5 problems on Object-Oriented Programming (OOP) principles.

## 1. The Secure Bank Account 🔒

Scenario: You are developing a simple banking application. For security, the account balance should not be directly accessible or modifiable from outside the class. All transactions (deposits and withdrawals) must go through specific methods that validate the transaction.

- **Topics Covered: Encapsulation**
- **Task:**
    1. Create a class named BankAccount.
    2. Give it a private property called $balance.
    3. Create a public constructor that accepts an initial balance (defaulting to 0).
    4. Create a public method deposit($amount) that adds a positive amount to the balance.
    5. Create a public method withdraw($amount) that subtracts a positive amount from the balance but only if there are sufficient funds.
    6. Create a public method getBalance() that returns the current balance.
- **Sample Usage:**

PHP

```
$account = new BankAccount(100); // Initial balance: $100


$account->deposit(50);

$account->withdraw(20);


echo "Current balance: " . $account->getBalance(); // Should show 130


// This should fail silently or return an error message

$account->withdraw(200);
```

echo "<br>Final balance: " . $account->getBalance(); // Should still be 130

- **Expected Output:**
- Current balance: 130
- Final balance: 130

## 2. The User Hierarchy 👑

Scenario: You're building a system with different types of users. All users share common properties like a username and a login method. However, an Admin user has special privileges, such as the ability to ban other users, which a regular Member does not.

- **Topics Covered: Inheritance**
- **Task:**
    1. Create a base class User with protected properties $username and $email.
    2. The User class should have a constructor to set these properties and a public method login() that returns a simple greeting message.
    3. Create a final class Admin that **extends** User.
    4. The Admin class should have a public method banUser($user) that prints a message indicating a user has been banned.
- **Sample Usage:**

PHP

$admin = new Admin("SuperAdmin", "admin@site.com");

echo $admin->login(); // Inherited method

echo "<br>";

$admin->banUser("JohnDoe"); // Method from Admin class

- **Expected Output:**
- User SuperAdmin has logged in.
- Admin SuperAdmin has banned the user: JohnDoe.

## 3. The Shape Calculator 📐

Scenario: You need to calculate the area of different geometric shapes (like a Circle and a Rectangle). While the formula for each is different, you want to ensure that any "shape" object in your system is guaranteed to have a method to calculate its area.

- **Topics Covered: Abstract Classes** and **Polymorphism**
- **Task:**
    1. Create an abstract class Shape.
    2. Inside Shape, define an abstract method calculateArea().
    3. Create two classes, Circle and Rectangle, that **extend** Shape.
    4. Implement the calculateArea() method in both Circle and Rectangle with the correct formulas (Area_circle=pitimesr2, Area_rectangle=widthtimesheight).
    5. Create a function that can take any Shape object and print its area.
- **Sample Usage:**

PHP

```php
function printArea(Shape $shape) {
   echo "The area is: " . $shape->calculateArea() . "<br>";
}


$circle = new Circle(5); // radius = 5
$rectangle = new Rectangle(4, 6); // width=4, height=6


printArea($circle);
printArea($rectangle);
```

- **Expected Output:**
- The area is: 78.539816339745
- The area is: 24

## 4. The Exportable Content

Scenario: Your Content Management System (CMS) manages different types of content, such as blog posts (BlogPost) and reports (Report). You want to add a feature that allows any type of content to be exported into different formats (e.g., JSON, CSV). You must enforce a rule that any "exportable" content type *must* have the ability to be converted to these formats.

- **Topics Covered: Interfaces**
- **Task:**
    1. Create an interface called Exportable with two method signatures: exportToJson() and exportToCsv().
    2. Create two classes, BlogPost and Report, that **implement** the Exportable interface.
    3. In each class, implement the interface methods to return a string representing the content in the specified format.
- **Sample Usage:**

PHP

```
$post = new BlogPost("My First Post", "This is the content.");

$report = new Report("Monthly Sales", ["sales" => 5000, "expenses" => 2000]);


echo "Blog Post as JSON: " . $post->exportToJson() . "<br>";

echo "Report as CSV: " . $report->exportToCsv() . "<br>";
```

- **Expected Output:**
- Blog Post as JSON: {"title":"My First Post","content":"This is the content."}
- Report as CSV: Monthly Sales,5000,2000

## 5. The Sharable Logger

Scenario: In your application, you have several unrelated classes, like Product and Order. You want both classes to have the ability to log messages (e.g., "Product created", "Order processed") without forcing them to inherit from a common parent class, which might not make logical sense.

- **Topics Covered: Traits**
- **Task:**
    1. Create a trait named Loggable.
    2. Inside the Loggable trait, create a public method log($message) that simply echoes a log entry with a timestamp.
    3. Create two classes, Product and Order.
    4. Make both classes **use** the Loggable trait.
    5. Instantiate objects of both classes and call the log() method on each to show that they share the functionality.
- **Sample Usage:**

PHP

```
$product = new Product("Laptop");

$product->log("New product added to inventory.");


$order = new Order(123);

$order->log("Order has been shipped.");
```

- **Expected Output:** (Timestamps will vary)
- [2025-08-14 22:32:46] Log: New product added to inventory.
- [2025-08-14 22:32:46] Log: Order has been shipped.