E-Commerce Order Fulfillment System (Inheritance + Polymorphism)

Goal

You will implement a **mini order processing engine** that supports multiple order types using **OOP principles**, especially:

- Inheritance (multi-level)
- Method overriding
- Abstract classes + abstract methods
- Interfaces (capabilities)
- Polymorphism in processing
- Clean separation of responsibilities

Problem Statement 1: Core Order Hierarchy

Business Scenario

A commerce platform processes different types of orders. Every order has:

- Order ID
- Customer email
- List of items
- Base amount (sum of item prices)
- Discount (optional)
- Status tracking

But different order types have different rules for:

- Tax calculation
- Extra fees (shipping/packaging/customs)
- Fulfillment behavior (digital delivery vs shipping)

Requirement

Create an abstract base class `Order` that defines the **core workflow** for checkout and enforces subtype-specific behavior.

`Order` must:

1. Store common data:
    - `orderId` (String)
    - `customerEmail` (String)
    - `items` (List<String> or String[])
    - `baseAmount` (double)
    - `discount` (double)
    - `status` (enum)
2. Provide:
    - `applyDiscount(double amount)` (validations required)
    - `summary()` that returns a readable one-line summary
    - `validate()` to ensure:
        - items not empty
        - baseAmount > 0
        - email contains `@`
3. Enforce abstract behaviors (must be abstract methods):
    - `calculateTax()`

    - `calculateExtraFees()`
    - `postPaymentFulfillment()`
4. Provide a **final** template method:
    - `final double checkout()`
    - This method must do:
        - validate()
        - compute: `total = baseAmount + tax + extraFees - discount`
        - prevent negative totals (min 0)
        - set status to PAID
        - call `postPaymentFulfillment()`
        - return final total

Status (enum)

Create `enum OrderStatus { PLACED, PAID, PACKED, SHIPPED, DELIVERED }`

Problem Statement 2: Implement Order Types (Inheritance)

Implement these classes:

A) `DigitalOrder extends Order`

- Tax: 5% of baseAmount
- Extra fees: 0
- Fulfillment: Immediately mark as `DELIVERED` and print:
    - `"Digital delivery link emailed to <email>"`

B) `PhysicalOrder extends Order` (intermediate parent)

Add:

- `shippingAddress` (String)
- `courierPartner` (String)

Rules:

- `validate()` must also ensure shippingAddress is non-empty
- Tax: 12%
- Extra fees = shipping(50) + packaging(20) = 70
- Fulfillment:
    - set status PACKED, print `"Order packed at warehouse. Courier: <courierPartner>"`
    - set status SHIPPED, print `"Order shipped to: <address>"`

C) `FragileOrder extends PhysicalOrder`

Rules:

- courierPartner must default to `"FragileExpress"`
- Extra fees: PhysicalOrder extra fees + fragile handling (80)
- Fulfillment prints an additional message:
    - `"Fragile packing done with foam + shock-proof box."`

D) `InternationalOrder extends PhysicalOrder`

Add:

- `destinationCountry` (String)
  Rules:
- courierPartner must default to `"GlobalShip"`
- Tax: 8%
- Extra fees: PhysicalOrder extra fees + customs duty (15% of baseAmount) + international shipping (300)
- Fulfillment prints:
  - `"Customs invoice generated for <destinationCountry>"`

E) `SubscriptionOrder extends PhysicalOrder`

Add:

- `months` (int)
  Rules:
- courierPartner default: `"MonthlyBoxCourier"`
- Extra fees: base fees (30 + 15) + logisticsMgmtFee (months * 10)
- Fulfillment prints:
  - `"Subscription schedule created for <months> months."`

Problem Statement 3: Order Processing Engine (Polymorphism)

Requirement

Create a class `OrderProcessor` with:

```
public double process(Order order)
```

It must:

1. Print `order.summary()`

2. Call `checkout()`
3. Print final payable amount
4. Print final status
5. Return payable amount

Key OOP expectation

`OrderProcessor` must **not** use `instanceof` or if-else on order types for tax/fees/fulfillment.
It must rely on **polymorphism**.

Problem Statement 4: Interfaces (Capabilities)

Add interfaces to make this more realistic and modular.

A) `Trackable`

```
interface Trackable {
    String getTrackingCode();
}
```

- Only physical orders are trackable
- Add a tracking code generation rule:
    - tracking code format: `<COURIER>-<ORDERID>-<RANDOM4DIGITS>`
    - Example: `GlobalShip-I-303-4821`

`PhysicalOrder` (and thus all its children) should implement Trackable.

B) `Refundable`

```
interface Refundable {
    double refund(double amount);
}
```

Rules:

- DigitalOrder: refund allowed only if status is DELIVERED within “instant” scenario (for assignment: allow refund always but deduct 10% platform fee)
- PhysicalOrder: refund allowed only if NOT DELIVERED; if shipped, refund deducts 5% restocking fee

You must implement refund logic via overriding where needed.

Problem Statement 5: Error Handling and Validations (Production Behavior)

Mandatory validations

- `applyDiscount()` must reject negative discount
- Total should never go below 0
- Subscription months must be >= 1
- Destination country must be non-empty for international
- Throw meaningful exceptions (IllegalArgumentException / IllegalStateException)

Deliverables (What students must submit)

1. Source code (Java)
2. `main()` demo program that:
    - Creates at least:
        - 1 DigitalOrder
        - 1 FragileOrder
        - 1 InternationalOrder

        - 1 SubscriptionOrder
    - Applies discounts to at least 2 orders
    - Processes all orders through `OrderProcessor`
    - Demonstrates tracking code for physical orders
    - Demonstrates refund for at least 2 cases (valid + invalid)

Acceptance Criteria (Checklist)

✅ Uses `abstract class Order` with `final checkout()`
✅ Uses overridden methods for tax/fees/fulfillment
✅ Uses multi-level inheritance (`Order → PhysicalOrder → SpecialOrder`)
✅ Uses interfaces `Trackable` and `Refundable` meaningfully
✅ No `instanceof` based decision logic in `OrderProcessor`
✅ Validations + exceptions implemented
✅ Output logs show status transitions and totals clearly

Testing Expectations

1. Discount cannot be negative
2. DigitalOrder total = base + 5% tax - discount
3. FragileOrder extra fees include +80 fragile
4. InternationalOrder extra fees include customs + 300
5. SubscriptionOrder extra fees depend on months
6. Refund rules behave correctly (allowed vs blocked)

Optional Extension

1. Add `CODOrder` vs `PrepaidOrder` as a second inheritance dimension using interfaces

2. Add `PaymentProcessor` strategy interface
3. Add `OrderCancellation` rules:
    - cannot cancel after DELIVERED
4. Persist orders in a simple in-memory repository

Git Instructions: Separate Branch per Problem Statement (Incremental)

Goal

You will implement each problem statement in isolation on its own branch, then merge into `main` (or create PRs). This mirrors real-world incremental delivery.

---

0) One-time setup

From your project folder:

```
git status
git checkout main
git pull
```

Create a clean baseline commit if you haven't:

```
git add .
git commit -m "chore: initial project scaffold"
```

Branch naming convention

Use a consistent pattern:

- `ps1-core-order`
- `ps2-order-types`
- `ps3-processor`
- `ps4-interfaces`
- `ps5-validations`
- `ps6-tests`

- `ps7-demo-output`

Problem Statement 1 Branch: Core Order + Status enum

```
git checkout -b ps1-core-order
```

Implement:

- `Order` (abstract)
- `OrderStatus` enum
- `checkout()` template method (final)
- base validation + discount rules

Then commit:

```
git add .
git commit -m "feat: add core Order abstraction and OrderStatus"
```

Merge back:

```
git checkout main
git merge ps1-core-order
```

Problem Statement 2 Branch: Implement all Order types

```
git checkout -b ps2-order-types
```

Implement:

- `DigitalOrder`

- `PhysicalOrder`
- `FragileOrder`
- `InternationalOrder`
- `SubscriptionOrder`

Commit + merge:

```
git add .
git commit -m "feat: implement order types with tax/fee/fulfillment
overrides"
git checkout main
git merge ps2-order-types
```

Problem Statement 3 Branch: OrderProcessor (Polymorphism)

```
git checkout -b ps3-processor
```

Implement:

- `OrderProcessor.process(Order order)`
- Ensure no `instanceof` logic for tax/fees/fulfillment

Commit + merge:

```
git add .
git commit -m "feat: add OrderProcessor with polymorphic processing"
git checkout main
git merge ps3-processor
```

Problem Statement 4 Branch: Interfaces Trackable + Refundable

```
git checkout -b ps4-interfaces
```

Implement:

- `Trackable` + tracking code generation for `PhysicalOrder`
- `Refundable` rules + overrides where needed

Commit + merge:

```
git add .
git commit -m "feat: add Trackable and Refundable capabilities"
git checkout main
git merge ps4-interfaces
```

Problem Statement 5 Branch: Strong validations + exceptions

```
git checkout -b ps5-validations
```

Implement:

- All validations + meaningful exceptions
- Ensure totals never negative

Commit + merge:

```
git add .
git commit -m "feat: add validations and exception handling for
business rules"
git checkout main
git merge ps5-validations
```

Tests Branch: JUnit tests

```
git checkout -b ps6-tests
```

Implement:

- Unit tests for each rule + edge cases

Commit + merge:

```
git add .
git commit -m "test: add JUnit coverage for tax, fees, discount,
refund, validations"
git checkout main
git merge ps6-tests
```

Demo/Output Branch: main() driver

```
git checkout -b ps7-demo-output
```

Implement:

- Demo runner that creates and processes all orders
- Shows tracking + refunds

Commit + merge:

```
git add .
git commit -m "feat: add demo runner showcasing all order flows"
git checkout main
git merge ps7-demo-output
```

Best practices (mandatory for students)

Before creating a new branch, always ensure `main` is up to date:

```
 git checkout main
git pull
```

-
- One problem statement per branch. No mixing.
- Commit messages must be meaningful: `feat:`, `test:`, `fix:`, `chore:`.