1. Lambada Expression:

interface Hello{

public void doCalculate(int a);

}

public class HelloWorld{

public static void main(String []args) throws InterruptedException{

System.out.println("Hello World");

HelloWorld helloWorld=new HelloWorld();

helloWorld.doCal();

System.out.println("From seperate thread : "+Thread.currentThread().getName());

for(int i=0;i<5;i++){

System.out.println("value of variable i is : "+i);

Thread.sleep(500);

helloWorld.multiThreadSample();

}

System.out.println("from main method thread : ");

helloWorld.multiThreadSample();

System.out.println("Finished main thred execution : ");

}

public int doCal(){

Hello h=(a)->

{

a=(a+5);

a=(a+5);

if(a==0) System.out.println("value of variable is zero : "+a);

else if(a>0) System.out.println("value of variable a is possitive : "+a);

else System.out.println("value of a is negative : "+a);

};

h.doCalculate(0);

return 0;

}

public void multiThreadSample() {

Runnable r=()->

{

try{

System.out.println("From seperate thread : "+Thread.currentThread().getName());

for(int i=0;i<10;i++){

Thread.sleep(100);

System.out.println("value of i is : "+i+", Thread name : "+Thread.currentThread().getName());

}

}

catch(InterruptedException e){

e.getMessage();

}

};

new Thread(r).start();

}

}

1. Lambada Expression and method reference:

import java.util.\*;

public class Example{

public static void main(String[] args) {

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

list.add("Rick");

list.add("Negan");

list.add("Daryl");

list.add("Glenn");

list.add("Carl");

//method reference

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

System.out.println("hi from method reference :::::");

// lambda expression

list.forEach(

(name)->

System.out.println(name)

);

}

}

1. Method reference and lambada expression for comprator and iterating over list for sorting number list

import java.util.\*;

public class Example{

public static void main(String[] args) {

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

list.add("Rick");

list.add("Negan");

list.add("Daryl");

list.add("Glenn");

list.add("Carl");

//method reference

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

System.out.println("hi from method reference :::::");

// lambda expression

list.forEach(

(name)->

System.out.println(name)

);

MyComparator comparator=new MyComparator();

List<Integer> numberList=Arrays.asList(2,5,7,3,1,9);

//method reference

// Collections.sort(numberList,comparator::compare);

//lambada expression

Collections.sort(

numberList,(first,second)->comparator.compare(first,second));

//method reference for iterating over list

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

}

private static class MyComparator{

public int compare(final Integer a,final Integer b){

return a.compareTo(b);

}

}

}

1. Custom object comparator

import java.util.\*;

public class Example{

public static void main(String[] args) {

Employee e1=new Employee("Ajit",1);

Employee e2=new Employee("Sameer",2);

Employee e3=new Employee("Balu",3);

List<Employee> eList=Arrays.asList(e1,e2,e3);

MyComparator comparator=new MyComparator();

Collections.sort(eList,comparator::compare);

/\* List<String> list=new ArrayList<String>();

list.add("Rick");

list.add("Negan");

list.add("Daryl");

list.add("Glenn");

list.add("Carl");

//method reference

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

System.out.println("hi from method reference :::::");

// lambda expression

list.forEach(

(name)->

System.out.println(name)

);

MyComparator comparator=new MyComparator();

List<Integer> numberList=Arrays.asList(2,5,7,3,1,9);

//method reference

// Collections.sort(numberList,comparator::compare);

//lambada expression

Collections.sort(

numberList,(first,second)->comparator.compare(first,second));

//method reference for iterating over list

numberList.forEach(System.out::println);\*/

}

private static class MyComparator<Employee>{

public int compare(Employee e1,Employee e2){

return e1.getName().compareTo(e2.getName());

}

}

}

class Employee{

private String name;

private int rollNumber;

Employee(String name,int rollNumber){

this.name=name;

this.rollNumber=rollNumber;

}

public void setName(String name){

this.name=name;

}

public String getName(){

return name;

}

public void setRollNumber(int rollNumber){

this.rollNumber=rollNumber;

}

public int gteRollNumber(){

return rollNumber;

}

}

1. Custom object sorting example:

import java.util.\*;

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int age;

private long salary;

public int getId(){

return id;

}

public String getName(){

return name;

}

public int getAge(){

return age;

}

public long getSalary(){

return salary;

}

public String toString(){

return id+","+name+","+age+","+salary;

}

// Many sort sequences can be created with different names.

public static Comparator<Employee> nameComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

};

public static Comparator<Employee> idComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return Integer.valueOf(e1.getId()).compareTo(Integer.valueOf(e2.getId()));

}

};

public Employee() { }

public Employee(int id, String name, int age, long salary){

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

// setters and getters.

// Only one sort sequence can be created with in the class.

@Override

public int compareTo(Employee e) {

//return Integer.valueOf(this.id).compareTo(Integer.valueOf(e.id));

//return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));

if (this.id > e.id) {

return 1;

}else if(this.id < e.id){

return -1;

}else {

return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));

}

}

public static void main(String[] args) {

Employee e1 = new Employee(5, "Yash", 22, 1000);

Employee e2 = new Employee(8, "Tharun", 24, 25000);

List<Employee> list = new ArrayList<Employee>();

list.add(e1);

list.add(e2);

Collections.sort(list); // call @compareTo(o1)

Collections.sort(list, Employee.nameComparator); // call @compare (o1,o2)

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

Collections.sort(list, Employee.idComparator); // call @compare (o1,o2)

}

}

// new comparator with sort lambada expression

import java.util.\*;

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int age;

private long salary;

public int getId(){

return id;

}

public String getName(){

return name;

}

public int getAge(){

return age;

}

public long getSalary(){

return salary;

}

public String toString(){

return id+","+name+","+age+","+salary;

}

// Many sort sequences can be created with different names.

public static Comparator<Employee> nameComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

};

public static Comparator<Employee> idComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return Integer.valueOf(e1.getId()).compareTo(Integer.valueOf(e2.getId()));

}

};

public static Comparator<Employee> salComparator=new Comparator<Employee>(){

@Override

public int compare(Employee e1,Employee e2){

return Integer.valueOf(Long.valueOf(e1.getSalary()).compareTo(Long.valueOf(e2.getSalary())));

}

};

public Employee() { }

public Employee(int id, String name, int age, long salary){

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

// setters and getters.

// Only one sort sequence can be created with in the class.

@Override

public int compareTo(Employee e) {

//return Integer.valueOf(this.id).compareTo(Integer.valueOf(e.id));

//return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));

if (this.id > e.id) {

return 1;

}else if(this.id < e.id){

return -1;

}else {

return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));

}

}

public static void main(String[] args) {

Employee e1 = new Employee(5, "Yash", 22, 1000);

Employee e2 = new Employee(8, "Tharun", 24, 25000);

List<Employee> list = new ArrayList<Employee>();

list.add(e1);

list.add(e2);

Collections.sort(list); // call @compareTo(o1)

Collections.sort(list, Employee.nameComparator); // call @compare (o1,o2)

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

System.out.println("-----------------------------");

list.forEach(list1->

System.out.println(list1)

);

Collections.sort(list, Employee.idComparator); // call @compare (o1,o2)

System.out.println("Sorting by using age:------------");

list.forEach(list2->

System.out.println(list2)

);

Collections.sort(list, Employee.salComparator); // call @compare (o1,o2)

System.out.println("Sorting by using salary:------------");

list.forEach(list2->

System.out.println(list2)

);

//using lambada expression for comparator

Comparator<Employee> salComp=

(Employee e3,Employee e4)->

{

return (e3.getId().compareTo(e4.getId()));

};

Collections.sort(list,salComp);

list.forEach(list3->System.out.println(list3));

}

}

//Final comparator with inner classes and lambada for comparator:

import java.util.\*;

public class Employee implements Comparable<Employee> {

private int id;

private String name;

private int age;

private long salary;

public int getId(){

return id;

}

public String getName(){

return name;

}

public int getAge(){

return age;

}

public long getSalary(){

return salary;

}

public String toString(){

return id+","+name+","+age+","+salary;

}

// Many sort sequences can be created with different names.

public static Comparator<Employee> nameComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

}

};

public static Comparator<Employee> idComparator = new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return Integer.valueOf(e1.getId()).compareTo(Integer.valueOf(e2.getId()));

}

};

public static Comparator<Employee> salComparator=new Comparator<Employee>(){

@Override

public int compare(Employee e1,Employee e2){

return Integer.valueOf(Long.valueOf(e1.getSalary()).compareTo(Long.valueOf(e2.getSalary())));

}

};

public Employee() { }

public Employee(int id, String name, int age, long salary){

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

// setters and getters.

// Only one sort sequence can be created with in the class.

@Override

public int compareTo(Employee e) {

//return Integer.valueOf(this.id).compareTo(Integer.valueOf(e.id));

//return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));

if (this.id > e.id) {

return 1;

}else if(this.id < e.id){

return -1;

}else {

return Character.toString(this.name.charAt(0)).compareToIgnoreCase(Character.toString(e.name.charAt(0)));

}

}

public static void main(String[] args) {

Employee e1 = new Employee(5, "Yash", 22, 1000);

Employee e2 = new Employee(1, "Tharun", 24, 25000);

List<Employee> list = new ArrayList<Employee>();

list.add(e1);

list.add(e2);

Collections.sort(list); // call @compareTo(o1)

Collections.sort(list, Employee.nameComparator); // call @compare (o1,o2)

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

System.out.println("-----------------------------");

list.forEach(list1->

System.out.println(list1)

);

Collections.sort(list, Employee.idComparator); // call @compare (o1,o2)

System.out.println("Sorting by using age:------------");

list.forEach(list2->

System.out.println(list2)

);

Collections.sort(list, Employee.salComparator); // call @compare (o1,o2)

System.out.println("Sorting by using salary:------------");

list.forEach(list2->

System.out.println(list2)

);

//using lambada expression for comparator

Comparator<Employee> salComp=

(Employee e3,Employee e4)->

{

return (Integer.valueOf(e3.getId()).compareTo(Integer.valueOf(e4.getId())));

};

Collections.sort(list,salComp);

list.forEach(list3->System.out.println(list3));

}

}

//Explanation for comparator and comparable:

1) Code Availabilty  
  
The first thing to note is that while using Comparable you have to implement it in your class i.e *you need to change your class.*Example  
  
  public class Book implements Comparable{  
   ...  
  }  
  
For this, code of that class should be availaible to you . If you dont have access to the code of that class (say class belongs to third party), then there is no choice but to use Comparator because *Comparator does not need to change the original class.*  
  
2) Single Versus Multiple Sorting Criteria  
  
If you have only single sorting criteria to sort your elements then you can use Comparable but if you have more than one sorting criterias then you have to go for Comparator *also*.  
  
3) Arays.sort() and Collection.sort()  
  
Using Comparable has a advantage over Comparator. If your class implements Comparable then Arrays.sort() and Collections.sort() can sort its instances automatically. You do not need to write *separate comparators* and pass them to *overloaded*sort() as shown [here](http://java-journal.blogspot.in/2011/01/comparable-and-comparator-part-3.html).   
  
4) As keys in SortedMap and SortedSet  
  
This is another advantage of Comparable over Comparator. Objects which implement Comparable interface can be used as keys in a SortedMap( like TreeMap) or as elements in a SortedSet  (like TreeSet). Otherwise you have to write separate Comparator and pass it in the constructor of TreeMap.  
  
5) More Number of classes Versus flexibilty  
  
Use of Comparable does not require creation of extra classes while use of Comparator requires writing of *separate comparators*i.e *more number of classes*.   
  
But this has a advantage also.You can add as many sorting criteria later as you want or modify the existing ones without changing the class whose instances you are sorting.  
  
 Thus comparators provides flexibilty while Comparable avoids extra classes.  
  
Note that you can also write Comparators as anonymous classes. In that case you can avoid separate comparators also.  
  
6) Interclass comparisions  
  
If you are going to compare *instances of same class* then you *should* use Comparable. Though we can also compare objects of different types while using Comparable as shown [here](http://java-journal.blogspot.in/2011/01/using-comparable-interface-to-compare.html) but we should avoid it.  
  
 If you are going to compare instances of different classes then you should use Comparator. But this was valid upto  pre Java 5, before the introduction of generics.  
  
With the introduction of generics syntax of Comparator has been changed from :  
  
public interface Comparator  to   public interface Comparator <T>  
  
and of compare() from:  
  
public int compare(Object o1, Object o2)  to  public int compare(T o1,T o2)  
  
As you are seeing in new syntax both o1 and o2 are of type T. If their types would be different as:  
  
compare(Integer o1, String o2)   
  
then this will give compile time error.  
  
Therefore if generic form of compare() is used then it compares objects of only same types.  
  
With non generic form of compare() you can still compare objects of different types.  
  
7) Natural Order  
  
If you are going to sort elements according to their natural order then you should use Comparable and for any other order different from natural order Comparator should be used.  
  
**Thus to answer above given questions:-**  
  
1) When to use Comparable and When to use Comparator?  
  
Ans:- When you are going to sort according to natural order, have single sort criteria and  have access to the class you would use Comparable.  
  
Otherwise,  
  
If you cannot change the class and have multiple sorting criteria use Comparator.  
  
  
2) Interviewer may give you some class and  some criteria  and ask you to write code to sort instances of that class based on that criteria.You have to decide there whether to use Comparable or to use Comparator. or even, not use any of them?  
  
Ans: For example,

If you are given a class named Employee and asked to sort it's instances by employees*id*, then you should go Comparable as it seems its natural order.

But if you have to sort them by its salary or Date of Joining () etc.(which does not seem to be natural order of employee) then you can go for either Comparable or Comparator. Both are legal. If you would use Comparator then you have to write extra classes as comparator.  
  
And if you have to sort employees on id and *also* on salary and Date of Joining, then use Comparable for *id* ad create comparators for salary and  Date of joining.  
  
Suppose you are given list of objects of type *Integer*. Then which interface you will use? None, as Integer already implements Comparable. Just pass its array (or List) to Arrays.sort() (or Collections.sort()).  
  
3) What is the difference between Comparable and Comparator?  
  
  
Check here for this: [Difference between Comparable and Comparator Interfaces.](http://java-journal.blogspot.in/2010/12/difference-between-comparable-and.html)  
  
 4) Which is preffered Comparable or Comparator?  
  
Comparable has advantage that it avoids the creation of more number of classes and also elements implementing Comparable can be used directly in utility functions like  Arrays.sort()  and Collections.sort().   
  
While Comparator is more flexible.It has advantage that it avoids the changing the class you are going to sort and more sorting criterias can be added later.   
  
So in my view one should go for Comparable first for that sorting criteria *which is not going to change in future* and for additional criterias we can use Comparator in addition also.  
  
  
5) Why do you need Comparable if  Comparator can also sort things?  
  
Because Comparable :-  
  
1) makes it easy to use the elements implementing it in some utility functions and classes like TreeSet and TreeMap.  
  
2) Avoids creation of new classes.

//lambada expression for custom object list for each:

Using can also use Java 8 stream API and do the same thing in one line.

If you want to print any specific property then use this syntax:

ArrayList<Room> rooms = new ArrayList<Room>();

rooms.forEach(room -> System.out.println(room.getName()));

if you want to print all the properties of Java object then use this:

ArrayList<Room> rooms = new ArrayList<Room>();

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