

<WA/>

2025

JavaScript (basics)

“The” language of the Web

Fulvio Corno

Luigi De Russis

Enrico Masala

JS

JavaScript

Cheat Sheet

JS

Programming Language of Web

Number()

PROPERTIES

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

METHODS

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

Regex()

PROPERTIES

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

METHODS

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

CLASSES

- .** any character
- \d** digit [0-9]
- \D** no digit [^0-9]
- \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...)
- \xn** char with code *N*
- \b** backspace
- \uN** char with unicode *N*
- \0** NUL char

CHARACTER SETS OR ALTERNATION

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

BOUNDARIES

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

GROUPING

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

QUANTIFIERS

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

ASSERTIONS

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

String()

PROPERTIES

- .length** string size

METHODS

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

Date()

METHODS

- .UTC(*y*, *m*, *d*, *h*, *i*, *s*, *ms*)** timestamp
- .now()** timestamp of current time
- .parse(*str*)** convert str to timestamp
- .setTime(*ts*)** set UNIX timestamp
- .getTime()** return UNIX timestamp

UNIT SETTERS (ALSO .setUTC() methods)

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

UNIT GETTERS (ALSO .getUTC() methods)

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

LOCALS & TIMEZONE METHODS

- .getTimezoneOffset()** offset in mins
- .toLocaleDateString(*locale*, *options*)**
- .toLocaleTimeString(*locale*, *options*)**
- .toLocaleString(*locale*, *options*)**
- .toISOString()** return UTC date
- .toString()** return American date
- .toISOString()** return ISO8601 date
- .toJSON()** return date ready for JSON

Array()

PROPERTIES

- .length** number of elements

METHODS

- .isArray(*obj*)** check if obj is array
- .includes(*obj*, *from*)** include element?
- .indexOf(*obj*, *from*)** find elem. index
- .lastIndexOf(*obj*, *from*)** find from end
- .join(*sep*)** join elements w/separators
- .slice(*ini*, *end*)** return array portion
- .concat(*obj1*, *obj2*...)** return joined array

MODIFY SOURCE ARRAY METHODS

- .copyWithin(*pos*, *ini*, *end*)** copy elems
- .fill(*obj*, *ini*, *end*)** fill array with obj
- .reverse()** reverse array & return it
- .sort(*cf(a,b)*)** sort array (unicode sort)
- .splice(*ini*, *del*, *o1*, *o2*...)** del&add elem

ITERATION METHODS

- .entries()** iterate key/value pair array
- .keys()** iterate only keys array
- .values()** iterate only values array

CALLBACK FOR EACH METHODS

- .every(*cb(e,i,a)*, *arg*)** test until false
- .some(*cb(e,i,a)*, *arg*)** test until true
- .map(*cb(e,i,a)*, *arg*)** make array
- .filter(*cb(e,i,a)*, *arg*)** make array w/true
- .find(*cb(e,i,a)*, *arg*)** return elem w/true
- .findIndex(*cb(e,i,a)*, *arg*)** return index
- .forEach(*cb(e,i,a)*, *arg*)** exec for each
- .reduce(*cb(p,e,i,a)*, *arg*)** accumulative
- .reduceRight(*cb(p,e,i,a)*, *arg*)** from end

ADD/REMOVE METHODS

- .pop()** remove & return last element
- .push(*o1*, *o2*...)** add element & return length
- .shift()** remove & return first element
- .unshift(*o1*, *o2*...)** add element & return len

Boolean()

no own properties or methods

Function()

PROPERTIES

- .length** return number of arguments
- .name** return name of function
- .prototype** prototype object

METHODS

- .call(*newthis*, *arg1*, *arg2*...)** change *this*
- .apply(*newthis*, *arg1*)** with args array
- .bind(*newthis*, *arg1*, *arg2*...)** bound func

number

- NaN** (not-a-number)
- string**
- boolean** (true/false)
- array**

date


- regular expression**
- function**
- object**
- undefined**

only available on ECMAScript 6

- static** (ex: Math.random())
- non-static** (ex: new Date().getDate())

argument required

argument optional

 Politecnico di Torino

Web Applications - 2024/2025

 CC BY NC SA

Number()

PROPERTIES

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

METHODS

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

RegExp()

PROPERTIES

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

METHODS

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

CLASSES

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...)
\xN char with code **N** **\b** backspace
\uN char with unicode **N** **\0** NUL char

CHARACTER SETS OR ALTERNATION

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

BOUNDARIES

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

GROUPING

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

QUANTIFIERS

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

ASSERTIONS

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

String()

PROPERTIES

length string size

METHODS

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

Date()

METHODS

UTC(y, m, d, h, i, s, ms) timestamp
now() timestamp of current time
parse(str) convert str to timestamp
setTime(ts) set UNIX timestamp
getTime() return UNIX timestamp

UNIT SETTERS (ALSO: setUTC() methods)

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

UNIT GETTERS (ALSO: getUTC() methods)

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

LOCALE & TIMEZONE METHODS

getTimezoneOffset() offset in mins
toLocaleDateString(locale, options)
toLocaleTimeString(locale, options)
toLocaleString(locale, options)
toUTCString() return UTC date
toString() return American date
toTimeString() return American time
toISOString() return ISO8601 date
toJSON() return date ready for JSON

Array()

PROPERTIES

length number of elements

METHODS

isArray(obj) check if obj is array
includes(obj, from) include element?
indexOf(obj, from) find elem. index
lastIndexOf(obj, from) find from end
join(sep) join elements w/separator
slice(ini, end) return array portion
concat(obj1, obj2...) return joined array

MODIFY SOURCE ARRAY METHODS

copyWithin(pos, ini, end) copy elems
fill(obj, ini, end) fill array with obj
reverse() reverse array & return it
sort(cf(a,b)) sort array (unicode sort)
splice(ini, del, o1, o2...) del&add elem

ITERATION METHODS

entries() iterate key/value pair array
keys() iterate only keys array
values() iterate only values array

CALLBACK FOR EACH METHODS

every(cb(e,i,a), arg) test until false
some(cb(e,i,a), arg) test until true
map(cb(e,i,a), arg) make array
filter(cb(e,i,a), arg) make array w/true
find(cb(e,i,a), arg) return elem w/true
findIndex(cb(e,i,a), arg) return index
forEach(cb(e,i,a), arg) exec for each
reduce(cb(p,e,i,a), arg) accumulative
reduceRight(cb(p,e,i,a), arg) from end

ADD/REMOVE METHODS

pop() remove & return last element
push(o1, o2...) add element & return length
shift() remove & return first element
unshift(o1, o2...) add element & return len

Boolean()

no own properties or methods

Function()

PROPERTIES

length return number of arguments
name return name of function
prototype prototype object

METHODS

call(newthis, arg1, arg2...) change **this**
apply(newthis, arg1) with args array
bind(newthis, arg1, arg2...) bound func

number

NaN (not-a-number)
string
boolean (true/false)
object
array

only available on ECMAScript 6

static (ex: Math.random())
non-static (ex: new Date().getDate())
argument required
argument optional

Math

PROPERTIES

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

METHODS

abs(x) absolute value
cbrt(x) cube root
clz32(x) return leading zero bits (32)
exp(x) return e^x
expm1(x) return e^x-1
hypot(x1, x2...) length of hypotenuse
imul(a, b) signed multiply
log(x) natural logarithm (base e)
log1p(x) natural logarithm (1+x)
log10(x) base 10 logarithm
log2(x) base 2 logarithm
max(x1, x2...) return max number
min(x1, x2...) return min number
pow(base, exp) return base^{exp}
random() float random number [0,1)
sign(x) return sign of number
sqrt(x) square root of number

ROUND METHODS

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

TRIGONOMETRIC METHODS

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

JSON

METHODS

parse(str, tf(k,v)) parse string to object
stringify(obj, repl|wl, sp) convert to str

Error()

PROPERTIES

name return name of error
message return description of error

Object()

PROPERTIES

constructor return ref. to object func.

METHODS

assign(dst, src1, src2...) copy values
create(proto, prop) create obj w/prop
defineProperties(obj, prop)
defineProperty(obj, prop, desc)
freeze(obj) avoid properties changes
getOwnPropertyDescriptor(obj, prop)
getOwnPropertyNames(obj)
getOwnPropertySymbols(obj)
getPrototypeOf(obj) return prototype
is(val1, val2) check if are same value
isExtensible(obj) check if can add prop
isFrozen(obj) check if obj is frozen
isSealed(obj) check if obj is sealed
keys(obj) return only keys of object
preventExtensions(obj) avoid extend
seal(obj) prop are non-configurable
setPrototypeOf(obj, proto) change prot

INSTANCE METHODS

hasOwnProperty(prop) check if exist
isPrototypeOf(obj) test in another obj
propertyIsEnumerable(prop)
toString() return equivalent string
toLocaleString() return locale version
valueOf() return primitive value

P Promise()

METHODS

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

p Proxy()

METHODS

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

globals

METHODS

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

Set()

PROPERTIES

size return number of items

METHODS

add(item) add item to set **WS**
has(item) check if item exists **WS**
delete(item) del item & return if del **WS**
clear() remove all items from set

ITERATION METHODS

entries() iterate items
values() iterate only value of items
forEach(cb(e,i,a), arg) exec for each

Map()

PROPERTIES

size return number of elements

METHODS

set(key, value) add pair key=value **WM**
get(key) return value of key **WM**
has(key) check if key exist **WM**
delete(key) del elem. & return if ok **WM**
clear() remove all elements from map

ITERATION METHODS

entries() iterate elements
keys() iterate only keys
values() iterate only values
forEach(cb(e,i,a), arg) exec for each

Symbol()

PROPERTIES

iterator specifies default iterator
match specifies match of regexp
species specifies constructor function

METHODS

for(key) search existing symbols
keyFor(sym) return key from global reg

Generator()

METHODS

next(value) return obj w/{value,done}
return(value) return value & true done
throw(throw) 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

JS

JS

JavaScript

Cheat Sheet

JS

Programming Language of Web

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

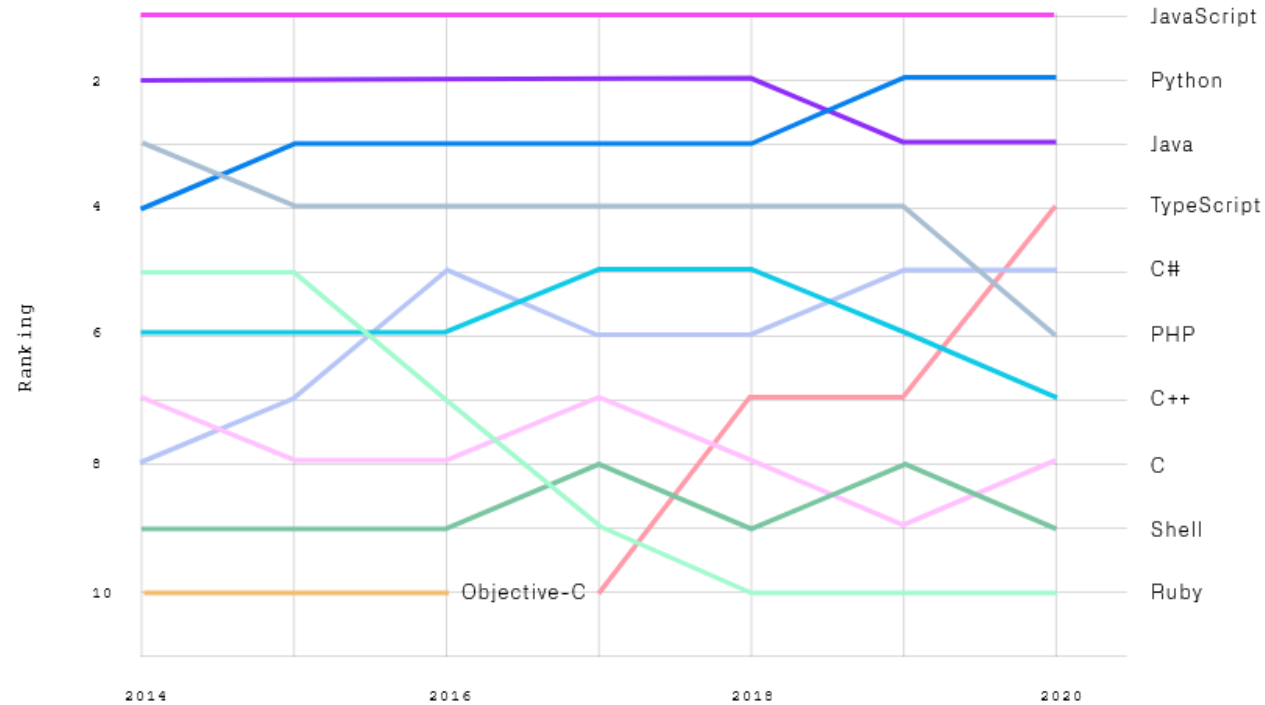
- What is JavaScript?
- History and versions
- Language structure
- Types, variables
- Expressions
- Control structures
- Arrays
- Strings

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 run 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
- ▶ **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
- ▶ **ES6 (June 2015)** Classes, Modules, Arrow Fs, Generators, Const/Let, Destructuring, Template Literals, Promise, Proxy, Symbol, Reflect
- ▶ **ES7 (June 2016)** Exponentiation operator (**) and Array Includes
- ▶ **ES8 (June 2017)** Async Fs, Shared Memory & Atomics

10
yrs

Main
target

ES9,
ES10,
...

Also: ES2015

Also: ES2016

Also: ES2017

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)



Search...

TABLE OF CONTENTS

- Introduction
- 1 Scope
- 2 Conformance
- 3 Normative References
- 4 Overview
- 5 Notational Conventions
- 6 ECMAScript Data Types and Values
- 7 Abstract Operations
- 8 Executable Code and Execution Contexts
- 9 Ordinary and Exotic Objects Behaviours
- 10 ECMAScript Language: Source Code
- 11 ECMAScript Language: Lexical Grammar
- 12 ECMAScript Language: Expressions
- 13 ECMAScript Language: Statements and Declarations
- 14 ECMAScript Language: Functions and Classes
- 15 ECMAScript Language: Scripts and Modules
- 16 Error Handling and Language Extensions
- 17 ECMAScript Standard Built-in Objects
- 18 The Global Object
- 19 Fundamental Objects
- 20 Numbers and Dates
- 21 Text Processing
- 22 Indexed Collections
- 23 Keyed Collections
- 24 Structured Data
- 25 Control Abstraction Objects
- 26 Reflection
- 27 Memory Model
- A Grammar Summary
- B Additional ECMAScript Features for Web Browsers
- C The Strict Mode of ECMAScript
- D Corrections and Clarifications in ECMAScript 2015 wit...
- E Additions and Changes That Introduce Incompatibiliti...
- F Colophon
- G Bibliography
- H Copyright & Software License

ecma
INTERNATIONAL

ECMA-262, 10th edition, June 2019
ECMAScript® 2019 Language Specification

Contributing to this Specification

This specification is developed on GitHub with the help of the ECMAScript community. There are a number of ways to contribute to the development of this specification:

GitHub Repository: <https://github.com/tc39/ecma262>
Issues: [All Issues](#), [File a New Issue](#)
Pull Requests: [All Pull Requests](#), [Create a New Pull Request](#)
Test Suite: [Test262](#)
Editors:

- Brian Terlson ([@bterlson](#))
- Bradley Farias ([@bradleymeck](#))
- Jordan Harband ([@ljharb](#))

Community:

- Mailing list: [es-discuss](#)
- IRC: [#tc39 on freenode](#)

Refer to the [colophon](#) for more information on how this document is created.

Introduction

<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
<code>FetchEvent</code>	🚧	40	Yes	44 ★	No	27	No	40	40	44	27	No	4.0
<code>FetchEvent()</code> constructor	🚧	40	Yes	44 ★	No	27	No	40	40	44	27	No	4.0
<code>client</code>	🚧 🚧 🚧	42	?	44	No	27	No	42	44	No	?	No	4.0
<code>clientId</code>	🚧	49	?	45 ★	No	36	No	49	49	45	36	No	5.0
<code>isReload</code>	🚧	45	17	44 ★	No	32	No	45	45	44	32	No	5.0
<code>navigationPreload</code>	🚧	59	?	?	No	46	No	59	59	?	43	No	7.0
<code>preloadResponse</code>	🚧	59	18	?	No	46	No	59	59	?	43	No	7.0
<code>replacesClientId</code>		No	18	65	No	No	No	No	No	65	No	No	No
<code>request</code>	🚧	Yes	?	44	No	Yes	No	Yes	Yes	?	Yes	No	Yes
<code>respondWith</code>	🚧	42 ★	?	59 ★	No	29	No	42 ★	42 ★	?	29	No	4.0
<code>resultingClientId</code>		72	18	65	No	60	No	72	72	65	50	No	No
<code>targetClientId</code>		?	?	?	No	?	No	?	?	?	?	No	?

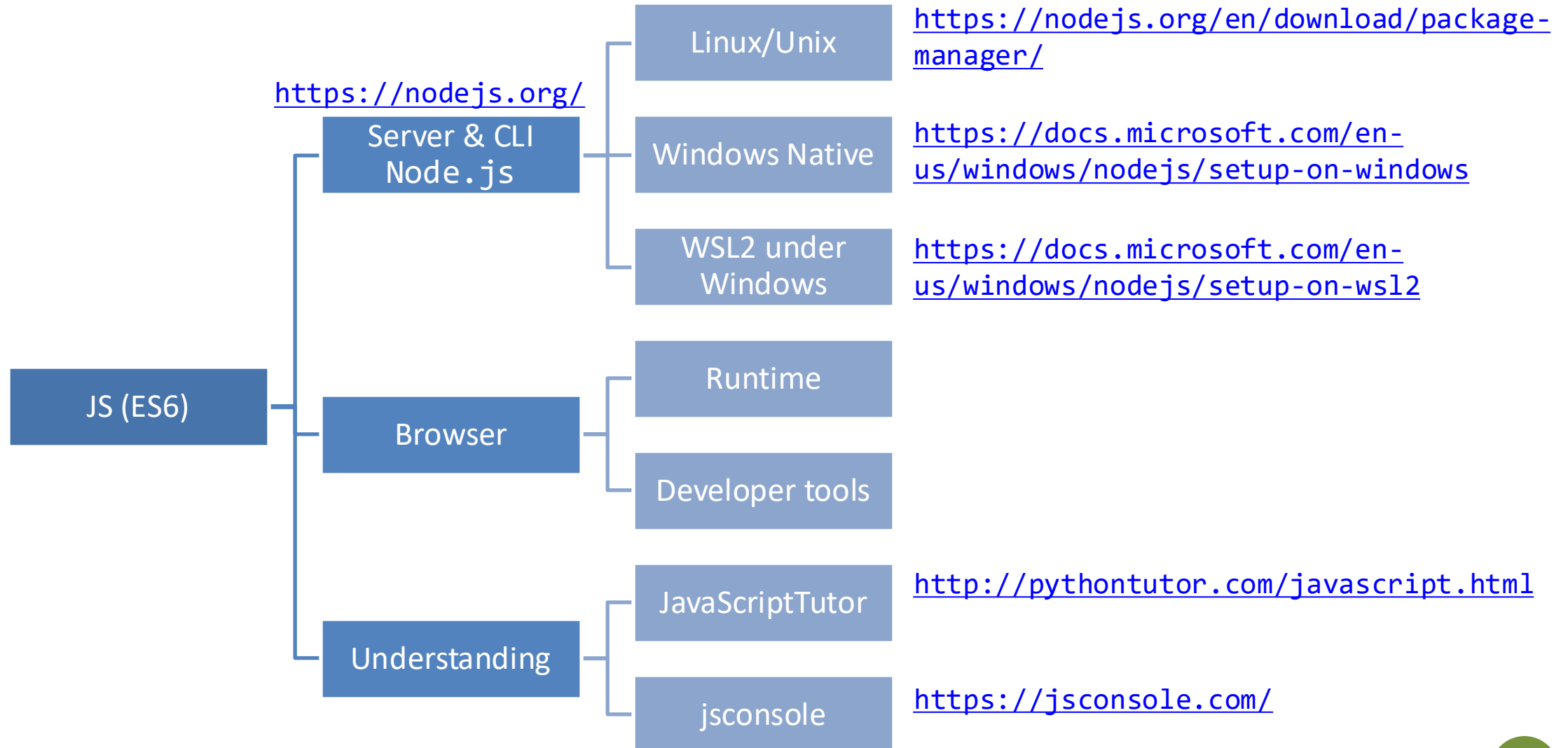
What are we missing?

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

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

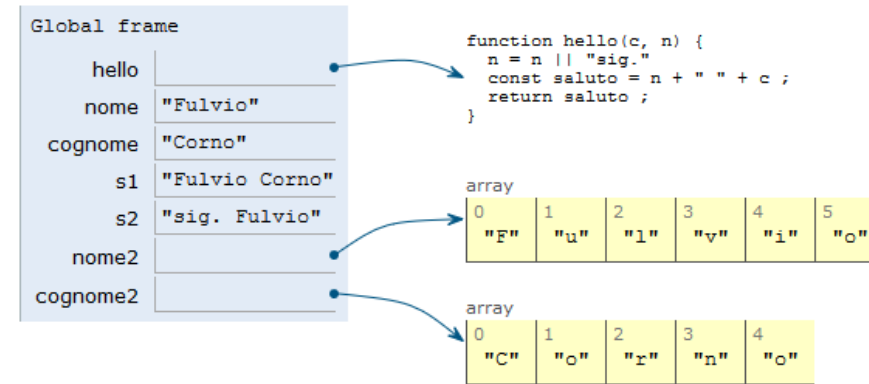
→ next line to execute

<< First < Prev Next > Last >>

Done running (16 steps)

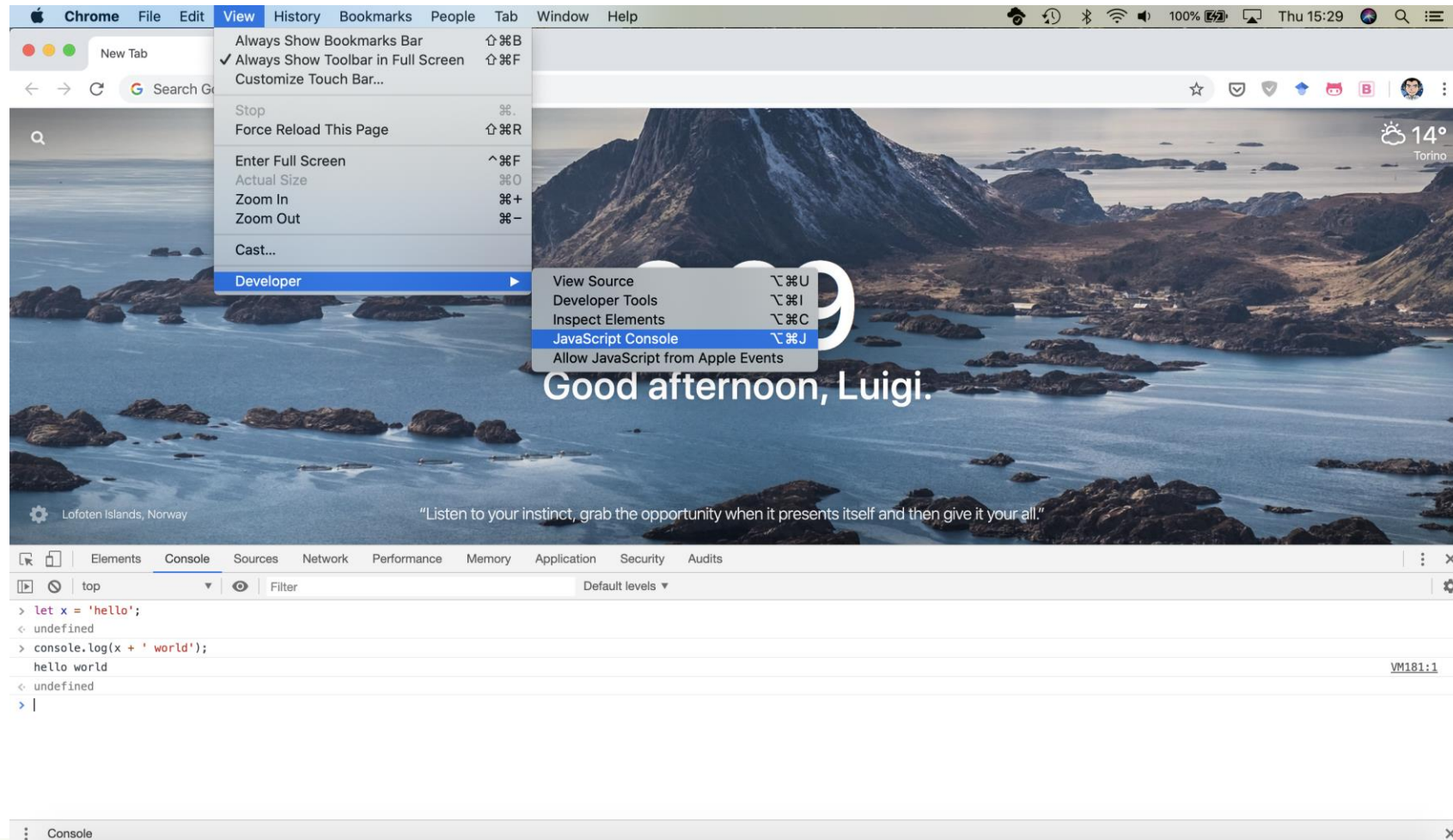
Frames

Objects



<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

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

- 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 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)
 - ...

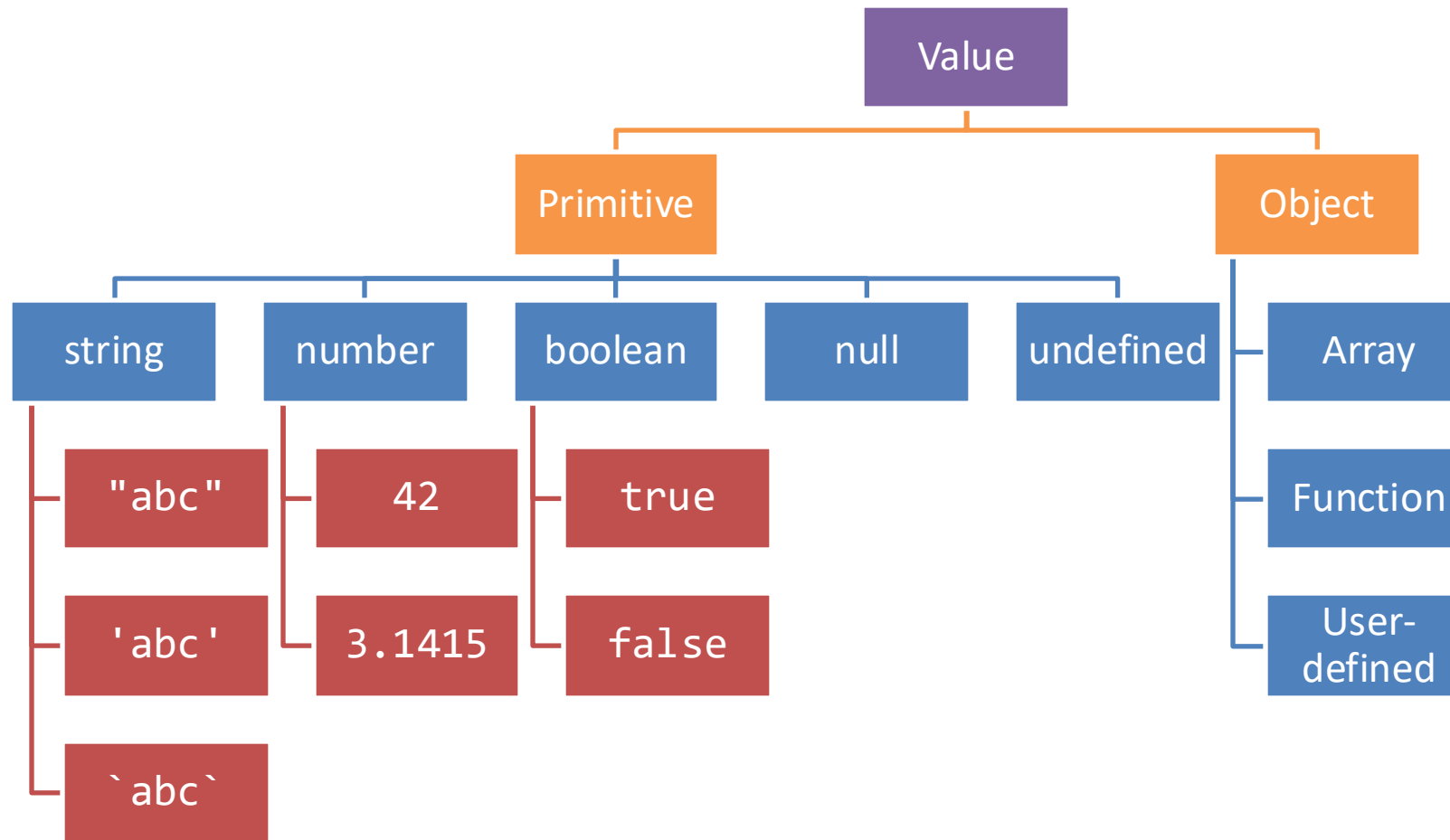


JavaScript – The language of the Web

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
let	Yes	No	Enclosing block {...}	No	<i>Preferred</i>
const	No [§]	No	Enclosing block {...}	No	<i>Preferred</i>
var	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

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

```
"use strict" ;

let a = 1 ;
const b = 2 ;
let c = true ;

{ // creating a new scope...
  let a = 5 ;
  console.log(a) ;
}

console.log(a) ;
```

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_Precedence#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 🔥	<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 }
```

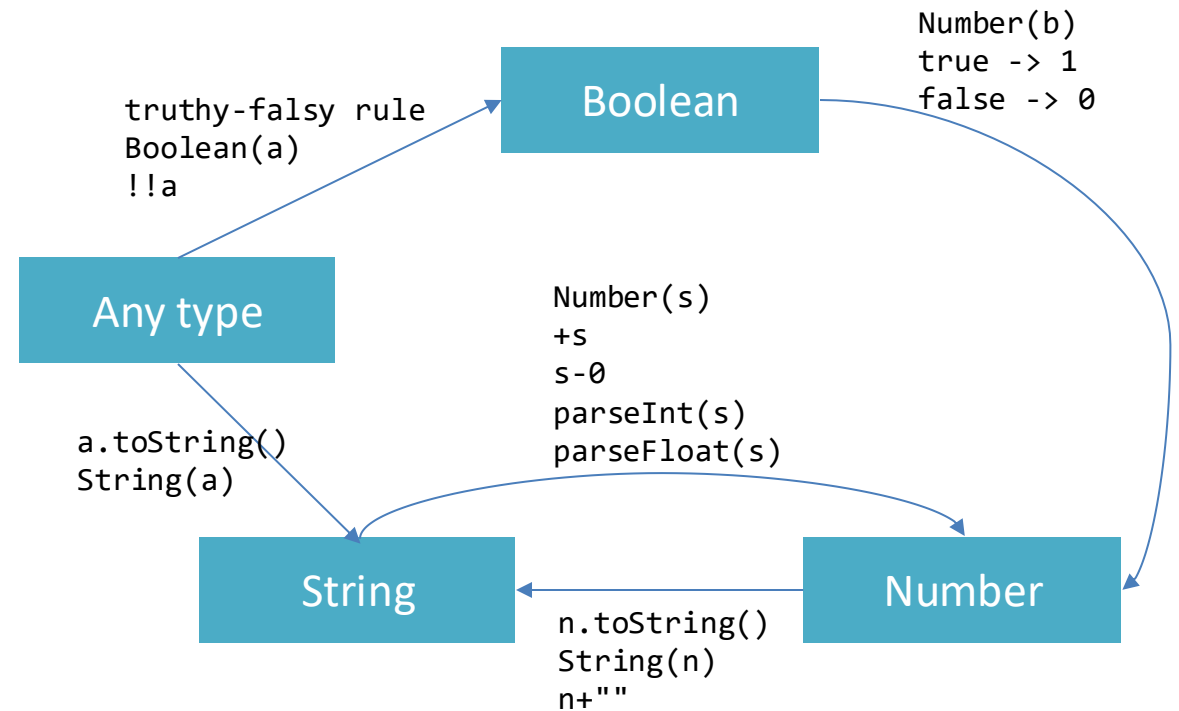
```
> b={x:1}  
{ x: 1 }
```

```
> 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



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 (+)

Decrement (--)

Division (/)

Exponentiation (**)

Increment (++)

Multiplication (*)

Remainder (%)

Subtraction (-)

Unary negation (-)

Unary plus (+)

Logical AND (&&)

Logical OR (||)

Logical NOT (!)

Nullish coalescing
operator (??)

Conditional operator (c ?
t : f)

typeof

Useful idiom:
a || b
if a then a else b
(a, with default b)

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;  
}
```

May also be a string

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


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 ;  
}
```

- 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

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

- 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 ;  
}
```

```
throw object ;
```

Exception object

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

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

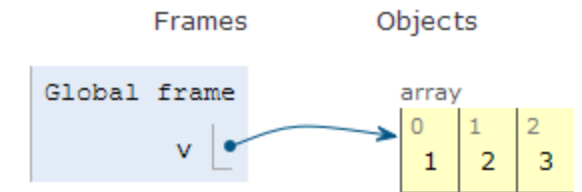
```
let v = [] ;
```

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

Do not access elements outside range

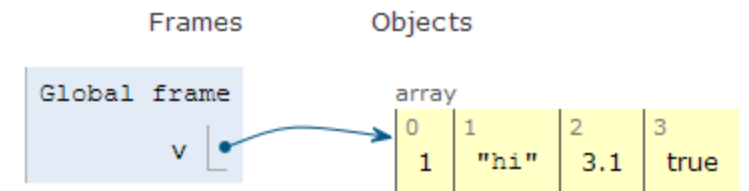
```
let v = [1, 2, 3] ;
```

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



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

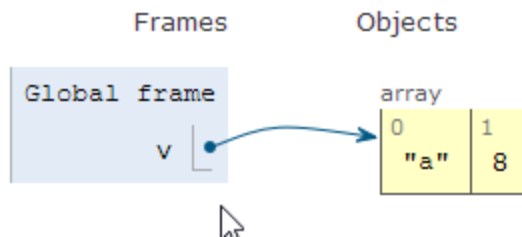
```
let v = Array.of(1, "hi", 3.1, true) ;
```



Adding elements

`.length` adjusts automatically

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

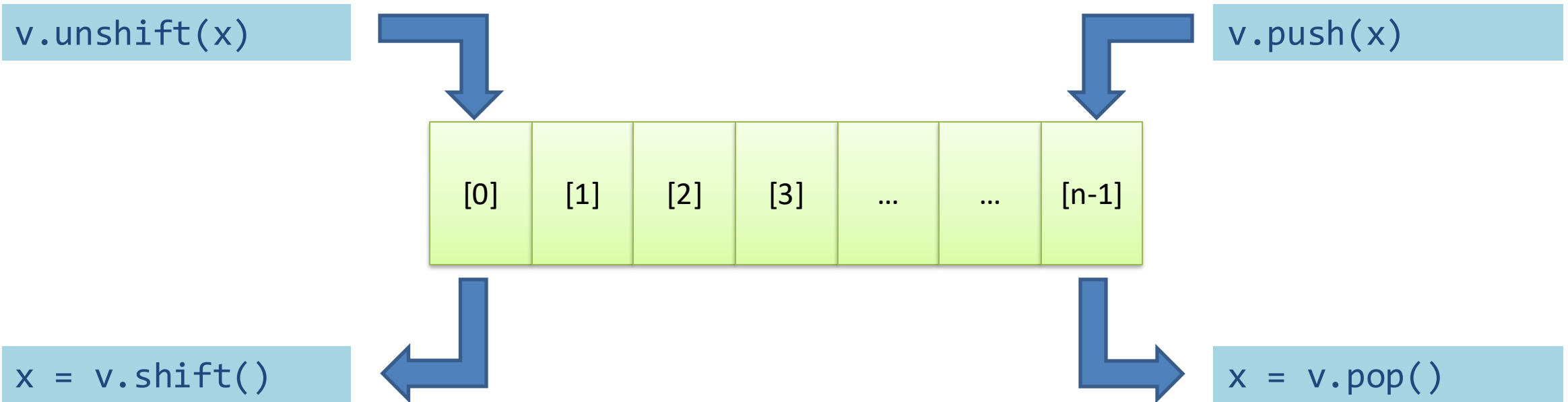


```
let v = [] ;  
v.push("a") ;  
v.push(8) ;  
v.length // 2
```

`.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 ;
```

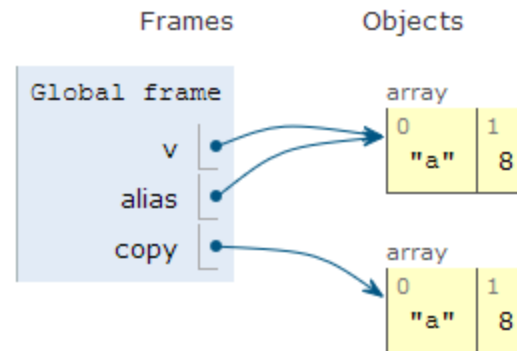
```
> console.log(v);  
[ 'a', 5 ]  
undefined  
> console.log(alias);  
[ 'a', 5 ]  
undefined
```

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

Functional style – later

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 sign 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/>

