Hey team — Now you are working as an SDE, and I want you to build a **real-feeling Flight Booking mini-app** together. This is not just an exercise — it’s how we’ll practise designing clean code, handling errors, and thinking about real product flows (validation → booking → payment → confirmation).

You’ll get experience with:

* object-oriented design (models + services),
* custom exceptions and validation,
* simple business logic (seat allocation, pricing), and

Follow the steps below exactly one by one. Ask questions when stuck, but try implementing first — that’s how you learn fastest.

# **Problem statement**

Build a small Flight Booking system that:

* keeps a catalog of flights (flight number, source, destination, seats by class),
* validates passenger details before booking (name, age, passport, seat type),
* checks seat availability and allocates a seat number,
* simulates a payment (basic billing),
* and returns a booking receipt (booking id, seat number, price).

If validation fails or seats aren’t available, throw meaningful custom exceptions and show friendly error messages.

# **Goals / Features**

1. Passenger validation (name, age ≥ 18, passport = exactly 8 alphanumeric chars, seat type one of Economy/Business/FirstClass).
2. Flights with seat inventories for each class and automatic seat number generation (E1, E2, B1, F1...).
3. Booking service to search flights and book a passenger on a flight.
4. Simple payment service to calculate price by seat class.
5. Produce a BookingReceipt if successful, or a clear error message if not.
6. Clean code structure and comments so interns can extend it later.

# **Class descriptions (fields and responsibilities)**

1. **Passenger**
   * Fields: name, age, passportNumber, seatClass, seatNumber (assigned on booking)
   * Methods: getters, setters
2. **SeatClass (enum)**
   * Values: ECONOMY, BUSINESS, FIRSTCLASS
3. **Flight**
   * Fields: flightNumber, source, destination, availableSeats (Map<SeatClass,Integer>), seatCounters (Map<SeatClass,Integer>), bookedPassengers (List)
   * Methods: getAvailableSeats(SeatClass), allocateSeat(SeatClass) → returns seat number or throws NoSeatsAvailableException, addPassenger(Passenger), getFlightSummary()
4. **FlightValidator**
   * Methods: validateName, validateAge, validatePassport, validateSeatClass, and validate(Passenger) which throws specific custom exceptions
5. **PaymentService**
   * Method: charge(SeatClass) → returns price, throws PaymentFailedException if payment fails (we’ll keep it simple and succeed by default)
6. **BookingService**
   * Fields: List<Flight> flights, FlightValidator validator, PaymentService paymentService
   * Methods: addFlight(Flight), searchByDestination(String), book(String flightNumber, Passenger p) → validates, allocates seat, charges, returns BookingReceipt
7. **BookingReceipt**
   * Fields: bookingId, flightNumber, passengerName, seatNumber, price
   * toString() for printing
8. **Custom exceptions**
   * InvalidNameException, InvalidAgeException, InvalidPassportException, InvalidSeatClassException, NoSeatsAvailableException, PaymentFailedException

# **Step-by-step development plan (each step, one by one)**

Follow these steps in order. Assign one intern to each step (or pair them) and rotate reviewers.

1. **Step 1 — Create models**
   * Implement Passenger and SeatClass enum.
   * Simple POJOs with getters/setters.
   * Unit check: instantiate a Passenger and print fields.
2. **Step 2 — Implement Flight**
   * Implement constructor to accept seat counts for each class.
   * Maintain availableSeats and seatCounters.
   * Implement allocateSeat(SeatClass) that returns a seat id (E1, B1, F1).
   * Unit check: create a flight, allocate a few seats, check seat numbers and remaining seats.
3. **Step 3 — Write custom exceptions**
   * Add InvalidNameException, InvalidAgeException, InvalidPassportException, InvalidSeatClassException, NoSeatsAvailableException, PaymentFailedException.
4. **Step 4 — Implement FlightValidator**
   * Write methods to validate name, age, passport, seatClass.
   * Implement validate(Passenger) which composes above validations and throws the specific exceptions.
   * Unit check: test with good and bad passengers.
5. **Step 5 — Implement PaymentService**
   * Map prices for seat classes (e.g., Economy = 5,000; Business = 15,000; FirstClass = 30,000).
   * Implement charge(SeatClass) returning price. (Keep it deterministic; optional simulate random failures later.)
   * Unit check: call charge for each class.
6. **Step 6 — Implement BookingService**
   * Fields: flight list, validator, paymentService.
   * Implement addFlight(Flight), searchByDestination(String).
   * Implement book(flightNumber, passenger):  
     + validate passenger,
     + get flight,
     + allocate seat (may throw NoSeatsAvailableException),
     + call paymentService.charge,
     + set seatNumber into passenger, add to flight passengers,
     + create and return BookingReceipt with a bookingId.
   * Unit check: Perform a full booking flow for a valid passenger.
7. **Step 7 — Write main / tester**
   * Create flights, add them to BookingService.
   * Create sample passengers (some valid, some invalid).
   * Try booking and print either receipt or error.
8. **Step 8 — Code review & improvements**
   * Review exceptions/messages, add logs, tidy names.
   * Add concurrency protection if multiple bookings at once (future task).
   * Add persistence or JSON export (future task).
9. **Step 9 — Stretch goals**
   * Add cancellation, seat re-allocation, multiple passengers per booking, CSV import/export, web front end.