

<WA1/>
<AW1/>
2023

JavaScript (basics)

“The” language of the Web

Fulvio Corno

Luigi De Russis

Enrico Masala

JavaScript		Cheat Sheet										
Programming Language of Web												
n Number() PROPERTIES <ul style="list-style-type: none"> n.POSITIVE_INFINITY +0 equivalent n.NEGATIVE_INFINITY -0 equivalent n.MAX_VALUE largest positive value n.MIN_VALUE smallest positive value n.EPSILON diff between 1 & smallest >1 n.NaN not-a-number value METHODS <ul style="list-style-type: none"> s.toExponential(dec) exp. notation s.toFixed(dec) fixed-point notation s.toPrecision(p) change precision b.isFinite(n) check if number is finite b.isInteger(n) check if number is int. b.isNaN(n) check if number is NaN n.parseInt(s, radix) string to integer n.parseFloat(s, radix) string to float 	s String() PROPERTIES <ul style="list-style-type: none"> s.length string size METHODS <ul style="list-style-type: none"> s.charAt(index) char at position s.charCodeAt(index) unicode at pos. s.fromCharCode(n1, n2...) code to char s.concat(str1, str2...) combine text b.startsWith(str, size) check beginning b.endsWith(str, size) check ending b.includes(str, from) include substring? n.indexOf(str, from) find substr index n.lastIndexOf(str, from) find from end n.search(regex) search & return index n.localeCompare(str, locale, options) a.match(regex) matches against string s.repeat(n) repeat string n times s.replace(str, regex, newstr, func) s.slice(ini, end) str between ini/end s.substr(ini, len) substr of len length s.substring(ini, end) subsrc fragment a.split(sep, regex, limit) divide string s.toLowerCase() string to lowercase s.toUpperCase() string to uppercase s.trim() remove space from begin/end s.raw() template strings with \${vars} 	a Array() PROPERTIES <ul style="list-style-type: none"> n.length number of elements METHODS <ul style="list-style-type: none"> b.isArray(obj) check if obj is array b.includes(obj, from) include element? n.indexOf(str, from) find elem. index n.lastIndexOf(str, from) find from end s.join(sep) join elements w/separat a.slice(ini, end) return array portion a.concat(obi1, ob2...) return joined array MODIFY SOURCE ARRAY METHODS <ul style="list-style-type: none"> a.copyWithin(pos, ini, end) copy elems a.fill(obj, ini, end) fill array with obj a.reverse() reverse array & return it a.sort(cf(a,b)) sort array (unicode sort) a.splice(ini, del, o1, o2...) del&add elem ITERATION METHODS <ul style="list-style-type: none"> a.entries() iterate key/value pair array a.keys() iterate only keys array a.values() iterate only values array CALLBACK FOR EACH METHODS <ul style="list-style-type: none"> b.every(cb(e,i), arg) test until false b.some(cb(e,i), arg) test until true a.map(cb(e,i,a), arg) make array a.filter(cb(e,i,a), arg) make array w/true a.find(cb(e,i,a), arg) return elem w/true n.findIndex(cb(e,i,a), arg) return index f.forEach(cb(e,i,a), arg) exec for each a.reduce(cb(p,e,i,a), arg) accumulative a.reduceRight(cb(p,e,i,a), arg) from end ADD/REMOVE METHODS <ul style="list-style-type: none"> a.pop() remove & return last element a.push(o1, o2...) add element & return length a.shift() remove & return first element a.unshift(o1, o2...) add element & return len 										
r RegExp() PROPERTIES <ul style="list-style-type: none"> n.lastIndex index to start global regexp s.flags active flags of current regexp b.global flag g (search all matches) b.ignoreCase flag i (match lower/upper) b.multiline flag m (match multiple lines) b.sticky flag y (search from lastIndex) b_unicode flag u (enable unicode feat.) s.source current regexp (w/o slashes) METHODS <ul style="list-style-type: none"> a.exec(str) exec search for a match b.test(str) check if regexp match w/str CLASSES <ul style="list-style-type: none"> .any character \t tabulator \d digit [0-9] \r carriage return \D no digit [^0-9] \n line feed \W any alphanumeric char [A-Za-z0-9_] \w no alphanumeric char [^A-Za-z0-9_] \s any space char (space, tab, enter...) \S no space char (space, tab, enter...) \N char with code N [nb] backspace \uN char with unicode N \u0000 NUL char CHARACTER SETS OR ALTERNATION <ul style="list-style-type: none"> [abc] match any character set [abc] match any char. set not enclosed a b match a or b BOUNDARIES <ul style="list-style-type: none"> ^ begin of input \$ end of input \b zero-width word boundary \B zero-width non-word boundary GROUPING <ul style="list-style-type: none"> (x) capture group (?:x) no capture group \n reference to group n captured QUANTIFIERS <ul style="list-style-type: none"> x* preceding x 0 or more times {0,} x+ preceding x 1 or more times {1,} x? preceding x 0 or 1 times {0,1} n{m} n occurrences of x n{x} at least n occurrences of x n,m{b} between n & m occurrences of x ASSERTIONS <ul style="list-style-type: none"> (?=y) x (only if x is followed by y) (?!=y) x (only if x is not followed by y) 	d Date() PROPERTIES <ul style="list-style-type: none"> n.UTC(y, m, d, h, i, s, ms) timestamp n.now() timestamp of current time n.parse(str) convert str to timestamp n.setTime(ts) set UNIX timestamp n.getTime() return UNIX timestamp UNIT SETTERS (ALSO .setUTC(i) methods) <ul style="list-style-type: none"> n.setFullYear(y, m, d) set year (yyyy) n.setMonth(m, d) set month (0-11) n.setDate(d) set day (1-31) n.setHours(h, m, s, ms) set hour (0-23) n.setMinutes(m, s, ms) set min (0-59) n.setSeconds(s, ms) set sec (0-59) n.setMilliseconds(ms) set ms (0-999) UNIT GETTERS (ALSO .getUTC(i) methods) <ul style="list-style-type: none"> n.getDate() return day (1-31) n.getDay() return day of week (0-6) n.getMonth() return month (0-11) n.getFullYear() return year (yyyy) n.getHours() return hour (0-23) n.getMinutes() return minutes (0-59) n.getSeconds() return seconds (0-59) n.getMilliseconds() return ms (0-999) LOCATE & TIMEZONE METHODS <ul style="list-style-type: none"> s.getTimezoneOffset() offset in mins s.toLocaleDateString(locale, options) s.toLocaleTimeString(locale, options) s.toLocaleString(locale, options) s.toUTCString() return UTC date s.toDateString() return American date s.toTimeString() return American time s.toISOString() return ISO8601 date s.toJSON() return date ready for JSON 	b Boolean() no own properties or methods f Function() PROPERTIES <ul style="list-style-type: none"> o.length return number of arguments s.name return name of function o.prototype prototype object METHODS <ul style="list-style-type: none"> o.call(newthis, arg1, arg2...) change this o.apply(newthis, arg1) with args array o.bind(newthis, arg1, arg2...) bound func <table border="1"> <tr> <td>n number</td> <td>d date</td> </tr> <tr> <td>b Na (not-a-number)</td> <td>r regular expression</td> </tr> <tr> <td>s string</td> <td>f function</td> </tr> <tr> <td>b boolean (true/false)</td> <td>o object</td> </tr> <tr> <td>a array</td> <td>u undefined</td> </tr> </table> <p>only available on ECMAScript 6</p> <ul style="list-style-type: none"> t static (ex: Math.random()) n non-static (ex: new Date().getDate()) argument required optional 	n number	d date	b Na (not-a-number)	r regular expression	s string	f function	b boolean (true/false)	o object	a array	u undefined
n number	d date											
b Na (not-a-number)	r regular expression											
s string	f function											
b boolean (true/false)	o object											
a array	u undefined											

n Number()

PROPERTIES

- o **POSITIVE_INFINITY** +oo equivalent
- o **NEGATIVE_INFINITY** -oo equivalent
- o **MAX_VALUE** largest positive value
- o **MIN_VALUE** smallest positive value
- o **EPSILON** diff between 1 & smallest >1
- o **NaN** not-a-number value

METHODS

- o **toExponential(dec)** exp. notation
- o **toFixed(dec)** fixed-point notation
- o **toPrecision(p)** change precision
- o **isFinite(n)** check if number is finite
- o **isInteger(n)** check if number is int.
- o **isNaN(n)** check if number is NaN
- o **parseInt(s, radix)** string to integer
- o **parseFloat(s, radix)** string to float

r RegExp()

PROPERTIES

- o **lastIndex** index to start global regexp
- o **flags** active flags of current regexp
- o **global** flag g (search all matches)
- o **ignoreCase** flag i (match lower/upper)
- o **multiline** flag m (match multiple lines)
- o **sticky** flag y (search from lastIndex)
- o **unicode** flag u (enable unicode feat.)
- o **source** current regexp (w/o slashes)

METHODS

- o **exec(str)** exec search for a match
- o **test(str)** check if regexp match w/str

CLASSES

- o any character \t tabulator
- o digit [0-9] \r carriage return
- o no digit [^0-9] \n line feed
- o any alphanumeric char [A-Za-z0-9_]
- o no alphanumeric char [^A-Za-z0-9_]
- o space char (space, tab, enter,...)
- o no space char (space, tab, enter,...)
- o N char with code N [\b] backspace
- o N char with unicode N \N NUL char

CHARACTER SETS OR ALTERNATION

- o [abc] match any character set
- o [abc] match any char. set not enclosed
- o a|b match a or b

BOUNDRIES

- o ^ begin of input \$ end of input
- o \b zero-width word boundary
- o \B zero-width non-word boundary

GROUPING

- o (x) capture group (?::x) no capture group
- o \n reference to group n captured

QUANTIFIERS

- o * preceding x 0 or more times {0,}
- o + preceding x 1 or more times {1,}
- o ? preceding x 0 or 1 times {0,1}
- o {n} occurrences of x
- o {n,} at least n occurrences of x
- o {n,m} between n & m occurrences of x

ASSERTIONS

- o (?=y) x (only if x is followed by y)
- o (?!y) x (only if x is not followed by y)

s String()

PROPERTIES

- o **length** string size

METHODS

- o **charAt(index)** char at position [i]
- o **charCodeAt(index)** unicode at pos.
- o **fromCharCode(n1, n2...)** code to char
- o **concat(str1,str2...)** combine text +
- o **b.startsWith(str, size)** check beginning
- o **b.endsWith(str, size)** check ending
- o **b.includes(str, from)** include substring?
- o **n.indexOf(str, from)** find substr index
- o **n.lastIndexOf(str, from)** find from end
- o **n.search(regex)** search & return index
- o **n.localeCompare(str, locale, options)**
- o **a.match(regex)** matches against string
- o **s.repeat(n)** repeat string n times
- o **s.replace(str|regex, newstr|func)**
- o **s.slice(ini, end)** str between ini/end
- o **s.substr(ini, len)** substr of len length
- o **s.substring(ini, end)** substr fragment
- o **a.split(sep, limit)** divide string
- o **s.toLowerCase()** string to lowercase
- o **s.toUpperCase()** string to uppercase
- o **s.trim()** remove space from begin/end
- o **s.raw()** template strings with \${vars}

d Date()

METHODS

- o **n.UTC(y, m, d, h, i, ms)** timestamp
- o **n.now()** timestamp of current time
- o **n.parse(str)** convert str to timestamp
- o **n.setTime(ts)** set UNIX timestamp
- o **n.getTimestamp()** return UNIX timestamp
- o **o.pop()** remove & return last element
- o **o.push(o1, o2...)** add element & return length
- o **o.shift()** remove & return first element
- o **o.unshift(o1, o2...)** add element & return len

UNIT SETTERS (ALSO .setUTC() methods)

- o **n.setFullYear(y, m, d)** set year (yyyy)
- o **n.setMonth(m, d)** set month (0-11)
- o **n.setDate(d)** set day (1-31)
- o **n.setHours(h, m, s, ms)** set hour (0-23)
- o **n.setMinutes(m, s, ms)** set min (0-59)
- o **n.setSeconds(s, ms)** set sec (0-59)
- o **n.setMilliseconds(ms)** set ms (0-999)

UNIT GETTERS (ALSO .getUTC() methods)

- o **n.getDate()** return day (1-31)
- o **n.getDay()** return day of week (0-6)
- o **n.getMonth()** return month (0-11)
- o **n.getFullYear()** return year (yyyy)
- o **n.getHours()** return hour (0-23)
- o **n.getMinutes()** return minutes (0-59)
- o **n.getSeconds()** return seconds (0-59)
- o **n.getMilliseconds()** return ms (0-999)

LOCALE & TIMEZONE METHODS

- o **n.getTimezoneOffset()** offset in mins
- o **n.toLocaleDateString(locale, options)**
- o **n.toLocaleTimeString(locale, options)**
- o **n.toLocaleString(locale, options)**
- o **n.UTCString()** return UTC date
- o **n.DateString()** return American date
- o **n.TimeString()** return American time
- o **nISOString()** return ISO8601 date
- o **n.toJSON()** return date ready for JSON

Math

PROPERTIES

- o **Euler's constant**
- o **LN2** natural logarithm of 2
- o **LN10** natural logarithm of 10
- o **LOG2E** base 2 logarithm of E
- o **LOG10E** base 10 logarithm of E
- o **PI** ratio circumference/diameter
- o **SQRT1_2** square root of 1/2
- o **SQRT2** square root of 2

METHODS

- o **a.abs(x)** absolute value
- o **cbrt(x)** cube root
- o **clz32(x)** return leading zero bits (32)
- o **exp(x)** return e^x
- o **expM1(x)** return e^x-1
- o **hypot(x1, x2...)** length of hypotenuse
- o **iimul(a, b)** signed multiply
- o **log(x)** natural logarithm (base e)
- o **log1p(x)** natural logarithm (1+x)
- o **log10(x)** base 10 logarithm
- o **log2(x)** base 2 logarithm
- o **max(x1, x2...)** return max number
- o **min(x1, x2...)** return min number
- o **pow(base, exp)** return base^{exp}
- o **random()** float random number [0,1]
- o **sign(x)** return sign of number
- o **sqrtn(x)** square root of number

ROUND METHODS

- o **ceil(x)** superior round (smallest)
- o **floor(x)** inferior round (largest)
- o **fround(x)** nearest single precision
- o **round(x)** round (nearest integer)
- o **trunc(x)** remove fractional digits

TRIGONOMETRIC METHODS

- o **a.acos(x)** arccosine
- o **a.acosh(x)** hyperbolic arccosine
- o **a.asin(x)** arcsine
- o **a.asinh(x)** hyperbolic arcsine
- o **a.atan(x)** arctangent
- o **a.atan2(x, y)** arctangent of quotient x/y
- o **a.atanh(x)** hyperbolic arctangent
- o **a.cos(x)** cosine
- o **a.cosh(x)** hyperbolic cosine
- o **a.sin(x)** sine
- o **a.sinh(x)** hyperbolic sine
- o **a.tan(x)** tangent
- o **a.tanh(x)** hyperbolic tangent

JSON

METHODS

- o **n.parse(str, tf(k,v))** parse string to object
- o **n.stringify(obj, repf|wl, sp)** convert to str

e Error()

PROPERTIES

- o **s.name** return name of error
- o **s.message** return description of error

Object

PROPERTIES

o **E** Euler's constant

o **LN2** natural logarithm of 2

o **LN10** natural logarithm of 10

o **LOG2E** base 2 logarithm of E

o **LOG10E** base 10 logarithm of E

o **PI** ratio circumference/diameter

o **SQRT1_2** square root of 1/2

o **SQRT2** square root of 2

METHODS

- o **a.assign(dst, src1, src2...)** copy values
- o **a.create(proto, prop)** create obj w/prop
- o **a.defineProperties(obj, prop)**
- o **a.defineProperty(obj, prop, desc)**
- o **a.freeze(obj)** avoid properties changes
- o **a.getOwnPropertyDescriptor(obj, prop)**
- o **a.getOwnPropertyNames(obj)**
- o **a.getOwnPropertySymbols(obj)**
- o **a.getPrototypeOf(obj)** return prototype
- o **a.isVal1, a.isVal2** check if are same value
- o **a.isExtensible(obj)** check if can add prop
- o **a.isFrozen(obj)** check if obj is frozen
- o **a.isSealed(obj)** check if obj is sealed
- o **a.keys(obj)** return only keys of object
- o **a.preventExtensions(obj)** avoid extend
- o **a.seal(obj)** prop are non-configurable
- o **a.setPrototypeOf(obj, prot)** change prot

INSTANCE METHODS

- o **b.hasOwnProperty(prop)** check if exist
- o **b.isPrototypeOf(obj)** test in another obj
- o **b.propertyIsEnumerable(prop)**
- o **c.toString()** return equivalent string
- o **c.toLocaleString()** return locale version
- o **d.valueOf()** return primitive value

p Promise()

METHODS

- o **a.all(obj)** return promise
- o **a.catch(onRejected(s))** = .then(undef,s)
- o **a.then(onFulfilled(v), onRejected(s))**
- o **a.race(obj)** return greedy promise (res/res)
- o **a.resolve(obj)** return resolved promise
- o **a.reject(reason)** return rejected promise

p Proxy()

METHODS

- o **a.apply(obj, arg, arglist)** trap function call
- o **a.construct(obj, arglist)** trap new op
- o **a.defineProperty(obj, prop, desc)**
- o **a.deleteProperty(obj, prop)** trap delete
- o **a.enumerate(obj)** trap for...in
- o **a.get(obj, prop, rec)** trap get property
- o **a.getOwnPropertyDescriptor(obj, prop)**
- o **a.getPrototypeOf(obj)**
- o **a.has(obj, prop)** trap in operator
- o **a.ownKeys(obj)**
- o **a.preventExtensions(obj)**
- o **a.set(obj, prop, value)** trap set property
- o **a.setPrototypeOf(obj, proto)**

globals

METHODS

- o **a.eval(str)** evaluate javascript code
- o **a.isFinite(obj)** check if is a finite number
- o **a.isNaN(obj)** check if is not a number
- o **a.parseInt(s, radix)** string to integer
- o **a.parseFloat(s, radix)** string to float
- o **a.encodeURIComponent(URI)** = to %3D
- o **a.decodeURIComponent(URI)** %3D to =

Set

PROPERTIES

o **n.size** return number of items

METHODS

o **a.add(item)** add item to set

o **a.has(item)** check if item exists

o **a.delete(item)** del item & return if del

o **a.clear()** remove all items from set

ITERATION METHODS

o **a.entries()** iterate items

o **a.values()** iterate only values of items

CALLBACK FOR EACH METHODS

o **a.forEach(cb(e,i,a), arg)** exec for each

m Map()

PROPERTIES

o **n.size** return number of elements

METHODS

o **m.set(key, value)** add pair key=value

o **m.get(key)** return value of key

o **m.has(key)** check if key exist

o **m.delete(key)** del elem. & return if ok

o **m.clear()** remove all elements from map

ITERATION METHODS

o **m.entries()** iterate elements

o **m.keys()** iterate only keys

o **m.values()** iterate only values

CALLBACK FOR EACH METHODS

o **m.forEach(cb(e,i,a), arg)** exec for each

Symbol()

PROPERTIES

o **s.iterator** specifies default iterator

o **s.match** specifies match of regexp

o **s.species** specifies constructor function

METHODS

o **s.for(key)** search existing symbols

o **s.keyFor(sym)** return key from global reg

g Generator()

METHODS

o **a.next(value)** return obj w/ value, done

o **a.return(value)** return value & true done

o **a.throw(exception)** throw an error

Others

FAST TIPS

var declare variable

let declare block scope local variable

const declare constant (read-only)

func(a=1) default parameter value

func(...a) rest argument (spread operator)

(a => {}) function equivalent (fat arrow)

'string \${a}' template with variables

0bn binary (2) number n to decimal

0on octal (8) number n to decimal

0xn hexadecimal (16) number n to decimal

for (i in array) { ... } iterate array, i = index

for (e of array) { ... } iterate array, e = value

class B extends A () {} class sugar syntax

Goal

- Learn JavaScript as a language
- Understand the specific semantics and programming patterns
 - We assume a programming knowledge in other languages
- Updated to ES6 (2015) language features
- Supported by server-side (Node.js) and client-side (browsers) run-time environments
 - More recent language additions also supported (through *transpiling*)

Outline

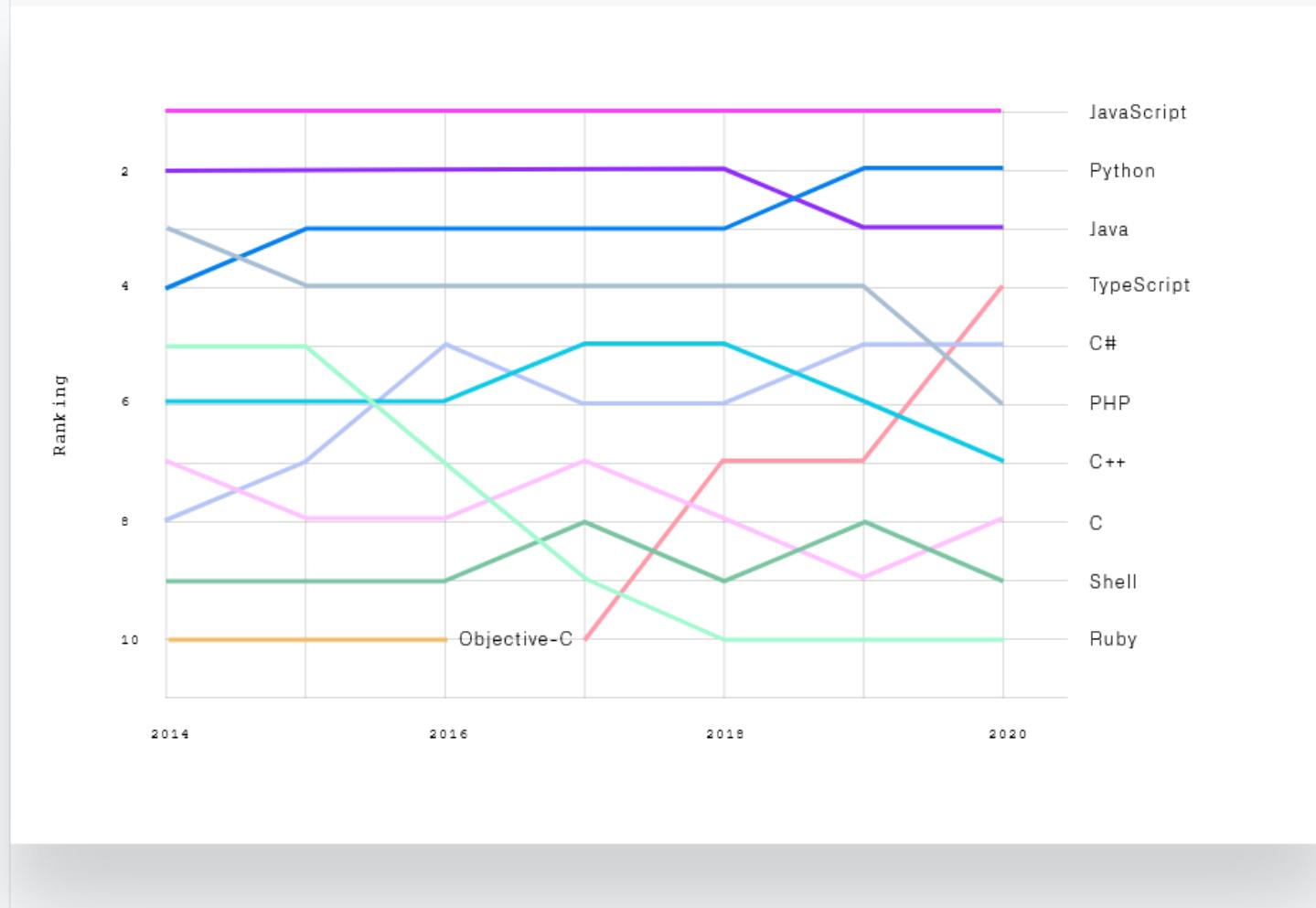
- What is JavaScript?
- History and versions
- Language structure
- Types, variables
- Expressions
- Control structures
- Arrays
- Strings

JavaScript – The language of the Web

WHAT IS JAVASCRIPT?

```
// The languages that dominated
```

Top languages over the years



source: <https://octoverse.github.com/#top-languages>

JavaScript

- JavaScript (JS) is a programming language
- It is currently the only programming language that a browser can execute natively...
- ... and it also runs on a computer, like other programming languages (thanks to Node.js)
- It has **nothing** to do with Java
 - named that way for *marketing reasons*, only
- The first version was written in 10 days (!)
 - several fundamental language decisions were made because of company politics and not technical reasons!

JavaScript – The language of the Web

HISTORY AND VERSIONS

JAVASCRIPT VERSIONS



Brendan Eich

- ▶ **JAVASCRIPT (December 4th 1995)** Netscape and Sun press release
- ▶ **ECMAScript Standard Editions:** <https://www.ecma-international.org/ecma-262/>
- ▶ **ES1 (June 1997)** Object-based, Scripting, Relaxed syntax, Prototypes
- ▶ **ES2 (June 1998)** Editorial changes for ISO 16262

10
yrs

- ▶ **ES3 (December 1999)** Regexp, Try/Catch, Do-While, String methods

- ▶ **ES5 (December 2009)** Strict mode, JSON, .bind, Object mts, Array mts
- ▶ **ES5.1 (June 2011)** Editorial changes for ISO 16262:2011

Main
target

- ▶ **ES6 (June 2015)** Classes, Modules, Arrow Fs, Generators, Const/Let, Destructuring, Template Literals, Promise, Proxy, Symbol, Reflect

Also: ES2015

- ▶ **ES7 (June 2016)** Exponentiation operator (**) and Array Includes

Also: ES2016

- ▶ **ES8 (June 2017)** Async Fs, Shared Memory & Atomics

Also: ES2017

ES9,
ES10,
...

JavaScript versions

- ECMAScript (also called ES) is the official name of JavaScript (JS) standard
- ES6, ES2015, ES2016 etc. are implementations of the standard
- All browsers used to run ECMAScript 3
- ES5, and ES2015 (=ES6) were huge versions of JavaScript
- Then, yearly release cycles started
 - By the committee behind JS: TC39, backed by Mozilla, Google, Facebook, Apple, Microsoft, Intel, PayPal, SalesForce, etc.
- **ES2015 (=ES6) is covered in this course**

Official ECMA standard (formal and unreadable)

The screenshot shows the official website for the ECMAScript 2019 Language Specification. At the top right is the ECMA International logo, which consists of a stylized orange circle followed by the word "ecma" in lowercase and "INTERNATIONAL" in uppercase. Below the logo, the title "ECMA-262, 10th edition, June 2019" is displayed in orange, and "ECMAScript® 2019 Language Specification" in a larger orange font. On the left side, there is a sidebar titled "TABLE OF CONTENTS" containing a detailed list of chapters from 1 to 27, along with sections A through H. The main content area features a section titled "Contributing to this Specification" with information about GitHub repositories, pull requests, editors, and community mailing lists. At the bottom of this section, there is a link to the "colophon". Below the sidebar, the word "Introduction" is visible.

<https://www.ecma-international.org/ecma-262/>

JavaScript Engines

- V8 (Chrome V8) by Google
 - used in Chrome/Chromium, Node.js and Microsoft Edge
- SpiderMonkey by Mozilla Foundation
 - Used in Firefox/Gecko
- ChakraCore by Microsoft
 - it was used in Edge
- JavaScriptCore by Apple
 - used in Safari

Standard vs. Implementation (in browsers)

Browser compatibility

Update compatibility data on GitHub

	Desktop						Mobile					
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
FetchEvent	40	Yes	44 *	No	27	No	40	40	44	27	No	4.0
FetchEvent() constructor	40	Yes	44 *	No	27	No	40	40	44	27	No	4.0
client	42	?	44	No	27	No	42	44	No	?	No	4.0
clientId	49	?	45 *	No	36	No	49	49	45	36	No	5.0
isReload	45	17	44 *	No	32	No	45	45	44	32	No	5.0
navigationPreload	59	?	?	No	46	No	59	59	?	43	No	7.0
preloadResponse	59	18	?	No	46	No	59	59	?	43	No	7.0
replacesClientId	No	18	65	No	No	No	No	No	65	No	No	No
request	Yes	?	44	No	Yes	No	Yes	Yes	?	Yes	No	Yes
respondWith	42 *	?	59 *	No	29	No	42 *	42 *	?	29	No	4.0
resultingClientId	72	18	65	No	60	No	72	72	65	50	No	No
targetClientId	?	?	?	No	?	No	?	?	?	?	No	?

What are we missing? ↗

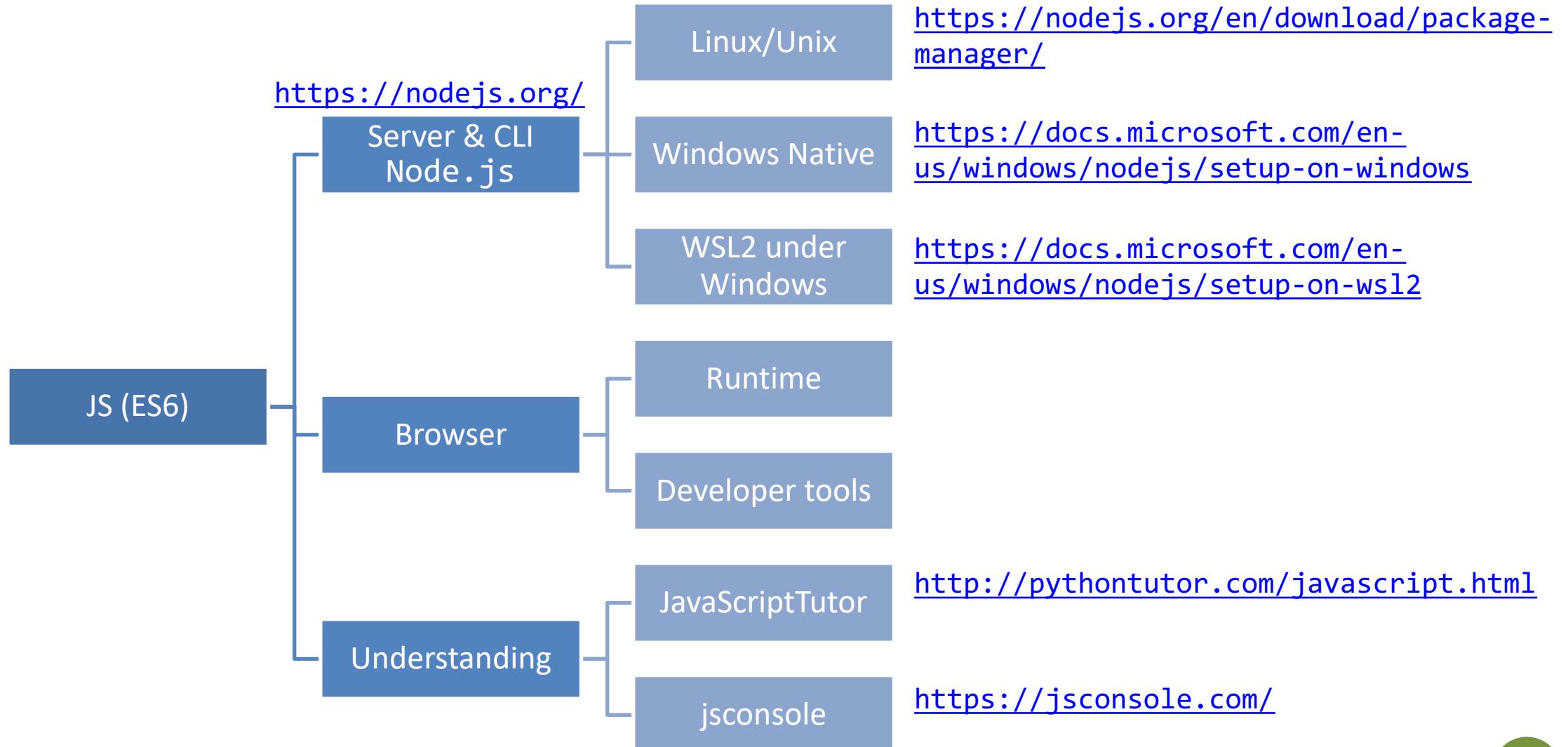
Legend:

- Full support
- No support
- Compatibility unknown
- Experimental. Expect behavior to change in the future.
- Non-standard. Expect poor cross-browser support.
- Deprecated. Not for use in new websites.
- See implementation notes.

JS Compatibility

- JS is *backwards-compatible*
 - once something is accepted as valid JS, there will not be a future change to the language that causes that code to become invalid JS
 - TC39 members: "we don't break the web!"
- JS is not *forwards-compatible*
 - new additions to the language will not run in an older JS engine and may crash the program
- **strict mode** was introduced to disable very old (and dangerous) semantics
- Supporting multiple versions is achieved by:
 - *Transpiling* – Babel (<https://babeljs.io>) converts from newer JS syntax to an equivalent older syntax
 - *Polyfilling* – user- (or library-)defined functions and methods that “fill” the lack of a feature by implementing the newest available one

JS Execution Environments



JavaScriptTutor

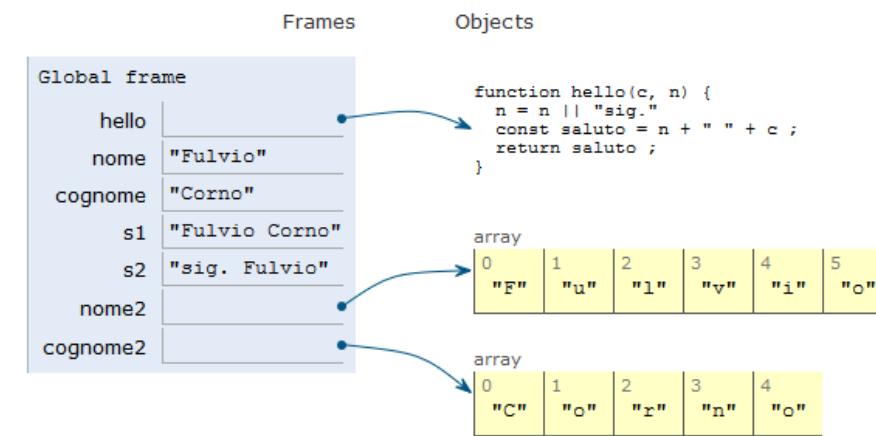
Write code in **JavaScript ES6** (drag lower right corner to resize code editor)

```
1 let nome = "Fulvio" ;
2 let cognome = "Corno" ;
3
4 function hello(c, n) {
5   n = n || "sig."
6   const saluto = n + " " + c ;
7   return saluto ;
8 }
9
10 let s1 = hello(cognome, nome)
11 let s2 = hello(nome)
12
13 let nome2 = [...nome]
14 let cognome2 = [...cognome]
```

▶ line that just executed
→ next line to execute

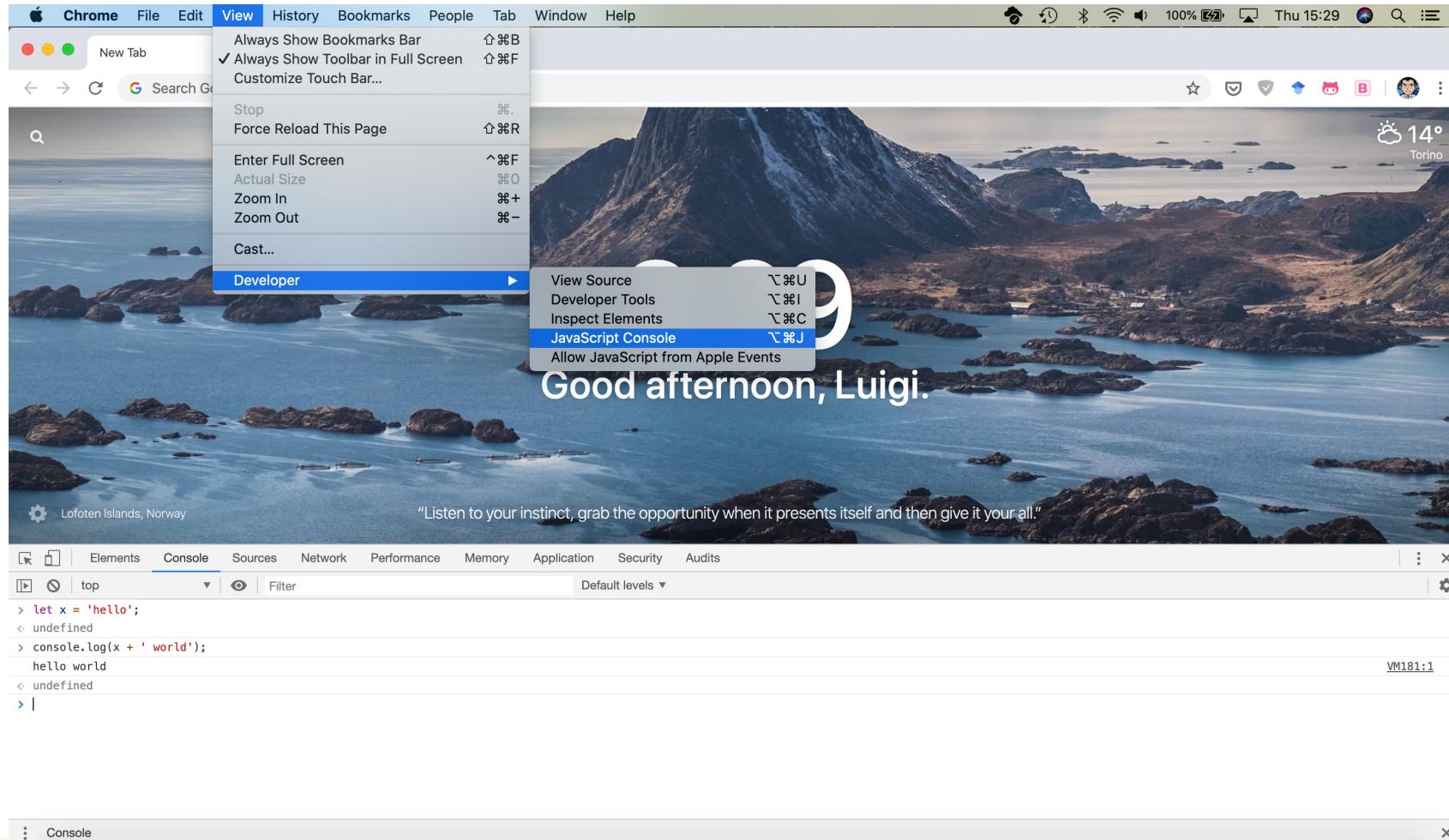
<< First < Prev Next > >> Last

Done running (16 steps)



<http://pythontutor.com/javascript.html>

Browser and JS console



JavaScript – The language of the Web

LANGUAGE STRUCTURE

Lexical structure

- One File = One JS program
 - Each file is loaded independently and
 - Different files/programs may communicate through *global state*
 - The “module” mechanism extends that (provides state sharing in a clean way)
- The file is entirely *parsed*, and then *executed* from top to bottom
- Relies on a *standard library*
 - and many additional *APIs* provided by the execution environment

Lexical structure

- JavaScript is written in Unicode (do not abuse), so it also supports non-latin characters for names and strings
 - even emoji
- Semicolons (;) are not mandatory (automatically inserted)
- Case sensitive
- Comments as in C /* .. */ and //
- Literals and identifiers (start with letter, \$, _)
- Some reserved words
- C-like syntax

```
> let ööö = 'appalled'  
> ööö  
'appalled'
```

```
> let x = '😊';  
< undefined  
> console.log(x);  
😊
```

Semicolon (;)

- Argument of debate in the JS community
- JS inserts them as needed
 - When next line starts with code that breaks the current one
 - When the next line starts with }
 - When there is return, break, throw, continue on its own line
- Be careful that forgetting semicolon can lead to unexpected behavior
 - A newline does not automatically insert a semicolon: if the next line starts with (or [, it is interpreted as function call or array access
- We will **loosely** follow the Google style guide, so we will always insert semicolons after each statement
 - <https://google.github.io/styleguide/jsguide.html>

Strict Mode

```
// first line of file  
"use strict" ;  
// always!!
```

- Directive introduced in ES5: "use strict" ;
 - Compatible with older version (it is just a string)
- Code is executed in *strict mode*
 - This fixes some important language deficiencies and provides stronger error checking and security
 - Examples:
 - fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode
 - eliminates some JavaScript silent errors by changing them to throw errors
 - functions invoked as functions and not as methods of an object have **this** undefined
 - cannot define 2 or more properties or function parameters with the same name
 - no octal literals (base 8, starting with 0)
 - ...



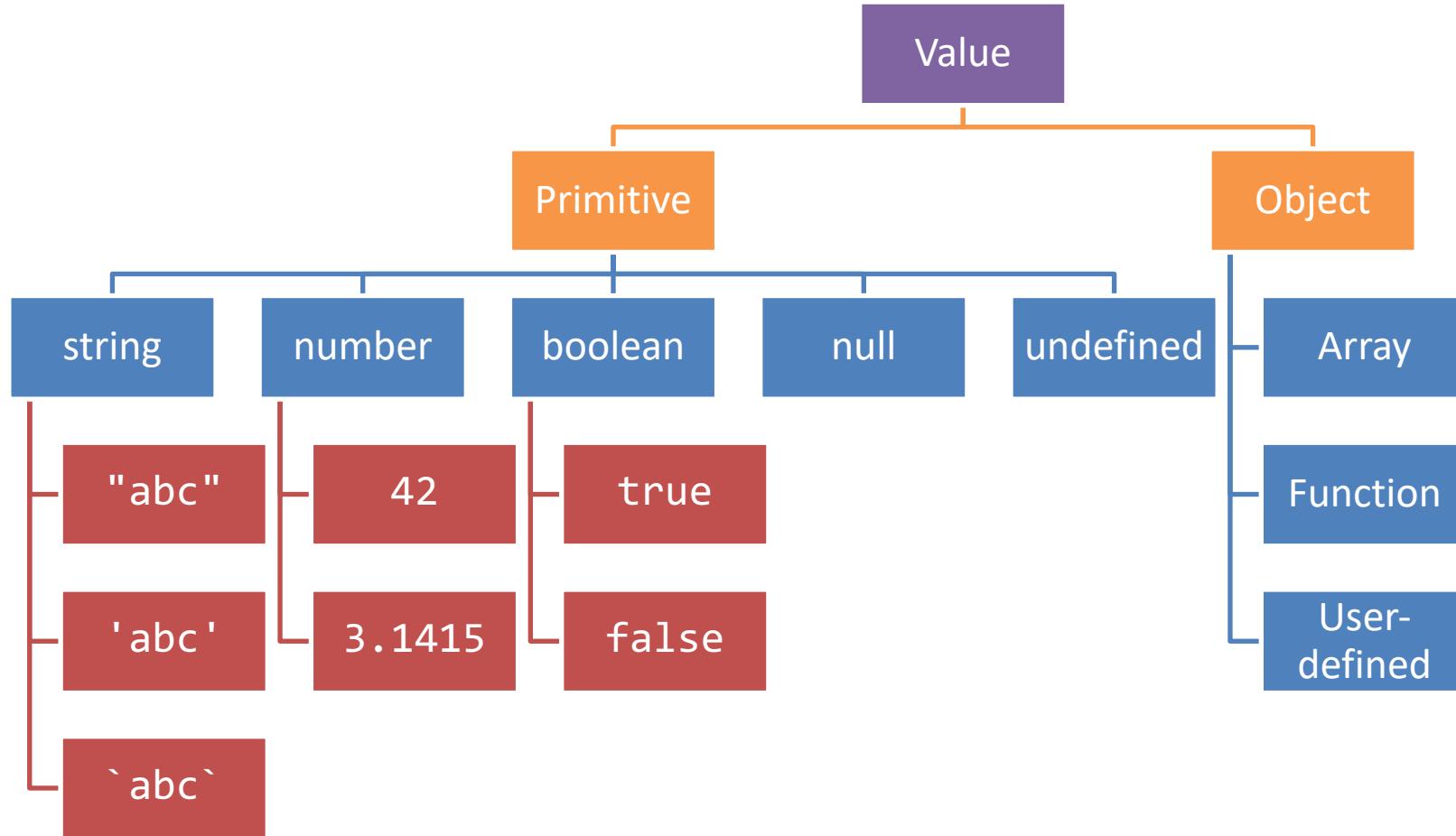
JavaScript: The Definitive Guide, 7th Edition
Chapter 2. Types, Values, and Variables

JavaScript – The language of the Web

TYPES AND VARIABLES

Values and Types

Values have types.
Variables don't.



Boolean, true-*truthy*, false-*falsy*, comparisons

- ‘boolean’ type with literal values: true, false
- When converting to boolean

- The following values are ‘falsy’

- 0, -0, NaN, undefined, null, '' (empty string)

- Every other value is ‘truthy’

- 3, 'false', [] (empty array), {} (empty object)

- Booleans and Comparisons

- a == b // convert types and compare results

- a === b // inhibit automatic type conversion and compare results

```
> Boolean(3)
true
> Boolean('')
false
> Boolean(' ')
true
```

Number

- No distinction between integers and reals
- Automatic conversions according to the operation
- There is also a distinct type "BigInt" (*ES11, July 2020*)
 - an arbitrary-precision integer, can represent 2^{53} numbers
 - `123456789n`
 - With suffix ‘n’

Special values

- **undefined**: variable declared but not initialized
 - Detect with: `typeof variable === 'undefined'`
 - `void x` always returns undefined
- **null**: an empty value
- Null and Undefined are called *nullish values*
- **NaN (Not a Number)**
 - It is actually a number
 - Invalid output from arithmetic operation or parse operation

Variables

- Variables are *pure references*: they refer to a *value*
- The same variable may refer to different values (even of different types) at different times

- Declaring a variable:
 - **let**
 - **const**
 - **var**

```
> v = 7 ;  
7  
> v = 'hi' ;  
'hi'
```

```
> let a = 5  
> const b = 6  
> var c = 7  
> a = 8  
8  
> b = 9  
Thrown:  
TypeError: Assignment to  
constant variable.  
> c = 10  
10
```

Variable declarations

Declarator	Can reassign?	Can re-declare?	Scope	Hoisting *	Note
<code>let</code>	Yes	No	Enclosing block <code>{...}</code>	No	Preferred
<code>const</code>	No [§]	No	Enclosing block <code>{...}</code>	No	Preferred
<code>var</code>	Yes	Yes	Enclosing function, or global	Yes, to beginning of function or file	<i>Legacy, beware its quirks, try not to use</i>
None (implicit)	Yes	N/A	Global	Yes	<i>Forbidden in strict mode</i>

[§] Prevents reassignment (`a=2`), does not prevent changing the value of the referred object (`a.b=2`)

* Hoisting = “lifting up” the definition of a variable (not the initialization!) to the top of the current scope (e.g., the file or the function)

Scope

```
"use strict" ;  
  
let a = 1 ;  
const b = 2 ;  
let c = true ;  
  
let a = 5 ; // SyntaxError: Identifier 'a' has already been declared
```

Scope

```
"use strict" ;  
  
let a = 1 ;  
const b = 2 ;  
let c = true ;  
  
{ // creating a new scope...  
    let a = 5 ;  
    console.log(a) ;  
}  
  
console.log(a) ;
```

Typically, you don't
create a new scope in
this way!

Each { } is called a **block**. 'let' and 'const' variables are *block-scoped*.

They exist only in their defined and inner scopes.

Scope and Hoisting

```
"use strict" ;  
  
function example(x) {  
    let a = 1 ;  
  
    console.log(a) ;      // 1  
    console.log(b) ;      // ReferenceError: b is not defined  
    console.log(c) ;      // undefined  
  
    if( x>1 ) {  
        let b = a+1 ;  
        var c = a*2 ;  
    }  
  
    console.log(a) ; // 1  
    console.log(b) ; // ReferenceError: b is not defined  
    console.log(c) ; // 2  
}  
  
example(2) ;
```



JavaScript: The Definitive Guide, 7th Edition
Chapter 2. Types, Values, and Variables
Chapter 3. Expressions and Operators

Mozilla Developer Network
JavaScript Guide » Expressions and operators

JavaScript – The language of the Web

EXPRESSIONS

Operators

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- String operators
- Conditional (ternary) operator
- Comma operator
- Unary operators
- Relational operators



Full reference and operator precedence:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precidence#Table

Assignment

- `let variable = expression ;` // declaration with initialization
- `variable = expression ;` // reassignment

Name	Shorthand operator	Meaning
Assignment	<code>x = y</code>	<code>x = y</code>
Addition assignment	<code>x += y</code>	<code>x = x + y</code>
Subtraction assignment	<code>x -= y</code>	<code>x = x - y</code>
Multiplication assignment	<code>x *= y</code>	<code>x = x * y</code>
Division assignment	<code>x /= y</code>	<code>x = x / y</code>
Remainder assignment	<code>x %= y</code>	<code>x = x % y</code>
Exponentiation assignment <small>⚠️</small>	<code>x **= y</code>	<code>x = x ** y</code>
Left shift assignment	<code>x <= y</code>	<code>x = x << y</code>
Right shift assignment	<code>x >= y</code>	<code>x = x >> y</code>
Unsigned right shift assignment	<code>x >>= y</code>	<code>x = x >>> y</code>
Bitwise AND assignment	<code>x &= y</code>	<code>x = x & y</code>
Bitwise XOR assignment	<code>x ^= y</code>	<code>x = x ^ y</code>
Bitwise OR assignment	<code>x = y</code>	<code>x = x y</code>

Comparison operators

Operator	Description	Examples returning true
Equal (==)	Returns <code>true</code> if the operands are equal.	<code>3 == var1</code> <code>"3" == var1</code> <code>3 == '3'</code>
Not equal (!=)	Returns <code>true</code> if the operands are not equal.	<code>var1 != 4</code> <code>var2 != "3"</code>
Strict equal (===)	Returns <code>true</code> if the operands are equal and of the same type. See also Object.is and sameness in JS .	<code>3 === var1</code>
Strict not equal (!==)	Returns <code>true</code> if the operands are of the same type but not equal, or are of different type.	<code>var1 !== "3"</code> <code>3 !== '3'</code>
Greater than (>)	Returns <code>true</code> if the left operand is greater than the right operand.	<code>var2 > var1</code> <code>"12" > 2</code>
Greater than or equal (>=)	Returns <code>true</code> if the left operand is greater than or equal to the right operand.	<code>var2 >= var1</code> <code>var1 >= 3</code>
Less than (<)	Returns <code>true</code> if the left operand is less than the right operand.	<code>var1 < var2</code> <code>"2" < 12</code>
Less than or equal (<=)	Returns <code>true</code> if the left operand is less than or equal to the right operand.	<code>var1 <= var2</code> <code>var2 <= 5</code>



Comparing Objects

- Comparison between objects with == or === compares the *references* to objects
 - True only if they are *the same object*
 - False if they are *identical objects*
- Comparison with < > <= >= first converts the object (into a Number, or more likely a String), and then compares the values
 - It works, but may be unpredictable, depending on the string format

```
> a={x:1}
{ x: 1 }

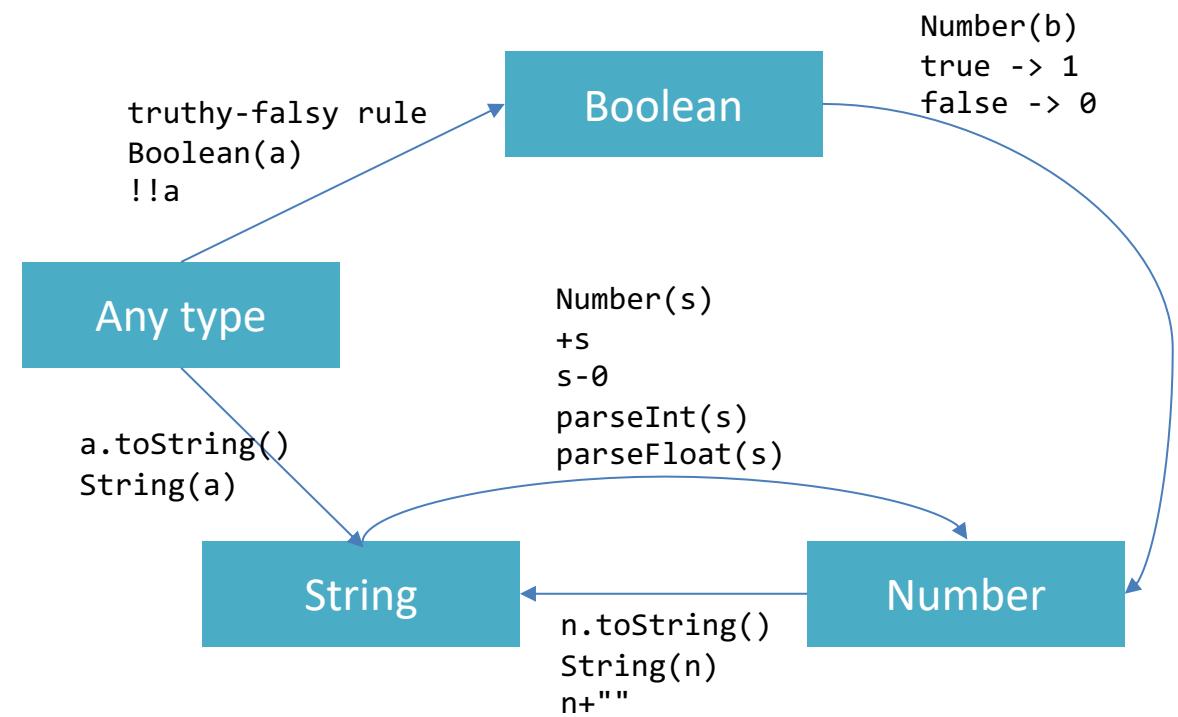
> b={x:1}
{ x: 1 }

> a==b
false

> a==b
false
```

Automatic Type Conversions

- JS tries to apply type conversions between primitive types, before applying operators
- Some language constructs may be used to “force” the desired conversions
- Using `==` applies conversions
- Using `===` prevents conversions



<https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/types-grammar/ch4.md>

Logical operators

Operator	Usage	Description
Logical AND (<code>&&</code>)	<code>expr1 && expr2</code>	Returns <code>expr1</code> if it can be converted to <code>false</code> ; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>&&</code> returns <code>true</code> if both operands are true; otherwise, returns <code>false</code> .
Logical OR (<code> </code>)	<code>expr1 expr2</code>	Returns <code>expr1</code> if it can be converted to <code>true</code> ; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code> </code> returns <code>true</code> if either operand is true; if both are false, returns <code>false</code> .
Logical NOT (<code>!</code>)	<code>!expr</code>	Returns <code>false</code> if its single operand that can be converted to <code>true</code> ; otherwise, returns <code>true</code> .

Common operators

Or string concatenation	Addition (+)	Logical AND (&&)	Useful idiom: a b if a then a else b (a, with default b)
	Decrement (--)	Logical OR ()	
	Division (/)	Logical NOT (!)	
	Exponentiation (**)	Nullish coalescing operator (??)	
	Increment (++)	Conditional operator (c ? t : f)	
	Multiplication (*)	typeof	
	Remainder (%)		
	Subtraction (-)		
	Unary negation (-)		
	Unary plus (+)		

Mathematical functions (Math global object)

- **Constants:** Math.E, Math.LN10, Math.LN2, Math.LOG10E, Math.LOG2E, Math.PI, Math.SQRT1_2, Math.SQRT2
- **Functions:** Math.abs(), Math.acos(), Math.acosh(), Math.asin(), Math.asinh(), Math.atan(), Math.atan2(), Math.atanh(), Math.cbrt(), Math.ceil(), Math.clz32(), Math.cos(), Math.cosh(), Math.exp(), Math.expm1(), Math.floor(), Math.fround(), Math.hypot(), Math.imul(), Math.log(), Math.log10(), Math.log1p(), Math.log2(), Math.max(), Math.min(), Math.pow(), Math.random(), Math.round(), Math.sign(), Math.sin(), Math.sinh(), Math.sqrt(), Math.tan(), Math.tanh(), Math.trunc()



JavaScript: The Definitive Guide, 7th Edition
Chapter 4. Statements

Mozilla Developer Network
[JavaScript Guide » Control Flow and Error Handling](#)
[JavaScript Guide » Loops and Iteration](#)

JavaScript – The language of the Web

CONTROL STRUCTURES

Conditional statements

```
if (condition) {  
    statement_1;  
} else {  
    statement_2;  
}
```

if truthy (beware!)

```
if (condition_1) {  
    statement_1;  
} else if (condition_2) {  
    statement_2;  
} else if (condition_n) {  
    statement_n;  
} else {  
    statement_last;  
}
```

```
switch (expression) {  
    case label_1:  
        statements_1  
        [break;]  
    case label_2:  
        statements_2  
        [break;]  
        ...  
    default:  
        statements_def  
        [break;]  
}
```

May also be a string

Loop statements

```
for ([initialExpression]; [condition]; [incrementExpression]) {  
    statement ;  
}
```

Usually declares loop
variable

```
do {  
    statement ;  
} while (condition);
```

May use break; or
continue;

```
while (condition) {  
    statement ;  
}
```

Special 'for' statements

```
for (variable in object) {
    statement ;
}
```

```
for (variable of iterable) {
    statement ;
}
```

- Iterates the variable over all the enumerable **properties** of an **object**
- Do not use to traverse an array (use numerical indexes, or for-of)

```
for( let a in {x: 0, y:3}) {
    console.log(a) ;
}
```

x
y

- Iterates the variable over all values of an *iterable object* (including Array, Map, Set, string, arguments ...)
- Returns the *values*, not the keys

```
for( let a of [4,7]) {
    console.log(a) ;
}
```

4
7

```
for( let a of "hi" ) {
    console.log(a) ;
}
```

h
i

Other iteration methods

- Functional programming (strongly supported by JS) allows other methods to iterate over a collection (or any iterable object)
 - `a.forEach()`
 - `a.map()`
- They will be analyzed later

Exception handling

```
try {  
    statements ;  
} catch(e){  
    statements ;  
}
```

```
try {  
    statements ;  
} catch(e) {  
    statements ;  
} finally {  
    statements ;  
}
```

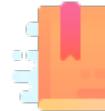
throw object ;

Exception object

Executed in any case, at
the end of try and catch
blocks

EvalError
RangeError
ReferenceError
SyntaxError
TypeError
URIError
DOMException

Contain fields: name,
message



JavaScript: The Definitive Guide, 7th Edition
Chapter 6. Arrays

Mozilla Developer Network
[JavaScript Guide » Indexed Collections](#)

JavaScript – The language of the Web

ARRAYS

Arrays

- Rich of functionalities
- Elements do not need to be of the same type
- Simplest syntax: []
- Property `.length`
- Distinguish between methods that:
 - Modify the array (**in-place**)
 - Return a **new** array

Creating an array

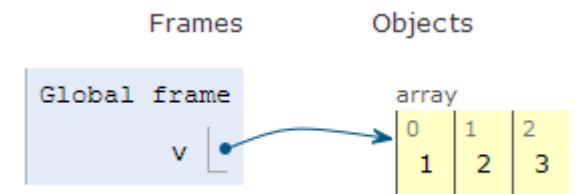
```
let v = [] ;
```

Elements are indexed at positions 0...length-1

Do not access elements outside range

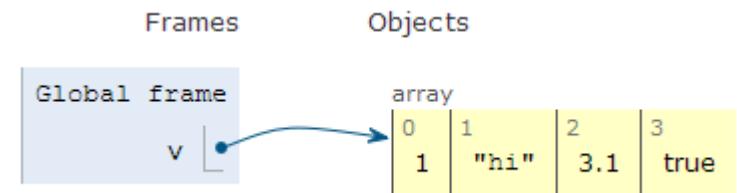
```
let v = [1, 2, 3] ;
```

```
let v = Array.of(1, 2, 3) ;
```



```
let v = [1, "hi", 3.1, true];
```

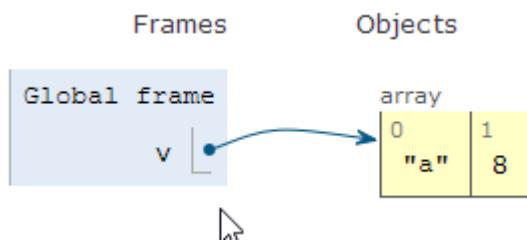
```
let v = Array.of(1, "hi",  
3.1, true) ;
```



Adding elements

.length adjusts automatically

```
let v = [] ;  
v[0] = "a" ;  
v[1] = 8 ;  
v.length // 2
```

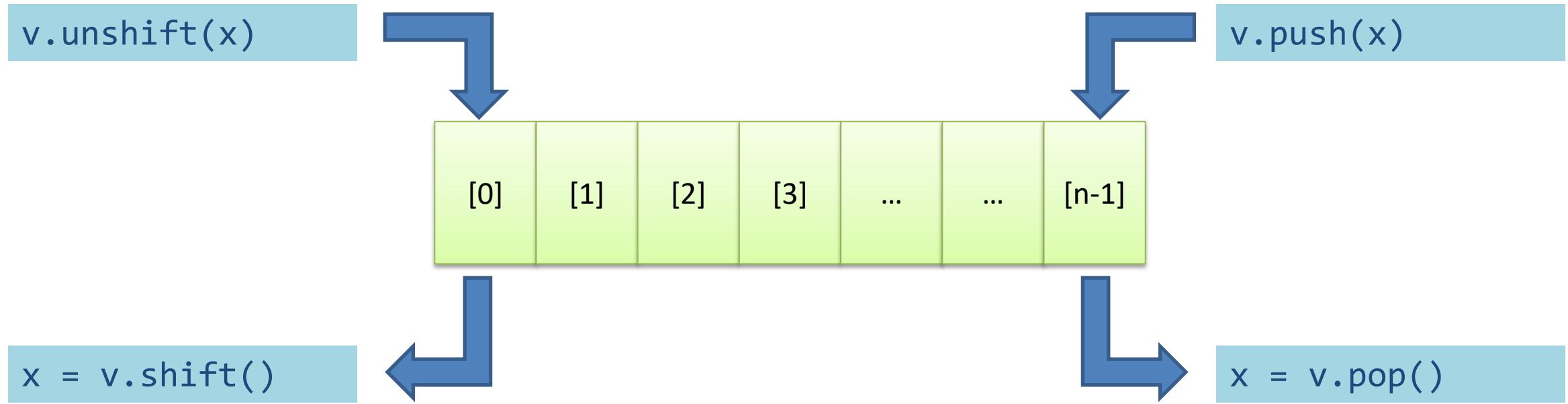


```
let v = [] ;  
v.push("a") ;  
v.push(8) ;  
v.length // 2
```

.push() adds at the end of the array

.unshift() adds at the beginning of the array

Adding and Removing from arrays (in-place)



Copying arrays

```
let v = [] ;  
v[0] = "a" ;  
v[1] = 8 ;  
  
let alias = v ;  
alias[1] = 5 ;
```

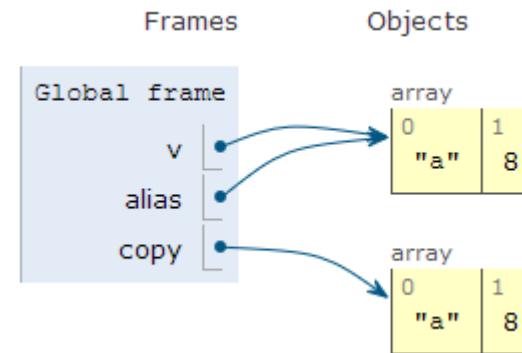
```
> console.log(v); ?  
[ 'a', 5 ]  
undefined  
> console.log(alias);  
[ 'a', 5 ]  
undefined
```

Copying arrays

```
let v = [] ;  
v[0] = "a" ;  
v[1] = 8 ;  
  
let alias = v ;  
let copy = Array.from(v) ;
```

Array.from creates a
shallow copy

Creates an array from
any iterable object



Iterating over Arrays

Preferred

- Iterators: `for ... of`, `for (...; ...; ...)`
- Iterators: `forEach (f)`
 - `f` is a function that processes the element
- Iterators: `every (f)`, `some (f)`
 - `f` is a function that returns true or false
- Iterators that return a new array: `map (f)`, `filter (f)`
 - `f` works on the element of the array passed as parameter
- Reduce: exec a callback function on all items to progressively compute a result

Main array methods

- `.concat()`
 - joins two or more arrays and returns a **new** array.
- `.join(delimiter = ',')`
 - joins all elements of an array into a (**new**) string.
- `.slice(start_index, upto_index)`
 - extracts a section of an array and returns a **new** array.
- `.splice(index, count_to_remove, addElement1, addElement2, ...)`
 - removes elements from an array and (optionally) replaces them, **in place**
- `.reverse()`
 - transposes the elements of an array, **in place**
- `.sort()`
 - sorts the elements of an array **in place**
- `.indexOf(searchElement[, fromIndex])`
 - searches the array for searchElement and returns the **index** of the first match
- `.lastIndexOf(searchElement[, fromIndex])`
 - like indexOf, but starts at the end
- `.includes(valueToFind[, fromIndex])`
 - search for a certain value among its entries, returning true or false

Destructuring assignment

- Value of the right-hand side of equal signal are extracted and stored in the variables on the left

```
let [x,y] = [1,2];
[x,y] = [y,x]; // swap
```

```
var foo = ['one', 'two', 'three'];
var [one, two, three] = foo;
```

- Useful especially with passing and returning values from functions

```
let [x,y] = toCartesian(r,theta);
```

Spread operator (3 dots: ...)

- Expands an iterable object in its parts, when the syntax requires a comma-separated list of elements

```
let [x, ...y] = [1,2,3,4]; // we obtain y == [2,3,4]
```

```
const parts = ['shoulders', 'knees'];
const lyrics = ['head', ...parts, 'and', 'toes']; // ["head", "shoulders",
"knees", "and", "toes"]
```

- Works on the left- and right-hand side of the assignment

Curiosity

- Copy by value:
 - `const b = Array.from(a)`
- Can be emulated by
 - `const b = Array.of(...a)`
 - `const b = [...a]`

Frequent
idiom



JavaScript: The Definitive Guide, 7th Edition
Chapter 2. Types, Values, and Variables

Mozilla Developer Network
JavaScript Guide » Text Formatting

JavaScript – The language of the Web

STRINGS

Strings in JS

- A string is an **immutable** ordered sequence of Unicode^(*) characters
- The **length** of a string is the number of characters it contains (not bytes)
- JavaScript's strings use zero-based indexing
 - The empty string is the string of length 0
- JavaScript does not have a special type that represents a single character (use length-1 strings).
- String literals may be defined with 'abc' or "abc"
 - Note: when dealing with JSON parsing, only " " can be correctly parsed

String operations

- All operations always return **new** strings
 - Consequence of immutability
- `s[3]`: indexing
- `s1 + s2`: concatenation
- `s.length`: number of characters
 - Note: `.length` , not `.length()`

String methods

Method	Description
<code>charAt</code> , <code>charCodeAt</code> , <code>codePointAt</code>	Return the character or character code at the specified position in string.
<code>indexOf</code> , <code>lastIndexOf</code>	Return the position of specified substring in the string or last position of specified substring, respectively.
<code>startsWith</code> , <code>endsWith</code> , <code>includes</code>	Returns whether or not the string starts, ends or contains a specified string.
<code>concat</code>	Combines the text of two strings and returns a new string.
<code>fromCharCode</code> , <code>fromCodePoint</code>	Constructs a string from the specified sequence of Unicode values. This is a method of the String class, not a String instance.
<code>split</code>	Splits a <code>String</code> object into an array of strings by separating the string into substrings.
<code>slice</code>	Extracts a section of a string and returns a new string.
<code>substring</code> , <code>substr</code>	Return the specified subset of the string, either by specifying the start and end indexes or the start index and a length.
<code>match</code> , <code>matchAll</code> , <code>replace</code> , <code>search</code>	Work with regular expressions.
<code>toLowerCase</code> , <code>toUpperCase</code>	Return the string in all lowercase or all uppercase, respectively.
<code>normalize</code>	Returns the Unicode Normalization Form of the calling string value.
<code>repeat</code>	Returns a string consisting of the elements of the object repeated the given times.
<code>trim</code>	Trims whitespace from the beginning and end of the string.

Unicode issues

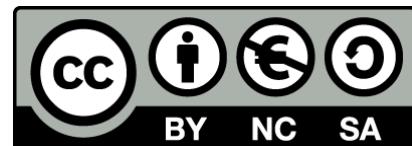
- Strings are a sequence of 16-bit Unicode ‘code units’
 - Fine for all Unicode characters from 0000 to FFFF
 - Characters (‘graphemes’) from 010000 to 10FFFF are represented by *a pair of code units* (and they occupy 2 index positions)
 - Therefore, not all string methods work well with Unicode characters above FFFF (e.g., emojis, flags, ...)
- For more details: <https://dmitripavlutin.com/what-every-javascript-developer-should-know-about-unicode/>

Template literals

- Strings included in `backticks` can embed expressions delimited by `{}$`
- The `value` of the expression is *interpolated* into the string

```
let name = "Bill";  
let greeting = `Hello ${ name }.`;  
// greeting == "Hello Bill."
```

- Very useful and quick for string formatting
- Template literals may also span multiple lines



License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
 - **Share** — copy and redistribute the material in any medium or format
 - **Adapt** — remix, transform, and build upon the material
 - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
 - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - **NonCommercial** — You may not use the material for commercial purposes.
 - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
 - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>