

Object Oriented Programming Lab Worksheet

Topics: `equals()` , `hashCode()` , `toString()` , Immutability, Value Objects, `Comparable<T>`

PART I: OBJECT CONTRACTS (EQUALS, HASHCODE, TOSTRING)

Question 1 Printing an Object via `toString()`

Create a class `Book` with fields `String title` and `String author` .

- Instantiate a `Book` object and print it directly using `System.out.println()` before overriding `toString()` . Note the default output (Class name followed by memory hash).
- Override `toString()` to return a clean, user-friendly string format: `"Book: <title> by <author>"` .
- Re-run the program in `main()` and observe the customized, readable output.

Question 2 Logical vs. Reference Equality (`equals()`)

Create a `Student` class with fields `int rollNo` and `String name` .

- Create two distinct instances: `s1 = new Student(101, "Aman")` and `s2 = new Student(101, "Aman")` .
- Compare them using `s1 == s2` and `s1.equals(s2)` before overriding `equals()` . Note why both evaluate to `false` .
- Override `equals(Object o)` to return `true` if two students have identical `rollNo` . Test `s1.equals(s2)` again to verify logical equality.

Hint: Remember to check for `this == o` , nullity, and type matching using `instanceof` before type-casting.

Question 3 Pairing `hashCode()` with `equals()`

In the same `Student` class from Question 2, address the contract between equals and hash codes.

- Without overriding `hashCode()` , add both `s1` and `s2` to a `HashSet<Student>` and check `set.size()` . Why does it contain 2 items?
- Now override `hashCode()` using `Objects.hash(rollNo)` (or `Integer.hashCode(rollNo)`).
- Add both instances to a fresh `HashSet<Student>` again. Verify that the size is now `1` as expected.

Question 4 Multi-Field Equality with `Point`

Create a class `Point` representing a 2D coordinate with fields `int x` and `int y` .

- Implement standard constructor, getters, `equals(Object o)` , and `hashCode()` based on **both** `x` and `y` .
- Create points `p1(2, 3)` , `p2(2, 3)` , and `p3(4, 5)` .
- In `main()` , demonstrate that `p1.equals(p2)` is `true` , `p1.equals(p3)` is `false` , and `p1.hashCode() == p2.hashCode()` .

PART II: IMMUTABILITY & VALUE OBJECTS

Question 5 A Basic Immutable Class (`Circle`)

Create a strictly immutable class `Circle` representing a geometric circle.

- Declare the class as `final` to prevent subclass alteration.
- Declare a private final field `final double radius` initialized through a constructor.
- Provide a public getter `getRadius()`, but **no setter methods**.
- Add a method `double getArea()` that computes $\pi \times r^2$.

Question 6 Value Object with State Evolution (`Time`)

Create an immutable class `Time` representing a time of day with `int hours` and `int minutes`.

- Make fields `private final` and validate inputs in the constructor (hours 0–23, mins 0–59).
- Implement a method `Time addMinutes(int mins)`.
- Ensure the method does **not** modify the existing instance; instead, it must calculate new values and return a **new** `Time` object.
- In `main()`, demonstrate that the original `Time` object remains unchanged after calling `addMinutes()`.

Question 7 Defensive Copying with Arrays (`GradeReport`)

Prevent encapsulation leaks when handling mutable reference types like arrays.

- Create an immutable class `GradeReport` with a field `private final int[] marks`.
- In the constructor, store a defensive clone: `this.marks = marks.clone();` instead of the direct reference.
- In the getter `getMarks()`, return `marks.clone()` instead of the internal array reference.
- In `main()`, demonstrate that modifying the input array outside the class does **not** affect `GradeReport`.

PART III: OBJECT ORDERING WITH COMPARABLE

Question 8 Natural Numeric Ordering (`Player`)

Create a `Player` class with fields `String name` and `int score`.

- Implement the `Comparable<Player>` interface.
- Override `compareTo(Player other)` to sort players by `score` in ascending order (lowest to highest).
- Store 4 players in an `ArrayList<Player>` and sort them using `Collections.sort()`. Print the sorted list.

Question 9 Alphabetical Ordering (`City`)

Create a `City` class with fields `String name` and `int population`.

- Implement `Comparable<City>` to define natural ordering alphabetically by `name`.
- In `compareTo(City other)`, delegate to Java's built-in `String.compareTo()`.
- Create a list with 3 cities, sort them with `Collections.sort()`, and print the sorted order.

Question 10 Reverse / Descending Ordering (`Product`)

Create a `Product` class with fields `String name` and `double price`.

- Implement `Comparable<Product>` to sort products by `price` in **descending** order (highest price first).
- Use `Double.compare(other.price, this.price)` to avoid floating-point subtraction errors.
- Create an array of 3 products and sort them using `Arrays.sort()`. Print the sorted list.

PART IV: COMPREHENSIVE INTEGRATION

Question 11 Complete Value Object: `car`

Synthesize all four core concepts into an immutable value object:

- Design an immutable class `Car` with `final String registrationNo` and `final int modelYear`.
- Override `toString()` returning `"Car [Reg: ..., Year: ...]"`.
- Override `equals()` and `hashCode()` based **strictly on `registrationNo`** (identity uniqueness).
- Implement `Comparable<Car>` to sort cars chronologically by `modelYear` in ascending order.
- In `main()`, populate an `ArrayList<Car>` with 3 unsorted cars, call `Collections.sort()`, and print them. Also verify that two car objects with identical registration numbers are recognized as duplicates by a `HashSet<Car>`.