Module 5 of 7

Elevator system

One car under LOOK, then two cars and a dispatcher, then scheduling policies as a Strategy, simulated tick by tick in Java.

The elevator is the machine-coding problem where an object model has to carry an algorithm. The classes are few (an Elevator with a floor, a direction and a sorted set of stops; a Dispatcher that owns the cars), but the behaviour has to be exactly right: which stop comes next, when the car turns, and which car answers a call.

You will build it the way a round grows it. First one car that moves one floor per tick under LOOK, the policy that keeps its direction while a stop lies ahead and turns when none does. Then a dispatcher that sends each hall call to the nearest car idle or heading its way. In the checkpoint, LOOK, SCAN and first come first served sit behind one Scheduler interface and serve the same stops three ways.

Every program is a simulation driven by a script of commands, so the same input always prints the same lines. The design behind it is in the elevator system note.

CONTENTS

1	Elevator system: one car, then a dispatcher	14 min
✓	Checkpoint: Elevator system	25 min

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

LESSON 1 · 14 MIN

Elevator system: one car, then a dispatcher

An elevator is the machine-coding problem where an object model has to carry a **scheduling rule**. The exercises grow it the way a round does: one car first, then two cars and a dispatcher. The elevator system note has the full design; this lesson is the part you type.

The classes

- **Elevator**: `floor`, `dir` and `stops`, a `TreeSet<Integer>`. Sorted, so the next stop above is `stops.higher(floor)` and below is `stops.lower(floor)`; a repeated call vanishes on its own.
- **Direction**: an enum `UP`, `DOWN`, `IDLE`. It is the car's **state**: it decides which stops count as ahead.
- **Dispatcher**: owns the cars and answers hall calls. A car call, pressed inside, skips it and goes straight to its car.
- **Scheduler**: the rule for the next stop, behind an interface so LOOK, SCAN or first come first served can be swapped (Strategy).

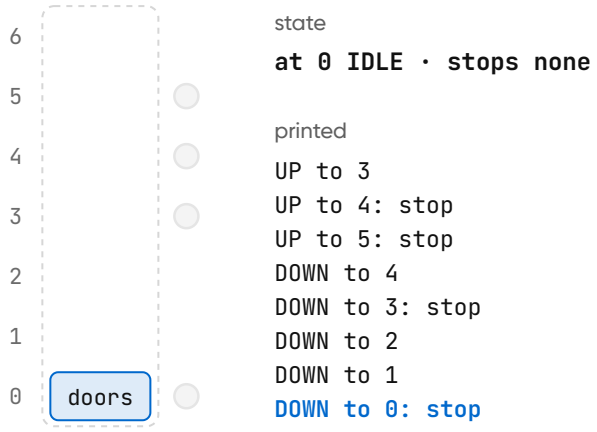
One tick at a time

A simulation moves time in ticks. On each `STEP` a car does exactly one thing:

1. No stops: it is idle and says so.
2. A stop at the floor it is on: the doors open, nothing moves.
3. Otherwise it moves one floor in its direction, and stops if that floor was asked for.

Then LOOK settles the direction: **keep it while a stop lies ahead, turn when none does, go idle when none is left.**

LOOK, one STEP at a time: up for 4 and 5, then down for 3 and 0



1. The car is idle at floor 2. CALL 5 gives it a direction, up, at once; CALL 0 is a second stop, below it.
2. STEP: a stop lies ahead at 5, so the car moves up one floor. Meanwhile CALL 4 comes in, ahead of the car, and joins this sweep.
3. STEP: the car moves to 4, which is a stop, so it is served in the same tick. CALL 3 arrives now, behind the car: it must wait.
4. STEP: up to 5, the last stop above. Nothing lies ahead any more, so LOOK turns the car down as it stops, instead of riding on to the top.
5. STEP: down to 4. It was served on the way up, so the car passes it; the stops ahead are 3 and 0.
6. STEP: down to 3, the call that came in behind the car. It waited exactly one sweep.
7. STEP: down to 2, no one asked for it, so the car keeps moving.
8. STEP: down to 1, the last floor before the remaining stop.
9. STEP: down to 0, the last stop. With no stops left the car goes idle; every call was served within two sweeps.

The traps

- **Nearest stop first.** Short to write, and it starves: a busy cluster near the car keeps winning while a far floor waits for ever. LOOK never turns while a stop lies ahead, so a call behind the car waits one sweep at most. In the figure, the call for 3 came in behind the car; it waited for the turn and was served on the way down.
- **The idle car.** Give it a direction the moment it takes its first call, or the dispatcher still sees it as free and keeps piling calls on it.
- **Moving in the wrong place.** The dispatcher chooses, the car moves. A dispatcher that edits floors is the mixed responsibility interviewers look for.

Dispatching

The dispatcher's rule, in order:

1. Among cars that are idle or already heading towards the floor, take the nearest.
2. If none is, take the nearest car anyway; it serves the call after it turns.
3. Ties go to the lower-numbered car, so the output is deterministic.

A car call never goes through these steps: it belongs to the car it was pressed in, even when the floor is behind it.

EXERCISE 1

One car under LOOK

Simulate one elevator, one floor per tick, under LOOK.

Input. The first line is `top start`: the floors are `0` to `top` and the car starts idle at `start`. Then one command per line:

- `CALL f`: add a stop at floor `f`. Outside `0..top` print `invalid floor f`. A stop that is already there changes nothing. An idle car takes the direction of its first call at once (up if `f` is above it, down if below). Prints nothing otherwise.
- `STEP`: one tick. With no stops print `idle at F`. If the floor the car is on is a stop, serve it without moving: `stop at F`. Otherwise move one floor in the car's direction and print `UP to F` or `DOWN to F`, adding `: stop` when that floor is a stop (it is served and removed). After the tick, LOOK settles the direction: no stops left means `IDLE`; no stop ahead in the current direction means the car turns.
- `STATUS`: `at F DIR stops a b c` with the stops in ascending order, or `stops none`.

Example

```
6 2
CALL 5
CALL 0
STEP
STEP
STATUS
```

prints

```
UP to 3
UP to 4
at 4 UP stops 0 5
```

INPUT 1

```
6 2
CALL 5
CALL 0
STATUS
STEP
CALL 4
STEP
CALL 3
STEP
STATUS
STEP
STEP
STEP
STEP
STEP
STEP
STATUS
```

OUTPUT 1

```
at 2 UP stops 0 5
UP to 3
UP to 4: stop
UP to 5: stop
at 5 DOWN stops 0 3
DOWN to 4
DOWN to 3: stop
DOWN to 2
DOWN to 1
DOWN to 0: stop
idle at 0
at 0 IDLE stops none
```

REFERENCE SOLUTION

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

public class Main {
    enum Direction { UP, DOWN, IDLE }

    static class Elevator {
        final int top;
        int floor;
        Direction dir = Direction.IDLE;
        final TreeSet<Integer> stops = new TreeSet<>();

        Elevator(int top, int floor) { this.top = top; this.floor = floor; }

        boolean stopAhead(Direction d) {
            if (d == Direction.UP) return stops.higher(floor) != null;
            if (d == Direction.DOWN) return stops.lower(floor) != null;
            return false;
        }

        String call(int f) {
            if (f < 0 || f > top) return "invalid floor " + f;
            stops.add(f);
            // An idle car takes the direction of its first call at once.
            if (dir == Direction.IDLE && f != floor) dir = f > floor ? Direction.UP
            return null;
        }

        String step() {
            if (stops.isEmpty()) return "idle at " + floor;
            String line;
            if (stops.remove(floor)) {
                line = "stop at " + floor;
            } else {
                floor += dir == Direction.UP ? 1 : -1;
                line = dir + " to " + floor;
                if (stops.remove(floor)) line += ": stop";
            }
            settle();
            return line;
        }

        // LOOK: keep the direction while a stop lies ahead, turn when none does, id
        void settle() {
            if (stops.isEmpty()) dir = Direction.IDLE;
            else if (dir == Direction.IDLE || !stopAhead(dir)) dir = stopAhead(Direc
        }

        String status() {
```

```
            StringJoiner s = new StringJoiner(" ");
            for (int f : stops) s.add(String.valueOf(f));
            return "at " + floor + " " + dir + " stops " + (stops.isEmpty() ? "none"
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] first = br.readLine().trim().split("\\s+");
        Elevator car = new Elevator(Integer.parseInt(first[0]), Integer.parseInt(fir
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) ≠ null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            String said = switch (p[0]) {
                case "CALL" → car.call(Integer.parseInt(p[1]));
                case "STEP" → car.step();
                case "STATUS" → car.status();
                default → null;
            };
            if (said ≠ null) out.append(said).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Two cars and a dispatcher

Two elevators, `E1` and `E2`, each moving exactly as in the first exercise, and a dispatcher that answers hall calls.

Input. The first line is `top a b`: floors `0` to `top`, `E1` idle at `a`, `E2` idle at `b`. Then one command per line:

- `CALL f`: a hall call. A car **heads towards** `f` when it is idle, going `UP` with `f` at or above its floor, or going `DOWN` with `f` at or below it. Choose the nearest car that heads towards `f`; if none does, the nearest car. A tie goes to `E1`. Add the stop to that car (an idle car takes the call's direction, as before) and print `call f → E1`. A floor outside `0..top` prints `invalid floor f`.
- `PRESS E f`: a car call, added straight to that car's stops. Prints nothing, or `unknown elevator E`, or `invalid floor f` (the car is checked first).

- `STEP` : one tick for `E1` , then `E2` ; each prints its line from the first exercise prefixed with its id, such as `E1 UP to 3` or `E2 idle at 9` .
- `STATUS` : both cars' status lines, prefixed the same way: `E1 at 3 UP stops 6` .

Example

```
10 0 8
CALL 3
CALL 9
STEP
```

prints

```
call 3 → E1
call 9 → E2
E1 UP to 1
E2 UP to 9: stop
```

E1 got 3 (both idle, E1 nearer). For 9, E1 heads towards it but E2 is idle and nearer.

INPUT 1

```
10 0 8
CALL 3
CALL 9
PRESS E1 6
STEP
STATUS
CALL 2
STEP
STEP
STATUS
```

OUTPUT 1

```
call 3 → E1
call 9 → E2
E1 UP to 1
E2 UP to 9: stop
E1 at 1 UP stops 3 6
E2 at 9 IDLE stops none
call 2 → E1
E1 UP to 2: stop
E2 idle at 9
E1 UP to 3: stop
E2 idle at 9
E1 at 3 UP stops 6
E2 at 9 IDLE stops none
```

REFERENCE SOLUTION

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

public class Main {
    enum Direction { UP, DOWN, IDLE }

    static class Elevator {
        final String id;
        int floor;
        Direction dir = Direction.IDLE;
        final TreeSet<Integer> stops = new TreeSet<>();

        Elevator(String id, int floor) { this.id = id; this.floor = floor; }

        boolean headingTo(int f) {
            return dir == Direction.IDLE || (dir == Direction.UP && f >= floor) || (
        }

        boolean stopAhead(Direction d) {
            if (d == Direction.UP) return stops.higher(floor) != null;
            if (d == Direction.DOWN) return stops.lower(floor) != null;
            return false;
        }

        void addStop(int f) {
            stops.add(f);
            if (dir == Direction.IDLE && f != floor) dir = f > floor ? Direction.UP
        }

        String step() {
            if (stops.isEmpty()) return id + " idle at " + floor;
            String line;
            if (stops.remove(floor)) {
                line = id + " stop at " + floor;
            } else {
                floor += dir == Direction.UP ? 1 : -1;
                line = id + " " + dir + " to " + floor;
                if (stops.remove(floor)) line += ": stop";
            }
            // LOOK: keep the direction while a stop lies ahead, turn when none does
            if (stops.isEmpty()) dir = Direction.IDLE;
            else if (dir == Direction.IDLE || !stopAhead(dir)) dir = stopAhead(Direc
            return line;
        }

        String status() {
            StringJoiner s = new StringJoiner(" ");
            for (int f : stops) s.add(String.valueOf(f));
            return id + " at " + floor + " " + dir + " stops " + (stops.isEmpty() ?
```

```
        }
    }

    static class Dispatcher {
        final int top;
        final List<Elevator> cars = new ArrayList<>();

        Dispatcher(int top) { this.top = top; }

        boolean valid(int f) { return f >= 0 && f <= top; }

        String call(int f) {
            if (!valid(f)) return "invalid floor " + f;
            Elevator best = null;
            // First pass: only cars idle or heading towards the call. Second: any c
            for (boolean strict : new boolean[] {true, false}) {
                for (Elevator e : cars)
                    if ((!strict || e.headingTo(f)) && (best == null || Math.abs(e.f
                if (best != null) break;
            }
            best.addStop(f);
            return "call " + f + " -> " + best.id;
        }

        String press(String id, int f) {
            Elevator car = null;
            for (Elevator e : cars) if (e.id.equals(id)) car = e;
            if (car == null) return "unknown elevator " + id;
            if (!valid(f)) return "invalid floor " + f;
            car.addStop(f);
            return null;
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] first = br.readLine().trim().split("\\s+");
        Dispatcher d = new Dispatcher(Integer.parseInt(first[0]));
        d.cars.add(new Elevator("E1", Integer.parseInt(first[1])));
        d.cars.add(new Elevator("E2", Integer.parseInt(first[2])));
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            switch (p[0]) {
                case "CALL" -> out.append(d.call(Integer.parseInt(p[1]))).append('\r
                case "PRESS" -> {
```

```
                    String said = d.press(p[1], Integer.parseInt(p[2]));
                    if (said ≠ null) out.append(said).append('\n');
                }
                case "STEP" → { for (Elevator e : d.cars) out.append(e.step()).appe
                case "STATUS" → { for (Elevator e : d.cars) out.append(e.status()).
                default → { }
            }
        }
        System.out.print(out);
    }
}
```

QUIZ

1. Why is a `TreeSet<Integer>` a good fit for an elevator's stops?

A. It is thread-safe, so buttons can add stops from many threads

B. It is sorted and drops duplicates, so the next stop above or below is one `higher` / `lower` call

C. It keeps the stops in the order they were pressed

D. It is faster than an array for every operation

2. A LOOK car is at floor 5 going up with stops at 8 and 2. A call for 3 arrives. When is 3 served?

A. Next, because it is the nearest stop

B. After 8, on the way down, before 2

C. After 2, on the way back up

D. Never, until the car goes idle

3. Where should the rule "which car answers this hall call" live?

A. In Elevator, as a method each car runs on itself

B. In the Dispatcher, which owns the cars

C. In the Main loop that reads the commands

D. In the Direction enum

4. Why does the dispatcher set an idle car's direction as soon as it assigns the car a call?

A. Java enums cannot stay IDLE once a set is non-empty

B. So the next call sees the car as moving that way, not as free and nearest for anything

C. To make the car move before the next tick

D. It does not matter; direction is only read when the car moves

ANSWER KEY · ELEVATOR SYSTEM: ONE CAR, THEN A DISPATCHER

1. B Sorted order turns "is there a stop ahead?" into `higher(floor)` or `lower(floor)`, O(log n), and a repeated call simply does not add a second entry. `TreeSet` is **not** thread-safe, and arrival order is what a queue keeps, which is first come first served, not LOOK.

2. B LOOK keeps its direction while a stop lies ahead, so it goes up to 8 first. With nothing above 8 it turns, and on the way down it reaches 3 before 2. Serving the nearest stop first is a different policy, and the one that starves far floors.

3. B Choosing among cars needs to see every car, so it belongs to the Dispatcher that owns them. Each Elevator only moves itself and keeps its own stops. Putting the choice in the input loop mixes parsing with policy, and the next requirement (capacity, a rush-hour rule) has nowhere clean to go.

4. B The dispatcher's rule reads direction: idle cars qualify for any floor. A car that has taken a call but still reads IDLE would keep attracting calls in both directions until it moved. Setting the direction does not move it; movement still happens only on a tick.

CHECKPOINT · 25 MIN

Checkpoint: Elevator system

This checkpoint covers the module: the elevator's classes, direction as state, LOOK, dispatching hall calls, and the scheduling policy as a Strategy.

How it works. Ten questions and one program. You need 70% on the questions and the program accepted to clear the module. Retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What does a LOOK car do when no stop lies ahead of it?
- How is SCAN different from LOOK, and when do they travel the same distance?
- Which cars may the dispatcher pick first, and what happens when none qualifies?
- Why does a car call skip the dispatcher?
- Where does the scheduling rule live, so a new policy changes no other class?

The program is the Strategy pattern at work: three policies behind one interface, the same stops served three ways. Work one line through by hand before you code it; the turn at the end of the shaft is where most wrong answers come from.

EXERCISE

Three scheduling policies

Put LOOK, SCAN and first come first served behind one `Scheduler` interface and serve the same stops with each.

Input. The first line is `top` (floors `0` to `top`). Each following line is `POLICY start DIR s1 s2 …`: a policy name, the car's floor, its direction (`UP` or `DOWN`) and the stops in the order they were pressed. Stops are distinct and differ from `start`.

Policies

- `LOOK`: serve every stop ahead in the current direction, nearest first; then turn and do the same.
- `SCAN`: the same, but when stops remain after a sweep, ride on to the end of the shaft (`top` going up, `0` going down) before turning. A car already at that end does not ride.
- `FCFS`: the stops in the order given.

Output. One line per input line: `POLICY: v1 v2 … (N floors)`, the floors in the order visited, SCAN's ride to an end written in brackets like `[10]`, and `N` the floors travelled from `start`. An unknown policy prints `unknown policy NAME`.

Example

```
10
LOOK 4 UP 7 3 9 6
SCAN 4 UP 7 3 9 6
```

prints

```
LOOK: 6 7 9 3 (11 floors)
SCAN: 6 7 9 [10] 3 (13 floors)
```

INPUT 1

```
10
LOOK 4 UP 7 3 9 6
SCAN 4 UP 7 3 9 6
FCFS 4 UP 7 3 9 6
```

OUTPUT 1

```
LOOK: 6 7 9 3 (11 floors)
SCAN: 6 7 9 [10] 3 (13 floors)
FCFS: 7 3 9 6 (16 floors)
```

REFERENCE SOLUTION

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

public class Main {
    enum Direction { UP, DOWN }

    /** One floor the car visits; turn is true for SCAN's ride to the end of the sho
    record Visit(int floor, boolean turn) {}

    interface Scheduler {
        List<Visit> order(int top, int start, Direction dir, List<Integer> stops);
    }

    /** The stops on one side of the floor, nearest first. */
    static List<Integer> ahead(TreeSet<Integer> left, int floor, Direction dir) {
        return dir == Direction.UP ? new ArrayList<>(left.tailSet(floor, false)) : r
    }

    static Direction other(Direction d) { return d == Direction.UP ? Direction.DOWN

    // LOOK: serve everything ahead, then turn at the last stop.
    static class LookScheduler implements Scheduler {
        public List<Visit> order(int top, int start, Direction dir, List<Integer> st
            TreeSet<Integer> left = new TreeSet<>(stops);
            List<Visit> visits = new ArrayList<>();
            int floor = start;
            Direction d = dir;
            while (!left.isEmpty()) {
                for (int f : ahead(left, floor, d)) { visits.add(new Visit(f, false)
                d = other(d);
            }
            return visits;
        }
    }

    // SCAN: serve everything ahead, and if stops remain ride to the end of the shaf
    static class ScanScheduler implements Scheduler {
        public List<Visit> order(int top, int start, Direction dir, List<Integer> st
            TreeSet<Integer> left = new TreeSet<>(stops);
            List<Visit> visits = new ArrayList<>();
            int floor = start;
            Direction d = dir;
            while (!left.isEmpty()) {
                for (int f : ahead(left, floor, d)) { visits.add(new Visit(f, false)
                if (left.isEmpty()) break;
                int end = d == Direction.UP ? top : 0;
                if (floor != end) { visits.add(new Visit(end, true)); floor = end; }
                d = other(d);
            }
```

```
            return visits;
        }
    }

    // First come, first served: the arrival order, whatever the direction.
    static class FcfsScheduler implements Scheduler {
        public List<Visit> order(int top, int start, Direction dir, List<Integer> st
            List<Visit> visits = new ArrayList<>();
            for (int f : stops) visits.add(new Visit(f, false));
            return visits;
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int top = Integer.parseInt(br.readLine().trim());
        Map<String, Scheduler> policies = new HashMap<>();
        policies.put("LOOK", new LookScheduler());
        policies.put("SCAN", new ScanScheduler());
        policies.put("FCFS", new FcfsScheduler());
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            Scheduler s = policies.get(p[0]);
            if (s == null) { out.append("unknown policy ").append(p[0]).append('\n')
            int start = Integer.parseInt(p[1]);
            Direction dir = Direction.valueOf(p[2]);
            List<Integer> stops = new ArrayList<>();
            for (int i = 3; i < p.length; i++) stops.add(Integer.parseInt(p[i]));
            StringJoiner order = new StringJoiner(" ");
            int floor = start, moved = 0;
            for (Visit v : s.order(top, start, dir, stops)) {
                order.add(v.turn() ? "[" + v.floor() + "]" : String.valueOf(v.floor(
                moved += Math.abs(v.floor() - floor);
                floor = v.floor();
            }
            out.append(p[0]).append(": ").append(order).append(" (").append(moved).a
        }
        System.out.print(out);
    }
}
```

CHECKPOINT QUESTIONS

1. What is the difference between SCAN and LOOK?

A. SCAN serves stops in arrival order; LOOK sorts them

B. SCAN rides to the end of the shaft before turning; LOOK turns at the last requested stop

C. LOOK can starve floors; SCAN cannot

D. There is none; they are two names for one policy

2. Car at floor 4 going UP, stops 7, 3, 9, 6 on floors 0–10. How many floors does LOOK travel?

A. 9

B. 11

C. 13

D. 16

3. Why is "always go to the nearest stop" a poor elevator policy?

A. It travels further than SCAN on every input

B. It can starve a far floor while nearer calls keep arriving

C. It needs a priority queue, which Java lacks

D. It cannot handle car calls

4. The building wants a rush-hour policy that parks idle cars at the lobby. With a Scheduler interface, what changes?

A. Elevator gets an if for rush hour in its step method

B. A new class implementing Scheduler; the dispatcher is given it, and Elevator is untouched

C. Direction gains a LOBBY value

D. Every existing scheduler gets a rushHour flag

5. Which pattern best describes the car's UP / DOWN / IDLE direction deciding which stops count as ahead?

A. Observer

B. State

C. Factory

D. Adapter

6. Floor displays and hall lanterns must update whenever a car moves. Which design fits?

A. Each display polls every car every millisecond

B. Displays subscribe to the car and are notified when its floor or direction changes (Observer)

C. The car holds a reference to every display class by name

D. The dispatcher prints to the displays from its call method

7. Two cars are both idle and both 3 floors from a hall call. The program must be deterministic. What do you do?

A. Pick one at random

B. Break the tie by a fixed rule, such as the lower car number

C. Send both cars

D. Wait for the next tick and decide then

8. Buttons on every floor can be pressed at the same moment from different threads. What keeps the stop sets consistent?

A. Nothing is needed; TreeSet is thread-safe

B. Assign under a lock, or have one dispatcher thread consume a queue of requests

C. Make the floor field volatile

D. Give every button its own copy of the stop set

9. A passenger inside car E1, going up at floor 6, presses 2. What happens?

A. The dispatcher may give floor 2 to another car

B. Floor 2 becomes E1's stop; E1 serves it after it turns

C. E1 turns at once and goes to 2

D. The press is ignored because 2 is behind the car

10. Why drive an elevator simulation with explicit ticks (STEP) instead of real time?

A. Java cannot sleep for less than a second

B. The output becomes a pure function of the input script, so it can be tested line by line

C. Ticks make the elevator faster

D. Real elevators move in ticks

ANSWER KEY · CHECKPOINT: ELEVATOR SYSTEM

1. B Both sweep: they serve every stop ahead before turning. SCAN carries on to the top or bottom floor first, LOOK turns as soon as nothing is ahead. Neither starves a floor, and arrival order is first come first served.

2. B LOOK visits 6, 7, 9 going up (5 floors), then turns and goes to 3 (6 floors): 11. SCAN would ride on to 10 first, making 13; first come first served (7, 3, 9, 6) makes 16.

3. B Nearest-first minimises the next move, so a stream of calls near the car keeps winning and a far floor may wait indefinitely. Sweeping policies bound the wait: every floor is reached within two sweeps. On many inputs nearest-first actually travels less, which is why it tempts.

4. B That is the point of the Strategy: a new policy is a new class, and the code that uses it (the dispatcher or the car's loop) depends only on the interface. Editing Elevator or every existing policy for each new rule is exactly what the open/closed principle warns against.

5. B Behaviour that depends on a mode the object is in is the State pattern; an enum is its simplest form, and a fuller design adds DoorsOpen and Maintenance as State classes. Observer would fit floor displays that follow the car.

6. B Observer lets the car announce a change without knowing who listens; adding a new kind of display is a new subscriber. Polling wastes work and lags; naming every display inside Elevator couples the car to the building's hardware.

7. B A fixed tie rule makes the same script print the same output every time, which is what a test, and a judge, can check. Randomness breaks reproducibility, sending both wastes a car, and waiting leaves a passenger standing.

8. B TreeSet is not thread-safe, and choosing a car then adding the stop must happen together. A lock around assignment, or a single thread draining a request queue, makes each assignment atomic. volatile only makes a single field's writes visible; it does not make a read-decide-write sequence atomic.

9. B A car call belongs to the car it was pressed in: the passenger is inside E1, so no other car can serve it. It becomes one of E1's stops and LOOK reaches it on the way down, after the stops still ahead.

10. B With ticks there is no clock and no thread timing in the result: the same commands always print the same lines. That is what makes the design testable in an interview and judgeable here. Real time would make the order of events depend on scheduling.
