

🎓 Smart University Management System

📚 The Scenario

Congratulations! You've been hired as a software architect for **TechVerse University**, a cutting-edge institution that's going fully digital. The university needs a comprehensive management system to handle students, faculty, courses, departments, resources, and campus facilities. Your job is to design and implement a robust, scalable system that can grow with the university!

🎯 Your Mission

Build a complete university ecosystem that manages academic operations, from student enrollment to course delivery, faculty management, library resources, and campus facilities. The system must be flexible enough to adapt to different types of users, courses, and scenarios while maintaining clean, professional code.

Core Requirements & OOP Concepts

① ENCAPSULATION

Protect sensitive personal and academic data!

Requirements:

- Student GPA, personal info (ID, email, phone) must be private
- Salary information for faculty members should be hidden
- Course enrollment capacity should be controlled
- Financial data (tuition fees, transactions) must be protected
- Only authorized methods can modify sensitive data

Example Scenario:

```
x student.gpa = 4.0 // Direct access not allowed!
✓ student.updateGrade("CS101", "A") // Proper grade update
✓ student.calculateGPA() // Controlled GPA calculation
x professor.salary = 100000 // Cannot directly change
✓ hr.adjustSalary(professor, 100000) // Through HR system
```

② INHERITANCE

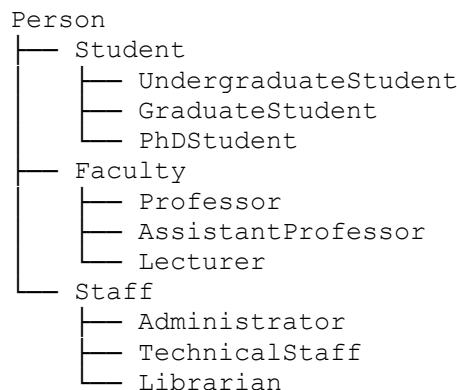
Create a hierarchical structure for university members!

Requirements:

- Base **Person** class with common attributes (name, ID, contact info)
- Specialized classes:
 - **Student** (**undergraduate**, **graduate** students)
 - UndergraduateStudent**
 - GraduateStudent** (**thesis**, **advisor**)
 - PhDStudent** (**dissertation**, **research**)
 - **Faculty** (**teaching** staff)
 - Professor**
 - AssistantProfessor**
 - Lecturer**
 - **Staff** (**administrative**)

```
Administrator  
TechnicalStaff  
Librarian
```

Example Hierarchy:



③ POLYMORPHISM

Different people interact with the system differently!

Requirements:

- Override `getRole()` method for each person type
- Override `calculateWorkload()`:
 - Students: based on credit hours
 - Professors: based on courses taught + research
 - Staff: based on assigned tasks
- Override `accessLibrary()`:
 - Students: borrow limit of 5 books
 - Faculty: borrow limit of 20 books
 - PhD students: extended borrowing period
- Override `displayDashboard()` to show role-specific information

Example:

```
List<Person> universityMembers = [student, professor, librarian]
for (person : universityMembers) {
    person.displayDashboard() // Each shows different info!
    person.getRole() // Returns different role types
}
```

④ ABSTRACTION

Define contracts without implementation details!

Requirements:

- Abstract `Person` class with abstract methods:
 - `register()`
 - `login()`
 - `getPermissions()`
 - `calculateFees()` or `calculateSalary()`
- Abstract `Course` class with methods:
 - `calculateFinalGrade()`
 - `checkPrerequisites()`
 - `generateSyllabus()`
- Force subclasses to implement their own logic
- No generic "Person" or "Course" objects can be created directly

5 INTERFACE

Multiple capabilities for different entities!

Requirements:

Teachable Interface:

- teach(Course course)
- assignGrades()
- holdOfficeHours()
- Implemented by: Professor, Lecturer, Teaching Assistants

Enrollable Interface:

- enrollInCourse(Course course)
- dropCourse(Course course)
- viewSchedule()
- Implemented by: All student types

Researchable Interface:

- publishPaper()
- conductResearch()
- applyForGrant()
- Implemented by: PhD Students, Professors

Payable Interface:

- processPayment()
- generateInvoice()
- getFinancialSummary()
- Implemented by: Students (tuition), Faculty (salary), Staff (salary)

Example:

Professor implements Teachable, Researchable, Payable
PhDStudent implements Enrollable, Researchable, Payable
UndergraduateStudent implements Enrollable, Payable

6 COMPOSITION (HAS-A, strong ownership)

Strong ownership relationships!

Requirements:

- Student **HAS-A** Transcript (can't exist without student)
- Student **HAS-A** EnrollmentRecord (dies with student)
- Course **HAS-A** Syllabus (part of the course)
- Professor **HAS-A** ResearchProfile (belongs to professor)
- Department **HAS-A** Budget (integral part of department)

Key Concept: When the owner is deleted, the owned object is also deleted!

Example:

```
class Student {  
    private Transcript transcript; // Created with student  
    private EnrollmentRecord record;  
  
    Student() {  
        transcript = new Transcript(); // Born together  
        record = new EnrollmentRecord();  
    }  
}
```

```
    }
}

// If student is removed, transcript is also destroyed
```

7 AGGREGATION (HAS-A, weak ownership)

Shared resource relationships!

Requirements:

- Student **HAS-A** list of `Course` objects (courses exist independently)
- Department **HAS-A** list of `Faculty` members (faculty can change departments)
- Library **HAS-A** list of `Book` objects (books exist even if library closes)
- Course **HAS-A** Professor as instructor (professor exists independently)
- Student **HAS-A** Advisor (advisor is a professor who exists separately)

Key Concept: If the container is deleted, the contained objects still exist!

Example:

```
class Course {
    private Professor instructor; // Professor exists independently
    private List<Student> enrolledStudents;

    Course(Professor prof) {
        instructor = prof; // Using existing professor
    }
}
// If course is cancelled, professor still exists
```

8 ASSOCIATION

Relationships between independent entities!

Requirements:

One-to-One:

- Student ↔ StudentIDCard
- Faculty ↔ OfficeRoom

One-to-Many:

- Professor → Multiple Courses (one professor teaches many courses)
- Department → Multiple Faculty members
- Course → Multiple Students

Many-to-Many:

- Students ↔ Courses (many students take many courses)
- Faculty ↔ ResearchProjects (multiple professors collaborate on multiple projects)
- Students ↔ Clubs (students join multiple clubs, clubs have multiple members)

Example Scenarios:

```
// One-to-Many
professor.getCoursesTeaching() // Returns list of courses
course.getInstructor() // Returns one professor

// Many-to-Many
student.getEnrolledCourses() // Multiple courses
course.getEnrolledStudents() // Multiple students
```

⌚ Complete System Requirements

You Must Implement:

1. **Person Management**
 - Student registration and profile management
 - Faculty hiring and assignment
 - Staff management
 2. **Academic Operations**
 - Course creation and management
 - Enrollment system (add/drop courses)
 - Grade management and GPA calculation
 - Prerequisite checking
 3. **Department System**
 - Department creation
 - Faculty assignment to departments
 - Course offerings per department
 - Budget management (composition)
 4. **Library System**
 - Book catalog
 - Borrowing system with different limits per role
 - Due dates and fines calculation
 - Resource sharing (aggregation)
 5. **Financial System**
 - Tuition fee calculation (varies by program)
 - Salary processing for faculty/staff
 - Scholarship management
 - Payment tracking
 6. **Facility Management**
 - Classroom allocation
 - Lab assignments (association with courses)
 - Office assignments (association with faculty)
 - Campus resources
-

❖ Bonus Challenges

1. **Notification System:** Send emails for enrollment confirmation, grade updates, fee reminders
 2. **Attendance Tracker:** Different rules for different course types
 3. **Scholarship System:** Merit-based, need-based, research scholarships
 4. **Course Prerequisites:** Complex prerequisite trees
 5. **Conflict Detection:** Schedule conflicts, room double-booking
 6. **Waiting Lists:** Course capacity management
 7. **Alumni System:** Graduated students with different privileges
-

☒ Deliverables

1. **Class Diagram** showing all relationships
2. **Working Code** with all 8 OOP concepts clearly implemented
3. **Test Scenarios** demonstrating:
 - Student enrolling in multiple courses
 - Professor teaching and grading
 - Library book borrowing
 - Department managing faculty
 - Financial transactions
 - Schedule generation

-
- 4. **Documentation** explaining where each OOP concept is used
-

💡 Tips for Success

- Start with the `Person` base class and build your hierarchy
 - Think about what data should be private (encapsulation)
 - Identify shared behaviors for interfaces
 - Distinguish between composition (strong) and aggregation (weak) ownership
 - Use polymorphism to write flexible, extensible code
 - Keep association relationships clear and navigable
-

🎓 Learning Outcomes

By completing this project, you'll understand:

- How to model real-world systems using OOP
- When to use inheritance vs. interfaces
- The difference between composition, aggregation, and association
- How encapsulation protects data integrity
- How polymorphism enables flexible code
- How abstraction enforces contracts

Good luck, and welcome to TechVerse University! ✨