Displayable (interface) — A tiny contract for anything that can show a human-readable line of info. Classes that implement it must provide displayInfo(), which the menu uses to print details (e.g., a vehicle's rate or identity). This helps keep printing logic consistent across different object types.
Vehicle (abstract) — The common parent of all vehicles. It stores basic identity (plate/owner) and defines the abstract getRatePerHour() so each subtype supplies its own price. It also implements displayInfo() so any Vehicle can be printed uniformly. Using Vehicle lets the lot treat cars, motorcycles, and EVs polymorphically.
Car — A concrete Vehicle that overrides getRatePerHour() with the car rate and (optionally) displayInfo() to include that rate. Anywhere the code expects a Vehicle, a Car can plug in with car-specific behavior.
Motorcycle — Another concrete Vehicle with its own hourly rate. It behaves like a Car from the lot's point of view (same API), but pricing differs. This demonstrates runtime polymorphism: the lot calls getRatePerHour() without caring which subtype it is.
EVCar — A concrete Vehicle for electric cars with its own rate. Same interface as the others; only the implementation of getRatePerHour() (and printed message) changes. Together, these three show inheritance + overriding.
Ticket — Represents a single active parking session: which vehicle is parked, where it sits (row/col), and when it started. It exposes fee(nowEpochMin) to compute what the driver owes on exit using **rounded-up hours**: minutes = now – start, hours = ceil(minutes / 60.0), fee = hours × ratePerHour. The program prints values from the ticket when you park or leave.
ParkingLot — The core model of the garage. It keeps the 2D grid of spots (E empty, O occupied) and an ArrayList of active Tickets. The **overloaded** park(...) either takes a ready Vehicle or builds one from (type, plate, owner), finds the first empty spot row-major, marks it occupied, and returns a new Ticket (or throws if full). leave(plate) finds the ticket by plate, frees the spot, computes the fee, and removes the ticket. It also provides spotMap() to print the grid and save()/load() to persist and restore the lot using **binary I/O** in a strict write/read order.
LotFullException — A checked exception thrown by park(...) when no empty spot exists. Its purpose is to make the "no capacity" case explicit and force the caller to handle it in the UI.
InvalidSpotException — A checked exception used when a spot index is out of bounds (e.g., isFree(r, c) with bad coordinates). It cleanly separates input/logic errors from normal control flow.
ParkingSystem (main) — The console UI loop. On start it creates a ParkingLot, calls load() to restore prior state, and shows a menu for Map/Rate/Park/Leave/Exit. Each option delegates to the lot's methods; on Exit it calls save() so the grid and tickets persist for the next run.