

# Dealing with the Second Hardest Thing in Computer Science

Thoughts on naming things for software development

Indrajeet Patil



Source code for these slides can be found [on GitHub](#).

# What you'll learn today

- Why naming impacts code quality and maintainability
- How naming improves software design and architecture
- Common naming pitfalls to avoid
- Practical strategies for clear, consistent, and meaningful names\*
- Tools and techniques for better naming (AI, code review)



Transform naming from an afterthought into a deliberate practice.

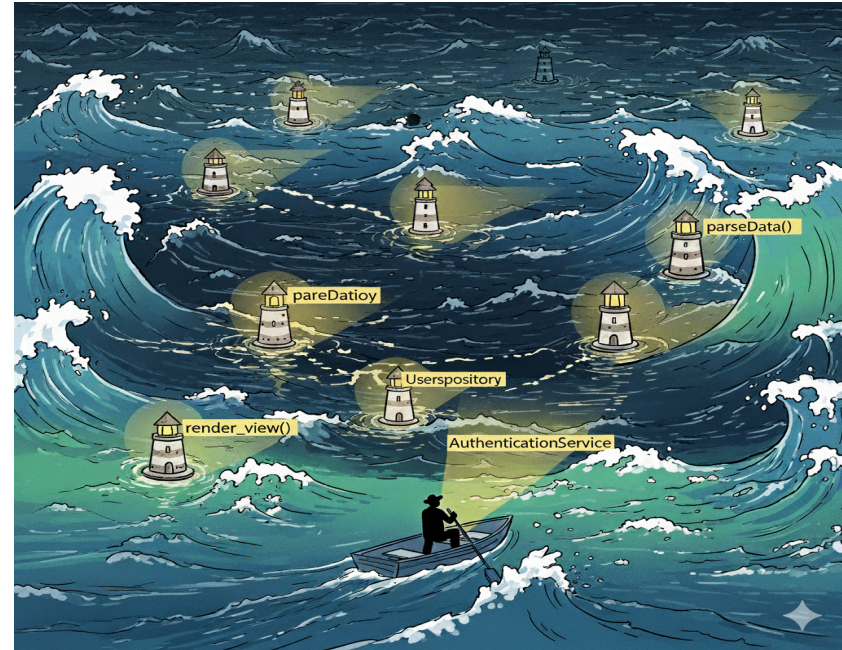
\*Despite Python examples, all the mentioned strategies are **language-agnostic**.

**“There are only two hard things in Computer Science: cache invalidation and naming things.”**

- Phil Karlton

# Why naming matters

*Navigating the codebase with good names as beacons of clarity*



# The hidden cost of poor naming

## Immediate consequences:

- Longer code reviews due to unclear intent
- Debugging becomes detective work
- Extensive onboarding needed for new team members

## Long-term impact:

- Technical debt accumulation from avoidance
- Higher bug introduction rates
- Risky refactoring due to uncertainty



### The multiplication effect

Poor naming spreads confusion throughout the entire system.

# Good names pay dividends

## Development velocity:

- Code reviews focus on logic, not deciphering
- Faster component targeting during debugging
- Confident feature development

## Maintenance benefits:

- Safe and predictable refactoring
- Root cause fixes over symptom patches
- Self-documenting code



### The investment mindset

Time spent on naming is not overhead—it's an investment that pays compound interest.

# Naming and good design

Illustrating benefits of thoughtful naming for software design using *function* as an example

# Following Unix philosophy

Unix philosophy specifies the golden rule for writing good a function:  
*“Do One Thing And Do It Well.”*

Finding a descriptive name for a function can inform us if we are following this rule.  
Consider a function to extract a table of regression estimates for a statistical model.  
For convenience, it also allows sorting the table by estimate.



## Naming is hard

Trying to find a name highlights that the function is doing more than one thing.

```
def extract_and_sort_estimates(model, sort="asc"):
    # code to extract estimates from model
    # code to sort table
    pass
```



## Naming is easy

These individual functions are easier to read, understand, and test.

```
def extract_estimates(model):
    # code to extract estimates from model
    pass

def sort_estimates(table, sort="asc"):
    # code to sort table
    pass
```

Functions with `and` or `or` in their names are dead giveaways that they don't follow the Unix philosophy.



# Function parameter names

When it comes to writing a good function, finding a good name for a parameter can also reveal design problems.

E.g. a boolean or flag parameter name means function is doing more than one thing.

Consider a function that converts Markdown or HTML documents to PDF.



## Boolean parameter necessary

Doing more than one thing.

```
def convert_to_pdf(file, is_markdown=False):  
    if is_markdown:  
        # code to convert Markdown to PDF  
        pass  
  
    if not is_markdown:  
        # code to convert HTML to PDF  
        pass
```



## Boolean parameter unnecessary

Doing one thing.

```
def convert_md_to_pdf(file):  
    # code to convert Markdown to PDF  
    pass  
  
def convert_html_to_pdf(file):  
    # code to convert HTML to PDF  
    pass
```

# Naming: The Do's and Don'ts

*“The beginning of wisdom is to call things by their proper name.” - Confucius*

# The Don'ts

You won't have to remember any of these rules if you follow the following principle:

**Names must be readable for the *reader*, not *author*, of code.**

# Confusion & Similarity

- **Avoid imprecise opposites**, since they can be **confusing**. E.g. parameter combination `begin/last` is worse than either `begin/end` or `first/last`.
- **Don't use hard-to-distinguish character pairs in names** (e.g., `l` and `I`, `o` and `0`, etc.). With certain fonts, `count0` and `counto` look identical.
- **Don't use similar names for entities with different meanings**. E.g. `PatientRecs` and `PatientReps` are easily confused because they are so similar. There should be at least two-letter difference: `PatientRecords` and `PatientReports`.
- **Avoid naming separate entities with homonyms**. Discussing entities named `waste` and `waist` is inevitably going to lead to confusion.
- **Don't use easily confused names**. Names that are too similar make great candidates for mistaken identity. E.g. `nn` and `nnn` are easily confused; use `n_square` and `n_cube` instead.

# Consistency & Standards

- **Don't use inconsistent abbreviations.** E.g. instead of using `numColumns` (*number of columns*) in one function and `noRows` (*number of rows*) in another, choose one abbreviation as a prefix and use it consistently.
- **Don't allow multiple English standards.** E.g. using both American and British English standards would have you constantly guessing if the variable is named (e.g.) `centre` or `center`. Adopt one standard and stick to it.
- **Don't use misleading abbreviations.** E.g., in Python, `str` conventionally refers to string type. Using `str` to mean “structure” in a function parameter will be misleading to other developers.
- **Avoid misleading names.** Names that imply wrong information are worse than no names. Update names when functionality changes. E.g. `accounts_list` should not be used for a dictionary; if `get_means()` changes to computing on-the-fly, rename it to `compute_means()`.

# Clarity & Communication Issues

- **Don't use pop-culture references in names.** Not everyone knows them. E.g. `female_birdsong_recording` is a better variable name than `thats_what_she_said`.
- **Don't use slang.** You can't assume current or future developers to be familiar with them. E.g. `exit()` is better than `hit_the_road()`.
- **Avoid unintended meanings.** Do your due diligence to check dictionaries (especially [Urban dictionary](#)!) if the word has unintended meaning. E.g. `export_data()` is a better function name than `dump()`.
- **Don't use uncommon English words.** Stick to common parlance that most developers understand. E.g. `start_process()` is better than `commence_process()`.
- **Don't use unpronounceable names.** While this is the weakest requirement, pronounceable names enable easier verbal communication. E.g. `generate_timestamp()` is better than `genymdhms()`.

# Technical & Maintainability Issues

- **Don't misspell to save a few characters.** Remembering spelling is difficult, and remembering *correct misspelling* even more so. E.g. don't use `hilite` instead of `highlight`. The benefit is not worth the cost here.
- **Don't use commonly misspelled words in English.** Using such names for variables can, at minimum, slow you down, or, at worst, increase the possibility of making an error. E.g. is it `accumulate`, `accummulate`, `acumulate`, or `acummulate`?!
- **Don't use numeric suffixes in names to specify levels.** E.g. variable names `level1`, `level2`, `level3` are not as informative as `beginner`, `intermediate`, `advanced`.
- **Don't use unsearchable names.** Single letters and very generic terms are hard to find and replace in a codebase. E.g. parameters `a` and `f` should be `arr` and `fun`.
- **Don't prioritize grammar over clarity.** Breaking grammatical rules can improve code readability. E.g. use `fishes`, `peoples`, `feedbacks` when the plural form aids comprehension.

# The Do's

You won't have to remember any of these rules if you follow the following principle:

**Good names reveal intention and eliminate guesswork.**



# Names should be self-documenting

How good a name is can be assessed by how detailed the accompanying comment needs to be.

Poor names require more comments:

```
# function to convert temperature
# from Fahrenheit to Celsius scale
# temp is the temperature in Fahrenheit
def unit_converter(temp: float):
    pass
```

Good names are self-documenting:

```
def fahrenheit_to_celsius(temp_fahrenheit: float):
    pass
```



Tip

Good names rarely require readers to read the documentation to understand what they represent.

# Names should be specific

Generic names are widely used and acceptable for short-lived contexts. However, as scope and complexity increase, specific names become essential for clarity.

For longer loops, use meaningful names instead of `i`, `j`, `k`:

```
# abstruse
inventory[i][j]
```

```
# crystal clear
inventory[warehouse][product]
```

All variables are temporary in some sense. Calling one `tmp` is inviting carelessness.

```
# generic name
tmp = a + b
result = tmp * 2
```

```
# more descriptive
sum_values = a + b
result = sum_values * 2
```



Tip

Even when you *think* you need generic names, you are better off using more descriptive names.

## Test function names should act as a comment

Unlike regular functions, long names are less problematic for test functions because they are not visible to users or called repeatedly throughout the codebase.

```
# bad: test_retrieve_commands
# good: test_all_saved_commands_should_be_retrieved
```

# Names should be difficult to misinterpret

Try your best to misinterpret candidate names and see if you succeed.

```
1 # ambiguous - what kind of size?
2 def get_size(
3     file_path: str,
4 ) -> int:
5     pass
```

How I interpret:

*“File size in bytes on disk”*

```
1 # clear - character count!
2 def get_character_count(
3     file_path: str,
4 ) -> int:
5     pass
```

In reality:

*“Number of characters in the file content”*



Tip

Precise and unambiguous names leave little room for misconstrual.

# Names should be appropriately abstract

Find the right level of detail and domain focus—precise enough to be clear, concise enough to be readable, and focused on *what* rather than *how*.

## Use context to eliminate redundancy:

```
# redundant in context
Router.run_router()
BeerShelf.beer_count
```

```
# leverages context
Router.run()
BeerShelf.count
```

## Avoid encoding implementation details in names:

```
# implementation details encoded
binary_search_users()
sql_query_products()
bonuses_pd # pandas DataFrame
hash_map_cache
```

```
# implementation independent
find_user()
fetch_products()
bonuses
cache
```

## Find the precision sweet spot:

```
# too imprecise → okay → good → unnecessarily precise
d → days → days_since_last_accident → days_since_last_accident_floor_4_lab_23
```



Tip

Good names focus on purpose, include critical details, and remain meaningful across implementations.

# Names should maintain standards

Standards **reduce cognitive burden**: readers can reuse knowledge across contexts.

**Avoid conflicting meanings and maintain consistency:**

```
# inconsistent - size means different things
size = len(x.encode('utf-8')) # bytes
size = len(a)                 # elements

# inconsistent - different words, same concept
CreditCardAccount().retrieve_expenditure()
DebitCardAccount().fetch_expenditure()
```

```
# consistent - clear distinctions
byte_size = len(x.encode('utf-8'))
length = len(a)

# consistent - same word, same concept
CreditCardAccount().retrieve_expenditure()
DebitCardAccount().retrieve_expenditure()
```

**Follow language and domain conventions:**

```
# violates conventions
class playerEntity:
    self.HairColor = ""
```

```
# follows conventions
class PlayerEntity:
    self.hair_color = ""
```

**Use consistent prefixes for IDE tab completion:**

```
# bad - scattered when tab-completing
parse_json()
xml_reader()
csv_processor()
```

```
# good - groups related functions
parse_json()
parse_xml()
parse_csv()
```

Following a standard consistently is more important than *which* standard you adopt.

# Unnecessary details in names should be removed...

```
# okay
convert_to_string()
file_object
str_name # Hungarian notation
```

```
# better
to_string()
file
name
```

## Avoid redundancy

- In type names, avoid using *class*, *data*, *object*, and *type* (e.g. bad: `classShape`, good: `Shape`)
- In function names, avoid using *be*, *do*, *perform*, etc. (e.g. bad: `doAddition()`, good: `add()`)

# but important details should be kept!

```
# okay
child_height
password
id
address
```

```
# better
child_height_cm
plaintext_password
hex_id
ip_address
```

## Tip

If some information is critical to know, it should be part of the name.

# Boolean names should be clear

Names for Boolean variables or functions should make clear what true and false mean. This can be done using prefixes (**is**, **has**, **can**, etc.).

```
# not great
if child:
    if parent_supervision:
        watch_horror_movie = True
```

```
# better
if is_child:
    if has_parent_supervision:
        can_watch_horror_movie = True
```

In general, use positive terms for Booleans since they are easier to process.

```
# double negation - difficult
is_firewall_disabled = False
```

```
# better
is_firewall_enabled = True
```

But if the variable is only ever used in its false version (e.g. `is_volcano_inactive`), the negative version can be easier to work with.



Tip

Boolean variable names should convey what true or false values represent.

# Choose domain-appropriate names

Select terminology that matches your context: computer science terms for technical concepts, problem domain terms for business logic.

## Use computer science terms for technical concepts:

```
# vague business language  
process_items_sequentially()  
store_thing_temporarily()
```

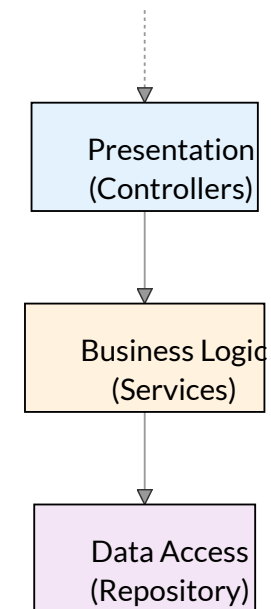
```
# precise CS terminology  
traverse_list()  
push_to_stack()
```

## Use problem domain terms for business concepts:

```
# generic technical terms  
validate_input_data()  
process_financial_records()
```

```
# domain-specific terms  
validate_loan_application()  
calculate_mortgage_payment()
```

Layered Architectur



Tip

Choose names that are meaningful to both developers and domain experts.



# Use appropriate grammatical forms

Follow consistent patterns: nouns for entities and data, verbs for actions.

**Classes and objects should use nouns:**

```
# verb-based - confusing
class ProcessPayment:
    pass

class HandleError:
    pass
```

```
# noun-based - clear
class PaymentProcessor:
    pass

class ErrorHandler:
    pass
```

**Methods that return values use nouns, action methods use verbs:**

```
# inconsistent grammar
user.get_name()      # returns name
user.save()          # performs action
user.validate_age()  # returns boolean
```

```
# consistent grammar
user.name()          # returns name
user.save()          # performs action
user.is_adult()      # returns boolean
```



Tip

**Grammatical consistency helps readers predict what methods do without reading documentation.**

# Utilizing tools

# Naming limitations of linters

Linters can only do so much when it comes to naming.

## What they CAN do:

- Enforce naming conventions
- Check for reserved keywords
- Detect naming pattern violations
- Flag overly short or long names
- Ensure consistent formatting

## What they CANNOT do:

- Understand the intent behind your code
- Suggest meaningful names based on context
- Assess whether names represent what entities do
- Determine problem domain consistency
- Evaluate clarity for future developers

### The fundamental limitation

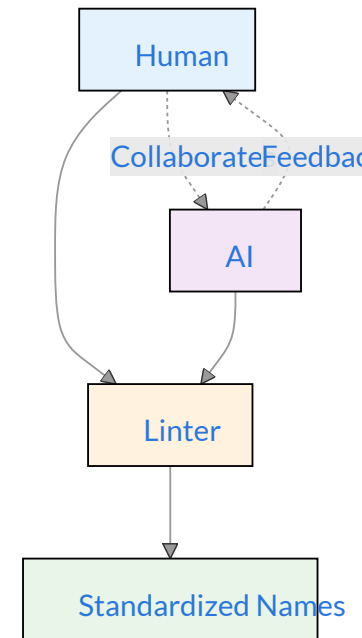
Linters can enforce *syntax* but not *semantics*. Good naming requires human understanding of both the problem and the solution.

# Generative AI tools can be valuable allies

AI tools have context of your entire codebase and can provide meaningful names.

## Why AI tools can help:

- Full context understanding of functions/classes
- Cross-domain pattern recognition
- Inconsistency detection across codebase
- Multiple naming suggestions with rationales

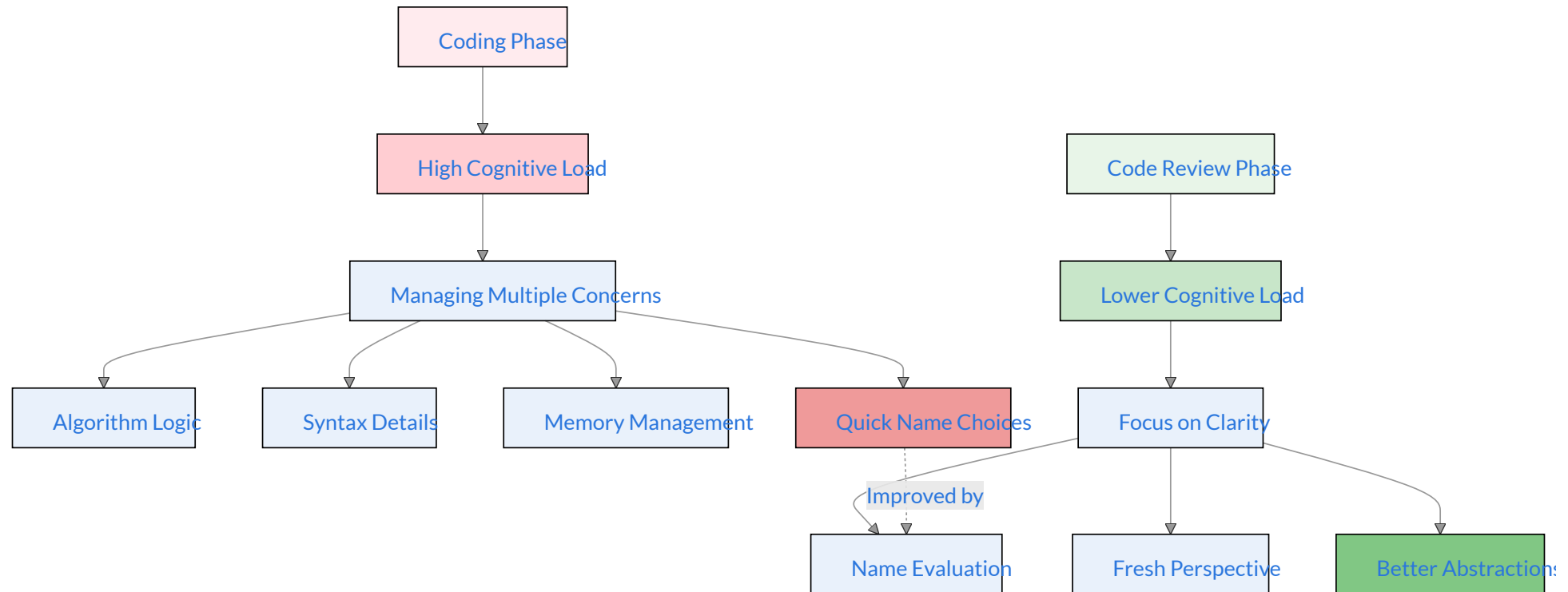


### Symbiotic Naming

Try to come up with good names yourself. Then, ask AI tools to validate, assess, or suggest improvements.

# Code Review: A fresh perspective

When coding, we operate at peak cognitive load and this mental overload makes it the worst time to choose thoughtful names.



## The Code Review Advantage

It provides the mental space needed to evaluate whether names truly capture the intent and abstraction level of code.

# Benefits of good names

*“In your name I will hope, for your name is good.” - Psalms 52:9*

# ***“What’s in a name?”*** Well, everything!

- Intent-revealing names make the **code easier to read**.
- Trying to find good names forces you to detach from the problem-solving mindset and to **focus on the bigger picture** that motivates this change. This is critical for thoughtful software design.
- Searching for precise names requires clarity, and seeking such clarity **improves your own understanding** of the code.
- Naming precisely and consistently **reduces ambiguities and misunderstandings**, reducing the possibility of bugs.
- Good names **reduce the need for documentation**.
- Consistent naming **reduces cognitive overload** for the developers and makes the code more maintainable.

# Naming is hard, but worth it

Invest time in good names early—they pay dividends by reducing system complexity.

**The more you do it, the easier it will get!**

*“Using understandable names is a foundational step to producing quality software.” - Al Sweigart*



# Thank You

And Happy Naming! 😊

# TL;DR Summary

## **Principle:** Names are a form of abstraction

“\*[T]he best names are those that focus attention on what is most important about the underlying entity, while omitting details that are less important.” - John Ousterhout

## **Importance:** Names are at the core of software design

If you can't find a name that provides the right abstraction for the underlying entity, the design may be unclear.

## **Properties:** Good names are precise and consistent

If a name is good, it is difficult to miss out on critical information about the entity or to misunderstand what it represents.

# ICYMI: Available casing conventions

There are various casing conventions used for software development.

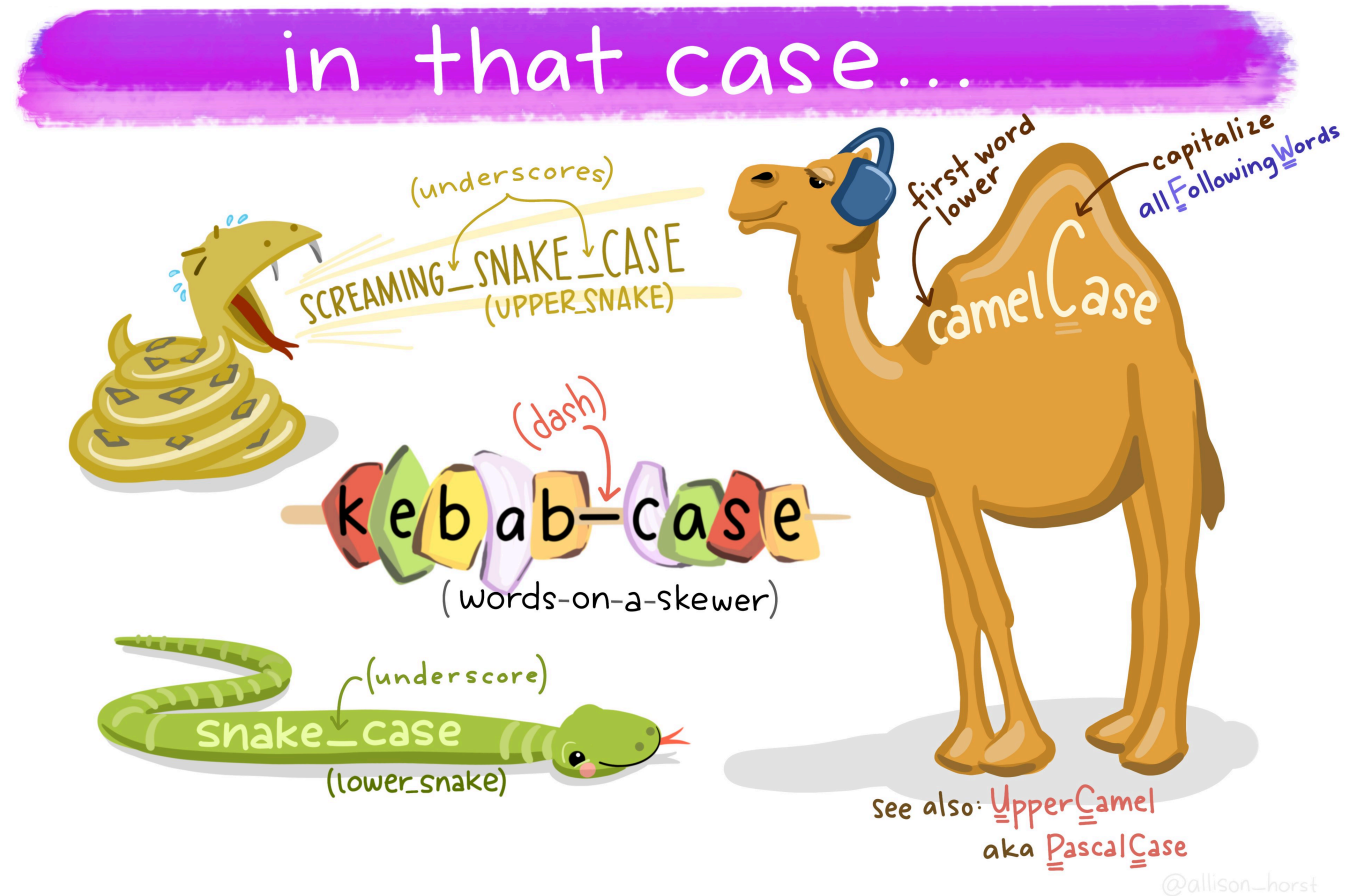


Illustration (CC-BY) by Allison Horst

# Further Reading

For a more detailed discussion about how to name things, see the following references.

# References

- McConnell, S. (2004). *Code Complete*. Microsoft Press. (pp. 259-290)
- Boswell, D., & Foucher, T. (2011). *The Art of Readable Code*. O'Reilly Media, Inc. (pp. 7-31)
- Martin, R. C. (2009). *Clean Code*. Pearson Education. (pp. 17-52)
- Hermans, F. (2021). *The Programmer's Brain*. Manning Publications. (pp. 127-146)
- Ousterhout, J. K. (2018). *A Philosophy of Software Design*. Palo Alto: Yaknyam Press. (pp. 121-129)
- Goodliffe, P. (2007). *Code Craft*. No Starch Press. (pp. 39-56)
- Padolsey, J. (2020). *Clean Code in JavaScript*. Packt Publishing. (pp. 93-111)
- Thomas, D., & Hunt, A. (2019). *The Pragmatic Programmer*. Addison-Wesley Professional. (pp. 238-242)
- [Ottinger's Rules for Variable and Class Naming](#)
- For a good example of organizational naming guidelines, see [Google C++ Style Guide](#).

# For more

If you are interested in good programming and software development practices, check out my other [slide decks](#).

# Find me at...

 LinkedIn

 GitHub

 Website

 E-mail