

Object Oriented Programming: Solutions & Guide

Topics: equals(), hashCode(), toString(), Immutability, Value Objects, and Comparable

Solution 1: Printing an Object (toString)

Key Takeaway: By default, `Object.toString()` produces `getClass().getName() + '@' + Integer.toHexString(hashCode())`. Overriding it gives human-readable state.

```
public class Book {
    private String title;
    private String author;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    @Override
    public String toString() {
        return "Book: " + title + " by " + author;
    }

    public static void main(String[] args) {
        Book b = new Book("The Alchemist", "Paulo Coelho");
        // System.out.println internally invokes b.toString()
        System.out.println(b);
    }
}
```

EXPECTED OUTPUT:

```
Book: The Alchemist by Paulo Coelho
```

Solution 2: Logical vs. Reference Equality (equals)

Key Takeaway: The == operator compares reference addresses. Overriding equals (Object o) defines logical equivalence between distinct heap objects.

```
public class Student {
    private int rollNo;
    private String name;

    public Student(int rollNo, String name) {
        this.rollNo = rollNo;
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true; // 1. Reflexivity / same reference
        if (o == null || getClass() != o.getClass()) return false; // 2. Null & type check
        Student other = (Student) o; // 3. Safe cast
        return this.rollNo == other.rollNo; // 4. State comparison
    }

    public static void main(String[] args) {
        Student s1 = new Student(101, "Aman");
        Student s2 = new Student(101, "Aman");
        System.out.println("s1 == s2: " + (s1 == s2)); // false (distinct objects)
        System.out.println("s1.equals(s2): " + s1.equals(s2)); // true (same rollNo)
    }
}
```

EXPECTED OUTPUT:

```
s1 == s2: false
s1.equals(s2): true
```

Solution 3: Pairing hashCode with equals

Key Takeaway: The Contract: If `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` MUST be true. Otherwise, hash collections (`HashSet/HashMap`) will store duplicate items.

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

public class StudentWithHash {
    private int rollNo;
    private String name;

    public StudentWithHash(int rollNo, String name) {
        this.rollNo = rollNo;
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        StudentWithHash that = (StudentWithHash) o;
        return this.rollNo == that.rollNo;
    }

    @Override
    public int hashCode() {
        return Objects.hash(rollNo);
    }

    public static void main(String[] args) {
        StudentWithHash s1 = new StudentWithHash(101, "Aman");
        StudentWithHash s2 = new StudentWithHash(101, "Aman");

        HashSet<StudentWithHash> set = new HashSet<>();
        set.add(s1);
        set.add(s2);

        System.out.println("Set size: " + set.size()); // Size is 1 because hash and equals match
    }
}
```

EXPECTED OUTPUT:

```
Set size: 1
```

Solution 4: Coordinate Comparison (Point)

Key Takeaway: When an entity's value is defined across multiple attributes, both `equals()` and `hashCode()` must incorporate all contributing attributes.

```
import java.util.Objects;

public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Point point = (Point) o;
        return this.x == point.x && this.y == point.y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    public static void main(String[] args) {
        Point p1 = new Point(2, 3);
        Point p2 = new Point(2, 3);
        Point p3 = new Point(4, 5);

        System.out.println("p1.equals(p2): " + p1.equals(p2)); // true
        System.out.println("p1.equals(p3): " + p1.equals(p3)); // false
        System.out.println("Hash p1 == p2: " + (p1.hashCode() == p2.hashCode())); // true
    }
}
```

EXPECTED OUTPUT:

```
p1.equals(p2): true
p1.equals(p3): false
Hash p1 == p2: true
```

Solution 5: A Basic Immutable Class (Circle)

Key Takeaway: An immutable class has: (1) final class, (2) private final fields, (3) constructor initialization, (4) no mutator (setter) methods.

```
public final class Circle {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public static void main(String[] args) {
        Circle c = new Circle(5.0);
        System.out.println("Radius: " + c.getRadius());
        System.out.printf("Area: %.2f\n", c.getArea());
    }
}
```

EXPECTED OUTPUT:

```
Radius: 5.0
Area: 78.54
```

Solution 6: Value Object with Operations (Time)

Key Takeaway: Operations on Value Objects never mutate the existing instance; they calculate the new state and return a brand-new instance (like `java.time.LocalDate` or `String`).

```
public final class Time {
    private final int hours;
    private final int minutes;

    public Time(int hours, int minutes) {
        this.hours = hours;
        this.minutes = minutes;
    }

    public Time addMinutes(int mins) {
        int totalMinutes = (this.hours * 60 + this.minutes) + mins;
        int newHours = (totalMinutes / 60) % 24;
        int newMins = totalMinutes % 60;
        return new Time(newHours, newMins); // Returns NEW instance!
    }

    @Override
    public String toString() {
        return String.format("%02d:%02d", hours, minutes);
    }

    public static void main(String[] args) {
        Time t1 = new Time(10, 30);
        Time t2 = t1.addMinutes(45); // 10:30 + 45 min = 11:15

        System.out.println("Original Time (t1): " + t1); // 10:30 (Unchanged!)
        System.out.println("New Time (t2):      " + t2); // 11:15
    }
}
```

EXPECTED OUTPUT:

```
Original Time (t1): 10:30
New Time (t2): 11:15
```

Solution 7: Defensive Copying with Arrays (GradeReport)

Key Takeaway: Making a reference field `final` only freezes the reference pointer, NOT the array contents. Defensive copying via `.clone()` prevents external mutation.

```
import java.util.Arrays;

public final class GradeReport {
    private final int[] marks;

    public GradeReport(int[] marks) {
        // Defensive copy on ingestion
        this.marks = marks.clone();
    }

    public int[] getMarks() {
        // Defensive copy on publication
        return this.marks.clone();
    }

    public static void main(String[] args) {
        int[] externalScores = {85, 90, 95};
        GradeReport report = new GradeReport(externalScores);

        // Attempting external tampering
        externalScores[0] = 0;
        report.getMarks()[1] = 0;

        // Internal state is safe!
        System.out.println("Report marks: " + Arrays.toString(report.getMarks()));
    }
}
```

EXPECTED OUTPUT:

```
Report marks: [85, 90, 95]
```

Solution 8: Natural Ordering (Player)

Key Takeaway: `Comparable<T>` provides natural sorting via `compareTo(T other)`: returns negative if `this < other`, 0 if equal, positive if `this > other`.

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

public class Player implements Comparable<Player> {
    private String name;
    private int score;

    public Player(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public int compareTo(Player other) {
        // Ascending order: lowest score first
        return Integer.compare(this.score, other.score);
    }

    @Override
    public String toString() {
        return name + " (" + score + ")";
    }

    public static void main(String[] args) {
        List<Player> players = new ArrayList<>();
        players.add(new Player("Rahul", 45));
        players.add(new Player("Sneha", 88));
        players.add(new Player("Amit", 62));
        players.add(new Player("Pooja", 15));

        Collections.sort(players);
        System.out.println("Sorted Players: " + players);
    }
}
```

EXPECTED OUTPUT:

```
Sorted Players: [Pooja (15), Rahul (45), Amit (62), Sneha (88)]
```

Solution 9: Alphabetical Sorting (City)

Key Takeaway: To order objects lexicographically, delegate directly to `String.compareTo()` on the appropriate field.

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

public class City implements Comparable<City> {
    private String name;
    private int population;

    public City(String name, int population) {
        this.name = name;
        this.population = population;
    }

    @Override
    public int compareTo(City other) {
        // Delegate to String's compareTo
        return this.name.compareTo(other.name);
    }

    @Override
    public String toString() {
        return name + " [" + population + "]";
    }

    public static void main(String[] args) {
        List<City> cities = new ArrayList<>();
        cities.add(new City("Mumbai", 12442373));
        cities.add(new City("Bengaluru", 8443675));
        cities.add(new City("Delhi", 11007835));

        Collections.sort(cities);
        System.out.println("Sorted Cities: " + cities);
    }
}
```

EXPECTED OUTPUT:

```
Sorted Cities: [Bengaluru [8443675], Delhi [11007835], Mumbai [12442373]]
```

Solution 10: Reverse / Descending Sort (Product)

Key Takeaway: Inverting the comparison operands (comparing other to this) produces descending order. Always use `Double.compare()` for floating-point safety.

```
import java.util.Arrays;

public class Product implements Comparable<Product> {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public int compareTo(Product other) {
        // Descending order: compare other.price to this.price
        return Double.compare(other.price, this.price);
    }

    @Override
    public String toString() {
        return name + " ($" + price + ")";
    }

    public static void main(String[] args) {
        Product[] inventory = {
            new Product("Mouse", 25.50),
            new Product("Laptop", 899.99),
            new Product("Headphones", 79.99)
        };

        Arrays.sort(inventory);
        System.out.println("Products (High to Low): " + Arrays.toString(inventory));
    }
}
```

EXPECTED OUTPUT:

```
Products (High to Low): [Laptop ($899.99), Headphones ($79.99), Mouse ($25.5)]
```

Solution 11: Complete Value Object (Car)

Key Takeaway: Synthesizing all concepts: immutability (final fields), identity equivalence via `registrationNo` (`equals/hashCode`), natural ordering via `modelYear` (`Comparable`), and readable state (`toString`).

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;

public final class Car implements Comparable<Car> {
    private final String registrationNo;
    private final int modelYear;

    public Car(String registrationNo, int modelYear) {
        this.registrationNo = registrationNo;
        this.modelYear = modelYear;
    }

    public String getRegistrationNo() { return registrationNo; }
    public int getModelYear() { return modelYear; }

    @Override
    public String toString() {
        return "Car [Reg: " + registrationNo + ", Year: " + modelYear + "]";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Car car = (Car) o;
        return Objects.equals(registrationNo, car.registrationNo);
    }

    @Override
    public int hashCode() {
        return Objects.hash(registrationNo);
    }

    @Override
    public int compareTo(Car other) {
        // Chronological sort by modelYear
        return Integer.compare(this.modelYear, other.modelYear);
    }

    public static void main(String[] args) {
        List<Car> garage = new ArrayList<>();
        garage.add(new Car("DL-01-AB-1234", 2021));
        garage.add(new Car("MH-02-CD-5678", 2018));
        garage.add(new Car("KA-03-EF-9012", 2024));

        Collections.sort(garage);
        System.out.println("Sorted by Year:");
        for (Car c : garage) {
            System.out.println("  " + c);
        }

        // Test HashSet uniqueness based on registrationNo
    }
}
```

```
HashSet<Car> registry = new HashSet<>();  
registry.add(new Car("DL-01-AB-1234", 2021));  
registry.add(new Car("DL-01-AB-1234", 2023)); // Same reg number, different year!  
System.out.println("Registry size (expected 1): " + registry.size());  
}  
}
```

EXPECTED OUTPUT:

```
Sorted by Year:  
Car [Reg: MH-02-CD-5678, Year: 2018]  
Car [Reg: DL-01-AB-1234, Year: 2021]  
Car [Reg: KA-03-EF-9012, Year: 2024]  
Registry size (expected 1): 1
```