

# JavaScript (basics)

“The” language of the Web

Alberto Monge Roffarello

JS
JavaScript
JS

Cheat Sheet

Programming Language of Web

<h3 style="margin: 0;">Number()</h3> <p><b>PROPERTIES</b></p> <ul style="list-style-type: none"> <li><b>Number.POSITIVE_INFINITY</b> +∞ equivalent</li> <li><b>Number.NEGATIVE_INFINITY</b> -∞ equivalent</li> <li><b>Number.MAX_VALUE</b> largest positive value</li> <li><b>Number.MIN_VALUE</b> smallest positive value</li> <li><b>Number.EPSILON</b> diff between 1 &amp; smallest &gt;1</li> <li><b>Number.NaN</b> not-a-number value</li> </ul> <p><b>METHODS</b></p> <ul style="list-style-type: none"> <li><b>Number.prototype.toExponential(dec)</b> exp. notation</li> <li><b>Number.prototype.toFixed(dec)</b> fixed-point notation</li> <li><b>Number.prototype.toPrecision(p)</b> change precision</li> <li><b>Number.prototype.isFinite(n)</b> check if number is finite</li> <li><b>Number.prototype.isInteger(n)</b> check if number is int.</li> <li><b>Number.prototype.isNaN(n)</b> check if number is NaN</li> <li><b>Number.parseInt(s, radix)</b> string to integer</li> <li><b>Number.parseFloat(s, radix)</b> string to float</li> </ul> <h3 style="margin: 0;">RegExp()</h3> <p><b>PROPERTIES</b></p> <ul style="list-style-type: none"> <li><b>RegExp.prototype.lastIndex</b> index to start global regexp</li> <li><b>RegExp.prototype.flags</b> active flags of current regexp</li> <li><b>RegExp.prototype.global</b> flag g (search all matches)</li> <li><b>RegExp.prototype.ignoreCase</b> flag i (match lower/upper)</li> <li><b>RegExp.prototype.multiline</b> flag m (match multiple lines)</li> <li><b>RegExp.prototype.sticky</b> flag y (search from lastIndex)</li> <li><b>RegExp.prototype.unicode</b> flag u (enable unicode feat.)</li> <li><b>RegExp.prototype.source</b> current regexp (w/o slashes)</li> </ul> <p><b>METHODS</b></p> <ul style="list-style-type: none"> <li><b>RegExp.prototype.exec(str)</b> exec search for a match</li> <li><b>RegExp.prototype.test(str)</b> check if regexp match w/str</li> </ul> <p><b>CLASSES</b></p> <ul style="list-style-type: none"> <li><b>.</b> any character</li> <li><b>\d</b> digit [0-9]</li> <li><b>\D</b> no digit [^0-9]</li> <li><b>\w</b> any alphanumeric char [A-Za-z0-9_]</li> <li><b>\W</b> no alphanumeric char [^A-Za-z0-9_]</li> <li><b>\s</b> any space char (space, tab, enter...)</li> <li><b>\S</b> no space char (space, tab, enter...)</li> <li><b>\xN</b> char with code N</li> <li><b>\uN</b> char with unicode N</li> </ul> <p><b>CHARACTER SETS OR ALTERNATION</b></p> <ul style="list-style-type: none"> <li><b>[abc]</b> match any character set</li> <li><b>[^abc]</b> match any char. set not enclosed</li> <li><b>a b</b> match a or b</li> </ul> <p><b>BOUNDARIES</b></p> <ul style="list-style-type: none"> <li><b>^</b> begin of input</li> <li><b>\$</b> end of input</li> <li><b>\b</b> zero-width word boundary</li> <li><b>\B</b> zero-width non-word boundary</li> </ul> <p><b>GROUPING</b></p> <ul style="list-style-type: none"> <li><b>(x)</b> capture group</li> <li><b>(?:x)</b> no capture group</li> <li><b>n</b> reference to group n captured</li> </ul> <p><b>QUANTIFIERS</b></p> <ul style="list-style-type: none"> <li><b>x*</b> preceding x 0 or more times {0,}</li> <li><b>x+</b> preceding x 1 or more times {1,}</li> <li><b>x?</b> preceding x 0 or 1 times {0,1}</li> <li><b>x{n}</b> n occurrences of x</li> <li><b>x{n,}</b> at least n occurrences of x</li> <li><b>x{n,m}</b> between n &amp; m occurrences of x</li> </ul> <p><b>ASSERTIONS</b></p> <ul style="list-style-type: none"> <li><b>x(?:=y)</b> x (only if x is followed by y)</li> <li><b>x(?:!y)</b> x (only if x is not followed by y)</li> </ul>	<h3 style="margin: 0;">String()</h3> <p><b>PROPERTIES</b></p> <ul style="list-style-type: none"> <li><b>String.prototype.length</b> string size</li> </ul> <p><b>METHODS</b></p> <ul style="list-style-type: none"> <li><b>String.prototype.charAt(index)</b> char at position</li> <li><b>String.prototype.charCodeAt(index)</b> unicode at pos.</li> <li><b>String.prototype.fromCharCode(n1, n2,...)</b> code to char</li> <li><b>String.prototype.concat(str1, str2,...)</b> combine text</li> <li><b>String.prototype.startsWith(str, size)</b> check beginning</li> <li><b>String.prototype.endsWith(str, size)</b> check ending</li> <li><b>String.prototype.includes(str, from)</b> include substring?</li> <li><b>String.prototype.indexOf(str, from)</b> find substr index</li> <li><b>String.prototype.lastIndexOf(str, from)</b> find from end</li> <li><b>String.prototype.search(regex)</b> search &amp; return index</li> <li><b>String.prototype.localeCompare(str, locale, options)</b></li> <li><b>String.prototype.match(regex)</b> matches against string</li> <li><b>String.prototype.repeat(n)</b> repeat string n times</li> <li><b>String.prototype.replace(str regex, newstr func)</b></li> <li><b>String.prototype.slice(ini, end)</b> str between ini/end</li> <li><b>String.prototype.substr(ini, len)</b> substr of len length</li> <li><b>String.prototype.substring(ini, end)</b> substr fragment</li> <li><b>String.prototype.split(sep regex, limit)</b> divide string</li> <li><b>String.prototype.toLowerCase()</b> string to lowercase</li> <li><b>String.prototype.toUpperCase()</b> string to uppercase</li> <li><b>String.prototype.trim()</b> remove space from begin/end</li> <li><b>String.prototype.raw()</b> template strings with \${vars}</li> </ul> <h3 style="margin: 0;">Date()</h3> <p><b>METHODS</b></p> <ul style="list-style-type: none"> <li><b>Date.prototype.UTC(y, m, d, h, i, s, ms)</b> timestamp</li> <li><b>Date.prototype.now()</b> timestamp of current time</li> <li><b>Date.prototype.parse(str)</b> convert str to timestamp</li> <li><b>Date.prototype.setTime(ts)</b> set UNIX timestamp</li> <li><b>Date.prototype.getTime()</b> return UNIX timestamp</li> </ul> <p><b>UNIT SETTERS (ALSO .setUTC() methods)</b></p> <ul style="list-style-type: none"> <li><b>Date.prototype.setFullYear(y, m, d)</b> set year (yyyy)</li> <li><b>Date.prototype.setMonth(m, d)</b> set month (0-11)</li> <li><b>Date.prototype.setDate(d)</b> set day (1-31)</li> <li><b>Date.prototype.setHours(h, m, s, ms)</b> set hour (0-23)</li> <li><b>Date.prototype.setMinutes(m, s, ms)</b> set min (0-59)</li> <li><b>Date.prototype.setSeconds(s, ms)</b> set sec (0-59)</li> <li><b>Date.prototype.setMilliseconds(ms)</b> set ms (0-999)</li> </ul> <p><b>UNIT GETTERS (ALSO .getUTC() methods)</b></p> <ul style="list-style-type: none"> <li><b>Date.prototype.getDate()</b> return day (1-31)</li> <li><b>Date.prototype.getDay()</b> return day of week (0-6)</li> <li><b>Date.prototype.getMonth()</b> return month (0-11)</li> <li><b>Date.prototype.getFullYear()</b> return year (yyyy)</li> <li><b>Date.prototype.getHours()</b> return hour (0-23)</li> <li><b>Date.prototype.getMinutes()</b> return minutes (0-59)</li> <li><b>Date.prototype.getSeconds()</b> return seconds (0-59)</li> <li><b>Date.prototype.getMilliseconds()</b> return ms (0-999)</li> </ul> <p><b>LOCALE &amp; TIMEZONE METHODS</b></p> <ul style="list-style-type: none"> <li><b>Date.prototype.getTimezoneOffset()</b> offset in mins</li> <li><b>Date.prototype.toLocaleDateString(locale, options)</b></li> <li><b>Date.prototype.toLocaleTimeString(locale, options)</b></li> <li><b>Date.prototype.toLocaleString(locale, options)</b></li> <li><b>Date.prototype.toUTCString()</b> return UTC date</li> <li><b>Date.prototype.toString()</b> return American date</li> <li><b>Date.prototype.toTimeString()</b> return American time</li> <li><b>Date.prototype.toISOString()</b> return ISO8601 date</li> <li><b>Date.prototype.toJSON()</b> return date ready for JSON</li> </ul>	<h3 style="margin: 0;">Array()</h3> <p><b>PROPERTIES</b></p> <ul style="list-style-type: none"> <li><b>Array.prototype.length</b> number of elements</li> </ul> <p><b>METHODS</b></p> <ul style="list-style-type: none"> <li><b>Array.prototype.isArray(obj)</b> check if obj is array</li> <li><b>Array.prototype.includes(obj, from)</b> include element?</li> <li><b>Array.prototype.indexOf(obj, from)</b> find elem. index</li> <li><b>Array.prototype.lastIndexOf(obj, from)</b> find from end</li> <li><b>Array.prototype.join(sep)</b> join elements w/separator</li> <li><b>Array.prototype.slice(ini, end)</b> return array portion</li> <li><b>Array.prototype.concat(obj1, obj2,...)</b> return joined array</li> </ul> <p><b>MODIFY SOURCE ARRAY METHODS</b></p> <ul style="list-style-type: none"> <li><b>Array.prototype.copyWithin(pos, ini, end)</b> copy elems</li> <li><b>Array.prototype.fill(obj, ini, end)</b> fill array with obj</li> <li><b>Array.prototype.reverse()</b> reverse array &amp; return it</li> <li><b>Array.prototype.sort(cf(a,b))</b> sort array (unicode sort)</li> <li><b>Array.prototype.splice(ini, del, o1, o2,...)</b> del&amp;add elem</li> </ul> <p><b>ITERATION METHODS</b></p> <ul style="list-style-type: none"> <li><b>Array.prototype.entries()</b> iterate key/value pair array</li> <li><b>Array.prototype.keys()</b> iterate only keys array</li> <li><b>Array.prototype.values()</b> iterate only values array</li> </ul> <p><b>CALLBACK FOR EACH METHODS</b></p> <ul style="list-style-type: none"> <li><b>Array.prototype.every(cb(e,i,a), arg)</b> test until false</li> <li><b>Array.prototype.some(cb(e,i,a), arg)</b> test until true</li> <li><b>Array.prototype.map(cb(e,i,a), arg)</b> make array</li> <li><b>Array.prototype.filter(cb(e,i,a), arg)</b> make array w/true</li> <li><b>Array.prototype.find(cb(e,i,a), arg)</b> return elem w/true</li> <li><b>Array.prototype.findIndex(cb(e,i,a), arg)</b> return index</li> <li><b>Array.prototype.forEach(cb(e,i,a), arg)</b> exec for each</li> <li><b>Array.prototype.reduce(cb(p,e,i,a), arg)</b> accumulative</li> <li><b>Array.prototype.reduceRight(cb(p,e,i,a), arg)</b> from end</li> </ul> <p><b>ADD/REMOVE METHODS</b></p> <ul style="list-style-type: none"> <li><b>Array.prototype.pop()</b> remove &amp; return last element</li> <li><b>Array.prototype.push(o1, o2,...)</b> add element &amp; return length</li> <li><b>Array.prototype.shift()</b> remove &amp; return first element</li> <li><b>Array.prototype.unshift(o1, o2,...)</b> add element &amp; return len</li> </ul> <h3 style="margin: 0;">Boolean()</h3> <p><b>PROPERTIES</b></p> <ul style="list-style-type: none"> <li><b>Boolean.prototype</b> no own properties or methods</li> </ul> <h3 style="margin: 0;">Function()</h3> <p><b>PROPERTIES</b></p> <ul style="list-style-type: none"> <li><b>Function.prototype.length</b> return number of arguments</li> <li><b>Function.prototype.name</b> return name of function</li> <li><b>Function.prototype.prototype</b> prototype object</li> </ul> <p><b>METHODS</b></p> <ul style="list-style-type: none"> <li><b>Function.prototype.call(newthis, arg1, arg2,...)</b> change this</li> <li><b>Function.prototype.apply(newthis, arg1)</b> with args array</li> <li><b>Function.prototype.bind(newthis, arg1, arg2,...)</b> bound func</li> </ul> <p><b>ONLY AVAILABLE ON ECMAScript 6</b></p> <ul style="list-style-type: none"> <li><b>Function.prototype.static</b> (ex: Math.random())</li> <li><b>Function.prototype.non-static</b> (ex: new Date().getDate())</li> <li><b>Function.prototype.argument</b> required</li> <li><b>Function.prototype.argument</b> optional</li> </ul>
--	---	---



**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
- parseInt(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

**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)

**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

**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)

**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(fill, 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

**Boolean()**

**PROPERTIES**

- no own properties or methods

**Array()**

**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(p,e,i,a), arg)** exec for each
- reduce(cb(p,e,i,a), arg)** accumulative
- reduceRight(cb(p,e,i,a), arg)** from end

**Array()**

**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

**LOCALS & TIMEZONE METHODS**

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

**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
- PI** 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<sup>x</sup>
- expm1(x)** return e<sup>x</sup>-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<sup>exp</sup>
- random()** float random number [0,1)
- sign(x)** return sign of number
- sqrt(x)** square root of number

**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

**Promise()**

**METHODS**

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

**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)**

**JSON**

**METHODS**

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

**Error()**

**PROPERTIES**

- name** return name of error
- message** return description of error

**Object()**

**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

**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

**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)** = %3D
- decodeURIComponent(URI)** %3D =

**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

**Set()**

**PROPERTIES**

- size** return number of items

**METHODS**

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

**ITERATION METHODS**

- entries()** iterate items
- values()** iterate only value of items

**CALLBACK FOR EACH METHODS**

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

**Map()**

**PROPERTIES**

- size** return number of elements

**METHODS**

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

**ITERATION METHODS**

- entries()** iterate elements
- keys()** iterate only keys
- values()** iterate only values

**CALLBACK FOR EACH METHODS**

- 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

**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)** = %3D
- decodeURIComponent(URI)** %3D =

# Goal

- Introducing JavaScript
- Understand the specific semantics and some programming patterns
- Objects, Functions, and Dates
- Updated to ES6 (2015) language features
- Supported by client-side (browsers) run-time environment

# Outline

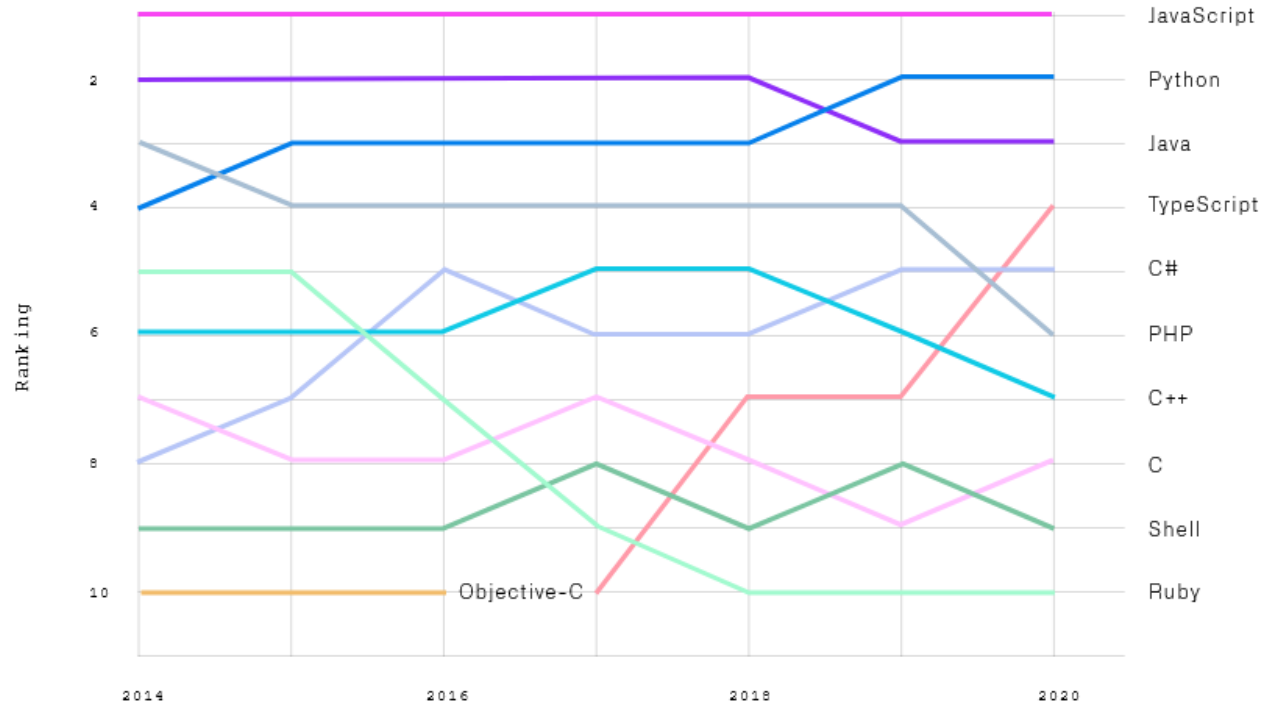
- What is JavaScript?
- History and versions
- Language structure
- Types, variables
- Expressions
- Control structures
- Arrays
- Strings
- Functions

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

Also: ES2015

Also: ES2016

Also: ES2017

10 yrs

Main target

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)



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, 10<sup>th</sup> 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	?

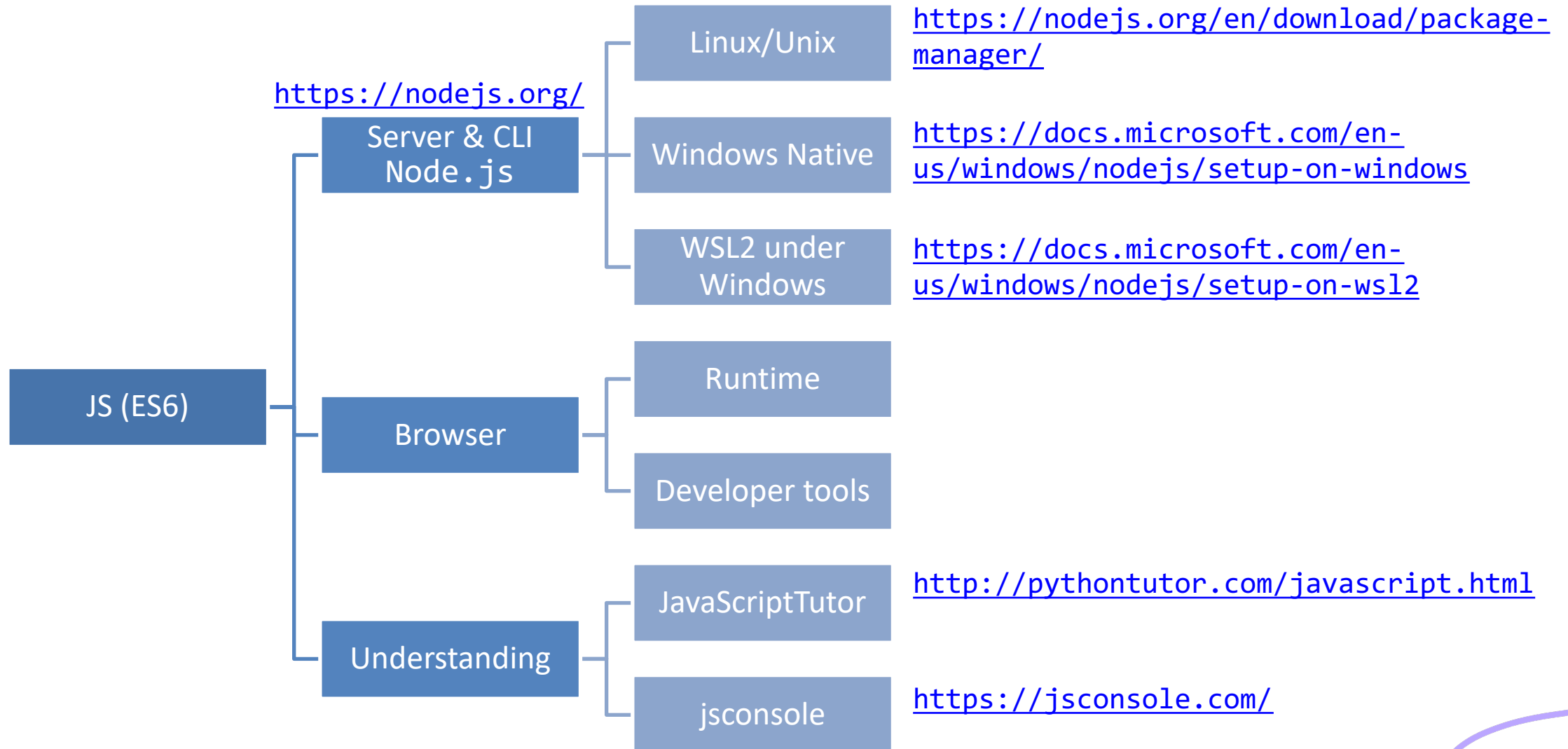
What are we missing?

- Full support
- 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

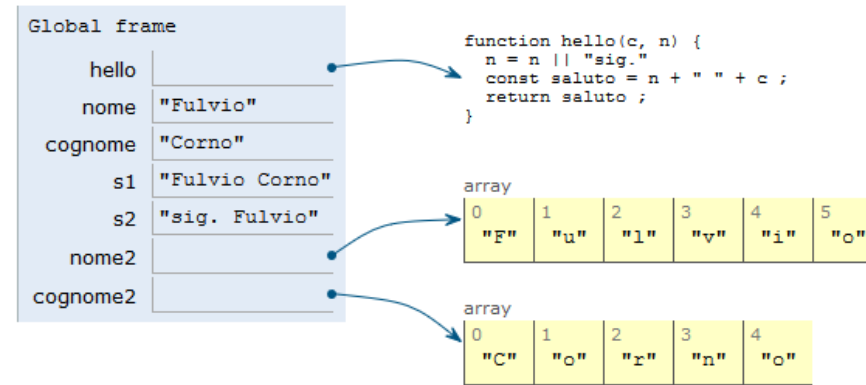


<< First < Prev Next > Last >>

Done running (16 steps)

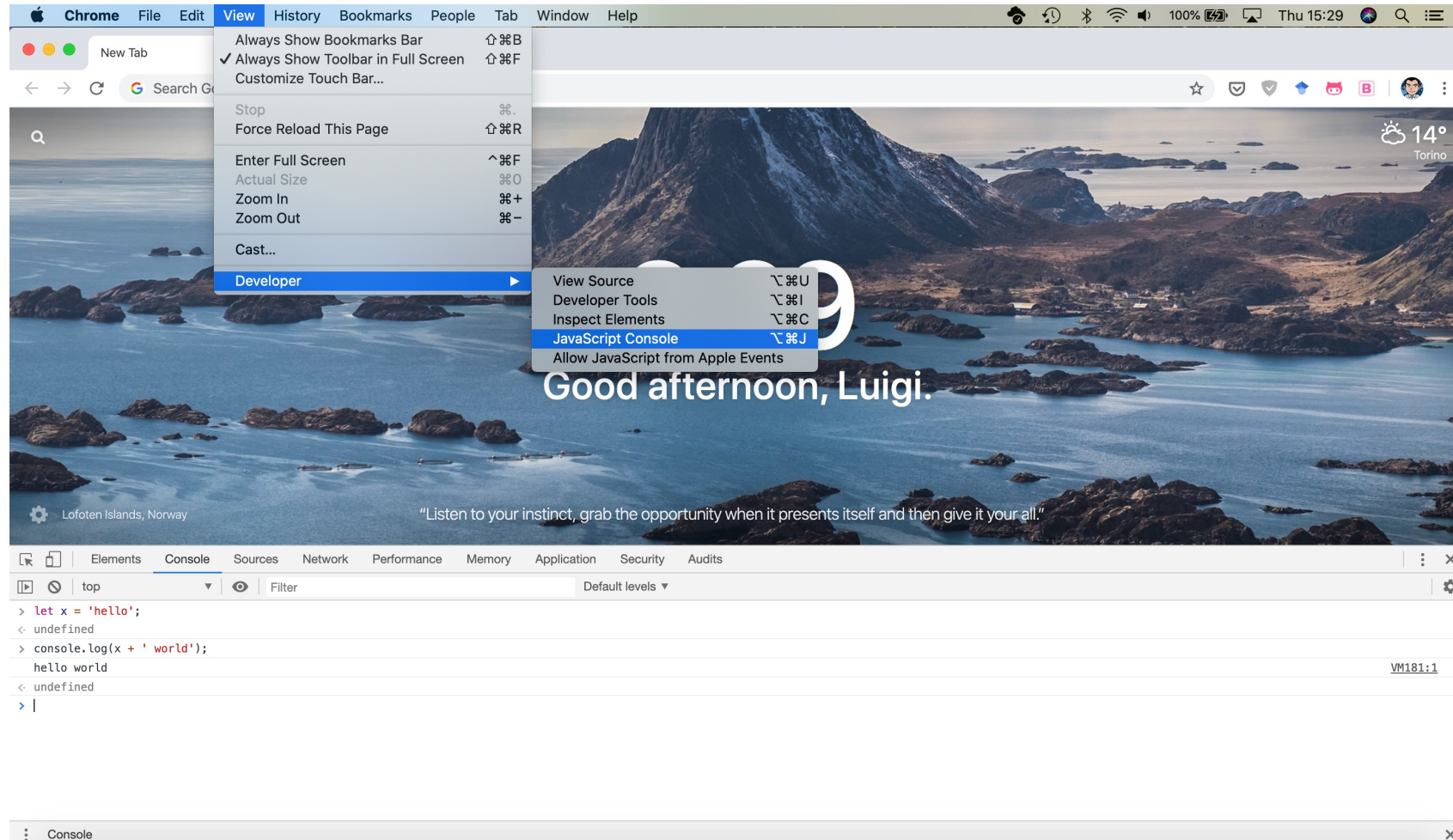
Frames

Objects





# 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)
    - ...



Mozilla Developer Network: The Script element  
<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script>

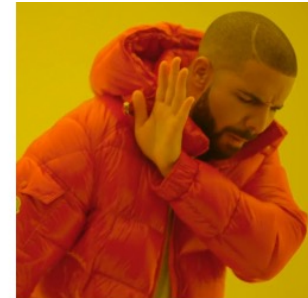
JS in the browser

# LOADING JS IN THE BROWSER

# Loading JavaScript In The Browser

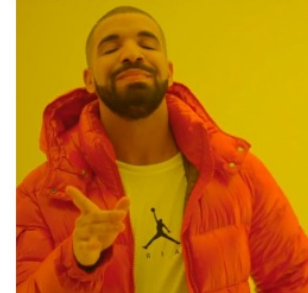
- JS must be loaded from an HTML document
- `<script>` tag
  - Inline

```
...  
<script>  
alert('Hello');  
</script>  
...
```



- External

```
...  
<script src="file.js"></script>  
...
```



<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script>



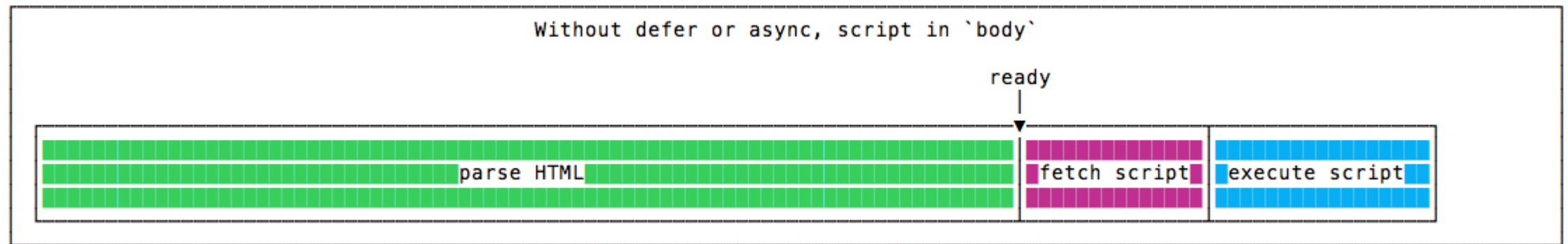
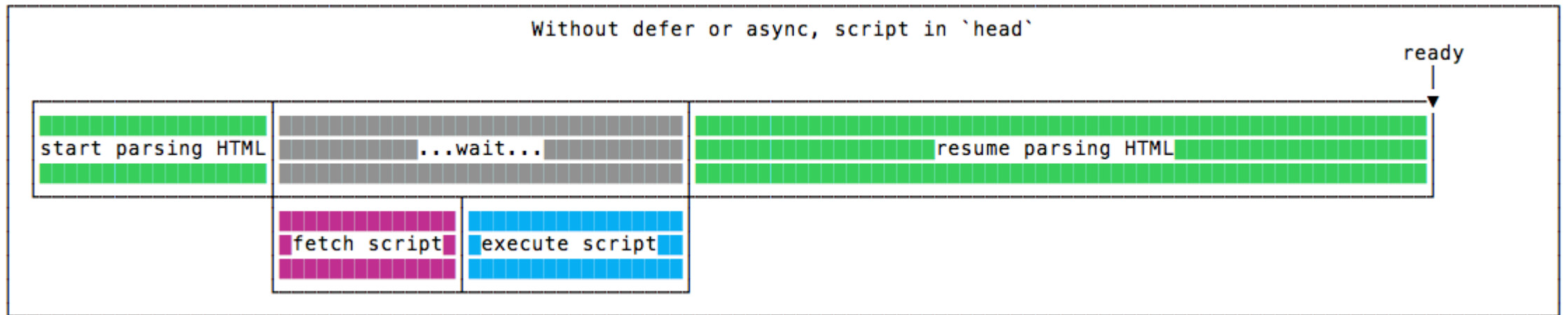
# Where To Insert The <script> Tag?

- In the <head> section
  - “clean” / “textbook” solution
  - Very **inefficient**: HTML processing is stopped until the script is loaded and executed
  - Quite **inconvenient**: the script executes when the document’s DOM does not exist yet
  - *But*: see after!
- **Just before the end** of the document
  - More efficient than the “textbook” solution

```
<!DOCTYPE html>
<html>
  <head>
    <title>Loading a script</title>
    <script src="script.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Loading a script</title>
  </head>
  <body>
    ...
    <script src="script.js"></script>
  </body>
</html>
```

# Performance Comparison In Loading JS

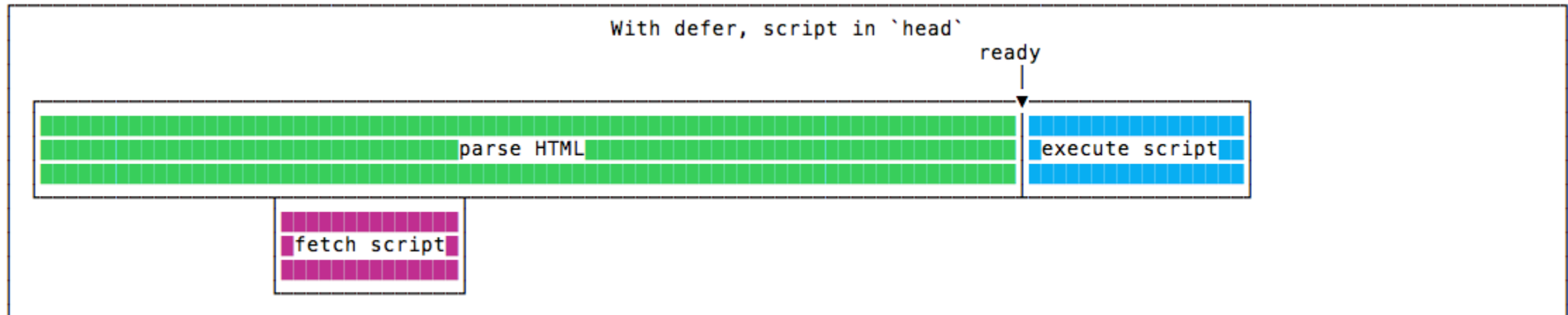
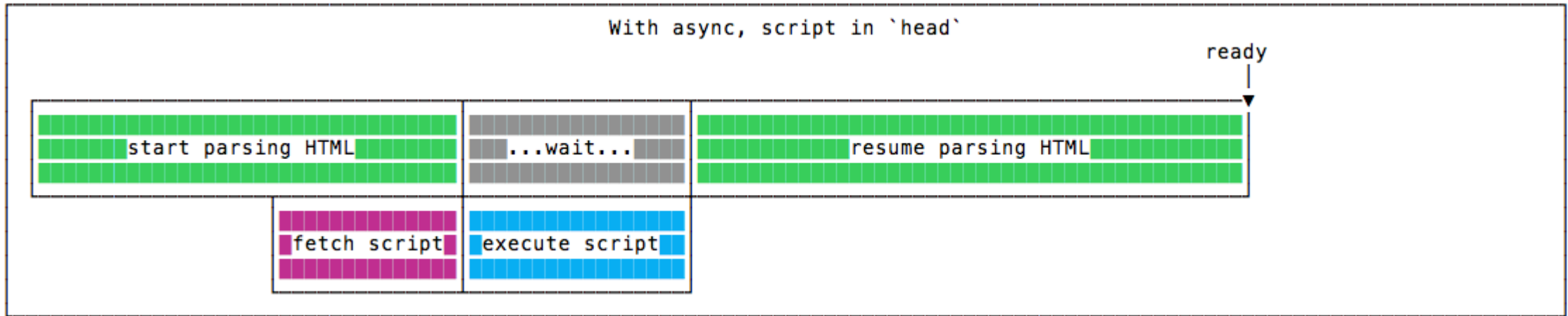


<https://flaviocopes.com/javascript-async-defer/>

# New Loading Attributes

- `<script async src="script.js"></script>`
  - Script will be fetched in parallel to parsing and evaluated as soon as it is available
  - Not immediately executed, not blocking
- `<script defer src="script.js"></script>` (*preferred*)
  - Indicate to a browser that the script is meant to be executed after the document has been parsed, but before firing DOMContentLoaded (that will wait until the script is finished)
  - Guaranteed to execute in the order they are loaded
- Both should be placed in the `<head>` of the document

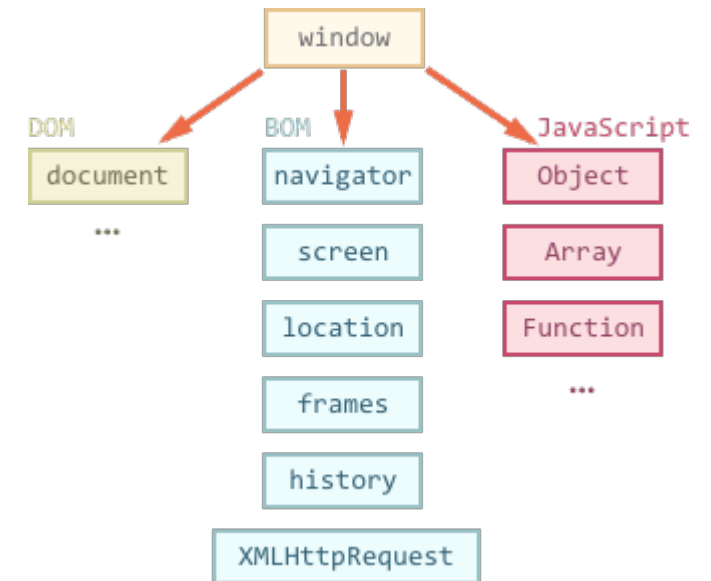
# defer vs. async



<https://flaviocopes.com/javascript-async-defer/>

# Where Does The Code Run?

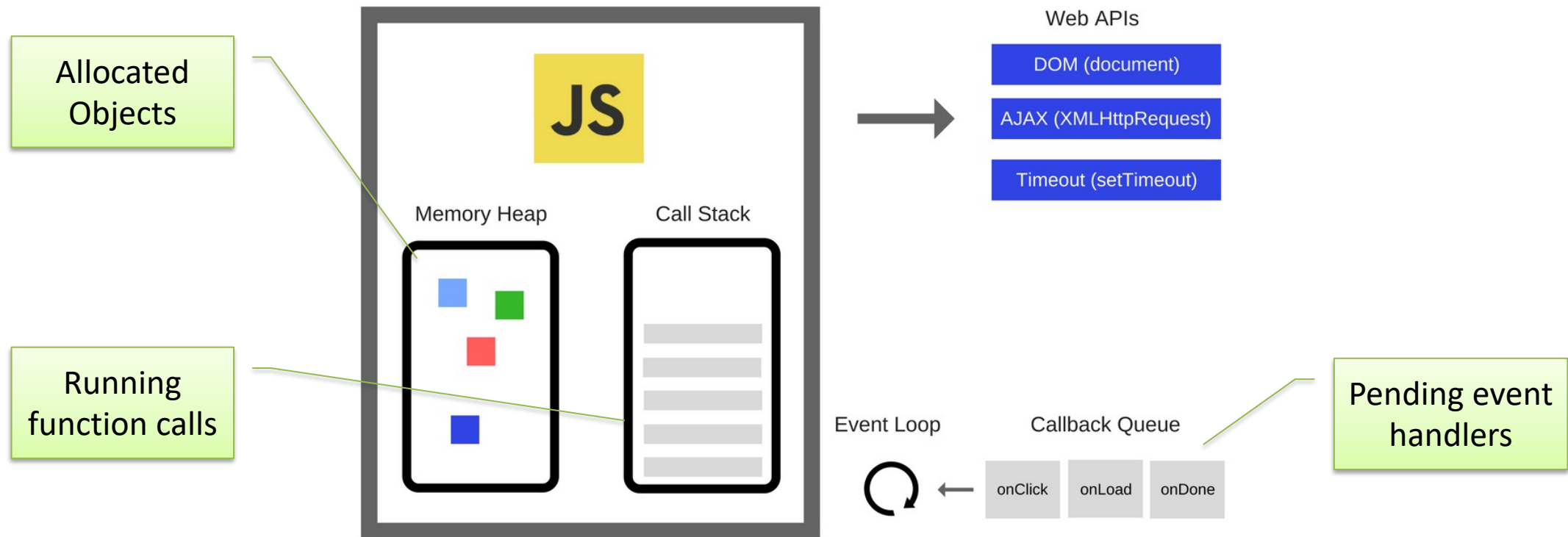
- Loaded and run in the browser *sandbox*
- Attached to a *global context*: the `window` object
- May access only a limited set of APIs
  - JS Standard Library
  - Browser objects (**BOM**)
  - Document objects (**DOM**)
- Multiple `<script>`s are independent
  - They all access the same global scope
  - To have structured collaboration, *modules* are needed



# Events and Event Loop

- Most phases of processing and interaction with a web document will generate *Asynchronous Events* (100's of different types)
- Generated events may be handled by:
  - *Pre-defined* behaviors (by the browser)
  - *User-defined event handlers* (in your JS)
  - Or just *ignored*, if no event handler is defined
- But JavaScript is *single-threaded*
  - Event handling is *synchronous* and is based on an *event loop*
  - Event handlers are queued on a *Message Queue*
  - The Message Queue is polled when the main thread is idle

# Execution Environment



# Event Loop

- During code execution you may
  - Call **functions** → the function call is pushed to the **call stack**
  - Schedule **events** → the call to the event handler is put in the **Message Queue**
    - Events may be scheduled also by external events (user actions, I/O, network, timers, ...)
- At any step, the JS interpreter:
  - If the **call stack** is not empty, pop the top of the **call stack** and executes it
  - If the call stack is **empty**, pick the head of the **Message Queue** and executes it
- A function call / event handler is **never** interrupted
  - Avoid blocking code!

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>

<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#what-is-the-event-loop>



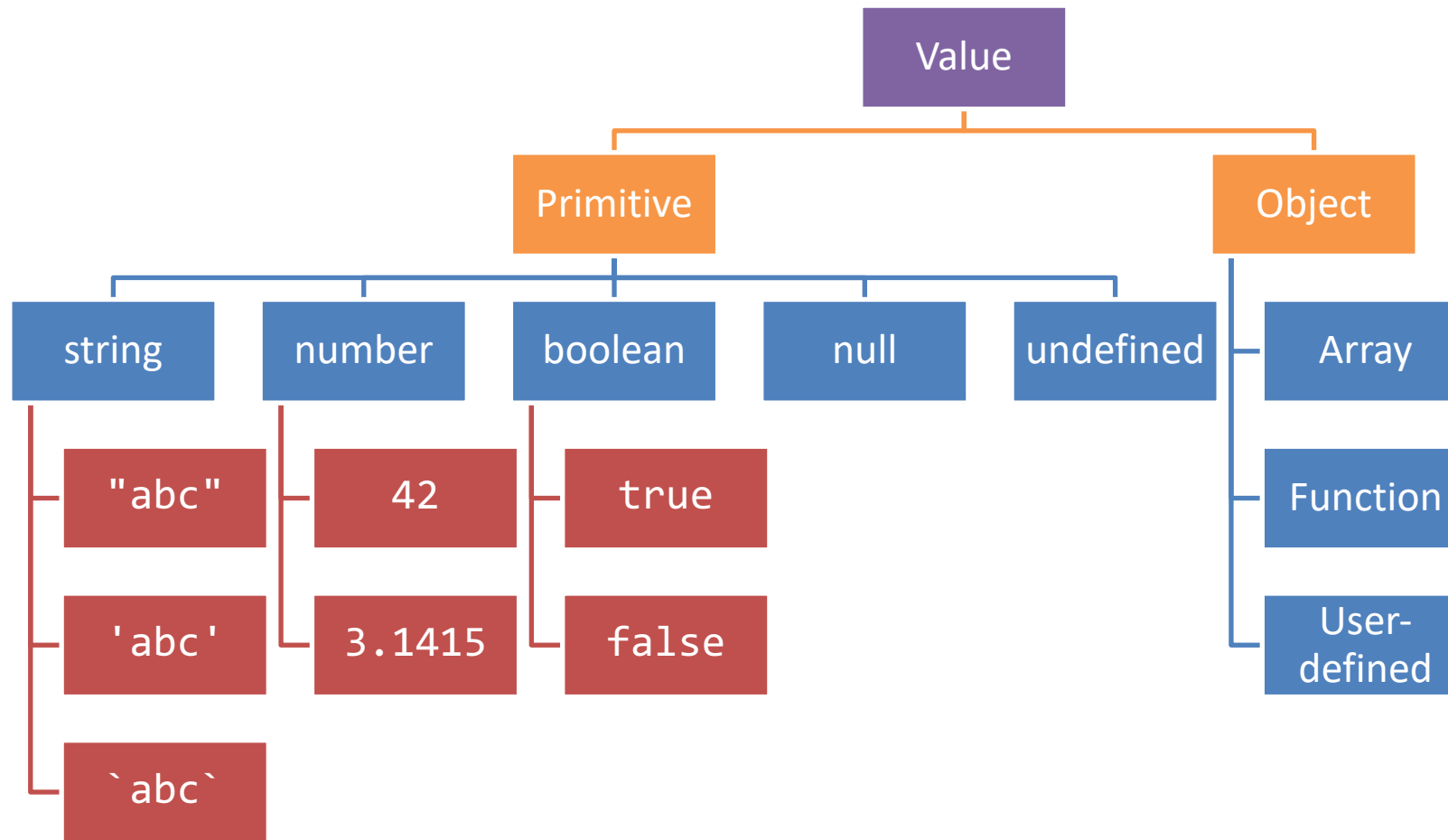


JavaScript – The language of the Web

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



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](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 &lt;&lt;= y</code>	<code>x = x &lt;&lt; y</code>
Right shift assignment	<code>x &gt;&gt;= y</code>	<code>x = x &gt;&gt; y</code>
Unsigned right shift assignment	<code>x &gt;&gt;&gt;= y</code>	<code>x = x &gt;&gt;&gt; y</code>
Bitwise AND assignment	<code>x &amp;= y</code>	<code>x = x &amp; 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 <a href="#">Object.is</a> and <a href="#">sameness in JS</a> .	<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 &gt; var1</code> <code>"12" &gt; 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 &gt;= var1</code> <code>var1 &gt;= 3</code>
Less than (<)	Returns <code>true</code> if the left operand is less than the right operand.	<code>var1 &lt; var2</code> <code>"2" &lt; 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 &lt;= var2</code> <code>var2 &lt;= 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
```

# Logical operators

Operator	Usage	Description
Logical AND ( <code>&amp;&amp;</code> )	<code>expr1 &amp;&amp; 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>&amp;&amp;</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;  
}
```

```
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 ;  
}
```

- 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()`
- We will not cover these methods in this course

# 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 = [] ;
```

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

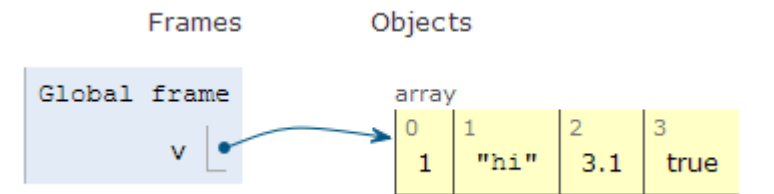
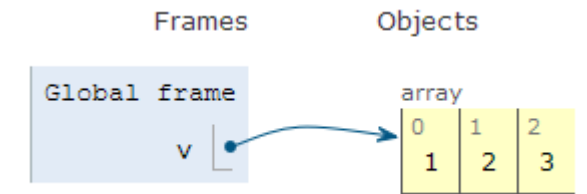
```
let v = [1, "hi", 3.1, true];
```

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

Do not access elements outside range

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

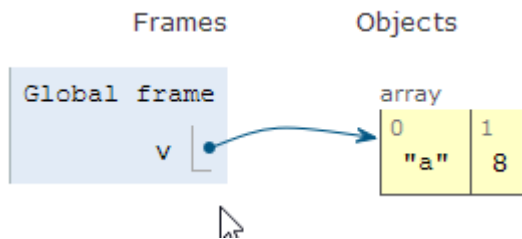
```
let v = Array.of(1, "hi", 3.1, true) ;
```



# Adding elements

`.length` adjusts automatically

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

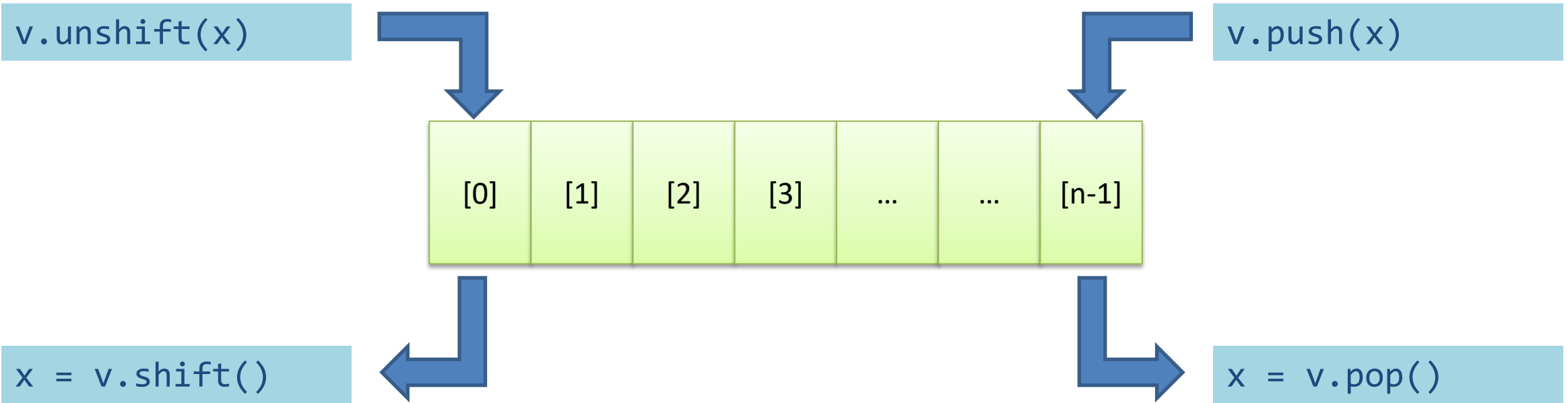


```
let v = [] ;  
v.push("a") ;  
v.push(8) ;  
v.length // 2
```

`.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 ;
```

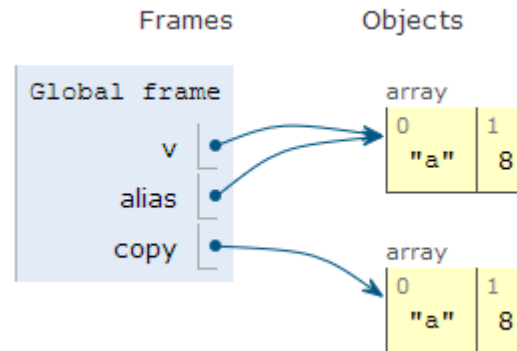
```
> console.log(v); ?  
[ 'a', 5 ]  
undefined  
> console.log(alias);  
[ 'a', 5 ]  
undefined
```

# 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<sup>(\*)</sup> 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.

# 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





## JavaScript: The Definitive Guide, 7th Edition Chapter 5. Objects

### Mozilla Developer Network

- [Learn web development JavaScript » Dynamic client-side scripting » Introducing JavaScript objects](#)
- [Web technology for developers » JavaScript » JavaScript reference » Standard built-in objects » Object](#)
- [Web technology for developers » JavaScript » JavaScript reference » Expressions and operators » in operator](#)

JavaScript – The language of the Web

# OBJECTS

# Big Warnings

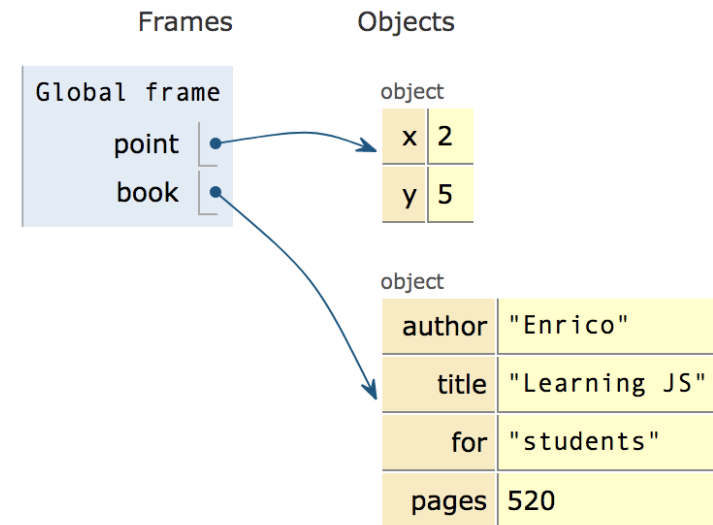
- In JavaScript, Objects may exist without Classes
  - Usually, Objects are **created directly**, without deriving them from a Class definition
- In JavaScript, Objects are dynamic
  - You may **add, delete, redefine** a *property* at any time
  - You may add, delete, redefine a *method* at any time
- In JavaScript, there are no access control methods
  - Every property and every method is always **public** (private/protected don't exist)
- There is no real difference between **properties and methods** (because of how JS functions work)

# Object

- An object is an **unordered collection of properties**
  - Each property has a **name** (key), and a **value**
- You store and retrieve *property values*, through the *property names*
- Object creation and initialization:

```
let point = { x: 2, y: 5 };  
  
let book = {  
  author : "Enrico",  
  title : "Learning JS",  
  for: "students",  
  pages: 520,  
};
```

Object literals syntax:  
{ "name": value,  
 "name": value, }  
or:  
{ name: value,  
 name: value, }



# Object Properties

## Property names are ...

- Identified as a **string**
- Must be unique in each object
- Created at object initialization
- Added after object creation
  - With assignment
- Deleted after object creation
  - With `delete` operator

## Property values are ...

- Reference to any **JS value**
- Stored inside the object
- May be **primitive** types
- May be **arrays**, other **objects**, ...
  - Beware: the object stores the reference, the value is *outside*
- May also be **functions** (*methods*)

# Accessing properties

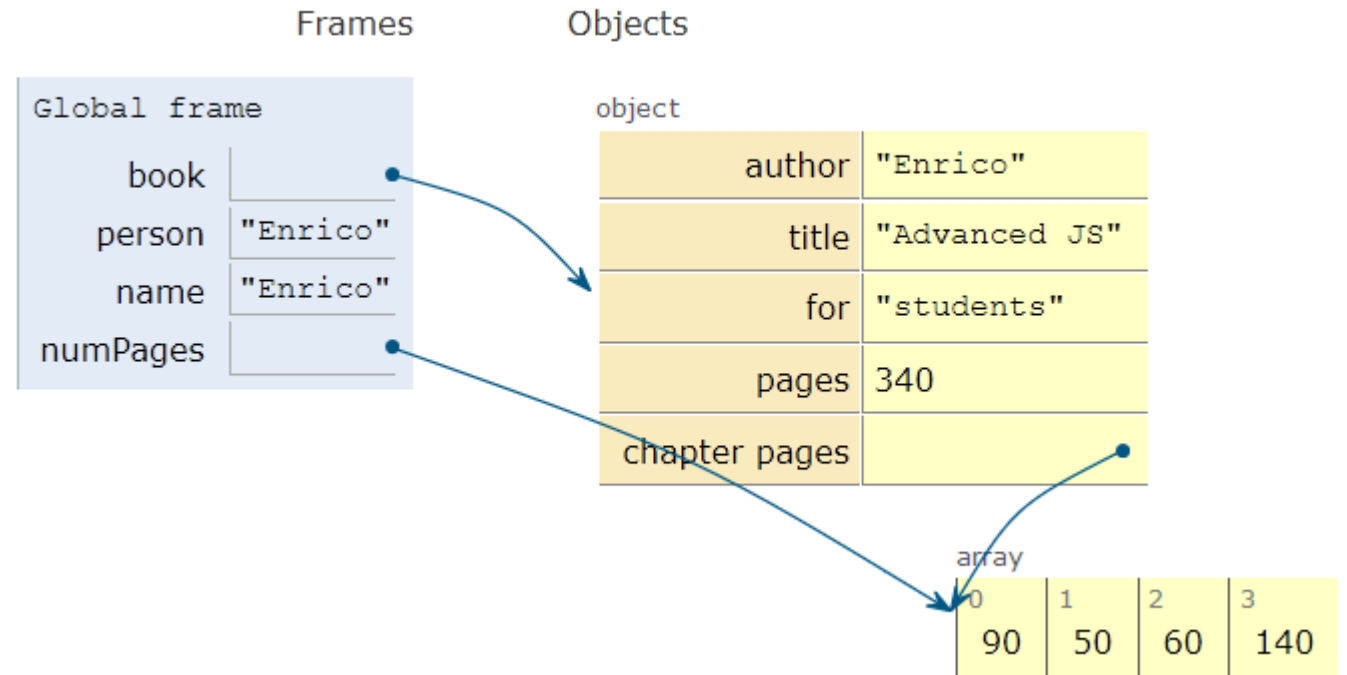
- Dot (.) or square brackets [] notation

```
let book = {  
  author : "Enrico",  
  title : "Learning JS",  
  for: "students",  
  pages: 340,  
  "chapter pages": [90,50,60,140]  
};
```

```
let person = book.author;  
let name = book["author"];  
let numPages =  
  book["chapter pages"];  
book.title = "Advanced JS";  
book["pages"] = 340;
```

The . dot notation and omitting the quotes are allowed **when the property name is a valid identifier, only.**

book.title or book['title']  
book['my title'] and not ~~book.my title~~



# Objects as associative arrays

- The `[]` syntax looks like array access, but the index is *a string*
  - Generally known as *associative arrays*
- Setting a non-existing property creates it:
  - `person["telephone"] = "0110901234";`
  - `person.telephone = "0110901234";`
- Deleting properties
  - `delete person.telephone;`
  - `delete person["telephone"];`

# Computed property names

- Flexibility in creating object properties
  - `{[prop]:value}` -> creates an object with property name equal to *the value of the variable prop*
  - `[]` can contain more complex expressions: e.g., *i*-th line of an object with multiple "address" properties (address1, address2, ...):  
`person["address"+i]`
    - **Using expressions is not recommended...**
- Beware of quotes:
  - `book["title"]` -> property called `title`
    - Equivalent to `book.title`
  - `book[title]` -> property called with the value of variable `title` (if exists)
    - If `title=="author"`, then equivalent to `book["author"]`
    - No equivalent in dot-notation

# Property access errors

- If a property is not defined, the (attempted) access returns `undefined`
- If unsure, must check before accessing
  - Remember: `undefined` is *falsy*, you may use it in Boolean expressions

```
let surname = undefined;
if (book) {
  if (book.author) {
    surname = book.author.surname;
  }
}
```

```
surname = book && book.author && book.author.surname;
```



# Iterating over properties

- **for .. in** iterates over the properties

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

```
x  
y
```

```
let book = {  
    author : "Enrico",  
    pages: 340,  
    chapterPages: [90,50,60,140],  
};
```

```
for (const prop in book)  
    console.log(`${prop} = ${book[prop]}`);
```

```
author = Enrico  
pages = 340  
chapterPages = 90,50,60,140
```

# Iterating over properties

- All the (enumerable) properties names (keys) of an object can be accessed as an array, with:

- `let keys = Object.keys(my_object) ;`

```
[ 'author', 'pages' ]
```

- All pairs [key, value] are returned as an array with:

- `let keys_values = Object.entries(my_object)`

```
[ [ 'author', 'Enrico' ], [ 'pages', 340 ] ]
```



# Object.assign

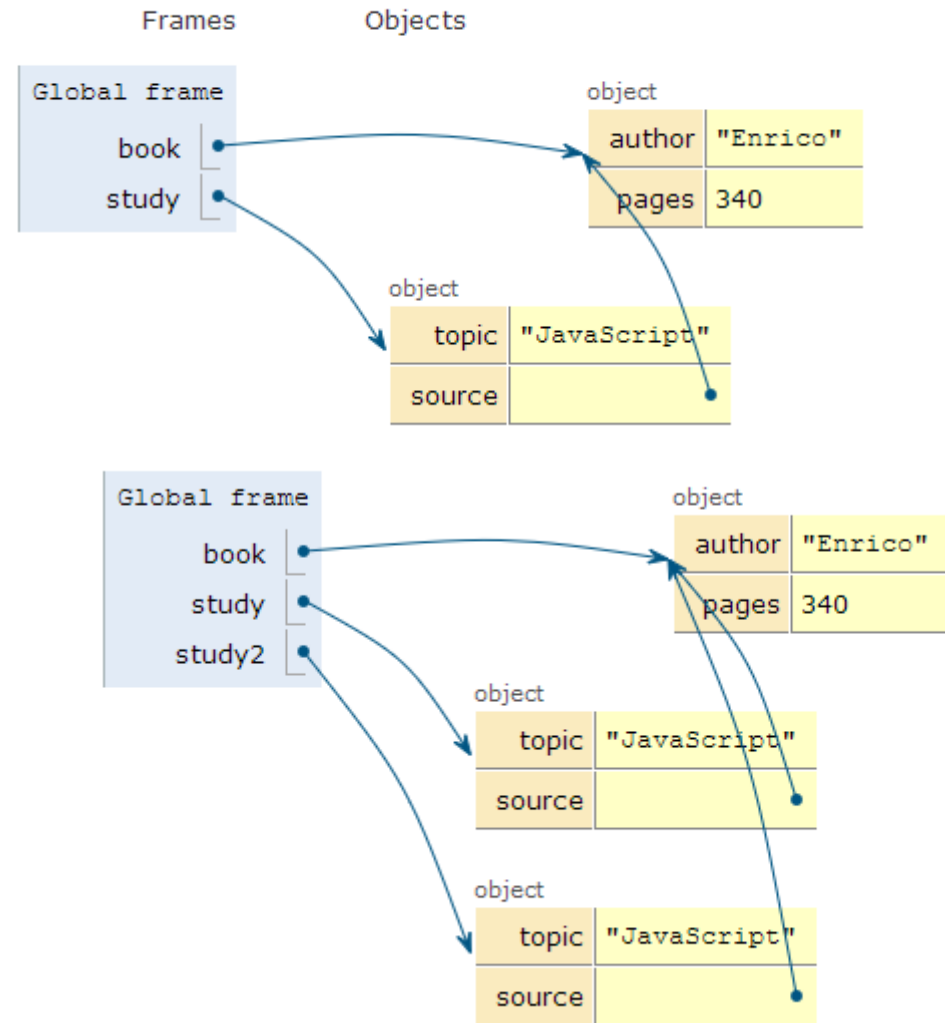
- `let new_object = Object.assign(target, source);`
- Assigns all the properties from the `source` object to the `target` one
- The target may be an existing object
- The target may be a new object: `{}`
- Returns the target object (after modification)

# Beware! Shallow copy, only

```
let book = {  
  author : "Enrico",  
  pages: 340,  
};
```

```
let study = {  
  topic: "JavaScript",  
  source: book,  
};
```

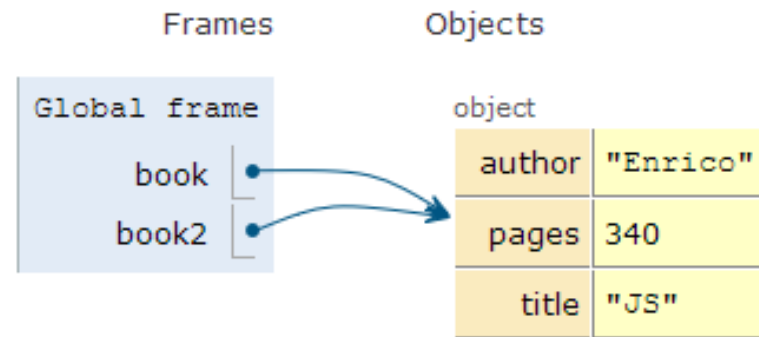
```
let study2 = Object.assign({},  
  study);
```



# Merge properties (on existing object)

- `Object.assign(target, source, default values, ..);`

```
let book = {  
  author : "Enrico",  
  pages: 340,  
};  
  
let book2 = Object.assign(  
  book, {title: "JS"}  
);
```

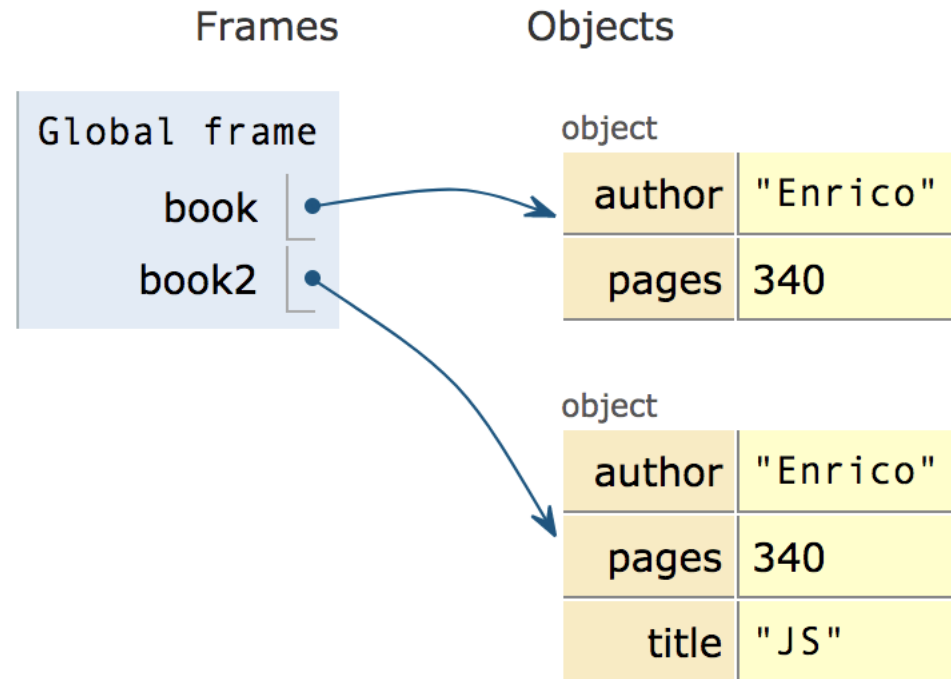


# Merge properties (on new object)

- `Object.assign(target, source, default values, ..);`

```
let book = {
  author : "Enrico",
  pages: 340,
};

let book2 = Object.assign(
  {}, book, {title: "JS"}
);
```



# Checking if properties exist

- Operator **in**
  - Returns true if property is in the object. Do not use with Array

```
let book = {  
  author : "Enrico",  
  pages: 340,  
};  
  
console.log('author' in book);  
delete book.author;  
console.log('author' in book);
```

```
true  
false
```

```
const v=['a','b','c'];  
  
console.log('b' in v);  
  
console.log('PI' in Math);
```

```
false  
true
```



# Object creation (equivalent methods)

- By object literal: `const point = {x:2, y:5} ;`
- By object literal (empty object): `const point = {} ;`
- By constructor: `const point = new Object() ;`
- By object static method create:  
`const point = Object.create({x:2,y:5}) ;`
- Using a *constructor function*

Preferred



JavaScript – The language of the Web

# FUNCTIONS

# Functions

- **One of the most important** elements in JavaScript
- Delimits a block of code with a private scope
- Can accept parameters and returns one value
  - Can also be an object
- Functions themselves **are objects** in JavaScript
  - They can be **assigned** to a variable
  - Can be **passed** as an argument
  - Used as a **return** value

# Declaring functions: 3 ways

## 1) Classic

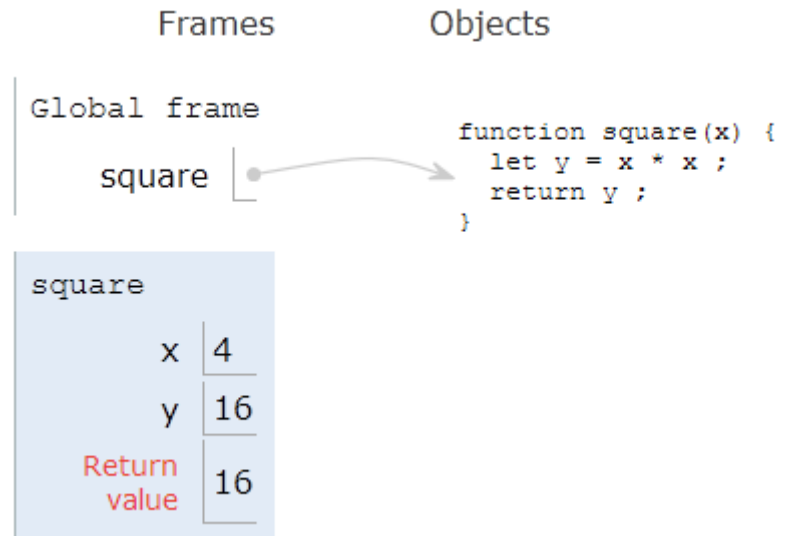
```
function do(params) {  
  /* do something */  
}
```

# Classic functions

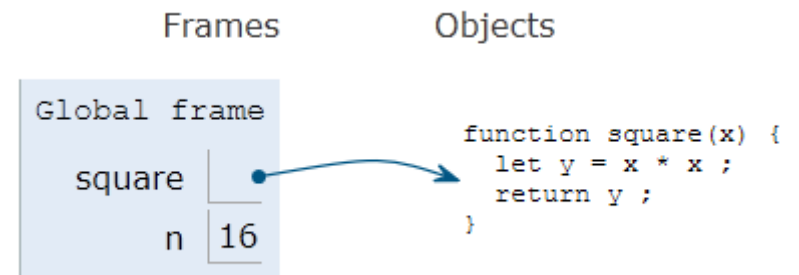
```
function square(x) {  
  let y = x * x ;  
  return y ;  
}
```

```
let n = square(4) ;
```

During execution



After execution



# Parameters

- Comma-separated list of parameter names
  - May assign a default value, e.g., `function(a, b=1) {}`
- Parameters are passed **by-value**
  - Copies of the **reference** to the object
- Parameters that are not passed in the function call get the value 'undefined'
- Check missing/optional parameters with:
  - `if(p===undefined) p = default_value ;`
  - `p = p || default_value ;`

# Variable number of parameters

- Syntax for functions with variable number of parameters, using the `...` operator (called “rest”)

```
function fun (par1, par2, ...arr) { }
```

- The “rest” parameter must be the last, and will deposit all extra arguments into an array

```
function sumAll(initVal, ...arr) {  
  let sum = initVal;  
  for (let a of arr) sum += a;  
  return sum;  
}  
sumAll(0, 2, 4, 5); // 11
```

# Declaring functions: 3 ways

## 1) Classic

```
function do(params) {  
  /* do something */  
}
```

## 2a) Function expression

```
const fn = function(params) {  
  /* do something */  
}
```

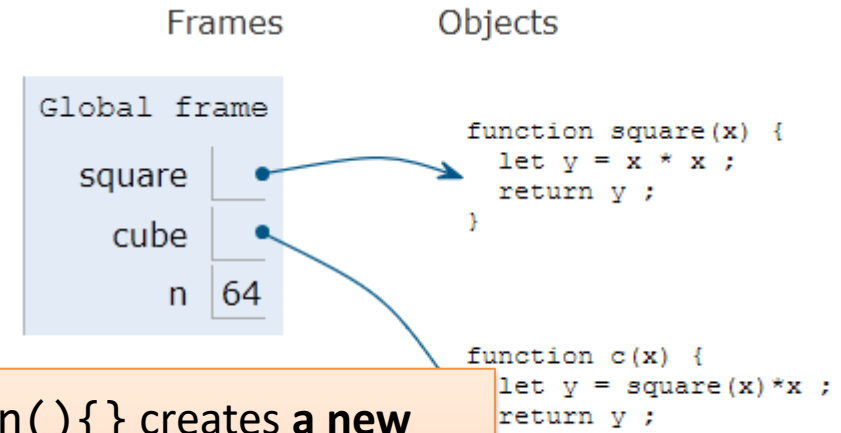
## 2b) Named function expression

```
const fn = function do(params) {  
  /* do something */  
}
```



# Function expression: indistinguishable

```
function square(x) {  
  let y = x * x ;  
  return y ;  
}  
  
let cube = function c(x) {  
  let y = square(x)*x ;  
  return y ;  
}  
  
let n = cube(4) ;
```



The *expression* `function() {}` creates a **new object of type 'function'** and returns the result.

Any variable may “refer” to the function and call it. You can also store that reference into an array, an object property, pass it as a parameter to a function, redefine it, ...

method

callback

# Declaring functions: 3 ways

## 1) Classic

```
function do(params) {  
  /* do something */  
}
```

## 2a) Function expression

```
const fn = function(params) {  
  /* do something */  
}
```

## 3) Arrow function

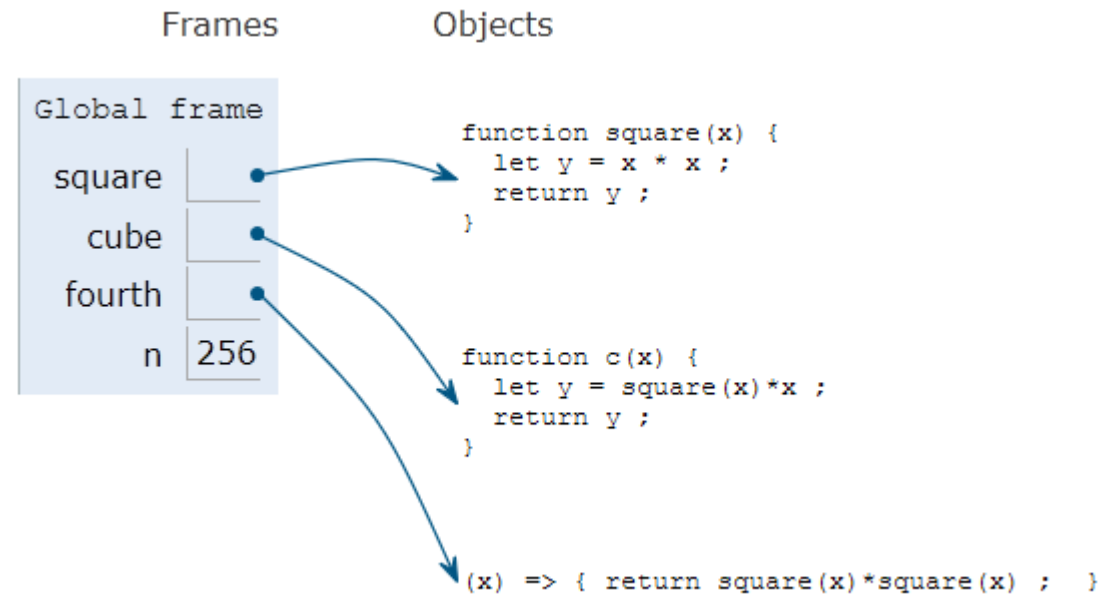
```
const fn = (params) => {  
  /* do something */  
}
```

## 2b) Named function expression

```
const fn = function do(params) {  
  /* do something */  
}
```

# Arrow Function: just a shortcut

```
function square(x) {  
  let y = x * x ;  
  return y ;  
}  
  
let cube = function c(x) {  
  let y = square(x)*x ;  
  return y ;  
}  
  
let fourth = (x) => { return  
square(x)*square(x) ; }  
  
let n = fourth(4) ;
```



# Parameters in arrow functions

```
const fun = () => { /* do something */ } // no params
```

```
const fun = param => { /* do something */ } // 1 param
```

```
const fun = (param) => { /* do something */ } // 1 param
```

```
const fun = (par1, par2) => { /* smtg */ } // 2 params
```

```
const fun = (par1 = 1, par2 = 'abc') => { /* smtg */ } // default values
```

# Return value

- Default: **undefined**
- Use **return** to return a value
- Only one value can be returned
- However, objects (or arrays) can be returned

```
const fun = () => { return ['hello', 5] ; }  
const [ str, num ] = fun() ;  
console.log(str) ;
```

- Arrow functions have **implicit return** if there is only one value

```
let fourth = (x) => { return square(x)*square(x) ; }  
let fourth = x => square(x)*square(x) ;
```

# Nested functions

- Function can be nested, i.e., defined within another function

```
function hypotenuse(a, b) {  
  const square = x => x*x ;  
  return Math.sqrt(square(a) + square(b));  
}
```

=> Preferred in nested functions

```
function hypotenuse(a, b) {  
  function square(x) { return x*x; }  
  return Math.sqrt(square(a) + square(b));  
}
```

- The inner function is *scoped within* the external function and cannot be called outside
- The inner function might *access variables declared* in the *outside* function



JavaScript: The Definitive Guide, 7th Edition  
Chapter 9.4 Dates and Times

Mozilla Developer Network  
Web technology for developers » JavaScript »  
JavaScript reference »  
Standard built-in objects » Date

Day.js  
<https://day.js.org/en/>

JavaScript – The language of the Web

# DATES

# Date object

- Store a time instant with *millisecond* precision, counted from Jan 1, 1970 UTC (Unix Epoch)
- Careful with time zones
  - Most methods work in local time (not UTC) the computer is set to

```
let now = new Date();
```

```
let newYearMorning = new Date(  
2021, // Year 2021  
0, // January (from 0)  
1, // 1st  
18, 15, 10, 743);  
// 18:15:10.743, local time
```

UTC vs Local time zone are confusing.

```
> new Date('2020-03-18')  
2020-03-18T00:00:00.000Z  
> new Date('18 March 2020')  
2020-03-17T23:00:00.000Z
```



Formatting is locale-dependent

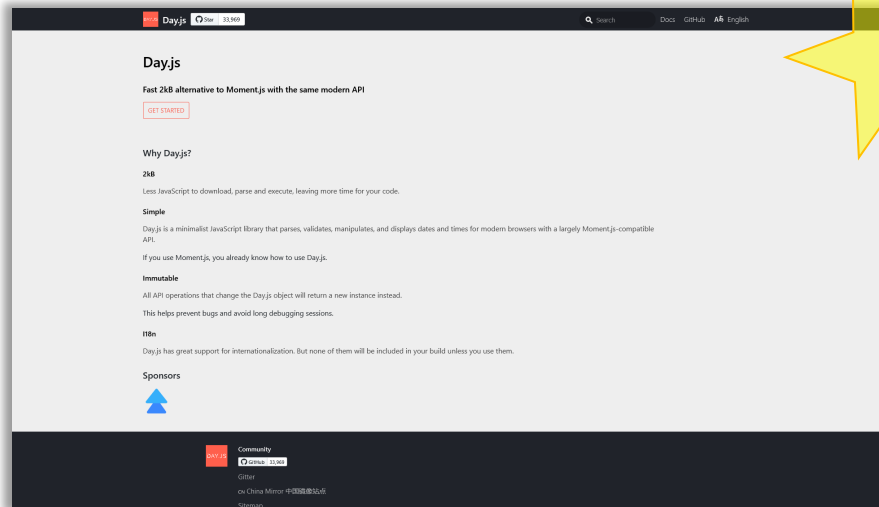


Comparisons are difficult (no way to specify which fields you want, must set them to zero explicitly)

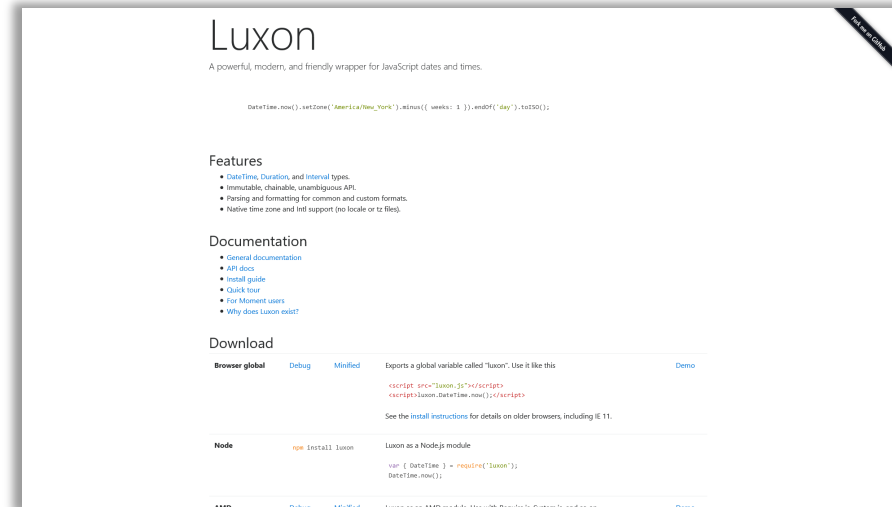




# Serious JS date/time handling libraries



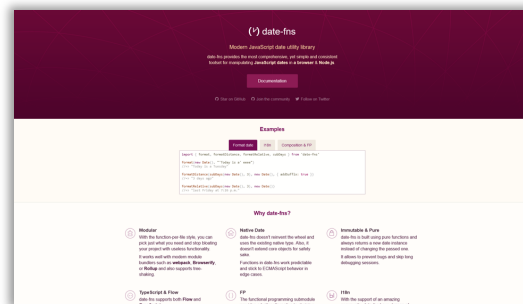
<https://day.js.org/>



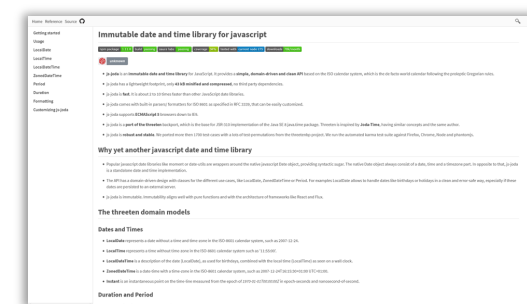
<https://moment.github.io/luxon/>



<https://momentjs.com/>



<https://date-fns.org/>



<https://js-joda.github.io/js-joda/>

# Day.js Library

- Goals
  - Compatible with moment.js
    - But very small (2kB)
  - Works in nodejs and in the browser
  - All objects are *immutable*
    - All API functions that modify a date, will always return a new object instance
  - Localization
  - Plugin system for extending functionality

DAY.JS <https://day.js.org/>

## Browser

EDIT

```
<script src="path/to/dayjs/dayjs.min.js"></script>
<script>
  dayjs().format()
</script>
```

## CDN resource

Day.js can be included by way of a CDN provider like [cdnjs.com](#), [unpkg](#) and [jsDelivr](#) ...

```
<!-- CDN example (jsDelivr) -->
<script src="https://cdn.jsdelivr.net/npm/dayjs@1/dayjs.min.js"></script>
<script>dayjs().format()</script>
```

Check here for more information about loading [locale](#) and [plugin](#).

# Basic operations with Day.js

## Creating date objects – dayjs() constructor

```
let now = dayjs() // today
let date1 = dayjs('2019-12-27T16:00');
    // from ISO 8601 format
let date2 = dayjs('20191227');
    // from 8-digit format
let date3 = dayjs(new Date(2019, 11, 27));
    // from JS Date object
let date5 = dayjs.unix(1530471537);
    // from Unix timestamp
```

By default, Day.js parses in local time

## Displaying date objects – format()

```
console.log(now.format());
    2021-03-02T16:38:38+01:00

console.log(now.format('YYYY-MM [on the] DD'));
    2021-03 on the 02

console.log(now.toString());
    Tue, 02 Mar 2021 15:43:46 GMT
```

By default, Day.js displays in local time

<https://day.js.org/docs/en/parse/parse>

# Get/Set date/time components

```
# obj.unit() -> get
# obj.unit(new_val) -> set

let now2 = now.date(15);
let now2 = now.set('date', 15);
                2021-03-15T16:50:26+01:00

let now3 = now.minute(45);
let now3 = now.set('minute', 45);
                2021-03-02T16:45:26+01:00

let today_day = now.day();
let today_day = now.get('day');
                2
```

Unit	Shorthand	Description
date	D	Date of Month
day	d	Day of Week (Sunday as 0, Saturday as 6)
month	M	Month (January as 0, December as 11)
year	y	Year
hour	h	Hour
minute	m	Minute
second	s	Second
millisecond	ms	Millisecond

<https://day.js.org/docs/en/get-set/get-set>

# Date Manipulation and Comparison

```
let wow = dayjs('2019-01-25').add(1, 'day').subtract(1, 'year').year(2009).toString() ;  
// "Sun, 25 Jan 2009 23:00:00 GMT"
```

- Methods to "modify" a date (and return a modified one)
- `.add` / `.subtract`
- `.startOf` / `.endOf`
- `d1.diff(d2, 'unit')`
- Specify the unit to be added/subtracted/rounded
- Can be easily *chained*
- Day.js objects can be compared
- `.isBefore` / `.isSame` / `.isAfter`
- `.isBetween`
- `.isLeapYear` / `.daysInMonth`

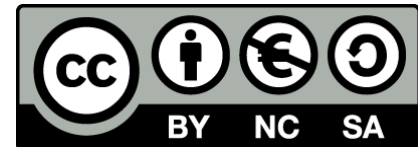
# Day.js Plugins

- To keep install size minimal, several functions are only available in *plugins*
- Plugins must be
  - Loaded
  - Registered into the libraries
- Then, functions may be freely used

```
const isLeapYear =  
  require('dayjs/plugin/isLeapYear') ;  
  // load plugin  
  
dayjs.extend(isLeapYear) ;  
  // register plugin  
  
console.log(now.isLeapYear()) ;  
  // use function
```

# Advanced Day.js Topics

- Localization / Internationalization
  - Language-aware and locale-aware parsing and formatting
  - Various formatting patterns for different locales/languages
- Durations
  - Measuring time intervals (the difference between two time instants)
  - Interval arithmetic
- Time Zones
  - Conversion between time zones



# 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/>

