Abstraction:

1. **Abstract Class Implementation:** Create an abstract class `Shape` with an abstract method `calculateArea()`. Create two subclasses `Circle` and `Rectangle`, which implement this method. Test the functionality by calculating the area of both shapes.
2. **Abstract Class with Multiple Methods:** Create an abstract class `Vehicle` with methods `startEngine()` and `stopEngine()`. Create subclasses `Car` and `Bike` that implement these methods. Write a driver class to test the methods.
3. **Interface Example:** Define an interface `Payment` with a method `makePayment()`. Implement this interface in classes `CreditCardPayment` and `CashPayment`. Test these classes by calling the `makePayment()` method.

Inheritance:

4. **Single Inheritance:** Create a base class `Animal` with a method `sound()`. Create a subclass `Dog` that extends `Animal` and overrides the `sound()` method to bark. Test the functionality by creating an object of `Dog` and calling the `sound()` method.
5. **Multilevel Inheritance:** Create a class `Employee` with fields like `name` and `salary`, and a method `getDetails()`. Create a subclass `Manager` that adds a field `teamSize` and another subclass `Developer` that adds a field `programmingLanguage`. Override the `getDetails()` method in both subclasses to include these new fields.
6. **Hierarchical Inheritance:** Create a class `Appliance` with a method `turnOn()`. Create two subclasses `Fan` and `AirConditioner` that inherit from `Appliance` and override `turnOn()` with specific functionality.

Polymorphism:

7. **Method Overloading:** Create a class `Calculator` with overloaded methods `add()` to handle the addition of two integers, two floats, and two doubles. Test each overloaded method by passing appropriate values.
8. **Method Overriding:** Create a base class `BankAccount` with a method `calculateInterest()`. Create subclasses `SavingsAccount` and `CurrentAccount` that override the `calculateInterest()` method with their specific implementations.
9. **Polymorphic Behavior:** Create a base class `Employee` with a method `work()`. Create subclasses `Engineer`, `Manager`, and `Technician`, each overriding the `work()` method. Demonstrate polymorphic behavior by creating an array of `Employee` objects and calling `work()` on each.

Encapsulation:

10. **Getters and Setters:** Create a class `Student` with private fields `name`, `age`, and `grade`. Provide public getter and setter methods to access and modify these fields. Write a main method to test the encapsulation by setting and getting the values.
11. **Encapsulation with Validation:** Create a class `Account` with private fields `accountNumber`, `balance`, and `accountHolder`. Provide getter and setter methods, ensuring validation in the setters (e.g., balance cannot be negative).
12. **Encapsulation in Real World:** Create a class `Book` with private fields for `title`, `author`, and `price`. Write methods to encapsulate the fields and add validation that ensures the price cannot be negative. Then, create a `Library` class that stores an array of books and allows adding/removing books using encapsulated methods.