9th June 2019

Ninja code

> *Learning without thought is labor lost; thought without learning is perilous.*
>
> Confucius

Programmer ninjas of the past used these tricks to sharpen the mind of code maintainers.

Code review gurus look for them in test tasks.

Novice developers sometimes use them even better than programmer ninjas.

Read them carefully and find out who you are – a ninja, a novice, or maybe a code reviewer?

Irony detected

Many try to follow ninja paths. Few succeed.

Brevity is the soul of wit

Make the code as short as possible. Show how smart you are.

Let subtle language features guide you.

For instance, take a look at this ternary operator `'?'`:

```
1  // taken from a well-known javascript library
2  i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Cool, right? If you write like that, a developer who comes across this line and tries to understand what is the value of `i` is going to have a merry time. Then come to you, seeking for an answer.

Tell them that shorter is always better. Initiate them into the paths of ninja.

One-letter variables

> *The Dao hides in wordlessness. Only the Dao is well begun and well completed.*
>
> Laozi (Tao Te Ching)

Another way to code faster is to use single-letter variable names everywhere. Like `a`, `b` or `c`.

A short variable disappears in the code like a real ninja in the forest. No one will be able to find it using "search" of the editor. And even if someone does, they won't be able to "decipher" what the name `a` or `b` means.

…But there's an exception. A real ninja will never use `i` as the counter in a `"for"` loop. Anywhere, but not here. Look around, there are many more exotic letters. For instance, `x` or `y`.

An exotic variable as a loop counter is especially cool if the loop body takes 1-2 pages (make it longer if you can). Then if someone looks deep inside the loop, they won't be able to quickly figure out that the variable named `x` is the loop counter.

Use abbreviations

If the team rules forbid the use of one-letter and vague names – shorten them, make abbreviations.

Like this:

- `list` → `lst`.
- `userAgent` → `ua`.
- `browser` → `brsr`.
- …etc

Only the one with truly good intuition will be able to understand such names. Try to shorten everything. Only a worthy person should be able to uphold the development of your code.

Soar high. Be abstract.

> *The great square is cornerless*
> *The great vessel is last complete,*
> *The great note is rarified sound,*
> *The great image has no form.*
>
> Laozi (Tao Te Ching)

While choosing a name try to use the most abstract word. Like `obj`, `data`, `value`, `item`, `elem` and so on.

- **The ideal name for a variable is `data`.** Use it everywhere you can. Indeed, every variable holds *data*, right?

  …But what to do if `data` is already taken? Try `value`, it's also universal. After all, a variable eventually gets a *value*.

- **Name a variable by its type: `str`, `num` …**

  Give them a try. A young initiate may wonder – are such names really useful for a ninja? Indeed, they are!

  Sure, the variable name still means something. It says what's inside the variable: a string, a number or something else. But when an outsider tries to understand the code, they'll be surprised to see that there's actually no information at all! And will ultimately fail to alter your well-thought code.

  The value type is easy to find out by debugging. But what's the meaning of the variable? Which string/number does it store?

There's just no way to figure out without a good meditation!

- **…But what if there are no more such names?** Just add a number: `data1, item2, elem5` …

Attention test

Only a truly attentive programmer should be able to understand your code. But how to check that?

One of the ways – use similar variable names, like `date` and `data`.

Mix them where you can.

A quick read of such code becomes impossible. And when there's a typo… Ummm… We're stuck for long, time to drink tea.

Smart synonyms

> *The hardest thing of all is to find a black cat in a dark room, especially if there is no cat.*
>
> Confucius

Using *similar* names for *same* things makes life more interesting and shows your creativity to the public.

For instance, consider function prefixes. If a function shows a message on the screen – start it with `display…`, like `displayMessage`. And then if another function shows on the screen something else, like a user name, start it with `show…` (like `showName`).

Insinuate that there's a subtle difference between such functions, while there is none.

Make a pact with fellow ninjas of the team: if John starts "showing" functions with `display...` in his code, then Peter could use `render..`, and Ann – `paint...`. Note how much more interesting and diverse the code became.

…And now the hat trick!

For two functions with important differences – use the same prefix!

For instance, the function `printPage(page)` will use a printer. And the function `printText(text)` will put the text on-screen. Let an unfamiliar reader think well over similarly named function `printMessage`: "Where does it put the message? To a printer or on the screen?". To make it really shine, `printMessage(message)` should output it in the new window!

Reuse names

> *Once the whole is divided, the parts need names.*
> *There are already enough names.*
> *One must know when to stop.*
>
> Laozi (Tao Te Ching)

Add a new variable only when absolutely necessary.

Instead, reuse existing names. Just write new values into them.

In a function try to use only variables passed as parameters.

That would make it really hard to identify what's exactly in the variable *now*. And also where it comes from. The purpose is to develop the intuition and memory of a person reading the code. A person with weak intuition would have to analyze the code line-by-line and track the changes through every code branch.

An advanced variant of the approach is to covertly (!) replace the value with something alike in the middle of a loop or a function.

For instance:

```
function ninjaFunction(elem) {
  // 20 lines of code working with elem

  elem = clone(elem);

  // 20 more lines, now working with the clone of the elem!
}
```

A fellow programmer who wants to work with `elem` in the second half of the function will be surprised… Only during the debugging, after examining the code they will find out that they're working with a clone!

Seen in code regularly. Deadly effective even against an experienced ninja.

Underscores for fun

Put underscores `_` and `__` before variable names. Like `_name` or `__value`. It would be great if only you knew their meaning. Or, better, add them just for fun, without particular meaning at all. Or different meanings in different places.

You kill two rabbits with one shot. First, the code becomes longer and less readable, and the second, a fellow developer may spend a long time trying to figure out what the underscores mean.

A smart ninja puts underscores at one spot of code and evades them at other places. That makes the code even more fragile and increases the probability of future errors.

Show your love

Let everyone see how magnificent your entities are! Names like `superElement`, `megaFrame` and `niceItem` will definitely enlighten a reader.

Indeed, from one hand, something is written: `super..`, `mega..`, `nice..` But from the other hand – that brings no details. A reader may decide to look for a hidden meaning and meditate for an hour or two of their paid working time.

Overlap outer variables

> *When in the light, can't see anything in the darkness.*
> *When in the darkness, can see everything in the light.*
>
> Guan Yin Zi

Use same names for variables inside and outside a function. As simple. No efforts to invent new names.

```
let user = authenticateUser();

function render() {
  let user = anotherValue();
  ...
  ...many lines...
  ...
  ... // <-- a programmer wants to work with user here and...
  ...
}
```

A programmer who jumps inside the `render` will probably fail to notice that there's a local `user` shadowing the outer one.

Then they'll try to work with `user` assuming that it's the external variable, the result of `authenticateUser()` … The trap is sprung! Hello, debugger…

Side-effects everywhere!

There are functions that look like they don't change anything. Like `isReady()`, `checkPermission()`, `findTags()` … They are assumed to carry out calculations, find and return the data, without changing anything outside of them. In other words, without "side-effects".

A really beautiful trick is to add a "useful" action to them, besides the main task.

An expression of dazed surprise on the face of your colleague when they see a function named `is..`, `check..` or `find...` changing something – will definitely broaden your boundaries of reason.

Another way to surprise is to return a non-standard result.

Show your original thinking! Let the call of `checkPermission` return not `true/false`, but a complex object with the results of the check.

Those developers who try to write `if (checkPermission(..))`, will wonder why it doesn't work. Tell them: "Read the docs!". And give this article.

Powerful functions!

> *The great Tao flows everywhere,*
> *both to the left and to the right.*
>
> Laozi (Tao Te Ching)

Don't limit the function by what's written in its name. Be broader.

For instance, a function `validateEmail(email)` could (besides checking the email for correctness) show an error message and ask to re-enter the email.

Additional actions should not be obvious from the function name. A true ninja coder will make them not obvious from the code as well.

Joining several actions into one protects your code from reuse.

Imagine, another developer wants only to check the email, and not output any message. Your function `validateEmail(email)` that does both will not suit them. So they won't break your meditation by asking anything about it.

Summary

All "pieces of advice" above are from the real code… Sometimes, written by experienced developers. Maybe even more experienced than you are ;)

- Follow some of them, and your code will become full of surprises.
- Follow many of them, and your code will become truly yours, no one would want to change it.
- Follow all, and your code will become a valuable lesson for young developers looking for enlightenment.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)