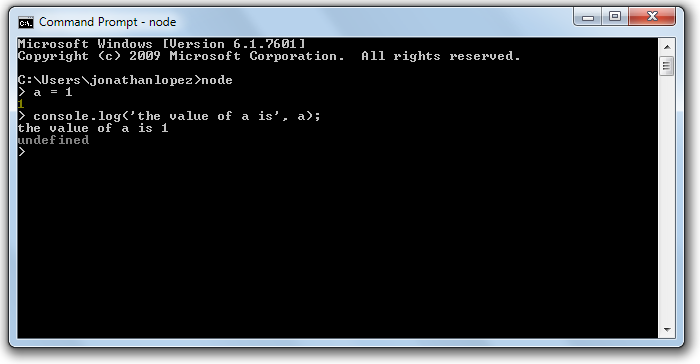
Simple Rules for Readability

# The correct attitude

“Play” is the operative word here. You’ll learn best by interactively exploring the limits of JavaScript.

You have NodeJS installed, right? In command prompt, simply type in Node <Enter>.

You’ll enter the Node REPL (Read-Evaluate-Print-Loop), which is essentially an interactive line-by-line JavaScript interpreter.



*Press ctrl+C twice to exit the REPL*

# Show Intent during variable declaration (in order of priority)

const x = 1; //signals ‘x is always 1’. Whenever the reader sees x, they’re confident it’s ‘1’.

let x = 1; //signals ‘x starts at 1 and will change’.

var x = 1; //weakest signal. 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 gives same output from same input
* It will not change any variables out of the function’s immediate scope.

Prefer

function isWithinCutoff (cutoffTime, dateObj) {

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

}

isPastCutoff (new Date(), 11); //you can test any time of the day, based 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 surprised how many opportunities there are for list-manipulation

filteredList = myList.filter(f) //returns a SUBSET. Returns a NEW array. Old one left intact.

transformedList = myList.map(f) //transforms ALL ELEMENTS. Returns a NEW array. Old one left intact.

sum = myList.reduce(f) //aggregates (ie sum) elements. Returns a NEW array. Old one left intact.

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

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

Since they’re 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() here causes side-effects (mutations) on var 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() doesn’t return anything. We’re done.

Avoid

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

Thank goodness we don't have only serious problems, but ridiculous ones as well. (1982, Edsger W. Dijkstra)

# Don’t use unneeded Variables

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

Prefer

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

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

}

Over

function summarize(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 Negative

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 |

# Named Functions are opportunities for reuseability

Any code appearing twice is a candidate for reuse.

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 zeroes and Nulls into false.*

# Prefer Strict Equality (===) comparisons

Building on the ‘truthiness’ above, prefer the ‘triple equals’ comparisons to avoid unintended type casting

Prefer

a === b //compares actual values

Over

a == b //compares ‘truthiness’ values

Reason

//suppose...

iAmTrue = true

iAm1 = 1

//comparisons...

iAmTrue === iAm1 //true === 1 returns FALSE!

iAmTrue == iAm1 //true == 1 returns TRUE!

# Prefer Ternary Operator ‘?’ for default assignment

Prefer

const userType = isAdmin ? ‘super’ : ‘normal’

Over

let userType = ‘’; //’let’ is a bad sign

if (isAdmin) {

userType = ‘super’

} else {

userType = ‘normal’

}

# Optimize for Readability

* Visual Noise is Mental Noise.
* Clear is better than Clever.
* Conciseness leads to Clarity.
* …though one can be *too* Concise.
* Programmer Time is more expensive than CPU Time.
  + Time spent…
    - understanding programs
    - reviewing programs
    - debugging programs
    - refactoring programs
    - writing post-mortems on programs that blew up in production
  + …is time NOT spent
    - Creating new products
    - Improving existing products
    - Improving skills to be better at creating products