**NOTE : Even though I collaborated for the Part A , PART B is completely my contribution.**Before you proceed let me explain you where the part 2 files are placed in the directory.

1.Ran PMD command using the built-in quickstart ruleset against the IPower.java source code and the output is stored in **pmd-report.html** in the root directory.

2.Fixed all the smells and then I made a new file named **ModifiedIpower.java**

Commit hash for this 6ab6afe4659dbe1561c90061ac5b3f1f0301dbac also submitted in submission.txt

3. Ran PMD against the new ModifiedIpower.java and that report is named as **Modified- pmd-report.html**

**F). In what ways are the results same ?**

What I observed is all the three tools highlighted the same general issues.

Handling Negative exponenets:

Both chatgpt and Co-pilot suggested to handle the negative exponents separately, but PMD did not suggested me regarding the handling of negative exponents.

Code duplication and Recursion : Chatgpt and Co-pilot both mention the risk of deep recursion,but PMD did not give me a detail regarding this

Null Handling : ChatGPT, Copilot, and PMD point out issues with cache handling in PowerCached

**In what way are the results Different ?**

Chatgpt provides detailed breakdown of the issues,suggesting the cache handling , thread safety for example – ConcurrentHashMap and how to solve recursion along with the code refactoring and explanations.

Copilot offers high-level-summary , suggesting improvement like compuetifabsent for caching and error handling of negative power but it is not preforming like ChatGPT for thread safety and recursion.

PMD mainly focuses on syntax errors and basic efficiency improvements but it didn’t perform any indepth refactoring like chatgpt or copilot .

Error Handling:

ChatGpt and Copilot suggest the explicit handling for negative exponents, with specific code fixes and the use of IllegalArgumentException

PMD did not provide any insights regarding this .

C**.) Based on PMD’s documentation, and what the GenAI tools tell you about how it did the analysis, discuss whether they  
interpret and perform “static analysis” using the same methods.**

PMD is static analysis tool means it scans the source code for potential issue , it follows what is defined in the rule set and find the patterns but doesn’t execute the code.it can find syntax errors and potential bugs .

However , chatgpt and Copilot did perform the static analysis in a more contextual way . they did examine my code in evaluated way following the best practices, coding standards, and logical flow.they went beyond the syntax errors and recommended me to improve my code for redability , efficiency and threadsafety.

Chatgpt concentrated more on code structure and logic , providing better refactoring techniques and code quality

Copilot suggest’s me and give me code snippets for common issues and focuses more on practical fixes .

**d. Answer: if you only had one of these 3 tools to choose, which would you use for static analysis of your production code  
and why?**

I would obviously like to use ChatGpt because it gives a comprehensive view, and reviews my code like a mentor stating what went wrong and how to correct it . even though you ask it for one specific error it tries to find the other error in you code if it exists , which is saving my time and effort .As we saw above I a extremely impressed by chatgpt response for thread safety, recursion depth, and logical improvements.

**a) (1) Run PMD using the built-in quickstart ruleset against the IPower.java source code, and copy the text output to a file (Made a html report please refer to it under the JUNIT-5-BoundaryTestingin the folder I submitted.)**

<html><head><title>PMD</title></head><body>

<center><h3>PMD report</h3></center><center><h3>Problems found</h3></center><table align="center" cellspacing="0" cellpadding="3"><tr>

<th>#</th><th>File</th><th>Line</th><th>Problem</th></tr>

<tr bgcolor="lightgrey">

<td align="center">1</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">11</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_bestpractices.html#missingoverride">The method 'toPower(int, int)' is missing an @Override annotation.</a></td>

</tr>

<tr>

<td align="center">2</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">13</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_codestyle.html#controlstatementbraces">This statement should have braces</a></td>

</tr>

<tr bgcolor="lightgrey">

<td align="center">3</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">15</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_codestyle.html#emptycontrolstatement">Empty for statement</a></td>

</tr>

<tr>

<td align="center">4</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">15</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_codestyle.html#controlstatementbraces">This statement should have braces</a></td>

</tr>

<tr bgcolor="lightgrey">

<td align="center">5</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">17</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_codestyle.html#unnecessarysemicolon">Unnecessary semicolon</a></td>

</tr>

<tr>

<td align="center">6</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">18</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_codestyle.html#unnecessarysemicolon">Unnecessary semicolon</a></td>

</tr>

<tr bgcolor="lightgrey">

<td align="center">7</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">27</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_bestpractices.html#missingoverride">The method 'toPower(int, int)' is missing an @Override annotation.</a></td>

</tr>

<tr>

<td align="center">8</td>

<td width="\*%">/Users/lamusamuel/Documents/Fall-2024/SER-515/constructionlab2/JUnit5-BoundaryTesting/src/main/java/solid3/IPower.java</td>

<td align="center" width="5%">28</td>

<td width="\*"><a href="https://docs.pmd-code.org/pmd-doc-7.7.0/pmd\_rules\_java\_codestyle.html#unnecessaryfullyqualifiedname">Unnecessary qualifier 'PowerCached': '\_\_cache' is already in scope because it is declared in an enclosing type</a></td>

</tr>

</table></body></html>

**b) (1) Ask ChatGPT (this is the only allowed use of ChatGPT) “Run static analysis on the following code: “ and copy-paste in the  
code in IPower.java. Then ask ChatGPT how it performed that analysis. Record the share URL, and also copy/paste your  
ChatGPT’s conversation into your word doc**

Chat GPT Conversation with modified code : <https://chatgpt.com/share/67402806-bc0c-8012-b357-141fe6a53353>

Convo With ChatGPT :

*Run static analysis on the following code:*

Response :

I’ll analyze your code for common issues, code quality, and adherence to best practices, focusing on correctness, efficiency, and readability.

**1. Interface Implementation:**

• ✅ The IPower interface is well-defined and implemented by both classes (PowerSimple and PowerCached), adhering to the Liskov Substitution Principle (L in SOLID).

**2. Code Structure and Readability:**

• **PowerSimple Class:**

• **Nested Loop Statement:**

for (int i = 0; i < pow; res \*= n, i++);

• ❌ **Code Smell:** Using commas to separate expressions inside the for loop is unconventional and may confuse readers. Consider splitting this loop into a more readable format.

• ✅ **Suggestion:**

for (int i = 0; i < pow; i++) {

    res \*= n;

}

• **Extraneous Semicolons:**

❌ Multiple unnecessary semicolons (;;;) after the loop make the code harder to maintain and can cause confusion.

**3. PowerCached Class:**

• **Cache Initialization:**

• In the toPower method:

if (entry == null) {

    entry = new HashMap<Integer, Integer>();

}

❌ **Issue:** If a new entry is created, it’s not added to the cache. Therefore, caching only occurs if the entry already exists, potentially losing efficiency.

✅ **Fix:** Add the newly created entry back to the main cache:

if (entry == null) {

    entry = new HashMap<>();

    PowerCached.\_\_cache.put(n, entry);

}

• **Double Cache Lookup:**

The method accesses the cache twice:

Integer res = e.get(pow);

if (res != null) { return res; }

✅ This pattern is acceptable but could benefit from using computeIfAbsent for conciseness:

return e.computeIfAbsent(pow, k -> n \* toCachedPower(e, n, pow - 1));

• **Thread Safety:**

❌ **Potential Issue:** The current implementation is not thread-safe. Multiple threads accessing or modifying the cache concurrently could cause inconsistencies.

✅ **Suggestion:** Use a ConcurrentHashMap instead of a regular HashMap or synchronize access to the cache:

private static Map<Integer, Map<Integer, Integer>> \_\_cache = new ConcurrentHashMap<>();

**4. Naming and Conventions:**

• **Variable Naming:**

Variables like e (in toCachedPower) could be renamed for better clarity (cacheEntry or powerCache).

• **Method Naming:**

Methods such as toCachedPower could benefit from a more descriptive name like calculateWithCache.

**5. Efficiency Considerations:**

• **Recursive Calculation:**

While the recursive approach works, for large pow values, it could lead to stack overflow. Consider an iterative or exponentiation-by-squaring method for better performance and to avoid deep recursion.

**6. General Recommendations:**

• Add comments explaining complex logic, especially around caching mechanisms.

• Ensure proper handling of negative or large pow values, as this code assumes non-negative pow.

**Summary of Issues Identified:**

1. **Readability:** Unnecessary semicolons and non-standard loop syntax.

2. **Cache Handling:** New cache entries aren’t being stored properly.

3. **Thread Safety:** Cache implementation is not thread-safe.

4. **Variable Naming:** Improve clarity with more descriptive names.

5. **Efficiency:** Consider iterative or optimized algorithms for large pow.

By addressing these issues, the code will be more robust, readable, and efficient!

*How did you Perform the Analysis ?*

I performed the static analysis by carefully reviewing your code against key principles of software development, focusing on the following areas:

**1. Code Logic and Correctness:**

• **Functionality Check:** Ensured that the logic in both classes (PowerSimple and PowerCached) correctly implements the exponentiation functionality.

• **Algorithm Review:** Examined the loops and recursive calls to confirm they calculate powers as expected.

• **Edge Cases:** Considered scenarios such as pow == 0 and potential handling of negative or large exponents.

**2. Code Structure and Readability:**

• **Syntax Inspection:** Identified unconventional syntax, such as:

for (int i = 0; i < pow; res \*= n, i++);

This syntax is technically valid but reduces readability.

• **Extraneous Elements:** Noted unnecessary semicolons (;;;) after statements, which are redundant and could lead to maintenance issues.

• **Best Practices:** Suggested improvements like splitting complex loop constructs into more readable formats.

**3. Data Management and Efficiency:**

• **Cache Implementation:** Checked how the caching mechanism is handled in PowerCached. I identified that newly created cache entries were not added back to the main cache:

if (entry == null) {

    entry = new HashMap<>();

    // Missing \_\_cache.put(n, entry);

}

• **Recursion Depth:** Recognized that deep recursion for large pow values could cause stack overflow, suggesting iterative or optimized methods like exponentiation-by-squaring.

**4. Concurrency and Thread Safety:**

• **Static Cache Usage:** Since static variables are shared across threads, I pointed out that concurrent access to HashMap might cause inconsistencies. Suggested using ConcurrentHashMap for thread-safe operations.

**5. Naming Conventions and Clarity:**

• Evaluated variable and method names for clarity. Suggested renaming variables like e to more descriptive names (e.g., cacheEntry).

• Recommended consistent and meaningful names to improve maintainability.

**6. Design Principles (SOLID and Clean Code):**

• Checked adherence to the **SOLID** principles, particularly:

• **Single Responsibility Principle (SRP):** Ensured each class has a single responsibility.

• **Liskov Substitution Principle (LSP):** Verified that PowerSimple and PowerCached correctly implement the IPower interface.

**Tools and Techniques Mimicked:**

Although I didn’t use automated static analysis tools directly (like SonarQube or Checkstyle), the analysis simulated their functionality by applying:

• **Pattern Matching:** Identifying common code smells.

• **Manual Walkthrough:** Stepping through code logic to detect potential flaws.

• **Best Practice Application:** Comparing your code to known Java best practices.

This manual approach is similar to what automated tools provide but includes a human context review for deeper insights!

C) Convo With Microsoft Co-pilot

A screenshot of a computer code

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1) Repeat (b) except this time use Microsoft’s Copilot. Again, record the URL and output of the conversation.  
Microsoft Co-pilot   
Let's perform a static analysis of the provided Java code. Here are some observations and potential issues:

**Code Structure and Design**

1. **Interface and Implementation**:
   * The IPower interface is well-defined, providing a clear contract for power calculation.
   * Two implementations (PowerSimple and PowerCached) demonstrate different approaches to calculating powers.
2. **Caching Mechanism**:
   * The PowerCached class uses a caching mechanism to store previously computed results, which is a good optimization for repeated calculations.

**Potential Issues**

1. **Syntax Errors**:
   * The use of Map&gt; and HashMap&gt;() is incorrect. It should be Map<Integer, Integer> and new HashMap<Integer, Integer>(). This is likely a result of HTML encoding (e.g., > being encoded as &gt;).
2. **Unused Variables**:
   * In PowerSimple, the loop has unnecessary semicolons (;) after the for statement, which can lead to confusion. The loop should be:
3. for (int i = 0; i < pow; i++) {
4. res \*= n;
5. }
6. **Potential NullPointerException**:
   * In PowerCached, if PowerCached.\_\_cache.get(n) returns null, the subsequent call to entry.get(pow) could lead to a NullPointerException if not handled properly.
7. **Type Safety**:
   * The use of raw types for Map (e.g., Map entry) should be avoided. Instead, use generics to ensure type safety:
8. Map<Integer, Integer> entry = PowerCached.\_\_cache.get(n);
9. **Recursive Depth**:
   * The recursive method toCachedPower could lead to a StackOverflowError for large values of pow. Consider using an iterative approach or tail recursion if applicable.
10. **Code Comments**:
    * While comments are present, they could be more descriptive. For example, explaining the caching strategy and the significance of the contract in toCachedPower.

**Suggestions for Improvement**

* **Refactor for Clarity**: Clean up the syntax errors and improve the readability of the loop in PowerSimple.
* **Enhance Error Handling**: Add checks to handle potential null values in the cache.
* **Use Generics**: Ensure all collections use generics for type safety.
* **Consider Iterative Solutions**: For power calculations, consider implementing an iterative approach to avoid deep recursion.

**Conclusion**

The code demonstrates a solid understanding of interfaces and caching, but it requires some corrections and improvements for robustness and clarity. Would you like to dive deeper into any specific part of the code or discuss further enhancements?

Q: How did you perform the Analysis?  
I performed the analysis by carefully reviewing the provided Java code for several key aspects:

1. **Syntax and Structure**: I looked for any syntax errors or structural issues, such as incorrect use of generics or misplaced semicolons.
2. **Logic and Functionality**: I assessed how the code implements its intended functionality, particularly the power calculations and caching mechanism.
3. **Error Handling**: I checked for potential runtime errors, such as NullPointerExceptions, and considered how the code handles edge cases.
4. **Code Quality**: I evaluated the overall readability and maintainability of the code, including the use of comments and adherence to best practices.
5. **Performance Considerations**: I considered the efficiency of the algorithms used, especially in terms of recursion depth and potential stack overflow issues.

By focusing on these areas, I aimed to provide a comprehensive overview of the code's strengths and weaknesses. If you have any specific questions about the analysis or want to explore a particular aspect further, feel free to ask!

**D) Fixed ALL the smells reported by PMD – please find it under src/main/java/solid3/ModifiedIPower3.java**

**e) Repeat the process**

**PMD Report**

<html><head><title>PMD</title></head><body>

<center><h3>PMD report</h3></center><center><h3>Problems found</h3></center><table align="center" cellspacing="0" cellpadding="3"><tr>

<th>#</th><th>File</th><th>Line</th><th>Problem</th></tr>

</table></body></html>

Chat GPT Conversation with modified code : <https://chatgpt.com/share/6740325b-35bc-8012-b2c1-b0ed3dbfed9f>  
  
Do static analysis on the following code :

**Static Analysis of the Provided Code**

The code appears to implement an interface ModifiedIPower with two classes, PowerSimple and PowerCached, offering different implementations for calculating the power of a number. Below is a static analysis with identified issues, potential improvements, and adherence to coding best practices.

**1. Class Structure and Interface Implementation:**

• **Good**: Both classes correctly implement the ModifiedIPower interface.

• **Suggestion**: Consider making the cache in PowerCached class private final to ensure it is not modified elsewhere.

**2. Memory Efficiency in PowerCached:**

• **Issue**: cache in PowerCached is not effectively used because the entry map is not stored back into the cache after being created. The cache will remain empty, and caching won’t work as intended.

• **Fix**: After initializing entry, store it back in the cache:

if (entry == null) {

    entry = new HashMap<>();

    cache.put(n, entry);  // Add this line to store the entry back in the cache

}

**3. Thread Safety:**

• **Issue**: PowerCached is not thread-safe. If multiple threads access the cache simultaneously, it may lead to inconsistent results or unexpected behavior.

• **Fix**: Use ConcurrentHashMap instead of HashMap or synchronize access to the cache:

private static Map<Integer, Map<Integer, Integer>> cache = new ConcurrentHashMap<>();

**4. Handling Negative Exponents:**

• **Issue**: Neither class handles negative exponents. This might lead to incorrect or unexpected results if negative values are passed.

• **Suggestion**: Add validation or handle negative exponents explicitly, possibly throwing an IllegalArgumentException or calculating the inverse.

**5. Code Duplication and Recursion Depth:**

• **Observation**: toCachedPower in PowerCached uses recursion to calculate the power, which might lead to stack overflow for large pow values.

• **Suggestion**: Consider using an iterative approach to avoid deep recursion and improve performance.

**6. Code Readability and Null Handling:**

• **Observation**: Explicitly setting Integer res = null; is unnecessary, as it is immediately assigned within the if-else block.

• **Suggestion**: Simplify the assignment:

int res;

if (pow == 0) {

    res = 1;

} else {

    res = e.getOrDefault(pow, -1);

    if (res == -1) {  // Indicates no cached value

        res = n \* toCachedPower(e, n, pow - 1);

        e.put(pow, res);

    }

}

**7. Coding Conventions and Best Practices:**

• **Consistent Naming**: The method toPower could follow Java naming conventions and be named calculatePower.

• **Javadoc Comments**: Add comments explaining the purpose of each class and method, especially for public methods.

**Summary of Key Changes:**

• Ensure entry is added back to the cache in PowerCached.

• Consider thread safety using ConcurrentHashMap.

• Handle negative exponents gracefully.

• Avoid deep recursion for better performance and reliability.

• Improve readability by simplifying variable initialization.

These changes will enhance the code’s correctness, performance, and maintainability.

How did you perform the analysis ?  
I performed the static analysis by examining the code for key aspects related to functionality, performance, and coding best practices. Here’s a detailed breakdown of the process:

**1. Code Structure Review:**

• **Objective:** Ensure classes, interfaces, and methods are well-defined and follow standard Java conventions.

• **Action:** Checked the implementation of the ModifiedIPower interface and its methods in PowerSimple and PowerCached.

**2. Logic Flow and Functionality:**

• **Objective:** Verify that each method correctly implements its intended functionality.

• **Action:**

• Reviewed toPower logic in both classes to ensure correct calculation.

• Checked the PowerCached class’s caching logic and identified that it doesn’t store the intermediate map (entry) back into the main cache, rendering the cache ineffective.

**3. Performance and Efficiency:**

• **Objective:** Identify areas where performance might degrade, such as inefficient caching or deep recursion.

• **Action:**

• Noted that the recursion in PowerCached could lead to a stack overflow for large pow values.

• Suggested an iterative approach to improve reliability.

**4. Memory Management:**

• **Objective:** Assess the use of data structures for efficiency and correct memory handling.

• **Action:**

• Identified the use of HashMap for caching and its potential thread-safety issues.

• Suggested using ConcurrentHashMap or synchronizing access to prevent race conditions in multi-threaded environments.

**5. Edge Case Handling:**

• **Objective:** Ensure the code handles all possible inputs gracefully.

• **Action:**

• Checked for handling of edge cases, such as pow == 0 and negative exponents.

• Noted the absence of logic for negative exponents and recommended adding validation or functionality for them.

**6. Code Style and Best Practices:**

• **Objective:** Ensure adherence to Java coding standards and improve readability.

• **Action:**

• Reviewed variable naming, method naming, and initialization practices.

• Suggested using getOrDefault and avoiding unnecessary initializations.

**7. Concurrency Considerations:**

• **Objective:** Identify potential issues in a multi-threaded environment.

• **Action:**

• Observed that cache is static but not synchronized, making it unsafe for concurrent access.

• Recommended thread-safe alternatives or synchronization.

**Tools and Techniques Used:**

• **Manual Inspection:** Reviewed each class and method for logical correctness.

• **Best Practice Check:** Compared the code against standard Java coding practices.

• **Pattern Matching:** Identified common patterns and potential pitfalls in caching and recursion logic.

This approach ensures comprehensive static analysis, addressing correctness, performance, and maintainability.

**Convo With Co-pilot :**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a chat

Description automatically generated**

To perform static analysis on your Java code, we can identify several potential issues and areas for improvement. Here’s a breakdown of the code you provided:

**Code Review Summary**

1. **Syntax Errors**:
   * The use of Map&gt; and HashMap&lt;&gt;() indicates that the angle brackets are incorrectly encoded. They should be Map<Integer, Integer> and new HashMap<>().
2. **Generics**:
   * The Map in PowerCached should specify its types for better type safety. For example:
3. private static Map<Integer, Integer> cache = new HashMap<>();
4. **Code Efficiency**:
   * The toPower method in PowerSimple uses a loop to calculate powers, which is fine for small values but can be inefficient for larger exponents. Consider using a more efficient algorithm like **exponentiation by squaring**.
5. **Null Handling**:
   * In PowerCached, when retrieving the cache entry, you should ensure that the cache is updated correctly. If entry is null, it should be put back into the cache after being initialized.
6. **Documentation**:
   * Adding JavaDoc comments to your methods and classes would improve code readability and maintainability. This is especially useful for public interfaces.
7. **Error Handling**:
   * Consider adding error handling for negative powers, as the current implementation does not address this scenario.

**Suggested Improvements**

Here’s a revised version of your PowerCached class with some of these suggestions applied:

class PowerCached implements ModifiedIPower {

private static Map<Integer, Integer> cache = new HashMap<>();

@Override

public int toPower(int n, int pow) {

if (pow < 0) {

throw new IllegalArgumentException("Power must be non-negative");

}

return cache.computeIfAbsent(pow, k -> toCachedPower(n, k));

}

private int toCachedPower(int n, int pow) {

if (pow == 0) {

return 1;

}

return n \* toCachedPower(n, pow - 1);

}

}

**Tools for Static Analysis**

To run static analysis on your Java code, consider using tools like:

* **SonarQube**: Great for continuous inspection of code quality.
* **PMD**: Useful for identifying potential bugs and code smells.
* **Checkstyle**: Helps enforce coding standards and style guidelines.

These tools can be integrated into your IDE (like IntelliJ or Eclipse) for real-time feedback as you code.

How did you perform the analysis ?

I performed the analysis by reviewing your Java code for common programming practices and potential issues. Here’s how I approached it:

1. **Syntax Check**: I looked for any syntax errors or incorrect usage of Java constructs, such as generics.
2. **Code Structure**: I assessed the overall structure of the classes and methods, ensuring they followed good design principles.
3. **Efficiency**: I considered the efficiency of algorithms used, particularly in the power calculation methods.
4. **Type Safety**: I checked for proper use of generics to enhance type safety and reduce runtime errors.
5. **Error Handling**: I evaluated whether the code handled edge cases, such as negative powers.
6. **Documentation**: I noted the absence of comments and documentation, which are important for maintainability.
7. **Best Practices**: I compared the code against common best practices in Java programming.