Java General Questions – 2017

# [**How to create singleton java class for multiple jvm support?**](https://stackoverflow.com/questions/12032895/how-to-create-singleton-java-class-for-multiple-jvm-support)

In java there is no direct way to achieve it, but there is a work around.

You can use in following manners.

1. Serialize the object by refining writeReplace() and readResolve() method and store in a shared drive so that all the applications running in different jvms have access to shared drive and path to deserialize the object.
2. Store the object directly in the database in blob format and deserialize it.
3. Serialize the object and store the file bytes.
4. Store the object in a Cache like Riak and let other application access to it.
5. Using shared library inside jdk just like **Runtime.getRuntime** .

How to store object in database, a basic and dummy example is given below.

/\*

\* mysql> CREATE TABLE java\_objects (

\* id INT AUTO\_INCREMENT,

\* name varchar(128),

\* object\_value BLOB,

\* primary key (id));

\*\*/

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.ResultSet;

**import** java.util.ArrayList;

**import** java.util.Date;

**import** java.util.List;

**public** **class** SerializeJavaObjects\_MySQL {

**static** **final** String WRITE\_OBJECT\_SQL = "INSERT INTO java\_objects(name, object\_value) VALUES (?, ?)";

**static** **final** String READ\_OBJECT\_SQL = "SELECT object\_value FROM java\_objects WHERE id = ?";

**public** **static** Connection getConnection() **throws** Exception {

String driver = "org.gjt.mm.mysql.Driver";

String url = "jdbc:mysql://localhost/databaseName";

String username = "root";

String password = "root";

Class.forName(driver);

Connection conn = DriverManager.getConnection(url, username, password);

**return** conn;

}

**public** **static** **long** **writeJavaObject**(Connection conn, Object object) **throws** Exception {

String className = object.getClass().getName();

PreparedStatement pstmt = conn.prepareStatement(WRITE\_OBJECT\_SQL);

// set input parameters

pstmt.setString(1, className);

pstmt.setObject(2, object);

pstmt.executeUpdate();

// get the generated key for the id

ResultSet rs = pstmt.getGeneratedKeys();

**int** id = -1;

**if** (rs.next()) {

id = rs.getInt(1);

}

rs.close();

pstmt.close();

System.out.println("writeJavaObject: done serializing: " + className);

**return** id;

}

**public** **static** Object **readJavaObject**(Connection conn, **long** id) **throws** Exception {

PreparedStatement pstmt = conn.prepareStatement(READ\_OBJECT\_SQL);

pstmt.setLong(1, id);

ResultSet rs = pstmt.executeQuery();

rs.next();

Object object = rs.getObject(1);

String className = object.getClass().getName();

rs.close();

pstmt.close();

System.out.println("readJavaObject: done de-serializing: " + className);

**return** object;

}

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

Connection conn = null;

**try** {

conn = getConnection();

System.out.println("conn=" + conn);

conn.setAutoCommit(false);

List<Object> list = **new** ArrayList<Object>();

list.add("This is a short string.");

list.add(**new** **Integer**(1234));

list.add(**new** Date());

**long** objectID = writeJavaObject(conn, list);

conn.commit();

System.out.println("Serialized objectID => " + objectID);

List listFromDatabase = (List) readJavaObject(conn, objectID);

System.out.println("[After De-Serialization] list=" + listFromDatabase);

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

conn.close();

}

}

}

# **Java Basics**

## **String, String intern, Integer == and equals**

**package** com.ddlab.rnd.java.basics;  
**public class** TestInteger {  
 **public static void** main(String[] args) {  
 Integer i1 = **new** Integer(10);  
 Integer i2 = **new** Integer(10);  
 System.***out***.println(**"(i1 == i2)-->"**+(i1 == i2));*//****false***System.***out***.println(**"(i1.equals(i2)-->"**+(i1.equals(i2)));*//****true***String s1 = **"abc"**;  
 String s2 = **"abc"**;  
 String s3 = **new** String(**"abc"**);  
 System.***out***.println(**"(s1 == s2)"**+(s1 == s2));*//****true***System.***out***.println(**"(s1 == s3)"**+(s1 == s3));*//****false***String s4 = s3.intern();  
 System.***out***.println(**"(s1 == s4)"**+(s1 == s4));*//****true , after String interning***}  
}

# Java Method Overloading Ambiguity

**public class** TestAmbiguity {  
  
 **public static void** m1(Object obj) {  
 System.***out***.println(**"Object ->"**+obj);  
 }  
  
 **public static void** m1(Double[] doubles) {  
 System.***out***.println(**"Double ->"**+doubles);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Here output is

**Double ->null**

The reason is the **array version is more specific than the Object-version**.

To create ambiguity, let us modify the class.

**public class** TestAmbiguity {  
  
 **public static void** m1(Object obj) {  
 System.***out***.println(**"Object ->"**+obj);  
 }  
  
 **public static void** m1(Double[] doubles) {  
 System.***out***.println(**"Double ->"**+doubles);  
 }  
  
 **public static void** m1(Double d) {  
 System.***out***.println(**"Double ->"**+d);  
 }  
  
 **public static void** main(String[] args) {  
 m1(**null**);  
 }  
}

It will give compilation issue because of method ambiguous.

The following code is perfect.

**public class** TestAmbiguity {  
  
 **public static void** m1(Object obj) {  
 System.***out***.println(**"Object ->"**+obj);  
 }  
  
 **public static void** m1(Double d) {  
 System.***out***.println(**"Double ->"**+d);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Output is **Double ->null**

The following code is perfect.

**public class** TestAmbiguity {  
  
 **public static void** m1(Object o) {  
 System.***out***.println(**"Object Verion"**);  
 }  
  
 **public static void** m1(String s) {  
 System.***out***.println(**"String Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Output is **String Version**.

The following code will give compilation issue because of ambiguity.

**public class** TestAmbiguity {  
  
 **public static void** m1(StringBuffer sb) {  
 System.***out***.println(**"StringBuffer** **Verion"**);  
 }  
  
 **public static void** m1(String s) {  
 System.***out***.println(**"String Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 m1(**null**);  
 }  
}

//Compilation issue, because of ambiguity

The following code will give more clarity.

**class** Parent {  
  
}  
**class** Child **extends** Parent {  
  
}  
**public class** TestAmbiguity {  
  
 **public static void** m1(Parent p) {  
 System.***out***.println(**"Parent Verion"**);  
 }  
  
 **public static void** m1(Child c) {  
 System.***out***.println(**"Child Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Output is **Child Version**.

Similarly, the following is of same type.

**public class** TestAmbiguity {  
  
 **public static void** m1(Object p) {  
 System.***out***.println(**"Parent Verion"**);  
 }  
  
 **public static void** m1(List c) {  
 System.***out***.println(**"List Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Output is **List Version**.

The main reason, in case of null and in case of overloaded method, **java chooses the object which is smaller that parent in the object hierarchy**.

Here Child < Parent, List < Object

It means which extends.

The following methods create ambiguity as we pass null.

**public static void** m1(Map m) {  
 System.***out***.println(**"Parent Verion"**);  
}  
  
**public static void** m1(List c) {  
 System.***out***.println(**"List Version"**);  
}

Another interesting case for interface and abstract class is given below for overloaded method.

**public class** TestAmbiguity {  
  
 **public static void** m1(AbstractList m) {  
 System.***out***.println(**"AbstractList Verion"**);  
 }  
  
 **public static void** m1(List c) {  
 System.***out***.println(**"List Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Output is **AbstractList Verion**.

**In case of both interface and implementation version, if null is passed, the expected output will be implementation version**.

**interface** I {  
  
}  
  
**class** Impl **implements** I {  
  
}  
**public class** TestAmbiguity {  
  
 **public static void** m1(I i) {  
 System.***out***.println(**"Interface Verion"**);  
 }  
  
 **public static void** m1(Impl impl) {  
 System.***out***.println(**"Implementation Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Output **Implementation Version**.

Rules

1. In case of method hierarchy, both interface and implementation class, **implementation class will be selected if null is passed**. It means Impl implements Interface
2. In case of Parent and Child, if null is passed, **child will be selected since Child extends Parent**.
3. In case of ambiguous methods along with array type, **array will be choosen** as it is a special case.
4. For all other case, always consider and select the object which is smaller.

**public class** TestAmbiguity {  
  
 **public static void** m1(List list) {  
 System.***out***.println(**"List Interface Verion"**);  
 }  
  
 **public static void** m1(AbstractList absList) {  
 System.***out***.println(**"AbstractList Version"**);  
 }  
  
 **public static void** m1(ArrayList al) {  
 System.***out***.println(**"ArrayList Version"**);  
 }  
  
 **public static void** main(String[] args) {  
 *m1*(**null**);  
 }  
}

Here output is **ArrayList Version**.

# **Method Overloading**

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. It is also applicable in case of Parent and Child class .

**class** parent {  
  
 **void** m1( **int** x , **int** y) {  
  
 }  
}  
  
**class** Child **extends** Parent {  
  
 **void** m1( **int** x , **float** y ) **throws** IOException {  
  
 }  
}

The above is not ~~method overriding~~, it is **method overloading**. But you will get compilation issue in the following case.

Parent p = **new** Child();  
p.m1(2, 3.5f);//**It will give compilation issue**

We know that static methods cannot be overridden. But the following gives compilation issue.

**class** Parent {  
 **public static void** m1( **int** x , **int** y) {  
  
 }  
}  
**class** Child **extends** Parent {  
 **public static void** m1( **int** x , **int** y ) **throws** IOException {  
  
 }  
}

No idea, why ?

# **Checked Exception**

The following will give compilation issue.

**public static void** m2() {  
 **try** {  
  
 }  
 **catch** (IOException ie) { //**Compilation Issue**  
  
 }  
}

**The reason above code for compilation issue is that the code inside try body does not generate IOException**.

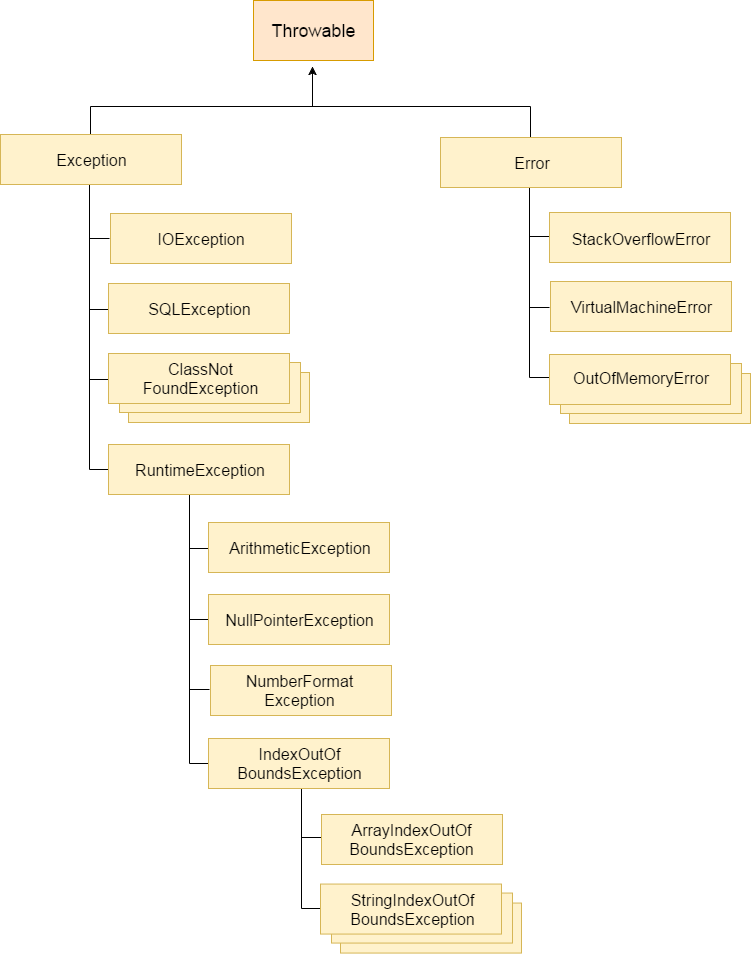
But the following is perfectly right.

**public static void** m1() **throws** IOException {  
  
}

The exception you created is a checked exception and must be thrown from somewhere to catch it. Any exception created by a java developer by extending Exception class is a checked exception. And the rules applicable for checked exception will be applied on such exceptions.

Another form of exception is called Unchecked Exception and usually created by extending RuntimeException Class. A developer is free to catch such exception without an explicit need for throwing it somewhere from your code.

**Exception Hierarchy** is given below.



Comparable, Comparator, Equality

# **Example on Comparable**

**package** com.ddlab.rnd.java.basics;  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.List;  
**class** Employee **implements** Comparable<Employee> {  
 **private int age**;  
 **private int sal**;  
 **public** Employee(**int** age , **int** sal) {  
 **this**.**age** = age;  
 **this**.**sal** = sal;  
 }  
  
 @Override  
 **public int** compareTo(Employee emp) {  
 ***//Ascending order*****if**(**age** == emp.**age**)  
 **return** 0;  
*// else if(age > emp.age)  
// return 1;  
// else  
// return -1;* ***//Descending order*****else if**(**age** > emp.**age**)  
 **return** -1;  
 **else  
 return** 1;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Employee{"** +  
 **"age="** + **age** +  
 **", sal="** + **sal** +  
 **'}'**;  
 }  
}  
**public class** ComparableComparator {  
 **public static void** main(String[] args) {  
 List<Employee> empList = **new** ArrayList<>();  
 empList.add( **new** Employee(23,5000));  
 empList.add( **new** Employee(24,6000));  
 empList.add( **new** Employee(25,7000));  
 System.***out***.println(empList);  
  
 Collections.*sort*(empList);  
 System.***out***.println(empList);  
 }  
}

Output

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}]

[Employee{age=25, sal=7000}, Employee{age=24, sal=6000}, Employee{age=23, sal=5000}] – Descending

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}] – Ascending

# **Example on Comparator**

**public class SalaryComparator** **implements** Comparator<Employee> {  
  
 @Override  
 **public int** compare(Employee e1, Employee e2) {  
 *//Ascending order  
// return e1.getAge() - e2.getAge();  
  
 //Descending order* **return** e2.getAge() - e1.getAge();  
 }  
}

**class Employee** {  
 **private int age**;  
 **private int sal**;  
  
 **public** Employee(**int** age , **int** sal) {  
 **this**.**age** = age;  
 **this**.**sal** = sal;  
 }  
  
 **public int** getAge() {  
 **return age**;  
 }  
  
 **public int** getSal() {  
 **return sal**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Employee{"** +  
 **"age="** + **age** +  
 **", sal="** + **sal** +  
 **'}'**;  
 }  
}  
**public class** ComparableComparator {  
 **public static void** main(String[] args) {  
 List<Employee> empList = **new** ArrayList<>();  
 empList.add( **new** Employee(23,5000));  
 empList.add( **new** Employee(24,6000));  
 empList.add( **new** Employee(25,7000));  
 System.***out***.println(empList);  
  
 Collections.*sort*(empList, **new** SalaryComparator());  
 System.***out***.println(empList);  
  
 }  
}

Output

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}]

[Employee{age=25, sal=7000}, Employee{age=24, sal=6000}, Employee{age=23, sal=5000}] – Descending

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}] – Ascending

**What will happen if the same class implements both Comparable and Comparator ?**

**package** com.ddlab.rnd.java.basics;  
  
**import** java.util.ArrayList;  
**import** java.util.Collections;  
**import** java.util.Comparator;  
**import** java.util.List;  
  
**class** Employee **implements** Comparable<Employee> , Comparator<Employee> {  
 **private int age**;  
 **private int sal**;  
  
 **public** Employee() {}  
  
 **public** Employee(**int** age , **int** sal) {  
 **this**.**age** = age;  
 **this**.**sal** = sal;  
 }  
  
 **public int** getAge() {  
 **return age**;  
 }  
  
 **public int** getSal() {  
 **return sal**;  
 }  
  
 @Override  
 **public int** compareTo(Employee emp) {  
 System.***out***.println(**"Comparable called ..."**);  
 *//Ascending order* **if**(**age** == emp.**age**)  
 **return** 0;  
 **else if**(**age** > emp.**age**)  
 **return** 1;  
 **else  
 return** -1;  
  
 *//Descending order  
// else if(age > emp.age)  
// return -1;  
// else  
// return 1;* }

@Override  
 **public int** compare(Employee e1, Employee e2) {  
 System.***out***.println(**"Comparator called ..."**);  
 *//Ascending order  
// return e1.getAge() - e2.getAge();  
  
 //Descending order* **return** e2.getAge() - e1.getAge();  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Employee{"** +  
 **"age="** + **age** +  
 **", sal="** + **sal** +  
 **'}'**;  
 }  
}  
**public class** ComparableComparator {  
 **public static void** main(String[] args) {  
 List<Employee> empList = **new** ArrayList<>();  
 empList.add( **new** Employee(23,5000));  
 empList.add( **new** Employee(24,6000));  
 empList.add( **new** Employee(25,7000));  
 System.***out***.println(empList);  
  
 Collections.*sort*(empList);  
 System.***out***.println(empList);  
  
 }  
}

The output is given below.

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}]

Comparable called ...

Comparable called ...

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}]

If we call like this.

**Collections.*sort*(empList , new Employee());  
System.*out*.println(empList);**

The output will be like this.

[Employee{age=23, sal=5000}, Employee{age=24, sal=6000}, Employee{age=25, sal=7000}]

Comparator called ...

Comparator called ...

[Employee{age=25, sal=7000}, Employee{age=24, sal=6000}, Employee{age=23, sal=5000}]

**It means a class can implement both Comparable and Comparator interface, default Collections.sort(list) will use Comparable and Collections.sort(list, new Employee()) will use Comparator interface**. That is why we have provided both parameterized constructor and a default constructor. The easiest way to implement comparable and comparator is use return (this.age – e.age) or return e1.getAge() – e2.getAge(). If both the objects are equal, it should be 0.

# **How to sort a List of Employees based upon two fields** ?

The problem statement is a list contains employee name, age and salary, I want to find out how many employees are senior but they are getting less salary.

## **Employee.java**

public class Employee implements Comparable<Employee> {

private int age;

private String name;

private int sal;

public Employee(int age, String name, int sal) {

this.age = age;

this.name = name;

this.sal = sal;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getSal() {

return sal;

}

public void setSal(int sal) {

this.sal = sal;

}

@Override

public String toString() {

return "Employee{" + "age=" + age + ", name=" + name + ", sal=" + sal + '}';

}

@Override

public int compareTo(Employee o) {

return this.age - ((Employee)o).getAge();

}

}

## **MyComparator.java**

package com.ddlab.rnd.core;

import java.util.Comparator;

public class MyComparator implements Comparator<Employee>{

@Override

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

**Employee e1 = (Employee) o1;**

**Employee e2 = (Employee) o2;**

**if( e2.getAge() > e1.getAge() && e2.getSal() < e1.getSal())**

**return 1;**

**else**

**return -1;**

**}**

}

## **TestMyComparator1.java**

package com.ddlab.rnd.core;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class TestMyComparator1 {

public static void main(String[] args) {

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

Employee e1 = new Employee(23, "John", 9000);

Employee e2 = new Employee(28, "Vidya", 2000);

Employee e3 = new Employee(27, "Ram", 5000);

Employee e4 = new Employee(33, "Hari", 1000);

Employee e5 = new Employee(25, "Shyam", 11000);

empList.add(e1);

empList.add(e2);

empList.add(e3);

empList.add(e4);

empList.add(e5);

System.out.println("EmpList--->"+empList);

// [Employee{age=23, name=John, sal=9000}, Employee{age=28, name=Vidya, sal=2000}, Employee{age=27, name=Ram, sal=5000}, Employee{age=33, name=Hari, sal=1000}, Employee{age=25, name=Shyam, sal=11000}]

Collections.sort(empList, new MyComparator());

System.out.println("EmpList--->"+empList);

//[Employee{age=33, name=Hari, sal=1000}, Employee{age=28, name=Vidya, sal=2000}, Employee{age=27, name=Ram, sal=5000}, Employee{age=25, name=Shyam, sal=11000}, Employee{age=23, name=John, sal=9000}]

}

}

# What will happen if you override equals() method without passing Object type

class Subject {

private String name;

public Subject(String name) {

this.name = name;

}

@Override

public int hashCode() {

int hash = 5;

hash = 47 \* hash + Objects.hashCode(this.name);

return hash;

}

public boolean equals(Subject obj) {

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

if (this == obj) {

return true;

}

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final Subject other = (Subject) obj;

if (!Objects.equals(this.name, other.name)) {

return false;

}

return true;

}

}

public class OverLoadingEquals {

public static void main(String[] args) {

Subject s1 = new Subject("Physics");

Subject s2 = new Subject("Physics");

System.out.println("s1.equals(s2)"+s1.equals(s2));//true

Set<Subject> set = new HashSet<Subject>();

set.add(s1);

System.out.println("set.contains(s2)---->"+set.contains(s2)); //false

//https://coderanch.com/t/378204/java/Overloading-overriding-equals-method

}

}

HashSet call equals(Object type) so that is why we get wrong result even if we have perfectly overriding the equals method. In case of Comparator and Comparable, it takes generic type, so you can pass actual object not the **Object** type.

The following will give compilation issue.

**public class Emp implements Comparable<Emp> {**

**@Override**

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

**return this.age - o.getAge();**

**}**

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

**return this.age - ((Emp) o).getAge();**

**}**

**}**

It will give the compilation issue as methods have same erasure …

The following example gives an interesting result.

public class Emp implements Comparable {

private int age;

private String name;

public Emp(int age, String name) {

this.age = age;

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

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

**System.out.println("---- Emp Called -----");**

**return this.age - o.getAge();**

**}**

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

**System.out.println("---- Object Called -----");**

**return this.age - ((Emp) o).getAge();**

**}**

public static void main(String[] args) {

List<Emp> empList = new ArrayList<>();

for (int j = 23; j < 33; j++) {

empList.add( new Emp( j, "Name-"+j ));

}

System.out.println("empList--->"+empList);

Collections.sort(empList);

System.out.println("empList--->"+empList);

}

}

Here the output is given below.

empList--->[com.ddlab.rnd.core1.Emp@15db9742, com.ddlab.rnd.core1.Emp@6d06d69c, com.ddlab.rnd.core1.Emp@7852e922, com.ddlab.rnd.core1.Emp@4e25154f, com.ddlab.rnd.core1.Emp@70dea4e, com.ddlab.rnd.core1.Emp@5c647e05, com.ddlab.rnd.core1.Emp@33909752, com.ddlab.rnd.core1.Emp@55f96302, com.ddlab.rnd.core1.Emp@3d4eac69, com.ddlab.rnd.core1.Emp@42a57993]

---- Object Called -----

---- Object Called -----

---- Object Called -----

---- Object Called -----

---- Object Called -----

---- Object Called -----

---- Object Called -----

---- Object Called -----

---- Object Called -----

empList--->[com.ddlab.rnd.core1.Emp@15db9742, com.ddlab.rnd.core1.Emp@6d06d69c, com.ddlab.rnd.core1.Emp@7852e922, com.ddlab.rnd.core1.Emp@4e25154f, com.ddlab.rnd.core1.Emp@70dea4e, com.ddlab.rnd.core1.Emp@5c647e05, com.ddlab.rnd.core1.Emp@33909752, com.ddlab.rnd.core1.Emp@55f96302, com.ddlab.rnd.core1.Emp@3d4eac69, [com.ddlab.rnd.core1.Emp@42a57993](mailto:com.ddlab.rnd.core1.Emp@42a57993)]

It means when Collections.sort() method is invoked it will call Comparable or Comparator compareTo or compare method of type Object type if there is no specific generic type. It is because of the following reason as shown in example.

public static void show(String s) {

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

}

public static void show(Object o) {

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

}

public static void main(String[] args) {

show((Object)"abc");//----Object----

}

The above prints only Object as we are passing Object type by making upper casting. It is because of the general concept that if a clerk is promoted to an officer will he behave like Clerk or Officer, ans is officer.

# How to write a multi comparator

package com.ddlab.rnd.core;

import java.util.Arrays;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class MultiComparator<T> implements Comparator<T> {

**private final List<Comparator<? super T>> comparators;**

public MultiComparator(List<Comparator<? super T>> comparators) {

this.comparators = comparators;

}

public MultiComparator(Comparator<? super T>... comparators) {

this(Arrays.asList(comparators));

}

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

**for (Comparator<? super T> c : comparators) {**

**int result = c.compare(o1, o2);**

**if (result != 0) {**

**return result;**

**}**

**}**

**return 0;**

**}**

public static <T> void sort(List<T> list, Comparator<? super T>... comparators) {

Collections.sort(list, new MultiComparator<T>(comparators));

}

}

**How to use**

package com.ddlab.rnd.core;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.List;

class AgeComparator implements Comparator<Employee> {

@Override

public int compare(Employee o1, Employee o2) {

Employee e1 = (Employee) o1;

Employee e2 = (Employee) o2;

return e2.getAge() - e1.getAge();

}

}

class SalComparator implements Comparator<Employee> {

@Override

public int compare(Employee o1, Employee o2) {

Employee e1 = (Employee) o1;

Employee e2 = (Employee) o2;

return e2.getSal() - e2.getSal();

}

}

public class TestMultiComparator {

public static void main(String[] args) {

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

for (int i = 1000 , j = 23; i < 5000; i=i+1000 , j++) {

empList.add( new Employee(j, "Name-"+j, i));

}

System.out.println("empList--->"+empList);

List<Comparator<? super Employee>> multiList = new ArrayList<>();

multiList.add(new AgeComparator());

multiList.add(new SalComparator());

MultiComparator<Employee> multi = new MultiComparator<>(multiList);

multi.sort(empList, multi);

System.out.println("Multi----->"+empList);

}

}