# **Chapter 3: Functions**

**SUMMARY**

Functions are the first line of organization in any program. It is critical because it is one of the primary things to consider when implementing a program.

* Functions should be short and if possible smaller than short.
* Functions should do one thing and should do it only.

**Blocks and Indenting**

* Blocks within statements should be one line long.
* Functions should not be long enough to hold nested structures
* Indent level of a function should not be greater than one or two to make it easy to understand.

**Sections within Functions**

* Functions that do one thing cannot be reasonably divided into sections.

**One Level of Abstraction per Function**

* There is a need that all functions have the same level of abstraction for it is easy to trace who violates a rule.

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

* It is the key to keeping functions short and making sure they do “one thing.” Making the code read like a top-down set of TO paragraphs is an effective technique for keeping the abstraction level consistent.

**Switch Statements**

* By nature, switch statements always do N things. We can’t always avoid switch statements, but we can make sure that each switch statement is buried in a low-level class and is never repeated.

**Use Descriptive Names**

* Choosing descriptive names will clarify the design of the module in your mind and help you to improve it.

**Function Arguments**

* The ideal number of arguments for a function is zero (niladic), one (monadic), two (dyadic), three (triadic), and if more than three (polyadic) but it requires special justification.
* Output arguments often cause us to double take.
* Input arguments is the next best thing to no arguments.

**Common Monadic Forms**

* Always choose a name that will make the distinction clear, and always use the two forms in a consistent context.
* The overall program is meant to interpret the function call as an event and use the argument to alter the state of the system
* *Event*: there is an existing input argument without output argument.

**Flag Arguments**

* It complicates the signature of the method.

**Dyadic Functions**

* It is a function with two arguments.
* If you are using dyads you should be aware that they come at a cost and should take advantage of what mechanism may be available to you to convert them into monads.

**Triads**

* It is a function with three arguments.
* Before you use triads, you should think carefully for it is more complicated than dyads.

**Argument Objects**

* It will reduce the number of arguments.

**Argument Lists**

* . Functions that take variable arguments can be monads, dyads, or even triads.

**Verbs and Keywords**

* Every argument should be paired with a nice verb/noun.
  + For example: **writeField**(name)

**Have No Side Effects**

* Function must promise to do one thing.
* Side effects creates *temporal coupling* which is confusing to understand.

**Output Arguments**

* Arguments are most naturally interpreted as inputs.
* Double-take on an argument is actually an output.
* Output arguments should be avoided.

**Command Query Separation**

* Functions should either do or answer something but not both.
* Doing both leads to confusion.
* To prevent confusion, separate the command from the query.

**Prefer Exceptions to Returning Error Codes**

* Returning error codes from command functions is a subtle violation of command query separation.
* It promotes commands being used as expressions in the predicates of if statements.
* Using exceptions can be separated from the happy path code and can be simplified.

**Extract Try/Catch Blocks**

* It confuses the structure of the code and mix error processing with normal processing.

**Error Handling is One Thing**

* A function that handles errors should do nothing else.

**Error.java Dependency Magnet**

* Returning error codes usually implies that there is some class or enum in which all the error codes are defined.

**Don’t Repeat Yourself**

* Duplication may be the root of all evil in software. It bloats the code and will require four-fold modification should the algorithm ever have to change.

**Structured Programming**

* Every function, and every block within a function should have one entry and one exit.

# **Chapter 4: Comments**

**SUMMARY**

Comments in terms of programming are indications that states the overall purpose of a specific method. It compensates us from our failures to express ourselves in code. Without comments we cannot fully understand or express ourselves.

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

* Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.
* Instead of explaining what you have messed, try spend it cleaning the mess.

**Explain Yourself in Code**

* To easily understand your code, explain it according to its intent.

**Good Comments**

* Good comment is the comment you found a way not to write.

**Legal Comments**

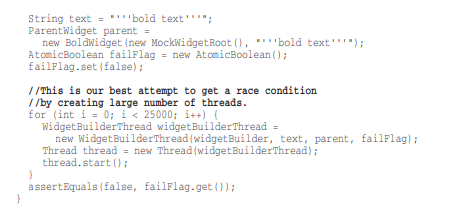
* Comments like this should not be contracts or legal tomes, refer to a standard license or other external document rather than putting it all in a comment.

**Informative Comments**

* It is better to use the name of the function to convey the information where possible.

**Explanation of Intent**

* The following figure is an example of a good explanation of intent.

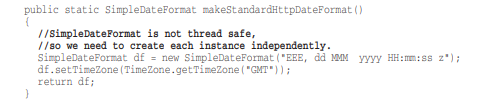
****

**Clarification**

* It is better to find a way to make that argument or return value clear in its own right; but when it’s part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful.

**Warning of Consequences**

* Here is one of the examples to prevent some overly eager programmer from using a static initializer in the name of efficiency.

****

**TODO Comments**

* It explains why the function has a degenerate implementation and what that function’s future should be.

**Amplification**

* A comment may be used to amplify the importance of something that may otherwise seem inconsequential.

**Javadocs in Public APIs**

* If you are writing a public API, then you should certainly write good javadocs for it.

**Bad Comments**

* Bad comments are crutches or excuses for poor code or justifications for insufficient decisions, amounting to little more than the programmer talking to himself.

**Mumbling**

* Any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you and is not worth the bits it consumes.

**Redundant Comments**

* Comments that is on repeat is not informative. It does not justify the code, or provide intent or rationale.

**Misleading Comments**

* The subtle bit of misinformation couched in a comment that is harder to read than the body of the code.

**Mandated Comments**

* Comments like this just clutter up the code, propagate lies, and lend to general confusion and disorganization

**Journal Comments**

* It is a process of adding a comment to the start of a module every time they edit it.
* Long journals are just more clutter to obfuscate the module.

**Noise Comments**

* It restates the obvious and provide no new information.

**Scary Noise**

* Comments that are written out of some misplaced desire to provide documentation.

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

* Example of this is:

****

When you can use this instead:



**Position Markers**

* Position Markers are like banners and if you overuse banners, they will fall into the background noise and be ignored.

**Closing Brace Comments**

* It serves only to clutter the kind of small and encapsulated functions that we prefer. So if you find yourself wanting to mark your closing braces, try to shorten your functions instead.

**Attributions and Bylines**

* There is no need to pollute the code with little bylines.
* They tend to stay around for years and years, getting less and less accurate and relevant.
* Source code control system is a better place for this kind of information.

**Commented-Out Code**

* It gathers like dregs at the bottom of a bad bottle of wine.

**HTML Comments**

* HTML in source code comments is an abomination, it makes the comments hard to read in the one place where they should be easy to read.

**Nonlocal Information**

* Don’t offer systemwide information in the context of a local comment.

**Too Much Information**

* Don’t put interesting historical discussions or irrelevant descriptions of details into your comments.

**Inobvious Connection**

* The connection between a comment and the code it describes should be obvious.
* The comment and the code should be able to understand easily.

**Function Headers**

* Short functions don’t need much description. A well-chosen name for a small function that does one thing is usually better than a comment header.

**Javadocs in Nonpublic Code**

* Generating javadoc pages for the classes and functions inside a system is not generally useful, and the extra formality of the javadoc comments amounts to little more than cruft and distraction.