Simple Rules for Signalling Intent

# Show Your Intent During Variable Declaration (in order of priority)

const x = 1; //signals x will forever be ‘1’. whenever the reader sees x, they’re confident it’s ‘1’.

let x = 1; //signals x is initially ‘1’, but you intend to mutate it later.

var x = 1; //weakest signal of intent. May change, may not.

# ‘Else’ considered harmful

Avoid using ‘if-else’ statements as huge logic branches.

* When readers encounter an ‘if-else’ statement, they’re asked to split their train of thought to travel down 2 opposing paths.
* You need to make this journey of ‘what-ifs’ as short as possible. No one wants to travel down a long path just to meet a deadend.
* If statements should be short ‘detours’ that quickly get back to the main path.

Prefer

If (err) {

return 0; //signals an initial validation check

}

return 1; //having this be the ‘one and only’ main processing branch clarifies the intent of the function.

Over

If (err) {

return 0; //since ‘if-true’ and ‘else’ is on the *same-level* indention, it signals *same-level* significance.

} else {

return 1; //same-level significance dilutes the intent of the function.

}

# Prefer “Pure” functions whenever possible

Pure means ‘does not produce side effects’

* Always produces same output given same input
* It will not mutate any out of the functions immediate scope.

Prefer

function isWithinCutoff (cutoffTime, dateObj) {

return dateObj.getHours() < cutoffTime; //Dependent on received arguments

}

isPastCutoff (new Date(), 11); //you can test at any time of the day, depending on passed arguments.

Over

function isWithinCutoff (cutoffTime) {

return new Date().getHours() < cutoffTime; //Dependent on real-time CPU time.

}

isPastCutoff(11); //You’ll only be able to get TRUE until 11am (real time).

# Know the built-in Array Functions

You’ll be surprise how many opportunities there are for list-manipulation

filteredList = myList.filter(f) //returns a SUBSET of myList. Returns a NEW array, leaves the old one intact.

transformedList = myList.map(f) //transforms ALL ELEMENTS of myList. Returns a NEW array, leaves the old one intact.

sum = myList.reduce(f) //aggregates (ie sum) all elements of myList. Returns a NEW array, leaves the old one intact.

nothing = myList.forEach(f) //use each element of myList as input for something. No return value. Good for logging only.

# Please don’t ruin the purity of built-in Array Functions

Since the built-in Array Functions are already pure, please don’t make it cause side-effects

Use

incrementedList = myList.map(x => x + 1) //map() returns the transformed list.

Avoid

let incrementedList = [];

myList.map(x => incrementedList.push(x + 1)) //map() causes side-effects (mutations) on incrementedList, which is out of scope.

# Prefer built-in Array Functions over For-Loops

Remember: visual noise is mental noise.

Prefer

oddNumbers = myList.filter(x => x % 2)

Over

let oddNumbers = [];

for (let i = 0; i < myList.length ; i++) { //much more elements to keep track of than what’s required.

if (myList[i] % 2 !== 0) {

oddNumbers.push(myList[i]);

}

}

# Use the correct built-in Array Function to signal Intent

Use

myList.forEach(x => console.log(‘the value is’, x)) //forEach() does not return anything. After list processing is done, we’re done.

Avoid

myList.map(x => console.log(‘the value is’, x)) //map() returns a new array, but is unused.

# Don’t use unneeded Variables

Variables signal future reuse of data, and asks the reader to add it to the list of things they’re juggling inside of their head.

Prefer

function summarizeInterestingData(f, g, h, list) {

return list.filter(f).map(g).reduce(h);

}

Over

function summarizeInterestingData(f, g, h, list) {

const filteredData = list.filter(f); //if this was a longer function, the reader may feel inclined to keep scrolling.

const transformedData = filteredData.map(g);

const sum = transformedData.reduce(h);

return sum;

}

*Honestly, the ideal form should not even be inside its own function, just use filter/map/reduce directly.*

# Don’t Be A Negative Person

Whenever possible, reduce negation. Say what you mean.

If unavoidable, prefer negating individuals over negating groups.

|  |  |
| --- | --- |
| *Avoid* | *Use Instead* |
| !false | true |
| !(x > 1) | x <= 1 |
| !(a && b) | a || b |
| !(a || b) | !a && !b |

# Vastly prefer Named Functions over Anonymous Functions whenever it appears more than once

Any code that appearing twice is a chance for reuseability.

Prefer

listA.filter(isOdd);

listB.filter(isOdd);

listC.filter(isOdd);

listC.filter(x => isOdd(x) && x > 10);

function isOdd(n) { return n % 2 }

Over

listA.filter(x => x % 2);

listB.filter(x => x % 2);

listC.filter(x => x % 2);

listD.filter(x => x % 2 && x > 10);

# Know ‘truthiness’, among datatypes to make your code more concise

Integers

true && 1; //good as true. Non-zeroes are considered TRUE.

true && -1; //good as true. Non-zeroes are considered TRUE.

true && 0; //good as false. For integers, only 0 is considered FALSE.

Strings

true && “something” //good as true. Non-empty string are considered TRUE

true && “ ” //good as true. Non-empty string are considered TRUE. Even string with just a space.

true && “” //good as false. Empty strings are considered FALSE

Objects, Nulls and Undefined

true && {} //good as true. Objects (even empty ones) are considered TRUE.

true && null //good as false.

true && undefined //good as false.

*Notice that I say “good as true”, instead of “returns true”. JavaScript will not turn your 0s and Nulls into false.*

# Use “Default Assignment” idiom when setting default values.

Building on the ‘truthiness’ above, you can take advantage of “short-circuits” (aka lazy evaluation).

JavaScript logical operators (&&, ||) are lazy. It evaluates left to right, and if it can already give a definite true/false answer, the rest won’t be evaluated.

const message = receivedMsg || “default msg”; //if receivedMsg is empty, it will fallback on the default msg.

# Optimize for Readability

* Visual Noise is Mental Noise.
* Clear is better than Clever.
* Conciseness leads to Clarity.
* One can be *too* Concise.
* Programmer Time is more expensive than CPU Time. (Where *Programmer Time* is time spent reading, debugging, and reviewing code).