

Day 9- Facilitation Guide

(TreeMap,Comparator,Comparable)

Index

- I. TreeMap
- II. Comparable and Comparator
- III. Overriding hashCode() & equals() method

For (1.5 hrs) ILT

During the previous session we learnt about LinkedHashMap,TreeSet and the Map interface in Java. We covered a variety of important concepts:

LinkedHashSet:

LinkedHashSet is a class in Java that extends HashSet.It maintains a linked list of the entries in the set, which provides predictable iteration order based on the order of insertion.Like HashSet, it does not allow duplicate elements.

TreeSet:

TreeSet is a class in Java that implements the SortedSet interface, which extends Set.It stores elements in a sorted tree structure (usually a Red-Black Tree) based on their natural order or a custom comparator.

HashMap:

HashMap is a class in Java that implements the Map interface.It stores key-value pairs and allows you to access values based on their associated keys.Does not maintain any particular order of the elements.Allows null keys and null values.

LinkedHashMap:

LinkedHashMap is a class in Java that extends HashMap.It maintains the order of elements based on their insertion order, in addition to providing key-value storage.Combines the functionality of a hash table and a linked list, offering both quick access and predictable iteration order.

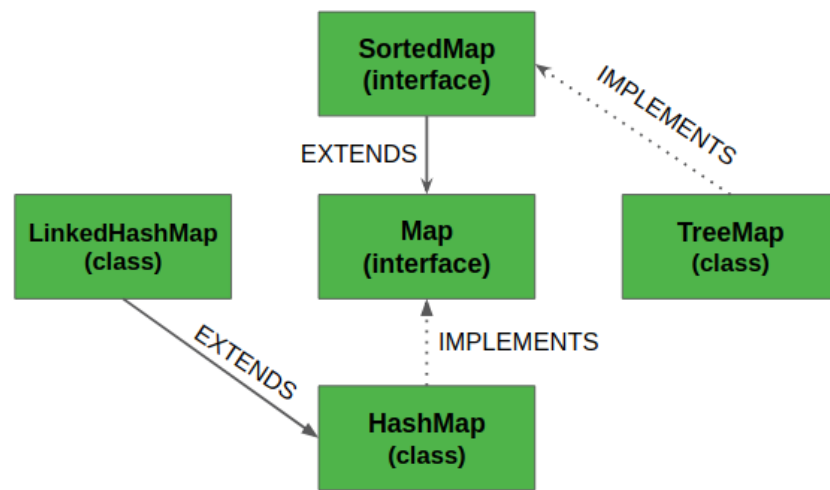
During this session, we will learn about TreeMap, the Comparator and Comparable interfaces, as well as the significance of overriding the hashCode() and equals() methods.

I. TreeMap in Java

TreeMap is a class that implements the Map interface and is part of the Java Collections Framework. It is designed to store key-value pairs in a sorted order based on the natural ordering of keys or a custom comparator. The keys in a TreeMap are stored in a red-black tree data structure, which allows for efficient retrieval, insertion, and deletion operations while maintaining a sorted order.

Key features and characteristics of the TreeMap class:

- **Sorted Order:** The primary feature of TreeMap is its automatic sorting of keys. Keys are sorted either in their natural order (if they implement the Comparable interface) or according to a provided custom comparator.
- **Efficiency:** The TreeMap provides efficient performance for basic operations like adding, removing, and retrieving elements. These operations have a time complexity of $O(\log n)$, making it suitable for scenarios where sorted access to data is required.
- **Navigational Methods:** TreeMap offers methods for navigating through the map using ranges of keys, such as firstKey(), lastKey(), lowerKey(), higherKey(), subMap(), and headMap(), which allow you to work with subsets of the map based on key ranges.
- **Implementation of SortedMap:** TreeMap also implements the SortedMap interface, which extends the Map interface and provides additional methods for dealing with sorted maps.



MAP Hierarchy in Java

Creating a TreeMap:

To create a TreeMap, you can simply instantiate it like this:

```
import java.util.TreeMap;
```

```
public class TreeMapExample {  
    public static void main(String[] args) {  
        // Creating a TreeMap with key as Integer and value as String  
        TreeMap<Integer, String> treeMap = new TreeMap<>();  
  
        // Adding elements to the TreeMap  
        treeMap.put(3, "Three");  
        treeMap.put(1, "One");  
        treeMap.put(2, "Two");  
    }  
}
```

Features of a TreeMap

Some important features of the TreeMap are as follows:

1. TreeMap is a member of the Java Collections Framework.
2. It implements Map interfaces including NavigableMap, SortedMap, and extends AbstractMap class.

3. TreeMap in Java does not allow null keys (like Map) and thus a NullPointerException is thrown. However, multiple null values can be associated with different keys.
4. Entry pairs returned by the methods in this class and its views represent snapshots of mappings at the time they were produced. They do not support the Entry.setValue method.

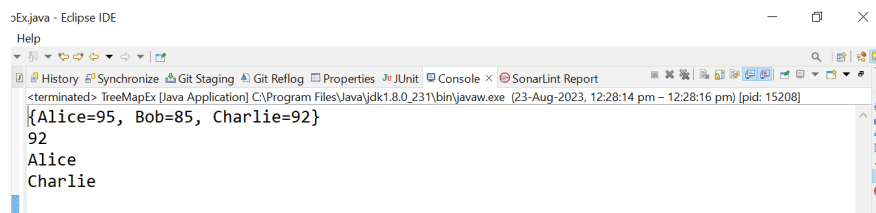
Let's take a closer look at a full example using TreeMap:

```
import java.util.TreeMap;
public class TreeMapEx {
    public static void main(String[] args) {

TreeMap<String, Integer> grades = new TreeMap<>();
grades.put("Alice", 95);
grades.put("Bob", 85);
grades.put("Charlie", 92);

System.out.println(grades); // Output: {Alice=95, Bob=85, Charlie=92}
System.out.println(grades.get("Charlie")); // Output: 92
System.out.println(grades.firstKey()); // Output: Alice
System.out.println(grades.lastKey()); // Output: Charlie
    }
}
```

Output:



In this example, the TreeMap automatically maintains the keys in sorted order, which makes it useful when you need to maintain data that should be accessible in a specific order. However, keep in mind that while TreeMap offers efficient sorted operations, it may have slightly slower performance than HashMap for certain scenarios due to the overhead of maintaining the sorted structure.

Here is an overview of some of the commonly used methods of the TreeMap class in Java along with examples:

1. put(key, value): This method is used to insert a key-value pair into the TreeMap.
Comparable Interface.

```
TreeMap<Integer, String> treeMap = new TreeMap<>();  
treeMap.put(3, "Three");  
treeMap.put(1, "One");  
treeMap.put(2, "Two");
```

2.get(key): This method retrieves the value associated with a given key.
String value = treeMap.get(2); // Retrieves "Two"

3.remove(key): This method removes the entry with the specified key from the TreeMap.

```
treeMap.remove(1); // Removes the entry with key 1
```

4.keySet(): This method returns a set containing all the keys in the TreeMap.
Set<Integer> keys = treeMap.keySet();

5.values(): This method returns a collection of all the values in the TreeMap.
Collection<String> values = treeMap.values();

6.entrySet(): This method returns a set of key-value pairs (Map.Entry objects) in the TreeMap.

```
Set<Map.Entry<Integer, String>> entries = treeMap.entrySet();
```

7.firstKey(): This method returns the first (lowest) key in the TreeMap.
Integer firstKey = treeMap.firstKey();

8.lastKey(): This method returns the last (highest) key in the TreeMap.
java

```
Integer lastKey = treeMap.lastKey();
```

9.higherKey(key): This method returns the least key strictly greater than the given key, or null if there is no such key.

```
Integer higher = treeMap.higherKey(1); // Returns 2
```

10.lowerKey(key): This method returns the greatest key strictly less than the given key, or null if there is no such key.

Integer lower = treeMap.lowerKey(2); // Returns 1

11.ceilingKey(key): This method returns the least key greater than or equal to the given key, or null if there is no such key.

Integer ceiling = treeMap.ceilingKey(2); // Returns 2

12.floorKey(key): This method returns the greatest key less than or equal to the given key, or null if there is no such key.

Integer floor = treeMap.floorKey(2); // Returns 2

These are just some of the methods available in the TreeMap class. You can use these methods to manipulate, retrieve, and navigate through the elements stored in the TreeMap. Keep in mind that most operations on a TreeMap have a time complexity of $O(\log n)$ due to the underlying balanced tree structure.

Let's check out a full example that uses all the TreeMap methods:

```
package queueExamples;
import java.util.Map;
import java.util.TreeMap;
public class TreeMapExample {
    public static void main(String[] args) {
        // Creating a TreeMap
        TreeMap<Integer, String> treeMap = new TreeMap<>();
        // Adding elements
        treeMap.put(3, "Three");
        treeMap.put(1, "One");
        treeMap.put(2, "Two");
        System.out.println("TreeMap: " + treeMap);
        // Retrieving values
        String value = treeMap.get(2);
        System.out.println("Value for key 2: " + value);
        // Removing an entry
        treeMap.remove(1);
        System.out.println("TreeMap after removing key 1: " + treeMap);
        // Iterating through keySet
        System.out.println("Keys:");
        for (Integer key : treeMap.keySet()) {
            System.out.println(key);
        }
    }
}
```

```

// Iterating through values
System.out.println("Values:");
for (String val : treeMap.values()) {
    System.out.println(val);
}

// Iterating through entrySet
System.out.println("Key-Value Pairs:");
for (Map.Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}

// First and last keys
Integer firstKey = treeMap.firstKey();
Integer lastKey = treeMap.lastKey();
System.out.println("First Key: " + firstKey);
System.out.println("Last Key: " + lastKey);

// Higher, lower, ceiling, and floor keys
Integer higherKey = treeMap.higherKey(1);
Integer lowerKey = treeMap.lowerKey(3);
Integer ceilingKey = treeMap.ceilingKey(2);
Integer floorKey = treeMap.floorKey(2);
System.out.println("Higher Key than 1: " + higherKey);
System.out.println("Lower Key than 3: " + lowerKey);
System.out.println("Ceiling Key than 2: " + ceilingKey);
System.out.println("Floor Key than 2: " + floorKey);
    }
}

```

Output:

```

Example.java - Eclipse IDE
/indow Help
<terminated> TreeMapExample [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (28-Aug-2023, 3:41:43 pm - 3:41:44 pm) [pid: 92]
[TreeMap: {1=One, 2=Two, 3=Three}
Value for key 2: Two
TreeMap after removing key 1: {2=Two, 3=Three}
Keys:
2
3
Values:
Two
Three
Key-Value Pairs:
2: Two
3: Three
First Key: 2
Last Key: 3
Higher Key than 1: 2
Lower Key than 3: 2
Ceiling Key than 2: 2
Floor Key than 2: 2

```

Food for thought:

The trainer can ask the students the following question to engage them in a discussion:

Now you have an understanding of TreeMap. Now imagine you're building a program that needs to keep track of appointments sorted by date and time. How could you utilize a TreeMap to achieve this efficiently?

II. Comparable and Comparator Interface

• Comparable Interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in the java.lang package and contains only one method named `compareTo(Object)`. It provides a single sorting sequence only, i.e., you can sort the elements on the basis of a single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

`public int compareTo(Object obj)`: It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Note:String class and wrapper classes implements the Comparable interface by default.So If you store the objects of string or wrapper classes in a list,set or map,it will be Comparable by default.

Note: To utilize Comparable, we must learn the sort() method found within the Collections class. Therefore, let's delve into the Collections class before delving into a practical example of how Comparable works.

Collections class

Collections class provides static methods for sorting the elements of a collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

Method of Collections class for sorting List elements

public static void sort(List myList): This method is present in java.util.Collections class. It is used to sort the elements present in the specified list of Collection in ascending order. It works similar to java.util.Arrays.sort() method but it is better than as it can sort the elements of Array as well as linked list, queue and many more present in it.

myList : A List type object we want to sort.

Note: This method doesn't return anything. This method we will use with the Comparable interface.

public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

Note: This method we will use with the Comparator interface.

Implementing Comparable:

To make a class comparable, you need to implement the Comparable interface and provide the compareTo method, which compares the current object with another object and returns a negative, zero, or positive integer based on the comparison.

```
public class Student implements Comparable<Student> {  
    private String name;  
    private int age;  
  
    // Constructor, getters, setters
```

```

    public int compareTo(Student otherStudent) {
        return this.age - otherStudent.age;
    }
}

```

Using Comparable for Sorting:

You can use the natural ordering defined by the compareTo method to sort objects in various Java classes like Arrays.sort(), Collections.sort(), and sorted data structures.

```

Student[] students = ...;
Arrays.sort(students); // Sorts based on age

```

```

List<Student> studentList = ...;
Collections.sort(studentList); // Sorts based on age

```

Comparing Other Fields:

You can compare objects based on multiple fields or attributes by chaining comparisons inside the compareTo method.

```

public int compareTo(Student otherStudent) {
    if (this.age != otherStudent.age) {
        return this.age - otherStudent.age;
    }
    return this.name.compareTo(otherStudent.name);
}

```

Let's check out a full example of Comparable interface:

Example:(Sort by Employee id)

```

package collectionsDemo;

//Creating class which is implemented Comparable interface
public class Employee implements Comparable<Employee> {
    private int id;
    private String name;
    private String address;

    //defining setter and getter method
    public int getId() {

```

```

        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
}

```

//defining default and parameterized constructor

```

public Employee(int id, String name, String address) {
    super();
    this.id = id;
    this.name = name;
    this.address = address;
}
public Employee() {
    super();
    // TODO Auto-generated constructor stub
}

```

//defining toString() method

@Override

```

public String toString() {
    return "Employee [Id=" + id + ", name=" + name + ", address=" + address + "];"
}

```

//Using compareTo() method to sort employee objects by id

@Override

```

public int compareTo(Employee o) //single sorting sequence only
{
    if(id==o.id) //emp1.id 10 emp2.id10

```

```

        return 0;
    else if(id>o.id)
        return 1;
    else
        return -1;
}
}

```

```
package collectionsDemo;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
```

```
public class ArrayListExample {

    public static void main(String[] args) {
        //Declaring arraylist of type Employee
        ArrayList<Employee> employees=new ArrayList<Employee>();

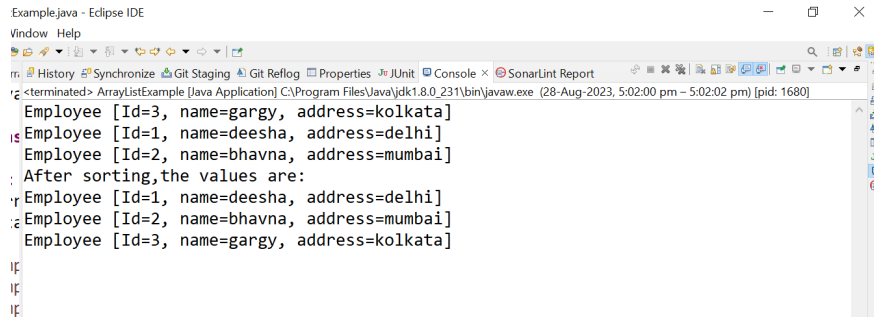
        Scanner sc=new Scanner(System.in);
        //adding 3 objects in to arraylist
        employees.add(new Employee(3, "gargy", "kolkata"));
        employees.add(new Employee(1, "deesha", "delhi"));
        employees.add(new Employee(2, "bhavna", "mumbai"));

        //displaying the elements using foreach loop
        for(Employee e:employees) {
            System.out.println(e);
        }

        //calling sort() method to sort employee based on id
        Collections.sort(employees);
        System.out.println("After sorting,the values are:");
        for(Employee e:employees) {
            System.out.println(e);
        }
    }
}

```

Output:

A screenshot of the Eclipse IDE interface. The main editor window displays a Java file named 'Example.java'. The code defines an 'Employee' class with attributes 'Id', 'name', and 'address'. It shows an initial list of three employees: [Id=3, name=gargy, address=kolkata], [Id=1, name=deesha, address=delhi], and [Id=2, name=bhavna, address=mumbai]. Below this, a comment states 'After sorting, the values are:', followed by the same three employees in sorted order: [Id=1, name=deesha, address=delhi], [Id=2, name=bhavna, address=mumbai], and [Id=3, name=gargy, address=kolkata]. The IDE's console window at the bottom shows the execution output, which matches the sorted list of employees. The top toolbar and menu bar are also visible.

Now that we've grasped the Comparable interface, let's proceed to comprehend the Comparator interface and its application.

- **Java Comparator interface:**

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in the java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollNo, name, age or anything else.

Methods of Java Comparator Interface:

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Creating a custom Comparator:

To create a custom comparator, you need to implement the Comparator interface and provide the compare method, which takes two objects as arguments and returns a negative, zero, or positive integer based on the comparison.

```
import java.util.Comparator;
```

```
class MyComparator implements Comparator<Integer> {  
    public int compare(Integer num1, Integer num2) {  
        return num2 - num1; // Descending order  
    }  
}
```

Using a Comparator:

You can use the custom comparator to sort objects using various Java classes such as TreeSet, TreeMap, and Collections.sort().

Let's take a look at a code snippet illustrating the Comparator interface:

```
import java.util.*;
```

```
public class ComparatorExample {  
    public static void main(String[] args) {  
        List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 3, 1));  
  
        // Using the custom comparator to sort in descending order  
        Collections.sort(numbers, new MyComparator());  
  
        System.out.println("Sorted numbers: " + numbers);  
    }  
}
```

Let's check out a full example of Comparator interface:

```
package collectiondemo;  
public class Student {  
    int rollNo;  
    String name;  
    int age;  
    Student(int rollNo,String name,int age){  
        this.rollNo=rollNo;  
        this.name=name;  
        this.age=age;  
    }  
}
```

```

    }
    public int getRollNo() {
        return rollNo;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}

```

AgeComparator.java:

This class defines comparison logic based on age. If the age of the first object is greater than the second, we are returning a positive value. It can be anyone such as 1, 2, 10. If the age of the first object is less than the second object, we are returning a negative value, it can be any negative value, and if the age of both objects is equal, we are returning 0.

```

import java.util.*;
class AgeComparator implements Comparator<Student>{
    public int compare(Student s1,Student s2){
        if(s1.age==s2.age)
            return 0;
        else if(s1.age>s2.age)
            return 1;
        else
            return -1;
    }
}

```

NameComparator.java

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

```

import java.util.*;
class NameComparator implements Comparator<Student>{
    public int compare(Student s1,Student s2){

```

```

return s1.name.compareTo(s2.name);
}
}

```

ComparatorDemo.java

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

```

package collectiondemo;
import java.util.ArrayList;
import java.util.Collections;

public class ComparatorDemo {
    public static void main(String[] args) {
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student(101,"Vijay",23));
        al.add(new Student(106,"Ajay",27));
        al.add(new Student(105,"Jai",21));

        System.out.println("Sorting by Name");

        //here we are invoking sort() method using NameComparator class to sort
list by name
        Collections.sort(al,new NameComparator());
        for(Student st: al){
            System.out.println(st.getRollNo()+" "+st.getName()+" "+st.getAge());
        }

        //here we are invoking sort() method using AgeComparator class to sort
age
list by

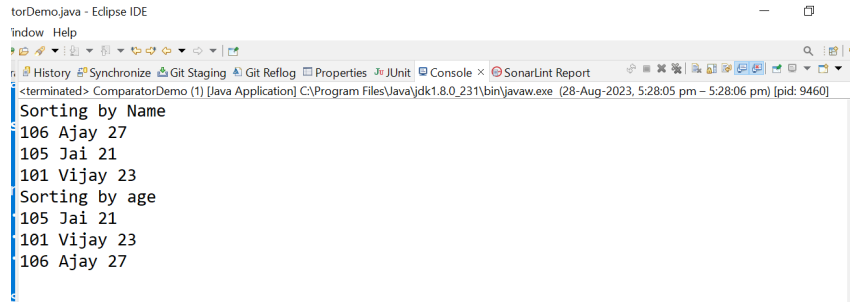
        System.out.println("Sorting by age");

        Collections.sort(al,new AgeComparator());
        for(Student st: al){
            System.out.println(st.getRollNo()+" "+st.getName()+"
st.getAge());
        }
    }
}

```


}

Output:



```
torDemo.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> ComparatorDemo (1) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (28-Aug-2023, 5:28:05 pm - 5:28:06 pm) [pid: 9460]
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Knowledge check...

The Trainer will conduct a short poll quiz to assess your knowledge on these topics.

1.What is the purpose of the Comparator interface in Java?

- a) To provide a default comparison logic for objects.
- b) To allow objects to be compared for reference equality.
- c) To define a custom comparison logic for objects.
- d) To ensure that objects are serializable.

2.When should you use Comparable over Comparator?

- a) When you want to compare objects of different classes.
- b) When you want to sort objects based on their natural order.
- c) When you want to sort objects using a custom comparison logic.
- d) When you want to sort objects using reference equality.

3.The higherKey method in TreeMap returns:

- a) The largest key in the TreeMap.
- b) The smallest key in the TreeMap.
- c) The key greater than the given key.
- d) The key lesser than the given key.

At the end of the quiz, the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.

III. Overriding hashCode() and equals()

The hashCode() and equals() methods in Java are used to determine object equality and to facilitate the correct functioning of certain data structures and algorithms. Here's the purpose of overriding these methods:

- **equals() Method:**

The equals() method is used to compare the equality of two objects. By default, the equals() method provided by the Object class checks for reference equality, meaning it compares whether two objects refer to the same memory location. However, in many cases, you may want to define your own notion of equality based on the object's state or specific criteria. Therefore, it is necessary to override the equals() method to provide a custom implementation.

When overriding equals(), you should follow these guidelines:

- Override equals() to compare the state of the objects rather than their references.
- Ensure that the equals() method is reflexive, symmetric, transitive, and consistent.
- Always override equals() when you override the hashCode() method (explained next) to maintain the contract between the two methods.

To override the equals() method, certain considerations should be kept in mind:

- Make sure the method has the same signature as public boolean equals(Object obj).
- Check if the argument obj is null and return false if it is.
- Use the instanceof operator to ensure that the argument is of the same class.
- Compare the relevant fields of the current object and the argument object to determine equality.
- Return true if the fields are equal, otherwise return false

Let's check out a full example:

```
public class Person {
```

```
private String name;  
private int age;
```

```
// Constructors, getters, setters
```

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null || getClass() != obj.getClass()) return false;  
    Person person = (Person) obj;  
    return age == person.age && Objects.equals(name, person.name);  
}  
}
```

- **hashCode() Method:**

The hashCode() method returns a hash code value that represents the object. Hash codes are used by various data structures, such as hash tables or hash-based collections (e.g., HashSet, HashMap), to efficiently store and retrieve objects.

When overriding hashCode(), you should follow these guidelines:

Ensure that objects that are equal according to equals() have the same hash code.

Override hashCode() consistently with equals(). If two objects are equal, their hash codes must be the same.

Use relevant fields in the object's state to calculate the hash code.

Overriding hashCode() Method:

The hashCode() method should be overridden consistently with the equals() method:

- 1.If two objects are equal according to the equals() method, they must have the same hash code.
- 2.Override hashCode() to use the same fields that you used for equals().

```
@Override
```

```
public int hashCode() {  
  
    return Objects.hash(name, age);  
}
```

```
}
```

By overriding `hashCode()` and `equals()` appropriately, you ensure that objects can be properly compared for equality and used effectively in data structures and algorithms that rely on these methods. This is particularly important when working with collections, such as `HashSet` or `HashMap`, as it ensures correct behavior when adding, removing, or searching for objects based on equality. `HashMap` and `HashSet` use the `hashCode` value of an object to find out how the object would be stored in the collection, and subsequently `hashCode` is used to help locate the object in the collection.

As such, Java will not produce any compilation error if `hashCode` is not overwritten along with `equals` but it will produce inconsistent results. Here is an example of `hashCode` implementation.

Let's explore the complete example:

```
import java.util.HashSet;
```

```
//here this class is having method hashCode() and equals() to check match the objects
```

```
public class HashCodeCheck {  
    private final String importantField;  
    private final String anotherField;  
    public HashCodeCheck(final String equalField, final String anotherField) {  
        this.importantField = equalField;  
        this.anotherField = anotherField;  
    }  
  
    //We have written hashCode() method with user define logic  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        result = prime * result  
            + ((importantField == null) ? 0 : importantField.hashCode());  
        return result;  
    }  
    @Override
```

```
//here we are defining equals() method where we are matching all parameters to check  
both objects are equal or not
```

```
public boolean equals(final Object obj) {
```

```

        if (this == obj) //here checking object's reference
            return true;
        if (obj == null) //checking object is null or not
            return false;
        if (getClass() != obj.getClass()) //checking both are from same class or not
            return false;
        //fetching hashCode()
        final HashCodeCheck other = (HashCodeCheck) obj;
        if (importantField == null) {
            if (other.importantField != null)
                return false;
        } else if (!importantField.equals(other.importantField))
            return false;
        return true;
    }

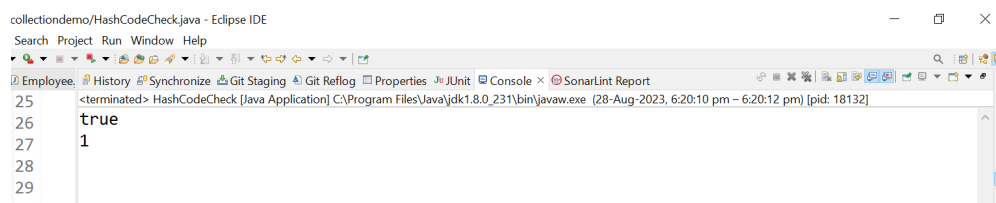
    public static void main(String[] args) {

        HashCodeCheck first = new HashCodeCheck("a","first");
        HashCodeCheck second = new HashCodeCheck("a","second");
        System.out.println (first.equals(second)); // true;
        HashSet <HashCodeCheck> test = new HashSet <HashCodeCheck> ();
        test.add(first);
        test.add(second); // It will not add as both the objects have same hash
code.

        System.out.println(test.size()); // It will be 1.
        // Now experiment by removing hashCode() or equals() or both.
    }
}

```

Output:



```

collectiondemo/HashCodeCheck.java - Eclipse IDE
Search Project Run Window Help
Employee... History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
25 <terminated> HashCodeCheck [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (28-Aug-2023, 6:20:10 pm - 6:20:12 pm) [pid: 18132]
26 true
27 1
28
29
30

```

In this example we have overridden both hashCode and equal. Now in this case two objects are equal. And they are considered the same while using Java collection classes. Because here hashCode is also implemented according to the value of the importantField.

Now let's look at an example of what happens if we don't override equals() but instead override hashCode() and try to use a Set. We will see the objects will not be equal if checked by **first.equals(second)**. However, HashSet will have only a single entry. Similarly if we do not implement hashCode but implement equals() then objects will be equal but there will be a single entry in the HashSet. Please check the result by removing either of the hashCode or both. Note that HashSet allows only unique values and uniqueness is judged by the hashCode value.

Note: You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.

Here is another example:

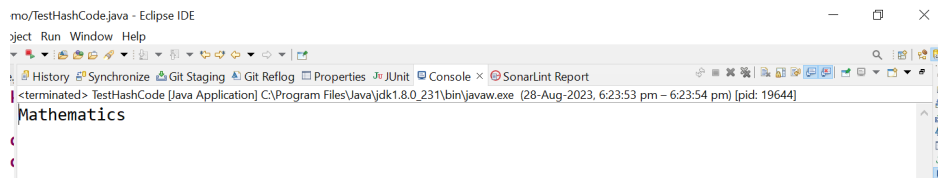
```
import java.io.*;
import java.util.*;
class HashCheck
{
    String name;
    int id;
    HashCheck(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
    @Override
    public boolean equals(Object obj)
    {
        if(this == obj)
            return true;
        if(obj == null || obj.getClass() != this.getClass())
            return false;
        HashCheck check = (HashCheck) obj;
        return (check.name.equals(this.name) && check.id == this.id);
    }
    @Override
    public int hashCode()
    {
        return this.id;
    }
}
```

```

    }
}
// Driver code
public class TestHashCode
{
    public static void main (String[] args)
    {
        HashCheck g1 = new HashCheck("Ajay", 1);
        HashCheck g2 = new HashCheck("Ajay", 1);
        Map<HashCheck, String> map = new HashMap<HashCheck, String>();
        map.put(g1, "History");
        map.put(g2, "Mathematics");
        for(HashCheck check : map.keySet())
        {
            System.out.println(map.get(check).toString());
        }
    }
}

```

Output:



In this case we override both methods properly. When we call **map.put(g1, "History")** it will hash to some bucket location and when we call **map.put(g2, "Mathematics")** it will generate the same hashCode value (same as g1) and replace the first value by second value. So, it replaces the old value of that key with a new value.

Overriding only the equals(Object) method

If we only override equals(Object) method, when we call **map.put(g1, "History")** it will hash to some bucket location and when we call **map.put(g2, "Mathematics")** it will hash to some other bucket location because of different hashCode value as hashCode() method has not been overridden. It will create two entries.

Overriding only hashCode() method

When we call `map.put(g1, "History")` it will generate a hashCode value and store it to the bucket location that is specified with this address (hashCode value). And when we call `map.put(g2, "Mathematics")` it generates the same hashCode value as the previous entry since key objects are the same and the hashCode() method has been overridden. So it should replace first with second as per rule. But it didn't. Reason is, when it iterates through that bucket and seeks to find k such that `k.equals("xyz")` i.e. if the searching key already exists. But it fails to find it because the equals(Object) method has not been overridden. It is a violation of the hashing rule.

Introducing ChatGPT

ChatGPT is a language model developed by OpenAI, and it's a sibling model to GPT-3. It's designed specifically for generating human-like text in a conversational context. ChatGPT is part of the GPT (Generative Pre-trained Transformer) family of models and is trained on a large corpus of text from the internet.

ChatGPT is an AI chatbot that uses natural language processing to create humanlike conversational dialogue. The language model can respond to questions and compose various written content, including articles, social media posts, essays, code and emails.

To use ChatGPT, you can access it through OpenAI's API. Here are the general steps to use ChatGPT programmatically:

Link: <https://chat.openai.com/>

Sign Up for Access If you haven't already, you need to sign up for access to OpenAI's API, as ChatGPT may not be freely available. Follow the registration process on the OpenAI website to get an API key.

Exercise:

Use ChatGPT to explore equals() and hashCode() method even in more deeper:

Put the below problem statement in the message box and see what ChatGPT says.

I'm learning Java. My instructor was discussing the concept of equals() method and hashCode() method. We have learnt already about this. I have one question in my mind,

Can you tell me how these methods contribute to the reliability and efficiency of searching and storing objects in data structures?

on New Responsibility x +

chat.openai.com

Default (GPT-3.5)

I'm learning Java. My instructor was discussing the concept of equals() method and hashCode() method. We have learnt already about this. I have one question arrived in my mind. Can you tell me How do these methods contribute to the reliability and efficiency of searching and storing objects in data structures?

Certainly! The `equals()` method and `hashCode()` method play a crucial role in the reliability and efficiency of searching and storing objects in data structures, especially in Java. Here's how they contribute to these aspects:

1. **Reliability in Searching:**

- **Equality Checking:** The `equals()` method is used to determine whether two objects are considered equal based on some defined criteria. This is essential for accurate searching because it ensures that you can find the exact object you're looking for.
- **Customized Comparison:** By overriding the `equals()` method in your classes, you can specify how objects of those classes should be compared for equality. This

Send a message

Regenerate

Activate Windows
Go to Settings to activate Windows.

Free Research Preview. ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT August 3 Version