Clean Code Cheat Sheet for Tetris Project

1. Readable Names:

Use clear, descriptive names for variables, classes, and methods. Avoid abbreviations or single-letter names except for loop variables (e.g., use `currentPiece` instead of `cp`).

2. Single Responsibility Principle:

Ensure every class or method handles only one responsibility. For instance, manage Tetromino logic in the `Piece` class and keep board-related functionality in the `GameBoard` class.

3. Avoid Magic Numbers:

Replace literal values like grid dimensions with named constants (e.g., `rowLength = 10`, `colLength = 15`). This improves readability and simplifies updates.

4. Modular Functions:

Break large methods into smaller, reusable functions. For example, separate collision detection, line clearing, and scoring logic into distinct methods.

5. Encapsulation:

Keep variables private unless absolutely necessary. Use getter and setter methods for controlled access, such as modifying or retrieving the current Tetromino position.

6. Descriptive Comments:

Add comments to explain complex logic (e.g., Tetromino rotation matrices). Avoid comments for self-explanatory code.

7. Follow the DRY Principle:

Avoid code duplication by using helper methods. Reuse logic for position validation, grid manipulation, and scoring across the project.

8. Error Handling:

Validate inputs and game states proactively. For example, check for collisions before allowing Tetromino movement to avoid runtime errors.

9. Consistent Formatting:

Use tools like Prettier or VSCode formatters to maintain consistent indentation, spacing, and line breaks across the codebase.

10. Testable Code:

Design methods to be independent and testable. Ensure functions like `clearLines` and `checkCollision` can be tested in isolation with predefined board states.

11. Minimize Dependencies:

Keep coupling low by making classes independent. Use dependency injection where necessary to avoid tightly bound modules.

12. Efficient Data Structures:

Use appropriate data structures for the game grid (e.g., a 2D list for the board) to optimize performance and simplify manipulation.

13. Avoid Nested Logic:

Flatten nested `if` statements using guard clauses for better readability and maintainability.

14. Readable Error Messages:

When handling exceptions or validations, provide user-friendly error messages for debugging or gameplay feedback.

15. Refactor Regularly:

Periodically review and refactor the code to remove redundancies, optimize performance, and improve clarity.