Documentation: What and Why?

An initial note: This guide is about documenting code, ***not*** about documenting the software created by that code. These concepts are related, of course, but are sufficiently distant that it is not sensible to cover them in the same guide.

In order to talk about code documentation, we first need to agree on what code documentation ***actually is***. This question is not quite so simple as it might initially seem; documentation comes in a wide variety of forms (e.g. example projects, usage guides, API docs, comments) and can be located in a wide variety of places (e.g. in-source, github markdown files, hosted webpages). What's more, some documentation is not even necessarily ***material*** - the choice of proper variable names and code structure can itself be a form of documentation ("self-documenting code"). The boundaries are fuzzy - concerns of documentation cannot be cleanly separated from other concerns of code architecture/style/implementation, and so documentation best-practices cannot be considered in isolation, either.

For the purposes of this guide, we will take "code documentation" to mean "material and practices meant to ***facilitate understanding of software tools.***" This is a very broad definition, and this guide will, in places, touch on the overlap between documentation and other code design concerns. Inevitably, much of the content will be about "traditional" forms of software documentation - that is, documents that describe code. Whenever possible, however, emphasis will be placed on the connection between this sort of documentation and the broader concept of documentation, and care will be taken to link the recommended content and form of the "traditional" documentation to broader concerns of style and coding philosophy.

We also need to concern ourselves with ***why*** we document code in the first place. This issue is somewhat less-subtle: we document code because failing to do so (eventually) renders it useless. When a tool is not documented, the knowledge of its use resides exclusively in the minds of the people currently working on or with it - often, this is little more than a single developer. Once a person ceases to work with the tool, their knowledge of its is eventually lost, both to others and even themselves (how well do you remember code you wrote two years ago?). Once working knowledge of the tool is no longer available, it is ***almost always*** more appealing to make a new tool than to learn the old tool without a guide. Thus, ***undocumented code is*** *throw-away code*. It is not possible to build a reusable tool without documentation, no matter how elegant.

Therefore, the fundamental purpose of documenting code is ***to make it easier to reuse the code than it is to rewrite it from scratch.***

The Fundamental Rule of Documentation

The purpose of documenting code is to make it easy to reuse. ***Incorrect*** documentation makes code ***harder*** to reuse. Incorrect documentation is ***worse*** than no documentation.

Documentation is not, as a rule, self-maintaining. Updating documentation takes time and effort. The more documentation a project has, the more time and effort must be spent keeping that documentation consistent with changes to the code.

Therefore, the fundamental rule of documentation is to ***never write more documentation than can be reasonably maintained.***

Violating this rule will not only waste time, but actively hurt your project. Match the scope of your documentation to the scope of your project - smaller projects naturally require less documentation than larger projects. ***Always*** consider how much effort will need to be made to keep a piece of documentation up-to-date before writing that documentation in the first place.

Documentation as Layers

Good code documentation is pervasive and multileveled. Often, a software engineer joining a new project will ask "where is the documentation?" On a robustly-documented project, the answer to this question is: "everywhere."

Understanding of a tool occurs at multiple levels. At the "highest" level, there are the macro-scale questions about the tool's purpose and higher-order architecture:

- "What problem does this tool attempt to solve?"
- "When might I want to use this tool?"
- "What are the driving philosophies behind the design of this tool? How are those driven by the tool's specific problem-space?"
- "What alternative tools should I consider?"

Slightly further down the abstraction-hierarchy, we have broad mechanics-oriented questions:

- "Where do I start after I've decided to use this tool?"
- "What features does this tool actually support?"
- "What does this tool look like when integrated into a typical project?"
- "What are the basic 'parts' of this tool, and how are they organized?"

Continuing downwards, we start to get into mechanical details:

- "How do I customize this tool to do specific tasks?"
- "What language features do I need to know to use this tool?"
- "What are the common mistakes that I can/will make when using this tool?"
- "How is this tool's interface likely to change in the future?"

And then, finally, implementation details:

- "How do the parts of this tool actually work?"
- "How might I modify parts of this tool to do things that I want, but that it does not yet support?"

All of these levels are important, and all of them require documentation. But the ***type*** of documentation required by each is not the same; high-level architectural decisions can be explained by technical prose, but low-level implementation details will almost always require either pseudocode or actual source.

Additionally, these levels distinguish themselves by their differential ***rates of change***. Low-level implementations tend to be volatile, while high-level problem-space constraints tend to move relatively slowly. This naturally leads to shearing layers, as things which change at similar rates end up factored together in code - and, likewise, in documentation.

Just as our actual understanding of a system "cleaves" naturally into layers based on levels-of-abstraction, so too does the ideal structure of project documentation. So, "where is the documentation?" It should be woven throughout. The structure of the documentation should match the structure of our understanding of the system.

Therefore, documentation is implemented in a layered fashion. In particular, code documentation for our software projects should have three layers:

1. **Usage Documentation**: This includes wide-scope overviews of the tool and its purpose, example projects, code excerpts, and design guides. This layer is usually the first one to be encountered/read by a

new user.
2. **API Documentation**: This includes rigorous reference material for all of the "surface" parts of the tool (i.e. public-facing API classes and methods). This layer is usually primary reference for an ordinary user making typical use of the tool.
3. **Source Documentation**: This includes comments, meaningful naming conventions, and best-practices for code design and readability. This ensures that advanced users are able investigate, debug, and/or modify the tool when necessary.

It is important to note that this "layering" is not perfect: these layers overlap, and depend on each other. Usage documentation should refer extensively to API documentation. API documentation is often source-generated, and so overlaps with best practices for code commenting. Nevertheless, this provides us with a baseline framework with which we can situate our best-practices.

The standards for each layer will be covered below in reverse-order, as that is the order in which the documentation is naturally developed during the course of code development.

Source Documentation: Style, Structure, Naming, and Comments

Source documentation is the most basic and fundamental form of code documentation. *All* projects have source documentation: at base, the code itself can be always considered a piece of documentation.

That said, not all source documentation is created equal. Some code elegantly explains itself; other code is deliberately impenetrable. Code readability (as with many things in code) follow the principle of least surprise - familiar patterns require less thought, and thus are easier to interpret.

This guide will set standards for the following four aspects of source documentation: **Code Style**, **Code Structure**, **Naming**, and **Comments**.

Code Style

A uniform code style is ***crucial*** to code readability. Consistency in style within a project is ***absolutely mandatory,*** and consistency across projects is highly desirable.

The foundation of a uniform code style is a coherent set of standards. Since cross-project consistency is desirable, it is almost always a good idea to ***begin with an existing set of standards***, rather than to write one's own. Accordingly, all MRAS coding styleguides are based on existing standards. Modifications are permitted, but should be both justified and, if possible, minor.

The following is a list of MRAS Application Team coding styleguides. NOTE: This list is still under-construction.

- Java: https://github.com/Oblarg/MREStuff/blob/master/javastandards.md

Code Structure

Adhering to style standards normalizes syntactic details, but it generally does nothing to normalize the general organization and design of the code. These are just as important, if not moreso, to rendering one's code readable by others. While it is impossible to give a fully-general set of standards for how one should structure their code - this is driven by details of the problem space in which the project is situated - there are nonetheless some general principles which it is a good idea to follow.

NOTE: To quote George Orwell: "Break any of these rules sooner than say anything outright barbarous." None of these are absolutes.

Separation of Responsibilities (AKA: Each Piece of Code Should Have a Job)

When attempting to understand a piece of code, one of the most natural and effective strategies is to attempt to break the code into pieces and figure out what each piece is responsible for. Humans have limited working memory; it is almost impossible to understand the entirety of a nontrivial software project at the same time.

Unfortunately, it is very easy to write code for which this strategy is doomed to failure, because either:

a) The code does not separate into smaller "pieces" (functions, classes, namespaces, whathaveyou) in the first place, or b) The pieces the code separates into do not reflect any sort of sensible division of the code's functionality

This is often the case with "proof-of-concept" code, which grows rapidly and hapazardly as the programmer becomes familiar with the problem-space and new requirements are encountered. Often, no thought at all is put into ***where*** new functionality should be added; this is determined by immediate convenience, or pure happenstance.

Therefore, ensure that your code separates into pieces, ***and*** that each piece has a clear functionality that can be understood without simultaneously understanding the rest of the code in full detail. In object-oriented programming, this is often best-accomplished by making wise choices as to one's class structure - but the concept is applicable in all contexts.

Avoid Globals (AKA: Pass Information Locally)

Global variables are almost always a bad idea. Since a global variable can be modified or read from anywhere in the code, they directly contradict the notion of factoring the code into chunks that can be understood in isolation. Code that makes heavy use of globals tends to be extremely difficult to read, and even harder to modify/debug.

There are, of course, some cases where globals are unavoidable. There are others when they are defensible design choices (e.g. a service that everything does actually need to interface with, and that there will never be more than one of). But if you find yourself using a global variable to pass state from one part of the program to another, you have almost certainly done something wrong.

Do Not Repeat Code Unnecessarily (AKA: Abstraction Is Good)

Figuring out what a piece of code does is hard enough. Having to figure out what the same piece of code does multiple times over is infuriating. Repetition is not just an inefficiency in ***writing*** code, but in ***reading*** it, as well.

Abstracting repeated code into a subroutine or class serves not only to remind the reader that they already know what it does, but it also allows them to view the code as a "black box" with a set of defined inputs and outputs. As mentioned, the most effective way to understand complex systems is by breaking them down into small chunks that can be understood in relative isolation; repeated code almost always represents such a chunk, and so the structure of your program should reflect that.

Therefore, when you find yourself repeating a chunk of code, strongly consider creating a reusable abstraction. Copy-pasting is a sure sign that abstraction is a good idea.

Do Not Overabstract (AKA: Abstraction Is Bad)

Abstraction is great, right up until it isn't so great. Perhaps the only thing worse than having to mentally replace a lengthy, repeated block of code with an imagined summary due to the lack of a sensible abstraction, is having to "unwrap" a summary of a summary of a summary of a summary only to find the equivalent of four lines of code at the bottom.

(For an intentionally egregious example of this, see Enterprise FizzBuzz)

This can occur for a number of reasons. Top-heavy, "waterfall" development practices often result in overabstracted code because the major design decisions are made before it becomes clear where flexibility is needed and where it is not. Even in less-structured environments, the temptation to allow for future expansion before the need for such has demonstrated itself is pervasive. "Scope creep" can eventually eat even the simplest of projects, as the incremental appeal of "just one more feature" is almost always more immediately salient than the eventual cost of repeatedly accumulating additional complexity.

Compounding this, developers familiar with a project find themselves in a uniquely poor situation to judge the onset of overabstraction. Familiarity with the code "collapses" the layers as they compound - those close to the code build mental shortcuts to "cut through the cruft." Even after the intent of the code has long-since become entirely obscure to the outside observer, the original developer retains privileged knowledge of its original purpose and design - at least, until they spend a sufficient amount of time away from the project, at which point they are often rendered as clueless as anyone else.

Therefore, stay constantly vigilant against "abstraction-creep." Do not add more abstraction to a project to facilitate a new feature until/unless it is ***certain*** that the new feature is needed. Beware scope-creep. Constantly solicit feedback on the clarity of your design structure from those ***without*** intimate familiarity with the project.

Avoid Deeply-Nested Control Statements (AKA: The Actual Reason for Column Limits)

```
if (cond1) {
  if (cond2) {
    for (i=0; i<10; i++) {
      while (cond3) {
        if (cond4) {
          // You've made a terrible mistake
```

Hopefully it is clear by now that the key to readable code is *not requiring the reader to think about too much at once.*

One of the easiest ways to violate this principle is by haphazardly nesting control statements. This is exceedingly easy to do while writing code, as the programmer (who knows what they intend to do) can build such nests from the "inside-out," limiting the noticeable complexity at each stage of growth. Unfortunately, it is also exceedingly difficult to comprehend later, even by the programmer who originally wrote the code, as one is forced to read the resulting mess from the "outside-in," keeping track of all of the previous levels at each stage. As the number of paths the programmer has to keep track of grows roughly exponentially with the depth of nesting, this rapidly becomes utterly intractable.

Additionally, the sheer visual noise of having many layers of indentation itself poses a barrier to readability, making it easy to lose track of identation levels and, eventually, requiring an unreasonable number of line continuations as the code butts up against the column limit. This is, actually, a ***feature*** of a conservative column limit in many coding standards; rather than placing a limit on the number of nested control statements, the strict column limit simply makes overly-deep nesting pragmatically unusable as the programmer runs out of space.

Therefore, do not deeply nest control statements. Good strategies for avoiding this include liberal use of early return/continue, collapsing nested if statements into single-level if statements with compound conditions, and, if all else fails, encapsulating some of the complexity in a subroutine. If the problem seems intractable even with these tools, this is likely a symptom of a fundamental problem with your code design.

Keep Your File Structure Orderly (AKA: Where the Heck Is the `main` Function?)

Your code itself may be extremely elegant and self-explanatory, but all of this is for naught if it is buried in a cluttered directory. Often, a new programmer entering a large project is overwhelmed by the sheer quantity of files. If the structure of the project does not make it clear which files are responsible for what ***without*** reading the code, the programmer is forced to engage in a blind search through the source - tracing calls from file to file - as they attempt to piece together the general architecture of the project.

This is an *infuriatingly difficult* process. Even if the code in each file is easy to read, the search time alone as one attempts to "back-trace" the control flow without any guidance can render a project uninterpretable. And if the code *isn't* easy to read, well, there's no hope at all. In the worst case, it can be impossible to even find the file responsible for primary program flow (e.g. one containing a `main()` function), leaving the programmer grasping at an assortment of parts with no idea what is determining how they are put together.

Therefore, always ensure that your project directory is sensibly-structured. Define a directory structure that matches your project structure, and document it in a README included in the root directory. *Always* consider how hard it will be for someone unfamiliar with the project to find the important files that control program flow, so that unfamiliar readers can work from the "inside-out" rather than from the "outside-in."

Use Standard Tools When Possible (AKA: No One Uses a DVORAK Keyboard)

The Principle of Least Surprise applies almost everywhere, including one's choice of which external tools to use in one's project.

Any time you use a nonstandard tool, anyone reading your code likely must also learn that tool. If you use a tool they're already likely to know, that's one less task for them to handle.

The standard tools are not always the "best" tools, when judged purely on their merits with respect to the technical problem-space of the project. However, as we've established, if no one is willing to re-use your code in the first place because they can't be bothered to figure out how it works, any marginal technical benefit from using a more-obscure tool ***won't matter***. Technical decisions have to be made in a broader context.

Therefore, always use the standard tool when the standard tool will work acceptably, even if a more-obscure tool might work slightly better. Always weigh the technical benefit of an alternative tool against the readability cost of diverging from the standard. Never use an obscure tool unless it is either extremely simple to learn or absolutely no standard tool will do.

Naming

One of the worst features of programming education, on the whole, is the canonical use of meaningless single-character variable names in pedagogical examples:

```
int i = 0;
char c = '%';
String s = "why is this practice so common";
```

Variable names are often treated as arbitrary placeholders whose meaning is granted by the context of the surrounding code. It's not hard to understand the origin of this mindset: from the point of view of the compiler (and thus of the machine executing the code), this is entirely accurate. Computer scientists are often all too happy to willfully perpetuate this mindset, with "coder culture" glorifying minimalistic code and machinelike images of the "ideal" programmer. If the computer does not care whether your integer is named `i`, `intVal`, or `rumplestiltskin`, why should you?

Of course, the obvious reason is that humans are ***not*** computers. It is a great irony that while we humans ***understand*** the code we write, the computers responsible for running it do ***not.*** Computers are dumb - computer code is precisely the means through which an intelligent actor (i.e. a human) instructs a fundamentally dumb machine to perform complicated tasks.

You don't read code like a computer reads code, and so you shouldn't write code to be read that way, either. The computer can "read" it equally well either-way - but you cannot, and the computer is not really "reading" it to begin with!

As variable names are one of the least-constrained aspects of a piece of code, they are naturally also one of the most important vehicles for facilitating understanding of the code. A variable name is, itself, a crucial piece of documentation. Choose it wisely!

Name Variables After Their Jobs (AKA: `i` Does Not Tell You Anything Useful)

A programmer usually cares more about what a variable ***does*** than what a variable ***is.*** Seeing that a variable is named `i` tells the reader that the variable is probably an integer; but that's one of the least-useful pieces of information we could encode in the variable name, as the declaration of the variable should make that obvious (in dynamically-typed languages, this is less the case, but that's a real shortcoming of dynamically-typed languages).

What a variable ought to be named is what a programmer is most likely to need reminding of when they see the variable. In almost all cases, this is "the variable's role in the piece of code being read."

Therefore, a variable's name should be a description of its *job*. If an integer is being used as a counter, name it `counter`.

Disambiguate Names by Increasing Specificity (AKA: Do Not Name Your Variables `Counter1` and `Counter2`)

In keeping with the above advice, you'll often run into a scenario where multiple variables have similar jobs. While writing code, it is ***extremely tempting*** to disambiguate the names of such variables by simply appending an additional token of some sort, without putting much thought into the token itself. The immediate concern to the

code's author is to have two different names, not necessarily to make sure that the difference in the names is itself a conveyor of meaningful information.

But to the reader of the code, such a disambiguation is useless; it is merely a sign that one must now tediously inspect the source to attempt to find the ***actual*** difference in the roles of the two variables. This takes time and effort - the minimization of which is the whole point of documentation.

Therefore, when disambiguating the names of two similar variables, *describe their jobs in greater detail*. Instead of `counter1` and `counter2`, disambiguate the variables with *what they are counting* (e.g. `instanceCounter`).

Disambiguate Important Names Preemptively (AKA: `Parameterizer` Probably Describes Multiple Things)

Applying the above advice while building a piece of software can be made much more difficult by the fact that, by default, we humans do not see the future.

A function, class, or variable may have a perfectly acceptable name, right up until the addition of another piece of code renders it ambiguous, at which point the programmer is forced to (potentiously arduously) go back and change the original (or, often, to give up and slap a `2` on the end of the name of whatever new thing is causing the problem - please do not do this).

It is important to note that sometimes, changing things as problems emerge is the *right thing to do*. Preemptive optimization can ruin projects. ***However***, sometimes it can be predicted when such a change would be particularly painful - for example, the name may be part of a public API (rendering any such change breaking). In these cases, it can help to take precautions.

Therefore, when a name is likely going to be difficult to change in the future, be sure to choose a sufficiently-specific name to begin with.

Make Compromises When Things Become Unweildy (AKA: No One Wants to Read `AppletWidgetInstanceParameterizerFactoryFactory`)

Descriptive naming does have limits. Occasionally, in pursuit of providing everything with unique and semantically-meaningful names, monstrosities are created. This is a common point of mockery for Java APIs (particularly Spring), which are liable to end up with class names so long and obtuse that they are indistinguishable from machine-generated jibberish.

Therefore, never be so specific with your naming that the sheer length of your names renders them unhelpful.

Comments

Finally, let's talk about comments. Comments are widely (and correctly) regarded as a crucial element of code documentation. In fact, in a lot of discussion, they're the *only* element discussed. This is somewhat problematic - "comment your code" is very vague advice, in the abstract. Comment it with *what?*

Effective code commenting ***must*** be done in combination with the other documentation practices discussed in this document. An awful lot of important information is ***not*** effectively conveyed through comments, and attempting to "make up" for shortcomings elsewhere in code readability through comments is usually a doomed endeavor.

Comments Are Not Translations (AKA: Line-by-Line Explanations Are Useless)

The following is an example of (regrettably) common commenting practice:

```
public int getNonNullEntries() {
  // Initialize counter
  int i = 0;
  // Loop over entries
  for (Entry e : entries) {
    // Check if entry is nonnull
    if (e != null) {
      // Increment counter if entry is nonnull
      i++;
    }
  }
  // Return counter value
  return i;
}
```

Not a single comment in the above example is particularly useful to the reader. The comments have simply translated the code, line-by-line, into pseudocode. A reader fluent in the language obtains no useful information from any of the comments; they are redundant, and waste space.

To the extent that these sort of comments provide any help at all, it is merely to "cover up" documentation mistakes in the source itself - for exmaple, we would not need a comment to tell us that `i` is a counter if it had simply been named `counter` (or, better, `entryCount` or `nonNullCount`).

Consider instead:

```
int getNonNullEntries() {

  int entryCount = 0;

  for (Entry entry : entries) {
    if (entry != null) {
      entryCount++
    }
  }

  return entryCount;
}
```

This is strictly more-readable, despite having no comments at all! In fact, a method this simple should likely *not* have any comments.

Therefore, do not generally use comments as natural-language translations of individual lines of code.

Of course, exceptions can be made for particularly obscure lines, which even a competent reader might have trouble otherwise parsing (though, if you find yourself using such lines often, it is probably bad coding practice on

your part).

Good Comments Are Summaries (AKA: The Map Is Smaller than the Territory)

The point of documentation is to make code easier to understand. Comments, as they are natural-language descriptions of the code, must therefore be easier to understand than the code itself. In most cases, this means that a comment should *summarize* the content being commented.

One of the reasons the line-by-line commenting described above does not work is that it is very hard to summarize a single line of code (provided that line is of reasonable length and complexity). In order to summarize something, details need to be omitted. To provide a reasonable choice as to which details can be left out, comments need to summarize a sufficiently-large block of code.

Thus, the choice of how large a "chunk" of code to summarize is important, and is one of the major determining factors as to whether one's comment is helpful or not. This ties directly into code structure - well-structured code offers clear points at which summaries are appropriate.

Therefore, use comments to describe a piece of code large enough to be meaningfully summarized. Try to guess at what points the reader will want high-level descriptions of the code. Don't comment a piece of code small enough to be easily understood by itself.

Note that many high-level summaries should actually be placed in source-generated documentation rather than in ordinary comments.

Comments Can Be Visual Aids (AKA: Delimeters Are Good)

One of the most important parts of keeping a large source file readable is ensuring that it cleaves readily into smaller "chunks." If it does not do this, navigating the "wall of text" can be nearly impossible.

Often, the structure of the code itself serves to do this (the code may factor cleanly into smaller objects, subroutines, etc). Sometimes, however, this is not the case. In these cases, comments are your friend - simply providing a visual indication of the on-page extent of a conceptual "block" of code is a crucially-important function.

Therefore, when code does not cleave itself naturally into smaller chunks, use comments to provide visual guidance as to the "pieces" of the code.

API Documentation

API Documentation provides the intermediate stage between source documentation and usage documentation - it consists of a collection of summaries of tool features adhering to a standard format, such as can be used as a reference by a programmer working with the tool.

API Documentation should generally be source-generated - i.e., auto-generated from comments located in the source. This allows the API documentation to serve "double-duty" as in-code documentation, removing the need for duplication of summaries between comments and API docs.

Use Source-Generated Documentation

Good commenting practice, as mentioned earlier, involves summarizing code for the reader. Often, code naturally presents us with convenient summarizeable "chunks" - in object-oriented programming, especially, methods and classes are extremely natural points for summarization.

Here, there exists an extremely fortuitious overlap in purposes - summarization of classes and methods is not only extremely beneficial for one reading the source, but those summaries can, when gathered together and presented in a structured manner, serve to summarize all the important public-facing points of the API. Accordingly, a number of tools exist to generate these API docs from specially-formatted comments in the source.

Using these tools not only removes the need to write separate API docs, but also (and more importantly) removes the need to *maintain* those docs.

Therefore, all projects should use `Javadoc <https://en.wikipedia.org/wiki/Javadoc>`__-style source-generated API docs. Almost all languages now support docs of this sort (e.g. `doxygen <http://www.doxygen.nl/>` for C++/C#), and the consistency in style offered by the use of uniform tools is a tremendous asset. For brevity, term "Javadoc" will be used to refer to all of these for the remainder of this document.

Don't Restate the Function Name (AKA: Everyone Already Knows What a Getter Does)

```
/**
 * Gets the date.
 *
 * @return The date.
 */
public Date getDate() {
  return date;
}
```

The first line of text in the javadoc above serves absolutely no purpose other than to add additional lines to the source file.

Therefore, omit descriptions that merely duplicate information:

```
/**
 * @return The date.
 */
public Date getDate() {
  return date;
}
```

Be Detailed (AKA: Why Didn't It Tell Me It'd Do That?)

It's important to remember that the person reading your documentation ***does not know what you know*** (even if that person is you at some point in the future!). The purpose of API documentation is to ***summarize the important information about a piece of code***. This doesn't just mean describing what the thing generally does -

the name of the thing can and should do that already. This means noting anything that might be surprising/unobvious about the functionality.

Therefore, don't do this:

```
/**
 * @return The current heading.
 */
public double getHeading() {
  return Math.IEEERemainder(heading, 360);
}
```

Instead, do this:

```
/**
 * @return The current heading in degrees, from -180 to 180 (multiple
turns will wrap).
 */
public double getHeading() {
  return Math.IEEERemainder(heading, 360);
}
```

Capitalize and Use Periods (AKA: This Isn't Really That Important, We Just Need a Style Standard)

Nothing's uglier than an API doc with inconsistent style. While there are multiple standards to choose from, and all of them are more-or-less equally fine, the easiest one to remember by far is, simply, "all text in the javadoc should be be formatted as complete sentences."

Therefore, all text in the javadoc should be formatted as complete sentences (i.e. be capitalized and use periods). This can seem a bit odd for param and return value descriptions, but it's simple and easy to remember.

Remember to Link to Related Documentation (AKA: Don't Make Me Dig Through the Documentation)

When a javadoc summary mentions another class or method and ***doesn't*** link to that class or method, the reader is forced to waste time looking for it. Almost all javadoc-style API docs support linking to other generated docs.

Therefore, always link to the documentation for mentioned code.

Always document params and return values (AKA: Inputs/Output Are Important)

Source-generated documentation systems basically always include features for individually documenting the parameters and return values of methods. Most editors support auto-generation of the required tags (usually `@param` and `@return`). If you ***don't*** set these up, the parameters will still have "descriptions" in the source-generated docs, but they will be empty.

Therefore, always do this.

Usage Documentation

The final "layer" of documentation to be written for a project is ***usage documentation***. This consists of large amounts of prose and supporting media (pictures, example code snippets, etc), and is intended to orient a new user as to what the project is, and how it is intended to be used.

For MRWolf projects, usage documentation will be contained in Sphinx projects. This provides a standard format and look for our documentation, and also supports a large number of standard and community-written features that make writing and maintaining our documentation easier.

The template for an MRWolf Sphinx Documentation Project can be found here: https://github.com/MRWolves/sphinx-template

Usage documentation and standards for the above template can be found here: https://sphinx-tutorial-temp.readthedocs.io/en/latest/

Remember, again, that you should never write more documentation that you can feasibly maintain. By nature, usage docs are the most involved and time-consuming type of documentation a project can have - small projects do not necessarily need usage docs, and should not have them if maintaining them would involve an unreasonable investment of effort.

Orient New Readers (AKA: Include a "Getting Started Page")

One of the most common pitfalls of usage docs is that, as they are invariably written by those closely familiar with the code, little attention is paid to how easy they are to naviage by someone ***not*** familiar with the code.

This issue has been mentioned before in this document, but it is by far most crucial at this level of documentation - usage docs are the "public face" of your project, and represent the point at which the "ideal reader" is ***furthest*** from the likely author.

A common result of this is that many usage docs lack any sort of clear introduction or orientation for the new reader - after all, the author is already familiar with the structure of the project and knows where to look for the information they want. This mistake is lethal - your documentation can have flawless content, but it is worthless if the user cannot ***find what they need.***

Therefore, always include a "Getting Started" page that directs the reader to the various parts of the usage documentation, and provides context for the project structure.

Do Not Write an Extended Tutorial (AKA: No One Is Going to Read Your Documentation End-to-End)

It is often tempting to write usage docs as one long tutorial. This is ***always*** a mistake.

Tutorials are a good thing - when they're the right tool for the job. They're effective at presenting concepts to a new reader in the order in which they're needed, removing the need for the reader to jump around the reference material trying to figure out which parts are supposed to plug into which other parts, and how. The logical flow of the tutorial supplants the need to search for content - as long as the user is following the tutorial, they will (so long as the tutorial is well-written) encounter almost precisely the information they need, in the order they need it.

By the same token, tutorials are impossible to navigate by any reader that is ***not*** following the flow of the tutorial. Hence, tutorials tend to make ***terrible*** reference material; once you no longer need a step-by-step guide, the

tutorial layout becomes a hindrance, not a help.

Additionally, tutorials require strong assumptions about what precisely the user wants to do - it is nearly impossible to write a "branching tutorial" without it becoming an unmanageable mess. If those assumptions are not correct, the reader will no longer encounter the correct information in the sequence of the tutorial, and will be frustrated. Moreover, the longer the tutorial is, the stronger the assumptions about what the reader actually ***wants to do*** have to be.

Thus, a sufficiently-lengthy tutorial is almost always going to "lose the reader" by its end - and, once the reader is lost, a tutorial becomes an uselessly-structured labyrinth of inscrutable ordering.

Therefore, keep your tutorials short and limited in scope. Arrange them logically with the rest of the documentation, with the goal of making them easily-searchable - do not attempt to write your documentation as a coherent end-to-end narrative. No one will ever read it that way. Allow for "sequence-breaks," and consistently link to other sections when they are referenced.

Do Not Write Only (Or Even Mostly) Tutorials (AKA: Reference Material Is Important)

Tutorials are great introductory material, but they rapidly wear thin as the user gains more than a cursory familiarity with the project. As mentioned above, navigating tutorials - when what you want is assorted information *in* those tutorials rather than a step-by-step buide - sucks.

Therefore, remember to write reference material that describes the library features themselves, rather than just guides that demonstrate how to use them in simple cases.

Use Multilayer Organization (AKA: Factor Related Things Together)

Just like in code, usage docs are much easier to navigate if conceptually-related pages are factored together. If your docs are short, this can be done simply by ordering the pages in a reasonable manner - for more substantial projects, however, this likely will not suffice, and the best solution is often to use a nested structure (just like how this article has sections and subsections).

Therefore, pay attention to your main TOC. If it's overlong, condense sections of related pages into their own subsections.

Link to API Documentation (AKA: Don't Make Me Navigate the Javadoc)

Remember that users encounter documentation from the "outside-in" rather than from the "inside-out." When writing usage documentation, the developer often has the API docs available and open either on their own or as comments in the source), and is able to use them to inform the writing of the usage docs. By default, however, the reader does ***not.***

Therefore, make frequent links to API documentation from your usage documentation. Whenever a new class is introduced, it should be accompanied by a link to the API documentation page for that class. Subsequent mentions of that class need not necessarily include links, though it can be helpful to do so - use your best judgment as to whether it is necessary/helpful.

Seeing "Outside-In" vs. "Inside-Out" - The Pitfalls of Being The Developer

The three layers mentioned above - source documentation, API documentation, and usage documentation - were ordered as they were to match the order in which the ***developer*** usually encounters them. A developer starts with the code, and works outwards towards usage documentation as the product becomes more mature. This progression - from the "inside-out" - fundamentally shapes how the developer sees their code, and their documentation.

In a way, this places the developer in a uniquely bad position to judge the quality of their own documentation - because this is precisely the opposite from the order new users will actually ***experience*** said documentation. A fundamental disconnect therefore exists between those who author a project, and those who encounter it after-the-fact.

Whenever possible, it's important for a developer to try to view their documentation from the "outside-in," as a new user would. Start by reading the most-abstract, highest-level piece of documentation you have. How much knowledge of the project would you need to make sense of it? Is this level of knowledge consistent with the likely experience of the intended audience of the documentation? This is a ***difficult*** exercise, but a necessary one.

And, ultimately, even this really doesn't suffice. You can never truly un-learn what you have learned about the project in the process of writing the thing. The only way to bridge the gap is constant, honest feedback from those on the outside. The definition of "outside," of course, can vary. Not every project is intended to be picked up by complete strangers - judging the likely level of familiarity of your documentation's audience is important. But you can ***never*** write good documentation if the only feedback comes from the people who ***don't need the documentation in the first place.***

A Closing Note: "Would I Hate These Developers?"

On a final note, always remember the question: "If I had to use this tool, would I hate the people who made it?"

The point of documentation is to make your code survive. If the answer to that question is "yes," your code will not survive.