# **Chapter 3: Functions**

**Small**

* **1st Rule:** Functions should be small
* **2nd Rule:** Functions should be smaller than that

**Blocks and Indenting**

* Blocks within conditional statements should be one line long, a function call.
  + This makes the function easy to read and understand

**Do One Thing**

* Functions should do one thing. They should it well. They should do it only.
  + A function should perform a specific task and not be added by additional responsibilities or logic as it is easier to understand, maintain, and test.

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

* An effective technique for keeping the abstraction level consistent.
* It is the key to keeping functions short and making sure they do "one thing."

**Switch Statements**

* It's hard to make a switch statement that does one thing as its nature is to always do N things.
* By ***polymorphism***, we can make sure that each switch statement is buried in a low-level class and is never repeated.

**Use Descriptive Names**

* Using a descriptive name will describe what the function does. As it easy to understand the intent of the function.
* Choosing descriptive names will clarify the design of the module in your mind and help you improve it, but be consistent in your names.

**Function Arguments**

* The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification
* Output arguments are harder to understand than input arguments.

**Common Monadic (one) Forms**

* Two common reasons to pass a single argument into a function:
  + Asking a question about that argument
  + Operating on that argument, transforming it into something else and returning it.
* Choose names that 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

**Flat Arguments**

* Flag Arguments are ugly.
* It complicates the signature of the method.

**Dyadic (two) Forms**

* A function with two arguments is harder to understand than a monadic function.
* 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 (three)**

* A function that takes three arguments.
* The issues of ordering, pausing, and ignoring are more than doubled.

**Argument Objects**

* Reduce the number of arguments

**Argument Lists**

* If the variable arguments are all treated identically, then they are equivalent to a single argument of type List.
* Function that takes variable arguments can be monads, dyads, or even triads.

**Verbs and Keywords**

* The function and argument should form a very nice verb or noun pair.

**Have No Side Effects**

* Side effects are lies.
* Function promises to do one thing, but it also does other hidden things.
* Side effect creates a temporal coupling.
  + You should make it clear in the name of the function, when having a temporal coupling.

**Output Arguments**

* Most naturally interpreted as inputs to a function.
* Output arguments should be avoided.

**Command Query Separation**

* Function should either do something or answer something, ***but not both***.
* Doing both often leads to confusion.
* The real solution is to separate the command from the query so that the ambiguity cannot occur.

**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 separate the error processing code from the happy path code and can be simplified.

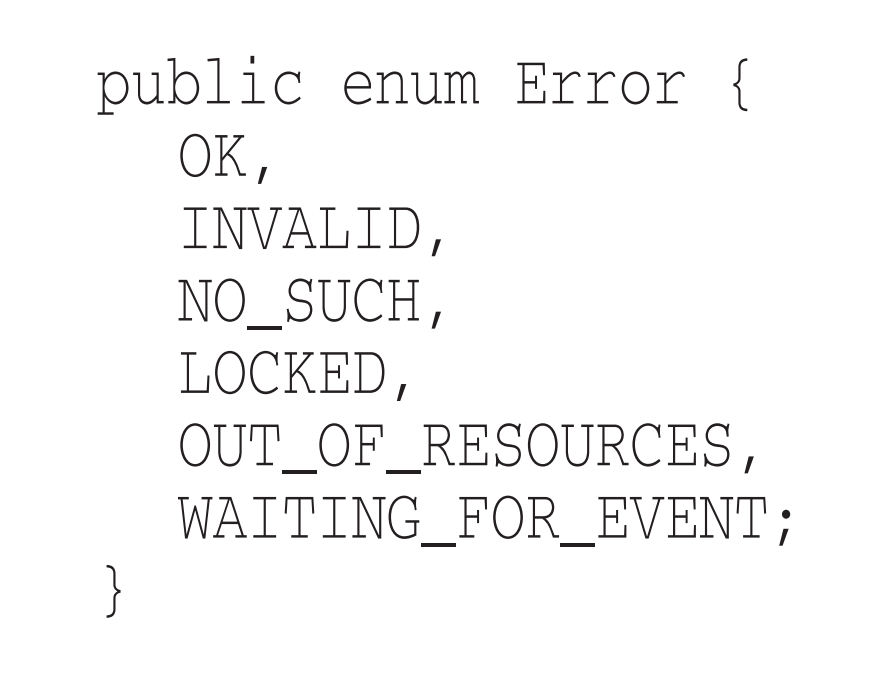
**Extract Try/Catch Blocks**

* Try/catch blocks are ugly in their own right.
* Better extract the bodies of the try and catch blocks out into functions of their own.

**Error Handling Is One Thing**

* Function should do one thing, as well as Error Handing.
* 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.
* Example of Dependency Magnet
* 
* New exceptions are derivatives of the exception class. They can be added without forcing any recompilation or redeployment.

**Don’t Repeat Yourself**

* Duplication is a problem because it bloats the code and will require four-fold modification should the algorithm ever have to change.
* Duplication may be the root of all evil in software.
* Innovations in software development have been an ongoing attempt to eliminate duplication from our source code.

**Structured Programming**

* Every function, and every block within a function should have one entry and one exit - ***Edsger Dijkstra's rules of structured programming.***
  + There should only be one return statement in a function, no break or continue statements in a loop, and never, ever, any goto statements.

# **Chapter 4: Comments**

* The proper use of comments is to compensate for our failure to express ourself in code.
* Comments are always failures
* Programmers should be disciplined enough to keep the comments in a high state of repair, relevance, and accuracy.
* Though comments are sometimes necessary, we will expend significant energy to minimize them.

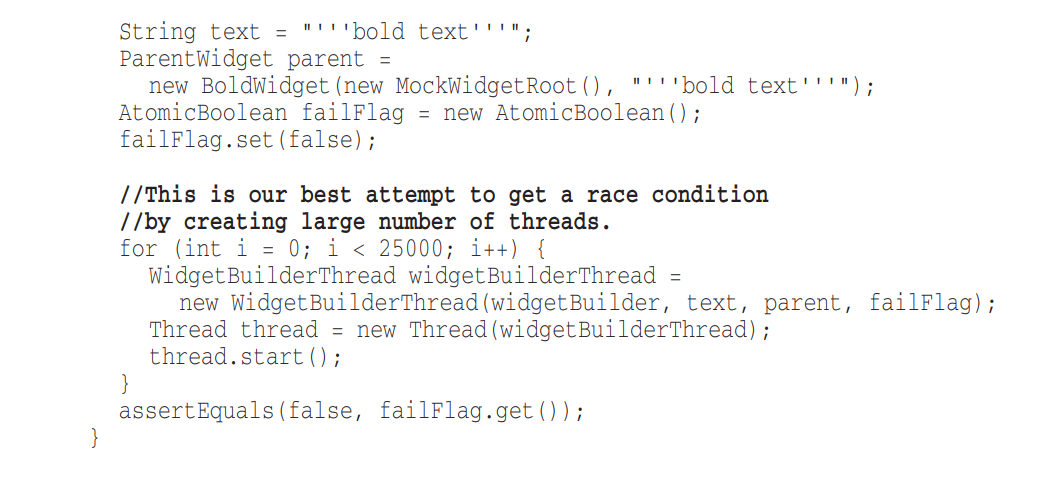
**Comments Do Not Make Up for Bad Code**

* Clear and expressive code with few comments is far superior too cluttered and complex code with lots of comments.
* Spend more time cleaning the mess rather than writing some comments about your mess

**Explain Yourself in Code**

* Explain it according to its intent.

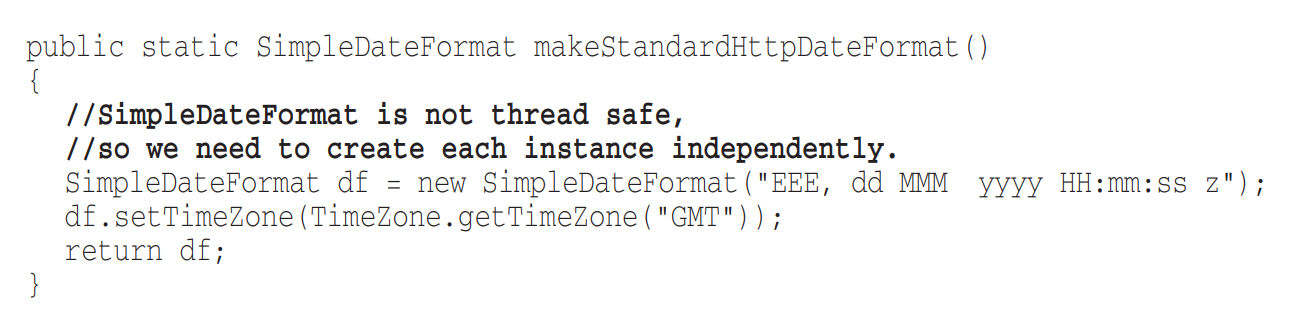
**Good Comments**

* Only true good comment is the comment you found a way not to write.
* **Legal Comments**
  + Copyright and authorship statements are necessary and reasonable things to put into a comment at start of each source file.
* **Informative Comments**
  + Useful to provide basic information with a comment.
  + It is better to use the name of the function to convey the information where possible
* Explanation of Intent
  + Sometimes a comment goes beyond just useful information about the implementation and provides the intent behind a decision. In the following case we see an interesting decision documented by a comment.
  + 

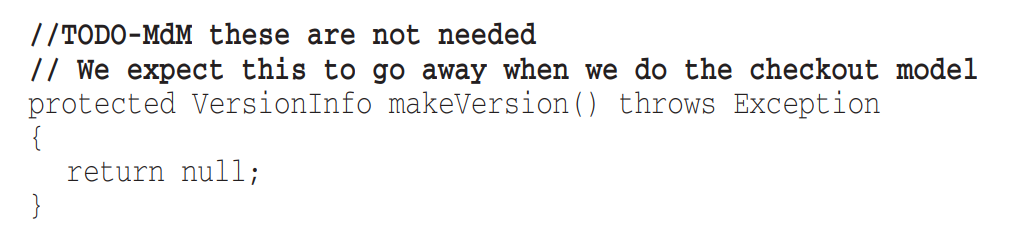
**Clarification**

* 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**

* The comment, while flippant, makes the point pretty well.
* It will prevent some overly 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.
* 
* **Bad Comments**
  + Whatever else a TODO might be, it is not an excuse to leave bad code in the system.

**Amplification**

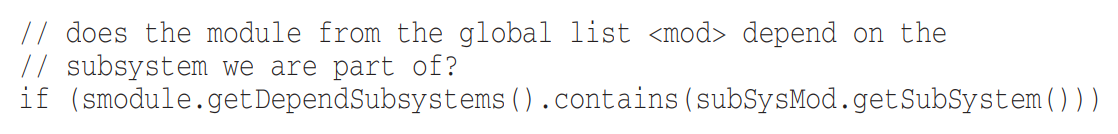
* A comment may be used to amplify the importance of something that may otherwise seem inconsequential.

**Javadocs in Public APIs**

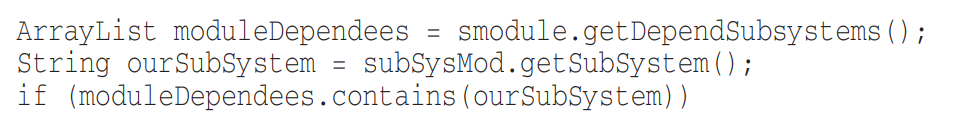
* If you are writing a public API, then you should certainly write good javadocs for it.
* Javadocs can be just as misleading, nonlocal, and dishonest as any other kind of comment

**Bad Comments**

* Most comments fall into this category.
* **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**
  + Comment that is repetitive is not more informative than the code. It does not justify the code, or provide intent or rationale.
* **Misleading Comments**
  + This could another programmer to blithely call this function in the wrong way. That programmer would then find himself in a debugging session trying to figure out why his code executed so slowly or wrongly.
* **Mandated Comments**
  + Comments like this just clutter up the code, propagate lies, and lend to general confusion and disorganization.
* **Journal Comments**
  + These comments are just more clutter to obfuscate the module. They should be ***completely removed.***
* **Noise Comments**
  + They restate the obvious and provide no new information.
  + Replace the temptation to create noise with the determination to clean your code. You’ll find it makes you a better and happier programmer.
* **Scary Noise**
  + Javadocs can also be noise with a purpose of nothing.
  + They are just redundant noisy comments written out of some misplaced desire to provide documentation
* **Don’t Use a Comment When You Can Use a Function or a Variable**
  + Example:



* + Instead:



* **Position Makers**
  + They are clutter that should be eliminated - especially the noisy train of slashes at the end.
* **Closing Brace Comments**
  + Although this might make sense for long functions with deeply nested structures, it serves only to clutter the kind of small and encapsulated functions that we prefer.
* **Commented-Out Code**
  + Commented-out code gathers like dregs at the bottom of a bad bottle of wine.
* **HTML Comments** 
  + If comments are going to be extracted by some tool (like Javadoc) to appear in a Web page, then it should be the responsibility of that tool, and not the programmer, to adorn the comments with appropriate HTML.
* **Nonlocal Information**
  + If you must write a comment, then make sure it describes the code it appears near.
  + 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 purpose of a comment is to explain code that does not explain itself. It is a pity when a comment needs its own explanation.
* **Function Headers**
  + 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.