Collections are like containers that group multiple items in a single unit. For example, a jar of chocolates, list of names, etc.

The Map is the only interface that doesn’t inherit from Collection interface but it’s part of the Collections framework. All the collections framework interfaces are present in java.util package.

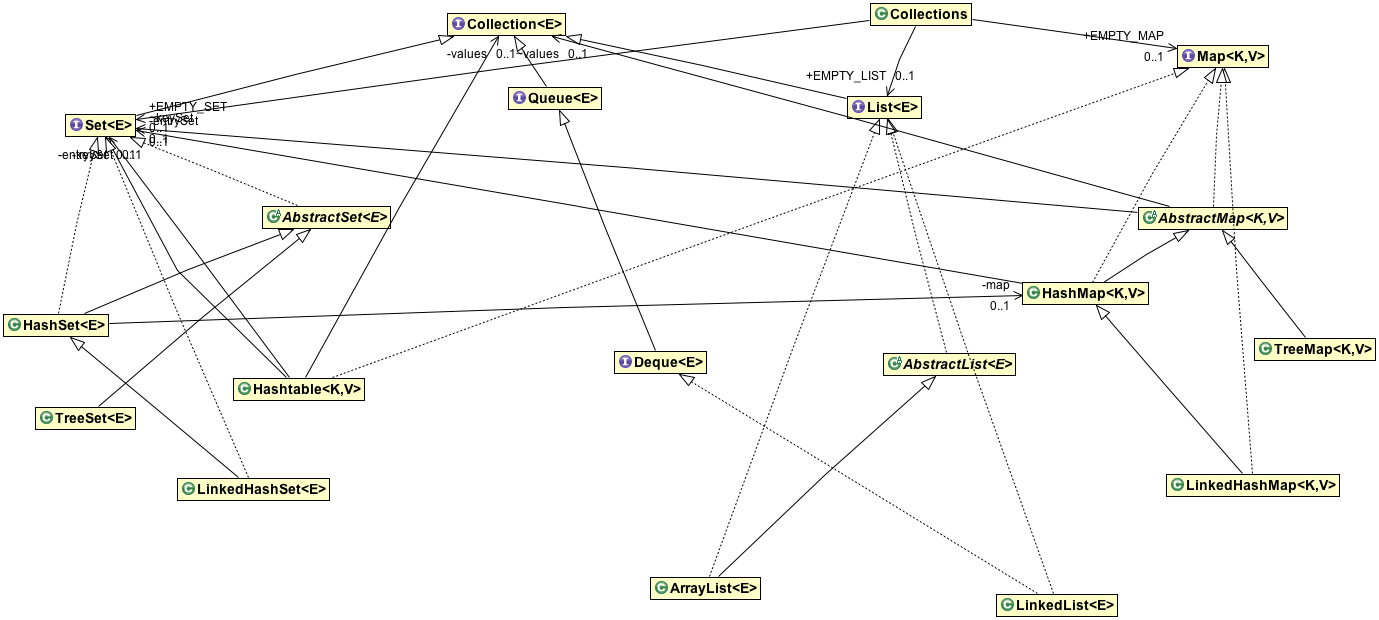
#### **What is a framework in Java**

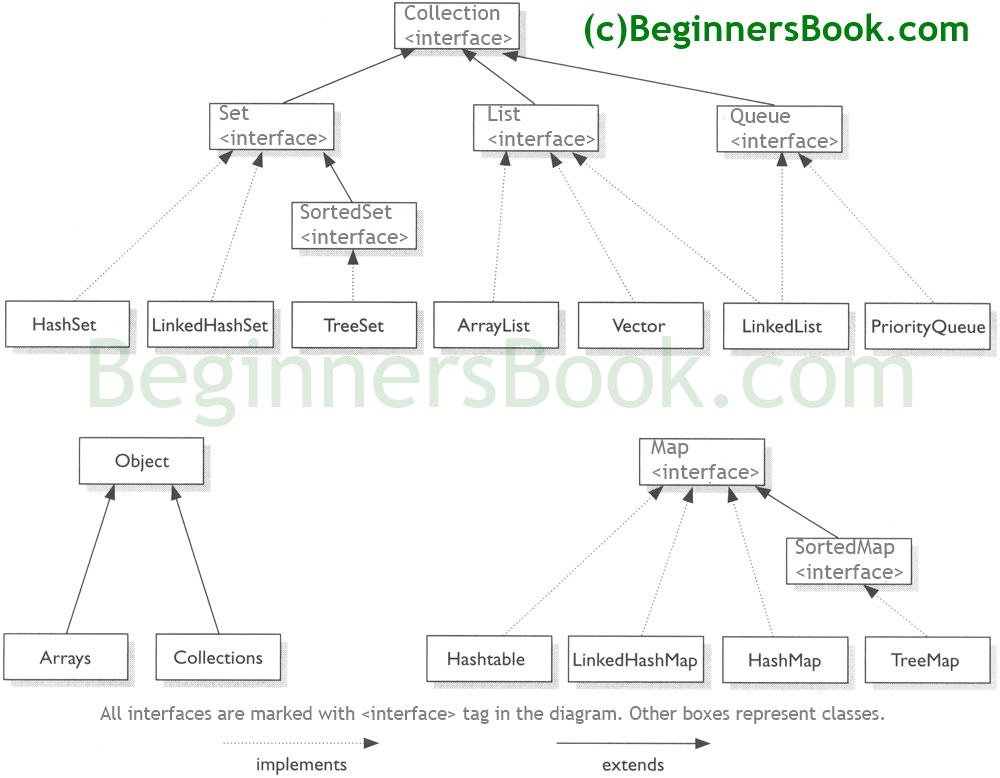
* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

#### **What is Collection framework**

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm





**Benefits of Java Collections Framework**

Java Collections framework have following benefits:

* **Reduced Development Effort** – It comes with almost all common types of collections and useful methods to iterate and manipulate the data. So we can concentrate more on business logic rather than designing our collection APIs.
* **Increased Quality** – Using core collection classes that are well tested increases our program quality rather than using any home developed data structure.
* **Reusability and Interoperability**
* **Reduce effort** – to learn any new API if we use core collection API classes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Main collection classes** | **D** | **O** | **S** | **TS** |
| **ArrayList** | **Yes** | **Yes** | **No** | **No** |
| **LinkedList** | **Yes** | **Yes** | **No** | **No** |
| **Vector** | **Yes** | **Yes** | **No** | **Yes** |
| **HashSet** | **No** | **No** | **No** | **No** |
| **LinkedHashSet** | **No** | **Yes** | **No** | **No** |
| **TreeSet** | **No** | **Yes** | **Yes** | **No** |
| **HashMap** | **No** | **No** | **No** | **No** |
| **LinkedHashMap** | **No** | **Yes** | **No** | **No** |
| **Hashtable** | **No** | **No** | **No** | **Yes** |
| **TreeMap** | **No** | **Yes** | **Yes** | **No** |

* **D:** Duplicate elements is allowed?
* **O:** Elements are ordered?
* **S:** Elements are sorted?
* **TS:** The collection is thread-safe?

From this table we can conclude the following characteristics of the main collections in Java Collection Frameworks:

* + All lists allow duplicate elements which are ordered by index.
  + All sets and maps do not allow duplicate elements.
  + All list elements are not sorted.
  + Generally, sets and maps do not sort its elements, except TreeSet and TreeMap – which sort elements by natural order or by a comparator.
  + Generally, elements within sets and maps are not ordered, except for:
    - * LinkedHashSet and LinkedHashMap have elements ordered by insertion order.
      * TreeSet and TreeMap have elements ordered by natural order or by a comparator.
  + There are only two collections are thread-safe: Vectorand Hastable. The rest is not thread-safe.

The only two legacy collections are thread-safe: Vector and Hashtable.

## Synchronized Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — Collection, Set, List, Map, SortedSet, and SortedMap — has one static factory method.

public static Collection synchronizedCollection(Collection c);

public static Set synchronizedSet(Set s);

public static List synchronizedList(List list);

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

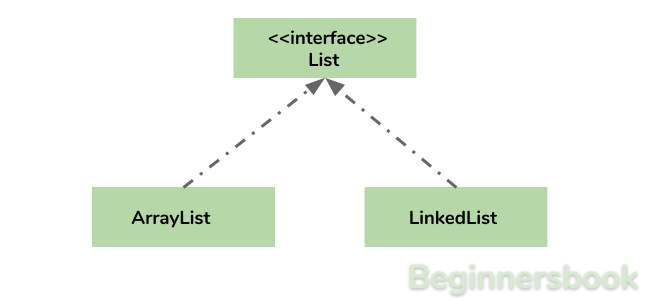
public static SortedSet synchronizedSortedSet(SortedSet s);

public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);

# ArrayList in java with example programs – Collections Framework

BY CHAITANYA SINGH | FILED UNDER: [JAVA COLLECTIONS](https://beginnersbook.com/category/java-collections/)

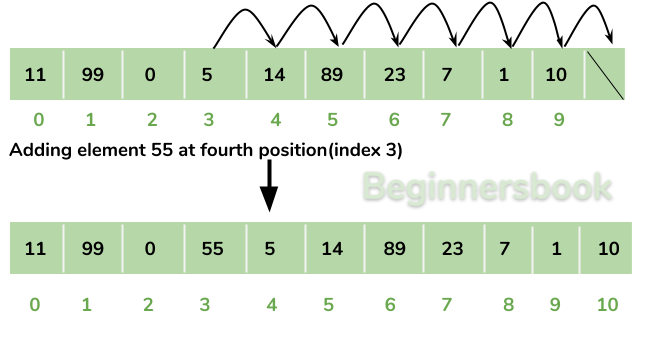
**Arraylist** class implements List interface and it is based on an Array data structure. It is widely used because of the functionality and flexibility it offers. Most of the developers **choose Arraylist over Array** as it’s a very good alternative of traditional java arrays. ArrayList is a resizable-array implementation of the List interface. It implements all optional list operations, and permits all elements, including null.



## Why ArrayList is better than Array?

The limitation with array is that it has a fixed length so if it is full you cannot add any more elements to it, likewise if there are number of elements gets removed from it the memory consumption would be the same as it doesn’t shrink.

On the other ArrayList can dynamically grow and shrink after addition and removal of elements (See the images below). Apart from these benefits ArrayList class enables us to use predefined methods of it which makes our task easy. Let’s see the diagrams to understand the addition and removal of elements from ArrayList and then we will see the programs.

**Adding Element in ArrayList at specified position:**  


**Removing Element from ArrayList:**  


There is a list of several tutorials on ArrayList at the end of this guide, refer it to understand and learn ArrayList concept fully.

## How to create an ArrayList?

We can create an ArrayList by writing a simple statement like this:

This statement creates an ArrayList with the name alist with type “String”. The type determines which type of elements the list will have. Since this list is of “String” type, the elements that are going to be added to this list will be of type “String”.

ArrayList<String> alist=new ArrayList<String>();

Similarly we can create ArrayList that accepts int elements.

ArrayList<Integer> list=new ArrayList<Integer>();

## How to add elements to an ArrayList?

We add elements to an ArrayList by using add() method, this method has couple of variations, which we can use based on the requirement. For example: If we want to add the element at the end of the List then simply do it like this:

alist.add("Steve"); //This will add "Steve" at the end of List

To add the element at the specified location in ArrayList, we can specify the index in the add method like this:

alist.add(3, "Steve"); //This will add "Steve" at the fourth position

Lets write the complete code:

import java.util.\*;

class JavaExample{

public static void main(String args[]){

ArrayList<String> alist=new ArrayList<String>();

alist.add("Steve");

alist.add("Tim");

alist.add("Lucy");

alist.add("Pat");

alist.add("Angela");

alist.add("Tom");

//displaying elements

System.out.println(alist);

//Adding "Steve" at the fourth position

alist.add(3, "Steve");

//displaying elements

System.out.println(alist);

}

}

**Output:**

[Steve, Tim, Lucy, Pat, Angela, Tom]

[Steve, Tim, Lucy, Steve, Pat, Angela, Tom]

**Note:** Since the index starts with 0, index 3 would represent fourth position not 3.

## How to remove elements from ArrayList?

We use remove() method to remove elements from an ArrayList, Same as add() method, this method also has few variations.

For example:

import java.util.\*;

class JavaExample{

public static void main(String args[]){

ArrayList<String> alist=new ArrayList<String>();

alist.add("Steve");

alist.add("Tim");

alist.add("Lucy");

alist.add("Pat");

alist.add("Angela");

alist.add("Tom");

//displaying elements

System.out.println(alist);

//Removing "Steve" and "Angela"

alist.remove("Steve");

alist.remove("Angela");

//displaying elements

System.out.println(alist);

//Removing 3rd element

alist.remove(2);

//displaying elements

System.out.println(alist);

}

}

**Output:**

[Steve, Tim, Lucy, Pat, Angela, Tom]

[Tim, Lucy, Pat, Tom]

[Tim, Lucy, Tom]

## Iterating ArrayList

In the above examples, we have displayed the ArrayList elements just by referring the ArrayList instance, which is definitely not the right way to displays the elements. The correct way of displaying the elements is by using an advanced for loop like this.

import java.util.\*;

class JavaExample{

public static void main(String args[]){

ArrayList<String> alist=new ArrayList<String>();

alist.add("Gregor Clegane");

alist.add("Khal Drogo");

alist.add("Cersei Lannister");

alist.add("Sandor Clegane");

alist.add("Tyrion Lannister");

//iterating ArrayList

for(String str:alist)

System.out.println(str);

}

}

**Output:**

Gregor Clegane

Khal Drogo

Cersei Lannister

Sandor Clegane

Tyrion Lannister

## ArrayList Example in Java

This example demonstrates how to create, initialize, add and remove elements from ArrayList. In this example we have an ArrayList of type “String”. We have added 5 String element in the ArrayList using the method add(String E), this method adds the element at the end of the ArrayList.

We are then adding two more elements in the ArrayList using method add(int index, String E), this method adds the specified element at the specified index, index 0 indicates first position and 1 indicates second position.

We are then removing the elements “Chaitanya” and “Harry” from the ArrayList and then we are removing the second element of the ArrayList using method remove(int index). Since we have specified the index as 1 (remove(1)), it would remove the second element.

import java.util.\*;

public class JavaExample {

public static void main(String args[]) {

/\* Creating ArrayList of type "String" which means

\* we can only add "String" elements

\*/

ArrayList<String> obj = new ArrayList<String>();

/\*This is how we add elements to an ArrayList\*/

obj.add("Ajeet");

obj.add("Harry");

obj.add("Chaitanya");

obj.add("Steve");

obj.add("Anuj");

// Displaying elements

System.out.println("Original ArrayList:");

for(String str:obj)

System.out.println(str);

/\* Add element at the given index

\* obj.add(0, "Rahul") - Adding element "Rahul" at first position

\* obj.add(1, "Justin") - Adding element "Justin" at second position

\*/

obj.add(0, "Rahul");

obj.add(1, "Justin");

// Displaying elements

System.out.println("ArrayList after add operation:");

for(String str:obj)

System.out.println(str);

//Remove elements from ArrayList like this

obj.remove("Chaitanya"); //Removes "Chaitanya" from ArrayList

obj.remove("Harry"); //Removes "Harry" from ArrayList

// Displaying elements

System.out.println("ArrayList after remove operation:");

for(String str:obj)

System.out.println(str);

//Remove element from the specified index

obj.remove(1); //Removes Second element from the List

// Displaying elements

System.out.println("Final ArrayList:");

for(String str:obj)

System.out.println(str);

}

}

**Output:**

Original ArrayList:

Ajeet

Harry

Chaitanya

Steve

Anuj

ArrayList after add operation:

Rahul

Justin

Ajeet

Harry

Chaitanya

Steve

Anuj

ArrayList after remove operation:

Rahul

Justin

Ajeet

Steve

Anuj

Final ArrayList:

Rahul

Ajeet

Steve

Anuj

## Methods of ArrayList class

In the above example we have used methods such as add() and remove(). However there are number of methods available which can be used directly using object of ArrayList class. Let’s discuss few important methods of ArrayList class.

1) **add( Object o)**: This method adds an object o to the arraylist.

obj.add("hello");

This statement would add a string hello in the arraylist at last position.

2) **add(int index, Object o)**: It adds the object o to the array list at the given index.

obj.add(2, "bye");

It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of array list.

3) **remove(Object o)**: Removes the object o from the ArrayList.

obj.remove("Chaitanya");

This statement will remove the string “Chaitanya” from the ArrayList.

4) **remove(int index)**: Removes element from a given index.

obj.remove(3);

It would remove the element of index 3 (4th element of the list – List starts with o).

5) **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.

obj.set(2, "Tom");

It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6)**int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

int pos = obj.indexOf("Tom");

This would give the index (position) of the string Tom in the list.

7) **Object get(int index)**: It returns the object of list which is present at the specified index.

String str= obj.get(2);

Function get would return the string stored at 3rd position (index 2) and would be assigned to the string “str”. We have stored the returned value in string variable because in our example we have defined the ArrayList is of String type. If you are having integer array list then the returned value should be stored in an integer variable.

8) **int size()**: It gives the size of the ArrayList – Number of elements of the list.

int numberofitems = obj.size();

9) **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.

obj.contains("Steve");

It would return true if the string “Steve” is present in the list else we would get false.

10) **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

obj.clear();

Sorting:

Comparable & Comparator;

Comparable and Comparator in Java are very useful for sorting the collection of objects. Java provides some inbuilt methods to sort primitive types array or Wrapper classes array or list. Here we will first learn how we can sort an array/list of primitive types and wrapper classes and then we will use **java.lang.Comparable** and **java.util.Comparator** interfaces to sort array/list of custom classes.

Let’s see how we can sort primitive types or Object array and list with a simple program.

package com.journaldev.sort;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class JavaObjectSorting {

/\*\*

\* This class shows how to sort primitive arrays,

\* Wrapper classes Object Arrays

\* @param args

\*/

public static void main(String[] args) {

//sort primitives array like int array

int[] intArr = {5,9,1,10};

Arrays.sort(intArr);

System.out.println(Arrays.toString(intArr));

//sorting String array

String[] strArr = {"A", "C", "B", "Z", "E"};

Arrays.sort(strArr);

System.out.println(Arrays.toString(strArr));

//sorting list of objects of Wrapper classes

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

strList.add("A");

strList.add("C");

strList.add("B");

strList.add("Z");

strList.add("E");

Collections.sort(strList);

for(String str: strList) System.out.print(" "+str);

}

}

Output of the above program is:

[1, 5, 9, 10]

[A, B, C, E, Z]

A B C E Z

Now let’s try to sort an array of objects.

package com.journaldev.sort;

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

this.id = id;

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

//this is overridden to print the user-friendly information about the Employee

public String toString() {

return "[id=" + this.id + ", name=" + this.name + ", age=" + this.age + ", salary=" +

this.salary + "]";

}

}

Here is the code I used to sort the array of Employee objects.

//sorting object array

Employee[] empArr = new Employee[4];

empArr[0] = new Employee(10, "Mikey", 25, 10000);

empArr[1] = new Employee(20, "Arun", 29, 20000);

empArr[2] = new Employee(5, "Lisa", 35, 5000);

empArr[3] = new Employee(1, "Pankaj", 32, 50000);

//sorting employees array using Comparable interface implementation

Arrays.sort(empArr);

System.out.println("Default Sorting of Employees list:\n"+Arrays.toString(empArr));

When I tried to run this, it throws the following runtime exception.

Exception in thread "main" java.lang.ClassCastException: com.journaldev.sort.Employee cannot be cast to java.lang.Comparable

at java.util.ComparableTimSort.countRunAndMakeAscending(ComparableTimSort.java:290)

at java.util.ComparableTimSort.sort(ComparableTimSort.java:157)

at java.util.ComparableTimSort.sort(ComparableTimSort.java:146)

at java.util.Arrays.sort(Arrays.java:472)

at com.journaldev.sort.JavaSorting.main(JavaSorting.java:41)

## Comparable and Comparator

Java provides **Comparable** interface which should be implemented by any custom class if we want to use [Arrays](https://www.journaldev.com/16770/java-arrays-java-util-arrays) or [Collections](https://www.journaldev.com/16635/collections-class-java-util-collections) sorting methods.

The Comparable interface has **compareTo(T obj)** method which is used by sorting methods, you can check any Wrapper, String or Date class to confirm this. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as an argument.

After implementing Comparable [interface](https://www.journaldev.com/1601/interface-in-java) in Employee class, here is the resulting Employee class.

## Java Comparator

Comparator interface compare(Object o1, Object o2) method need to be implemented that takes two Object argument, it should be implemented in such a way that it returns negative int if the first argument is less than the second one and returns zero if they are equal and positive int if the first argument is greater than the second one.

Comparable and Comparator interfaces use [Generics](https://www.journaldev.com/1663/java-generics-example-method-class-interface) for compile-time type checking, learn more about [Java Generics](https://www.journaldev.com/1663/java-generics-example-method-class-interface).

Here is how we can create different Comparator implementation in the Employee class.

# **Difference between Comparable and Comparator**

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. |
| 2) Comparable **affects the original class**, i.e., the actual class is modified. | Comparator **doesn't affect the original class**, i.e., the actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is present in **java.lang** package. | A Comparator is present in the **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method. |

## Java Comparable Example

Let's see the example of a Comparable interface that sorts the list elements on the basis of age.

*File: TestSort3.java*

1. //Java Program to demonstrate the use of Java Comparable.
2. //Creating a class which implements Comparable Interface
3. **import** java.util.\*;
4. **import** java.io.\*;
5. **class** Student **implements** Comparable<Student>{
6. **int** rollno;
7. String name;
8. **int** age;
9. Student(**int** rollno,String name,**int** age){
10. **this**.rollno=rollno;
11. **this**.name=name;
12. **this**.age=age;
13. }
14. **public** **int** compareTo(Student st){
15. **if**(age==st.age)
16. **return** 0;
17. **else** **if**(age>st.age)
18. **return** 1;
19. **else**
20. **return** -1;
21. }
22. }
23. //Creating a test class to sort the elements
24. **public** **class** TestSort3{
25. **public** **static** **void** main(String args[]){
26. ArrayList<Student> al=**new** ArrayList<Student>();
27. al.add(**new** Student(101,"Vijay",23));
28. al.add(**new** Student(106,"Ajay",27));
29. al.add(**new** Student(105,"Jai",21));
31. Collections.sort(al);
32. **for**(Student st:al){
33. System.out.println(st.rollno+" "+st.name+" "+st.age);
34. }
35. }
36. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSort3)

Output:

105 Jai 21

101 Vijay 23

106 Ajay 27

## Java Comparator Example

Let's see an example of the Java Comparator interface where we are sorting the elements of a list using different comparators.

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **if**(s1.age==s2.age)
5. **return** 0;
6. **else** **if**(s1.age>s2.age)
7. **return** 1;
8. **else**
9. **return** -1;
10. }
11. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **return** s1.name.compareTo(s2.name);
5. }
6. }

**TestComparator.java**

In this class, we are printing the values of the object by sorting on the basis of name and age.

1. //Java Program to demonstrate the use of Java Comparator
2. **import** java.util.\*;
3. **import** java.io.\*;
4. **class** TestComparator{
5. **public** **static** **void** main(String args[]){
6. //Creating a list of students
7. ArrayList<Student> al=**new** ArrayList<Student>();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
12. System.out.println("Sorting by Name");
13. //Using NameComparator to sort the elements
14. Collections.sort(al,**new** NameComparator());
15. //Traversing the elements of list
16. **for**(Student st: al){
17. System.out.println(st.rollno+" "+st.name+" "+st.age);
18. }
20. System.out.println("sorting by Age");
21. //Using AgeComparator to sort the elements
22. Collections.sort(al,**new** AgeComparator());
23. //Travering the list again
24. **for**(Student st: al){
25. System.out.println(st.rollno+" "+st.name+" "+st.age);
26. }
28. }
29. }

Output:

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by Age

105 Jai 21

101 Vijay 23

106 Ajay 27

//<https://www.journaldev.com/552/java-tricky-interview-questions>

# Hashcode() and equals()

# **Overriding Equals and Hashcode Method in Java Example**

   By [Dhiraj](https://www.linkedin.com/in/dhiraj-ray-devglan" \t "_blank),   25 February, 20174K

In this article we will be discussing about what is equals() and hashcode() method defined in Object class in java along with why is it required to override them. We will also take a look into how to override these methods, the contract between equals() and hashcode() method, things to remember while overriding equals() and hashcode() and about different automatic ways to generate overriden methods.

## What is equals() and hashcode()

Both of these are methods defined in java.lang.Object class. We use equals() method to compare if two objects are meaningfully equivalent means whether the two objects themselves(not the references) are equal(). To check whether the object references are equal or not, we use == operator which again uses object hash code value to evaluate it.

Coming to hashcode(), it is used to generate hash code value for an object and this hash code value is used by some collection classes to compare objects which in turn increases the performance of large collections of objects.

Let us define our Emplyee class for which we will be overriding the hashcode() and equals() method.

package com.devglan.core;

public class Employee {

private String name;

private int age;

private String address;

public Employee(){}

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

this.age = age;

this.name = name;

this.address = address;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

## Why Overriding Equals() and Hashcode() is Required

First let us look into the default implementation of equals() in java.lang.Object class.

public boolean equals(Object obj) {

return (this == obj);

}

Above code sample shows that the default implementation only compares the object reference to decide whether an object is equal or not and these references are nothing but the hash code values which is generated by hashcode() method.So, unless you override equals() and hashcode() two objects are considered equal only if the two references refer to the same object and hence, there will be mismatch while using these objects as a key in any java collections.

For example let us create a map with key as an object and string as value as below. Populate it with some dummy entries. Now, create that same object with same set of prameters and try to fetch value from that map and see the output.

Following is an example.

package com.devglan.core;

import java.util.HashMap;

import java.util.Map;

public class HashcodeTest {

public static void main(String [] args){

Map<Employee, String> empMap = new HashMap<Employee, String>();

Employee emp1 = new Employee("John", 23, "Bangalore");

Employee emp2 = new Employee("Max", 29, "Chennai");

empMap.put(emp1, "John details");

empMap.put(emp2, "Max details");

System.out.println(emp1);

System.out.println(emp2);

Employee emp3 = new Employee("John", 23, "Bangalore");

System.out.println(empMap.get(emp3));

}

}

If you run above java class, you will see an output as below in the console

com.devglan.core.Employee@15db9742

com.devglan.core.Employee@6d06d69c

null

Though that map has that object as a key but you got null because you have not overriden equals() and hashcode() and java is using its own default implementations to compare objects and generate hash codes and hence there is a mismatch. But if you override the equals() and hashcode() method you will get the perfect result as hashmap internally uses hashcode() method to generate hash code of any object and equals() method to check if those objects are meaningfully equal.

**Other Interesting Posts**

[Why wait(), notify(), notifyAll() Defined in Object class](https://www.devglan.com/corejava/why-wait-notify-notifyall-defined-in-object-class)

[Comparable and Comparator in Java with Example](https://www.devglan.com/corejava/comparable-and-comparator-in-java-with-example)

[Sorting HashMap by Key and Value](https://www.devglan.com/corejava/sorting-hashmap-by-key-and-value)

[Hello World Java Program Breakup](https://www.devglan.com/corejava/hello-world-java-program-breakup)

[Serialization and Deserialization in Java](https://www.devglan.com/corejava/serialization-and-deserialization-java-example)

[Convert hashmap to List in java](https://www.devglan.com/corejava/convert-hashmap-to-list-java)

## How to Override HashCode Method

Following is the sample code we added in the Emplyee class to override the hashcode() method

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + (name == null ? 0 :name.hashCode());

result = prime \* result + age;

result = prime \* result + (address == null ? 0 : address.hashCode());

return result;

}

Here the idea to use 31 as hash is just to ensure distinct hashcode for distinct object. You need to calculate hash for different memeber and return the total as a unique hash code.

## How to Override Equals() Method

Following is the sample code we added in the Emplyee class to override the equals() method

@Override

public boolean equals(Object obj){

if(obj == null){

return false;

}

if(obj instanceof Employee && this == obj) {

return true;

}

Employee newEmp = (Employee)obj;

if(age != newEmp.age) {

return false;

}

if(name != null && !name.equalsIgnoreCase(newEmp.name)){

return false;

}

if(address != null && !address.equalsIgnoreCase(newEmp.address)){

return false;

}

return true;

}

While overriding equals method, it is very much required to check for null condition and proper object casting. We have used instanceof check here before typecasting. Then we are checking whether each memebers of the object are meaningfully equivalent or not to another object. Based on this the final boolean result is evaluated.

Now, again you run the class HashcodeTest.java as a java application and you will get a perfect result as below:

com.devglan.core.Employee@8ac51512

com.devglan.core.Employee@93e7d68e

John details

## Equals and HashCode Contract

The hashcode() and equals() methods contract can be summarized as below;

1. If two objects are equal by equals() method then their hash code values must be same.

2. If two objects are not equal by equals() method then thier hash code may be same or different.

## Automatic Generation of Equals() and HashCode()

The different IDE tools such as NetBeans, Eclipse, IntellijIdea have default support to generate hashcode() and equals() overriden methods. But sometimes that is not very useful. It completely depends on the data structure of your pojo.

SpringUtils has also support to override equals() and hashcode() methods but again that depends on the complexity.

Also there are is an opensource apache library that can override hashcode() and equals() method.

**Conclusion**

In order to achieve a fully working custom equality mechanism, it is mandatory to override **hashcode()**each time you override ***equals().*** Follow the tips below and you'll never have leaks in your custom equality mechanism:

* If two objects are equal, they MUST have the same hash code.
* If two objects have the same hash code, it doesn't mean that they are equal.
* Overriding ***equals()***alone will make your business fail with hashing data structures like: ***HashSet, HashMap, HashTable*** ... etc.
* Overriding ***hashcode()***alone doesn't force Java to ignore memory addresses when comparing two objects.