Silly Tricks From A Lazy Javascripter

# Why do I have to keep typing console.log all the time?

Lively version

console.log(‘hello world 1’) //ok

console.log(‘hello world 2’) //meh

console.log(‘hello world 3’) //yawn

Lazy version

log = console.log //we can borrow the function directly from console object

log(‘hi 1’) //just as good

log(‘hi 2’) //just like fine wine...

log(‘hi3 ’) //... it gets better with age.

# Lazily toggle a Boolean

Why shouldn’t flipping a Boolean be as simple as flipping a light switch?

Lively

if (b === true) { //there has to be another way

b = false;

} else {

b = true;

}

b = b === true ? false : true //good

b = b ? false : true //better

Lazy

b = !b //lazy is best

# Never waste effort equating to TRUE or FALSE

Lively

If (b === true) {

doStuff()

}

Lazy

if (b) {

doStuff()

}

# Composing Objects

Don’t give a variable another name if you don’t have to

Lively

a = 1;

b = 2;

c = 3;

obj = {a: a, b: b, c: c} //returns {a:1, b:2, c:3}

Lazy

a = 1;

b = 2;

obj = {a, b, c} //returns {a:1, b:2, c:3}, which is just as good. Even better as we saved on typing.

# Object function definition shorthand

Lively

{

v: 1,

doubleMe: function() { //verbose

return v \* 2;

}

}

Lazy

{

v: 1,

doubleMe() { //cleaner

return v \* 2;

}

}

# Quickie Cloning of Objects

Lively

original = { a: 1, b: 2 };

clone = {}

for (property in original) {

clone[property] = original[property]; //yes, for many months this was how I did it.

}

Lazy

original = { a: 1, b: 2 };

clone = JSON.parse(JSON.stringify(original)); //turn into string, then convert to JSON again.

Ok, that was overboard. This is lazy and correct

original = { a: 1, b: 2 };

clone = Object.assign({}, original) //elements original get copied to the {} and assigned to clone

There’s actually an even better method using Object Spread Operator (...), but too bad Chrome and the Node JS 6 we have doesn’t support it yet.

# Quickie Cloning of Arrays

…

# Steal any function

You can pretty much make an Alias for any function you want to use often. Ahhhh, much less typing.

Logging:

Lively

console.log(‘hi’)

console.log(‘hello’)

Lazy

log = console.log

log(‘hi’)

log(‘hello’)

Object Cloning:

Lively

newObj1 = Object.assign ({}, oldObj)

newObj2 = Object.assign ({}, oldObj)

Lazy

clone = Object.assign

newObj1 = clone({}, oldObj)

newObj2 = clone({}, oldObj)

Works particularly great for those deeply-nested function names:

Lively

argTail = Array.prototype.slice.call(arguments, 1) //be able to use actual Array functions on Array-like objects

drop2 = Array.prototype.slice.call(arguments, 2)

drop3 = Array.prototype.slice.call(arguments, 3)

Lazy

slice = Array.prototype.slice

argTail = slice.call(arguments, 1)

drop2 = slice.call(arguments, 2)

drop3 = slice.call(arguments, 3)

# Substrings

# Join 2 Arrays

Lively

low = [1, 2, 3];

high = [4, 5, 6];

join = [];

for (x in low) {

join.push(x);

}

for (x in high) {

join.push(x);

}

log(join); //returns [1, 2, 3, 4, 5, 6]

Lazy

low = [1, 2, 3];

high = [4, 5, 6];

joined = low.concat(high) //returns [1, 2, 3, 4, 5, 6]. concat() is concatenation, returns a new Array.

Lazy and Fancy

low = [1, 2, 3];

high = [4, 5, 6];

joined = [...low, ...high] //returns [1, 2, 3, 4, 5, 6]. Those triple-dots are actual JS symbols called ‘Spread Operators’

# Sort (in-place)

JS has a horrible built-in array sort() method, because it mutates the *original* array (that is NOT a good thing).

Given

Lazy and careless

orig = [4, 5, 6, 1, 2, 3]

sorted = orig.sort()

log(sorted) //ok, its sorted [1, 2, 3, 4, 5, 6] but at what cost?

log(orig) //orig is ruined [1, 2, 3, 4, 5, 6].

Lazy and considerate

orig = [4, 5, 6, 1, 2, 3];

sorted = orig.slice().sort(); //slice w/ no arguments gives you a copy of the entire original array

log(sorted); //ok, its sorted [1, 2, 3, 4, 5, 6].

log(orig); //orig is intact [4, 5, 6, 1, 2, 3].

# Arrow Functions save a lot of lives from strokes. Keystrokes.

Just make sure you understand the uniqueness of Arrow Functions

Lively

function doubleMe(x) {

return x \* 2;

}

Lazy

doubleMe = x => x \* 2; //I like one liners

One more example

Lively

function isOdd(x) {

return x % 2;

}

Lazy

isOdd = x => x % 2; //This represents 34.48% savings in ‘strokes. Trust me I did the math.

# Existential Crisis (Arrays)

# Existential Crisis (Objects)

# Use ‘||’ 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.

# Better yet, get familiar with ‘Destructuring’ syntax for defaults

Destructuring is signalled by “{ }” on the *left side* of assignment operator (=)

const optionsObj = {a: 1, b: 2} //let’s make a sample ‘options’ object which we’ll destructure...

let {a, b} = optionsObj //a=1, b=2 //Get ‘a’ and ‘b’ inside the source (optionsObj)

{a=10, b=20, c=30} = optionsObj //a=1, b=2, c=30 //We can have defaults if the source doesn’t have it

{a: c} = optionsObj //c=1 //We can set a custom target “get a, assign to c”

A more practical use case is using Destructuring inside functions

function f(a, options) {

const {b=0, c=0} = options;

return [a,b,c];

}

F(1, {b=2, c=3}) //[1,2,3]

F(1, {b=2}) //[1,2,0]

F(1) //[1,0,0]

Destructuring also happens in the function argument

# Everyone is Unique

Suppose you only want a list of unique values from a list

Lively

items = [1,2,3,1,2,3]

for ...

Lazy

items = [1,2,3,1,2,3]

unique = items.filter(function(item, index, array) {

return array.indexOf(item) === index; //only returns the first occurrence

})

log(unique); //shows [1,2,3]

Super Lazy

items = [1,2,3,1,2,3]

unique = items.filter((item, index, array) => array.indexOf(item) === index) //Arrow Functions give us a one-liner.

log(unique); //shows [1,2,3]

FP (Functional Programming) wins in the laziness department

getUnique = xs => xs.filter((x, i, a) => a.indexOf(x) === i); //Functional Programmers are masters of laziness

items = [1,2,3,1,2,3]

magic = ['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a'];

getUnique(items); //returns [1,2,3]

getUnique(magic); //returns [‘a’,’b’,’r’,’c’,’d’]

# Conclusion

Well that was fun to write. I hope you learned something about the *expressivity* of JavaScript (ie more than one way to do things).

Truthfully, these aren’t so much ‘hacks’ as they are JavaScript *Idioms* – culturally accepted best-in-class solutions to the most common problems.

Compact. Concise. And most importantly – Correct.