Module 6 of 7

Movie ticket booking

Seat holds that expire, payments that confirm or release, pricing as a Strategy, and the race for one seat settled by versions, in Java.

A booking system sells something that runs out: one seat, one show, one buyer. The machine-coding problem (BookMyShow, in most rounds) is less about the many entities than about one of them, the ShowSeat, and the rules for moving it between available, held and booked.

You will build those rules as commands. Holds take all the chosen seats or none and expire after a fixed time; a payment books what is still held or releases it at once; a lapsed hold reads as available without any timer. Then the second program adds seat categories and a pricing rule chosen per show, a Strategy the booking code never looks inside. The checkpoint stages the race itself: two buyers read the same seat, and a version number decides which write wins.

Time is the script's own clock, so every case is reproducible. The full design, including the database locks, is in the movie ticket booking note.

CONTENTS

1	Movie ticket booking: holds, payments and the race	14 min
✓	Checkpoint: Movie ticket booking	25 min

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

LESSON 1 · 14 MIN

Movie ticket booking: holds, payments and the race

A booking system sells a contested resource: one seat, one show, one buyer. The machine-coding version tests three things: the **seat state**, the **hold that expires**, and keeping rules such as pricing out of the core. The movie ticket booking note has the whole design, including the database locks; here you build the in-memory model.

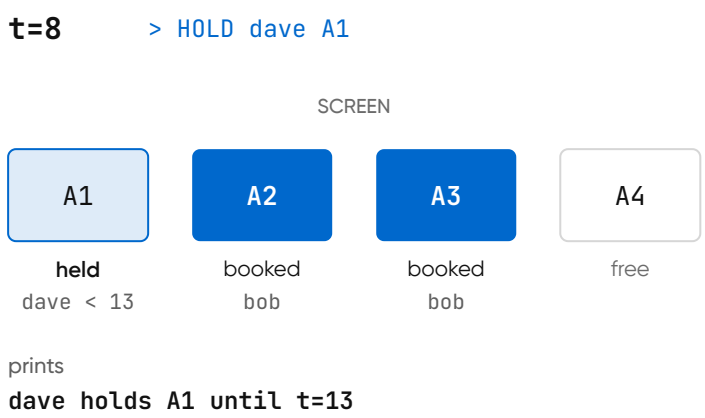
The classes

- **Seat**: physical, never changes (`A1` , row and number).
- **ShowSeat**: one seat *for one show*: `status` (AVAILABLE, HELD, BOOKED), `holder` , `heldUntil` . All the action is here, not on Seat.
- **Show**: the seat map, in layout order, with `hold` , `pay` and `show` .
- **PricingStrategy**: `price(seat)` , swapped per show: weekday, weekend, matinee.

The flow, as commands

1. `HOLD user seats…` : check every seat first; if all are free, hold them all until `now + hold` .
2. `PAY user ok` : the user's still-held seats become BOOKED.
3. `PAY user fail` : they are released at once, not when the hold runs out.
4. `TICK n` : time passes. A hold whose time is up is **read** as available; nothing has to fire.

Holds that expire: four seats, four buyers, one clock



1. Four seats for one show, all available; a hold lasts 5 minutes. The clock is the script's own: it moves only on TICK.
2. alice holds A1 and A2 together. Both are free, so both become HELD by her until t=5.
3. bob wants A2 and A3. A2 is alice's, so the hold is refused whole: bob gets nothing, not even the free A3.
4. Two minutes pass. Nothing is fired; the holds just have less time left.
5. alice's card is declined. Her seats are released at once, not at t=5, so nobody is kept waiting for a payment that has already failed.
6. bob tries again and gets A2 and A3 together, until t=7.
7. Another minute: it is t=3, and bob's hold is still good.
8. bob's payment succeeds while his hold is valid, so A2 and A3 become BOOKED for this show.
9. carol holds A1 until t=8. She never pays.
10. The clock reaches t=8. carol's hold is still stored on the seat, but it is read as lapsed: A1 shows as available without any timer.
11. dave asks for A1 and gets it: a lapsed hold is free to take, until t=13.

The traps

- **Half a hold.** Holding seats one at a time and failing on the third leaves two held. Check all, then change all.
- **A timer per hold.** Threads and schedulers make the output depend on timing. Store `heldUntil` and compare it with the clock whenever the seat is read.
- **Paying for a lapsed hold.** A payment confirms only what is *still* held by that user; if the hold lapsed, the seat may already be someone else's.
- **Status on Seat.** A seat is booked for one show, not for ever. Status lives on ShowSeat.

GAME · QUICK FIRE

What does the system answer?

Seats hold for 5 minutes. Each round is one moment of a show's evening.

1/4

t=0 alice holds A1 A2. At t=1 bob asks for A2 A3.

A bob cannot hold A2: held

B bob holds A3 until t=6

C bob holds A2 A3 until t=6

Pricing as a Strategy

The second exercise adds seat categories by row and a pricing rule chosen per show. The rule is one interface with one method; the booking code asks it for a price and never names a day of the week. A new rule, a festival surcharge say, is one more class. Prices are whole rupees here, so no floating point ever touches the money.

EXERCISE 1

Holds that expire

Model one show's seats with holds and payments.

Input. The first line is `rows cols hold`: rows are lettered `A`, `B`, … and seats numbered `1` to `cols` (`A1`, `A2`, …), and a hold lasts `hold` minutes. The clock starts at `t=0`. Then one command per line:

- `HOLD user s1 s2 …`: all or none. Check the seats in the order given and stop at the first problem: `user cannot hold S: no such seat`, `: booked` or `: held` (held by anyone, the same user included, and not lapsed). If every seat is free, hold them all until `now + hold` and print `user holds s1 s2 … until t=T` (seats in the order given).
- `PAY user ok` or `PAY user fail`: applies to every seat the user still holds (a hold is valid while `now < heldUntil`), taken in layout order. `ok` books them: `user booked A1 B2`. `fail` releases them at once: `user payment failed, released A1 B2`. With no such seats: `user has no active hold`.
- `TICK n`: the clock moves on `n` minutes. Prints nothing.
- `SHOW`: `t=N`, then one line per row such as `A: . H X .`, where `.` is available (a lapsed hold included), `H` held and `X` booked.

Seats in one command are distinct.

Example

```
1 3 5
HOLD alice A1 A2
HOLD bob A2 A3
PAY alice ok
SHOW
```

prints

```
alice holds A1 A2 until t=5
bob cannot hold A2: held
alice booked A1 A2
t=0
A: X X .
```

INPUT 1

```
1 4 5
HOLD alice A1 A2
HOLD bob A2 A3
TICK 2
PAY alice fail
HOLD bob A2 A3
TICK 1
PAY bob ok
HOLD carol A1
TICK 5
SHOW
HOLD dave A1
SHOW
```

OUTPUT 1

```
alice holds A1 A2 until t=5
bob cannot hold A2: held
alice payment failed, released A1 A2
bob holds A2 A3 until t=7
bob booked A2 A3
carol holds A1 until t=8
t=8
A: . X X .
dave holds A1 until t=13
t=8
A: H X X .
```

REFERENCE SOLUTION

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

public class Main {
    enum Status { AVAILABLE, HELD, BOOKED }

    /** One seat for one show: all the state lives here, not on the physical seat. */
    static class ShowSeat {
        final String id;
        Status status = Status.AVAILABLE;
        String holder;
        int heldUntil;

        ShowSeat(String id) { this.id = id; }

        // The expiry is read, never fired: a lapsed hold is simply available.
        boolean availableAt(int now) {
            return status == Status.AVAILABLE || (status == Status.HELD && now >= he
        }

        boolean heldBy(String user, int now) {
            return status == Status.HELD && now < heldUntil && holder.equals(user);
        }

        void release() { status = Status.AVAILABLE; holder = null; }
    }

    static class Show {
        final int holdMinutes;
        final int rows;
        final int cols;
        final Map<String, ShowSeat> seats = new LinkedHashMap<>(); // layout order:
        int now = 0;

        Show(int rows, int cols, int holdMinutes) {
            this.rows = rows; this.cols = cols; this.holdMinutes = holdMinutes;
            for (int r = 0; r < rows; r++)
                for (int c = 1; c <= cols; c++) {
                    String id = (char) ('A' + r) + String.valueOf(c);
                    seats.put(id, new ShowSeat(id));
                }
        }

        String hold(String user, List<String> ids) {
            // All or none: every seat is checked before any is changed.
            for (String id : ids) {
                ShowSeat s = seats.get(id);
                if (s == null) return user + " cannot hold " + id + ": no such seat"
                if (!s.availableAt(now)) return user + " cannot hold " + id + ": " +
```

```
            }
            for (String id : ids) {
                ShowSeat s = seats.get(id);
                s.status = Status.HELD;
                s.holder = user;
                s.heldUntil = now + holdMinutes;
            }
            return user + " holds " + String.join(" ", ids) + " until t=" + (now + h
        }

        String pay(String user, boolean ok) {
            List<String> mine = new ArrayList<>();
            for (ShowSeat s : seats.values()) {
                if (!s.heldBy(user, now)) continue;
                mine.add(s.id);
                if (ok) s.status = Status.BOOKED;
                else s.release();
            }
            if (mine.isEmpty()) return user + " has no active hold";
            return ok ? user + " booked " + String.join(" ", mine) : user + " paymer
        }

        String show() {
            StringBuilder b = new StringBuilder("t=" + now);
            List<ShowSeat> all = new ArrayList<>(seats.values());
            for (int r = 0; r < rows; r++) {
                b.append('\n').append((char) ('A' + r)).append(':');
                for (int c = 0; c < cols; c++) {
                    ShowSeat s = all.get(r * cols + c);
                    b.append(' ').append(s.status == Status.BOOKED ? 'X' : s.availab
                }
            }
            return b.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] first = br.readLine().trim().split("\\s+");
        Show show = new Show(Integer.parseInt(first[0]), Integer.parseInt(first[1]),
        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 "HOLD" -> out.append(show.hold(p[1], Arrays.asList(p).subList(2
                case "PAY" -> out.append(show.pay(p[1], p[2].equals("ok"))).append('
```

```
                case "TICK" -> show.now += Integer.parseInt(p[1]);
                case "SHOW" -> out.append(show.show()).append('\n');
                default -> { }
            }
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Pricing as a Strategy

Price seats by category with a pricing rule chosen per show.

Input. The first line is `rows cols` (seats `A1` … as before). The second line gives each row's category in order: `REGULAR`, `PREMIUM` or `RECLINER`, whose base prices are 200, 360 and 520 rupees. The show starts on the `WEEKDAY` rule. Then one command per line:

- `PRICING NAME`: switch the rule and print `pricing NAME`, or `unknown pricing NAME` (the rule stays). The rules: `WEEKDAY` is the base price, `WEEKEND` is base × 5 / 4, `MATINEE` is base × 3 / 4, in whole rupees.
- `QUOTE s1 s2 …`: `A1 250 + C2 450 = 700` at the current rule (a single seat: `A1 250 = 250`), or `no such seat S` for the first unknown seat.
- `BOOK user s1 s2 …`: all or none, checked in order: `user cannot book S: no such seat` or `: booked`. Otherwise `user booked s1 s2 … for TOTAL` at the current rule.
- `REVENUE`: `revenue N`, the sum of every booking at the price it was sold for.

Example

```
3 4
REGULAR REGULAR PREMIUM
QUOTE A1 C2
PRICING WEEKEND
QUOTE A1 C2
```

prints

```
A1 200 + C2 360 = 560
pricing WEEKEND
A1 250 + C2 450 = 700
```

INPUT 1

```
3 4
REGULAR REGULAR PREMIUM
QUOTE A1 C2
BOOK ann A1 C2
PRICING WEEKEND
QUOTE A1 C2
BOOK raj A2 C2
BOOK raj A2 C3
REVENUE
```

OUTPUT 1

```
A1 200 + C2 360 = 560
ann booked A1 C2 for 560
pricing WEEKEND
A1 250 + C2 450 = 700
raj cannot book C2: booked
raj booked A2 C3 for 700
revenue 1260
```

REFERENCE SOLUTION

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

public class Main {
    enum Category {
        REGULAR(200), PREMIUM(360), RECLINER(520);
        final int base;
        Category(int base) { this.base = base; }
    }

    // The Strategy: the booking code asks for a price and never names a day.
    interface PricingStrategy {
        String name();
        int price(Category c);
    }

    static class WeekdayPricing implements PricingStrategy {
        public String name() { return "WEEKDAY"; }
        public int price(Category c) { return c.base; }
    }

    static class WeekendPricing implements PricingStrategy {
        public String name() { return "WEEKEND"; }
        public int price(Category c) { return c.base * 5 / 4; }
    }

    static class MatineePricing implements PricingStrategy {
        public String name() { return "MATINEE"; }
        public int price(Category c) { return c.base * 3 / 4; }
    }

    static final Map<String, PricingStrategy> RULES = new HashMap<>();
    static {
        for (PricingStrategy p : List.of(new WeekdayPricing(), new WeekendPricing(),
    }

    static class Show {
        final Map<String, Category> seats = new LinkedHashMap<>();
        final Set<String> booked = new HashSet<>();
        PricingStrategy pricing = RULES.get("WEEKDAY");
        int revenue = 0;

        String setPricing(String name) {
            PricingStrategy p = RULES.get(name);
            if (p == null) return "unknown pricing " + name;
            pricing = p;
            return "pricing " + name;
        }
```

```
    String quote(List<String> ids) {
        StringJoiner parts = new StringJoiner(" + ");
        int total = 0;
        for (String id : ids) {
            Category c = seats.get(id);
            if (c == null) return "no such seat " + id;
            int price = pricing.price(c);
            parts.add(id + " " + price);
            total += price;
        }
        return parts + " = " + total;
    }

    String book(String user, List<String> ids) {
        int total = 0;
        for (String id : ids) { // all or none
            if (!seats.containsKey(id)) return user + " cannot book " + id + ":
            if (booked.contains(id)) return user + " cannot book " + id + ": boo
            total += pricing.price(seats.get(id));
        }
        booked.addAll(ids);
        revenue += total; // the price is fixed when the seat is sold
        return user + " booked " + String.join(" ", ids) + " for " + total;
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String[] size = br.readLine().trim().split("\\s+");
    String[] cats = br.readLine().trim().split("\\s+");
    int rows = Integer.parseInt(size[0]), cols = Integer.parseInt(size[1]);
    Show show = new Show();
    for (int r = 0; r < rows; r++)
        for (int c = 1; c <= cols; c++) show.seats.put((char) ('A' + r) + String
    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 "PRICING" -> show.setPricing(p[1]);
            case "QUOTE" -> show.quote(Arrays.asList(p).subList(1, p.length));
            case "BOOK" -> show.book(p[1], Arrays.asList(p).subList(2, p.length)
            case "REVENUE" -> "revenue " + show.revenue;
            default -> "unknown command " + p[0];
        };
        out.append(said).append('\n');
    }
```

```
        System.out.print(out);
    }
}
```

QUIZ

1. Where should a seat's status (available, held, booked) live?

A. On Seat, the physical seat

B. On ShowSeat, one per seat per show

C. On Booking only

D. On Cinema, as a map of every seat

2. bob asks to hold A2 and A3. A3 is free but A2 is held by alice. What should happen?

A. bob holds A3 only

B. Nothing changes: the hold is all or none

C. bob holds both and alice loses A2

D. bob waits until alice pays

3. How does a hold expire in the exercise's design?

A. A thread sleeps for the hold time, then releases the seat

B. heldUntil is stored on the seat, and a HELD seat past it reads as available

C. A cleanup job runs every minute and is required for correctness

D. The buyer's browser releases it

4. alice's card is declined while her seats are held until t=5. When can others take them?

A. At t=5, when the hold runs out

B. At once: a failed payment releases the hold

C. Never: declined seats are blocked for the show

D. Only after an admin frees them

ANSWER KEY · MOVIE TICKET BOOKING: HOLDS, PAYMENTS AND THE RACE

1. B A seat is booked for one show, not for ever: A1 at 6 pm and A1 at 9 pm are different things to sell. ShowSeat is that pair, so status, holder and expiry live there. Seat stays physical and never changes.

2. B A buyer who asked for two seats together does not want one of them, so a hold checks every seat first and changes none if any is taken. Taking A3 alone would leave a half-done booking to clean up.

3. B Storing the expiry and comparing it whenever the seat is read needs no timer and makes the behaviour depend only on the clock's value. A cleanup job can still tidy rows, but correctness never waits for it.

4. B The hold exists to cover the payment. Once the payment has failed there is nothing to wait for, so the seats go back immediately; keeping them held to t=5 just turns away other buyers.

CHECKPOINT · 25 MIN

Checkpoint: Movie ticket booking

This checkpoint covers the module: ShowSeat and its states, all-or-none holds that expire, payments that confirm or release, pricing as a Strategy, and how two buyers racing for one seat are kept apart.

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:

- Which object holds a seat's status, and why not Seat?
- What makes a lapsed hold read as available without a timer?
- What happens to held seats when a payment fails, and when it succeeds after the hold lapsed?
- How does a version number let exactly one of two buyers win?
- Where does a pricing rule live, so a new one changes no booking code?

The program is the race itself: buyers read a seat's version and then try to write, and a write with a stale version loses. Trace the example by hand first; the case most people miss is a buyer whose own earlier write made their read stale.

EXERCISE

Two buyers, one seat

Decide races for seats with **optimistic locking**: every seat has a version, every change of status raises it by one, and a hold is a conditional write that succeeds only if the version is still the one the buyer read.

Input. The first line lists the seat ids. Then one command per line:

- `READ user S`: the buyer reads the seat; remember the version they saw (a later read replaces it). Print `user reads S: STATUS vN`.
- `HOLD user S`: if the user never read `S`, print `user must read S first`. If the seat's version differs from the one they read, print `user lost S: version is N, read M`. If the versions match but the seat is not `AVAILABLE`, print `user cannot hold S: STATUS`. Otherwise it becomes `HELD` by the user: `user holds S vN` with the new version.

- `BOOK user S`: if the user holds it, it becomes `BOOKED`: `user booked S vN`.
- `RELEASE user S`: if the user holds it, it becomes `AVAILABLE`: `user released S vN`. For both, a seat the user does not hold prints `user does not hold S`.
- `SHOW`: every seat in the order listed: `A1 AVAILABLE v0, A2 BOOKED v4`.

A seat id that is not listed prints `no such seat S` for any command.

Example

```
A1 A2
READ alice A2
READ bob A2
HOLD alice A2
HOLD bob A2
```

prints

```
alice reads A2: AVAILABLE v0
bob reads A2: AVAILABLE v0
alice holds A2 v1
bob lost A2: version is 1, read 0
```

INPUT 1

```
A1 A2 A3
READ alice A2
READ bob A2
HOLD alice A2
HOLD bob A2
READ bob A2
HOLD bob A2
RELEASE alice A2
READ bob A2
HOLD bob A2
BOOK bob A2
SHOW
```

OUTPUT 1

```
alice reads A2: AVAILABLE v0
bob reads A2: AVAILABLE v0
alice holds A2 v1
bob lost A2: version is 1, read 0
bob reads A2: HELD v1
bob cannot hold A2: HELD
alice released A2 v2
bob reads A2: AVAILABLE v2
bob holds A2 v3
bob booked A2 v4
A1 AVAILABLE v0, A2 BOOKED v4, A3
AVAILABLE v0
```

REFERENCE SOLUTION

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

public class Main {
    enum Status { AVAILABLE, HELD, BOOKED }

    static class ShowSeat {
        final String id;
        Status status = Status.AVAILABLE;
        String holder;
        int version = 0; // raised by every change of status

        ShowSeat(String id) { this.id = id; }

        void change(Status to, String who) { status = to; holder = who; version++; }
    }

    static class Show {
        final Map<String, ShowSeat> seats = new LinkedHashMap<>();
        /** What each buyer last read: "user seat" → version. */
        final Map<String, Integer> reads = new HashMap<>();

        String read(String user, ShowSeat s) {
            reads.put(user + " " + s.id, s.version);
            return user + " reads " + s.id + ": " + s.status + " v" + s.version;
        }

        // The conditional write: UPDATE ... SET status = HELD, version = version +
        String hold(String user, ShowSeat s) {
            Integer seen = reads.get(user + " " + s.id);
            if (seen == null) return user + " must read " + s.id + " first";
            if (seen ≠ s.version) return user + " lost " + s.id + ": version is " +
            if (s.status ≠ Status.AVAILABLE) return user + " cannot hold " + s.id +
            s.change(Status.HELD, user);
            return user + " holds " + s.id + " v" + s.version;
        }

        String book(String user, ShowSeat s) {
            if (s.status ≠ Status.HELD || !s.holder.equals(user)) return user + " c
            s.change(Status.BOOKED, user);
            return user + " booked " + s.id + " v" + s.version;
        }

        String release(String user, ShowSeat s) {
            if (s.status ≠ Status.HELD || !s.holder.equals(user)) return user + " c
            s.change(Status.AVAILABLE, null);
            return user + " released " + s.id + " v" + s.version;
        }
```

```
        String show() {
            StringJoiner j = new StringJoiner(", ");
            for (ShowSeat s : seats.values()) j.add(s.id + " " + s.status + " v" + s
            return j.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Show show = new Show();
        for (String id : br.readLine().trim().split("\\s+")) show.seats.put(id, new
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            if (p[0].equals("SHOW")) { out.append(show.show()).append('\n'); continu
            ShowSeat s = show.seats.get(p[2]);
            if (s == null) { out.append("no such seat ").append(p[2]).append('\n');
            String said = switch (p[0]) {
                case "READ" -> show.read(p[1], s);
                case "HOLD" -> show.hold(p[1], s);
                case "BOOK" -> show.book(p[1], s);
                case "RELEASE" -> show.release(p[1], s);
                default -> "unknown command " + p[0];
            };
            out.append(said).append('\n');
        }
        System.out.print(out);
    }
}
```

CHECKPOINT QUESTIONS

1. alice and bob both read seat A2 at version 0. alice's conditional hold lands first. What does bob's `UPDATE … WHERE id = 'A2' AND version = 0` do?

A. Holds A2 for bob as well

B. Changes 0 rows, so bob knows he lost

C. Throws a deadlock error

D. Waits until alice pays

2. Why is a `synchronized` method in the booking service not enough to prevent double booking in production?

A. synchronized is too slow

B. The service runs on several servers, and a Java lock only guards one process

C. synchronized does not work on Map objects

D. It is enough; nothing else is needed

3. Which is a correct pessimistic way to hold a seat?

A. Read the seat, check it is free in code, then write HELD

B. `SELECT … FOR UPDATE` on the seat row, check, write HELD, commit, all in one short transaction

C. Write HELD first and check afterwards

D. Lock the whole cinema table for the duration of the payment

4. Where should the call to the payment gateway happen?

A. Inside the transaction that holds the seats

B. Outside any lock or transaction, after the hold is committed

C. Before the seats are held

D. Inside a synchronized block

5. A payment succeeds at t=9 but the hold lapsed at t=8. What is correct?

A. Book the seats anyway; the money arrived

B. Confirm only seats still held by this buyer; otherwise refund

C. Book the seats and cancel whoever holds them now

D. Extend the hold by the time the payment took

6. Which pattern fits "weekday, weekend and matinee prices, chosen per show"?

A. Singleton

B. Strategy

C. Observer

D. Builder

7. Mail, SMS and analytics all need to react when a booking is confirmed. Which design fits?

A. The booking service calls each one directly by name

B. They subscribe to a booking-confirmed event (Observer)

C. Each one polls the bookings table

D. The payment gateway notifies them

8. What does storing `heldUntil` on ShowSeat buy you, compared with a timer per hold?

A. Nothing; it is the same

B. Expiry needs no thread: a lapsed hold reads as available, and behaviour depends only on the clock's value

C. It makes holds last for ever

D. It lets the buyer extend the hold

9. Which database constraint is a backstop against two confirmed bookings of one seat for one show?

A. An index on the movie title

B. A unique constraint on (show, seat) in the booked-seats table

C. A foreign key from seat to cinema

D. A NOT NULL on the price

10. In optimistic locking, why does every change of status raise the version, not only a hold?

A. To count how many times the seat was sold

B. So any read taken before any change is detected as stale, whatever the change was

C. Because MySQL requires it

D. It does not need to; only holds matter

ANSWER KEY · CHECKPOINT: MOVIE TICKET BOOKING

1. B alice's write moved the version to 1, so bob's WHERE matches nothing and the update reports 0 rows changed. That count is the signal: exactly one of two racing writes can succeed, with no lock held while either user thinks.

2. B Two requests for the same seat can reach two different servers, each with its own lock. The decision must be made where every server sees it: in the database, with a conditional update, `SELECT … FOR UPDATE`, or a unique constraint.

3. B Locking the row before the check means no one else can read-and-write it in between; committing quickly keeps others waiting only briefly. Checking without a lock is the race itself, and holding a lock across a payment call keeps it for seconds.

4. B A gateway can take seconds; inside a transaction it holds locks the whole time. Hold first (short transaction), pay outside, then confirm or release in another short write. Paying before holding risks charging for a seat that is gone.

5. B After t=8 the seats may already be held or booked by someone else. Confirming only what this buyer still holds keeps one buyer per seat; a late payment for a lost seat is refunded.

6. B Each pricing rule is one class behind a `price(seat)` interface, and the show is given one. The booking code calls `price` and never names a day, so a new rule is a new class and nothing else changes.

7. B With Observer the booking announces one event and does not know who listens; adding push notifications is a new subscriber. Calling each by name couples the core to every channel, and polling lags and wastes queries.

8. B A timer per hold means threads, cancellation and timing-dependent results. A stored expiry is checked whenever the seat is read, so a lapsed hold is simply available, with no moving parts and a reproducible outcome.

9. B Even if application logic slipped, the database refuses a second row for the same show and seat. It is a backstop, not the main guard: the conditional hold should have stopped the second buyer long before.

10. B A buyer who read AVAILABLE v2 must not succeed after the seat was held and released (v4) and held again; only a version that moves on every change makes every old read stale. In the checkpoint's program, a buyer's own hold makes their own read stale too.

CodeKairo · Low-Level Design study plan · Module 6: Movie ticket booking. Solve the exercises online to track your progress and earn XP.