Problem 1: Inheritance

Create a Java program that models a simple bank account system using inheritance. Define a base class called Account with the following properties

- `-accountNumber (integer)`
- `-accountHolder (String)`
- `-balance (double)`

The Account class should have methods for:

-A constructor to initialize the account details.

- `-deposit(double amount)`: To deposit money into the account.
- `-withdraw(double amount)`: To withdraw money from the account (assuming sufficient balance).

Create two subclasses: `SavingsAccount` and `CurrentAccount`.

`SavingsAccount:`

- -inherit from Account.
- -Add an additional property `interestRate (double)`.
- -Override the `withdraw` method to ensure that the balance doesn't go below a certain limit (e.g., 500).

`CurrentAccount:`

- -Inherit from Account.
- -Add an additional property `overdraftLimit (double)`.
- -Override the `withdraw` method to allow overdrawing up to the overdraft limit.

Create a `Main` class with a `main` method to test your classes. Instantiate objects of both `SavingsAccount` and `CurrentAccount`, deposit and withdraw money, and print the account details after each transaction.

Implement the `Account`, `SavingsAccount`, and `CurrentAccount` classes to satisfy the requirements of the problem. Feel free to modify or extend the problem as needed. Happy coding!

Problem2: Polymorphism

Create a Java program that models a hierarchy of geometric shapes using polymorphism. Define a base class called `Shape` with an abstract method `calculateArea()`.

`Shape Class`:

-An abstract class with the abstract method `calculateArea()`.

-Add a property `shapeName (String)` to store the name of the shape.

-A constructor to initialize the `shapeName`.

Subclasses:

`Circle:`

-Subclass of `Shape`.

-Add properties `radius (double)`.

-Implement the `calculateArea()` method to calculate the area of the circle.

`Rectangle`:

-Subclass of `Shape`.

-Add properties `length and width (both double)`.

-Implement the `calculateArea()` method to calculate the area of the rectangle.

`Triangle:`

-Subclass of `Shape`.

-Add properties `base and height (both double)`.

-Implement the `calculateArea()` method to calculate the area of the triangle.

Create a `Main` class with a `main` method to test your classes. Create an array of `Shape` objects that includes circles, rectangles, and triangles. Iterate through the array and print the name and area of each shape.

Implement the `Shape`, `Circle`, `Rectangle`, and `Triangle` classes to satisfy the requirements of the problem. Ensure that polymorphism is used effectively, allowing the `calculateArea()` method to be called on different shapes through a common interface. Feel free to modify or extend the problem as needed. Happy coding!

Problem3: Abstraction

Create a simple library management system using abstraction. Define an abstract class called `LibraryItem` with the following properties:

- `-title (String)`: The title of the library item.
- `-itemID (int)`: A unique identifier for the item.
- `-numCopies (int)`: The number of copies available.

Include an abstract method `displayInfo()` that prints the information about the library item.

Subclasses:

`Book`:

- -Subclass of `LibraryItem`.
- -Additional property: `author (String)`.
- -Implement the `displayInfo()` method to display information about the book.

`DVD:`

-Subclass of `LibraryItem.`

-Additional property: `director (String).`

-Implement the `displayInfo()` method to display information about the DVD.

Create a `Library` class that manages an array of `LibraryItem` objects. Include methods to add items to the library, display information about all items in the library, and check out an item (decrement the number of copies) if available.

Create a `Main` class with a `main` method to test your classes. Instantiate objects of the `Book` and `DVD` classes, add them to the library, display information about all items, and check out an item.

Implement the `LibraryItem, Book, DVD, and Library` classes to satisfy the requirements of the problem. Ensure that abstraction is effectively used, providing a common interface through the `LibraryItem` class. Feel free to modify or extend the problem as needed. Happy coding!

Problem4: Encapsulation

Create a simple Student class to represent student information with encapsulation. The `Student` class should have the following private properties:

`-studentId (int)`: A unique identifier for each student.

`-name (String)`: The name of the student.

`-age (int)`: The age of the student.

`-grade (char)`: The grade of the student.

Include getter and setter methods for each property. Ensure that the `studentId` is generated automatically and cannot be modified externally.

Create a `StudentManagementSystem` class to manage an array of `Student` objects. Include methods to add a student, display information about all students, and update the grade of a specific student.

Implement the `Student` class with encapsulation using private properties and appropriate getter and setter methods. Ensure that the `studentId` is generated automatically and cannot be modified externally. Implement the `StudentManagementSystem` class to manage an array of `Student` objects with methods to add students, display student information, and update student grades. Feel free to modify or extend the problem as needed. Happy coding!

Problem 5: Exception Handling

Create a Java program that simulates a scenario where a `NullPointerException` can occur and handle it appropriately.
Define a class called `Person` with the following properties:

```
-name (String)
-age (int)
```

Create another class called `PersonManager` with the following methods:

```
-displayPersonDetails(Person person)
```

-This method should take a `Person` object as a parameter and display the person's details (name and age).

-If the `Person` object or any of its properties is `null`, catch the `NullPointerException` and print an error message.

`Main Class:`

Create a `Main` class to test the `PersonManager` class. In the `main` method, perform the following operations:

-Create a `Person` object with a name and age and display its details using `displayPersonDetails`.

-Attempt to display the details of a `Person` object that is `null`.

-Attempt to display the details of a `Person` object with a `null` name.

The expected output should handle `NullPointerException` gracefully and print appropriate error messages. Implement the `Person and PersonManager` classes with the specified methods to handle `NullPointerException`. Feel free to modify or extend the problem as needed. Happy coding!