Metodologie per la Programmazione per il Web - MF0437 *Javascript*

Docente

Giancarlo **Ruffo**[giancarlo.ruffo@uniupo.it]

Informazioni, materiale e risorse su:

moodle [https://www.dir.uniupo.it/course/view.php?id=16455]

Slide adattate da una versione precedente a cura del Prof. Alessio Bottrighi

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

What is JavaScript?

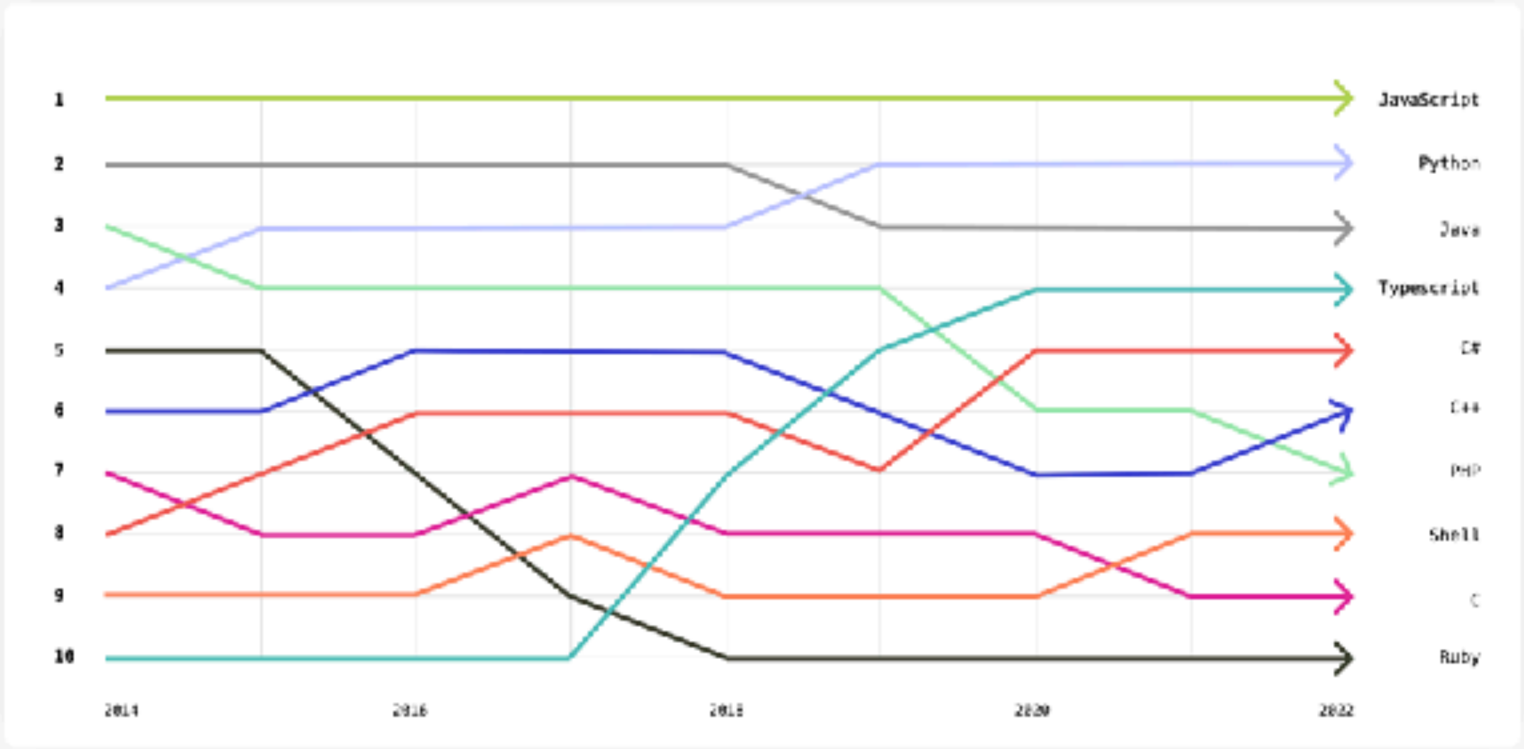
JavaScript – The language of the Web

JavaScript
Fundamentals

Top languages used in 2022

JavaScript continues to reign supreme and Python held steady in the second place position over the past year in large part due to its versatility in everything from development to education to machine learning and data science.

TypeScript also held firm in fourth place year-over-year. Notably, PHP dropped from sixth to seventh place in 2022.

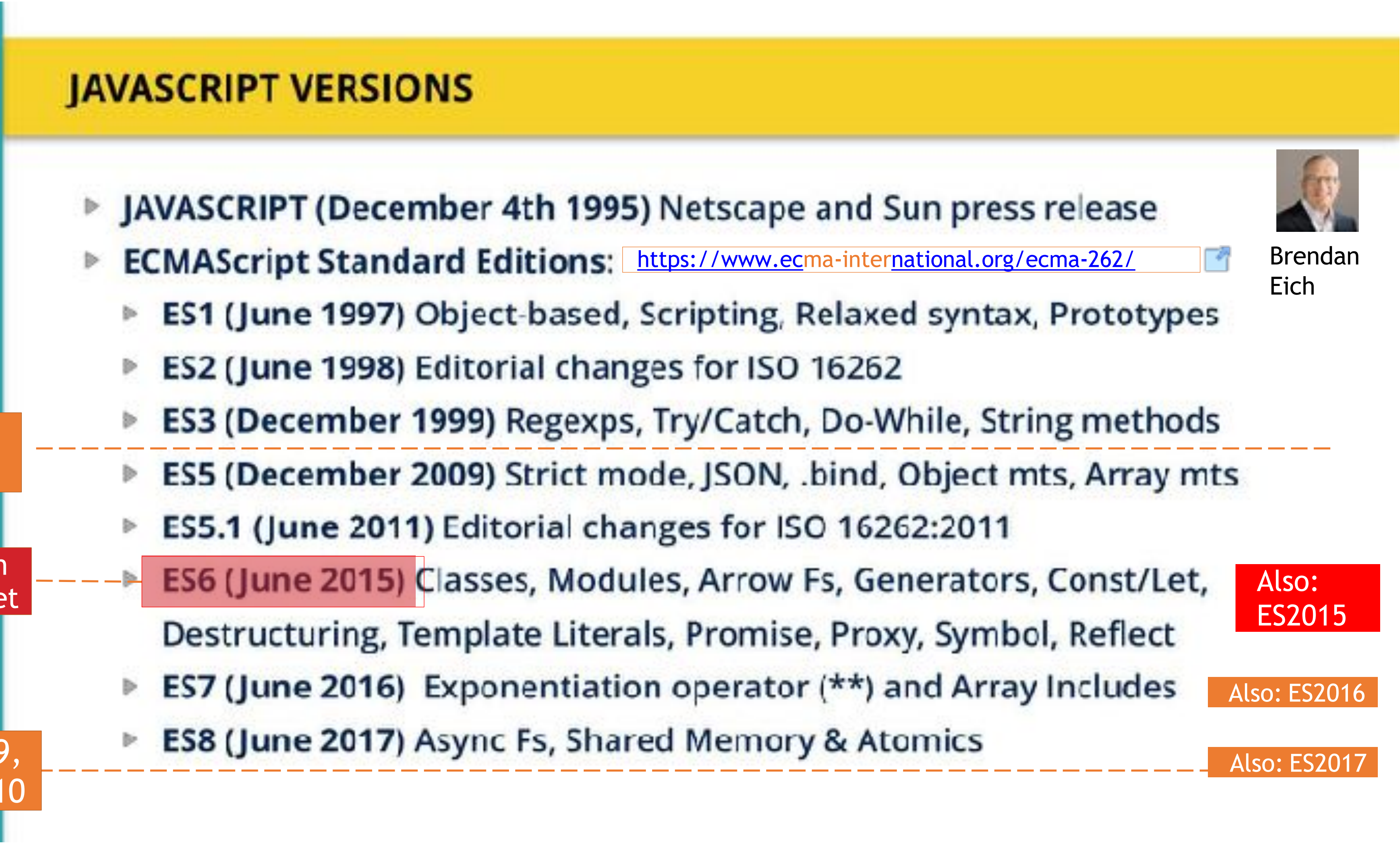


source: https://octoverse.github.com/#top-languages

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!

History and versions



JAVASCRIPT VERSIONS

- 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) Regexps, 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

Brendan Eich

10 yrs

Main target

Also: ES2015

Also: ES2016

Also: ES2017

ES9, ES10

https://www.slideshare.net/RafaelCasusoRomate/javascript-editions-es7-es8-and-es9-vs-v8

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 the following**

Official ECMA standard (formal and unreadable)

Search...

TABLE OF CONTENTS

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.

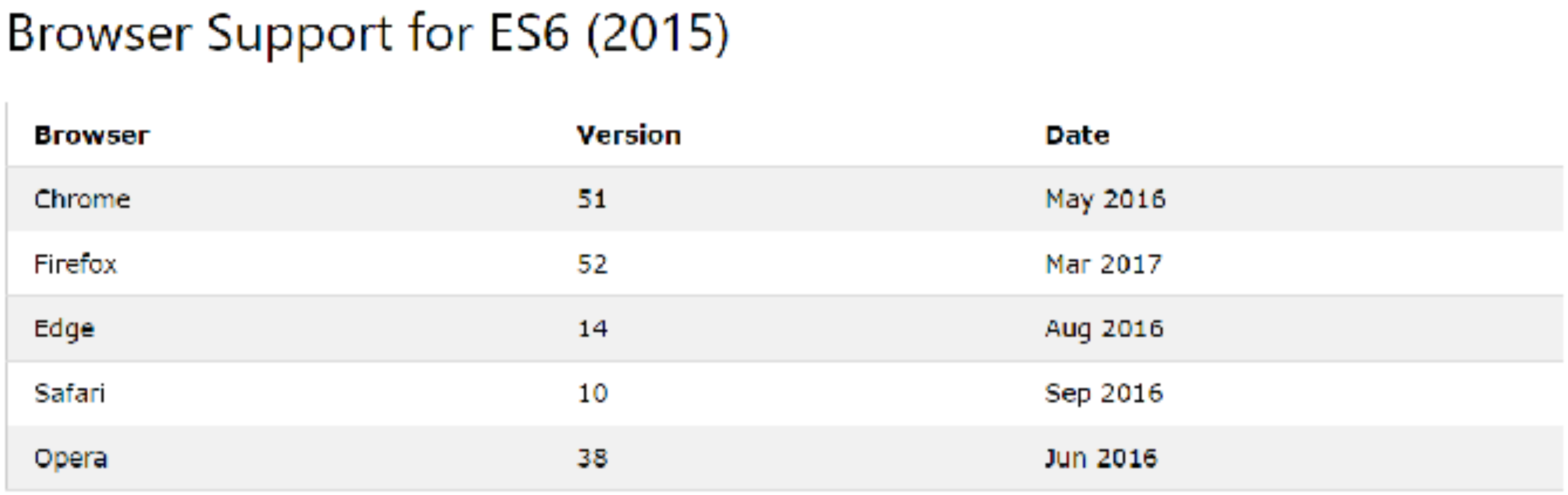
Introduction

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
- Rhino by Mozilla
  - written in Java

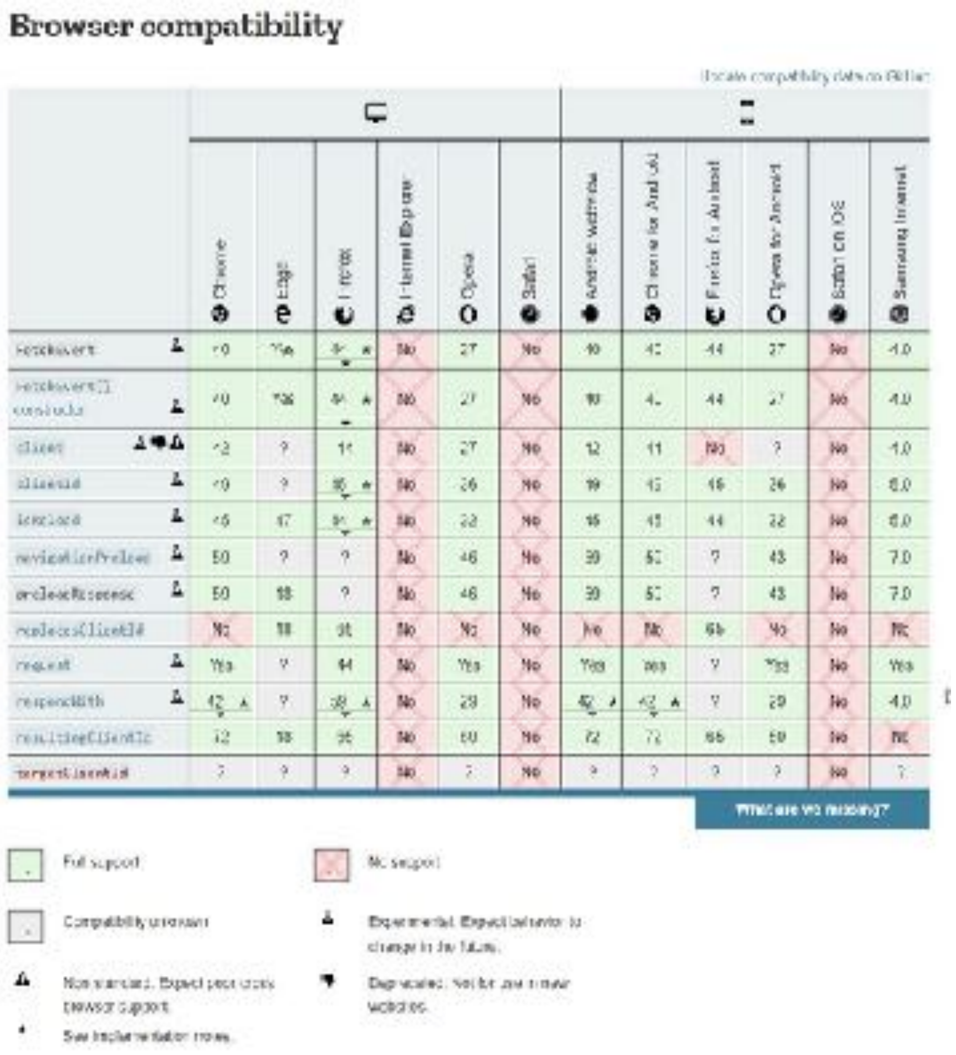
Standard vs. Implementation (in browsers)



Browser Support for ES6 (2015)

Browser	Version	Date
Chrome	51	May 2016
Firefox	52	Mar 2017
Edge	14	Aug 2016
Safari	10	Sep 2016
Opera	38	Jun 2016

Standard vs. Implementation (in browsers)



Browser compatibility

Update compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
FetchEvent	40	Yes	44	No	27	No	40	40	44	27	No	4.0
FetchEvent() constructor	40	Yes	44	No	27	No	40	40	44	27	No	4.0
client	42	?	44	No	27	No	42	44	No	?	No	4.0
clientId	49	?	45	No	26	No	49	45	45	26	No	5.0
isReload	45	17	45	No	32	No	45	45	44	32	No	5.0
navigationPreload	59	?	?	No	46	No	59	59	?	43	No	7.0
preloadResponse	59	18	?	No	46	No	59	59	?	43	No	7.0
replacesClientId	No	18	65	No	No	No	No	No	65	No	No	No
request	Yes	?	44	No	Yes	No	Yes	Yes	?	Yes	No	Yes
respondWith	42	?	59	No	29	No	42	42	?	29	No	4.0
resultingClientId	72	18	65	No	60	No	72	72	65	50	No	No
targetClientId	?	?	?	No	?	No	?	?	?	?	No	?

What are we missing?

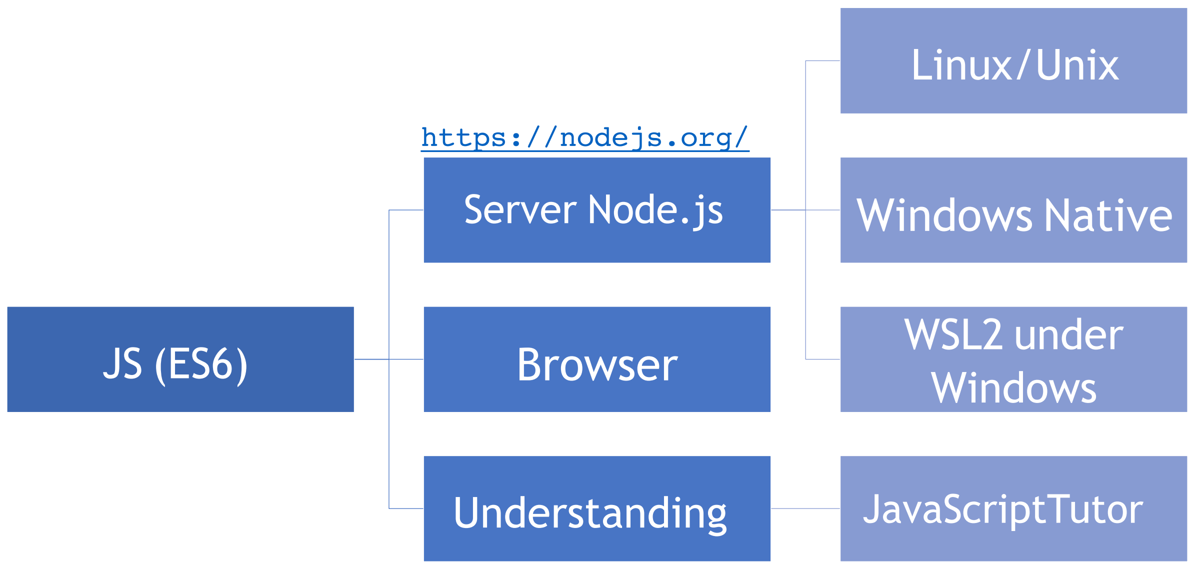
🟩	Full support	🟥	No support
▢	Compatibility unknown	🧪	Experimental. Expect behavior to change in the future.
⚠	Non-standard. Expect poor cross-browser support.	👎	Deprecated. Not for use in new websites.
*	See implementation notes.		

https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Browser_support_for_JavaScript_APIs

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



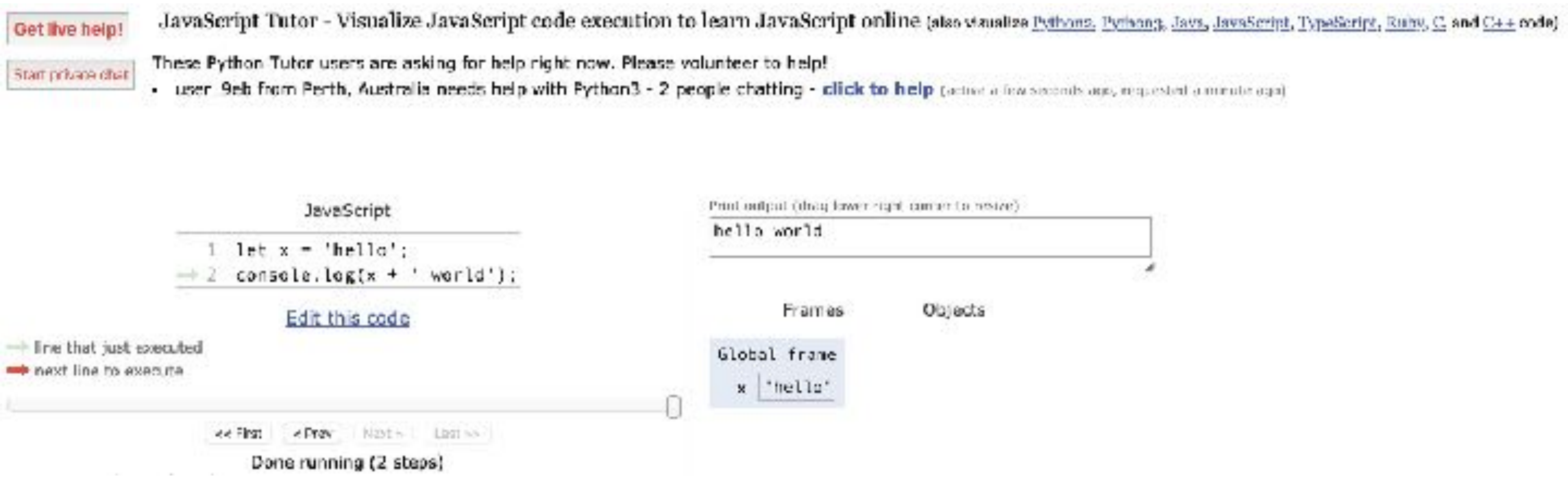
https://nodejs.org/en/download/package-manager/

https://docs.microsoft.com/en-us/windows/nodejs/setup-on-windows

https://docs.microsoft.com/en-us/windows/nodejs/setup-on-wsl2

http://pythontutor.com/javascript.html

JavaScriptTutor



JavaScript Tutor - Visualize JavaScript code execution to learn JavaScript online (also visualize Python2, Python3, Java, JavaScript, TypeScript, Ruby, C and C++ code)

These Python Tutor users are asking for help right now. Please volunteer to help!

- user_9eb from Perth, Australia needs help with Python3 - 2 people chatting - **click to help** (active a few seconds ago, requested a minute ago)

JavaScript

```
let x = 'hello';
console.log(x + ' world');
```

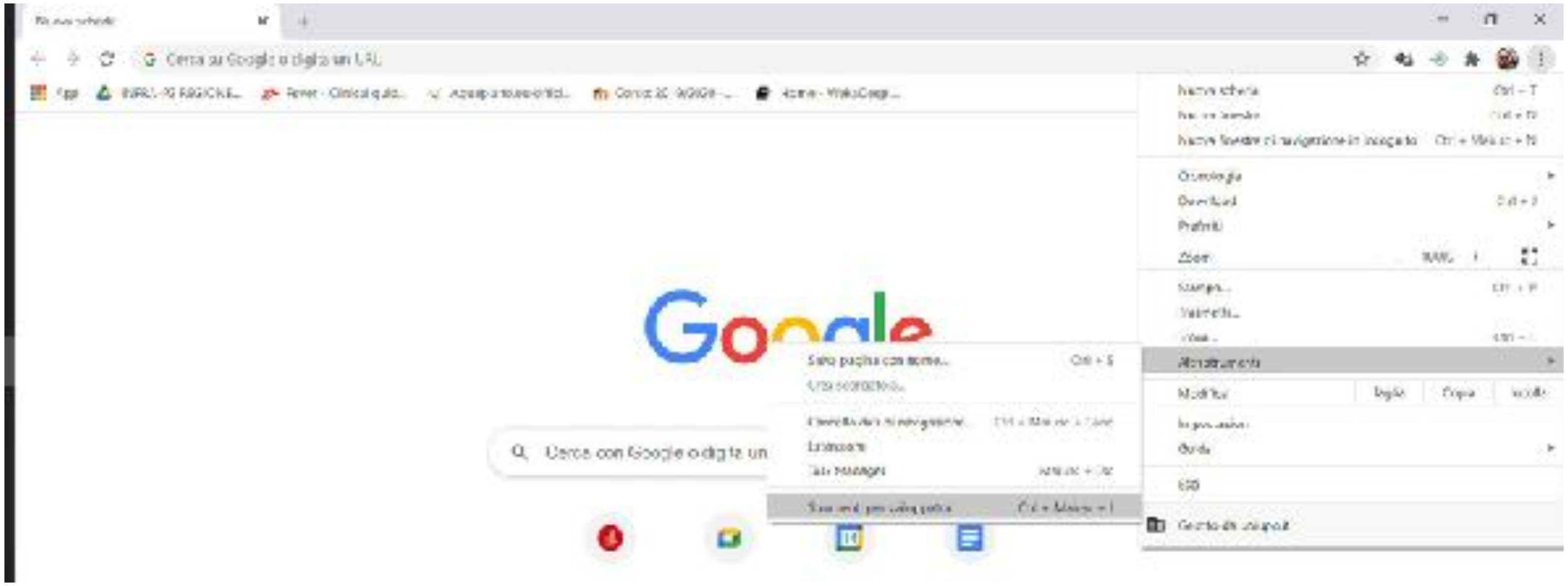
Done running (2 steps)

Print output (drag lower right corner to resize)

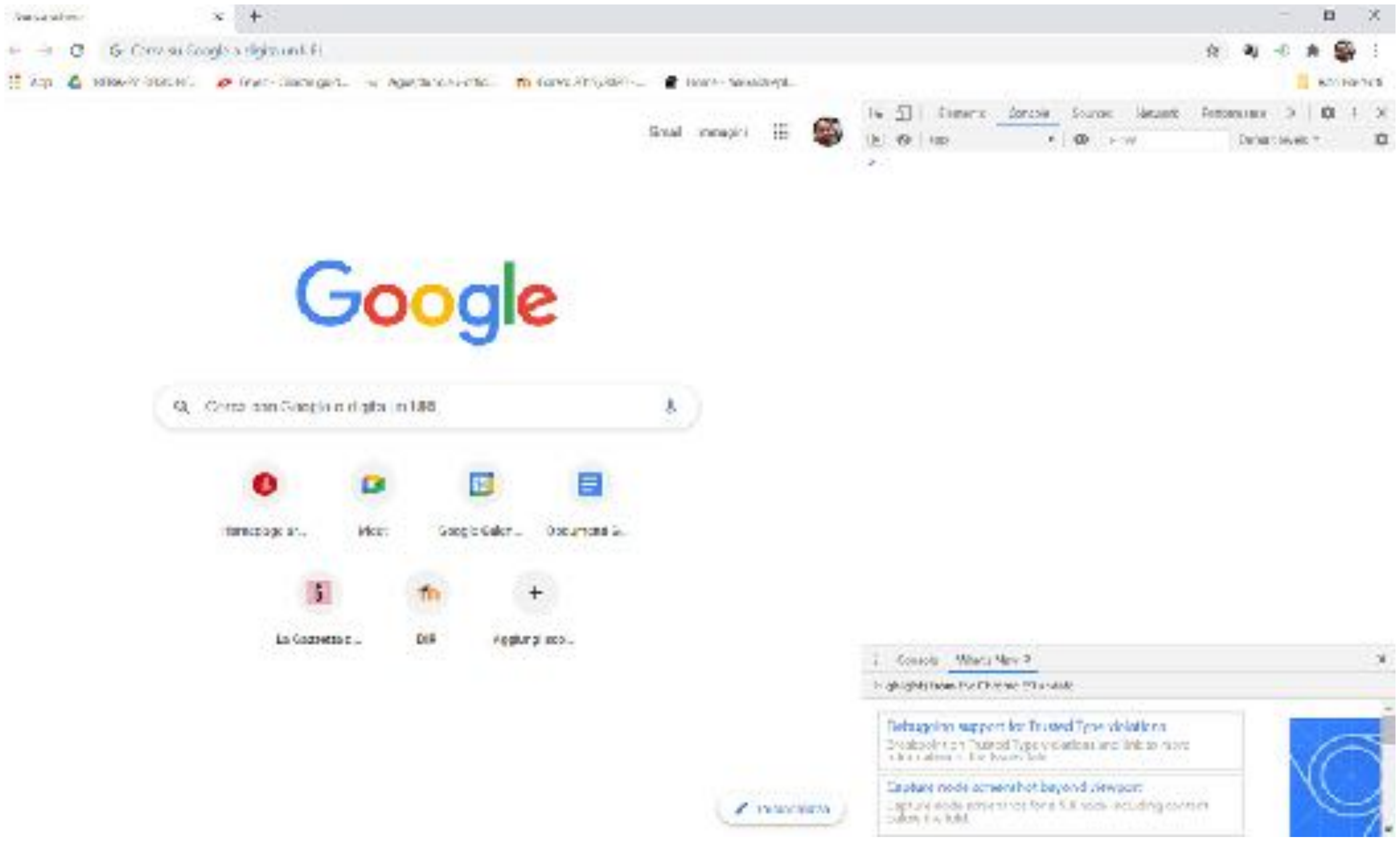
```
hello world
```

http://pythontutor.com/javascript.html

Browser and JS console


Cerca su Google o digita un URL
Nuova scheda Ctrl + T
Nuova finestra Ctrl + N
Nuova finestra di navigazione in incognito Ctrl + Maiusc + N
Cronologia
Download Ctrl + J
Preferiti
Zoom
Stampa... Ctrl + P
Trasmetti...
Trova... Ctrl + F
Altri strumenti
Modifica Taglia Copia Incolla
Impostazioni
Guida
Esci
Salva pagina con nome... Ctrl + S
Crea scorciatoia...
Cancella dati di navigazione... Ctrl + Maiusc + Canc
Estensioni
Task Manager Maiusc + Esc
Strumenti per sviluppatori Ctrl + Maiusc + I
Cerca con Google o digita un URL

Browser and JS console


Google
Elements
Console
Sources
Network
Performance
Console
What's New
Debugging support for Trusted Type violations
Capture node screenshot beyond viewport

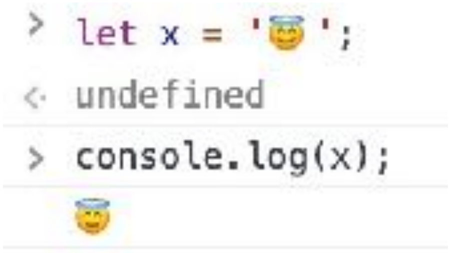
Lexical structure

- One file = one JS program
  - Each file is loaded independently
  - Different files/programs may communicate through global state
  - The "module" mechanism extends that
    - it provides state sharing in a clean way
- The file is entirely parsed, and then executed from top to bottom
- Relies on a standard library
  - plus 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
  - they are automatically inserted (see next slide)
- Case sensitive
- Comments as in C (/*..*/ and //)
- Literals and identifiers (only start with letter, $, _)
- Some reserved words (e.g., while, let, for, int, if, …)
- C-like syntax



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

Semicolon (;)

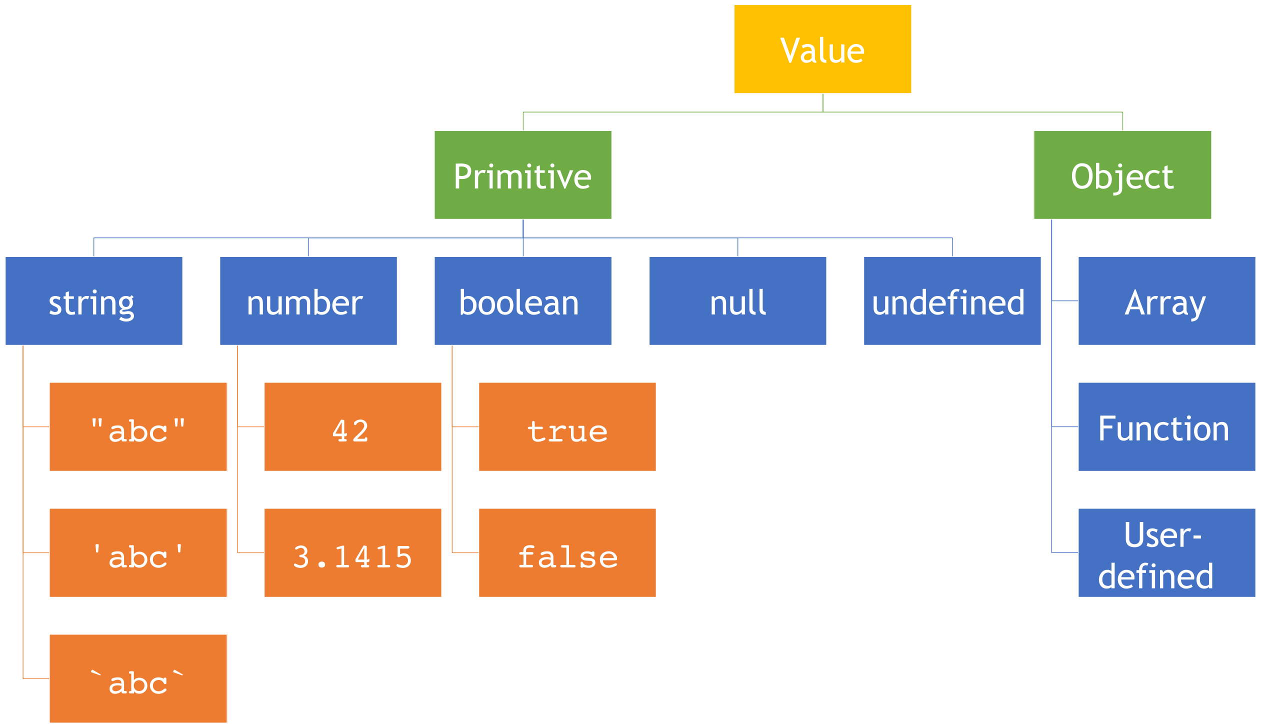
- Argument of debate in the JS community
- JS inserts them as needed
  - When next line starts with code that breaks the current one
  - When the next line starts with }
  - When there is **return, break, throw**, **continue** on its own line
- Be careful that forgetting semicolon can lead to **unexpected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

Strict Mode

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

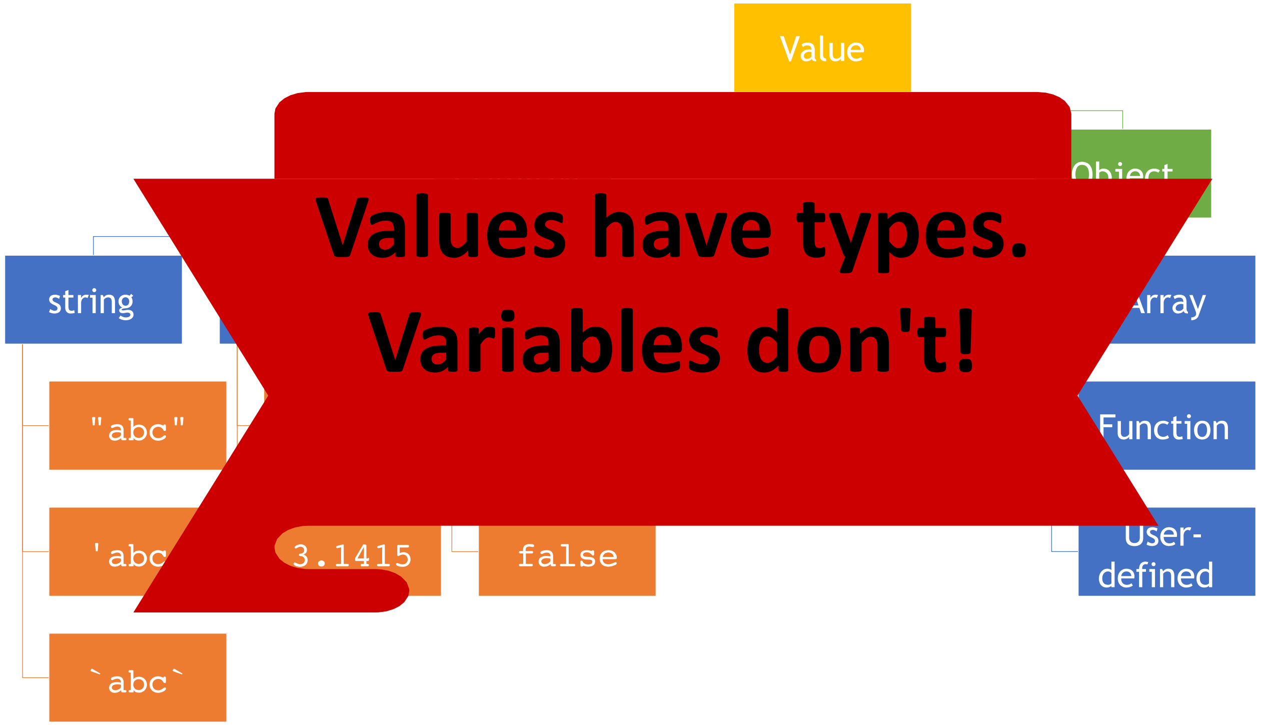
- Directive introduced in ES5: "use strict";
  - compatible with older version (it is just a string)
- Code is executed in strict mode
  - it fixes some important language deficiencies and provides stronger error checking and security
  - examples:
    - eliminates some JavaScript silent errors by changing them to throw errors
    - 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
    - prohibits some syntax likely to be defined in future versions of ECMAScript
    - cannot define 2 or more properties or function parameters with the same name
    - no octal literals (base 8, starting with 0)
    - …

Values and Types



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

Values and Types



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

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

- 'boolean' type with literal values: true, false
- When converting to boolean
- The following values are 'false'
  - 0, -0, 0.0, NaN, undefined, null, '' (empty string)
- Every other value is 'true'
  - 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
- Integer numbers max out at 2^{53} - 1

- There is also a distinct type "BigInt" (ES11, July 2020)
  - an arbitrary-precision integer, can represent numbers larger than 2^{53} - 1,
  - 123456789n   //With suffix   'n'
  - BigInt("9007199254740991")

Special values

- **Undefined**: variable declared but not initialized
  - detectable with:typeof variable === '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

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

- The same variable may refer to different values (even of different types) at different times

- Three ways to declare a variable:
  - `let`
  - `const`
  - `var`

```
> 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	Examples	Can reassign?	Can re-declare?	Scope	Hoisting *	Note
`let`	`let a; let a=2;`	Yes	No	Enclosing block {…}	No	***Preferred***
`const`	`const a=2;`	No §	No	Enclosing block {…}	No	***Preferred (x2)***
`var`	`var a; var a=2;`	Yes	Yes	Enclosing function, or global	Yes, to beginning of function or file	*Legacy, beware its quirks, try not to use*
None (implicit)	a=2;	Yes	N/A	Global	Yes	*Forbidden in strict mode*

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

* Hoisting = "lifting up" the declaration of a variable (<u>not</u> 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";
                                       var c ; // hoisted
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);
```

Variabili locali e globali

- **Variabile locale**: esiste solo all'interno di una funzione particolare
- **Variabile globale**: può essere acceduta e modificata da qualsiasi parte del codice JS nella pagina

```
var a = 2;                // variabile globale
// qui: a = 2, b = undefined, c = undefined
                                        var b; // hoisted
function scopeTest() {
      a = 2*2;
      b = 3;              // dichiarata globale implicitamente
      var c = 8;          // variabile locale
// qui: a = 4, b = 3, c = 8
}

scopeTest();
// qui: a = 4, b = 3, c = undefined
```

Expressions

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

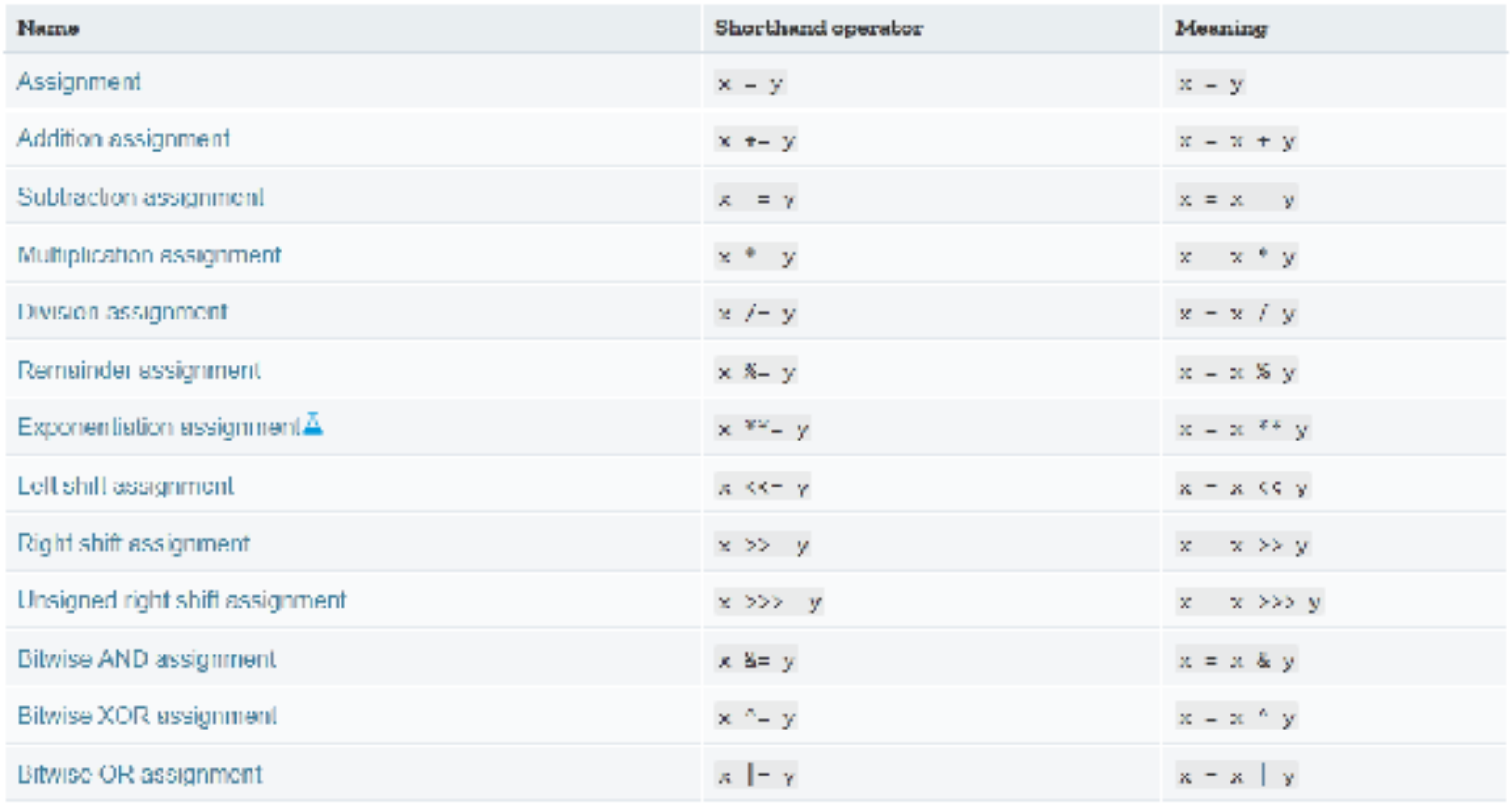
Operators

- 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/O perator_Precedence#Table

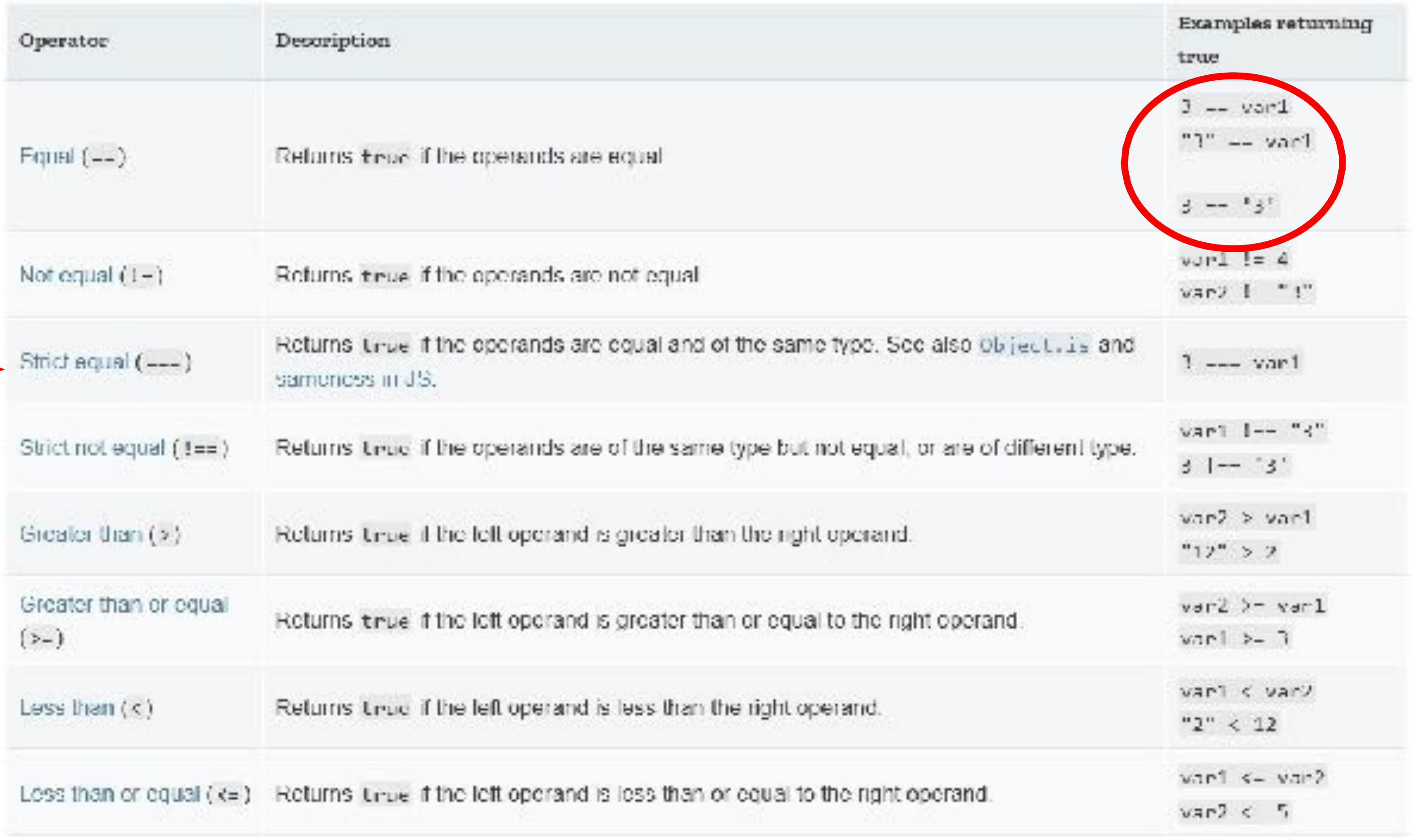
Assignment

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



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

Comparison operators



Operator	Description	Examples returning true

Equal (`==`)	Returns `true` if the operands are equal.	`3 == var1`
		`"3" == var1`
		`3 == '3'`

Not equal (`!=`)	Returns `true` if the operands are not equal.	`var1 != 4`
		`var2 != "3"`

Strict equal (`===`)	Returns `true` if the operands are equal and of the same type. See also `Object.is` and sameness in JS.	`3 === var1`

Strict not equal (`!==`)	Returns `true` if the operands are of the same type but not equal, or are of different type.	`var1 !== "3"`
		`3 !== '3'`

Greater than (`>`)	Returns `true` if the left operand is greater than the right operand.	`var2 > var1`
		`"12" > 2`

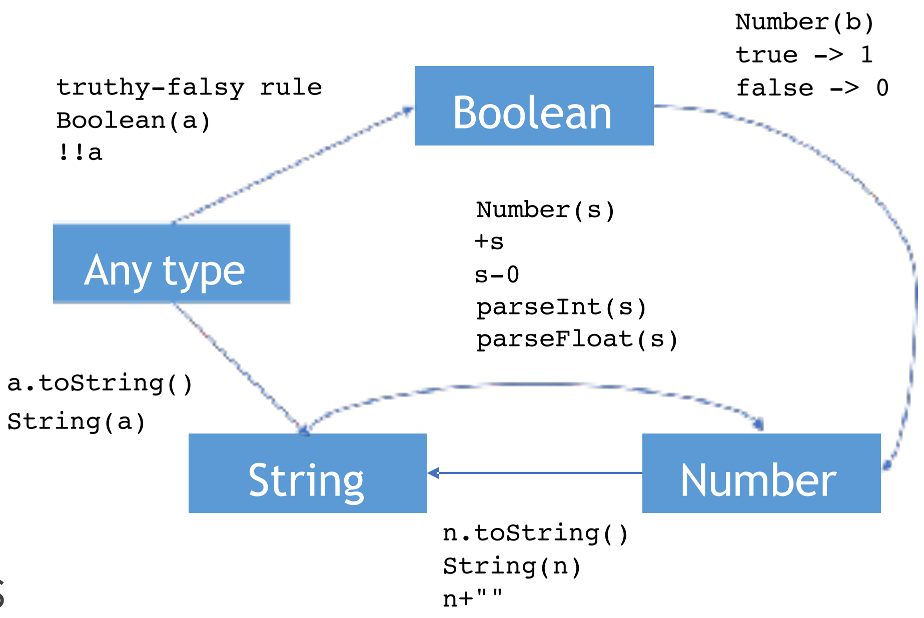
Greater than or equal (`>=`)	Returns `true` if the left operand is greater than or equal to the right operand.	`var2 >= var1`
		`var1 >= 3`

Less than (`<`)	Returns `true` if the left operand is less than the right operand.	`var1 < var2`
		`"2" < 12`

Less than or equal (`<=`)	Returns `true` if the left operand is less than or equal to the right operand.	`var1 <= var2`
		`var2 <= 5`

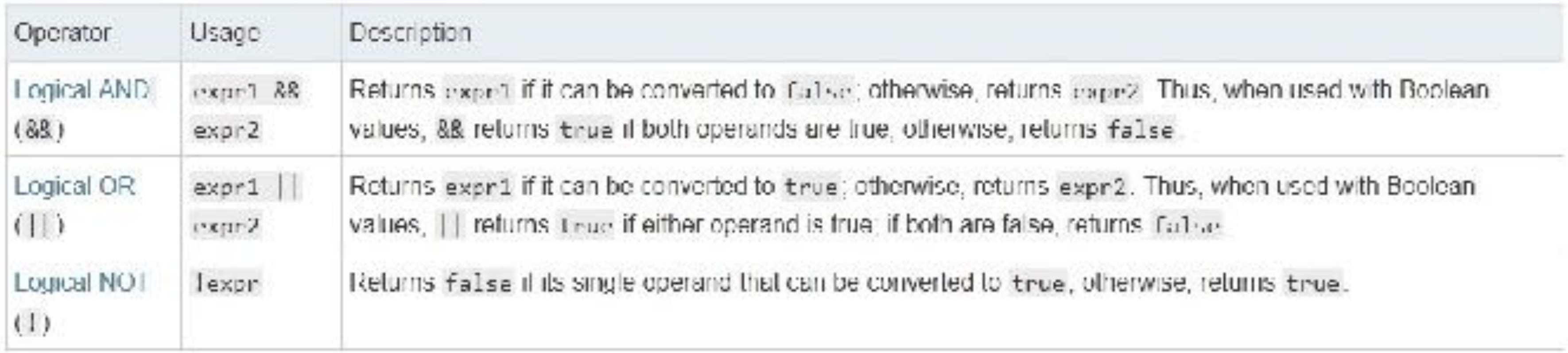
Automatic Type Conversions

- 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



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

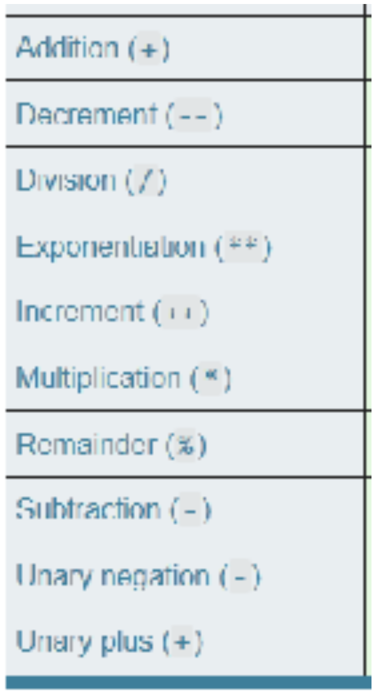
Logical operators



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

Common operators

Or string concatenation



Addition (+)

Decrement (--)

Division (/)

Exponentiation (**)

Increment (++)

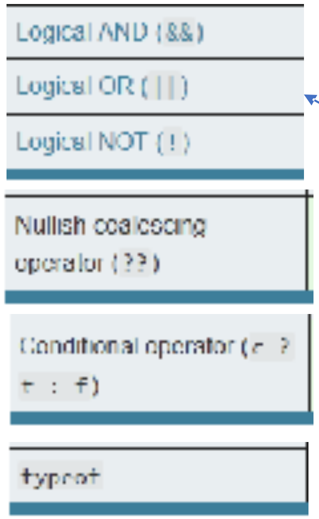
Multiplication (*)

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)

Operatori logici e risultati non boleani

```
let s = "Espresso";
let n = null;

let e = (s) ? s : "Coffee"; // e is "Espresso"

let e = s || "Coffee";                 // e is "Espresso"
let f = n || "Coffee";                 // f is "Coffee"
let g = n || s;                        // g is "Espresso"
let h = 0 || n;                        // h is null
```

- Questo comportamento può essere sfruttato per assegnare dei valori di default:

```
let donation = value || 5.00;
```

Mathematical functions (`Math` building 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()`

Control Structures

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

Mozilla Developer Network
JavaScript Guide » Control Flow and Error Handling
JavaScript Guide » Loops and Iteration

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_de
     f [break;]
}
```

Loop statements

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

Usually declare loop variable

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

May use break;
or continue;

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

Special 'for' statements

Preferr
ed

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

```
x
y
```

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

```
4
7
```

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

```
h
i
```

Other iteration methods

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

Exception handling

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

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

```
throw object ;
```

Exception object

```
EvalError
RangeError
ReferenceError
SyntaxError
TypeError
URIError
DOMException
```

Contain fields: name, message

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

Strings

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

Mozilla Developer Network
JavaScript Guide » Text Formatting

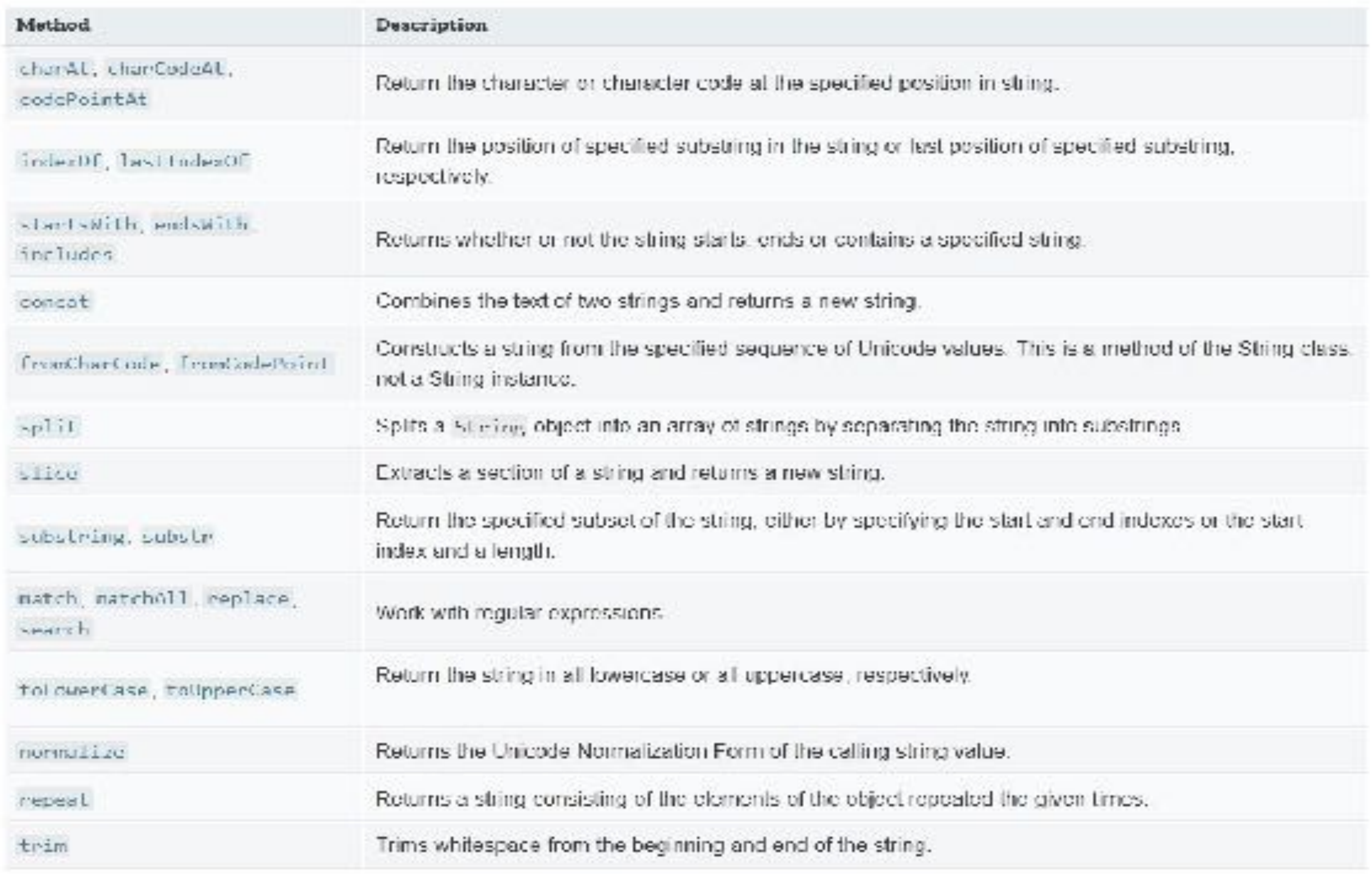
Strings in JS

- A string is an **immutable** ordered sequence of Unicode characters
- The length of a string is the number of characters it contains
- 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
- `s[3]`: indexing
- `s1 + s2`: concatenation
- `s.length`: number of characters

String methods



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

Arrays

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

Mozilla Developer Network
JavaScript Guide » Indexed Collections

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

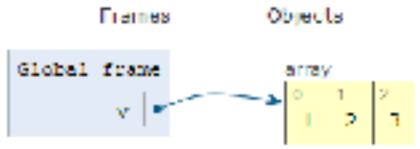
Elements are indexed at positions 0…length-1

Do not access elements outside range

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

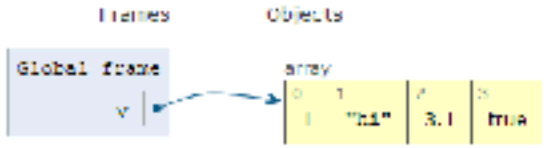
```
let v = [1, 2, 3];
```

```
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

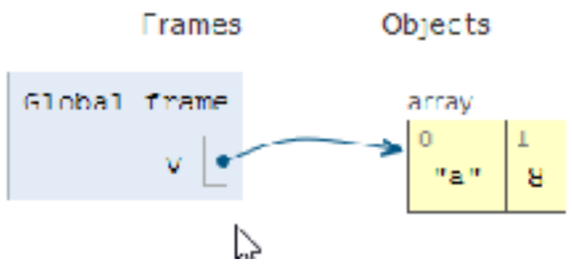
```
let v = [] ;
v[0] = "a" ;
v[1] = 8 ;
v.length // 2
```

.length adjusts automatically

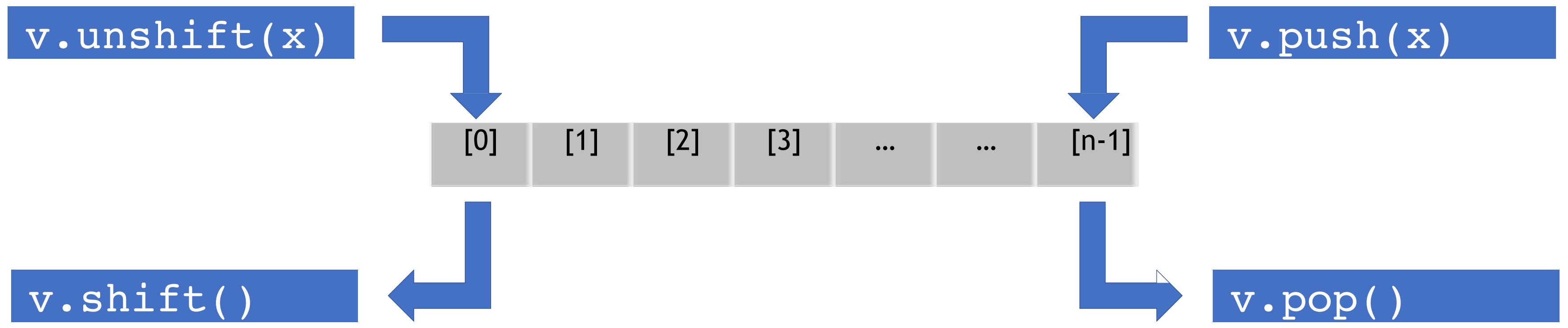
```
let v = [] ;
v.push("a") ;
v.push(8) ;
v.length // 2
```

.push() adds at the end of the array

.unshift() adds at the beginning of the array



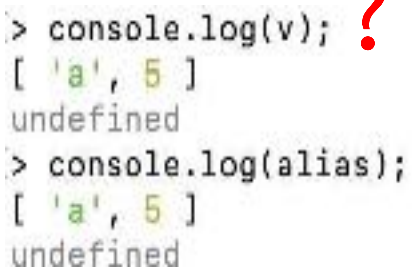
Adding and Removing from arrays (in-place)



Copying arrays

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

let alias = v ;
 alias[1] = 5 ;
```



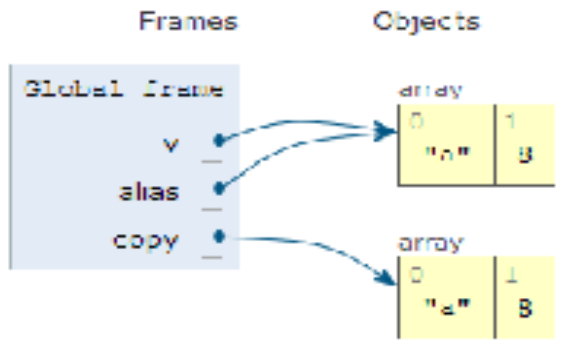
```
> console.log(v);
[ 'a', 5 ]
undefined
> console.log(alias);
[ 'a', 5 ]
undefined
```

?

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)

Iterating over Arrays: Example

```
const v = ['a', 'b', 1] ;

for (const element of v) {
  console.log(element) ;
}
```

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];

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 interable 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]`