GROUP 4 - Chess Game (PHP Web Application)

- OOP Implementation Guide

1. **Encapsulation** is the bundling of data (attributes/properties) and the methods (functions) that operate on that data into a single unit (the class). It also involves hiding the internal state of an object and restricting direct access, requiring all interaction to occur through controlled public methods.

Implementation:

- **AbstractPiece**: This class encapsulates the fundamental properties of any chess piece:
  - Data: protected $color, protected $position, protected $symbol, protected $hasMoved.
  - Access Control: The properties are marked as protected, meaning they cannot be accessed directly from outside the piece hierarchy (e.g., from ChessGame or the global script).
  - Controlled Access Methods: Public methods like getColor(), getPosition(), and setPosition() are provided as the *only* way to interact with and modify the piece's state. For example, setPosition() not only updates the position but also sets $this->hasMoved = true;, controlling the internal state consistently.
- **ChessGame**: This class encapsulates the entire game state:
  - Data: private array $board, private string $turn, private string $message, private array $history.
  - Methods: initializeBoard(), makeMove($from, $to), getBoard(), etc., which are the only ways to access or change the game state. The implementation of a move is hidden within makeMove().

2. Abstraction focuses on showing only essential information to the user and hiding the complex implementation details. In PHP, this is often achieved using Abstract Classes and Interfaces.

Implementation**:**

- **AbstractPiece:** This is an **Abstract Class**. It defines a common blueprint for all pieces but cannot be instantiated itself.
  - **Required Methods (Abstract)**: It declares abstract methods that *must* be implemented by all concrete piece classes:
    - abstract protected function getGeometricMoves(array $board): array;
    - abstract public function getLegalMoves(array $board): array;
- **Common Functionality (Concrete):** It provides common, complex functionality that all pieces need, hiding the implementation from subclasses and the game logic:
  - **Sliding Movement:** The highly reusable and complex logic for Rook, Bishop, and Queen movement is implemented once in protected function calculateSlidingMoves().
  - **King Safety Logic:** The critical logic to prevent a piece from moving into check is abstracted into a protected function filterSafeMoves(), which every piece uses in its getLegalMoves() method. The game logic (ChessGame::makeMove) only calls getLegalMoves(), and doesn't need to know *how* that list of moves was filtered for safety.

3. Inheritance allows a new class (subclass/child) to inherit the properties and methods of an existing class (superclass/parent). This promotes code reuse and establishes a clear hierarchical relationship (the "is-a" relationship).

Implementation:

- **Piece Hierarchy:** All concrete piece classes (Rook, Pawn, Bishop, Queen, King, Knight) **inherit** from AbstractPiece.
    - class Rook extends AbstractPiece { ... }
- **Code Reuse:** Each concrete piece class automatically inherits all the common attributes (e.g., $color, $position, $hasMoved) and common methods (e.g., getColor(), setPosition(), calculateSlidingMoves(), filterSafeMoves(), posToIndices(), etc.).
- **Specific Implementation:** Each subclass then provides its own unique implementation for the abstract methods defined in the parent, such as getGeometricMoves(), which calculates the specific movement patterns for that piece (e.g., a Rook's straight lines vs. a Knight's 'L' shape).

4. Polymorphism allows objects of different classes to be treated through a uniform interface (like an abstract class or method call). The same method call can produce different, correct results based on the object type.

Implementation:

- **getLegalMoves():** This is the primary example of polymorphism. In the ChessGame class, when calculating moves:

  ```
  $legalMoves = $piece->getLegalMoves($this->board);
  ```

  The variable $piece could be an instance of Rook, Pawn, King, etc. When getLegalMoves() is called, the PHP engine determines the *actual* type of the $piece object and executes the version of the method specifically implemented for that piece.

- **Board Iteration:** The ChessGame class stores all pieces in the $board array. When the game logic iterates over the board, it doesn't need to check if ($piece is Rook) or if ($piece is Knight) to find the piece's symbol or moves. It simply calls the shared methods, and the correct, polymorphic version is executed:

  ```
  echo "<span class='piece-symbol ...'>" . $piece->getSymbol() . "</span>";
  ```

- **Simulated Attack Check:** In AbstractPiece::isSquareAttacked(), the loop calls the generic getGeometricMoves() on every opposing piece:

  ```
  $moves = $piece->getGeometricMoves($board);
  ```

    - Even though the piece types are different, they all share this method signature, ensuring the attack calculation works uniformly across all piece types.