“Order Management + Fraud/Support Logs”

You’re building an e-commerce backend. These objects appear everywhere:

- `Customer` (identity: email)
- `Address` (value object: all fields matter)
- `Order` (identity: `orderId`)
    - `PhysicalOrder` extends `Order` (has shipping)
    - `DigitalOrder` extends `Order` (has download link)

Real problems you must solve

1. **Logging / Debugging**
    - Support engineers read logs like: “refund failed for order …”
    - Default `toString()` (`Order@4ab2…`) is useless.
    - But you **must not log sensitive data** (emails partly masked, never print token/link fully).
2. **Deduplication and Lookups**
    - Orders are stored in `HashSet` / used as `HashMap` keys.
    - Two `Order` objects with the same `orderId` but loaded from different services should be considered **the same logical order**.
    - `equals()` must be correct and consistent, otherwise:
        - duplicates appear
        - lookups fail
        - `remove()` doesn’t work
3. **Inheritance trap**

- If you use `instanceof` casually in `equals()`, you can break **symmetry** between `Order` and `PhysicalOrder`.

Solution Design Choices (the “why”)

Equality strategy (what defines “same object”)

- `Order`: identity-based equality → **only `orderId`**
- `Customer`: identity-based equality → **email normalized** (lowercase)
- `Address`: value-based equality → **all fields**
- `LineItem`: value-based equality → **sku + quantity + unitPrice**

Inheritance strategy

For `Order.equals()` use **`getClass()`** (strict type match) unless you intentionally want cross-subclass equality.

Why? To avoid symmetry issues like:

- `order.equals(physicalOrder)` true but `physicalOrder.equals(order)` false.

Code (Complete, Practical)

1) Value object: `Address` (value equality)

```
import java.util.Objects;

final class Address {
    private final String line1;
    private final String city;
    private final String state;
    private final String country;
    private final String pincode;

    public Address(String line1, String city, String state, String country,
String pincode) {
        this.line1 = Objects.requireNonNull(line1);
        this.city = Objects.requireNonNull(city);
        this.state = Objects.requireNonNull(state);
        this.country = Objects.requireNonNull(country);
        this.pincode = Objects.requireNonNull(pincode);
```

```
    }

    @Override
    public String toString() {
        // Avoid over-verbosity; enough for logs
        return "Address{line1='" + line1 + "', city='" + city + "',
state='" + state +
                "', country='" + country + "', pincode='" + pincode + "'}";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Address address = (Address) o;
        return line1.equals(address.line1)
                && city.equals(address.city)
                && state.equals(address.state)
                && country.equals(address.country)
                && pincode.equals(address.pincode);
    }

    @Override
    public int hashCode() {
        return Objects.hash(line1, city, state, country, pincode);
    }
}
```

Why this is “correct”:

- Address is a *value*: if all fields same, it’s the same address.
- Immutable fields ⇒ equals consistency stays stable.

---

2) Identity object: `Customer` (identity equality)

```
import java.util.Locale;
import java.util.Objects;

final class Customer {
    private final String customerId;   // internal id
    private final String email;        // identity in business
    private final String fullName;
```

```
    public Customer(String customerId, String email, String fullName) {
        this.customerId = Objects.requireNonNull(customerId);
        this.email = normalizeEmail(email);
        this.fullName = Objects.requireNonNull(fullName);
    }

    private static String normalizeEmail(String email) {
        Objects.requireNonNull(email);
        return email.trim().toLowerCase(Locale.ROOT);
    }

    private static String maskEmail(String email) {
        // e.g., suhani.shah@gmail.com -> s***@gmail.com
        int at = email.indexOf('@');
        if (at <= 1) return "***";
        return email.charAt(0) + "***" + email.substring(at);
    }

    @Override
    public String toString() {
        // Mask PII in logs
        return "Customer{customerId='" + customerId + "', email='" +
maskEmail(email) +
                "', fullName='" + fullName + "'}";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Customer customer = (Customer) o;
        // identity: email (normalized)
        return email.equals(customer.email);
    }

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

Why this is realistic:

- Systems often treat email as customer identity across microservices.

- Masking avoids leaking PII in logs.

---

3) `Order` base class (identity equality + safe `toString()`)

```
import java.time.Instant;
import java.util.List;
import java.util.Objects;

abstract class Order {
    protected final String orderId;          // identity
    protected final Customer customer;
    protected final List<LineItem> items;
    protected final Instant createdAt;

    protected Order(String orderId, Customer customer, List<LineItem>
items) {
        this.orderId = Objects.requireNonNull(orderId);
        this.customer = Objects.requireNonNull(customer);
        this.items = List.copyOf(Objects.requireNonNull(items));
        this.createdAt = Instant.now();
    }

    public final String getOrderId() { return orderId; }

    public final double getBaseAmount() {
        return items.stream().mapToDouble(LineItem::lineTotal).sum();
    }

    /** Keep base toString short; subclasses append their own fields. */
    protected String baseToString() {
        return "orderId='" + orderId + "', customer=" + customer + ",
itemCount=" + items.size() +
               ", baseAmount=" + getBaseAmount() + ", createdAt=" +
createdAt;
    }

    @Override
    public String toString() {
        // base representation (subclasses can override and call
baseToString)
        return getClass().getSimpleName() + "{" + baseToString() + "}";
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        // IMPORTANT: strict class check avoids symmetry issues with
subclasses
        if (o == null || getClass() != o.getClass()) return false;
        Order other = (Order) o;
        // identity equality: only orderId
        return orderId.equals(other.orderId);
    }

    @Override
    public int hashCode() {
        // must match equals: same orderId => same hashCode
        return Objects.hash(orderId);
    }
}
```

Key points interviewers want:

- `equals()` uses **getClass** to avoid inheritance issues.
- Equality is based on **identity** (`orderId`), not mutable fields.
- `hashCode()` aligned with equals.
- `toString()` is stable and helpful.

---

4) Subclasses with extra fields (and safe `toString()`)

PhysicalOrder

```
import java.util.Objects;

final class PhysicalOrder extends Order {
    private final Address shippingAddress;
    private final String trackingCode;

    public PhysicalOrder(String orderId, Customer customer, List<LineItem>
items,
                         Address shippingAddress, String trackingCode) {
        super(orderId, customer, items);
        this.shippingAddress = Objects.requireNonNull(shippingAddress);
```

```
        this.trackingCode = Objects.requireNonNull(trackingCode);
    }

    private static String maskTracking(String code) {
        // show only last 4 chars
        if (code.length() <= 4) return "****";
        return "****" + code.substring(code.length() - 4);
    }

    @Override
    public String toString() {
        return "PhysicalOrder{" + baseToString() +
                ", shippingAddress=" + shippingAddress +
                ", trackingCode='" + maskTracking(trackingCode) + "'" +
                "}";
    }

    // equals/hashCode inherited from Order (strict class match),
    // so PhysicalOrder equality is still orderId-based and safe.
}
```

DigitalOrder

```
import java.util.Objects;

final class DigitalOrder extends Order {
    private final String downloadToken;

    public DigitalOrder(String orderId, Customer customer, List<LineItem>
items, String downloadToken) {
        super(orderId, customer, items);
        this.downloadToken = Objects.requireNonNull(downloadToken);
    }

    private static String maskToken(String token) {
        // never print full token
        if (token.length() <= 6) return "***";
        return token.substring(0, 3) + "***" +
token.substring(token.length() - 3);
    }

    @Override
    public String toString() {
        return "DigitalOrder{" + baseToString() +
                ", downloadToken='" + maskToken(downloadToken) + "'" +
                "}";
```

```
    }
}
```

---

5) LineItem (value equality)

```
import java.util.Objects;

final class LineItem {
    private final String sku;
    private final int qty;
    private final double unitPrice;

    public LineItem(String sku, int qty, double unitPrice) {
        if (qty <= 0) throw new IllegalArgumentException("qty must be >
0");
        if (unitPrice < 0) throw new IllegalArgumentException("unitPrice
must be >= 0");
        this.sku = Objects.requireNonNull(sku);
        this.qty = qty;
        this.unitPrice = unitPrice;
    }

    public double lineTotal() {
        return qty * unitPrice;
    }

    @Override
    public String toString() {
        return "LineItem{sku='" + sku + "', qty=" + qty + ", unitPrice=" +
unitPrice + "}";
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        LineItem that = (LineItem) o;
        return qty == that.qty
                && Double.compare(unitPrice, that.unitPrice) == 0
                && sku.equals(that.sku);
    }

    @Override
    public int hashCode() {
```

```
        return Objects.hash(sku, qty, unitPrice);
    }
}
```

Demonstration: Why this solution matters

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        Customer c1 = new Customer("C-101", "Suhani.Shah@Gmail.com",
"Suhani Shah");
        Customer c2 = new Customer("C-999", "suhani.shah@gmail.com", "S.
Shah");

        System.out.println(c1); // masked email
        System.out.println("Customers equal? " + c1.equals(c2)); // true
(normalized identity)

        List<LineItem> items = List.of(
                new LineItem("SKU-RED-01", 2, 499.0),
                new LineItem("SKU-BLUE-02", 1, 999.0)
        );

        Address addr = new Address("221B Baker St", "London", "London",
"UK", "NW16XE");

        Order o1 = new PhysicalOrder("ORD-1", c1, items, addr,
"TRACK-ABCDEF1234");
        Order o2 = new PhysicalOrder("ORD-1", c1, items, addr,
"TRACK-XYZ9990000"); // same orderId

        System.out.println(o1); // readable, masked tracking
        System.out.println("Orders equal? " + o1.equals(o2)); // true (same
orderId)

        Set<Order> set = new HashSet<>();
        set.add(o1);
        set.add(o2);
        System.out.println("Set size: " + set.size()); // 1 (dedup works)
    }
}
```

What you get:

- Useful logs (`toString`)
- Correct equality (`equals`)
- Correct hashing behavior in sets/maps (`hashCode`)
- No sensitive leaks
- No inheritance symmetry bugs

Common interview traps this solution avoids

1. **Overriding `equals()` but not `hashCode()`** → breaks `HashSet/HashMap`
2. Using mutable fields (like `status`, `balance`) in `equals()` → violates consistency
3. Printing `this` inside `toString()` → infinite recursion
4. Using `instanceof` incorrectly with inheritance → breaks symmetry/transitivity
5. Logging secrets (tokens, card numbers, full emails)