Chapter 3: Functions

**Small!**

* Functions should be small, with a preference for them to be shorter than 20 lines long.
* Longer functions can make code harder to read and search.
* The size of the function should correspond to its scope. Using single-letter names or numeric constants can make it difficult to locate information in a body of text. It is recommended to use longer names for variables that a programmer might need to search.

**Blocks and Indenting**

* Functions should be short, with blocks within control statements being a single line long, usually a function call.
* This helps keep the enclosing function small and provides documentation through the descriptive function name.
* Functions should not be large enough to hold nested structures, with an indent level not greater than one or two for readability.

**Do One Thing**

* Example 3-1 violates guidelines for clear and concise functions, being complex and unclear.
* Example 3-3 follows the "one thing" principle, making it clear, concise and easily readable, with a limited indent level.

**One Level of Abstraction per Function**

* Functions should be concise with one-liner blocks within control statements, typically consisting of function calls.
* This promotes a small, readable function and clear documentation through descriptive function names.
* Nested structures should be avoided, with an indent level no greater than two for readability.

**Reading Code from Top to Bottom: The Stepdown Rule**

* Functions should be clear, concise, and easy to understand by using a top-down narrative style and performing a single task.
* Code should have a consistent level of abstraction, with blocks within control statements being a single line long, usually a function call and indent level not exceeding one or two for readability.
* The Stepdown Rule helps maintain this level of abstraction for easier code maintenance.

**Switch Statements**

* Switch statements can become complex and violate design principles.
* To simplify, use polymorphism and move the switch statement to a low-level class.
* Functions can then be dispatched through the employee interface for better maintainability.

**Use Descriptive Names**

* Choosing descriptive names for functions is crucial to making code clean and readable.
* Functions should be focused, with clear, easy-to-read names that accurately describe their purpose.
* A descriptive name is preferable to a short, cryptic name or long comment.
* create a name that clearly states what the function does.

**Function Arguments**

* Function arguments should be limited to zero or one for best clarity.
* More than one argument takes conceptual power and makes testing more difficult.
* Output arguments can also be confusing compared to input arguments. The ideal function has only one input argument.

**Common Monadic Forms**

* Single argument functions have 2 main uses, asking a question or transforming input into return value.
* A less common use is for event with no output argument but changes system state.
* It's crucial to choose names clearly showing the function form and avoiding arguments used wrongly, such as using output argument for transformation instead of return value.

**Flag Arguments**

* Flag arguments in functions are discouraged because they indicate that the function does more than one thing, depending on the value of the flag.
* A method call such as render(true) is confusing to a reader. A better approach would be to split the function into separate functions for each action, such as renderForSuite() and renderForSingleTest().

**Dynamic Functions**

* Functions with one argument are simpler and clearer, while functions with two arguments are more complex and can lead to bugs.
* When using two arguments, make sure they are necessary and try to simplify by using one argument if possible, such as by making the second argument a member variable or creating a new class.

**Triads**

* Functions with three arguments are less readable than those with one or two arguments and should be avoided unless necessary.

**Argument Objects**

* When a function takes many arguments, it's a sign that some of them can be grouped into a class of their own.
* This can improve the clarity of the code and make the function easier to understand.

**Argument Lists**

* Functions with variable arguments can have up to three arguments, and if they are treated identically, they can be treated as a single argument of type List.

**Verbs and Keywords**

* The importance of naming functions is emphasized in order to effectively convey the purpose and order of arguments.
* A good monad name should form a clear verb-noun pair, such as "writeField(name)" which indicates that "name" is being written as a field.

**Have No Side Effects**

* Functions should not have side effects as they create hidden changes that can result in unexpected behavior and confusion.
* Side effects can lead to temporal couplings and order dependencies that make it difficult to understand when a function can be called.
* Functions should be limited to doing one thing, so if a function has a side effect, it should be reflected in the name of the function.

**Output Arguments**

* Functions should be short, with a single line within control statements, typically a function call.
* This enhances readability and documentation through the descriptive function name.
* Nested structures should not be included in functions, with an indent level no greater than one or two.
* Arguments are best interpreted as inputs to a function and output arguments should be avoided.

**Command Query Separation**

* Functions should either perform an action or return information, not both.
* Combining these often causes confusion, it is best to separate the action and the query into separate functions, so that the meaning is clear.

**Prefer Exceptions to Returning Error Codes**

* Returning error codes from functions violates the command-query separation principle and leads to complex and nested error handling code.
* Exceptions can be used to simplify error processing by separating it from the normal flow of the code.

**Extract Try/Catch Blocks**

* Extracting the try and catch blocks into separate functions can improve the clarity of error processing code and simplify it.
* This separates error handling from normal processing and makes the code easier to understand and modify.

**Error Handling Is One Thing**

* Functions should focus on one specific task, and error handling should be a separate task.
* When a function handles errors, it should only perform that error handling and nothing else.

**The Error.java Dependency Magnet**

* Using error codes in a program can lead to dependencies in many other classes, which must import and use them.
* This makes changes to the error codes difficult, as they require all related classes to be recompiled and redeployed.
* Using exceptions avoids this issue, as new exceptions can be added without requiring recompilation or redeployment.

**Don’t Repeat Yourself**

* Duplication in code increases size and risks of errors. To reduce these risks, software development strategies aim to eliminate duplication and improve code readability and maintainability.
* To eliminate duplication, various software development strategies have been developed, including database normal forms, object-oriented programming, structured programming, aspect-oriented programming, and component-oriented programming.

Chapter 4: Comments

**Comments Do Not Make Up for Bad Code**

* Don't write comments to explain bad code. Instead, clean the code to make it clear and expressive.

**Explain Yourself in Code**

* Write code that clearly explains your intent instead of relying on comments.

**Good Comments**

* Comments are not always necessary or beneficial. Good code is clear and expressive without comments, and it's better to spend time improving the code than writing comments.
* The best comment is one you didn't have to write.

**Legal Comments**

* Some comments are required by corporate coding standards for legal reasons, such as copyright and authorship statements.
* These comments should be concise and refer to a standard license or external document instead of including all terms and conditions in the comment.

**Informative Comments**

* Sometimes basic information in a comment is useful, but it's better to convey it through the function name.

**Explanation of Intent**

* A comment explains the intent behind a decision in the code

**Clarification**

* It is better to use function names to convey information instead of relying on comments.
* However, comments can be used to explain the intent behind a decision or clarify the meaning of an argument or return value.
* It is necessary to be careful and accurate when writing clarifying comments.

**Warning of Consequences**

* It warn other programmers about consequences of code by using comments.
* To prevent thread safety issues, we use comments to explain why a new instance of SimpleDateFormat must be created for each instance of the code.

**TODO Comments**

* TODO comments are reminders for tasks that need to be done in the future.
* They should not be used as excuses for leaving bad code.
* It's important to regularly scan and eliminate TODOs to keep the code clean.

**Amplification**

* Comments can also be used to emphasize a code that’s important that may be overlook

**Javadocs in Public APIs**

* A well-described public API is beneficial and a must-have.
* Writing good javadocs for a public API is important but beware of misleading, nonlocal, and dishonest comments.

**Bad Comments**

* The majority of comments are not useful and serve as justifications for poor code or insufficient decisions, essentially serving as the programmer talking to themselves.

**Mumbling**

* Many comments are poor excuses for bad code or justifications for poor decisions, merely the programmer talking to themselves.
* Ensure the comment is the best it can be before writing it.
* A comment that forces the reader to look elsewhere for its meaning has failed and isn't worth the space it takes up.

**Redundant Comments**

* The javadocs are redundant and only add clutter to the code. They do not add any value to the understanding of the code and can be considered a waste of time and space.

**Misleading Comments**

* Clear and precise comments are crucial in technical writing.
* Inaccurate comments can lead to confusion and misinterpretation, causing errors in the code.
* It is important to thoroughly review comments before submitting to ensure that they provide valuable information to the reader.

**Mandated Comments**

* It only serves to obscure the code and creates the possibility for false information and misdirection.
* Having strict rules that mandate every function or variable must have a comment is counter-productive and can lead to meaningless, repetitive comments.
* It is more important to focus on writing meaningful and accurate comments that provide valuable information and clarify the purpose of the code.

**Journal Comments**

* People sometimes add a log of every change made to a module in the form of comments at the start. This can result in long, run-on journal entries with many pages.

**Noise Comments**

* This type of comments state the obvious and offer no new information

**Scary Noise**

* Sometimes comments are just redundant and add no value.
* Javadocs can also be noisy and redundant, providing no useful information.

**Don’t Use a Comment When You Can Use a Function or a Variable**

* The original code contains a redundant comment and can be refactored to improve code readability and maintainability by removing it.

**Position Markers**

* The use of banners in source code should be limited and only used when the benefit is significant, as overuse can cause them to be ignored and become background noise.

**Closing Brace Comments**

* Special comments on closing braces in source code should be avoided, especially in small and encapsulated functions, as they can add unnecessary clutter.
* Shortening the functions instead is a better approach.

**Attributions and Bylines**

* Bylines in source code are unnecessary and can become less accurate and relevant over time.
* Source code control systems are better suited to keep track of changes made to the code.

**Commented-Out Code**

* Commenting out code is discouraged because it tends to stay in the code for years, leading to confusion about its importance and accuracy.
* Source code control systems can be used to store code instead.

**HTML Comments**

* It is considered bad practice to include HTML in source code comments as it makes the comments hard to read in the editor/IDE.
* Ideally, comments should be easy to understand and free from formatting distractions.

**Nonlocal Information**

* Write comments that describe the code they are near. Don't include information that is not related to the code.

**Too Much Information**

* Keep only relevant information in comments. Don't include interesting historical discussions or irrelevant details.

**Inobvious Connection**

* Don't write unclear comments. Make sure the code and the comment match and are easily understood together.
* The purpose of a comment is to explain code that is unclear on its own.

**Function Headers**

* Small functions don't need comments, use descriptive names for that function instead.

**Javadocs in Nonpublic Code**

* Javadocs are not suitable for non-public code and generate unnecessary distraction.