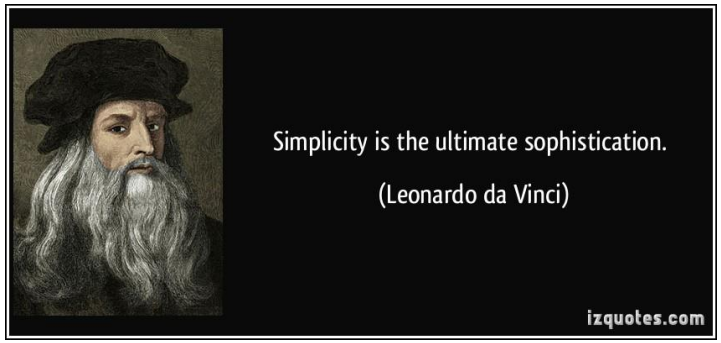
Simplicity, Readability, and JavaScript

Sep 2017, Jonathan Lopez

Simplicity Counts

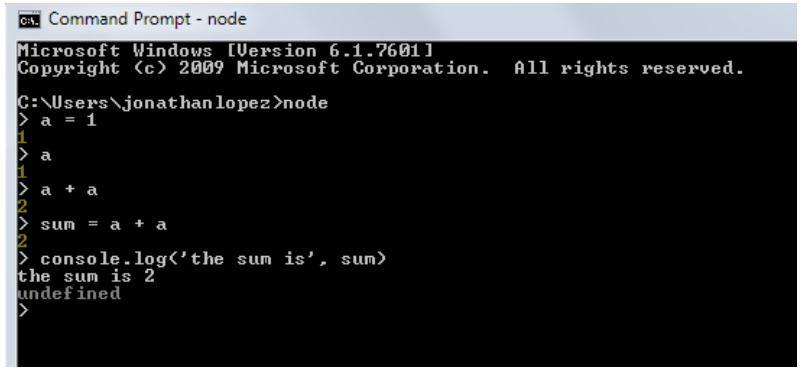
Comprehending a program is an exercise in being a human compiler – following code, looping thru flow-control statements, and juggling variables and their values in short-term memory. The more we can eliminate 1) State “aka Variables”, 2) Flow Controls [if/else/for], and 3) Lines of Code, the easier comprehension becomes. The following are a few simple tips that worked well for me when structuring JS code.



Play This

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), an interactive 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. No one wants to travel down a long path just to meet a dead-end.

Prefer

```
If (b) {
    return 0;       //signals an initial check
}

someProcess();
return 1;           //having this be the 'one and only' main processing branch clarifies the function's intent.
```

Over

```
If (b) {
    return 0;       //since 'if-true' and 'else' is on a same-level indention, it signals same-level significance.
} else {
    someProcess();
    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
- Won’t change variables out of the function’s immediate scope.
- “Easier to reason about” – because the variables are not spread over an acre of code.

Prefer

```
function isWithinCutoff (cutoff, dateObj) {
    return dateObj.getHours() < cutoff;      //fully dependent on received arguments
}

isPastCutoff (new Date(), 11);               //You can test any time of the day, based on passed arguments.
```

Over

```
function isWithinCutoff (cutoff) {
    return new Date().getHours() < cutoff;   //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, and JavaScript has got you covered.

```
filteredList    = myList.filter(f)   //returns a SUBSET.              Returns a NEW array. Old one left intact.
transformedList = myList.map(f)      //transforms EACH ELEMENT.       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.
```

Vastly 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

- .filter() = get a subset
- .map() = transform each element
- .reduce() = summations, aggregations
- .forEach() = side-effects only (ie logging)

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.
```

Don’t use unneeded Variables

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

Prefer

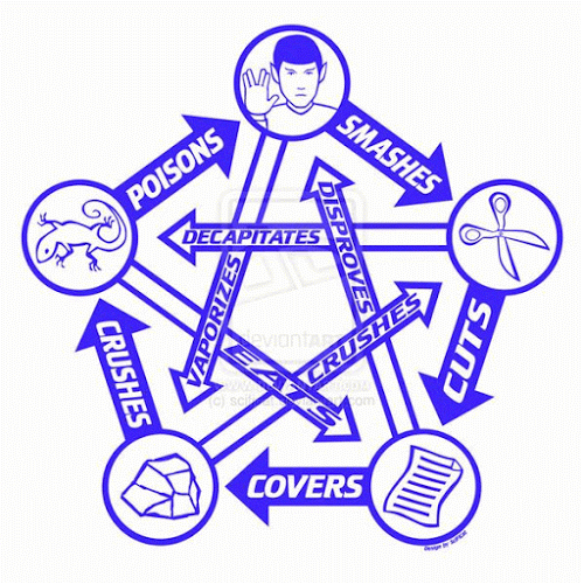
```
function summarize(f, g, h, list) {
    return list.filter(f).map(g).reduce(h);             //filter, then transform, then summarize.
}                                                       // no intermediate variables required.
```

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; Say what you mean



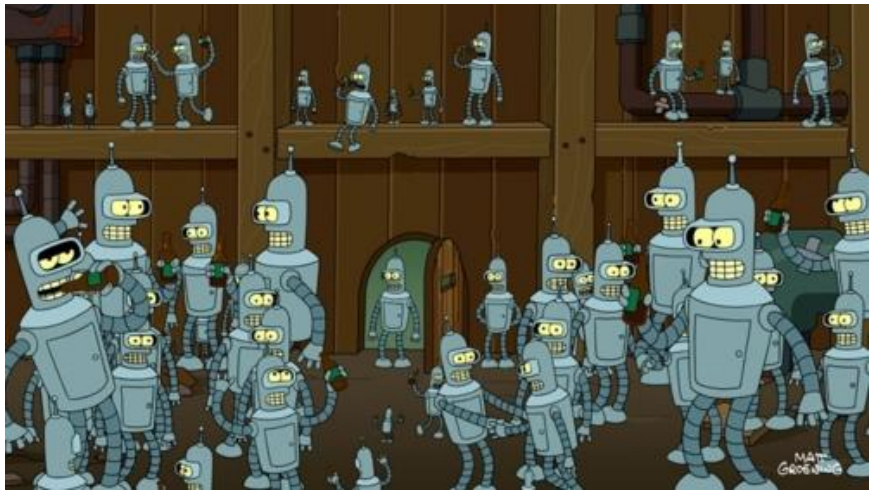
Whenever possible, reduce negation.

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 reusability

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) {                    //declared once, used 4x.
    return n % 2
}
```

Over

```
listA.filter(x => x % 2);              //its odd that the 'isOdd' formula appears 4x
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

A lot of the primitive datatypes (number/bool/string) can be compared to each other

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* (i.e. comparing apples to oranges)

Prefer

```
a === b           //compares actual values
```

Over

```
a == b            //compares 'truthiness' values
```

Proof

```
//suppose...
iAmTrue = true
iAm1    = 1

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

Prefer Ternary Operator '?' for default assignment

Prefer

```
const userType = isAdmin ? 'super' : 'normal'    //hooray for CONSTants and one-liners!
```

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.
    - Because programmers tell the CPU what to do.
    - The law of leverage – multiplies gains or losses

Thanks for reading. More tips to come in Part II.
