet = **new** Greeter();

greet.greet(**new** HelloWorldGreeting());

mygeeting greetingFunction = () -> System.***out***.println("Hello world from perform");

Myadd addFunction = (**int** a, **int** b) -> a+b;

}

**interface** mygeeting{

**void** foo();

}

**interface** Myadd{

**int** add(**int** a, **int** b);

}

}

Difference between

Greeting helloWorld = **new** HelloWorldGreeting();

Greeting greetingFunction = () -> System.***out***.println("Hello world from perform lambda");

greet.greet(helloWorld);

helloWorld.perform();

greetingFunction.perform(); // it looks like actually implementation of Greeting interface.

**Inline implementation of interface**

Greeting inlineImplentation = **new** Greeting() {// no name enormous class

**public** **void** perform() {

System.***out***.println("Inline implementation of interface");

}

};

Lambda function is One of the way to create enormous class

**Runnable using Lambda**

**public** **class** RannableExample {

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

// **TODO** Auto-generated method stub

Thread thread = **new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

System.***out***.println("Printable in the runnable");

}

});

thread.run();

Thread mylambdaThread = **new** Thread(() -> System.***out***.println("Printable in the runnable in lambda function"));

mylambdaThread.run();

}

}

Lambda function act like implementation of interface specified.

**Functional Interface:**

Interface having only one Abstract method that is called Functional interface.

Example : Rannable interface

Use Annotation

@FunctionalInterface – java.lang lib – default package no need to import

The very first point about lambda expression is that **y**ou can pass a lambda expression to a method which accepts a functional interface i.e. a method which is either annotated by @Functional annotation e.g. interfaces from java.util.function package, or an interface with single abstract method e.g. [Comparable](http://javarevisited.blogspot.sg/2014/02/java-comparable-example-for-natural-order-sorting.html), [Comparator](http://www.java67.com/2014/11/java-8-comparator-example-using-lambda-expression.html), [Runnable](http://www.java67.com/2016/01/7-differences-between-extends-thread-vs-implements-Runnable-java.html), [Callable](http://javarevisited.blogspot.sg/2016/08/useful-difference-between-callable-and-Runnable-in-Java.html), ActionListener etc.

Lambda expression is like **anonymous function**

Lambda expression in Java is of SAM type i.e. Single abstract method, which means you can pass a lambda expression to a method which expects an object of a class with exactly one abstract method.

Another example is Collections.sort() which accepts a Comparator, another interface with just one abstract method compare(), you can pass a lambda expression to it

Collections.*sort*(sigIpAddressList, **new** Comparator<String>() {

@Override

**public** **int** compare(String item1, String item2) {

**byte**[] ba1 = *getInetAddress*(item1).getAddress();

**byte**[] ba2 = *getInetAddress*(item2).getAddress();

//we have to compare each byte

**for**(**int** i = 0; i < ba1.length; i++) {

**int** b1 = *unsignedByteToInt*(ba1[i]);

**int** b2 = *unsignedByteToInt*(ba2[i]);

**if**(b1 == b2)

**continue**;

**if**(b1 < b2)

**return** -1;

**else**

**return** 1;

}

**return** 0;

}

});

@FunctionalInterface

**public** **interface** Greeting {

**public** **void** perform();

}

**List function in java 7:**

**public** **class** ListUsageLambda {

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

// **TODO** Auto-generated method stub

List<Person> list = Arrays.*asList*(

**new** Person("sachin","chippalkatti",23),

**new** Person("Anish","chippalkatti",213),

**new** Person("Anand","chippalkatti",93),

**new** Person("gun","chippalkatti",27),

**new** Person("raju","chippalkatti",123)

);

// sort the list by first name

Collections.*sort*(list, **new** Comparator<Person>() {

@Override

**public** **int** compare(Person o1, Person o2) {

// **TODO** Auto-generated method stub

**return** o1.getName().compareTo(o2.getName());

}

});

// Create method to print element of list

System.***out***.println("Print all the person in the list\n\n");

*printAll*(list);

// Create method to print last name starters with C

System.***out***.println("Printing last name beginning with c\n\n");

*printAllStartC*(list);

// Create method to print last name starters with dynamic value

System.***out***.println("Printing last name beginning with c dynamic way\n\n");

*printAllStartCondition*(list, **new** Condition() {

@Override

**public** **boolean** test(Person p) {

// **TODO** Auto-generated method stub

**return** p.getLastname().startsWith("c");

}

});

System.***out***.println("Printing name beginning with s dynamic way\n\n");

*printAllStartCondition*(list, **new** Condition() {

@Override

**public** **boolean** test(Person p) {

// **TODO** Auto-generated method stub

**return** p.getName().startsWith("s");

}

});

}

**private** **static** **void** printAllStartC(List<Person> list) {

// **TODO** Auto-generated method stub

**for**(Person p: list) {

**if**(p.getLastname().startsWith("c")) {

System.***out***.println(p);

}

}

}

**private** **static** **void** printAllStartCondition(List<Person> list, Condition c) {

// **TODO** Auto-generated method stub

**for**(Person p: list) {

**if**(c.test(p)) {

System.***out***.println(p);

}

}

}

**private** **static** **void** printAll(List<Person> list) {

// **TODO** Auto-generated method stub

**for**(Person p : list)

System.***out***.println(p);

}

}

**interface** Condition{

**boolean** test(Person p);

}

**List function in java 8:**

**public** **class** ListUsageLambdaJava8 {

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

// **TODO** Auto-generated method stub

List<Person> list = Arrays.*asList*(

**new** Person("sachin","chippalkatti",23),

**new** Person("Anish","chippalkatti",213),

**new** Person("Anand","chippalkatti",93),

**new** Person("gun","chippalkatti",27),

**new** Person("raju","chippalkatti",123)

);

// sort the list by first name

Collections.*sort*(list, (p1,p2) -> p1.getName().compareTo(p2.getName()));

// Create method to print element of list

System.***out***.println("Print all the person in the list\n\n");

*printAllStartCondition*(list, (p) ->**true**);

// Create method to print last name starters with C

System.***out***.println("Printing last name beginning with c\n\n");

*printAllStartC*(list);

// Create method to print last name starters with dynamic value

System.***out***.println("Printing last name beginning with c dynamic way\n\n");

*printAllStartCondition*(list, (p) -> p.getLastname().startsWith("c"));

System.***out***.println("Printing name beginning with s dynamic way\n\n");

*printAllStartCondition*(list, (p) -> p.getName().startsWith("s"));

}

**private** **static** **void** printAllStartC(List<Person> list) {

// **TODO** Auto-generated method stub

**for**(Person p: list) {

**if**(p.getLastname().startsWith("c")) {

System.***out***.println(p);

}

}

}

**private** **static** **void** printAllStartCondition(List<Person> list, Condition c) {

// **TODO** Auto-generated method stub

**for**(Person p: list) {

**if**(c.test(p)) {

System.***out***.println(p);

}

}

}

**private** **static** **void** printAll(List<Person> list) {

// **TODO** Auto-generated method stub

**for**(Person p : list)

System.***out***.println(p);

}

**interface** Condition{

**boolean** test(Person p);

}

}

Out of box interfaces from libraries as function type instead of interface

Predicate class is generic class.

Predicate<T>

Package java.util.function

Example:

System.***out***.println("Printing name beginning with s dynamic using predicate generic class\n\n");

*printAllStartPredicate*(list, (p) -> p.getName().startsWith("s"), (p) -> System.***out***.println(p.getName()));

**private** **static** **void** printAllStartPredicate(List<Person> list, Predicate<Person> c, Consumer<Person> consumer) {

// **TODO** Auto-generated method stub

**for**(Person p: list) {

**if**(c.test(p)) {

consumer.accept(p);

}

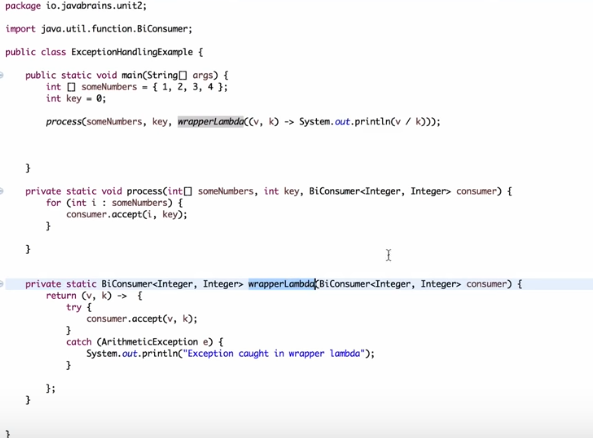
}

}

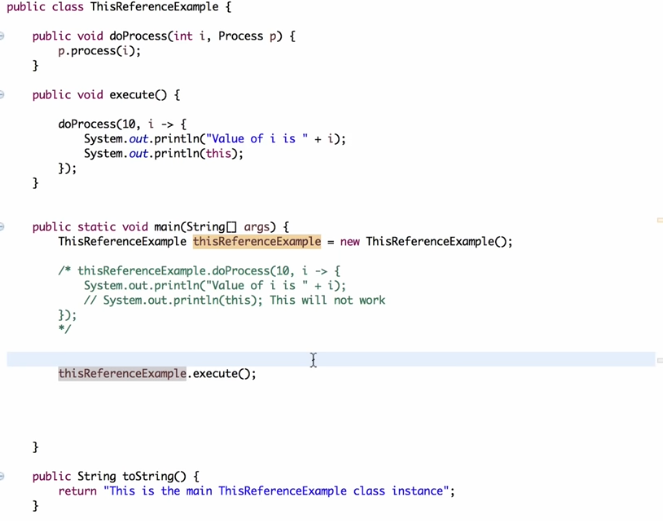
Functional interfaces provide target types for lambda expressions and method references.

<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

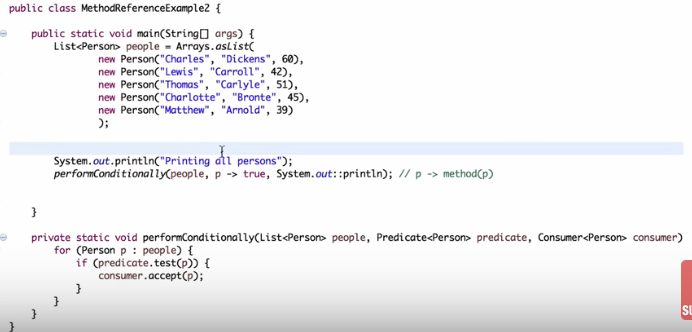
Exception Handling in Lambda

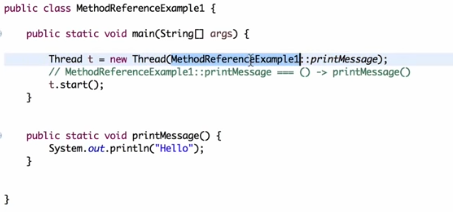


Lambda This reference:

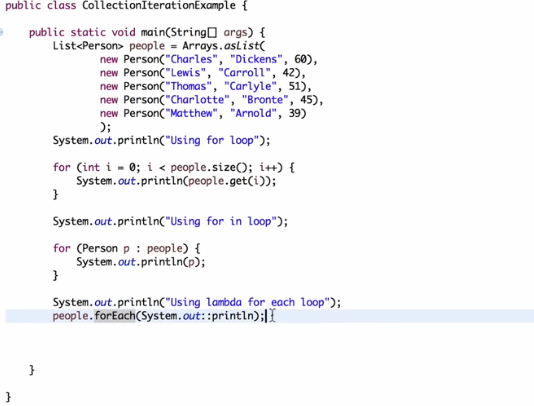


**Method References**





**Collection Iteration**



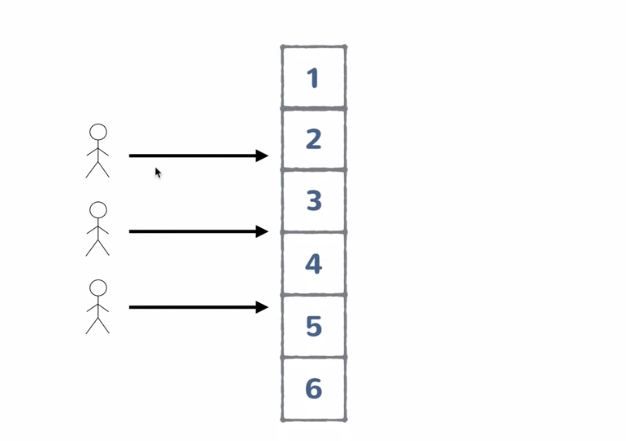
For and for -each is external iterator.

forEach is internal iterator.

**Introduction to stream**

Stream works like Assembly line. Action are done parallel. Hence support parallel execution.

stream provides an elegant lazy evaluation of an expression, and it also supports intermediate and terminal operations

****

System.***out***.println("Printing the list using forEach");

list.forEach(System.***out***::println);

System.***out***.println("Printing the list using streams");

list.stream()

.forEach(System.***out***::println);

System.***out***.println("Printing the list using streams and filtering with condition");

list.stream()

.filter(p-> p.getName().startsWith("s")) //Condition function interface having test method return Boolean value.

.forEach(System.***out***::println);

// Count the empty strings

List<String> strList = Arrays.asList("abc", "", "bcd", "", "defg", "jk");

**long** count = strList.stream()

.filter(x -> x.isEmpty())

.count();

System.out.printf("List %s has %d empty strings %n", strList, count);

// Count String with length more than 3

**long** num = strList.stream()

.filter(x -> x.length()> **3**)

.count();

System.out.printf("List %s has %d strings of length more than 3 %n",

strList, num);

// Count number of String which startswith "a"

count = strList.stream()

.filter(x -> x.startsWith("a"))

.count();

System.out.printf("List %s has %d strings which startsWith 'a' %n",

strList, count);

// Remove all empty Strings from List

List<String> filtered = strList.stream()

.filter(x -> !x.isEmpty())

.collect(Collectors.toList());

System.out.printf("Original List : %s, List without Empty Strings : %s %n",

strList, filtered);

// Create a List with String more than 2 characters

filtered = strList.stream()

.filter(x -> x.length()> **2**)

.collect(Collectors.toList());

System.out.printf("Original List : %s, filtered list : %s %n", strList, filtered);

// Convert String to Uppercase and join them using coma

List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany", "Italy",

"U.K.","Canada");

String G7Countries = G7.stream()

.map(x -> x.toUpperCase())

.collect(Collectors.joining(", "));

System.out.println(G7Countries);

// Create List of square of all distinct numbers

List<Integer> numbers = Arrays.asList(**9**, **10**, **3**, **4**, **7**, **3**, **4**);

List<Integer> distinct = numbers.stream()

.map( i -> i\*i).distinct()

.collect(Collectors.toList());

System.out.printf("Original List : %s, Square Without duplicates : %s %n",

numbers, distinct);

//Get count, min, max, sum, and average for numbers

List<Integer> primes = Arrays.asList(**2**, **3**, **5**, **7**, **11**, **13**, **17**, **19**, **23**, **29**);

IntSummaryStatistics stats = primes.stream()

.mapToInt((x) -> x)

.summaryStatistics();

System.out.println("Highest prime number in List : " + stats.getMax());

System.out.println("Lowest prime number in List : " + stats.getMin());

System.out.println("Sum of all prime numbers : " + stats.getSum());

System.out.println("Average of all prime numbers : " + stats.getAverage());

}

**Default Method:**

**define non-abstract methods in interfaces.**

Use **default** keyword to method

interface Poet {

default void write() {

System.out.println("Poet's default method");

}

}

interface Writer {

default void write() {

System.out.println("Writer's default method");

}

}

public class Multitalented implements Poet, Writer{

public static void main(String args[]){

Multitalented john = new Multitalented();

john.write(); // which write method to call, from Poet

// or, from Writer

}

}

Output:

Compile Time Error : class Multitalented inherits unrelated

defaults for write() from types Poet and Writer

## **Diamond Problem with Default Methods**

you need to override the write() method in your implementation class i.e. class Multitalented here, this will remove the ambiguity, making the compiler happy enough to compile this class.

public class Multitalented implements *Poet*, *Writer*{

@Override

public void write(){

System.out.println("Writing stories now days");

}

public static void main(String args[]){

Multitalented john = new Multitalented();

john.write(); // This will call Multitalented#write() method

}

}

Output:

Writing stories now days

# ConcurrentHashMap compute() and computeIfPresent()/ computeIfAbsent()

The compute() function tries to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). The entire function is performed **atomically**.

map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))

ConcurrentMap<**String**, **Integer**> map = **new** ConcurrentHashMap<>();

map.put("apple", 3);

map.put("mango", 4);

map.compute("apple", (key, value) -> value == **null** ? 1 : value + 1);

ConcurrentMap<**String**, LongAdder> map2 = **new** ConcurrentHashMap<>();

map2.computeIfAbsent("apple", key -> **new** LongAdder()).increment();

map2.computeIfAbsent("mango", key -> **new** LongAdder()).increment();

map2.computeIfAbsent("apple", key -> **new** LongAdder()).increment();

**//adds to map and increment**

# Map Reduce

# Essentially we map a set of values then we reduce it with a function such as average or sum into a single number.

peoples.stream().mapToInt(p-> p.getAge())

.average()

.getAsDouble();

**String.join()**

String banks = String.join("|", "Citibank", "Bank of America", "Chase");

System.out.println("banks: " + banks);

Output

banks: Citibank|Bank of America|Chase

List<String> payCompanies = Arrays.asList("Apple pay", "Samsung Pay", "Paypal");

String wallats = String.join(",", payCompanies);

System.out.println("electronic wallats : " + wallats);

Output

electronic wallets : Apple pay,Samsung Pay,Paypal

**Stream.flatMap()**

the flatMap() function is used to convert a Stream of Stream into a list of values.

public class Test {

public static void main(String args[]) {

List<String> teamIndia = Arrays.asList("Virat", "Dhoni", "Jadeja");

List<String> teamAustralia = Arrays.asList("Warner", "Watson", "Smith");

List<String> teamEngland = Arrays.asList("Alex", "Bell", "Broad");

List<String> teamNewZeland = Arrays.asList("Kane", "Nathan", "Vettori");

List<String> teamSouthAfrica = Arrays.asList("AB", "Amla", "Faf");

List<String> teamWestIndies = Arrays.asList("Sammy", "Gayle", "Narine");

List<String> teamSriLanka = Arrays.asList("Mahela", "Sanga", "Dilshan");

List<String> teamPakistan = Arrays.asList("Misbah", "Afridi", "Shehzad");

List<List<String>> playersInWorldCup2016 = new ArrayList<>();

playersInWorldCup2016.add(teamIndia);

playersInWorldCup2016.add(teamAustralia);

playersInWorldCup2016.add(teamEngland);

playersInWorldCup2016.add(teamNewZeland);

playersInWorldCup2016.add(teamSouthAfrica);

playersInWorldCup2016.add(teamWestIndies);

playersInWorldCup2016.add(teamSriLanka);

playersInWorldCup2016.add(teamPakistan);

// Let's print all players before Java 8

List<String> listOfAllPlayers = new ArrayList<>();

for(List<String> team : playersInWorldCup2016){

for(String name : team){

listOfAllPlayers.add(name);

}

}

System.out.println("Players playing in world cup 2016");

System.out.println(listOfAllPlayers);

// Now let's do this in Java 8 using FlatMap

List<String> flatMapList = playersInWorldCup2016

.stream()

.flatMap(pList -> pList.stream())

.collect(Collectors.toList());

System.out.println("List of all Players using Java 8");

System.out.println(flatMapList);

}

}

**.forEach()**

When you call forEach() on parallel stream, the [order of iteration](http://java67.blogspot.com/2014/05/3-examples-to-loop-map-in-java-foreach.html) is not guaranteed, but you can ensure that ordering by calling forEachOrdered() method.

here is two forEach() method in Java 8, one defined inside Iterable and other inside java.util.stream.Stream class.

public class Java8Demo {

public static void main(String args[]) {

List<String> alphabets = new ArrayList<>(Arrays.asList("aa", "bbb", "cac", "dog"));

// looping over all elements using Iterable.forEach() method

alphabets.forEach(s -> System.out.println(s));

// You can even replace lambda expression with method reference

// because we are passing the lambda parameter as it is to the

// method

alphabets.forEach(System.out::println);

// you can even do something with lambda parameter e.g. adding a comma

alphabets.forEach(s -> System.out.print(s + ","));

// There is one more forEach() method on Stream class, which operates

// on stream and allows you to use various stream methods e.g. filter()

// map() etc

alphabets.stream().forEach(System.out::println);

// let's now only print elmements which startswith "a"

alphabets.stream()

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

.forEach(System.out::println);

// let's filter out only which has length greater than 2

alphabets.stream()

.filter(s -> s.length() > 2)

.forEach(System.out::println);

// now, let's print length of each string using map()

alphabets.stream()

.mapToInt(s -> s.length())

.forEach(System.out::println);

// how about calculating sum of length of all string

alphabets.stream()

.mapToInt(s -> s.length())

.sum();

}

}

# Map Function

Stream API also provides methods like mapToDouble(), mapToInt(), and mapToLong() which returns DoubleStream, IntStream and LongStream, which are specialized stream for double, int and long data types.

List<Integer> squares = numbers.stream()

.map( i -> i\*i)

.collect(Collectors.toList());

**filter(**)

The **filter(**) method as its name suggests is used to perform filtering based upon some boolean conditions.

List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 12, 18);

Integer lcm = listOfNumbers.stream()

.filter(i -> i % 2 == 0)

.filter(i -> i % 3 == 0)

.findFirst().get();

**Peek()**

the peek() method returns a stream consisting of the elements of this stream and performs the action requested by the client

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

/\*\*

\* Java 8 peek() method example

\*/

public class Test {

public static void main(String[] args) {

List<String> versions = new ArrayList<>();

versions.add("Lollipop");

versions.add("KitKat");

versions.add("Jelly Bean");

versions.add("Ice Cream Sandwidch");

versions.add("Honeycomb");

versions.add("Gingerbread");

// filtering all vaersion which are longer than 7 characters

versions.stream()

.filter(s -> s.length() > 7)

.peek(e -> System.out.println("After the first filter: " + e))

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

.peek(e -> System.out.println("After the second filter: " + e))

.collect(Collectors.toSet());

}

}

After the first filter: Lollipop

After the first filter: Jelly Bean

After the first filter: Ice Cream Sandwich

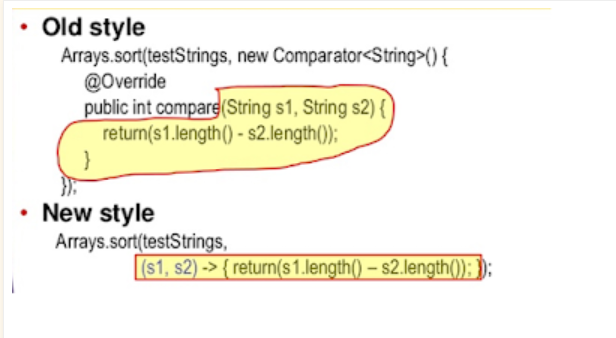
After the first filter: Honeycomb

After the second filter: Honeycomb

After the first filter: Gingerbread

**Comparator and Comparable classes in Java 8**

can use a lambda expression to implement Comparator and Comparable interface in Java, and not just these two interfaces but to implement any interface, which has only one abstract method because those are known as SAM (Single Abstract Method) Type and lambda expression in Java supports that.

****

**final** Comparator<TrainingCourses> PRICE\_COMPARATOR = **new** Comparator<TrainingCourses>() {

**@Override**

**public** **int** **compare**(TrainingCourses t1, TrainingCourses t2) {

**return** t1.price().compareTo(t2.price());

}

};

Collections.sort(onlineCourses, PRICE\_COMPARATOR);

Collections.sort(onlineCourses, (c1, c2) -> c2.price().compareTo(c1.price()));

**Following are FunctionalInterface:**

**@FunctionalInterfaces**

**public** **interface** **Comparator**<T> {

....

}

**@FunctionalInterface**

**public** **interface** **Runnable** {

.......

}

# sort HashMap by values in Java 8

The Map.Entry class, which is a [nested static class](https://javarevisited.blogspot.com/2012/12/inner-class-and-nested-static-class-in-java-difference.html#axzz5caMgsIIs) of java.util.Map interface is also not behind, it has got two additional methods **comparingByKey()** and **comparingByValue()** which can be used to sort a Map by key and values. They can be used along with sorted() method of Stream to [sort a HashMap by values](http://www.java67.com/2014/04/2-ways-to-sort-hashmap-in-java-example.html) in Java.

// now, let's collect the sorted entries in Map

Map<String, Integer> sortedByPrice = ItemToPrice.entrySet()

.stream()

.sorted(Map.Entry.<String, Integer>comparingByValue())

.forEach(System.out::println);

The Map returned by the previous statement was not sorted because ordering was lost while collecting result in Map you need to use the [LinkedHashMap](http://www.java67.com/2012/08/difference-between-hashmap-and-LinkedHashMap-Java.html) to preserve the order

Map result = map.entrySet()

.stream()

.sorted(Map.Entry.comparingByKey())

.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,

(oldValue, newValue) -> oldValue, LinkedHashMap::new));

Map<String, Integer> sortedByValueDesc = ItemToPrice.entrySet()

.stream()

.sorted(Map.Entry.<String, Integer>comparingByValue().reversed())

.collect(Collectors.toMap(Map.Entry::getKey,

Map.Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));

First of all, we **cannot** iterate a Map directly using [iterators](https://www.geeksforgeeks.org/iterators-in-java/), because Map are not [Collection.](https://www.geeksforgeeks.org/collections-in-java-2/) Also before going further, you must know a little-bit about [Map.Entry<K, V>](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html) interface.

Since all maps in Java implement [Map](https://www.geeksforgeeks.org/map-interface-java-examples/) interface, following techniques will work for any map implementation ([HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/), [TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java/), [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples/), [Hashtable](https://www.geeksforgeeks.org/java-util-hashtable-class-java/), etc.)

1. **Iterating over Map.entrySet() using For-Each loop :**

Map.entrySet() method returns a collection-view(Set<Map.Entry<K, V>>) of the mappings contained in this map. So we can iterate over key-value pair using getKey() and getValue() methods of [Map.Entry<K, V>](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html). This method is most common and should be used if you need both map keys and values in the loop. Below is the java program to demonstrate it.

// using for-each loop for iteration over Map.entrySet()

        for (Map.Entry<String,String> entry : gfg.entrySet())

            System.out.println("Key = " + entry.getKey() +

                             ", Value = " + entry.getValue());

**Converting List to Map**

Map<String, Course > **result** **=** listOfCourses

.stream()

.collect(

Collectors.toMap(Course**::getTitle**,

Function.identity()));

All it needs is one method to extract the key, which is Course::getTitle and one method to extract value which is Function.identity() i.e. the object itself

Remove the duplicate:

Map<String, Course > **result** **=** listOfCourses

.stream()

.collect(

Collectors.toMap(Course**::getTitle**,

Function.identity(),

(e1, e2) **-**> e2,

LinkedHashMap**::new**));

**StringJoiner**

The java.util.StringJoiner can be used to join any number of arbitrary String, a list of String, or an array of String in Java.

StringJoiner joiner = new StringJoiner(",");

joiner.add("Java");

joiner.add("C++");

joiner.add("Python");

joiner.add("Ruby");

String text = joiner.toString();

System.out.println("comma separated String: " + text);

Output

comma separated String: Java,C++,Python,Ruby

# [Difference between CompletableFuture, Future](https://stackoverflow.com/questions/35329845/difference-between-completablefuture-future-and-rxjavas-observable)

While using **Future**, we do not get notified when it is complete neither does it provides us a callable method which will automatically be called when the result is available but **CompletableFuture** provides us **with a** lot of callable methods which can be used as per our use case.