Module 3 of 7

Parking lot

Levels of sized spots, a best-fit SpotStrategy, a vehicle factory, tickets and a PricingStrategy swapped while the lot runs.

The parking lot is the classic warm-up of the machine-coding round: small enough for an hour, rich enough to show composition, an enum that carries a rule, a factory, and the Strategy pattern twice — once for *which spot*, once for *how much*.

You will code best-fit allocation across levels (lowest level first, smallest spot that fits), then tickets and fees with an hourly or flat tariff that can be switched while the lot is running. The checkpoint puts both together. Hidden cases run a full lot, a truck that finds no large spot, a bike pushed into a large spot because nothing smaller is free, and a tariff changed between two departures.

It is the practice half of the note Design a Parking Lot.

CONTENTS

1	Parking lot	12 min
✓	Checkpoint — Parking lot	30 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 12 MIN

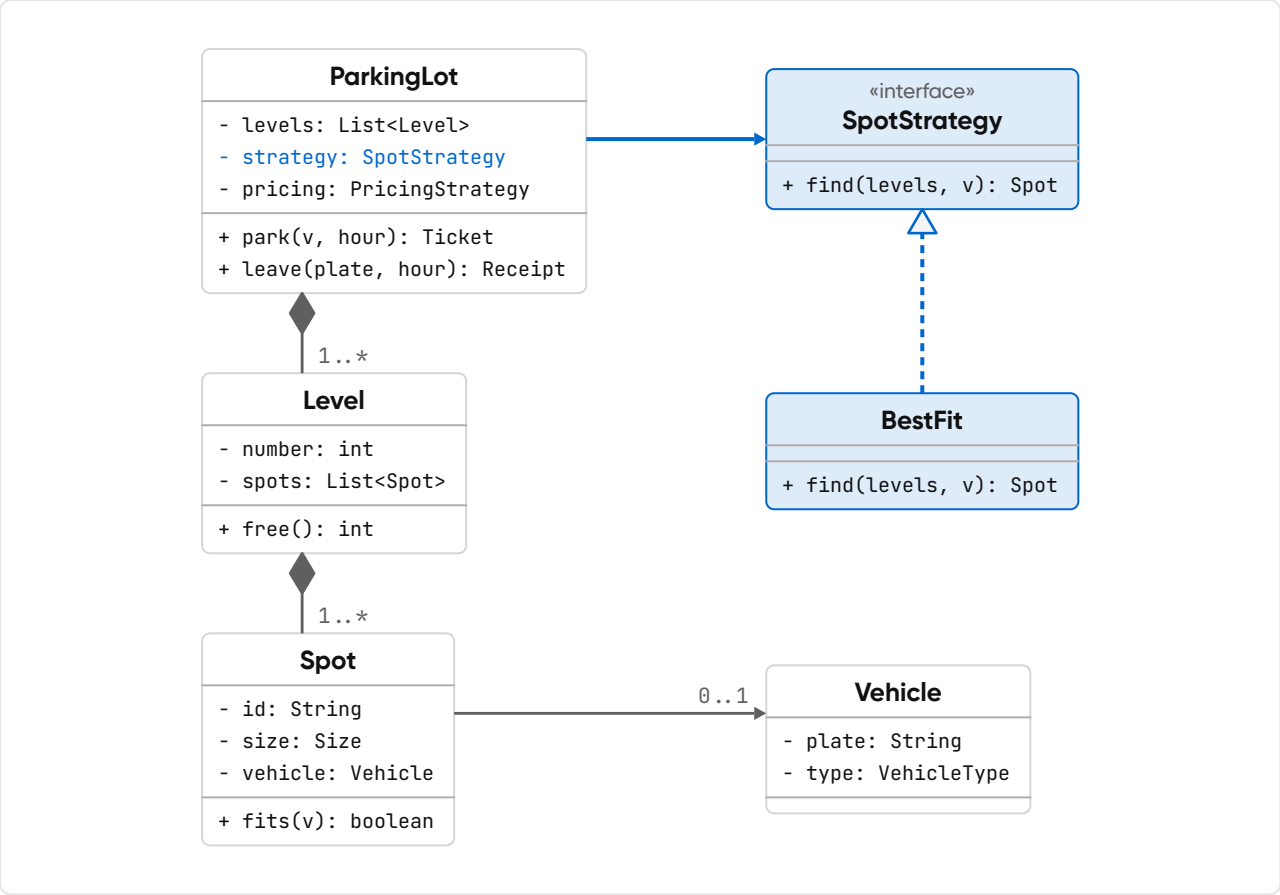
Parking lot

The warm-up of the machine-coding round, and the best place to show two patterns working together. The full design discussion — gates, concurrency, display boards — is in the note Design a Parking Lot. Here we code the core.

The model

- **ParkingLot** owns **Levels**; a level owns **ParkingSpots** (composition).
- A spot has a **Size** — `S < M < L` — and holds 0..1 **Vehicle**.
- A **VehicleType** knows the smallest size it needs: bike S, car M, truck L. "Fits" is one comparison: `free && size ⩾ needs`.
- A **Ticket** remembers the vehicle, the spot and the entry hour.

Parking lot classes: composition below, a strategy beside



1. ParkingLot is what the gates call: park issues a ticket, leave returns a receipt. It holds the levels and two strategies, and decides neither the spot nor the fee itself.
2. The lot owns its levels and each level owns its spots — composition both times. A level keeps its spots in a List, in the order they were given.
3. A spot holds at most one vehicle. Its Size and the vehicle type's need are both ordered S, M, L, so fits is a single comparison: free, and size at least the need.
4. Which spot is the strategy's business. The lot calls SpotStrategy.find; BestFit is one implementation, and a nearest-to-the-lift rule would be another class.

Strategy, twice

The lot never decides *which* spot or *how much*. It asks:

- `SpotStrategy.find(levels, vehicle)` — here **best fit**: lowest level first, then the smallest free spot that fits, then the order the spots were listed.
- `PricingStrategy.fee(ticket, exitHour)` — **hourly** (a rate per type, at least one hour) or **flat**.

A new tariff is a new class; the lot can even switch tariff while running, which is what the second program's `PRICING` command does.

Best fit: lowest level first, then the smallest spot that fits




```
PARK KA05 CAR
→ KA05 parked at level 1 spot M1
```

1. Two levels, all spots free. A bike fits any spot, a car a medium or large one, a truck only a large one; best fit tries level 1 before level 2.
2. KA01, a car needing size M: on level 1, M1 and L1 fit a car; best fit takes the smallest, M1, keeping bigger spots for bigger vehicles.
3. KA02, a bike needing size S: on level 1, S1 and L1 fit a bike; best fit takes the smallest, S1, keeping bigger spots for bigger vehicles.
4. KA03, a truck needing size L: on level 1 only L1 fits a truck, so that is the spot.
5. KA04, a car needing size M: level 1 has no free spot that fits a car, so best fit moves to level 2 and takes M1.
6. KA01 leaves and level 1 spot M1 is free again. Nothing else moves: parked vehicles keep their spots.
7. KA05, a car needing size M: on level 1 only M1 fits a car, so that is the spot.

Factory and errors

- A `VehicleFactory` turns `"CAR"` into a `Vehicle` and refuses an unknown type in one place.
- Refusals (already parked, not parked, unknown type) are exceptions printed once by `main`; *no spot* is a normal answer, not an error.

The traps

- **if-chains on type** inside `park`: a new vehicle type edits the lot. Put the size on the type.
- **Fee logic in `ParkingLot`**: the lot changes every time marketing changes the tariff.
- **Searching by `HashMap` iteration**: spot order must be the order you were given — keep spots in a `List`.

EXERCISE 1

Park and leave, best fit

The first line of input is the number of levels `n`; the next `n` lines list each level's spot ids, in order. Spot ids start with their size — `S1` small, `M2` medium, `L1` large. A bike fits any spot, a car a medium or large one, a truck only a large one. Then commands, one per line:

Command	Prints
`PARK <plate> <BIKE|CAR|TRUCK>`	`<plate> parked at level <L> spot <id>`, or `no spot for <plate>`
`LEAVE <plate>`	`<plate> left level <L> spot <id>`
`STATUS`	each level as `level <L>: <k> free` or `level <L>: full`, joined by `,`

Best fit: try the lowest level first; on a level, take the smallest free spot that fits; among spots of that size, the one listed first. Move to the next level only when nothing on this one fits.

Refusals print `error: <reason>`: `unknown vehicle type <T>` (checked first), `<plate> is already parked`, `<plate> is not parked`, `unknown command <WORD>`.

INPUT 1

```
2
S1 M1 L1
M1 M2
PARK KA01 CAR
PARK KA02 BIKE
PARK KA03 TRUCK
PARK KA04 CAR
STATUS
LEAVE KA01
PARK KA05 CAR
STATUS
```

OUTPUT 1

```
KA01 parked at level 1 spot M1
KA02 parked at level 1 spot S1
KA03 parked at level 1 spot L1
KA04 parked at level 2 spot M1
level 1: full, level 2: 1 free
KA01 left level 1 spot M1
KA05 parked at level 1 spot M1
level 1: full, level 2: 1 free
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static class ParkingException extends RuntimeException {
        ParkingException(String message) { super(message); }
    }

    /** Spot sizes, smallest first: a spot fits a vehicle when its size is at least
    enum Size { S, M, L }

    enum VehicleType {
        BIKE(Size.S), CAR(Size.M), TRUCK(Size.L);

        final Size needs;

        VehicleType(Size needs) { this.needs = needs; }
    }

    static class Vehicle {
        final String plate;
        final VehicleType type;

        Vehicle(String plate, VehicleType type) { this.plate = plate; this.type = ty
    }

    /** The one place that turns a type name into a Vehicle — and refuses an unknown
    static class VehicleFactory {
        static Vehicle create(String plate, String type) {
            for (VehicleType t : VehicleType.values()) {
                if (t.name().equals(type)) return new Vehicle(plate, t);
            }
            throw new ParkingException("unknown vehicle type " + type);
        }
    }

    static class Spot {
        final int level;
        final String id;
        final Size size;
        Vehicle vehicle; // null while free

        Spot(int level, String id) {
            this.level = level;
            this.id = id;
            this.size = Size.valueOf(id.substring(0, 1)); // S1, M2, L1: the first l
        }

        boolean fits(Vehicle v) { return vehicle == null && size.compareTo(v.type.ne
```

```
        String name() { return "level " + level + " spot " + id; }
    }

    static class Level {
        final int number;
        final List<Spot> spots = new ArrayList<>(); // in the order given: ties go t

        Level(int number, String[] ids) {
            this.number = number;
            for (String id : ids) spots.add(new Spot(number, id));
        }

        int free() {
            int n = 0;
            for (Spot s : spots) if (s.vehicle == null) n++;
            return n;
        }
    }

    interface SpotStrategy {
        /** A free spot that fits, or null. */
        Spot find(List<Level> levels, Vehicle v);
    }

    /** Lowest level first; on it, the smallest free spot that fits; ties go to the
    static class BestFit implements SpotStrategy {
        public Spot find(List<Level> levels, Vehicle v) {
            for (Level level : levels) {
                Spot best = null;
                for (Spot s : level.spots) {
                    if (s.fits(v) && (best == null || s.size.compareTo(best.size) <
                }
                if (best != null) return best;
            }
            return null;
        }
    }

    static class ParkingLot {
        private final List<Level> levels;
        private final SpotStrategy strategy;
        private final Map<String, Spot> parked = new HashMap<>();

        ParkingLot(List<Level> levels, SpotStrategy strategy) { this.levels = levels

        /** The spot taken, or null when no free spot fits. */
        Spot park(Vehicle v) {
            if (parked.containsKey(v.plate)) throw new ParkingException(v.plate + "
```

```
        Spot spot = strategy.find(levels, v);
        if (spot == null) return null;
        spot.vehicle = v;
        parked.put(v.plate, spot);
        return spot;
    }

    Spot leave(String plate) {
        Spot spot = parked.remove(plate);
        if (spot == null) throw new ParkingException(plate + " is not parked");
        spot.vehicle = null;
        return spot;
    }

    String status() {
        StringBuilder sb = new StringBuilder();
        for (Level level : levels) {
            if (sb.length() > 0) sb.append(", ");
            int free = level.free();
            sb.append("level ").append(level.number).append(": ").append(free ==
        }
        return sb.toString();
    }
}

static String handle(ParkingLot lot, String[] p) {
    switch (p[0]) {
        case "PARK": {
            Vehicle v = VehicleFactory.create(p[1], p[2]);
            Spot spot = lot.park(v);
            return spot == null ? "no spot for " + p[1] : p[1] + " parked at " +
        }
        case "LEAVE":
            return p[1] + " left " + lot.leave(p[1]).name();
        case "STATUS":
            return lot.status();
        default:
            return "error: unknown command " + p[0];
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int n = Integer.parseInt(br.readLine().trim());
    List<Level> levels = new ArrayList<>();
    for (int i = 1; i <= n; i++) levels.add(new Level(i, br.readLine().trim().sp
    ParkingLot lot = new ParkingLot(levels, new BestFit());
    StringBuilder out = new StringBuilder();
    String line;
```

```
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String reply;
            try {
                reply = handle(lot, line.split("\\s+"));
            } catch (ParkingException e) {
                reply = "error: " + e.getMessage();
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Tickets and a pricing strategy

One level this time: the first line lists its spot ids. Spot ids start with their size — `S1` small, `M2` medium, `L1` large. A bike fits any spot, a car a medium or large one, a truck only a large one. Allocation is best fit, as before. Parking issues a ticket with the entry hour; leaving charges a fee with the **pricing strategy in force at that moment**, which starts as hourly.

- **Hourly**: BIKE 10, CAR 20, TRUCK 40 for each hour parked (`exit - entry`), at least one hour.
- **Flat**: the same amount whatever the vehicle and the time.

Command	Prints
`PARK <plate> <TYPE> <hour>`	`<plate> parked at <id>`, or `no spot for <plate>`
`LEAVE <plate> <hour>`	`<plate> left <id>, paid <fee>`
`PRICING HOURLY` / `PRICING FLAT <amount>`	`pricing: hourly` / `pricing: flat <amount>`
`EARNINGS`	`earnings: <total of every fee charged>`

Refusals print `error: <reason>`: `unknown vehicle type <T>`, `<plate> is already parked`, `<plate> is not parked`, `unknown pricing <WORD>`, `unknown command <WORD>`.

INPUT 1

```
S1 M1 L1
PARK KA01 CAR 9
PARK KA02 BIKE 10
LEAVE KA01 12
LEAVE KA02 10
EARNINGS
```

OUTPUT 1

```
KA01 parked at M1
KA02 parked at S1
KA01 left M1, paid 60
KA02 left S1, paid 10
earnings: 70
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static class ParkingException extends RuntimeException {
        ParkingException(String message) { super(message); }
    }

    /** Spot sizes, smallest first: a spot fits a vehicle when its size is at least
    enum Size { S, M, L }

    enum VehicleType {
        BIKE(Size.S), CAR(Size.M), TRUCK(Size.L);

        final Size needs;

        VehicleType(Size needs) { this.needs = needs; }
    }

    static class Vehicle {
        final String plate;
        final VehicleType type;

        Vehicle(String plate, VehicleType type) { this.plate = plate; this.type = ty
    }

    /** The one place that turns a type name into a Vehicle — and refuses an unknown
    static class VehicleFactory {
        static Vehicle create(String plate, String type) {
            for (VehicleType t : VehicleType.values()) {
                if (t.name().equals(type)) return new Vehicle(plate, t);
            }
            throw new ParkingException("unknown vehicle type " + type);
        }
    }

    static class Spot {
        final int level;
        final String id;
        final Size size;
        Vehicle vehicle; // null while free

        Spot(int level, String id) {
            this.level = level;
            this.id = id;
            this.size = Size.valueOf(id.substring(0, 1)); // S1, M2, L1: the first l
        }

        boolean fits(Vehicle v) { return vehicle == null && size.compareTo(v.type.ne
```

```
    String name() { return "level " + level + " spot " + id; }
}

static class Level {
    final int number;
    final List<Spot> spots = new ArrayList<>(); // in the order given: ties go t

    Level(int number, String[] ids) {
        this.number = number;
        for (String id : ids) spots.add(new Spot(number, id));
    }

    int free() {
        int n = 0;
        for (Spot s : spots) if (s.vehicle == null) n++;
        return n;
    }
}

interface SpotStrategy {
    /** A free spot that fits, or null. */
    Spot find(List<Level> levels, Vehicle v);
}

/** Lowest level first; on it, the smallest free spot that fits; ties go to the
static class BestFit implements SpotStrategy {
    public Spot find(List<Level> levels, Vehicle v) {
        for (Level level : levels) {
            Spot best = null;
            for (Spot s : level.spots) {
                if (s.fits(v) && (best == null || s.size.compareTo(best.size) <
                }
            if (best != null) return best;
        }
        return null;
    }
}

static class Ticket {
    final Vehicle vehicle;
    final Spot spot;
    final int entryHour;

    Ticket(Vehicle vehicle, Spot spot, int entryHour) { this.vehicle = vehicle;
}

interface PricingStrategy {
    int fee(Ticket ticket, int exitHour);
```

```
    String describe();
}

/** A rate per vehicle type for each hour parked, at least one hour. */
static class Hourly implements PricingStrategy {
    private final Map<VehicleType, Integer> rates = new EnumMap<>(VehicleType.cl

    Hourly() {
        rates.put(VehicleType.BIKE, 10);
        rates.put(VehicleType.CAR, 20);
        rates.put(VehicleType.TRUCK, 40);
    }

    public int fee(Ticket ticket, int exitHour) {
        int hours = Math.max(1, exitHour - ticket.entryHour);
        return hours * rates.get(ticket.vehicle.type);
    }

    public String describe() { return "hourly"; }
}

static class Flat implements PricingStrategy {
    private final int amount;

    Flat(int amount) { this.amount = amount; }

    public int fee(Ticket ticket, int exitHour) { return amount; }

    public String describe() { return "flat " + amount; }
}

static PricingStrategy pricingFrom(String[] p) {
    if (p[1].equals("HOURLY")) return new Hourly();
    if (p[1].equals("FLAT")) return new Flat(Integer.parseInt(p[2]));
    throw new ParkingException("unknown pricing " + p[1]);
}

/** What a departure answers: the spot freed and the fee charged. */
static class Receipt {
    final Spot spot;
    final int fee;

    Receipt(Spot spot, int fee) { this.spot = spot; this.fee = fee; }
}

static class ParkingLot {
    private final List<Level> levels;
    private final SpotStrategy strategy;
    private PricingStrategy pricing = new Hourly();
```

```
    private final Map<String, Ticket> tickets = new HashMap<>();
    private int earnings = 0;

    ParkingLot(List<Level> levels, SpotStrategy strategy) { this.levels = levels

    void setPricing(PricingStrategy pricing) { this.pricing = pricing; }

    /** The ticket issued, or null when no free spot fits. */
    Ticket park(Vehicle v, int hour) {
        if (tickets.containsKey(v.plate)) throw new ParkingException(v.plate + '
        Spot spot = strategy.find(levels, v);
        if (spot == null) return null;
        spot.vehicle = v;
        Ticket ticket = new Ticket(v, spot, hour);
        tickets.put(v.plate, ticket);
        return ticket;
    }

    /** Frees the spot and charges with the tariff in force now. */
    Receipt leave(String plate, int hour) {
        Ticket ticket = tickets.remove(plate);
        if (ticket == null) throw new ParkingException(plate + " is not parked")
        ticket.spot.vehicle = null;
        int fee = pricing.fee(ticket, hour);
        earnings += fee;
        return new Receipt(ticket.spot, fee);
    }

    int earnings() { return earnings; }

    String status() {
        StringBuilder sb = new StringBuilder();
        for (Level level : levels) {
            if (sb.length() > 0) sb.append(", ");
            int free = level.free();
            sb.append("level ").append(level.number).append(": ").append(free ==
        }
        return sb.toString();
    }
}

static String handle(ParkingLot lot, String[] p) {
    switch (p[0]) {
        case "PARK": {
            Vehicle v = VehicleFactory.create(p[1], p[2]);
            Ticket ticket = lot.park(v, Integer.parseInt(p[3]));
            return ticket == null ? "no spot for " + p[1] : p[1] + " parked at '
        }
        case "LEAVE": {
```

```
                Receipt r = lot.leave(p[1], Integer.parseInt(p[2]));
                return p[1] + " left " + r.spot.id + ", paid " + r.fee;
            }
            case "PRICING": {
                PricingStrategy pricing = pricingFrom(p);
                lot.setPricing(pricing);
                return "pricing: " + pricing.describe();
            }
            case "EARNINGS":
                return "earnings: " + lot.earnings();
            case "STATUS":
                return lot.status();
            default:
                return "error: unknown command " + p[0];
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        List<Level> levels = new ArrayList<>();
        levels.add(new Level(1, br.readLine().trim().split("\\s+")));
        ParkingLot lot = new ParkingLot(levels, new BestFit());
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String reply;
            try {
                reply = handle(lot, line.split("\\s+"));
            } catch (ParkingException e) {
                reply = "error: " + e.getMessage();
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

QUIZ

1. Level 1 has a free **L1** only; level 2 has a free **M1**. A car arrives. Where does best fit (as specified here) park it?

A. Level 2 M1, the better fit

B. Level 1 L1 — the lowest level with any spot that fits comes first

C. Nowhere

D. Whichever was freed most recently

2. Marketing wants a weekend flat fee. With the Strategy design, what changes?

A. ParkingLot.leave gets an if for weekends

B. A new PricingStrategy class; ParkingLot is untouched

C. Ticket gets a weekend flag

D. VehicleType gets a weekend rate

3. Why give `VehicleType` the size it needs (`CAR(Size.M)`) instead of a `switch` in `park` ?

A. Enums are faster

B. A new vehicle type then carries its own rule; nothing else changes

C. Java forbids switch on enums

D. It saves memory

4. A truck arrives and no large spot is free. What should `park` do?

A. Throw an exception

B. Return null (or an empty result) that main prints as "no spot"

C. Park it in two medium spots

D. Wait until a spot frees

ANSWER KEY · PARKING LOT

1. B Best fit here is *lowest level first*, then the smallest fitting spot on that level. Level 1 has a spot that fits, so level 2 is never searched. A different rule (smallest spot anywhere) would be another SpotStrategy — that is the point of the interface.

2. B The lot depends only on the PricingStrategy interface, so a new tariff is a new class — open for extension, closed for modification.

3. B A switch on type inside the lot must be edited for every new type, and it tends to be copied. Putting the need on the enum keeps the rule with the data, and `fits` becomes one comparison.

4. B A full lot is an expected outcome, not a programming error or a bad request, so it is a normal return value. Exceptions are for refusals such as an unknown type or a plate already inside.

CHECKPOINT · 30 MIN

Checkpoint — Parking lot

This checkpoint covers the module: composition of lot, levels and spots; a vehicle type that carries the size it needs; the vehicle factory; best-fit allocation; tickets; and a pricing strategy that can be switched while the lot runs.

How it works. Ten questions and one program. You need 70% on the questions and the program accepted to clear the module.

The program puts both lesson programs together: several levels, best fit, tickets with entry hours, and hourly or flat pricing. Before you code, check your design against these:

- Which class decides *which spot*, and which decides *how much*?
- Where is the rule "a car fits a medium or large spot" written — once?
- What does `park` return when nothing fits, and what does it throw?

Hidden cases switch the tariff between two departures, fill a level so a vehicle moves up, and charge a vehicle that leaves in the hour it arrived.

EXERCISE

A multi-level lot with fees

Put both programs together. The first line is the number of levels `n`, then `n` lines of spot ids. Spot ids start with their size — `S1` small, `M2` medium, `L1` large. A bike fits any spot, a car a medium or large one, a truck only a large one. Allocation is best fit (lowest level, then smallest fitting spot, then the first listed); pricing starts hourly (BIKE 10, CAR 20, TRUCK 40 an hour, at least one hour) and can be switched.

Command	Prints
`PARK <plate> <TYPE> <hour>`	`<plate> parked at level <L> spot <id>` or `no spot for <plate>`
`LEAVE <plate> <hour>`	`<plate> left level <L> spot <id>, paid <fee>`
`PRICING HOURLY` / `PRICING FLAT <amount>`	`pricing: hourly` / `pricing: flat <amount>`
`STATUS`	`level <L>: <k> free` or `level <L>: full` for each level, joined by `,`
`EARNINGS`	`earnings: <total>`

Refusals print `error: <reason>`: `unknown vehicle type <T>`, `<plate> is already parked`, `<plate> is not parked`, `unknown pricing <WORD>`, `unknown command <WORD>`.

INPUT 1

```
2
S1 M1
L1
PARK A CAR 1
PARK B TRUCK 1
PARK C CAR 2
STATUS
LEAVE A 4
PARK C CAR 4
STATUS
EARNINGS
```

OUTPUT 1

```
A parked at level 1 spot M1
B parked at level 2 spot L1
no spot for C
level 1: 1 free, level 2: full
A left level 1 spot M1, paid 60
C parked at level 1 spot M1
level 1: 1 free, level 2: full
earnings: 60
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static class ParkingException extends RuntimeException {
        ParkingException(String message) { super(message); }
    }

    /** Spot sizes, smallest first: a spot fits a vehicle when its size is at least
    enum Size { S, M, L }

    enum VehicleType {
        BIKE(Size.S), CAR(Size.M), TRUCK(Size.L);

        final Size needs;

        VehicleType(Size needs) { this.needs = needs; }
    }

    static class Vehicle {
        final String plate;
        final VehicleType type;

        Vehicle(String plate, VehicleType type) { this.plate = plate; this.type = ty
    }

    /** The one place that turns a type name into a Vehicle — and refuses an unknown
    static class VehicleFactory {
        static Vehicle create(String plate, String type) {
            for (VehicleType t : VehicleType.values()) {
                if (t.name().equals(type)) return new Vehicle(plate, t);
            }
            throw new ParkingException("unknown vehicle type " + type);
        }
    }

    static class Spot {
        final int level;
        final String id;
        final Size size;
        Vehicle vehicle; // null while free

        Spot(int level, String id) {
            this.level = level;
            this.id = id;
            this.size = Size.valueOf(id.substring(0, 1)); // S1, M2, L1: the first l
        }

        boolean fits(Vehicle v) { return vehicle == null && size.compareTo(v.type.ne
```

```
    String name() { return "level " + level + " spot " + id; }
}

static class Level {
    final int number;
    final List<Spot> spots = new ArrayList<>(); // in the order given: ties go t

    Level(int number, String[] ids) {
        this.number = number;
        for (String id : ids) spots.add(new Spot(number, id));
    }

    int free() {
        int n = 0;
        for (Spot s : spots) if (s.vehicle == null) n++;
        return n;
    }
}

interface SpotStrategy {
    /** A free spot that fits, or null. */
    Spot find(List<Level> levels, Vehicle v);
}

/** Lowest level first; on it, the smallest free spot that fits; ties go to the
static class BestFit implements SpotStrategy {
    public Spot find(List<Level> levels, Vehicle v) {
        for (Level level : levels) {
            Spot best = null;
            for (Spot s : level.spots) {
                if (s.fits(v) && (best == null || s.size.compareTo(best.size) <
                }
            if (best != null) return best;
        }
        return null;
    }
}

static class Ticket {
    final Vehicle vehicle;
    final Spot spot;
    final int entryHour;

    Ticket(Vehicle vehicle, Spot spot, int entryHour) { this.vehicle = vehicle;
}

interface PricingStrategy {
    int fee(Ticket ticket, int exitHour);
```

```
    String describe();
}

/** A rate per vehicle type for each hour parked, at least one hour. */
static class Hourly implements PricingStrategy {
    private final Map<VehicleType, Integer> rates = new EnumMap<>(VehicleType.cl

    Hourly() {
        rates.put(VehicleType.BIKE, 10);
        rates.put(VehicleType.CAR, 20);
        rates.put(VehicleType.TRUCK, 40);
    }

    public int fee(Ticket ticket, int exitHour) {
        int hours = Math.max(1, exitHour - ticket.entryHour);
        return hours * rates.get(ticket.vehicle.type);
    }

    public String describe() { return "hourly"; }
}

static class Flat implements PricingStrategy {
    private final int amount;

    Flat(int amount) { this.amount = amount; }

    public int fee(Ticket ticket, int exitHour) { return amount; }

    public String describe() { return "flat " + amount; }
}

static PricingStrategy pricingFrom(String[] p) {
    if (p[1].equals("HOURLY")) return new Hourly();
    if (p[1].equals("FLAT")) return new Flat(Integer.parseInt(p[2]));
    throw new ParkingException("unknown pricing " + p[1]);
}

/** What a departure answers: the spot freed and the fee charged. */
static class Receipt {
    final Spot spot;
    final int fee;

    Receipt(Spot spot, int fee) { this.spot = spot; this.fee = fee; }
}

static class ParkingLot {
    private final List<Level> levels;
    private final SpotStrategy strategy;
    private PricingStrategy pricing = new Hourly();
```

```
    private final Map<String, Ticket> tickets = new HashMap<>();
    private int earnings = 0;

    ParkingLot(List<Level> levels, SpotStrategy strategy) { this.levels = levels

    void setPricing(PricingStrategy pricing) { this.pricing = pricing; }

    /** The ticket issued, or null when no free spot fits. */
    Ticket park(Vehicle v, int hour) {
        if (tickets.containsKey(v.plate)) throw new ParkingException(v.plate + '
        Spot spot = strategy.find(levels, v);
        if (spot == null) return null;
        spot.vehicle = v;
        Ticket ticket = new Ticket(v, spot, hour);
        tickets.put(v.plate, ticket);
        return ticket;
    }

    /** Frees the spot and charges with the tariff in force now. */
    Receipt leave(String plate, int hour) {
        Ticket ticket = tickets.remove(plate);
        if (ticket == null) throw new ParkingException(plate + " is not parked")
        ticket.spot.vehicle = null;
        int fee = pricing.fee(ticket, hour);
        earnings += fee;
        return new Receipt(ticket.spot, fee);
    }

    int earnings() { return earnings; }

    String status() {
        StringBuilder sb = new StringBuilder();
        for (Level level : levels) {
            if (sb.length() > 0) sb.append(", ");
            int free = level.free();
            sb.append("level ").append(level.number).append(": ").append(free ==
        }
        return sb.toString();
    }
}

static String handle(ParkingLot lot, String[] p) {
    switch (p[0]) {
        case "PARK": {
            Vehicle v = VehicleFactory.create(p[1], p[2]);
            Ticket ticket = lot.park(v, Integer.parseInt(p[3]));
            return ticket == null ? "no spot for " + p[1] : p[1] + " parked at '
        }
        case "LEAVE": {
```

```
                Receipt r = lot.leave(p[1], Integer.parseInt(p[2]));
                return p[1] + " left " + r.spot.name() + ", paid " + r.fee;
            }
            case "PRICING": {
                PricingStrategy pricing = pricingFrom(p);
                lot.setPricing(pricing);
                return "pricing: " + pricing.describe();
            }
            case "EARNINGS":
                return "earnings: " + lot.earnings();
            case "STATUS":
                return lot.status();
            default:
                return "error: unknown command " + p[0];
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        List<Level> levels = new ArrayList<>();
        for (int i = 1; i <= n; i++) levels.add(new Level(i, br.readLine().trim().sp
        ParkingLot lot = new ParkingLot(levels, new BestFit());
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String reply;
            try {
                reply = handle(lot, line.split("\\s+"));
            } catch (ParkingException e) {
                reply = "error: " + e.getMessage();
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

CHECKPOINT QUESTIONS

1. Which relation is right between `ParkingLot` and `Level`?

A. Inheritance: a Level is a ParkingLot

B. Composition: the lot owns its levels

C. Dependency only

D. Aggregation of independent lots

2. Which pattern lets the lot change *which spot* a vehicle gets without editing `ParkingLot`?

A. Singleton

B. Observer

C. Strategy

D. Decorator

3. What does a `VehicleFactory` add over calling `new Vehicle(...)` everywhere?

A. Speed

B. One place that maps a type name to a vehicle and refuses unknown ones

C. Thread safety

D. It replaces the enum

4. A car parks at hour 9 and leaves at hour 9. Hourly CAR rate 20, minimum one hour. What is the fee?

A. 0

B. 20

C. 40

D. It is refused

5. Pricing is switched from hourly to flat 50 while a truck is parked. It leaves later. Which tariff applies in this design?

A. Hourly, the one at entry

B. Flat 50, the one in force at exit

C. The cheaper of the two

D. Both, added

6. Two gates call `park` at the same moment for the last free spot. What prevents a double booking in a single process?

A. Nothing is needed

B. Making find-and-take one atomic step, for example a lock around it

C. A volatile field on Spot

D. Sorting the spots

7. Why is "no spot" returned as a value while "already parked" is an exception in this design?

A. Exceptions are slow

B. A full lot is a normal outcome; a plate already inside is a request the lot refuses

C. Java requires it

D. There is no reason

8. Spots are kept in a `List` per level, not a `HashMap`. Why does that matter for the output?

A. Lists are faster to search

B. Best fit breaks ties by the order spots were listed; a HashMap's iteration order is unspecified

C. HashMap cannot hold Spot objects

D. It does not matter

9. A new requirement: electric cars must park in charging spots. What is the cleanest change?

A. An if in ParkingLot.park

B. A new VehicleType and spot capability, with a SpotStrategy that honours it

C. Remove the factory

D. A separate lot for electric cars

10. Is a `Singleton` ParkingLot a good idea in this exercise?

A. Yes, always

B. Usually not: passing one instance in is simpler to test and allows several lots

C. Yes, because there is one main

D. Only for the pricing

ANSWER KEY · CHECKPOINT — PARKING LOT

1. B Levels exist only as part of their lot and are created with it — composition. A level is not a kind of lot, so inheritance is wrong.

2. C A SpotStrategy interface (best fit, nearest to the gate, EV spots first) is a family of interchangeable algorithms behind one method — Strategy.

3. B Creation logic — parsing the type, validating it, choosing a subclass if there are several — lives in one place, so adding a vehicle type changes the factory and the enum, not every caller.

4. B `max(1, 9 - 9) = 1` hour × 20 = 20. The minimum keeps a short stay from being free.

5. B The lot asks its current PricingStrategy at `leave`. If a business wants the entry tariff, the ticket would carry its strategy — another design decision to state aloud.

6. B Finding a free spot and taking it must happen together; otherwise both gates see it free. A lock (or a conditional update in a database) makes the second caller retry and get another spot or none.

7. B Callers must always handle a full lot, so it belongs in the return type. Invalid requests are exceptional and get a reason the caller can print.

8. B A deterministic tie-break needs a defined order. A `List` keeps the input order; a HashMap could return a different spot on another JDK.

9. B The new vehicle kind and its need go into the type and the spot, and the allocation rule into a strategy. The lot's own code stays the same.

10. B A singleton is global state: tests cannot build a fresh lot, and a second lot (another site) becomes impossible. Create one and pass it to whoever needs it.
