COSC 113: Computer Science II
Final Group Project Requirements
10/28/2025 to 12/11/2025 [6 weeks]

Objective: This project requires teams of 2–4 students to develop a functional and thoroughly tested application (console or simple GUI). The deliverable must showcase advanced Java programming skills and an understanding of professional, collaborative development workflows.

Technology:

- **Java Version:** The project must use Java 17 (LTS) or a more recent version, leveraging modern Java language features (e.g., records, switch expressions, text blocks) where appropriate.
- **Version Control:** GitHub must be used for version control. Strict adherence to the GitHub Flow branching model is required, including the use of feature branches, regular commits, and Pull Requests (PRs) with basic peer code review before merging to the main branch.
- **Collaborative Contribution:** The final repository history must clearly show commits from all group members, demonstrating equitable distribution of work. The repository history itself is part of the grading criteria.
- **Dependency Management:** The project should be structured with a build automation tool. Maven or Gradle is recommended to manage dependencies (e.g., for testing frameworks or external libraries).
- **Testing and Quality Assurance** (Optional, but Strongly Encouraged): Teams are highly encouraged to implement Unit Tests using JUnit 5. If you choose to include testing, it must demonstrate rigor by achieving a minimum of 70% statement coverage for your core business logic. Coverage must be verified and reported using an industry-standard tool like JaCoCo.

Architecture:

- **Object-Oriented Programming (OOP):** The design must clearly demonstrate the application of all four core OOP principles: Encapsulation, Inheritance, Polymorphism, and Abstraction.
- **Data Structures:** The implementation must utilize an appropriate Collection Framework class (e.g., ArrayList, HashMap, PriorityQueue) or a custom implementation of a Heap, Stack, or Queue. The accompanying report should explain the time complexity justification for its use.

Deliverables:

- **Code Report and Documentation:** A comprehensive report detailing:
  - **Architecture & Design:** Clear, concise UML Class Diagrams and an explanation of the chosen Design Pattern.
  - **Data Structure Justification:** Explanation of the data structure(s) and their benefits.
  - **Contribution Log:** Explicit delineation of specific contributions from each team member, linking to relevant GitHub PRs/commits.
- **Professional Presentation:** A live demonstration and presentation showcasing the project's features, testing approach, and architectural decisions.
- **Advanced Java Mastery:** Implementation must demonstrate robust use of File I/O (e.g., reading/writing data to CSV, JSON, or a simple flat file for persistence) and Exception Handling (e.g., custom exceptions).

Sample Project: The Digital Library Manager

The goal of this project is to develop a robust Java application to automate a library's core operations. This includes managing the inventory of resources (books, media), tracking member accounts, and handling loans, returns, and reservations.

Demonstration of Requirements in the Library Manager:

The structure of the Library Manager provides a clear roadmap for meeting all technical requirements:

Technology & Workflow: The system is built on Java 17. Collaborative work is managed via a GitHub repository utilizing clear feature branches (e.g., "add-book," "process-loan," "member-registration") and formalized Pull Requests for code review and merging.

Object-Oriented Programming (OOP):

- **Encapsulation:** Details like a book's title, author, and ISBN are securely managed within the Book class.
- **Inheritance:** Specialized resource types, such as ReferenceBook and FictionBook, inherit properties from a base Book class.
- **Polymorphism:** A common method, like getDisplayDetails(), is overridden in subclasses to provide specific formatting for different book types.
- **Abstraction:** An LibraryItem interface defines common functionality (e.g., checkOut(), isAvailable()) that all resources must implement.

Data Structures & Memory: Dynamic object creation (for Book, Member, and Loan objects) relies on the Heap. The core inventory can leverage a HashMap for efficient, ***O(1)*** retrieval of items by their unique ISBN. The project documentation will explain this memory and performance choice.

Testing & Advanced Concepts: JUnit tests are implemented for critical operations like searching, loan processing, and due date calculation. The system uses File I/O (e.g., loading and saving data to a file for persistence) and employs Exception Handling to gracefully manage errors like invalid member IDs or trying to check out an unavailable book.

Deliverables: The final Code Report will feature UML class diagrams to visualize the Book hierarchy and detail each team member's specific contributions to the codebase.

Other Project Ideas

These alternative proposals are designed to demonstrate the required OOP principles and advanced Java concepts effectively. Students are free to propose an original project if it meets the complexity and technical requirements.

Core OOP and Management Systems

These projects focus on building foundational object-oriented programming (OOP) skills by modeling real-world entities, their relationships, and fundamental system behaviors.

- **Library Catalog System:** Model **Books**, **Members**, and **Librarians**. This project is ideal for using **abstract classes** (like **Person**) and implementing key behaviors with **interfaces** (like **Borrowable**) to manage item flow.
- **Employee Payroll Calculator:** Design an **abstract Employee** class with specialized subclasses (**Full-Time**, **Part-Time**). The core concept here is implementing a **Taxable** interface to process and calculate wages.
- **University Enrollment Tracker:** Manage **Students**, **Professors**, and **Courses**. This requires a solid use of class relationships, an abstract **Person** class, and the **Gradable** interface for assignment scoring.
- **Hospital Operations Manager:** Connect **Doctors** and **Patients** via **Appointments**. The system should use the **Schedulable** interface and demonstrate **Aggregation** by linking a doctor to their specific **Specialization**.

E-Commerce and Composition-Focused Apps
These ideas are excellent for demonstrating how objects can be aggregated or composed together to form larger, functional units, a crucial OOP concept.

- **Shopping Cart Simulation:** Build a **Cart** that holds various **Product** objects, strongly emphasizing **Composition**. Use an abstract **User** class (**Admin/Customer**) and interfaces like **Discountable** to apply rules.
- **Flight Reservation System:** Link **Flights** and **Passengers** to **Reservations**. Focus on **Composition** by linking a reservation directly to a **FlightSeat** object and implementing the **Bookable** interface.
- **Online Food Delivery App:** Connect **Restaurants** to **Menu Items** and process a user's **Order**. This involves using the **Deliverable** interface and defining an abstract **User** class for different system roles.
- **Vehicle Rental Service:** Create a hierarchy of **Vehicle** types (**Car**, **Truck**, **Bike**) derived from an abstract base. Use **Aggregation** to track the relationship between a **Customer** and their **Rental** contract.

Advanced Systems and Simulations
These projects allow the group to explore more complex class hierarchies, intricate behavior modeling, and nested object relationships.

- **Advanced Banking Simulator:** Design an **abstract Account** with specific types (**Savings**, **Checking**). Model all account actions using a **Transaction** interface and use **Composition** to link a **Customer** to all their various accounts.
- **Smart Home Device Controller:** Build a system around the **Controllable** interface and an abstract **Device** class with subclasses (**Light**, **Thermostat**). Use **Composition** to logically place these devices inside a **Room** object.
- **Social Media Prototype:** Develop core objects like **User**, **Post**, and **Comment**. Implement behavioral interfaces like **Shareable** and **Reportable**, with an emphasis on **nested composition** for comments within a post.
- **Online Quiz/Test System:** Model different question types using an **abstract Question** class with subclasses (**MCQ**, **True/False**). Implement a **Scorable** interface and use **Aggregation** to combine questions into a **Test** object.