Module 1 of 7

From requirements to classes

Read a problem for nouns and verbs, give a shared fact its own class, keep each rule with its data — then code a library desk and a role system judged on behaviour.

Every machine-coding round starts the same way: a paragraph of requirements and an empty editor. This module is the method for the first ten minutes — which nouns become classes, which verbs become methods, when a link between two objects deserves a class of its own, and where each rule should live.

You will model a library desk (books, members and the loans between them) and a role-based permission system (roles that inherit from roles, with cycles refused), and the checkpoint adds a course registry with seats and a waitlist. Each program reads a script of commands and prints what the system answers; hidden cases run the refusals, which is where a weak model breaks first.

It pairs with the notes Introduction to Low-Level Design and UML class diagrams.

CONTENTS

1	From requirements to classes	12 min
✓	Checkpoint — From requirements to classes	30 min

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

LESSON 1 · 12 MIN

From requirements to classes

A machine-coding round opens with a paragraph of requirements and a blank editor. The first ten minutes decide the next fifty: the classes you pick are the ones you will be coding against. This lesson is the method, and the two programs after it are judged on what your classes do. The theory is in Introduction to Low-Level Design and UML class diagrams.

Read the requirements for nouns and verbs

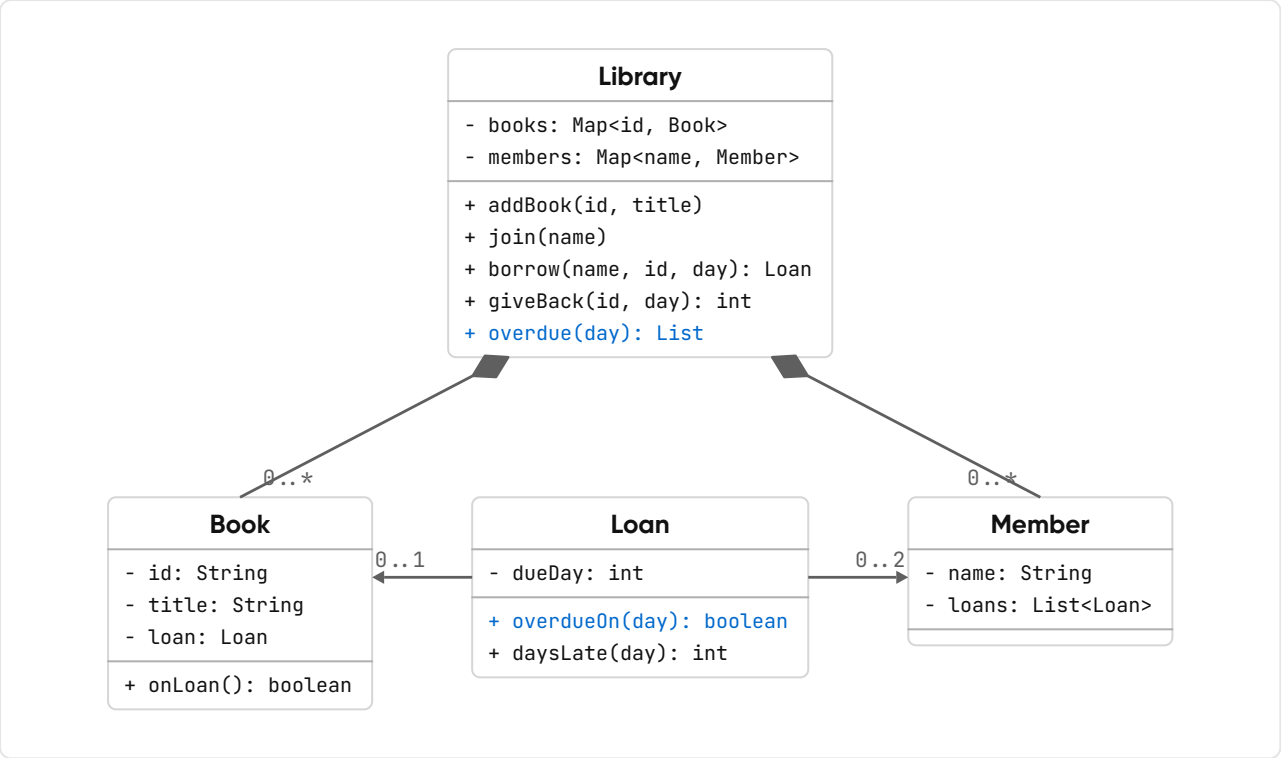
> A library lends **books** to **members**. A member may **borrow** up to two books for 14 days and **return** them; the desk can list the books that are **overdue**.

- **Nouns that hold state become classes**: Book (id, title), Member (name, loans), Library (everything on the shelf).
- **Verbs become methods**: `borrow`, `giveBack`, `overdue`.
- **Facts become fields**: the 14-day period and the limit of two are constants, not classes.
- **Actors stay outside**: the person at the desk calls `borrow`; nothing about them is stored, so no Librarian class.

Give a shared fact its own class

The due day belongs to neither the book nor the member: it belongs to *this book lent to this member*. That is a **Loan**, an association class.

Library desk: from the brief's nouns to four classes



1. Library is the class the desk calls: it finds books and members by key and offers the verbs of the brief — add, join, borrow, give back, list overdue.
2. The library owns its books — composition, a filled diamond at the owner. A Book knows its id and title, and whether it is out right now.
3. Members are owned the same way. A Member holds its own loans, which is what the two-book limit is counted from.
4. The due day belongs to neither side, so the link gets a class: a Loan points at one book and one member. A book has at most one loan, a member at most two.
5. Each rule sits with its data: overdueOn and daysLate read only the due day, so they live on Loan, and Library.overdue just asks each loan.

- **Composition**: the library owns its books and members.
- **Association**: a Loan points at one Book and one Member; a book has 0..1 loans, a member 0..2.
- **Put each rule with its data**: `overdueOn(day)` reads only the due day, so it lives on Loan. The Library finds objects and calls them; it is not a god class holding every rule.

Run it

Every command is a method call, and every refusal is a rule some object owns.

The library desk running the exercise's first case

books

B1	Dune	on the shelf
B2	Emma	bo · due 17

members

ada · 0/2 books
bo · 1/2 books

```
OVERDUE 18
→ overdue: B2
```

1. The state after the ADD and JOIN lines: two books on the shelf, two members holding nothing. Every later command is one method call on this model.
2. ada borrows B1 on day 1: a new Loan links the book and the member, due on day 15, and both sides point at it.
3. bo asks for B1, but the book already holds a loan. The Book refuses with an exception and main prints it after "error:"; nothing changes.
4. bo borrows B2 on day 3: a new Loan links the book and the member, due on day 17, and both sides point at it.
5. OVERDUE 16 asks every loan whether day 16 is past its due day: B1. Nothing is stored, so tomorrow's answer is never stale.
6. B1 comes back on day 20. The loan ends, the member is down a book, and Loan.daysLate works out the 5 days late.
7. OVERDUE 18 asks every loan whether day 18 is past its due day: B2. Nothing is stored, so tomorrow's answer is never stale.

- Refusals are **exceptions** with a message, caught once in `main`, so no method returns magic strings.
- Output order never depends on a `HashMap`: books live in a `TreeMap` sorted by id.

The traps

- **Storing derived state**: an `overdue` flag is wrong tomorrow; compute it from the due day.
- **Strings for objects**: `Map<String, String> loans` from book id to member name cannot hold a due day; the first new requirement breaks it.

- **One class doing everything**: if every method of your design is on `Library`, the entities are only records.

EXERCISE 1

Library desk

Model a small library with `Book`, `Member`, `Loan` and a `Library` that finds them. The program reads commands until end of input, one per line, and prints one line for each.

Command	Prints
`ADD <id> <title…>`	`added <id>` (the title is the rest of the line)
`JOIN <name>`	`welcome <name>`
`BORROW <name> <id> <day>`	`<name> borrowed <id> "<title>", due day <day + 14>`
`RETURN <id> <day>`	`<id> returned on time`, or `<id> returned <n> days late` (`1 day late`)
`OVERDUE <day>`	`overdue: <ids>` — books on loan whose due day is before `<day>`, in id order as text (`B1 B10 B2`) — or `overdue: none`

A member may hold **at most two** books. Refusals print `error: <reason>`, checked in this order:

- `ADD` of an existing id: `book <id> already exists`; `JOIN` of an existing name: `member <name> already exists`
- `BORROW`: `no member <name>`, then `no book <id>`, then `<id> is on loan`, then `<name> already has 2 books`
- `RETURN`: `no book <id>`, then `<id> is not on loan`
- any other command: `unknown command <WORD>`

A book returned on its due day is on time; it is overdue from the day after.

INPUT 1

```
ADD B1 Dune
ADD B2 Emma
JOIN ada
JOIN bo
BORROW ada B1 1
BORROW bo B1 2
BORROW bo B2 3
OVERDUE 16
RETURN B1 20
RETURN B1 21
OVERDUE 18
```

OUTPUT 1

```
added B1
added B2
welcome ada
welcome bo
ada borrowed B1 "Dune", due day 15
error: B1 is on loan
bo borrowed B2 "Emma", due day 17
overdue: B1
B1 returned 5 days late
error: B1 is not on loan
overdue: B2
```

REFERENCE SOLUTION

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

public class Main {
    static final int LOAN_DAYS = 14;
    static final int MAX_LOANS = 2;

    /** A rule the library refuses to break; the message is what the desk prints aft
    static class LibraryException extends RuntimeException {
        LibraryException(String message) { super(message); }
    }

    static class Book {
        final String id;
        final String title;
        Loan loan; // null while the book is on the shelf

        Book(String id, String title) { this.id = id; this.title = title; }
        boolean onLoan() { return loan =≠ null; }
    }

    static class Member {
        final String name;
        final List<Loan> loans = new ArrayList<>();

        Member(String name) { this.name = name; }
    }

    /** The link between a book and a member, holding the fact that belongs to neith
    static class Loan {
        final Book book;
        final Member member;
        final int dueDay;

        Loan(Book book, Member member, int dueDay) { this.book = book; this.member =
        boolean overdueOn(int day) { return day > dueDay; }
        int daysLate(int day) { return Math.max(0, day - dueDay); }
    }

    static class Library {
        private final Map<String, Book> books = new TreeMap<>(); // sorted by id, so
        private final Map<String, Member> members = new HashMap<>();

        void addBook(String id, String title) {
            if (books.containsKey(id)) throw new LibraryException("book " + id + " a
            books.put(id, new Book(id, title));
        }

        void join(String name) {
```

```
        if (members.containsKey(name)) throw new LibraryException("member " + na
        members.put(name, new Member(name));
    }

    Loan borrow(String name, String id, int day) {
        Member member = members.get(name);
        if (member == null) throw new LibraryException("no member " + name);
        Book book = book(id);
        if (book.onLoan()) throw new LibraryException(id + " is on loan");
        if (member.loans.size() >= MAX_LOANS) throw new LibraryException(name +
        Loan loan = new Loan(book, member, day + LOAN_DAYS);
        book.loan = loan;
        member.loans.add(loan);
        return loan;
    }

    /** Ends the loan and answers how many days late the book came back. */
    int giveBack(String id, int day) {
        Book book = book(id);
        if (!book.onLoan()) throw new LibraryException(id + " is not on loan");
        Loan loan = book.loan;
        book.loan = null;
        loan.member.loans.remove(loan);
        return loan.daysLate(day);
    }

    List<String> overdue(int day) {
        List<String> ids = new ArrayList<>();
        for (Book book : books.values()) {
            if (book.onLoan() && book.loan.overdueOn(day)) ids.add(book.id);
        }
        return ids;
    }

    private Book book(String id) {
        Book book = books.get(id);
        if (book == null) throw new LibraryException("no book " + id);
        return book;
    }
}

static String handle(Library library, String line) {
    String[] p = line.split("\\s+");
    switch (p[0]) {
        case "ADD": {
            String[] parts = line.split("\\s+", 3);
            library.addBook(parts[1], parts[2]);
            return "added " + parts[1];
        }
```

```
            case "JOIN":
                library.join(p[1]);
                return "welcome " + p[1];
            case "BORROW": {
                Loan loan = library.borrow(p[1], p[2], Integer.parseInt(p[3]));
                return p[1] + " borrowed " + loan.book.id + " \"" + loan.book.title
            }
            case "RETURN": {
                int late = library.giveBack(p[1], Integer.parseInt(p[2]));
                if (late == 0) return p[1] + " returned on time";
                return p[1] + " returned " + late + (late == 1 ? " day" : " days") +
            }
            case "OVERDUE": {
                List<String> ids = library.overdue(Integer.parseInt(p[1]));
                return "overdue: " + (ids.isEmpty() ? "none" : String.join(" ", ids)
            }
            default:
                return "error: unknown command " + p[0];
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Library library = new Library();
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String reply;
            try {
                reply = handle(library, line);
            } catch (LibraryException e) {
                reply = "error: " + e.getMessage();
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Roles and permissions

Model role-based access control: a `Role` holds permissions and may **inherit** other roles; a `User` holds roles; a user can do whatever any of their roles — or any role those inherit, at any depth —

allows. Inheritance is live: a link added after an assignment counts at once.

Command	Prints
`ROLE <role> [perm…]`	`role <role>: <perms sorted>` or `role <role>: none`
`INHERIT <child> <parent>`	`<child> inherits <parent>`
`USER <name>`	`user <name>`
`ASSIGN <user> <role>`	`<user> is now <role>`
`REVOKE <user> <role>`	`<user> is no longer <role>`
`CAN <user> <perm>`	`yes` or `no`
`PERMS <user>`	`<user>: <every permission, sorted>` or `<user>: none`

Refusals print `error: <reason>`:

- `role <r> already exists`, `user <u> already exists`
- `no role <r>` (checking the child before the parent, the user before the role), `no user <u>`
- `INHERIT`: `<child> already inherits <parent>` for an existing direct link, then `cycle` when the parent is the child or already inherits from it
- `ASSIGN`: `<user> is already <role>`; `REVOKE`: `<user> is not <role>` (also for a role that does not exist)
- any other command: `unknown command <WORD>`

INPUT 1

```
ROLE viewer read
ROLE editor edit
INHERIT editor viewer
USER ana
ASSIGN ana editor
CAN ana read
CAN ana delete
PERMS ana
```

OUTPUT 1

```
role viewer: read
role editor: edit
editor inherits viewer
user ana
ana is now editor
yes
no
ana: edit read
```

REFERENCE SOLUTION

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

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

    static class Role {
        final String name;
        final Set<String> permissions = new TreeSet<>();
        final List<Role> parents = new ArrayList<>();

        Role(String name) { this.name = name; }

        /** True when this role reaches the other one through its parents. */
        boolean inheritsFrom(Role other) {
            for (Role parent : parents) {
                if (parent == other || parent.inheritsFrom(other)) return true;
            }
            return false;
        }

        /** Its own permissions and every ancestor's. A cycle is refused when it is
        void collect(Set<String> into) {
            into.addAll(permissions);
            for (Role parent : parents) parent.collect(into);
        }
    }

    static class User {
        final String name;
        final Set<Role> roles = new LinkedHashSet<>();

        User(String name) { this.name = name; }

        Set<String> permissions() {
            Set<String> all = new TreeSet<>();
            for (Role role : roles) role.collect(all);
            return all;
        }

        boolean can(String permission) { return permissions().contains(permission);
    }

    static class AccessControl {
        private final Map<String, Role> roles = new HashMap<>();
        private final Map<String, User> users = new HashMap<>();
```

```
    Role defineRole(String name, List<String> permissions) {
        if (roles.containsKey(name)) throw new AccessException("role " + name +
        Role role = new Role(name);
        role.permissions.addAll(permissions);
        roles.put(name, role);
        return role;
    }

    void inherit(String childName, String parentName) {
        Role child = role(childName);
        Role parent = role(parentName);
        if (child.parents.contains(parent)) throw new AccessException(childName
        if (parent == child || parent.inheritsFrom(child)) throw new AccessExcep
        child.parents.add(parent);
    }

    void addUser(String name) {
        if (users.containsKey(name)) throw new AccessException("user " + name +
        users.put(name, new User(name));
    }

    void assign(String userName, String roleName) {
        User user = user(userName);
        Role role = role(roleName);
        if (!user.roles.add(role)) throw new AccessException(userName + " is alr
    }

    void revoke(String userName, String roleName) {
        User user = user(userName);
        Role role = roles.get(roleName);
        if (role == null || !user.roles.remove(role)) throw new AccessException(
    }

    User user(String name) {
        User user = users.get(name);
        if (user == null) throw new AccessException("no user " + name);
        return user;
    }

    private Role role(String name) {
        Role role = roles.get(name);
        if (role == null) throw new AccessException("no role " + name);
        return role;
    }
}

static String list(Set<String> items) {
    return items.isEmpty() ? "none" : String.join(" ", items);
}
```

```
    static String handle(AccessControl acl, String[] p) {
        switch (p[0]) {
            case "ROLE": {
                Role role = acl.defineRole(p[1], Arrays.asList(p).subList(2, p.lengt
                return "role " + role.name + ": " + list(role.permissions);
            }
            case "INHERIT":
                acl.inherit(p[1], p[2]);
                return p[1] + " inherits " + p[2];
            case "USER":
                acl.addUser(p[1]);
                return "user " + p[1];
            case "ASSIGN":
                acl.assign(p[1], p[2]);
                return p[1] + " is now " + p[2];
            case "REVOKE":
                acl.revoke(p[1], p[2]);
                return p[1] + " is no longer " + p[2];
            case "CAN":
                return acl.user(p[1]).can(p[2]) ? "yes" : "no";
            case "PERMS":
                return p[1] + ": " + list(acl.user(p[1]).permissions());
            default:
                return "error: unknown command " + p[0];
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        AccessControl acl = new AccessControl();
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) ≠ null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String reply;
            try {
                reply = handle(acl, line.split("\\s+"));
            } catch (AccessException e) {
                reply = "error: " + e.getMessage();
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

QUIZ

1. A requirement says: *"a member borrows a book for 14 days."* Where should the due date live?

A. A field on Book

B. A field on Member

C. A Loan class that links one Book and one Member

D. A Map<String, Integer> inside Library from book id to due day

2. Which phrase from a parking-lot brief should **not** become a class?

A. "each level has spots"

B. "the attendant hands the driver a ticket"

C. "a ticket records the entry time"

D. "spots come in three sizes"

3. `Library.isOverdue(Loan loan, int day)` reads only `loan.dueDay`. What is wrong with it?

A. Nothing — the Library coordinates everything

B. It should be static

C. It sits on the wrong class: the rule reads only Loan's data, so it belongs on Loan

D. It should store the answer in a field so it is not recomputed

4. A library **owns** its books: when a library object is discarded, its books go with it. Which UML relation is that?

A. Inheritance

B. Composition (filled diamond)

C. Aggregation (hollow diamond)

D. Dependency (dashed arrow)

ANSWER KEY · FROM REQUIREMENTS TO CLASSES

1. C The due date is a fact of *this book lent to this member*, so it belongs to the link itself: an association class, **Loan**. On Book it cannot say who has it; on Member it needs one per book; a map of ids to numbers works until the next requirement (fines, renewals) needs more than one fact about the loan.

2. B The attendant and the driver are actors: they call the system's methods and nothing about them is stored. Level, Spot and Ticket hold state, and the sizes become an enum the spot holds.

3. C Put a method on the class that owns the data it needs. A method that reaches into another object's fields to decide something is the first step to a god class; `loan.overdueOn(day)` keeps the rule next to the data. Storing the answer is worse still — it goes stale the next day.

4. B Composition: the part's life is bound to the whole. Aggregation is a looser "has" where the part lives on its own (a team and its players); inheritance is "is a"; a dependency is a use, such as a method parameter.

CHECKPOINT · 30 MIN

Checkpoint — From requirements to classes

This checkpoint covers the whole module: reading a brief for nouns and verbs, association classes, composition versus association, where a rule should live, and keeping the model free of I/O.

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.

The program is a course registry with seats and a waitlist. Before you code, sketch it:

- Which classes hold state? Which one only finds objects and calls them?
- Which class decides between a seat and the waitlist, and who moves in when a seat frees up?
- Which collection keeps a student's courses in code order?

Hidden cases drop students from the waitlist as well as from seats, enrol someone twice and name students and courses that do not exist — the refusals are part of the design, so give each one a single home.

EXERCISE

Course registry with a waitlist

Model a course registry: `Course` (a code and a number of seats, the students enrolled and a waitlist, both in arrival order), `Student`, and a `Registry` that finds them. Commands, one per line until end of input:

Command	Prints
`COURSE <code> <seats>`	`course <code>, capacity <seats>`
`STUDENT <name>`	`student <name>`
`ENROL <name> <code>`	`<name> enrolled in <code>` when a seat is free, else `<name> waitlisted for <code> at position <k>`
`DROP <name> <code>`	from the waitlist: `<name> left the waitlist for <code>`; from a seat: `<name> dropped <code>`, and when someone was waiting, `<name> dropped <code>, <first waiting> moves in`
`ROSTER <code>`	`<code>: <enrolled> | waitlist: <waiting>`, each list space-separated in arrival order or `none`
`COURSES <name>`	`<name>: <codes enrolled, sorted>` or `<name>: none` (waitlists do not count)

Refusals print `error: <reason>`: `course <code> already exists`, `student <name> already exists`, `no student <name>` (checked first), `no course <code>`, `<name> is already in <code>` (enrolled or waiting), `<name> is not in <code>`, and `unknown command <WORD>`.

INPUT 1

```
COURSE CS101 2
STUDENT ann
STUDENT ben
STUDENT cal
ENROL ann CS101
ENROL ben CS101
ENROL cal CS101
ROSTER CS101
DROP ann CS101
ROSTER CS101
COURSES cal
```

OUTPUT 1

```
course CS101, capacity 2
student ann
student ben
student cal
ann enrolled in CS101
ben enrolled in CS101
cal waitlisted for CS101 at position 1
CS101: ann ben | waitlist: cal
ann dropped CS101, cal moves in
CS101: ben cal | waitlist: none
cal: CS101
```

REFERENCE SOLUTION

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

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

    static class Student {
        final String name;
        Student(String name) { this.name = name; }
    }

    static class Course {
        final String code;
        final int capacity;
        final List<Student> enrolled = new ArrayList<>();
        final List<Student> waitlist = new ArrayList<>();

        Course(String code, int capacity) { this.code = code; this.capacity = capaci

        boolean has(Student s) { return enrolled.contains(s) || waitlist.contains(s)

        /** 0 when the student took a seat, otherwise their place in the waitlist. *
        int enrol(Student s) {
            if (enrolled.size() < capacity) {
                enrolled.add(s);
                return 0;
            }
            waitlist.add(s);
            return waitlist.size();
        }

        /** Frees the student's seat; answers who moved in from the waitlist, or nul
        Student drop(Student s) {
            enrolled.remove(s);
            if (waitlist.isEmpty()) return null;
            Student next = waitlist.remove(0);
            enrolled.add(next);
            return next;
        }
    }

    static class Registry {
        private final Map<String, Course> courses = new TreeMap<>(); // sorted by co
        private final Map<String, Student> students = new HashMap<>();

        Course addCourse(String code, int capacity) {
            if (courses.containsKey(code)) throw new RegistryException("course " + c
```

```
        Course c = new Course(code, capacity);
        courses.put(code, c);
        return c;
    }

    void addStudent(String name) {
        if (students.containsKey(name)) throw new RegistryException("student " +
        students.put(name, new Student(name));
    }

    String enrol(String name, String code) {
        Student s = student(name);
        Course c = course(code);
        if (c.has(s)) throw new RegistryException(name + " is already in " + coc
        int place = c.enrol(s);
        return place == 0 ? name + " enrolled in " + code : name + " waitlisted
    }

    String drop(String name, String code) {
        Student s = student(name);
        Course c = course(code);
        if (c.waitlist.remove(s)) return name + " left the waitlist for " + code
        if (!c.enrolled.contains(s)) throw new RegistryException(name + " is not
        Student next = c.drop(s);
        return name + " dropped " + code + (next == null ? "" : ", " + next.name
    }

    String roster(String code) {
        Course c = course(code);
        return code + ": " + names(c.enrolled) + " | waitlist: " + names(c.waitl
    }

    String coursesOf(String name) {
        Student s = student(name);
        List<String> codes = new ArrayList<>();
        for (Course c : courses.values()) if (c.enrolled.contains(s)) codes.add(
        return name + ": " + (codes.isEmpty() ? "none" : String.join(" ", codes)
    }

    private static String names(List<Student> list) {
        if (list.isEmpty()) return "none";
        StringBuilder sb = new StringBuilder();
        for (Student s : list) sb.append(sb.length() == 0 ? "" : " ").append(s.r
        return sb.toString();
    }

    private Student student(String name) {
        Student s = students.get(name);
        if (s == null) throw new RegistryException("no student " + name);
```

```
            return s;
        }

        private Course course(String code) {
            Course c = courses.get(code);
            if (c == null) throw new RegistryException("no course " + code);
            return c;
        }
    }

    static String handle(Registry registry, String[] p) {
        switch (p[0]) {
            case "COURSE": {
                Course c = registry.addCourse(p[1], Integer.parseInt(p[2]));
                return "course " + c.code + ", capacity " + c.capacity;
            }
            case "STUDENT":
                registry.addStudent(p[1]);
                return "student " + p[1];
            case "ENROL":
                return registry.enrol(p[1], p[2]);
            case "DROP":
                return registry.drop(p[1], p[2]);
            case "ROSTER":
                return registry.roster(p[1]);
            case "COURSES":
                return registry.coursesOf(p[1]);
            default:
                return "error: unknown command " + p[0];
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Registry registry = new Registry();
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String reply;
            try {
                reply = handle(registry, line.split("\\s+"));
            } catch (RegistryException e) {
                reply = "error: " + e.getMessage();
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
```

```
    }
}
```

CHECKPOINT QUESTIONS

1. In a hotel booking brief, which pair is best modelled with an association class?

A. Hotel and Room

B. Guest and Room, for a stay with check-in and check-out dates

C. Room and RoomType

D. Hotel and Address

2. Your design stores `Map<String, String> loans` from book id to member name. A new requirement adds fines by days late. What breaks?

A. Nothing, a fine is computed from the map

B. The map has nowhere to keep the due day, so a Loan object with its own fields is needed

C. Only the output format

D. The HashMap must become a TreeMap

3. Why should `Course.enrol` decide between a seat and the waitlist, rather than `Registry.enrol`?

A. Because Registry must stay static

B. Because the rule reads only the course's capacity and lists — it is the course's own data

C. Because Java forbids it

D. Because the waitlist must be public

4. Which of these is a sign of a **god class**?

A. A class with an enum field

B. A class whose methods are all one line

C. One class holding every map and every rule while the others are only data holders

D. A class that throws exceptions

5. A role-based access system lets a role inherit another. Which check stops `INHERIT a c` when `c` already inherits from `a`?

A. `a.parents.contains(c)`

B. `c == a || c.inheritsFrom(a)` — the new parent must not already reach the child

C. `a.inheritsFrom(c)`

D. Sorting the roles by name

6. What is the main reason a refusal is thrown as an exception and printed once in `main`, rather than each method printing its own message?

A. Exceptions are faster than strings

B. The model stays free of I/O, so the same classes serve a console, an API or a test

C. Java requires it

D. It makes output order random

7. Why does the library keep its books in a `TreeMap` instead of a `HashMap`?

A. TreeMap lookups are O(1)

B. HashMap cannot hold strings

C. OVERDUE must print in a fixed order, and a HashMap's iteration order is unspecified

D. TreeMap uses less memory

8. An interviewer asks: *"what if a member can reserve a book that is on loan?"* What change does a good model absorb most cleanly?

A. A boolean `reserved` on Member

B. A new Reservation class linking Member and Book, with the book keeping a queue of them

C. Renaming Loan to Reservation

D. A static list of names in Library

9. A requirement says *"a member can borrow at most two books"*. How should the 2 appear in code?

A. As the literal 2 wherever it is checked

B. As a named constant (or a configurable policy) read in one place

C. As a class Two

D. As a field on every Book

10. In the course registry, a waitlisted student drops the course. Who moves in?

A. The first student on the waitlist

B. Nobody — no seat was freed

C. The last student on the waitlist

D. The dropped student keeps the position

ANSWER KEY · CHECKPOINT — FROM REQUIREMENTS TO CLASSES

1. B A stay joins one guest to one room and carries facts of its own — the dates, the price agreed. That is a **Booking**, an association class. A hotel composes its rooms, a room type is an enum or a class the room refers to, and an address is a value the hotel holds.

2. B Strings standing in for objects hold exactly one fact. The moment a link needs a second fact (a due day, a fine, a renewal count), the design needs a real Loan. Modelling the link as a class from the start costs nothing.

3. B Each rule lives with its data. The registry finds the student and the course and asks the course; then a second entry point (an admin screen, a bulk import) gets the same rule for free instead of copying it.

4. C A god class owns everything, so every change touches it and nothing can be tested alone. The cure is to move each rule onto the entity whose data it reads, leaving the coordinator to find objects and call them.

5. B Adding `c` as a parent of `a` closes a loop exactly when `c` already reaches `a` (or is `a`). Checking before the link is added keeps every later walk of the parents finite.

6. B Business classes that print are tied to one front end. Throwing a typed exception with a reason keeps the rules testable and lets each caller decide how to report them.

7. C A `HashMap` iterates in an order that depends on hashes and capacity, so output built from it can differ between runs or JDKs. A `TreeMap` iterates in key order (O(log n) lookups), which is what a judged listing needs.

8. B A reservation is another link between a member and a book with facts of its own (when, the order of the queue), so it is a new association class. The existing Loan stays as it is — the change is additive.

9. B A rule's number is a constant with a name, checked in one method. When the limit changes, or becomes per-membership, there is one place to change — the first step towards a policy object if it grows.

10. B Leaving the waitlist frees no seat, so the roster is unchanged; only a student giving up a seat promotes the first one waiting. Writing the two cases separately is what the hidden case checks.

CodeKairo · Low-Level Design study plan · Module 1: From requirements to classes. Solve the exercises online to track your progress and earn XP.