|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| Package is under java.lang.Comparable | Package is under java.util.Comparator |
| Comparable is used to compare the instance of the same class in Java | Comparator can be used to compare instances of same and different classes. |
| public int compareTo(Movies m){  return this.getName().compareTo(m.getName());  } | public int compare(Movies m1, Movies m2){  return m1.getName().compareTo(m2.getName());  } |
| Sorting logic must be in same class whose objects are being sorted. | Sorting logic is separate class. |
| Collections.sort(list);  Here objects will be sorted on the basis of compareTo() method. | Collections.sort(list, nameComparator)  Here objects will be sorted on the basis of compare method. |

**COMPARABLE EXAMPLE**

**import java.util.ArrayList;**

**import java.util.Collections;**

**import java.util.Comparator;**

**import java.util.Date;**

**import java.util.List;**

**class Employ implements Comparable {**

**String name;**

**Integer employeeId;**

**Integer salary;**

**Date joiningDate;**

**public Employ(String name, Integer employeeId, Integer salary, Date joiningDate) {**

**super();**

**this.name = name;**

**this.employeeId = employeeId;**

**this.salary = salary;**

**this.joiningDate = joiningDate;**

**}**

**@Override**

**public String toString() {**

**return "Employee [name=" + name + ", employeeId=" + employeeId + ", salary=" + salary + ", joiningDate="**

**+ joiningDate + "]";**

**}**

**@Override**

**public int compareTo(Object o) {**

**int c;**

**Employ e2 = (Employ)o;**

**c = name.compareTo(e2.name);**

**if(c==0)**

**c= salary.compareTo(e2.salary);**

**if(c==0)**

**c=joiningDate.compareTo(e2.joiningDate);**

**if(c==0)**

**c= employeeId.compareTo(e2.employeeId);**

**return c;**

**}**

**}**

**public class ComparableExample {**

**@SuppressWarnings("deprecation")**

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

**Employ emp1 = new Employ("ank",14, 2000, new Date(2016 - 1900, 11, 14));**

**Employ emp2 = new Employ("dav",15, 500, new Date(2016 - 1900, 11, 23));**

**Employ emp3 = new Employ("ank",10, 1000, new Date(2016 - 1900, 11, 22));**

**Employ emp4 = new Employ("sam",8, 9000, new Date(2016 - 1900, 11, 29));**

**Employ emp5 = new Employ("ank", 9, 1000, new Date(2016 - 1900, 11, 19));**

**List<Employ> l = new ArrayList<Employ>();**

**l.add(emp1);**

**l.add(emp2);**

**l.add(emp3);**

**l.add(emp4);**

**l.add(emp5);**

**Collections.sort(l);**

**l.forEach(la->{ System.out.println(la +"" );});**

**//Collections.sort(l, Collections.reverseOrder(new EmployeeComparator()));**

**Collections.reverse(l);;**

**System.out.println("REverse Order:");**

**l.forEach(la->{ System.out.println(la +"" );});**

**}**

**}**

OUTPUT:

Employee [name=ank, employeeId=9, salary=1000, joiningDate=Mon Dec 19 00:00:00 IST 2016]

Employee [name=ank, employeeId=10, salary=1000, joiningDate=Thu Dec 22 00:00:00 IST 2016]

Employee [name=ank, employeeId=14, salary=2000, joiningDate=Wed Dec 14 00:00:00 IST 2016]

Employee [name=dav, employeeId=15, salary=500, joiningDate=Fri Dec 23 00:00:00 IST 2016]

Employee [name=sam, employeeId=8, salary=9000, joiningDate=Thu Dec 29 00:00:00 IST 2016]

REverse Order:

Employee [name=sam, employeeId=8, salary=9000, joiningDate=Thu Dec 29 00:00:00 IST 2016]

Employee [name=dav, employeeId=15, salary=500, joiningDate=Fri Dec 23 00:00:00 IST 2016]

Employee [name=ank, employeeId=14, salary=2000, joiningDate=Wed Dec 14 00:00:00 IST 2016]

Employee [name=ank, employeeId=10, salary=1000, joiningDate=Thu Dec 22 00:00:00 IST 2016]

Employee [name=ank, employeeId=9, salary=1000, joiningDate=Mon Dec 19 00:00:00 IST 2016]

**COMPARTOR EXAMPLE**

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.Date;

**import** java.util.List;

**class** Employ {

String name;

Integer employeeId;

Integer salary;

Date joiningDate;

**public** Employ(String name, Integer employeeId, Integer salary, Date joiningDate) {

**super**();

**this**.name = name;

**this**.employeeId = employeeId;

**this**.salary = salary;

**this**.joiningDate = joiningDate;

}

@Override

**public** String toString() {

**return** "Employee [name=" + name + ", employeeId=" + employeeId + ", salary=" + salary + ", joiningDate="

+ joiningDate + "]";

}

}

**class** EmployComparator **implements** Comparator<Employ> {

**public** **int** compare(Employ e1, Employ e2) {

**int** c;

c = e1.name.compareTo(e2.name);

**if**(c==0)

c= e1.salary.compareTo(e2.salary);

**if**(c==0)

c=e1.joiningDate.compareTo(e2.joiningDate);

**if**(c==0)

c= e1.employeeId.compareTo(e2.employeeId);

**return** c;

}

}

**public** **class** ComparatorExample3 {

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

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

Employ emp1 = **new** Employ("ank",14, 2000, **new** ~~Date~~(2016 - 1900, 11, 14));

Employ emp2 = **new** Employ("dav",15, 500, **new** ~~Date~~(2016 - 1900, 11, 23));

Employ emp3 = **new** Employ("ank",10, 1000, **new** ~~Date~~(2016 - 1900, 11, 22));

Employ emp4 = **new** Employ("sam",8, 9000, **new** ~~Date~~(2016 - 1900, 11, 29));

Employ emp5 = **new** Employ("ank", 9, 1000, **new** ~~Date~~(2016 - 1900, 11, 19));

List<Employ> l = **new** ArrayList<Employ>();

l.add(emp1);

l.add(emp2);

l.add(emp3);

l.add(emp4);

l.add(emp5);

Collections.*sort*(l, **new** EmployComparator());

l.forEach(la->{ System.***out***.println(la +"" );});

//Collections.sort(l, Collections.reverseOrder(new EmployeeComparator()));

Collections.*sort*(l, **new** EmployComparator().reversed());

System.***out***.println("REverse Order:");

l.forEach(la->{ System.***out***.println(la +"" );});

}

}

OUTPUT:

Employee [name=ank, employeeId=9, salary=1000, joiningDate=Mon Dec 19 00:00:00 IST 2016]

Employee [name=ank, employeeId=10, salary=1000, joiningDate=Thu Dec 22 00:00:00 IST 2016]

Employee [name=ank, employeeId=14, salary=2000, joiningDate=Wed Dec 14 00:00:00 IST 2016]

Employee [name=dav, employeeId=15, salary=500, joiningDate=Fri Dec 23 00:00:00 IST 2016]

Employee [name=sam, employeeId=8, salary=9000, joiningDate=Thu Dec 29 00:00:00 IST 2016]

REverse Order:

Employee [name=sam, employeeId=8, salary=9000, joiningDate=Thu Dec 29 00:00:00 IST 2016]

Employee [name=dav, employeeId=15, salary=500, joiningDate=Fri Dec 23 00:00:00 IST 2016]

Employee [name=ank, employeeId=14, salary=2000, joiningDate=Wed Dec 14 00:00:00 IST 2016]

Employee [name=ank, employeeId=10, salary=1000, joiningDate=Thu Dec 22 00:00:00 IST 2016]

Employee [name=ank, employeeId=9, salary=1000, joiningDate=Mon Dec 19 00:00:00 IST 2016]

**Cloning**

Cloning is process of creating an exact copy of an existing object in the memory.

Clone () method of java.lang.Object class is used for cloning process.

This method creates exact copy of an object which is called through field-by-field assignment and return the reference of that object.

|  |  |
| --- | --- |
| **Shallow Cloning** | **Deep Cloning** |
| Cloned Object and Original Object is not 100% dis joint. | Cloned Object and original Object is 100% disjoint. |
| Default version of clone method creates the shallow copy of an object. | To create a deep copy of an object, you have to override clone method. |
| Shallow copy is preferred if an object has only primitive fields. | Deep copy is preferred if an object has reference to other objects as fields. |
| Shallow Copy is fast and less expensive | Deep Copy is slow and very expensive. |
| The shallow copy of an object will have exact copy of all the fields of original object.  If the original objects has any reference to other objects as fields, then only reference of those objects are copied into clone object, copy of those objects are not created. | Deep copy of an object will have an exact copy of all the fields of original object like shallow copy.  But in additional, if original object has any reference to other objects as fields, then copy of those objects are also created by calling clone () method on this. |
| public Object clone() throws CloneNotSupportedException {  return (Course)super.clone();  } | public Object clone() throws CloneNotSupportedException{  Student student = (Student)super.clone();  student.course = (Course)course.clone();  return student;  } |

**Shallow Cloning Example**

**public** **class** Student **implements** Cloneable {

**int** id;

String name;

Course course;

**public** Student(**int** id, String name, Course course) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.course = course;

}

**public** Object clone() **throws** CloneNotSupportedException{

**return** (Student)**super**.clone();

}

}

**public** **class** Course {

String subject;

**public** Course(String subject) {

**this**.subject = subject;

}

}

**public** **class** ShallowCloning {

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

Course science = **new** Course("Science");

Student student1 = **new** Student(1,"Guru",science);

Student student2 = **null**;

**try** {

student2 = (Student)student1.clone();

}**catch**(CloneNotSupportedException e) {

}

student2.name ="Gurunathan";

System.***out***.println(student1.name);

System.***out***.println(student2.name);

System.***out***.println(student1.course.subject);//Science

student2.course.subject = "MATHS";

System.***out***.println(student1.course.subject);//MATHS

System.***out***.println(student2.course.subject);//MATHS

}

}

**Deep Cloning Example**

**public** **class** Student **implements** Cloneable {

**int** id;

String name;

Course course;

**public** Student(**int** id, String name, Course course) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.course = course;

}

**public** Object clone() **throws** CloneNotSupportedException{

Student student = (Student)**super**.clone();

student.course = (Course)course.clone();

**return** student;

}

}

**public** **class** Course **implements** Cloneable{

String subject;

**public** Course(String subject) {

**this**.subject = subject;

}

**public** Object clone() **throws** CloneNotSupportedException{

**return** (Course)**super**.clone();

}

}

**public** **class** DeepCloning {

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

Course science = **new** Course("Science");

Student student1 = **new** Student(1,"Guru",science);

Student student2 = **null**;

**try** {

student2 = (Student)student1.clone();

}**catch**(CloneNotSupportedException e) {

}

System.***out***.println(student1.course.subject);//Science

student2.course.subject = "MATHS";

System.***out***.println(student1.course.subject);//Science

System.***out***.println(student2.course.subject);//MATHS

}

}

**Class Loader**

Class Loader loads the class for execution.

Class file is input to the class loader sub system.

The class loader reads the .class file and save the byte code in the method area.

Class Loader is responsible for finding and loading classes at run time.

Class Loader load the class based on the following class loader orders.

First, **Boot Strap Loader** – It loads those classes which are essential for JVM to function properly.

Example – jre/lib/rt.jar.

Then **Extension class loader** is child of boot strap class loader.

It loads thee jar file located in jre/lib/ext directory.

Finally, **system/application class loader** is child of extension class loader.

It loads the class from class path.

When we run Java program, it is loaded first by the class loader.

**JDK – Java Development Kit**

Java Development Kit is a software development kit contains JRE and development tools such as compiler, Debugger.

JDK is an environment to develop and run the Java application.

JDK is for development purpose whereas JRE is for running the Java Program.

JDK and JRE both contains JVM so that we can run our Java program.

**JRE – Java Runtime Environment**

Java Runtime Environment is an environment within which the Java Virtual Machine runs.

You can run the code in JRE, but you can’t develop and compile the code in JRE.

JRE contains JVM and class libraries(rt.jar) and compiled class files.

**JVM – Java Virtual Machine**

Java Virtual Machine is the virtual machine that runs or executes the byte code produced by the compiler.

JVM translates the byte code into machine language.

When you run the Java program, Java compiler first compiles your java code to byte code. Then, the JVM translates the byte code into native machine code.

Internally JVM translates byte code into native machine code based on the operating system.