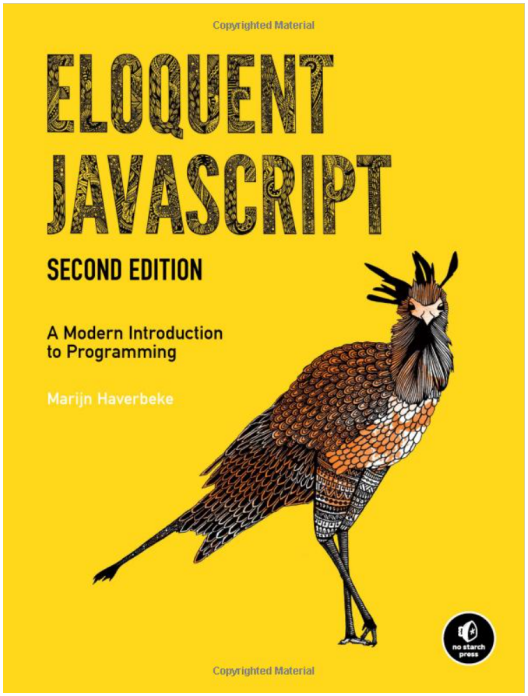
Douglas Crockford “Javascript: The Good Parts” O’Reilly

Douglas Crockford's popular book really highlighted the idea that javascript has good parts that exist within some aspects that cause lots of problems (the bad parts?)

Recommended Books:



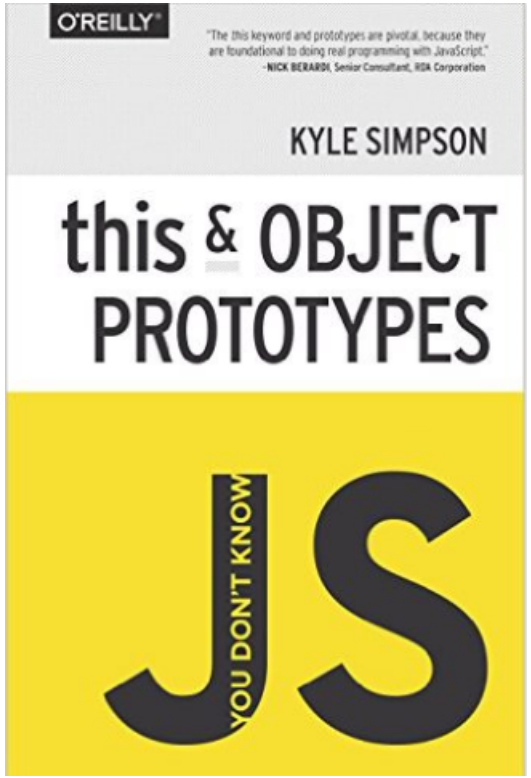
Elequent JavaScript by Marijn Haverbeke

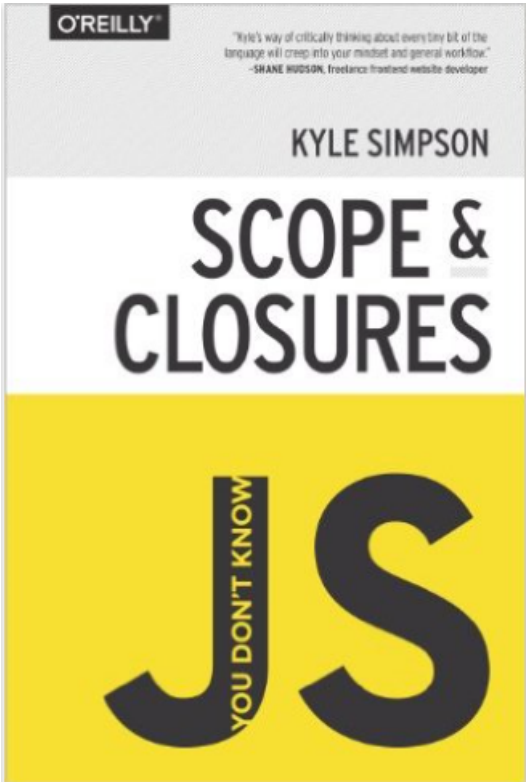
Many other books came out that tried to provide style and usage guidelines to help tame javascript

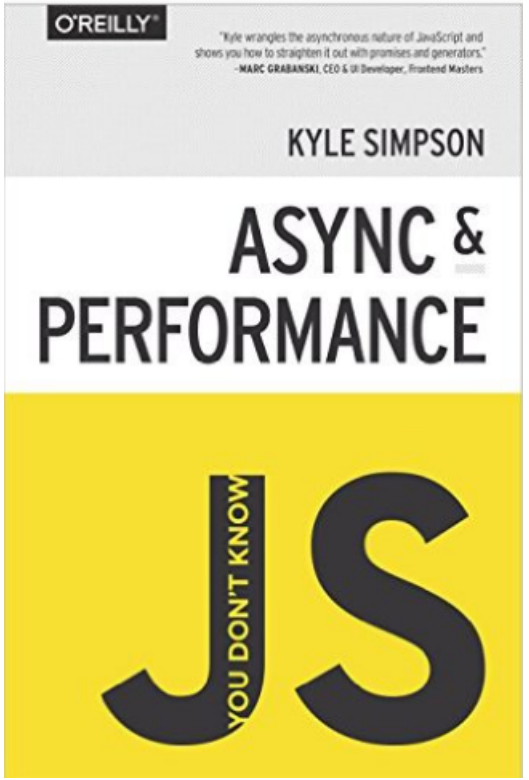
The titles of many books on javascript seem to suggest that writing javascript code is problematic -why would that be?

Recommended Books:

Kyle Simpson's "You Don't Know JS" series has been very popular and goes deeply into specific topic areas of javascript



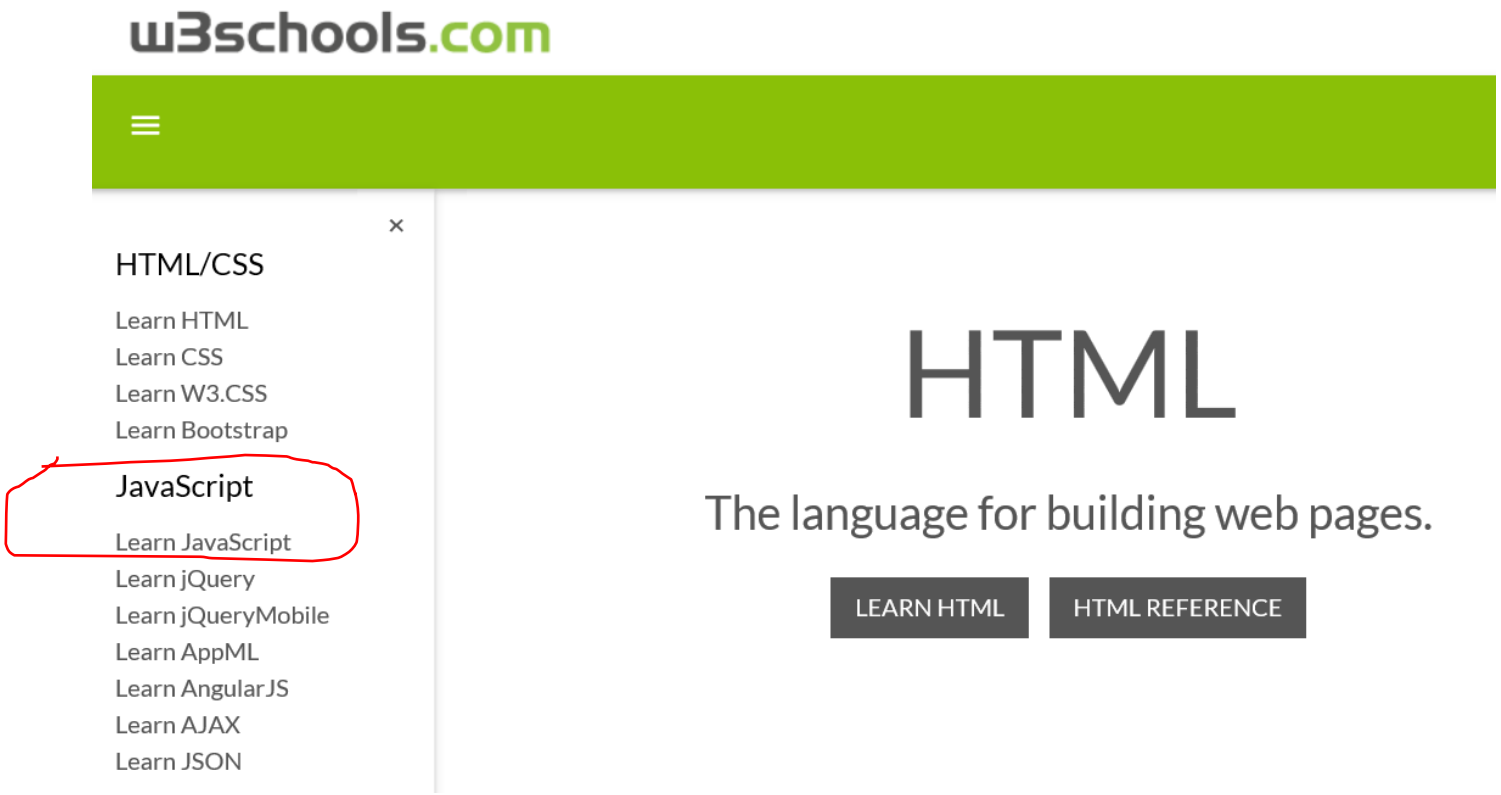




...

Recommended Starting Place:

w3schools.com/



...

Javascript Pitfalls:

Here we will summarize some of the pitfalls that drove students, and myself, crazy in previous course offerings.

These slides are a summary of things we experienced writing and debugging javascript code in COMP 2406

Danger Will Robinson

Throughout this presentation and other course notes you will see some "Danger Will Robinson" slides.

These all show something that tripped up students in past offerings of the course. (They actually happened to someone in the class).

Please contribute your own "Danger Will Robinson" slide if you come across something you want to warn us about. (Or let us know if some are no longer valid.)

Also if you have good code examples that illustrate a danger pass them along so we can post them

Javascript Primitive Types

Javascript has primitive and object data types.

What they are depends on which book you read or which website you visit.

Here are some common versions of what you find.

Primitives: boolean number string symbol (ES6) undefined null	boolean number string symbol (ES6) undefined	boolean number string symbol (ES6) undefined
Objects: object function	object function	object

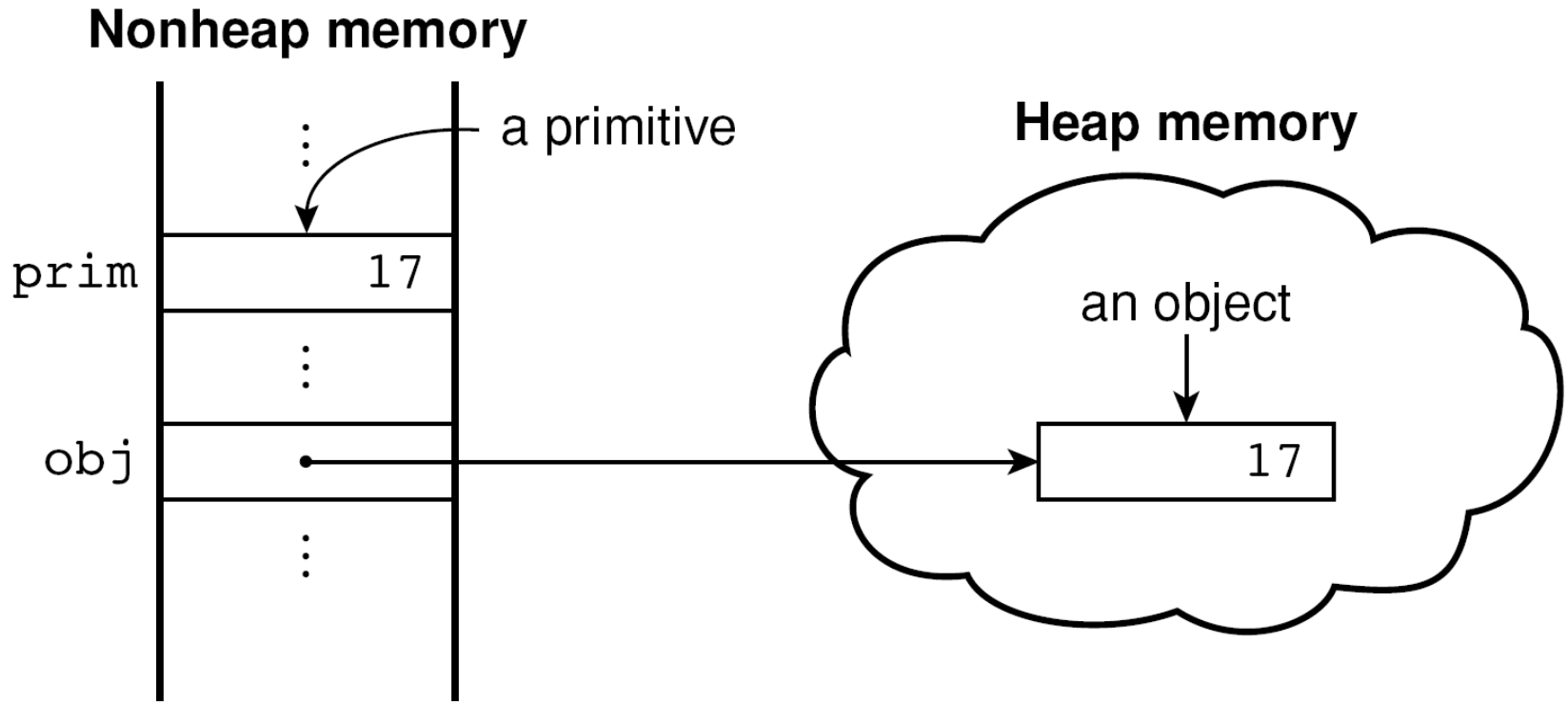


Figure 4.1 Primitives and objects

```
PRIMITIVES vs. OBJECTS in PROGRAMMING LANGUAGES:
prim1 == prim2; //can check for equality
prim = prim1;   //makes a copy

obj1 == obj2; //check for identity, not equality
obj = obj1;   //refer to same object.
```

Danger Will Robinson

- Javascript uses IEEE 754 standard binary floating point arithmetic
- Unfortunately languages based on IEEE 754 cannot handle decimal fractions exactly (binary rep. cannot represent 1/10, just like decimal fraction cannot handle 1/3 exactly)
- for example 0.1 + 0.2 does not yield 0.3

```
c:\2406Node>node
> 0.1 + 0.2
0.30000000000000004
>
```

- Most frequently reported bug in javascript code
- Suggestion: if dealing with money, convert to dollars and whole cents, do the arithmetic with whole numbers and then convert final result to fraction

Number in Javascript

Javascript is the first modern language I've encountered that does NOT have an int (integer) data type.

In javascript all numbers are real (instances are floating point numbers).

Java:	C++	javascript
3/5 == 0	3/5 == 0	3/5 == 0.6

A sacred rule in programming is: Never compare real numbers for equality:

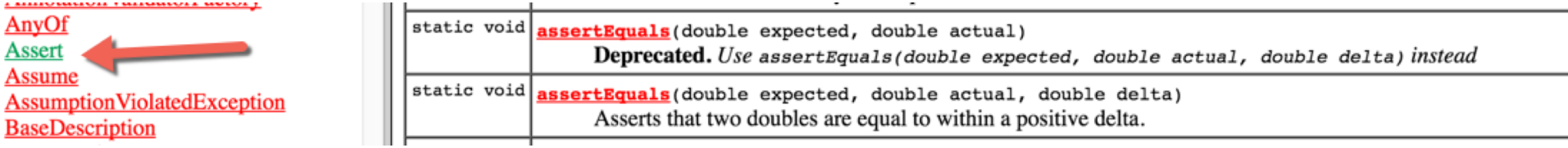
For real values x and y:

```
if(x == y) {…}   //NEVER DO THIS

if(x < y) {…}   //OK
if(x <= y) {…}   //OK
if(x > y) {…}   //OK
if(x >= y) {…}   //OK
```

…Number in Javascript

"Amusing" example from Android's JUnit testing Java library:



Deprecated sounds very polite!

Concatenation Operator

- The + operator both concatenates as strings or adds numbers
- Unless both operators are numbers they will both be turned into strings and concatenated as strings
- This behaviour is source of many bugs
- If you want + to mean add you must make sure that both operands are numbers.

```
> 42 + 5
47
> 42 + '5'
'425'
```

Danger Will Robinson

- `parseInt('16')` → 16
- `parseInt('16 acres')` → 16

parseInt() reads till it finds a non digit or end

- `parseInt('08')` → 0

(because leading 0 indicates base 8 not base 10, and 8 is not a digit in base 8. Many bugs related to parsing dates)

Suggestion: always use the version of parseInt that allows you to specify the radix

`parseInt('08', 10)` → 8

NOW:

`parseInt('08')` → 8

$ Character

The $ in javascript is just another letter that can be used for variable names or function names.

People confuse it for some special operator.

```
> var $ = 42;
> $;   //42
> var $name$$$ = 'Sue';
> $name$$$; //'Sue'
> var $ = function(n){console.log(n)};
> $(2);  //2
> function $(n){console.log(n)};
> $(2); //2
```

Jquery uses a global variable $ to represent it's library

```
$.post("positionData",jsonStr,function(data, status){});
```

Reserved Words

Reserved words cannot be used directly as variable or function names but can be used as objects property names

```
var method // ok
var class // illegal
function for(){return 42} // illegal
object = {box: value} // ok
object = {var: value} //ok
object = {'var': value} // ok
object.box = value // ok
object.var = value //ok
object['var'] = value // ok
```

Semicolon Insertion

```
function f(){
        return
        3;
}
f(); //undefined

function f(){
   return 3;
}
f();  //3
```

- Javascript tries to correct "what it thinks" might be faulty programs by inserting "what it thinks" are missing semicolons.
- Sometimes places semicolons at the end of a line if "it thinks" that will complete an expression

```
return
5+6;
```

will become

```
return;
5+6;
```

Suggest: keep expressions on one line

especially `return` statements.

or end a line with something that cannot be a valid statement if you added a semicolon.

Ironic that we battle extra ; instead of missing ;.

Semicolon Insertion

```
c:\>node
> function f1(n){return 42 + n;}
undefined
> f1(2);
44
> function f2(n){
... return
... 42 + n;
... }
undefined
> f2(2);
undefined
> function f3(n){
... return 42 +
... n;
... }
undefined
> f3(2);
44
>
```

- **Javascript tries to correct "what it thinks" might be faulty programs by inserting "what it thinks" are missing semicolons.**
- **Sometimes places semicolons at the end of a line if "it thinks" that will complete an expression**

```
return
5+6;
```

will become

```
return;
5+6;
```

Suggest: keep expressions on one line

especially `return` statements.

or end a line with something that cannot be a valid statement if you added a semicolon.

Ironic that we battle extra ; instead of missing ;.

Semicolon Insertion

```
> function badfunc(n){
... return 42 + n;
... 42 + n;
... }
undefined
> badfunc(2);
44
```

- But wait, it gets worse
- Javascript does not seem to care if you write code that is unreachable

```
return
5+6;
```

will become

```
return;
5+6;
```

but

```
function(n){
    return 5+6;
    5 + 4;
}
```

Does not generate an error even though 5 + 4; is an unreachable statement.

This legitimizes semi colon insertion that would not make sense.

Falsey Values

Javascript has many values that evaluate to `false` in a boolean test

0	Number
NaN	Number
''	String
false	Boolean
null	Object
undefined	Undefined

They cannot be used interchangeably. The following will not work to find out if an object is missing a member because value would be `undefined` not `null`

```
value = myObject[name]
if(value == null)
        alert(name + 'not found')
```

Seems fixed

http://dorey.github.io/JavaScript-Equality-Table/

`true`	■	if (true) { /* executes */ }
`false`	□	if (false) { /* does not execute */ }
`1`	■	if (1) { /* executes */ }
`0`	□	if (0) { /* does not execute */ }
`-1`	■	if (-1) { /* executes */ }
`"true"`	■	if ("true") { /* executes */ }
`"false"`	■	if ("false") { /* executes */ }
`"1"`	■	if ("1") { /* executes */ }
`"0"`	■	if ("0") { /* executes */ }
`"-1"`	■	if ("-1") { /* executes */ }
`""`	□	if ("") { /* does not execute */ }
`null`	□	if (null) { /* does not execute */ }
`undefined`	□	if (undefined) { /* does not execute */ }
`Infinity`	■	if (Infinity) { /* executes */ }
`-Infinity`	■	if (-Infinity) { /* executes */ }
`[]`	■	if ([]) { /* executes */ }
`{}`	■	if ({}) { /* executes */ }
`[[]]`	■	if ([[]]) { /* executes */ }
`[0]`	■	if ([0]) { /* executes */ }
`[1]`	■	if ([1]) { /* executes */ }
`NaN`	□	if (NaN) { /* does not execute */ }

Danger Will Robinson

- In Javascript `undefined` and `NaN` are not constants
- They are global variables and you can change their values
- Don't !!!
- (May be fixed now?)

Seems fixed

Leaving off the var (or let, or const)


```
function f(n){
     v1=n; v1++; return v1;
};
f(2);   //3
v1;     //3

function g(n){
    var v2=n; v2++; return v2;
};

g(2); //3
v2;   //ReferenceError: v2 is not defined
```

- Variables declared implicitly (without `var` keyword), even locally in functions, have GLOBAL scope (become global variables).
- This creates bugs that are hard to track down. Pollutes global scope.
- Originally done as a "feature" for novice programmers –it causes all kinds of problems in Javascript
- Suggest: Never declare variables without `var, let, or const`

Danger Will Robinson

With var:

```
function f(){
    if(…){
      var x = {}
      x.name = 'Lou'
      console.log(x.name)
    }
    else if(…){
      x.name = 'Sue'
      console.log(x.name)
    }
}
```

Change var to let:

```
function f(){
    if(…){
      let x = {}
      x.name = 'Lou'
      console.log(x.name)
    }
    else if(…){
      x.name = 'Sue'
      console.log(x.name)
    }
}
```

- Typical ES5-ES6 migration bug:
- Changing var to let
- Can you spot what will go wrong (Hint the code breaks)

Danger Will Robinson


```
CMD>node
> function f(){
... console.log('x='+x);
... if(1<2) {
..... var x=42;
..... console.log('x='+x);
..... }
... }
undefined
> f()
x=undefined
x=42
undefined
>
```

- var declaration is "hoisted" to top of function scope (not block scope).
- Initialization of variable is not hoisted.
- Can be source of many bugs

Variable Hoisting:

```
function f(){
    console.log(x);
    //stuff;
    {
      var x = 42;
    }
    //more stuff;
};
```

becomes:

```
function f(){
    var x; //undefined
    console.log(x);
    //stuff;
    {
      x = 42;
    }
    //more stuff;
};
```

Danger Will Robinson


```
CMD>node
> function f(){
... console.log('x='+x);
... if(1<2){
..... let x=42;
..... console.log('x='+x);
..... }
... }
undefined
> f()
ReferenceError: x is not defined
    at f (repl:2:18)
    at repl:1:1
    at sigintHandlersWrap (vm.js:2
    at sigintHandlersWrap (vm.js:7
    at ContextifyScript.Script.run
    at REPLServer.defaultEval (rep
    at bound (domain.js:280:14)
    at REPLServer.runBound [as eva
    at REPLServer.<anonymous> (rep
    at emitOne (events.js:101:20)
>
```

Variable Hoisting:

```
function f(){

    //stuff;
    {
      let x = 42;
    }
    //more stuff;
};
```

becomes:

```
function f(){
    var x; //undefined
    //stuff;
    {
      let x = 42;
    }
    //more stuff;
};
```

- Javascript 6 has `let` and `const` instead of in addition to `var`.
- Variables declared with `let` are "not hoisted" and have block, not function scope.
- (technically they are hoisted but refering to the uninitialized value is illegal)

Danger Will Robinson

- Javascript has two sets of equality operators `===`, `!==` and their evil twins: `==` and `!=`
- The good ones work as you would expect: If two operands are of the same type and have the same value then `===` gives true and `!==` gives false
- The "evil" ones do the right things if the operands are of the same type, but if not they will try to coerce the values into the same type. The rules by which this is done are complex and hard to remember. Here are some interesting cases


```
'' == '0'          //false
0 == ''            //true
0 == '0'           //true
false == 'false'   //false
false == '0'       //true
false == undefined //false
false == null      //false
null == undefined  //true
```

All of these produce `false` with the `===` operator

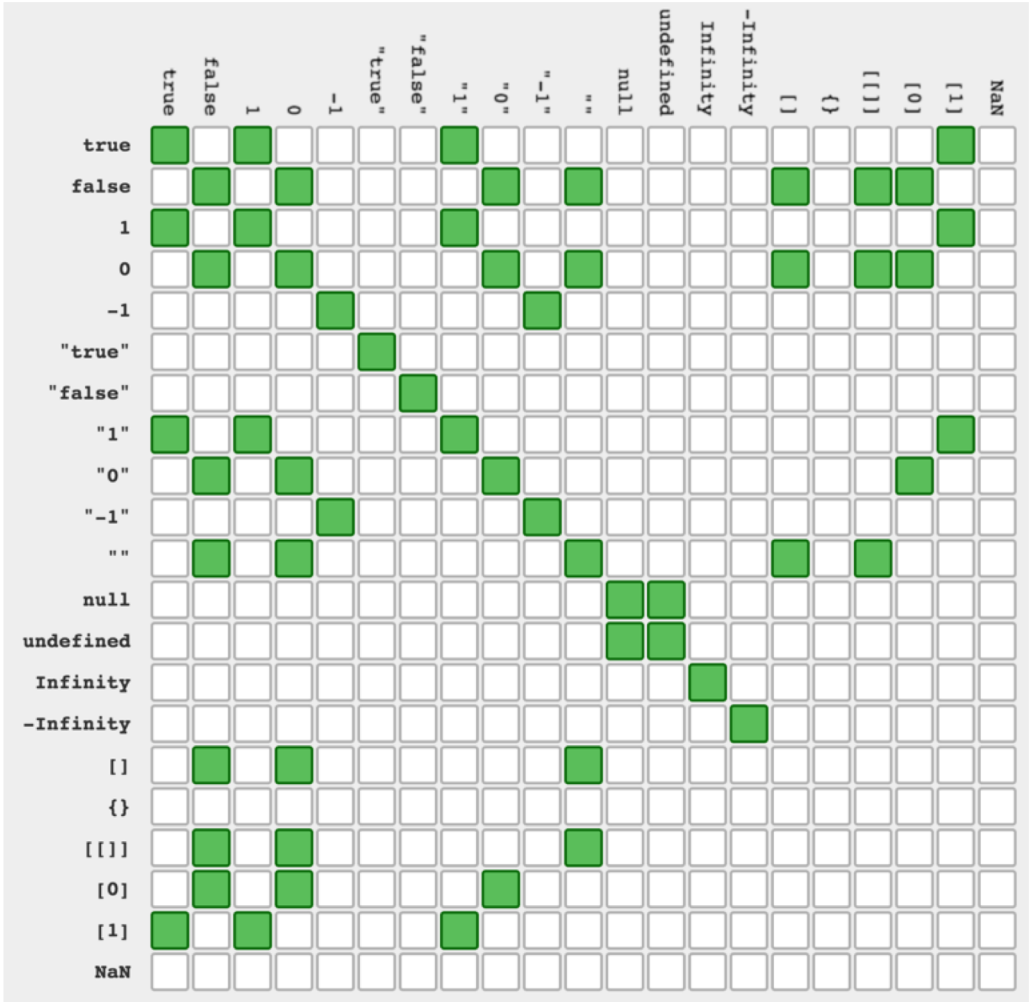
Danger Will Robinson

- Javascript has two sets of equality operators ===, !== and their evil twins: == and !=
- The good ones work as you would expect: If two operands are of the same type and have the same value then === gives true and !== gives false
- The "evil" ones do the right things if the operands are of the same type, but if not they will try to coerce the values into the same type. The rules by which this is done are complex and hard to remember. Here are some interesting cases

42 == '42' //true	42 === '42' //false
'42' == 42 //true	'42' === 42 //false
'42' == 40 + 2 //true	'42' === 40 + 2 //false
42 == '40+2' //false	42 === '40+2' //false
42 == '4'+'2' //true	42 == '4'+'2' //false

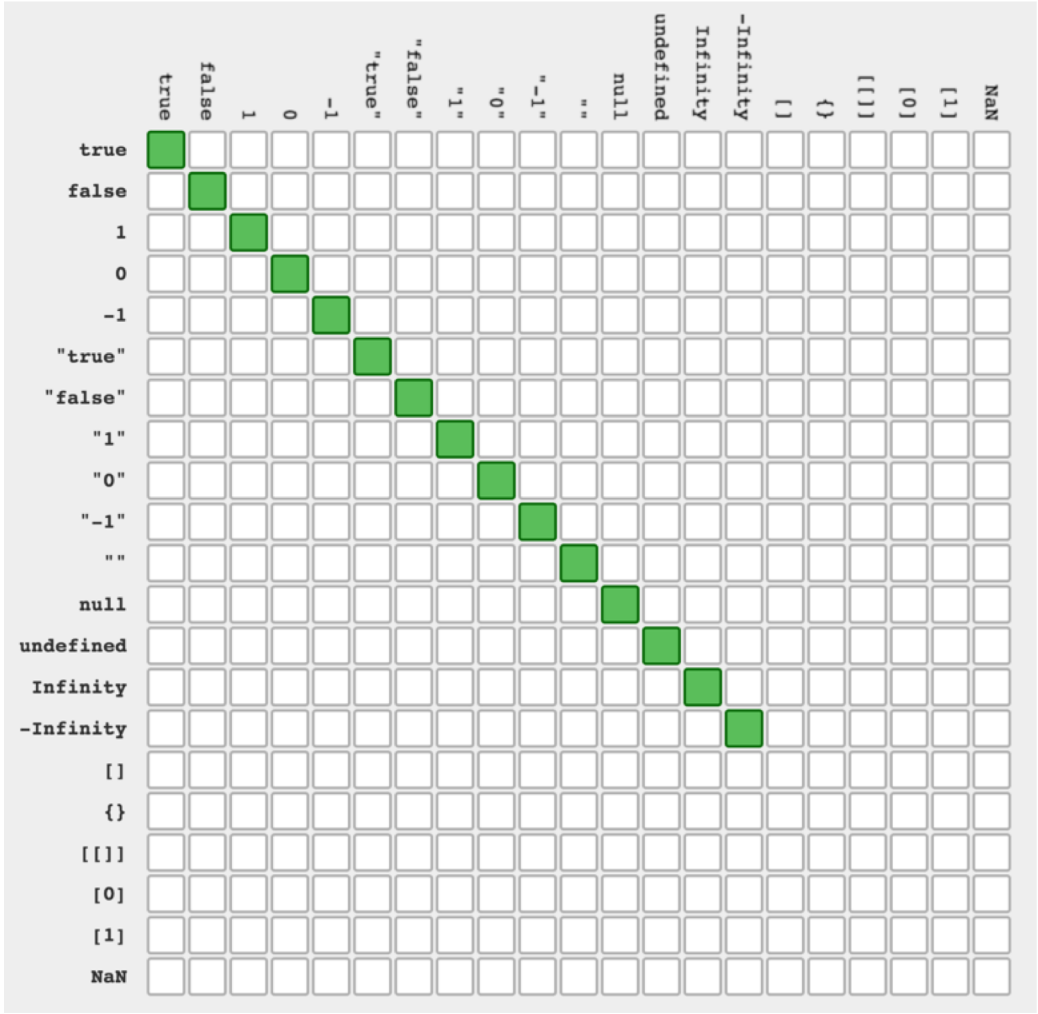
http://dorey.github.io/JavaScript-Equality-Table/

a == b



http://dorey.github.io/JavaScript-Equality-Table/

a === b



Danger Will Robinson

- `typeof NaN === 'number'  //true`

- `NaN === NaN //false`
  `NaN !== NaN //true`

- `isNan(x)` can distinguish between numbers and NaN but includes `Infinity` as a number
- `isFinite(x)` will determine if `x` is a number but not infinity. Unfortunately `isFinite()` will also coerce `x` into a number if it is not one.

Best to define your own `isNumber()` function like this

```
function isNumber(x) {
  return typeof x === 'number' && isFinite(x)
}
```

Danger Will Robinson

- `typeof null` is 'object'
- So you cannot test for null with `if(typeof x == 'null')`

- Worse: you cannot test if something is an object with `if(typeof x == 'object')` because it might be null e. `typeof null; //object`
- To test if something is object or array use the following (because my_value would be false if it were null)

```
if (my_value && typeof my_value ==='object'){
      // my_value is an object or an array!
}
```

Danger Will Robinson

- **Switch statements will fall through to the next case unless an explicit `break` disrupts the flow. This is intended as a feature but leads, in fact, to hard-to-find bugs**


```
switch (bordersize) {
  case "0": document.write("<table>");
            break;
  case "1": document.write("<table border = '1'>");
            break;
  case "4": document.write("<table border = '4'>");
            break;
  case "8": document.write("<table border = '8'>");
            break;
  default:  document.write("Error - invalid choice: ",
                           bordersize, "<br />");
}
```

Without the `break` cases would fall through to the next case and execute that code as well

Danger Will Robinson

- `typeof` operator does not distinguish between arrays and objects
- How to tell if a reference is an array or object?

One way:

```
if(x && typeof x === 'object' &&
  x.constructor === Array){
      //here if x is an array
}
```

Note this test will not work on function arguments "array" because it is not a real array.

ES6: function arguments are now real arrays –I think.

Danger Will Robinson

`new` operator and `this`

What happens below if the `new` is forgotten when trying to create a plane?


```
function plane(newMake, newModel, newYear){
    this.make = newMake;
    this.model = newModel;
    this.year = newYear;
}
```

```
myPlane = new plane("Cessna","Centurian","1970");
```

Answer: the `this` variable in the constructor binds with the global Object, from which everyone inherits, and the constructor then proceeds to clobber this object with new property values --this is very bad!

Danger Will Robinson

`new` operator and `this`

What happens below if the `new` is forgotten when trying to create object instance?


```
$node
> function car(make,model){this.make=make; this.model=model}
undefined
> let mycar = new car('Toyota','Camry')
undefined
> mycar
car { make: 'Toyota', model: 'Camry' }
> let yourcar = car('Toyota','Civic')
undefined
> yourcar
undefined
> global.make
'Toyota'
> global.model
'Civic'
>
```

Notice no new

Danger Will Robinson

```
let arr = [ 13, 42, 86, 99 ]
for(let k in arr) console.log(arr[k])
//13 42 86 99

for(let i = 0; i<arr.length; i++) console.log(arr[i])
//13 42 86 99

arr.name = 'Lou'

for(let i = 0; i<arr.length; i++) console.log(arr[k])
//13 42 86 99

for(k in arr) console.log(arr[k])
//13 42 86 99 Lou
```

- **Do not use a for-in loop to loop over the characters of a string or the elements of an array. If properties are added to the string or array the loop will loop over those as well.**

ES6: Use FOR-OF loops instead

With For-of loop (ES6)


```
let arr = [ 13, 42, 86, 99 ]

for(let k in arr) console.log(arr[k])
//13 42 86 99

for(let i = 0; i<arr.length; i++) console.log(arr[i])
//13 42 86 99

for(let k of arr) console.log(k)
//13 42 86 99

arr.name = 'Lou'

for(let k in arr) console.log(arr[k])
//13 42 86 99 Lou

for(let i = 0; i<arr.length; i++) console.log(arr[k])
//13 42 86 99

for(let k of arr) console.log(k)
//13 42 86 99
```

Danger Will Robinson

```
let str = 'Louis'
for(let i=0; i<str.length; i++) console.log(str[i])
//L o u i s

for(let k in str) console.log(str[k])
//L o u i s

str.__proto__.name = 'Bob'
for(var i=0; i<str.length; i++) console.log(str[i])
//L o u i s

for(let k in str) console.log(str[k])
//L o u i s Bob
```

- Do not use a for-in loop to loop over the characters of a string or the elements of an array. If properties are added to the string or array the loop will loop over those as well.

- Although javascript claims strings are primitives, they have a prototype which can have properties added to them (so they are actually objects!)

Wait –it gets worse

```
let str = 'Louis'
for(let i=0; i<str.length; i++) console.log(str[i])
//L o u i s

for(let k in str) console.log(str[k])
//L o u i s

str.__proto__.name = 'Bob'
for(var i=0; i<str.length; i++) console.log(str[i])
//L o u i s

for(let k in str) console.log(str[k])
//L o u i s Bob

let newstr = 'Sue'
for(let k in newstr) console.log(newstr[k])
//S u e Bob
```

- Adding properties to a strings .__proto__ prototype will break the for-in loop FOR ALL STRINGS IN YOUR PROGRAM