Inheritance:

1. Library Management System

Problem Statement:
You are tasked with designing a Library Management System using inheritance. The system should manage various types of books and their details.

Requirements:

1. Create a superclass `Book` with the following fields:
    - `title` (String): The title of the book.
    - `author` (String): The author of the book.7
    - `ISBN` (String): A unique identifier for the book.
2. Provide a constructor and methods to display book details.
3. Create subclasses:
    - `Fiction`: Represents fiction books, includes an additional field `genre` (e.g., fantasy, mystery).
    - `NonFiction`: Represents non-fiction books, includes an additional field `subject` (e.g., history, science).
4. Each subclass should override a `displayDetails` method to display specific details based on its type.
5. In the `Main` class, create instances of both subclasses, set their attributes, and display their details using the overridden method.

2. Vehicle Management System

Problem Statement:
Develop a Vehicle Management System to categorize and manage different types of vehicles.

Requirements:

1. Create a superclass Vehicle with the following fields:
    - brand (String): The brand of the vehicle.
    - model (String): The model of the vehicle.
    - year (int): The year the vehicle was manufactured.
2. Provide a method displayDetails() in Vehicle to display basic details.
3. Create subclasses:
    - Car: Includes an additional field numberOfDoors.
    - Truck: Includes an additional field payloadCapacity.
4. Override the displayDetails() method in both subclasses to include specific details.
5. In the Main class, create instances of Car and Truck and display their details using the displayDetails() method.

3. Animal Classification System

Problem Statement:
Design an Animal Classification System to categorize animals based on their types.

Requirements:

1. Create a superclass Animal with fields:
    - name (String): The name of the animal.
    - habitat (String): The animal's natural habitat.

2. Provide a method displayInfo() in the Animal class to display details.
3. Create subclasses:
    - Mammal: Includes an additional field isDomestic (boolean).
    - Bird: Includes an additional field canFly (boolean).
4. Override the displayInfo() method in the subclasses to include specific details.
5. In the Main class, create instances of Mammal and Bird and display their details using the overridden method.

Encapsulation:

1. Inventory Management System

Problem Statement:
Create an Inventory Management System to track and manage product details while ensuring data security.

Requirements:

1. Create a Product class with fields:
    - productId (String): A unique identifier for each product.
    - name (String): The product's name.
    - quantity (int): The number of items available.
    - price (double): The price per unit.
2. Make all fields private and provide:
    - Getters for all fields.
    - Setters for quantity and price with validation (e.g., quantity should be non-negative, price should be greater than zero).
3. Provide a method displayProductDetails() to show product information.
4. In the Main class, create several product objects, update their quantities and prices using setters, and display their details using the displayProductDetails() method.

2. Hospital Management System

Problem Statement:
Create a system to manage patient information while protecting sensitive data using encapsulation.

Requirements:

1. Create a Patient class with the following fields:
    - patientId (int): A unique identifier.
    - name (String): The patient's name.
    - age (int): The patient's age.
    - medicalHistory (String): Sensitive data related to the patient.
2. Make all fields private and provide:
    - Getters for all fields.
    - A setter for age with validation (e.g., age should be between 0 and 150).
    - No setters for patientId or medicalHistory.
3. Implement a displayInfo() method to display non-sensitive details (e.g., name and age).
4. In the Main class, create instances of Patient, modify age using the setter, and display details using the displayInfo() method.

3. Banking Management System

Create a BankAccount class with **private** fields to ensure that sensitive account information is not directly accessible:

- accountNumber (String): A unique identifier for each account.
- accountHolderName (String): The name of the account holder.
- balance (double): The account balance.
- pin (int): A four-digit PIN required for secure transactions.

Controlled Access via Getters and Setters

- Provide **getters** for accountNumber and accountHolderName (as they should be readable but not modifiable).
- Provide a **getter** for balance, but restrict access by requiring a valid PIN for verification.
- Provide **setters** for balance, ensuring deposits and withdrawals go through controlled methods.
- Do not provide a setter for accountNumber to maintain immutability.

Secure Transactions

- deposit(double amount): Increases the balance but ensures the deposit amount is positive.
- withdraw(double amount, int pin):
  - Requires the correct PIN for withdrawal.
  - Ensures sufficient funds before deducting the amount.
- checkBalance(int pin): Returns the account balance only if the correct PIN is provided.

Data Validation and Security

- The system should prevent:
  - Setting negative balances.
  - Depositing or withdrawing invalid amounts.
  - Accessing balance or withdrawing money without the correct PIN.

Polymorphism:

1. Payment Processing System

Problem Statement:
Design a Payment Processing System to handle different types of payments using polymorphism.

Requirements:

1. Create an abstract class `Payment` with the following:
   - A method `processPayment()` that is abstract.
   - Fields such as `amount` (double).
2. Implement subclasses:
   - `CreditCardPayment`: Represents payments made via credit card, includes fields like `cardNumber` and `cardHolderName`.
   - `UPIPayment`: Represents payments made via UPI, includes fields like `upiId`.
3. Each subclass should provide its own implementation of `processPayment()`, including specific details about the payment.
4. In the `Main` class, demonstrate polymorphism by calling `processPayment()` on a list of `Payment` objects that include different types of payments.

2. Online Shopping System

Problem Statement:
Design an Online Shopping System that calculates discounts for different types of customers.

Requirements:

1. Create a superclass Customer with the following fields:
    - name (String): The customer's name.
    - purchaseAmount (double): The total purchase amount.
2. Include a method calculateDiscount() in Customer, which is overridden by subclasses.
3. Create subclasses:
    - RegularCustomer: Provides a discount of 5% on purchases.
    - PremiumCustomer: Provides a discount of 10% on purchases.
4. In the Main class, demonstrate polymorphism by creating a list of customers and invoking the calculateDiscount() method to calculate the discount for each type of customer.

3. Media Player

Problem Statement:
Build a Media Player that supports different types of media files using polymorphism.

Requirements:

1. Create a superclass MediaFile with fields:
    - fileName (String): The name of the media file.
    - fileSize (double): The size of the file in MB.
2. Include a method play() in MediaFile, which is overridden by subclasses.
3. Create subclasses:
    - AudioFile: Represents audio files, includes an additional field duration.
    - VideoFile: Represents video files, includes fields like resolution and frameRate.

Demonstrate polymorphism by storing instances of AudioFile and VideoFile in a list and invoking the play() method on each object.

ABSTRACTION:

1. Shape Drawing Application

Problem Statement:
Design a system to draw various shapes using abstraction.

Requirements:

1. Create an abstract class Shape with the following:
    - An abstract method draw().
    - A method calculateArea() to calculate the shape's area.
2. Implement subclasses:
    - Circle: Includes a field radius and implements the draw and calculateArea methods.
    - Rectangle: Includes fields length and width and implements the draw and calculateArea methods.

3. In the Main class, create instances of Circle and Rectangle, invoke their methods, and display the area and shape details.

2. Banking System

Problem Statement:
Develop a system for different types of bank accounts using abstraction.

Requirements:

1. Create an abstract class BankAccount with the following fields:
    - accountNumber (String).
    - balance (double).
2. Include abstract methods:
    - deposit(double amount).
    - withdraw(double amount).
3. Create subclasses:
    - SavingsAccount: Limits the number of monthly withdrawals.
    - CurrentAccount: Allows overdraft with a specified limit.
4. In the Main class, demonstrate abstraction by creating instances of both account types and performing operations like deposit and withdrawal.

3. Payment Gateway System

Problem Statement:
Develop a Payment Gateway System to process payments from various sources.

Requirements:

1. Create an abstract class PaymentGateway with fields:
    - transactionId (String): Unique for each transaction.
    - amount (double): The transaction amount.
2. Include abstract methods:
    - processPayment(): To process the payment.
    - validateDetails(): To validate payment details.
3. Create subclasses:
    - CreditCardGateway: Validates credit card details and processes the payment.
    - UPIGateway: Validates UPI details and processes the payment.
4. Demonstrate abstraction by implementing and invoking methods from each gateway type.

Exception Handling

1. Banking Application

Problem Statement:
You are building a banking application that handles exceptions while processing transactions.

Requirements:

1. Create a BankAccount class with the following fields:
    - accountNumber (String).
    - balance (double).

2. Implement methods for:
    - deposit(double amount): Adds money to the account.
    - withdraw(double amount): Deducts money, ensuring the balance doesn't go negative.
3. Throw custom exceptions for:
    - InsufficientBalanceException: Thrown when a withdrawal amount exceeds the balance.
    - InvalidAmountException: Thrown when a deposit or withdrawal amount is negative.
4. In the Main class, simulate deposit and withdrawal operations and handle exceptions gracefully using try-catch.

2. Student Grading System

Problem Statement:
Create a grading system for students that handles invalid data using custom exceptions.

Requirements:

1. Create a Student class with fields:
    - studentId (int).
    - name (String).
    - marks (int): Marks in a subject.
2. Implement methods to:
    - Assign marks to a student.
    - Validate that the marks are between 0 and 100.
3. Throw custom exceptions:
    - InvalidMarksException: If marks are outside the valid range.
4. In the Main class, demonstrate exception handling by assigning valid and invalid marks to students and catching the exceptions.

3. File Processing System

Problem Statement:
Design a File Processing System that handles errors during file operations.

Requirements:

1. Create a FileProcessor class with methods to:
    - Read data from a file.
    - Write data to a file.
2. Implement exception handling for:
    - FileNotFoundException: If the file to be read does not exist.
    - IOException: For general input/output errors.
3. Include a custom exception EmptyFileException to be thrown when attempting to read an empty file.
4. In the Main class, demonstrate file operations and handle exceptions appropriately using try-catch.

Multithreading:

1. Online Ticket Booking System

Problem Statement:
Design a system to handle simultaneous ticket bookings using multithreading.

Requirements:

1. Create a TicketBooking class that represents a ticket booking system. It should include:
    - availableTickets (int): Represents the number of tickets available.
2. Implement a bookTicket(int tickets) method to allow booking tickets. Ensure that:
    - The method is synchronized to avoid race conditions.
    - Tickets cannot be booked if the requested number exceeds available tickets.
3. Create multiple threads representing users trying to book tickets simultaneously.
4. Use thread-safe mechanisms to ensure accurate booking results and prevent overselling of tickets.

2. Restaurant Order System

Problem Statement:
Simulate a restaurant order system where multiple waiters take orders simultaneously.

Requirements:

1. Create an Order class to represent a customer's order, including fields like orderId and orderDetails.
2. Create a Waiter class that implements the Runnable interface. The run() method should simulate taking orders by printing order details to the console.
3. Use a shared Queue to store orders, and ensure thread-safe operations while adding and processing orders.

In the Main class, create multiple waiter threads, add orders to the queue, and process them simultaneously

3. Chat Application

Problem Statement:
Simulate a Chat Application where multiple users can send messages simultaneously.

Requirements:

1. Create a ChatRoom class with methods to:
    - Add a message to the chat room.
    - Display all messages in the chat room.
2. Create a User class that implements the Runnable interface. Each user thread should represent a user sending messages to the chat room.
3. Use thread-safe mechanisms to ensure consistent message storage and retrieval.
4. In the Main class, create multiple user threads and simulate chat activity.

Collection Framework

1. Employee Directory

Problem Statement:
Design an Employee Directory using the Java Collection Framework.

Requirements:

1. Use a HashMap to store employee details where:
    - Key: employeeId (Integer).
    - Value: Employee object.
2. Provide methods to:
    - Add a new employee.

    - Search for an employee by ID.
    - Remove an employee.
    - Display all employees in sorted order of their names (use TreeMap or Comparator).
3. Demonstrate the addition, search, and removal operations in the Main class.

2. Student Records Management

Problem Statement:
Develop a system to manage student records using Java's collection framework.

Requirements:

1. Use an ArrayList to store student details. Each student should be represented by a Student object with fields like studentId, name, and marks.
2. Provide methods to:
    - Add a new student.
    - Remove a student by ID.
    - Search for a student by name.
    - Display all students sorted by marks (use Comparator).
3. Demonstrate operations like adding, searching, and sorting students in the Main class.

3. Movie Database Management

Problem Statement:
Create a Movie Database to store and manage movie details using Java's collection framework.

Requirements:

1. Use a HashSet to store unique movie details. Each movie should have fields:
    - movieId (String): A unique identifier.
    - title (String): The movie's title.
    - rating (double): The movie's rating.
2. Provide methods to:
    - Add a movie to the database.
    - Search for a movie by title.
    - Display all movies sorted by their rating in descending order (use TreeSet or Comparator).
3. Demonstrate the operations in the Main class.

Final Project:

Banking Management System

Problem Statement:
Design a **Banking Management System** that allows customers to manage their accounts, perform transactions, and access banking services while incorporating **inheritance**, **polymorphism**, **encapsulation**, **abstraction**, **exception handling**, **multithreading**, and the **collection framework**.

Requirements:

1. Inheritance: Banking Account Types

- Create a superclass BankAccount with fields:
  - accountNumber (String): A unique identifier for each account.
  - accountHolderName (String): The name of the account holder.
  - balance (double): The current account balance.
- Include methods:
  - deposit(double amount): To deposit money into the account.
  - withdraw(double amount): To withdraw money, ensuring sufficient balance.
  - displayAccountDetails(): To display the account's details.
- Create subclasses:
  - SavingsAccount: Adds a minimumBalance field and ensures that withdrawals do not reduce the balance below the minimum.
  - CurrentAccount: Adds an overdraftLimit field and allows withdrawals up to the overdraft limit.

---

2. Polymorphism: Transaction Processing

- Provide a processTransaction() method in BankAccount, overridden by subclasses for specific transaction behaviors.
- Demonstrate polymorphism by maintaining a list of BankAccount objects and invoking processTransaction() for different account types.

---

3. Encapsulation: Customer Management

- Create a Customer class with the following private fields:
  - customerId (String): A unique identifier for each customer.
  - name (String): The customer's name.
  - accounts (List<BankAccount>): A list of accounts associated with the customer.
- Provide:
  - Getters for all fields.
  - A setter for name to update the customer's name.
  - A method addAccount(BankAccount account) to associate an account with the customer.
  - A method getAccountDetails() to retrieve details of all accounts associated with the customer.

---

4. Abstraction: Banking Operations

- Create an abstract class BankingService with abstract methods:
  - createAccount(Customer customer, String accountType, double initialDeposit): To create a new account.
  - transferFunds(String fromAccount, String toAccount, double amount): To transfer money between accounts.
  - closeAccount(String accountNumber): To close an account.
- Implement BankingService in a concrete class Bank, which maintains a list of customers and accounts.

---

5. Exception Handling: Transaction Errors

- Create custom exceptions:
  - InsufficientBalanceException: Thrown when a withdrawal exceeds the available balance or the overdraft limit.
  - AccountNotFoundException: Thrown when an account is not found for a given account number.
  - InvalidAmountException: Thrown when a deposit or withdrawal amount is zero or negative.
- Use try-catch blocks to handle these exceptions and provide meaningful feedback to the user.

6.Collection Framework: Data Storage

- Use a HashMap<String, BankAccount> to store accounts, with the key being the accountNumber.
- Use a HashMap<String, Customer> to store customer details, with the key being the customerId.
- Implement methods to:
  - Search for customers by name or account number.
  - Display all accounts in the bank, sorted by balance (use TreeMap or Comparator).

Workflow:

1. **Initialize the Bank System:**
   - Add predefined customers and accounts using HashMap.
2. **Account Creation:**
   - Allow customers to create either a SavingsAccount or CurrentAccount with an initial deposit.
3. **Perform Transactions:**
   - Customers can deposit money, withdraw funds, or transfer money between accounts.
   - Handle invalid or insufficient funds using exceptions.
4. **Concurrent Transactions:**
   - Use multiple threads to simulate customers performing transactions simultaneously.
5. **Search and Display:**
   - Search for accounts or customers using their unique identifiers.
   - Display all accounts in the bank sorted by their balance.
6. **Account Closure:**
   - Allow customers to close accounts, ensuring their balance is zero.

Example Scenario:

1. **Bank Initialization:**
    - Add a customer, "John Doe," with a savings account and an initial deposit of $10,000.
2. **Transactions:**
    - John deposits $1,000 and withdraws $500.
    - A second customer, "Jane Smith," transfers $2,000 from her current account to John's savings account.
3. **Concurrent Transactions:**
    - Simulate John and Jane performing transactions at the same time using threads.
4. **Account Queries:**
    - Search for Jane's account by her account number and display her details.
5. **Exception Handling:**
    - Handle scenarios like:
        - Jane attempts to withdraw $50,000, exceeding her overdraft limit.
        - John tries to deposit a negative amount.

Project: Online Shopping Management System

Problem Statement:
Develop an **Online Shopping Management System** where users can browse products, add items to their cart, make payments, and manage orders. The system must demonstrate key Java concepts in its design.

Requirements:

1. Inheritance: Product and Category

- Create a superclass Product with the following fields:
    - productId (String): A unique identifier for each product.
    - productName (String): The name of the product.
    - price (double): The price of the product.
    - stock (int): The available stock for the product.
- Include methods:
    - displayProductDetails(): To display product details.
    - updateStock(int quantity): To reduce or increase stock based on purchases or restocking.
- Create subclasses:
    - Electronics: Adds fields like warrantyPeriod (int) and brand (String).
    - Clothing: Adds fields like size (String) and material (String).

---

2. Polymorphism: Discount Application

- Provide a calculateDiscount() method in Product, overridden in subclasses:
    - Electronics may have a fixed 10% discount.
    - Clothing may have a 20% discount during sales.
- Demonstrate polymorphism by calculating discounts for a list of Product objects.

---

3. Encapsulation: User Account Management

- Create a User class with the following private fields:
  - userId (String): A unique identifier for the user.
  - name (String): The user's name.
  - cart (List<Product>): A list of products added to the cart.
- Provide:
  - Getters for all fields.
  - Setters for name to update user information.
  - Methods:
    - addToCart(Product product): To add a product to the user's cart.
    - removeFromCart(Product product): To remove a product from the cart.
    - viewCart(): To display the contents of the cart.

---

4. Abstraction: Order Processing

- Create an abstract class Order with abstract methods:
  - placeOrder(User user): To confirm and save the order.
  - cancelOrder(String orderId): To cancel an existing order.
- Implement Order in a concrete class OnlineOrder, which manages a list of orders and their statuses.

---

5. Exception Handling: Errors in Shopping

- Create custom exceptions:
  - OutOfStockException: Thrown when a user tries to purchase an item that is out of stock.
  - InvalidPaymentException: Thrown when payment details are invalid or insufficient funds are detected.
  - EmptyCartException: Thrown when a user tries to place an order without adding items to the cart.
- Use try-catch blocks to handle these exceptions and provide appropriate error messages.

---

6. Multithreading: Payment Processing

- Create a PaymentTask class that implements Runnable. This task:
  - Simulates payment processing for multiple users concurrently.
  - Ensures thread-safe access to the payment gateway.
- In the Main class, create threads to handle payments for multiple orders simultaneously.

---

7. Collection Framework: Inventory and Orders

- Use a HashMap<String, Product> to store all products, where the key is the productId.
- Use a List<Order> to maintain order history.
- Implement methods to:
  - Search for products by name or category using streams or filters.
  - Display all products sorted by price or stock (use TreeMap or a custom Comparator).

Workflow:

1. **Initialize Inventory:**
    - Add predefined Electronics and Clothing products to the inventory.
2. **User Actions:**
    - Users can browse products and add items to their cart.
    - View the cart and proceed to checkout.
3. **Placing Orders:**
    - Validate cart contents to ensure stock is available.
    - Handle errors like empty carts or out-of-stock products using exceptions.
4. **Payment Processing:**
    - Simulate payment processing for orders using multiple threads.
    - Handle invalid payments through custom exceptions.
5. **Search and Display:**
    - Search for products by name or category.
    - Display products sorted by price or stock availability.
6. **Order Management:**
    - Confirm or cancel orders.
    - View order history for each user.

Example Scenario:

1. **Product Initialization:**
    - Add a laptop (Electronics) with a price of $1,000 and stock of 10.
    - Add a t-shirt (Clothing) with a price of $20 and stock of 50.
2. **User Actions:**
    - User "John Doe" browses and adds the laptop and t-shirt to the cart.
    - Another user, "Jane Smith," adds the same laptop to their cart.
3. **Concurrent Transactions:**
    - John and Jane both proceed to checkout at the same time.
    - Multithreading ensures the stock is updated correctly for the first successful transaction.
4. **Exception Handling:**
    - Jane encounters an OutOfStockException as the laptop is no longer available.
    - John completes the payment successfully.
5. **Order Management:**
    - John views his order history.
    - Jane cancels her incomplete order.

E-Library Management System

Problem Statement:
Design an **E-Library Management System** that allows users to borrow, return, and search for books while demonstrating a combination of **inheritance**, **polymorphism**, **encapsulation**, **abstraction**, **exception handling**, **multithreading**, and **collection framework**.

Requirements:

1. Inheritance: Library Structure

- Create a superclass LibraryItem with the following fields:
  - itemId (String): A unique identifier for the item.
  - title (String): The title of the book or resource.
  - availability (boolean): Whether the item is currently available or not.
- Include a method displayDetails() in the LibraryItem class to display the item's details.
- Create subclasses:
  - Book: Adds fields such as author (String) and genre (String).
  - Journal: Adds fields such as publisher (String) and issueNumber (int).

2. Polymorphism: Item Management

- Provide a borrowItem() method in the LibraryItem class that is overridden by both Book and Journal to display specific messages when borrowing different types of items.
- Demonstrate polymorphism by invoking borrowItem() for a list of LibraryItem objects.

3. Encapsulation: User Information

- Create a User class with the following private fields:
  - userId (String): A unique identifier for the user.
  - name (String): The user's name.
  - borrowedItems (List<LibraryItem>): A list of items borrowed by the user.
- Provide:
  - Getters for all fields.
  - Setters for name only.
  - A method addBorrowedItem(LibraryItem item) to add an item to the user's borrowed list.
  - A method removeBorrowedItem(LibraryItem item) to remove an item when returned.

---

4. Abstraction: Library Operations

- Create an abstract class LibraryOperations with abstract methods:
  - searchItem(String title): To search for an item by its title.
  - returnItem(LibraryItem item, User user): To handle the return process.
  - addItem(LibraryItem item): To add new items to the library.
- Implement LibraryOperations in a concrete class Library, which maintains a list of LibraryItem objects.

---

5. Exception Handling: Invalid Operations

- Create custom exceptions:
  - ItemNotAvailableException: Thrown when a user attempts to borrow an unavailable item.
  - ItemNotFoundException: Thrown when a user searches for an item that doesn’t exist.
  - MaxBorrowLimitExceededException: Thrown when a user attempts to borrow more than 5 items.
- Use try-catch blocks to handle these exceptions and provide meaningful error messages to the user.

---

6. Multithreading: Concurrent Borrowing

- Create a BorrowingTask class that implements Runnable. This task:
  - Simulates a user borrowing an item from the library.

    - Ensures thread-safe access to the Library's inventory using synchronization.
- In the Main class, create multiple threads for users borrowing items concurrently.

---

7. Collection Framework: Library Inventory

- Use a HashMap<String, LibraryItem> to store library items, where the key is the itemId.
- Use an ArrayList<User> to manage users.
- Implement methods to:
    - Add new items to the inventory.
    - Search for items by title using streams or filtering.
    - Display all items sorted by title (use TreeMap or a Comparator).

Workflow:

1. The library starts with a predefined set of books and journals.
2. Users can:
    - Search for items by title.
    - Borrow items (if available).
    - Return items.
3. Each user can borrow up to 5 items. If they attempt to exceed this limit, an exception is thrown.
4. Multiple users can perform actions (like borrowing) simultaneously through threads.
5. All borrowed items are recorded in the user's borrowedItems list.
6. Display all library items sorted by title, along with their availability status.

Example Scenario:

1. **Initialize Library:**
    - Add books and journals to the library inventory using HashMap.
2. **Add Users:**
    - Create user profiles and store them in an ArrayList.
3. **Borrowing Process:**
    - Users search for a specific book by title.
    - If the book is available, they borrow it; otherwise, an ItemNotAvailableException is thrown.
4. **Concurrent Borrowing:**
    - Use multiple threads to simulate concurrent borrowing.
5. **Returning Items:**
    - Users return borrowed items, updating the library inventory.

Project: Employee Management System with Role Hierarchy

Problem Statement:

Design an **Employee Management System** that manages different types of employees within an organization. The system should demonstrate **upcasting** and **downcasting** while assigning tasks and calculating bonuses based on employee roles.

Requirements:

1. Class Hierarchy

- Create a superclass Employee with the following fields:
  - employeeId (String): A unique identifier.
  - name (String): Employee's name.
  - salary (double): Basic salary of the employee.
- Include methods:
  - displayDetails(): Displays employee information.
  - calculateBonus(): Returns 10% of the salary as a bonus (default behavior).
- Create subclasses:
  - Manager:
    - Adds teamSize (int).
    - Overrides calculateBonus() to return 20% of the salary as a bonus.
  - Developer:
    - Adds programmingLanguage (String).
    - Overrides calculateBonus() to return 15% of the salary as a bonus.

2. Upcasting Demonstration

- Store all employee objects in a List<Employee> collection.
- Use **upcasting** to assign Manager and Developer objects to the Employee reference type.
- Iterate through the list and invoke displayDetails() and calculateBonus() using the Employee reference.

3. Downcasting Demonstration

- Use **downcasting** to access subclass-specific properties:
  - If the employee is a Manager, display the teamSize.
  - If the employee is a Developer, display the programmingLanguage.
- Use the instanceof operator to check the object type before downcasting.

4. Encapsulation

- Make all fields private.
- Provide appropriate **getters and setters** for each field.
- Validate that the salary is **non-negative** and teamSize is **greater than zero**.

Example Scenario:

1. **Employee Initialization:**
   - Create a Manager named "Alice" with a salary of $50,000 and a team size of 5.
   - Create a Developer named "Bob" with a salary of $40,000 and a programming language of "Java".
2. **Upcasting:**
   - Store both objects in a list of type Employee.
3. **Polymorphic Behavior:**
   - Iterate through the list and calculate bonuses using upcasting.
4. **Downcasting:**

- Identify the employee type and display additional role-specific information using downcasting.

5. **Data Validation:**
   - Prevent negative salaries and invalid team sizes during object creation.