Naming cheatsheet 命名作弊表

Naming things is hard. This sheet attempts to make it easier.

命名事物是困难的，这张纸试图让它变得更容易。

Although these suggestions can be applied to any programming language, I will use JavaScript to illustrate them in practice.

虽然这些建议可以应用于任何编程语言，但我将使用 JavaScript 在实践中进行演示。

English language 英语

Use English language when naming your variables and functions.

在命名变量和函数时使用英语。

```
/* Bad */
const primerNombre = 'Gustavo'
const amigos = ['Kate', 'John']

/* Good */
const firstName = 'Gustavo'
const friends = ['Kate', 'John']
```

> Like it or not, English is the dominant language in programming: the syntax of all programming languages is written in English, as well as countless documentations and educational materials. By writing your code in English you dramatically increase its cohesiveness.
>
> 不管你喜欢与否，英语是编程中的主导语言: 所有编程语言的语法都是用英语书写的，还有无数的文档和教育材料。通过用英语编写代码，你可以极大地增加代码的凝聚力。

Naming convention 变数命名原则

Pick **one** naming convention and follow it. It may be `camelCase`, or `snake_case`, or anyhow else, it does not matter. What matters is for it to remain consistent.

选择一个变数命名原则，然后跟随它。它可能是 camelCase，或者 snake case，或者无论如何，都无关紧要。重要的是保持一致性。

```
/* Bad */
const page_count = 5
const shouldUpdate = true

/* Good */
const pageCount = 5
const shouldUpdate = true

/* Good as well */
```

```
const page_count = 5
const should_update = true
```

S-I-D

A name must be *short*, *intuitive* and *descriptive*:

一个名字必须简短、直观、描述性强:

- **Short 简短**. A name must not take long to type and, therefore, remember; . 一个名字不能打太久，因此要记住;
- **Intuitive**. A name must read naturally, as close to the common speech as possible;
- **Descriptive**. A name must reflect what it does/possesses in the most efficient way.

```
/* Bad */
const a = 5 // "a" could mean anything
const isPaginatable = a > 10 // "Paginatable" sounds extremely unnatural
const shouldPaginatize = a > 10 // Made up verbs are so much fun!

/* Good */
const postCount = 5
const hasPagination = postCount > 10
const shouldDisplayPagination = postCount > 10 // alternatively
```

Avoid contractions

Do **not** use contractions. They contribute to nothing but decreased readability of the code. Finding a short, descriptive name may be hard, but contraction is not an excuse for not doing so.

```
/* Bad */
const onItmClk = () => {}

/* Good */
const onItemClick = () => {}
```

Avoid context duplication

A name should not duplicate the context in which it is defined. Always remove the context from a name if that doesn't decrease its readability.

```
class MenuItem {
  /* Method name duplicates the context (which is "MenuItem") */
  handleMenuItemClick = (event) => { ... }
```

```
  /* Reads nicely as `MenuItem.handleClick()` */
  handleClick = (event) => { ... }
}
```

Reflect the expected result

A name should reflect the expected result.

```
/* Bad */
const isEnabled = itemCount > 3
return <Button disabled={!isEnabled} />

/* Good */
const isDisabled = itemCount <= 3
return <Button disabled={isDisabled} />
```

Naming functions

A/HC/LC Pattern

There is a useful pattern to follow when naming functions:

```
prefix? + action (A) + high context (HC) + low context? (LC)
```

Take a look at how this pattern may be applied in the table below.

Name	Prefix	Action (A)	High context (HC)	Low context (LC)
`getPost`		`get`	`Post`	
`getPostData`		`get`	`Post`	`Data`
`handleClickOutside`		`handle`	`Click`	`Outside`
`shouldDisplayMessage`	`should`	`Display`	`Message`	

> **Note:** The order of context affects the meaning of a variable. For example, `shouldUpdateComponent` means *you* are about to update a component, while `shouldComponentUpdate` tells you that *component* will update on itself, and you are but controlling when it should be updated. In other words, **high context emphasizes the meaning of a variable**.

Actions

The verb part of your function name. The most important part responsible for describing what the function *does*.

`get`

Accesses data immediately (i.e. shorthand getter of internal data).

```
function getFruitCount() {
  return this.fruits.length
}
```

> See also compose.

`set`

Sets a variable in a declarative way, with value `A` to value `B`.

```
let fruits = 0

function setFruits(nextFruits) {
  fruits = nextFruits
}

setFruits(5)
console.log(fruits) // 5
```

`reset`

Sets a variable back to its initial value or state.

```
const initialFruits = 5
let fruits = initialFruits
setFruits(10)
console.log(fruits) // 10

function resetFruits() {
```

```
  fruits = initialFruits
}

resetFruits()
console.log(fruits) // 5
```

`fetch`

Request for some data, which takes some indeterminate time (i.e. async request).

```
function fetchPosts(postCount) {
  return fetch('https://api.dev/posts', {...})
}
```

`remove`

Removes something *from* somewhere.

For example, if you have a collection of selected filters on a search page, removing one of them from the collection is `removeFilter`, **not** `deleteFilter` (and this is how you would naturally say it in English as well):

```
function removeFilter(filterName, filters) {
  return filters.filter((name) => name !== filterName)
}

const selectedFilters = ['price', 'availability', 'size']
removeFilter('price', selectedFilters)
```

> See also delete.

`delete`

Completely erases something from the realms of existence.

Imagine you are a content editor, and there is that notorious post you wish to get rid of. Once you clicked a shiny "Delete post" button, the CMS performed a `deletePost` action, **not** `removePost`.

```
function deletePost(id) {
  return database.find({ id }).delete()
}
```

> See also remove.

`compose`

Creates new data from the existing one. Mostly applicable to strings, objects, or functions.

```
function composePageUrl(pageName, pageId) {
  return `${pageName.toLowerCase()}-${pageId}`
}
```

> See also get.

`handle`

Handles an action. Often used when naming a callback method.

```
function handleLinkClick() {
  console.log('Clicked a link!')
}

link.addEventListener('click', handleLinkClick)
```

Context

A domain that a function operates on.

A function is often an action on *something*. It is important to state what is its operable domain, or at least an expected data type.

```
/* A pure function operating with primitives */
function filter(predicate, list) {
  return list.filter(predicate)
}

/* Function operating exactly on posts */
function getRecentPosts(posts) {
  return filter(posts, (post) => post.date === Date.now())
}
```

> Some language-specific assumptions may allow omitting the context. For example, in JavaScript, it's common that `filter` operates on Array. Adding explicit `filterArray` would be unnecessary.

--

Prefixes

Prefix enhances the meaning of a variable. It is rarely used in function names.

`is`

Describes a characteristic or state of the current context (usually `boolean`).

```
const color = 'blue'
const isBlue = color === 'blue' // characteristic
const isPresent = true // state

if (isBlue && isPresent) {
  console.log('Blue is present!')
}
```

`has`

Describes whether the current context possesses a certain value or state (usually `boolean`).

```
/* Bad */
const isProductsExist = productsCount > 0
const areProductsPresent = productsCount > 0

/* Good */
const hasProducts = productsCount > 0
```

`should`

Reflects a positive conditional statement (usually `boolean`) coupled with a certain action.

```
function shouldUpdateUrl(url, expectedUrl) {
  return url !== expectedUrl
}
```

`min / max`

Represents a minimum or maximum value. Used when describing boundaries or limits.

```
/**
 * Renders a random amount of posts within
 * the given min/max boundaries.
```

```
 */
function renderPosts(posts, minPosts, maxPosts) {
  return posts.slice(0, randomBetween(minPosts, maxPosts))
}
```

`prev / next`

Indicate the previous or the next state of a variable in the current context. Used when describing state transitions.

```
function fetchPosts() {
  const prevPosts = this.state.posts

  const fetchedPosts = fetch('...')
  const nextPosts = concat(prevPosts, fetchedPosts)

  this.setState({ posts: nextPosts })
}
```

Singular and Plurals

Like a prefix, variable names can be made singular or plural depending on whether they hold a single value or multiple values.

```
/* Bad */
const friends = 'Bob'
const friend = ['Bob', 'Tony', 'Tanya']

/* Good */
const friend = 'Bob'
const friends = ['Bob', 'Tony', 'Tanya']
```