Naming Conventions in Go: Short but Descriptive

The written and unwritten rules

Dhia Apr 18, 2020 · 4 min read★

Photo by chuttersnap on Unsplash.

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

That’s not a funny joke. Writing is easy, but reading is painful. Do you ever find yourself wondering what a certain variable refers to or what the purpose of a certain package is? Well, that’s why we need rules and conventions.

But while conventions are meant to make life easier, they may get overvalued and misused. It is very crucial to set in place conventions and rules for naming, but following them blindly can be harmful.

In this article, I will go through some of the important variable naming conventions in Go (the written and unwritten rules) and how they can be abused — especially in the case of short variable naming. Package and file naming, as well as project structure, are not in the scope of this article, as they are worth a separate article.

The Written Rules in Go

Like any other programming language, Go has naming rules. Moreover, the naming has a semantic effect that decides on the visibility of identifiers outside a package.

MixedCaps

The convention in Go is to use `MixedCaps` or `mixedCaps` (simply camelCase) rather than underscores to write multi-word names. If an identifier needs to be visible outside the package, its first character should be uppercase. If you don't have the intention to use it in another package, you can safely stick to `mixedCaps`.

```
package awesome

type Awesomeness struct {

}

// Do is an exported method and is accessible in other packages
func (a Awesomeness) Do() string {
 return a.doMagic("Awesome")
}

// doMagic is where magic happens and only visible inside awesome
func (a Awesomeness) doMagic(input string) string {
 return input
}
```

If you try to call `doMagic` from the outside, you will get a compile-time error.

Interface names

> *"By convention, one-method interfaces are named by the method name plus an -er suffix or similar modification to construct an agent*

> *noun:* `Reader` , `Writer` , `Formatter` , `CloseNotifier` *etc.” —* *Go’s official documentation*

The rule of thumb is `MethodName + er = InterfaceName` . The tricky part here is when you have an interface with more than one method. Naming following the convention will not always be obvious. Should I split the interface into multiple interfaces with a single method? I think it’s a subjective decision that depends on the case.

Getters

As mentioned in the official docs, Go doesn’t have automatic support for setters and getters, but it doesn’t ban it. It just has some rules for it:

> *“There’s nothing wrong with providing getters and setters yourself, and it’s often appropriate to do so, but it’s neither idiomatic nor necessary to put* `Get` *into the getter's name.”*

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

But it’s worth mentioning that if a setter does not perform any special logic, it’s better to just export the field and get rid of the setter and getter methods. If you are an oop advocate, it may sound weird. But it is not.

The Unwritten Rules in Go

Some rules are not officially documented, but they are widespread within the community.

Shorter variable names

Go’s community promotes using shorter descriptive variables, but I think this specific convention is quite misused. Somehow, it’s common to forget about the descriptive part of it. A descriptive name will allow the reader to understand what it is about even before seeing it in action.

> “Programs must be written for people to read, and only incidentally for machines to execute.”

— Harold Abelson

- Single-letter identifier: This is especially used for local variables with limited scope. We can all agree that we don't need `index` or `idx` to understand it's an incrementer. Using a single-letter identifier when it's just limited to the loop scope is common and recommended.

```
for i := 0; i < len(pods); i++ {
   //
}
...
for _, p := range pods {
  //
}
```

- Shorthand name: **S**horthand names are recommended when possible as long as they are easy to understand by anyone reading the code for the first time. The wider the scope of use is, the more it needs to be descriptive.

```
pid // Bad (does it refer to podID or personID or productID?)
spec // good (refers to Specification)
addr // good (refers to Address)
```

Unique names

This is about acronyms such as API, HTTP, etc. or names like ID and DB.Conventionally, we keep these words in their original form:

- `userID` instead of `userId`

- `productAPI` instead of `productApi`

Line length

There is no fixed line length in Go, yet avoiding long lines is always recommended.

Conclusion

I tried to summarize the common naming rules in Go as well as when and to which extent we can apply them. I also explained the main idea behind shorter naming in Go, which is to find a balance between being concise and descriptive.

Conventions are there to guide you — not hinder you. So you should feel comfortable breaking them whenever it feels appropriate and still serves the general purpose.