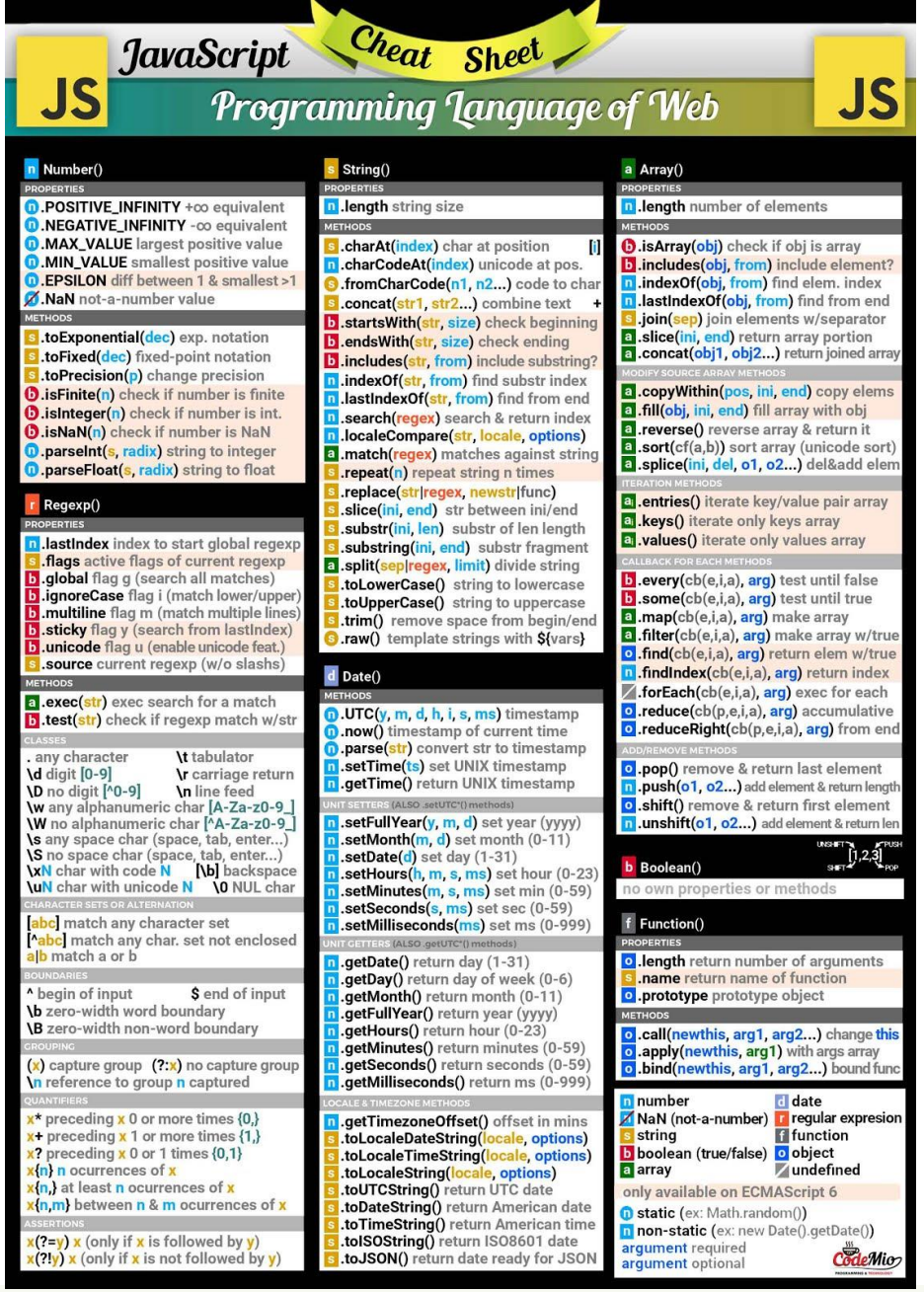
<WA1/>
<AW1/>
2026

JavaScript (basics)

The language of the Web

Fulvio Corno
Luigi De Russis



Goal

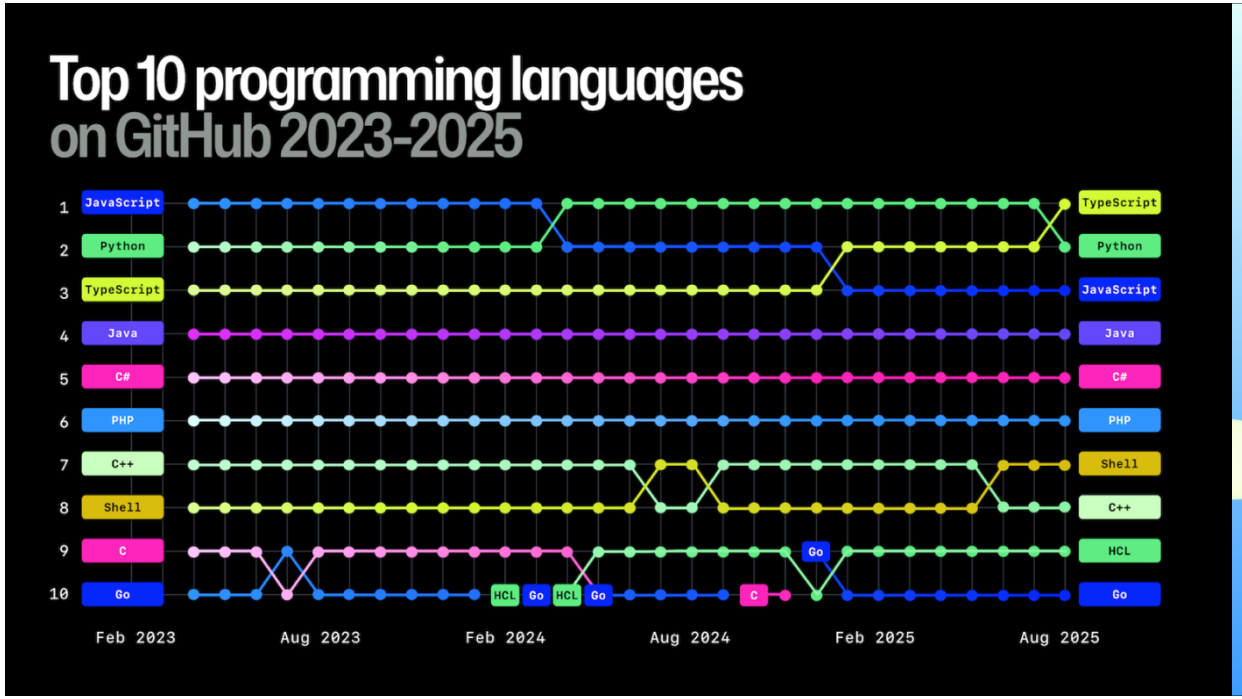
- Learn JavaScript as a language
- Understand the specific semantics and programming patterns
  - We assume a programming knowledge in other languages
- Updated to ES6 (2015) language features
- Supported by server-side (`Node.js`) and client-side (browsers) run-time environments
  - More recent language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

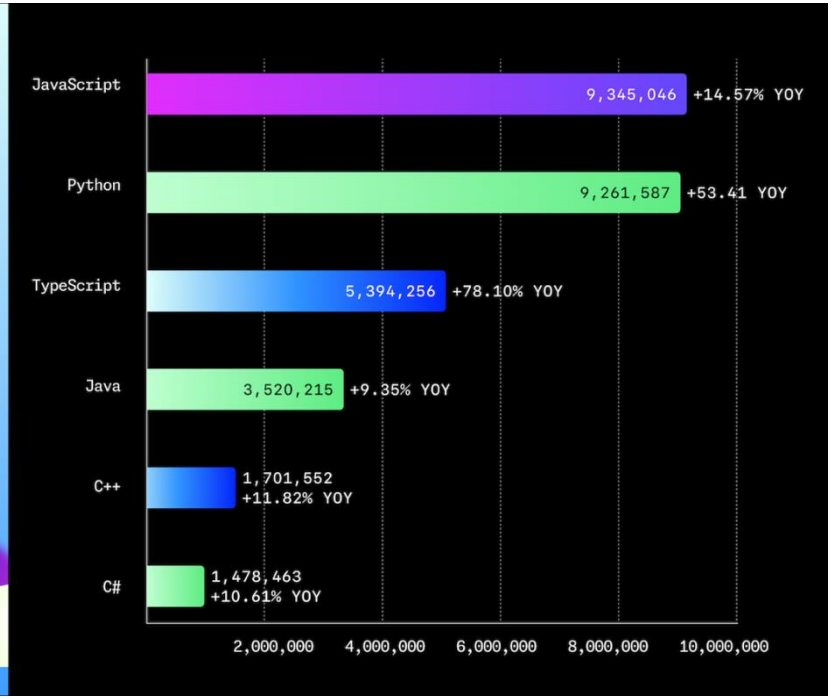
Outline

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

JavaScript – The language of the Web

WHAT IS JAVASCRIPT?





source: https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1

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)** Regexps, Try/Catch, Do-While, String methods

10 yrs

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

Main target

  - **ES6 (June 2015)** Classes, Modules, Arrow Fs, Generators, Const/Let, Destructuring, Template Literals, Promise, Proxy, Symbol, Reflect — Also: ES2015
  - **ES7 (June 2016)** Exponentiation operator (**) and Array Includes — Also: ES2016

ES9, ES10, ...

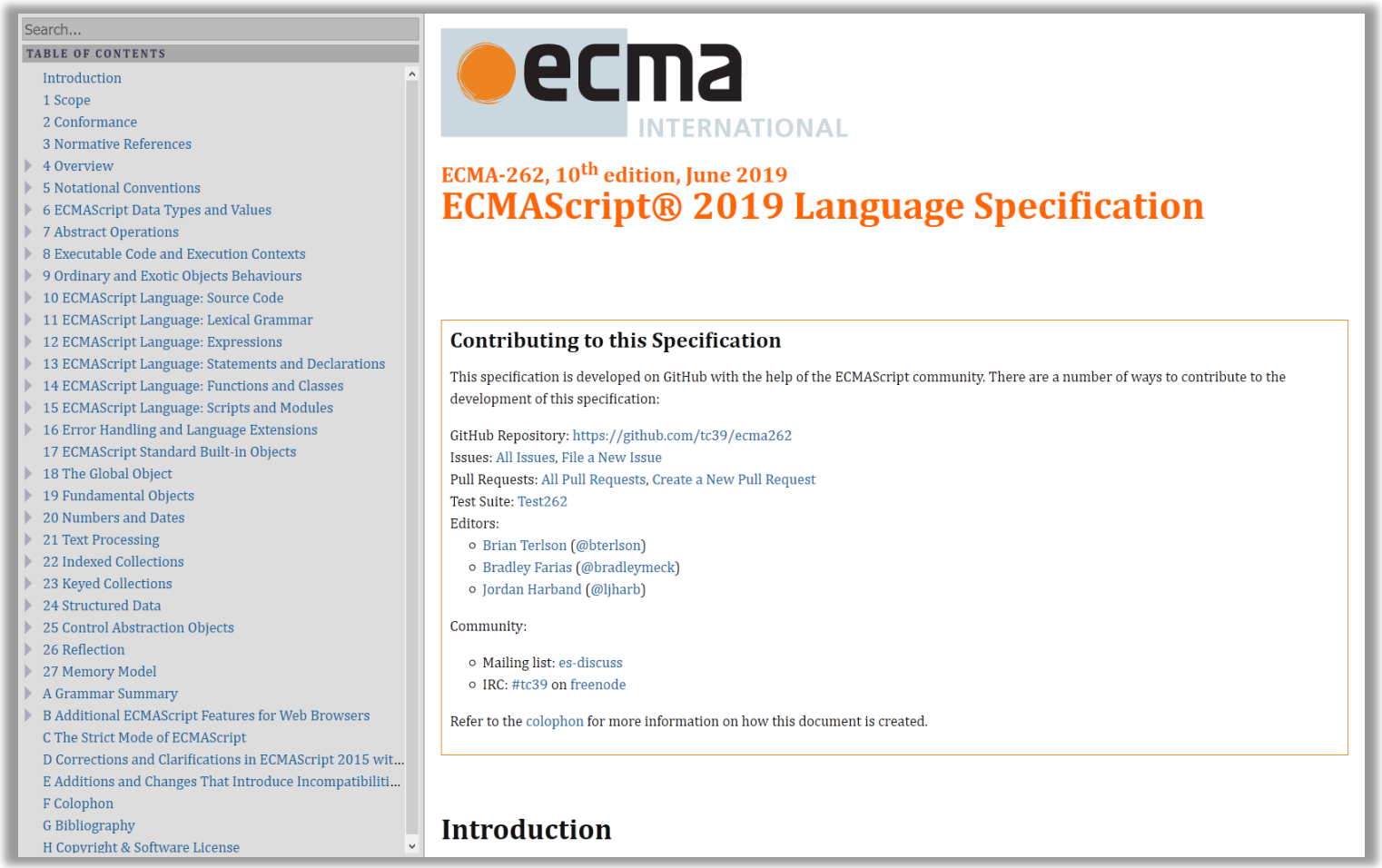
  - **ES8 (June 2017)** Async Fs, Shared Memory & Atomics — Also: ES2017

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 this course**

Official ECMA standard (formal and unreadable)



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

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

What are we missing?

Full support

No support

Compatibility unknown

Experimental. Expect behavior to change in the future.

Non-standard. Expect poor cross-browser support.

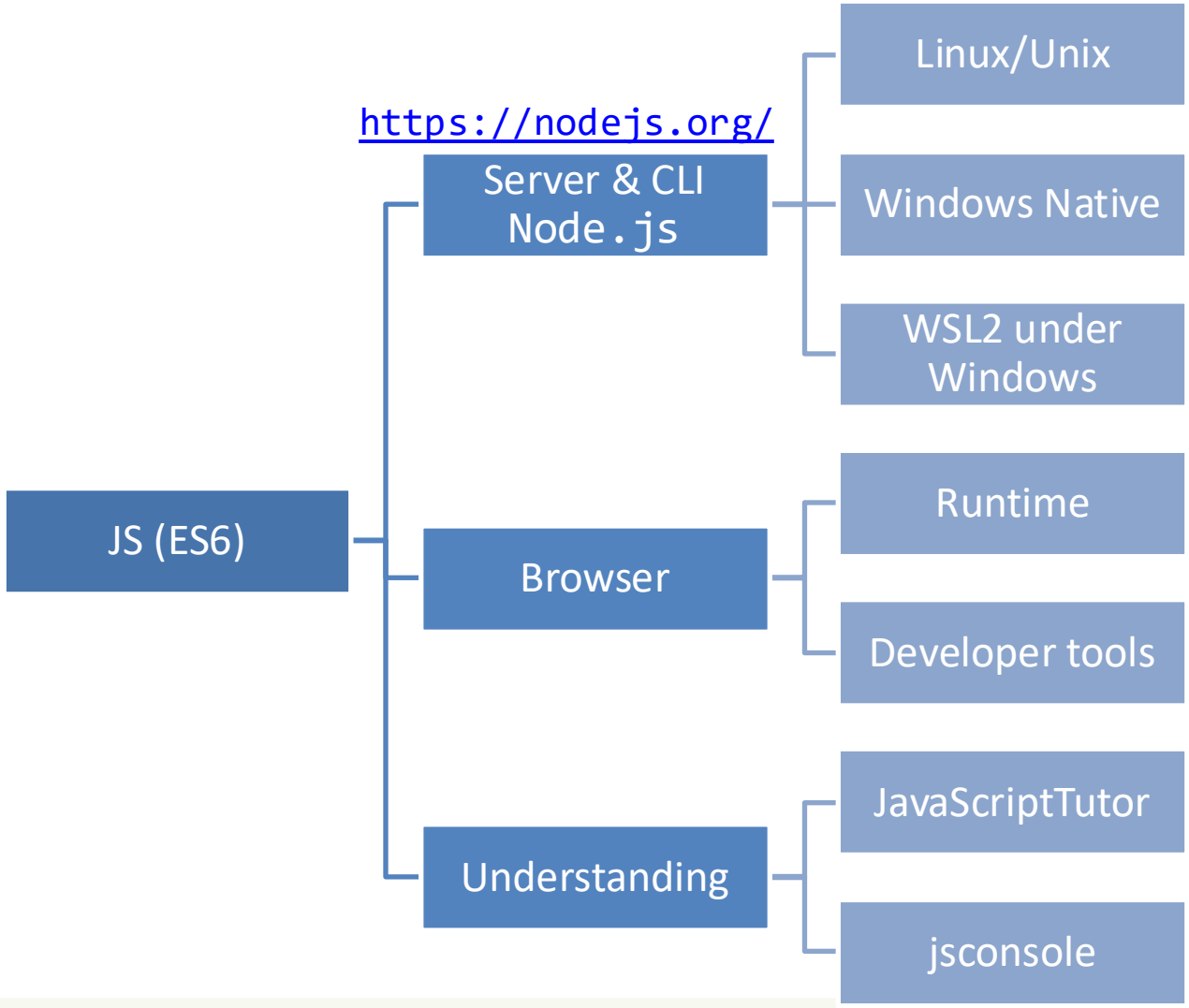
Deprecated. Not for use in new websites.

★ 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



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

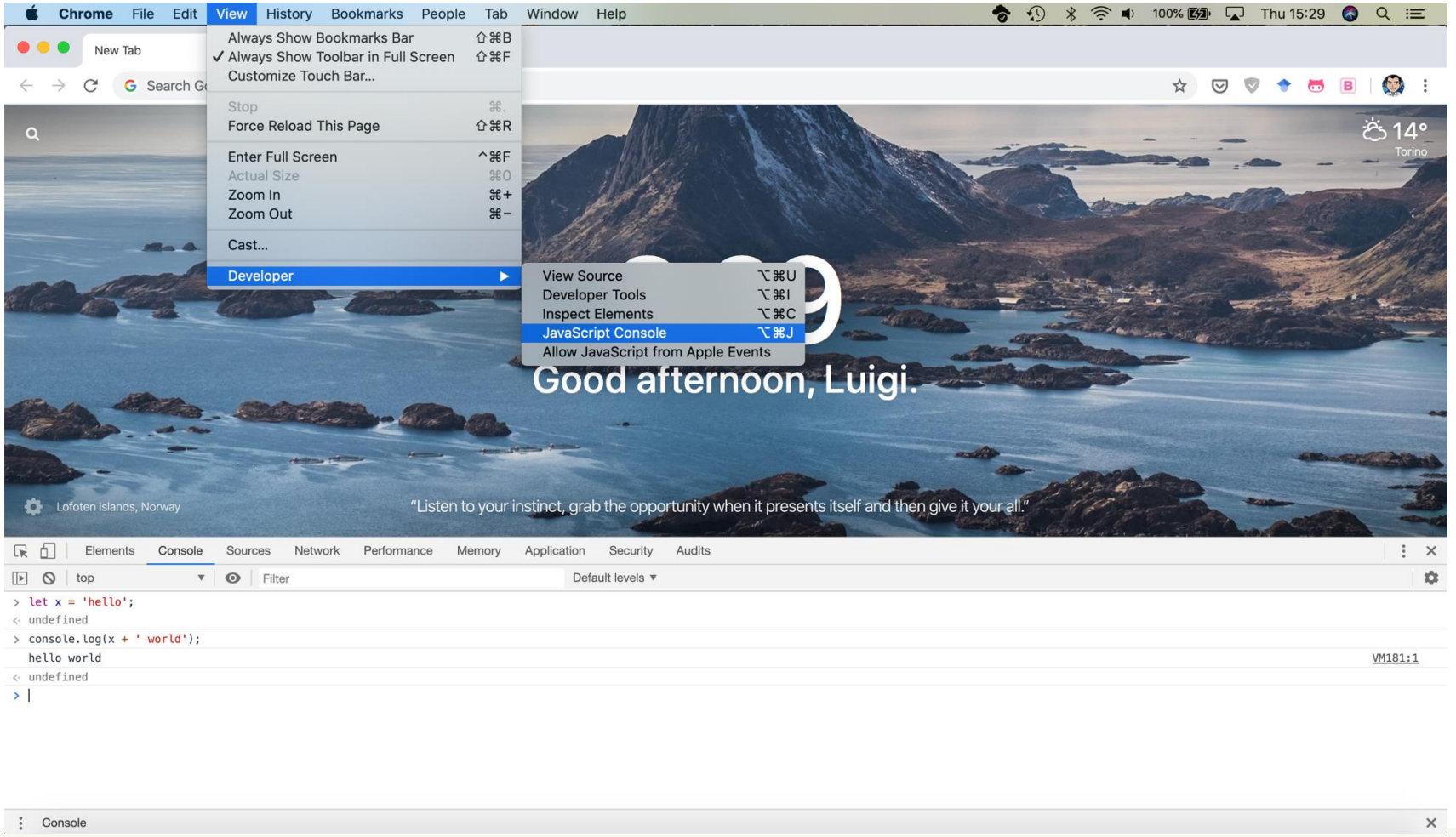
https://docs.microsoft.com/en-us/windows/nodejs/setup-on-windows

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

http://pythontutor.com/javascript.html

https://jsconsole.com/

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 öööö = 'appalled'
> öööö
'appalled'
```

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

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

Semicolon (`;`)

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

Strict Mode

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

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

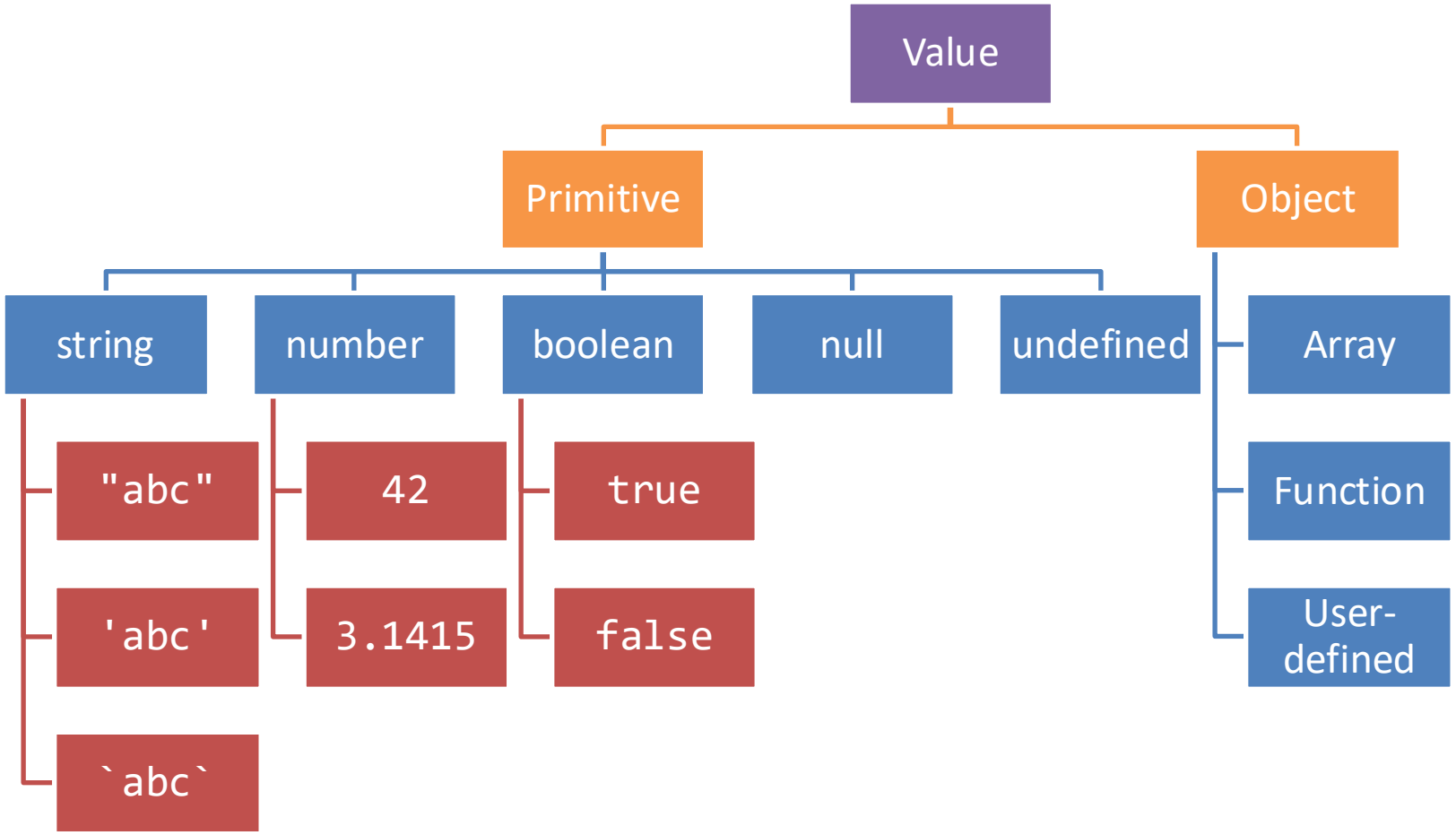
JavaScript: The Definitive Guide, 7th Edition
Chapter 2. Types, Values, and Variables

JavaScript – The language of the Web

TYPES AND VARIABLES

Values and Types

Values have types.
Variables don't.



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

- 'boolean' type with literal values: `true`, `false`
- When converting to boolean
  - The following values are 'falsy'
    - `0`, `-0`, `NaN`, `undefined`, `null`, `''` (empty string)
  - Every other value is 'truthy'
    - `3`, `'false'`, `[]` (empty array), `{}` (empty object)

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

- Booleans and Comparisons
  - `a == b` *// convert types and compare results*
  - `a === b` *// inhibit automatic type conversion and compare results*

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

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

- 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`

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

Variable Declarations

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

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

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

Scope

```
"use strict" ;

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

let a = 5 ; // SyntaxError: Identifier 'a' has already been declared
```

Scope

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

```
"use strict" ;

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

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

console.log(a) ;
```

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

They exist only in their defined and inner scopes.

Scope and Hoisting

```
"use strict" ;

function example(x) {
    let a = 1 ;

    console.log(a) ;    // 1
    console.log(b) ;    // ReferenceError: b is not defined
    console.log(c) ;    // undefined

    if( x>1 ) {
        let b = a+1 ;
        var c = a*2 ;
    }

    console.log(a) ; // 1
    console.log(b) ; // ReferenceError: b is not defined
    console.log(c) ; // 2
}

example(2) ;
```

```
var c ; // hoisted
```

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

Mozilla Developer Network
JavaScript Guide » Expressions and operators

JavaScript – The language of the Web

EXPRESSIONS

Operators

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

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

Assignment

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

Name	Shorthand operator	Meaning
Assignment	`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`

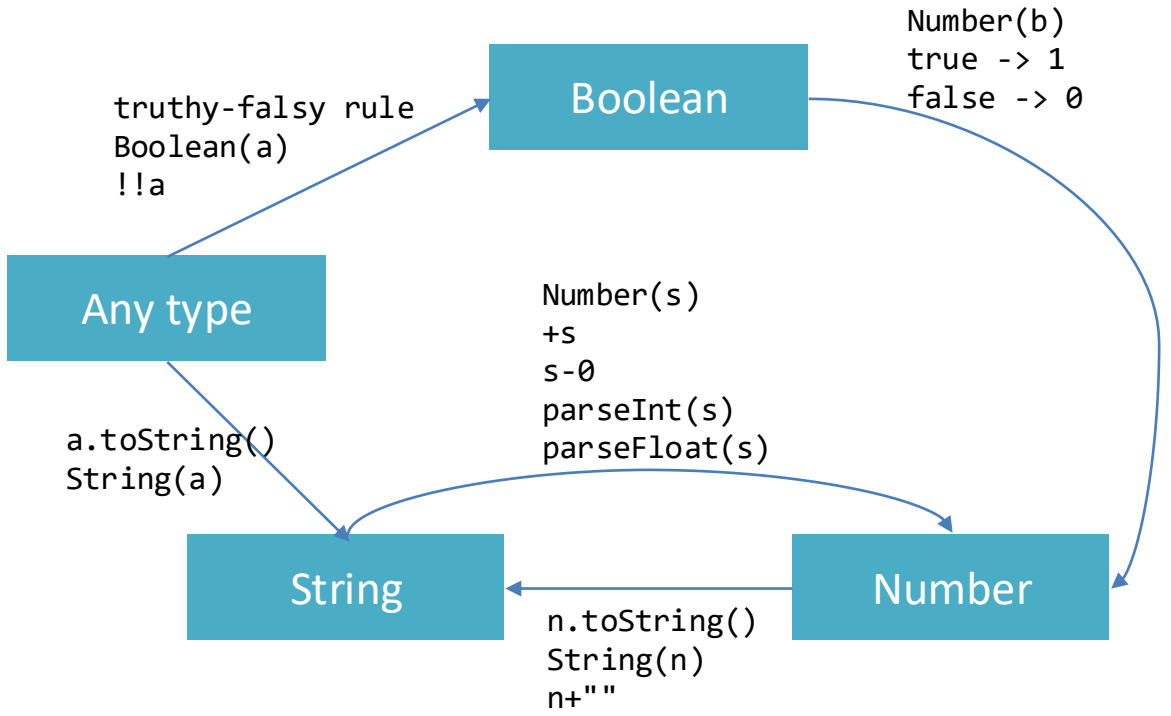
Comparing Objects

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

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

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

> a===b
false

> a==b
false
```

Automatic Type Conversions

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



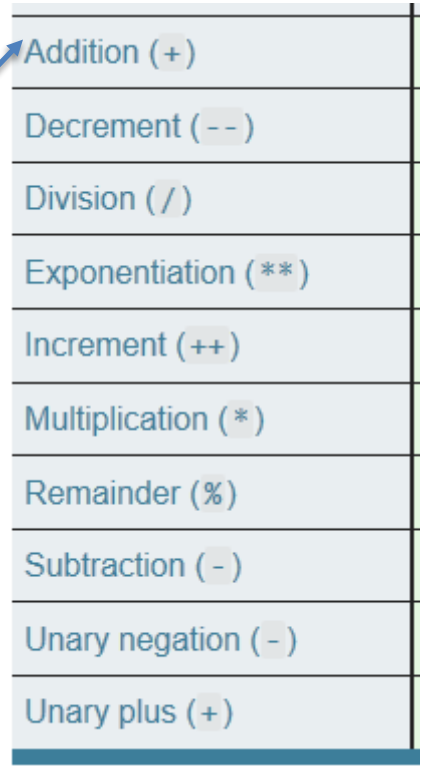
https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/types-grammar/ch4.md

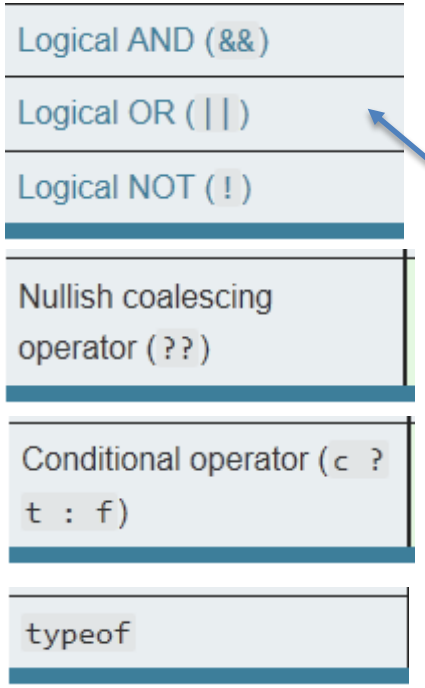
Logical operators

Operator	Usage	Description
Logical AND (`&&`)	`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





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

Mathematical functions (`Math` global object)

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

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

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

JavaScript – The language of the Web

CONTROL STRUCTURES

Conditional statements

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

if truthy (beware!)

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

May also be a string

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

Loop statements

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

Usually declares loop variable

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

May use `break;` or `continue;`

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

Special '`for`' statements

Preferred

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

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

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

```
x
y
```

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

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

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

```
4
7
```

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

```
h
i
```

Other iteration methods

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

Exception handling

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

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

Exception object

```
throw object ;
```

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

Contain fields: name, message

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

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

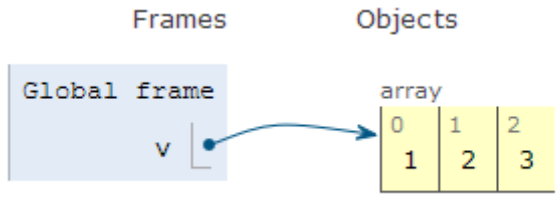
Elements are indexed at positions 0...length-1

Do not access elements outside range

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

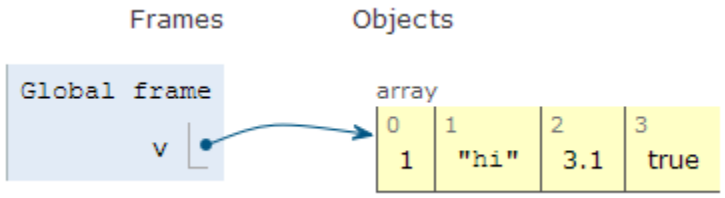
```
let v = [1, 2, 3] ;
```

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




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

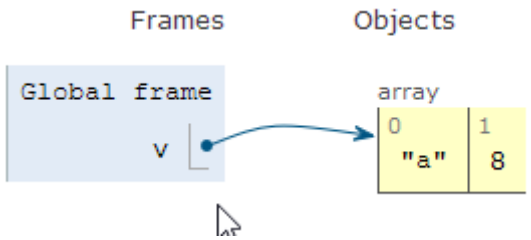
```
let v = Array.of(1, "hi",
3.1, true) ;
```



Adding elements

`.lenght` adjusts automatically

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

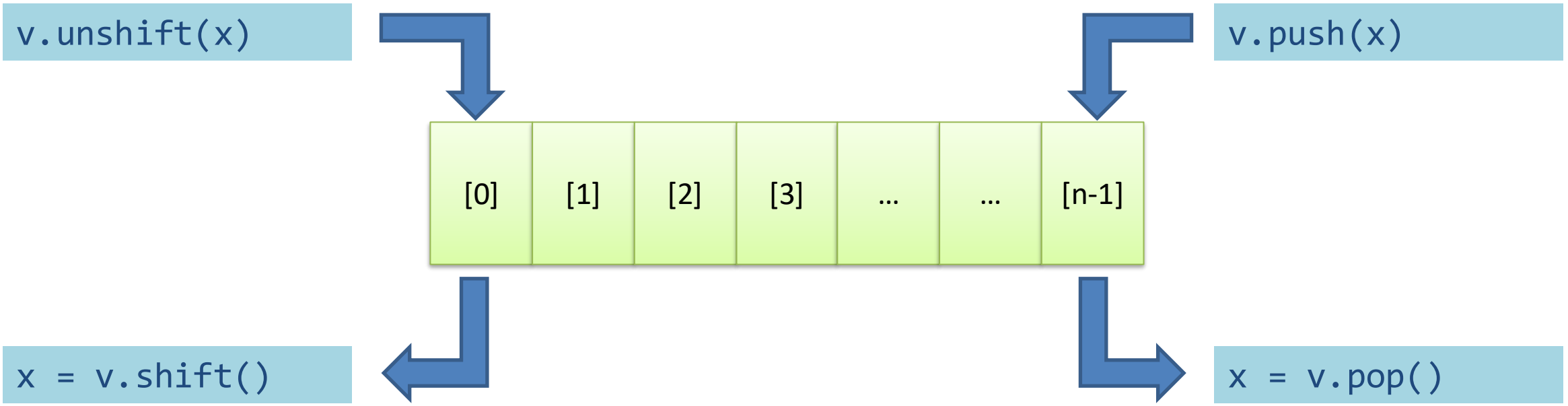



```
let v = [] ;
v.push("a") ;
v.push(8) ;
v.length // 2
```

`.push()` adds at the end of the array

`.unshift()` adds at the beginning of the array

Adding and Removing from arrays (in-place)



Copying arrays

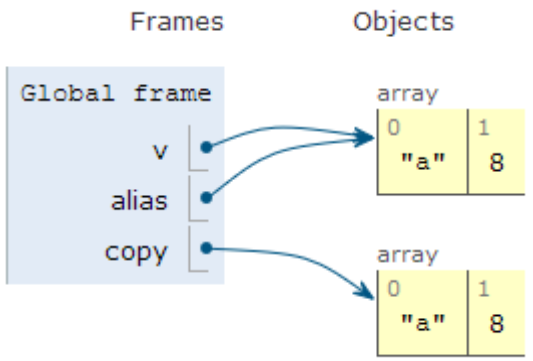
```
let v = [] ;
v[0] = "a" ;
v[1] = 8 ;

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

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

?

Copying arrays

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

let alias = v ;
let copy = Array.from(v) ;
```



Array.from creates a *shallow copy*

Creates an array from any iterable object

Iterating over Arrays

Preferred

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

Functional style – **later**

Main array methods

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

Destructuring assignment

- Value of the right-hand side of equal 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 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]`

Frequent idiom

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

Mozilla Developer Network
JavaScript Guide » Text Formatting

JavaScript – The language of the Web

STRINGS

Strings in JS

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

String operations

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

String methods

Method	Description
`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.

Unicode issues

- Strings are a sequence of 16-bit Unicode ‘code units’
  - Fine for all Unicode characters from 0000 to FFFF
  - Characters (‘graphemes’) from 010000 to 10FFFF are represented by *a pair of code units* (and they occupy 2 index positions)
  - Therefore, not all string methods work well with Unicode characters above FFFF (e.g., emojis, flags, …)

- For more details: https://dmitripavlutin.com/what-every-javascript-developer-should-know-about-unicode/

Template literals

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

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

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

License

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