Chapter 4 – Comments – Summary

Comments should:

- Be well-placed to provide helpful information.
- Compensate for the limitations in expressing intent through code.
- Include legal comments for copyright and licensing.
- Offer informative, explanatory, clarification, warning, and TODO comments.
- Be included in public APIs using well-described Javadocs.

Comments to avoid:

- Frivolous and dogmatic comments that clutter the code.
- Comments added just for the sake of having them.
- Comments that attempt to cover up bad code; code improvement should be preferred.
- Overuse of comments, hindering code readability.
- Redundant comments that provide no value.
- Comments that are more time-consuming to read than the code they describe.
- Useless, redundant Javadocs that clutter and obscure code.

Good comments:

- Informative comments that provide basic information.
- Explanation of intent comments that justify coding decisions.
- Clarification comments that make obscure code readable.
- Warning comments that alert others to potential consequences.
- TODO comments that mark tasks requiring attention.

Specific bad comment categories:

- Misleading Comments: Comments that are imprecise and could mislead other programmers by not accurately describing the code's behavior.
- Mandated Comments: Discouraging rules that require comments on every function or var
- Journal Comments: Avoiding excessive tracking of changes in comments.
- Noise Comments: Discouraging redundant and obvious comments.
- Scary Noise: Distracting and unnecessary comments within catch blocks.
- Don't Use a Comment When You Can Use a Function or a Variable
- Position Markers: Cautioning against excessive use of banners or comment blocks.
- Closing Brace Comments: Considering such comments as potential code smell.
- Attributions and Bylines: Suggesting that author attribution is better managed by source code control systems.
- Commented-Out Code: Avoiding the practice of commenting out code.
- HTML Comments: Cautioning against adding HTML markup to code comments.
- Nonlocal Information: Comments should be relevant to the local code context.
- Too Much Information: should not contain excessive historical or irrelevant details.
- Inobvious Connection: should have an obvious connection to the code they describe.
- Function Headers: Recognizing that short functions with clear names may not need extensive comment headers.
- Javadocs in Nonpublic Code: Stating that Javadoc-style comments are not necessary for nonpublic code and may be overly formal and distracting.

Chapter 7 – Error Handling – Summary

Error Handling Best Practices:

- Error handling is essential in programming to address abnormal input and device failures.
- Clean code should effectively manage errors while maintaining code clarity.
- Prioritize exceptions over error flags or return codes for code simplification.
- Begin with a try-catch-finally statement when working with exceptions for consistency and expected behavior.
- Favor unchecked exceptions (runtime exceptions) over checked exceptions in application development.
- Include comprehensive context within exceptions for error source and location identification.
- Define exception classes based on their intended catch and handling.
- Consider the Special Case Pattern to simplify code and reduce conditional complexity.
- Prefer throwing exceptions or returning special case objects over returning null.
- Minimize the use of null as an argument when feasible.

Error Handling To Avoid:

- Excessive reliance on error flags or return codes, which can clutter code and diminish readability.
- Complex and intertwined error handling logic that obscures the primary code's purpose.
- Strict adherence to checked exceptions, potentially leading to code modification cascades.
- Inadequate error context within exceptions, complicating diagnosis and debugging.
- Repetitive catch blocks for similar exception types from third-party APIs.
- Managing special cases with convoluted conditional checks or excessive null checks.

Must Do:

- Implement error handling as a distinct concern with clear, context-rich error messages.
- Use exceptions as the primary error handling mechanism to simplify code.
- Initiate with a try-catch-finally statement when working with exceptions.
- Provide informative error context within exceptions for enhanced diagnosis and logging.
- Define exception classes based on their usage and handling.
- Contemplate encapsulating special cases using the Special Case Pattern.
- Favor throwing exceptions or returning special case objects over using null.

Conclusion:

- Clean code emphasizes readability and robust error handling.
- Error handling should be managed separately, implemented with clarity, and supported by appropriate context for effective debugging and maintenance.