* **Hashcode**
* Hashcode is a unique code generated by the JVM at time of object creation.
* The hashcode is always the same if the object doesn’t change
* An object can also be searched with this unique code.
* It can be used to perform some operation on hashing related algorithms like hashtable, hashmap etc.
* public int hashCode()
* returns integer value which represents hashcode.
* Objects that are equal (according to their *equals()*) must return the same hash code.
* Two **different** objects can have the **same** hash code.
* Hash codes are not guaranteed to be unique. They just help **speed things up** in data structures. If a conflict happens (same hash code for different objects), the system will **double-check** the actual objects to see if they’re really equal or not.

@Override

public int hashCode() {

return 1;

}

 This means: **Every user gets the same hash code** (1).

 It still follows Java’s rules — but it's very **inefficient**.

@Override

public int hashCode() {

int hash = 7;

hash = 31 \* hash + (int) id;

hash = 31 \* hash + (name == null ? 0 : name.hashCode());

hash = 31 \* hash + (email == null ? 0 : email.hashCode());

return hash;

}

 Uses **prime numbers** like 31 to get **better distribution** of objects.

 Helps avoid "hash collisions" — when two different objects get the same hash code.

A **collision** happens when **different objects** get the **same hash code**.

Java handles this by:

* Storing multiple objects in the **same "bucket"** using a **linked list**.
* If too many objects go into a bucket (Java 8+), it switches to a **tree** for faster access.

Map<User, User> users = new HashMap<>();

User user1 = new User(1L, "John", "john@domain.com");

users.put(user1, user1);

// New object with same data as user1

User userCopy = new User(1L, "John", "john@domain.com");

if (users.containsKey(userCopy)) {

System.out.print("User found in the collection");

} else {

System.out.print("User NOT found in the collection");

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

User user = (User) o;

return id == user.id &&

Objects.equals(name, user.name) &&

Objects.equals(email, user.email);

}

@Override

public int hashCode() {

int hash = 7;

hash = 31 \* hash + (int) id;

hash = 31 \* hash + (name == null ? 0 : name.hashCode());

hash = 31 \* hash + (email == null ? 0 : email.hashCode());

return hash;

}

Then:  
✅ user1.equals(userCopy) will return true  
✅ user1.hashCode() == userCopy.hashCode() will be true  
➡️ **So the map will find userCopy as a key.**

**✅ Output:**

User found in the collection

**❌ Case 2: If hashCode() or equals() Is NOT Overridden**

Then:

* user1.hashCode() != userCopy.hashCode() (different memory addresses)
* user1.equals(userCopy) is false (default behavior compares memory addresses)

➡️ **The map will not find userCopy**, even though the data is the same.

**❌ Output:**

pgsql

CopyEdit

User NOT found in the collection

* **Nested Interface in Java**
* **interfaces as members of a class**. Such an interface is called a**member interface or nested interface**.
* Interfaces declared outside any class can have only public and default access specifiers.
* In Java,**nested interfaces** can be declared with the **public, protected, package-private (default), or private access specifiers.**
* A top-level interface (not nested) can only be declared as public or package-private (default).

// protected specifier for nested interface

import java.util.\*;

class Parent {

protected interface Test {

void show();

}

}

class Child implements Parent.Test {

public void show(){

System.out.println("show method of interface");

}

}

// Driver Class

class GFG

{

public static void main(String[] args)

{

Parent.Test obj;

Child t = new Child();

obj = t;

obj.show();

}

}

// working of interface inside another interface

import java.util.\*;

// Nested Interface-Interface

interface Parent {

interface Test {

void show();

}

}

class Child implements Parent.Test {

public void show() {

System.out.println("show method of interface");

}

}

// Main Class

class GFG

{

public static void main(String[] args)

{

Parent.Test obj;

Child t = new Child();

obj = t;

obj.show();

}

}

// interface cannot have non-public member interface

import java.util.\*;

interface Parent {

protected interface Test {

void show();

}

}

class Child implements Parent.Test {

public void show()

{

System.out.println("show method of interface");

}

}

class GFG

{

public static void main(String[] args)

{

Parent.Test obj;

Child t = new Child();

obj = t;

obj.show();

}

}

Error

illegal combination of modifiers: public and protected  
 protected interface Yes

* **Marker Interface in Java**
* Marker Interface in Java is an empty interface means having no field or methods.
* Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

**Java Comparator Interface**

* **Comparator interface in Java**is used to **sort the objects** of user-defined classes.
* **java.util** package
* *public int compare(Object obj1, Object obj2):*
* It will return a negative integer if **obj1 < obj2**.
* It will return 0 if both objects are **equal**.
* It will return a positive integer if **obj1 > obj2**.
* The **sort()** method of the **Collections class** is used to sort the elements of a List by the given comparator.
* Internally the**sort()** method does call **Compare method** of the classes it is sorting.

// Using Comparator Interface

import java.util.\*;

// Define the Student class

class Student {

int rollno;

String name;

// Constructor

Student(int rollno, String name) {

this.rollno = rollno;

this.name = name;

}

// Method to print Student

// details in main()

@Override

public String toString() {

return rollno + ": " + name;

}

}

// Helper class implementing Comparator interface

class SortbyRoll implements Comparator<Student>

{

// Compare by roll number in ascending order

public int compare(Student a, Student b) {

return a.rollno - b.rollno;

}

}

// Driver Class

public class Geeks

{

public static void main(String[] args)

{

// List of Students

List<Student> students = new ArrayList<>();

// Add Elements in List

students.add(new Student(111, "Mayank"));

students.add(new Student(131, "Anshul"));

students.add(new Student(121, "Solanki"));

students.add(new Student(101, "Aggarwal"));

// Sort students by roll number

// using SortbyRoll comparator

Collections.sort(students, new SortbyRoll());

System.out.println("Sorted by Roll Number ");

// Iterating over entries to print them

for (int i = 0; i < students.size(); i++)

System.out.println(students.get(i));

}

}

**Comparator vs Comparable**

| **Feature** | **Comparator** | **Comparable** |
| --- | --- | --- |
| Sorting Logic Location | Defined externally | Defined within the class (Internally) |
| Multiple Sorting Orders | Supported | Not supported |
| Interface Methods | compare() | compareTo() |
| Functional Interface | Yes | No |
| Usage | Flexible and reusable | Simple and tightly coupled |

* **Java Functional Interfaces**
* A functional interface in Java is an interface that contains only one abstract method.
* Functional interfaces can have multiple default or static methods, but only one abstract method.
* [Runnable](https://www.geeksforgeeks.org/runnable-interface-in-java/), [ActionListener](https://www.geeksforgeeks.org/java-actionlistener-in-awt/), and [Comparator](https://www.geeksforgeeks.org/comparator-interface-java/) are common examples of Java functional interfaces.
* From Java 8 onwards, lambda expressions and method references can be used to represent the instance of a functional interface.
* A**f**unctional interface can also extend another functional interface.
* @FunctionalInterface annotation is used to ensure that the functional interface cannot have more than one abstract method. In case more than one abstract methods are present, the compiler flags an “**Unexpected @FunctionalInterface annotation**” message. However, it is not mandatory to use this annotation.
* @FunctionalInterface annotation is optional but it is a good practice to use. It helps catching the error in early stage by making sure that the interface has only one abstract method.

// Define a functional interface

@FunctionalInterface

interface Square {

int calculate(int x);

}

class Geeks {

public static void main(String args[]) {

int a = 5;

// lambda expression to

// define the calculate method

Square s = (int x) -> x \* x;

// parameter passed and return type must be

// same as defined in the prototype

int ans = s.calculate(a);

System.out.println(ans);

}

}

* **Types of Functional Interfaces in Java**
* Consumer - The [consumer interface](https://www.geeksforgeeks.org/java-8-consumer-interface-in-java-with-examples/) of the functional interface is the one that accepts only one argument or a gentrified argument. The consumer interface has no return value.
* The [Predicate interface](https://www.geeksforgeeks.org/java-8-predicate-with-examples/)represents a boolean-valued function of one argument. It is commonly used for filtering operations in streams.
* A [function](https://www.geeksforgeeks.org/function-interface-in-java-with-examples/) is a type of functional interface in Java that receives only a single argument and returns a value after the required processing. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long.
* The [Supplier](https://www.geeksforgeeks.org/supplier-interface-in-java-with-examples/)functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output.The different extensions of the Supplier functional interface hold many other suppliers functions like [BooleanSupplier](https://www.geeksforgeeks.org/java-8-booleansupplier-interface-with-examples/" \t "_blank), [DoubleSupplier](https://www.geeksforgeeks.org/java-8-doublesupplier-interface-with-examples/" \t "_blank), [LongSupplier](https://www.geeksforgeeks.org/java-8-longsupplier-interface-with-examples/" \t "_blank), and [IntSupplier](https://www.geeksforgeeks.org/java-8-intsupplier-interface-with-examples/" \t "_blank). The return type of all these further specializations is their corresponding primitives only.