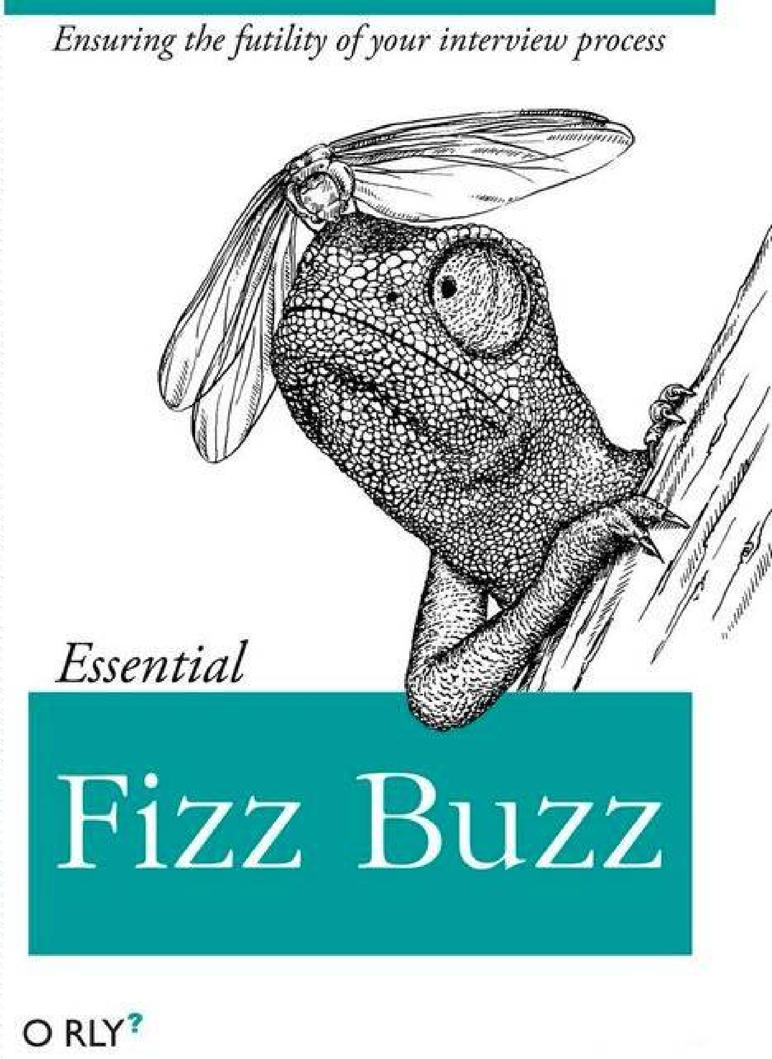
Introduction to JavaScript

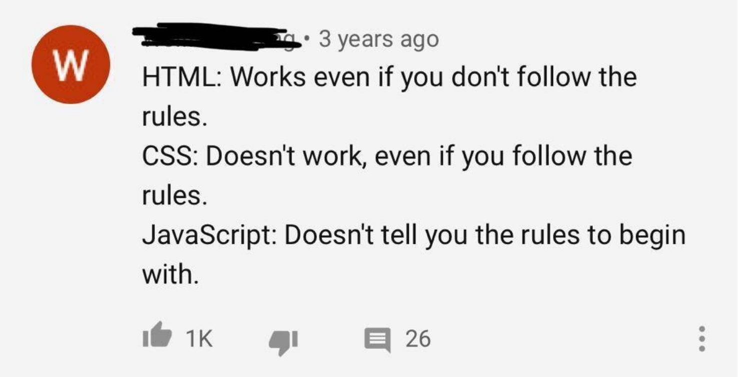
or why 0.1+0.2 is not equal to 0.3
and that is a good thing

Frontend Junior Program - 2021



Agenda

1. ECMA-262
2. Structure
3. Data Types
4. Operators
5. Loops



[redacted] • 3 years ago

HTML: Works even if you don't follow the rules.
CSS: Doesn't work, even if you follow the rules.
JavaScript: Doesn't tell you the rules to begin with.

ECMA-262

JavaScript

JS is defined as OOP in the Standard

ECMAScript is an **object-oriented programming** language for performing computations and manipulating computational objects within a host environment.

JavaScript is a programming language that enables you to create and update Content dynamically, control multimedia, animate images, and pretty much everything else.

In practice:

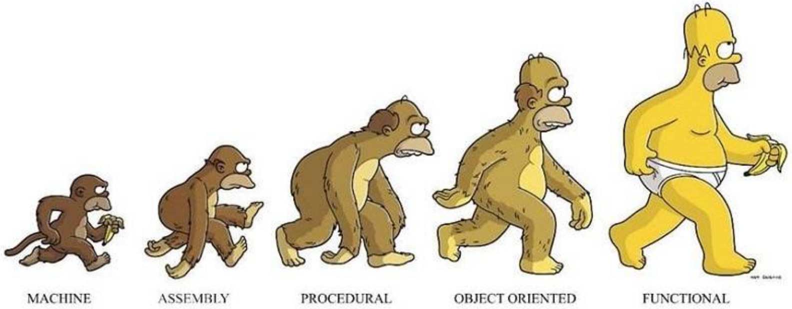
- Imperative
- (Prototype-based) Object-oriented
- Functional
- Asynchronous (since ES2015 (Promises), and ES2017 (async/await))

MDN: JavaScript

Compatibility Matrix

Paradigms in nutshell

```
function imperative(input) {
  let done = firstDoThis(input);
  return nextDoThat(done);
}


class ObjectOriented {
  constructor(input) {
    this.done = input;
  }

  doFirst() {
    this.done = firstDoThis(this.done);
  }

  doNext() {
    return nextDoThat(this.done);
  }
}


let functional = input => nextDoThat(firstDoThis(input));
```



ECMAScript?

JavaScript === ECMAScript

Since publication of the first edition in 1997, ECMAScript has grown to be one of the world's most widely used general-purpose programming languages. It is best known as the language embedded in web browsers but has also been widely adopted for server and embedded applications.

ECMAScript is based on several originating technologies, the most well-known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser.

The development of the ECMAScript Language Specification started in November 1996. The first edition of this ECMA Standard was adopted by the ECMA General Assembly of June 1997.

ECMAScript® 2020 Language Specification

ECMAScript!

The language described by the standard is called “ECMAScript”, not JavaScript. A different name was chosen because Sun (now Oracle) had a trademark for the latter name. The “ECMA” in “ECMAScript” comes from the organization that hosts the primary standard.

The original name of that organization was ECMA, an acronym for European Computer Manufacturers Association.

The initial all-caps acronym explains the spelling of ECMAScript. In principle, JavaScript and ECMAScript mean the same thing.

How JavaScript was created

History

ECMAScript 6 === ES6 === ECMAScript 2015

ECMAScript 1 (June 1997): First version of the standard.

ECMAScript 2 (June 1998): Small update to keep ECMA-262 in sync with the ISO standard.

ECMAScript 3 (December 1999): Adds many core features – “[…] better regular expressions, string handling, new control statements [do-while, switch], try/catch exception handling, […]”

ECMAScript 4 (abandoned in July 2008): Would have been a massive upgrade (with static typing, modules, namespaces, and more), but ended up being too ambitious and dividing the language’s stewards.

ECMAScript 5 (December 2009): Brought minor improvements – a few standard library features and strict mode.

ECMAScript 5.1 (June 2011): Another small update to keep Ecma and ISO standards in sync.

ECMAScript 6 (June 2015): A large update that fulfilled many of the promises of ECMAScript 4. This version is the first one whose official name – ECMAScript 2015 – is based on the year of publication.

ECMAScript 2016 (June 2016): First yearly release. The shorter release life cycle resulted in fewer new features compared to the large ES6.

…

Timeline of ECMAScript versions

Standardization - TC39 Committee

TC39 is the committee that evolves JavaScript.

Starting from 2016, there are yearly releases, but the standardization is an ongoing process, browsers adopts the features continuously.

Ecma International, Technical Committee 39 - ECMAScript

Evolution

New versions are always completely backward compatible.

Old features aren’t removed or fixed. Instead, better versions of them are introduced.

If aspects of the language are changed, it is done inside new syntactic constructs. That is, you opt in implicitly.

no more pain…

```
let can = { I: { use: () => "yes!" } }
```

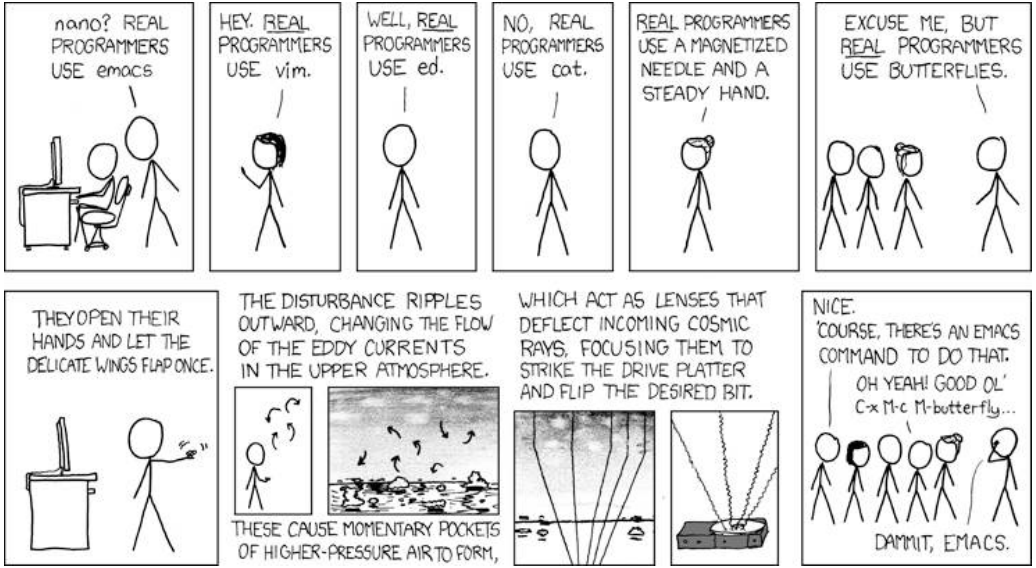
```
can && can.I && can.I.use() ➡ can?.I?.use?.()
```

TC39 is the committee that **evolves** JavaScript.

Fun Fun Function: Optional Chaining Operator in JavaScript

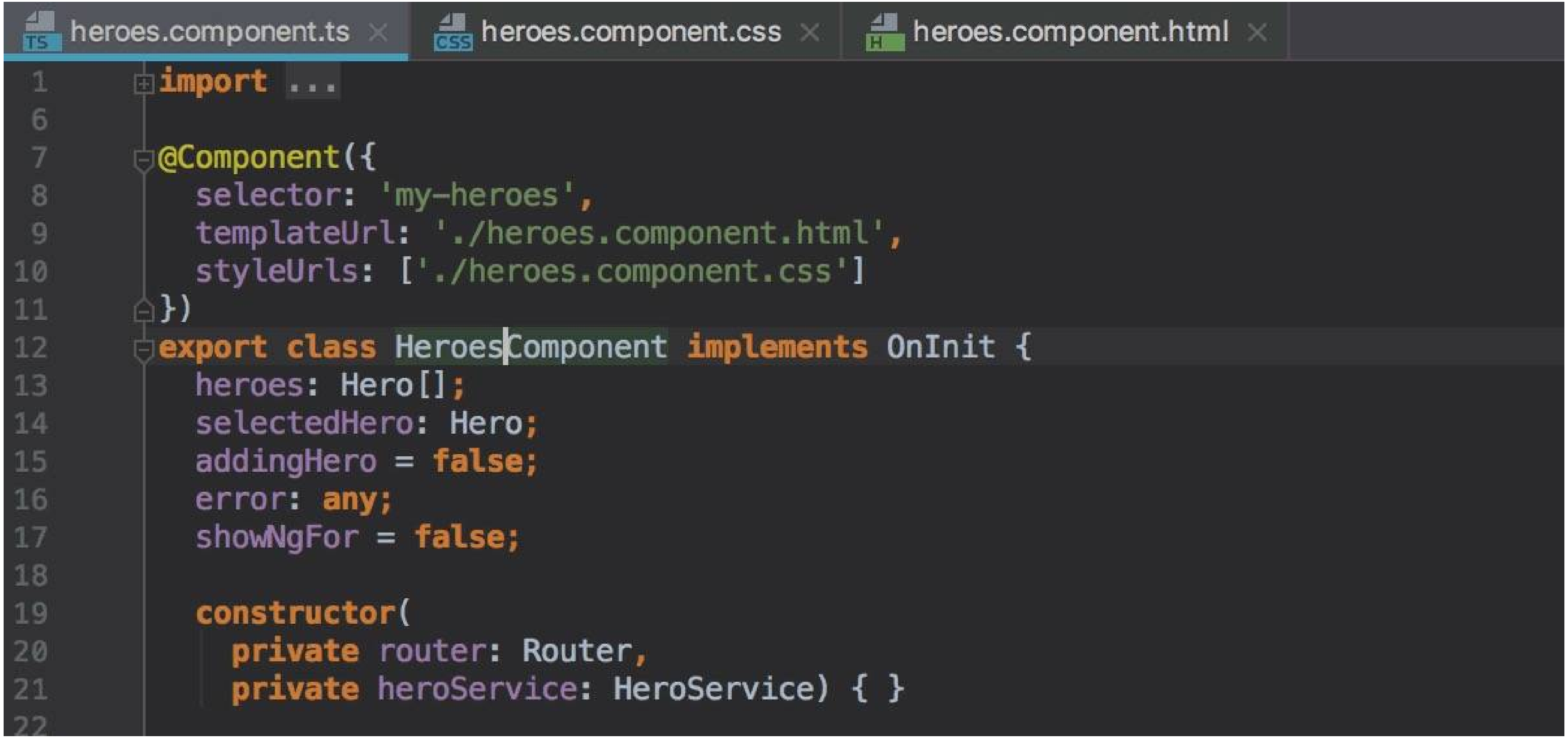
JavaScript Editors - What to look for

- Strong ES2015+ support
  - Autocompletion
  - Parse ES6 imports
  - Report unused imports
  - Automated refactoring
- Framework IntelliSense
- Built-in terminal
- Linter integrations
- Git integration
- Extension support


nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.
THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.
THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.
THESE CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM,
WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.
NICE.
'COURSE, THERE'S AN EMACS COMMAND TO DO THAT.
OH YEAH! GOOD OL' C-x M-c M-butterfly...
DAMMIT, EMACS.

real programmers use butterflies!

IntelliSense in action



```
import ...

@Component({
  selector: 'my-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  addingHero = false;
  error: any;
  showNgFor = false;

  constructor(
    private router: Router,
    private heroService: HeroService) { }
```

Editors

Atom

WebStorm

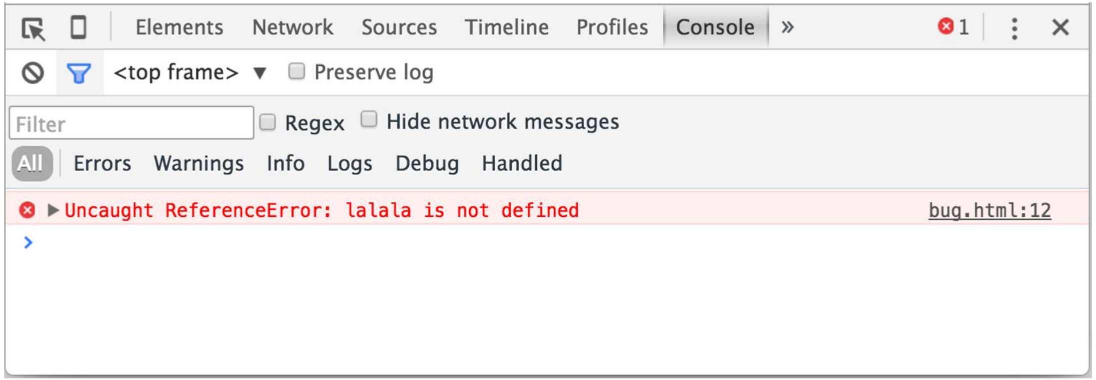
Visual Studio Code

What editor do we use?

Predominantly, 2 editors: VSCode and WebStorm. Usually, you can use either, however, there could be specific projects (where we can use only supported applications), or stubborn team leads, who could politely push you to one direction or other, just to create a consistent setup and workflow to keep the productivity on a high level.

The codebases are usually very complex and could require a professional tool to be able to work effectively.

Developer console



```
▶ Uncaught ReferenceError: lalala is not defined
```

Google Chrome

Press F12 or, if you’re on Mac, then Cmd + Opt + J.

Firefox, Edge and others

Most other browsers use F12 to open developer tools.

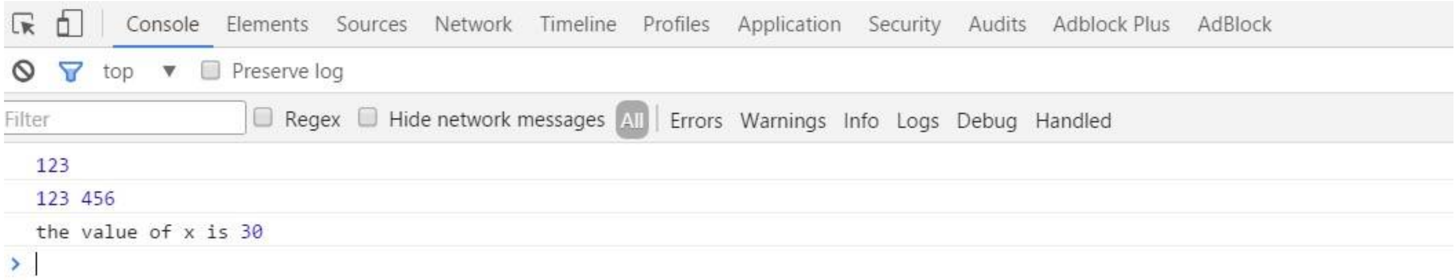
JS output and debugging

For JavaScript value output you can use console.log or alert function.

```
console.log(123);
console.log(123, 456);
var x = 30;
console.log('the value of x is', x);
alert(123);
alert('the value of x is');
```

In production code these are not allowed – the linter should break the build with these.

You can execute JS directly in console, open dev tools (F12) and go in "Console" tab.

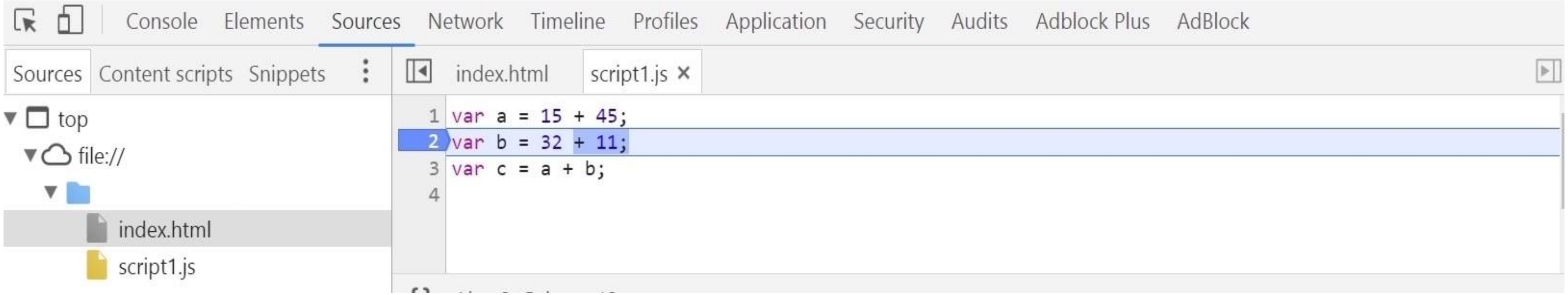


```
123
123 456
the value of x is 30
>
```

JS output and debugging

You can stop execution of JS and take a look what is going on there.

Open dev tools and go in 'Sources' tab.



```
var a = 15 + 45;
var b = 32 + 11;
var c = a + b;

```

In production:

- use Sentry or Datadog or similar error trackers, report aggregators and monitoring tools
- remove `console.log`s on production-grade deployments (e.g., with a Webpack plugin)

HOW TO START

<script> element

```
<script>
  document
    .getElementById("demo")
    .innerText = "My First JavaScript";
</script>
```

with inline script

JavaScript programs can be inserted in any part of an HTML document with the help of the <script> element.

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
```

importing external scripts

- separates HTML and JS: easier to maintain
- cached JavaScript files can speed up page loads

HTML Living Standard: The script element

More about: defer/async

"use strict"

```
'use strict';
var v = 'Hello!';

function strict() {
  return "Hi! I'm a strict mode function! ";
}
```

- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- As an example, in normal JavaScript, mistyping a variable name creates a new globa strict mode, this will throw an error, making it impossible to accidentally create a gl variable.
- In normal JavaScript, a developer will not receive any error feedback assigning value writable properties.
- In strict mode, any assignment to a non-writable property, a getter-only property, a property, a non-existing variable, or a non-existing object, will throw an error.

Expressions and statements

JavaScript distinguishes expressions and statements.
An expression produces a value and can be written wherever a value is expected, for example as an argument in a function call.

A statement performs an action. Loops and if statements are examples of statements.

```
// expression
2 - 1;
true;
```

results in a value

```
// statement
const user = false;
if (user) {
    welcomeMessage('hello dear user');
} else {
    warning('please signIn');
}
```

does something, which has a side effect

Definition

More about: Keywords and reserved w

A variable is a named container for a value.

The name that refers to a variable is called an identifier.

variable name
value

Variables

Variable names start with a letter, underscore (_), or dollar sign ($).

Subsequent characters can also be digits (0-9). ISO 8859-1 or Unicode letters.

In practice, we use camelCase for variables.

Valid examples

Number_hits

Temp99

$credit

and _name

userName

Declaration, Initialization and Assignment

DECLARATION

Registers a variable in the corresponding scope

⬇

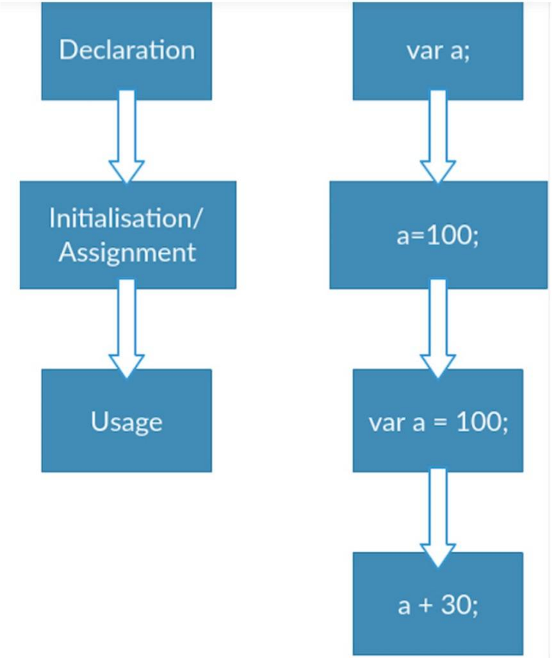
INITIALIZATION

Allocates memory for the variable

⬇

ASSIGNMENT

Assigns a specified value to the variable



Declaration Types

`var`

Declares a variable, optionally initializes it with value.

By default every variable gets undefined as value.

`let`

Declares a block-scoped, local variable, optionally initializing it to a value.

`const`

Declares a block-scoped, read-only named constant.

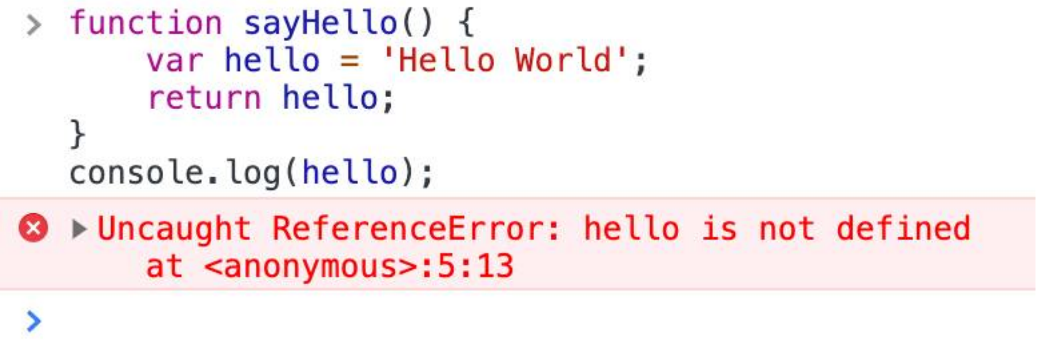
var

Variables declared with var are available in the scope of the enclosing function.
If there is no enclosing function, they are available globally.

```
var x;                // Declaration and implicit initialization (with undefined)
x = 'Hello World’;    // Assignment
var y = 'Hello World’;// All in one
```

var

This will cause ReferenceError, as the variable hello is only available within the sayHello function.



```
> function sayHello() {
      var hello = 'Hello World';
      return hello;
  }
  console.log(hello);
✕ ▶Uncaught ReferenceError: hello is not defined
      at <anonymous>:5:13
>
```

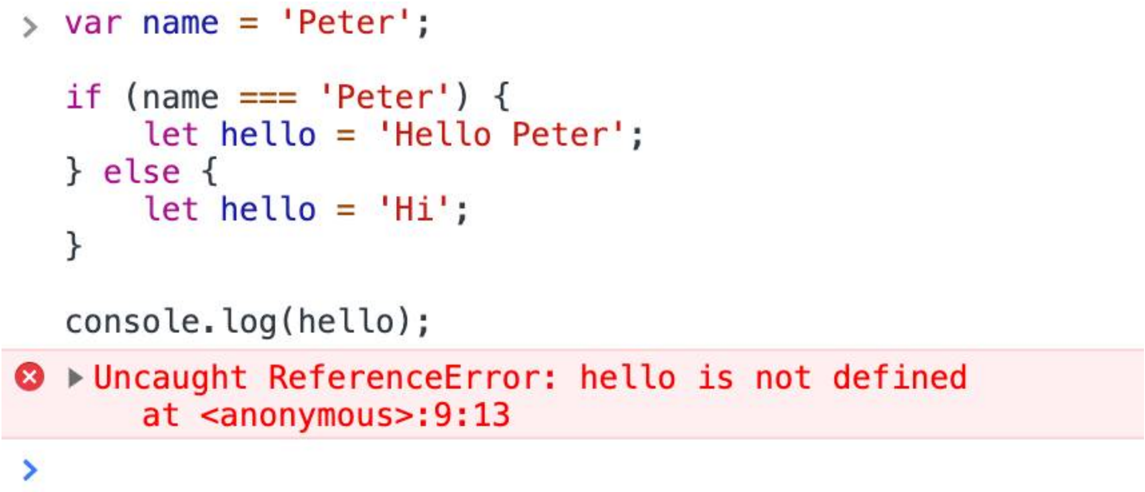
let

Its scope is not only limited to the enclosing function, but also to its enclosing block statement. A block statement is everything inside { and }, (e.g., an if condition or loop). The benefit of let is that it reduces the possibility of errors, as variables are only available within a smaller scope.

```
let x;                    // Declaration
x = 'Hello World’;        // Assignment
let y = 'Hello World’;    // All in one
```

let

This will cause an ReferenceError as hello is only available inside the enclosing block – in this case the if condition.



```
> var name = 'Peter';

  if (name === 'Peter') {
      let hello = 'Hello Peter';
  } else {
      let hello = 'Hi';
  }

  console.log(hello);
✖ ▶Uncaught ReferenceError: hello is not defined
      at <anonymous>:9:13
>
```

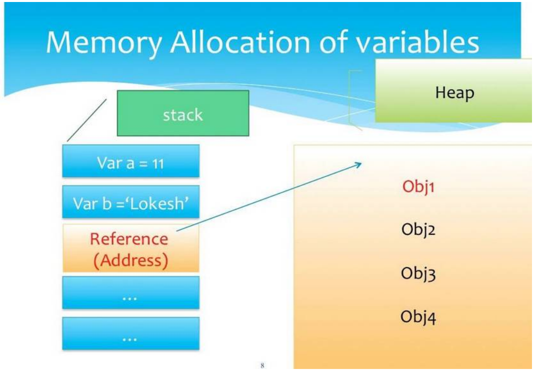
const

Variables declared via const are immutable.

A const is limited to the scope of the enclosing block, like let. Constants should be used whenever a value must not change during run time.

It must be initialized immediately:

```
const x = 'Hello World’;
```



Here, the variable is a reference (like a pointer to a memory address)**, and while it is immutable, the content** (the value of the referred memory address)**, is not!**

```
const obj = {
  a: 1
};
```

```
obj.b = 2;
```

Accidental global creation

All of the above named declarations can be written in the global context (i.e. outside of any funct
Even within a function, omitting var, let or const makes the variable automatically global.

```
function sayHello() {
  hello = "Hello World";

  return hello;
}

sayHello();

console.log(hello);
```

! To avoid accidentally declaring global variables you can use strict mode, but linters will catch it

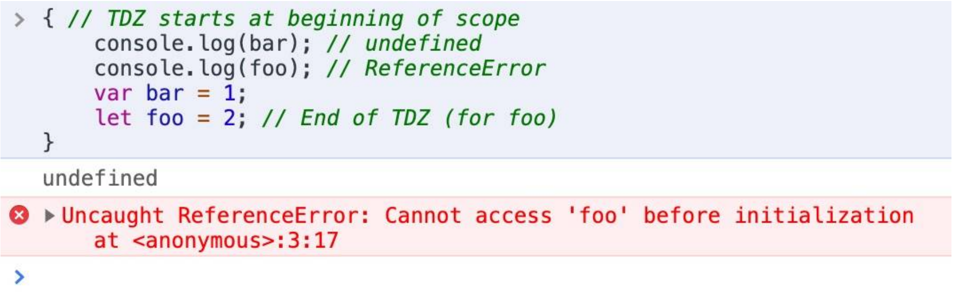
Hoisting and the Temporal Dead Zone

Variable declarations will always internally be hoisted (moved) to the top of the current scope.

```
console.log(hello);
var hello;
hello = "I'm a variable";
```

```
var hello;
console.log(hello);
hello = "I'm a variable";
```

Variable declarations with let and const though won't get initialized with undefined,
thus the Temporal Dead Zone!



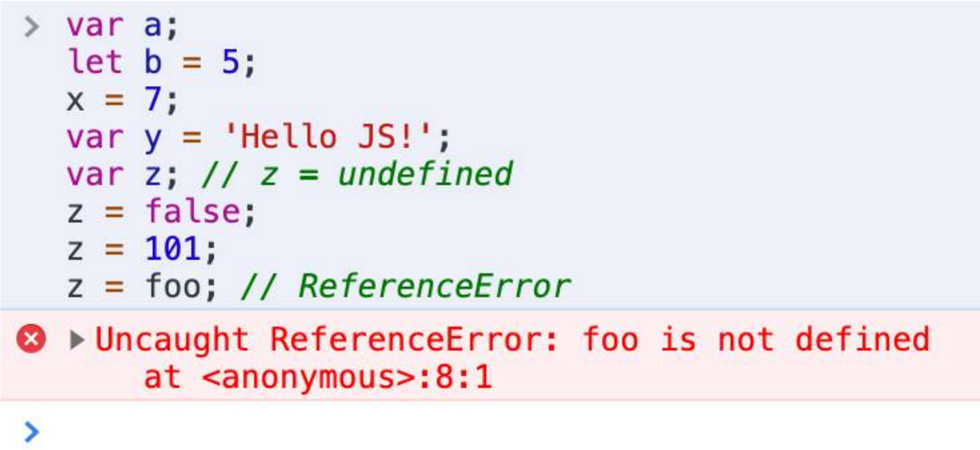
```
> { // TDZ starts at beginning of scope
      console.log(bar); // undefined
      console.log(foo); // ReferenceError
      var bar = 1;
      let foo = 2; // End of TDZ (for foo)
  }
  undefined
✖ ▶Uncaught ReferenceError: Cannot access 'foo' before initialization
      at <anonymous>:3:17
>
```

MDN: Temporal dead zone (T

Evaluating variables

A variable declared using the var or let keyword with no assigned value specified has the value of undefined.

An attempt to access an undeclared variable will result in a ReferenceError exception being thrown.



```
> var a;
  let b = 5;
  x = 7;
  var y = 'Hello JS!';
  var z; // z = undefined
  z = false;
  z = 101;
  z = foo; // ReferenceError
✖ ▶Uncaught ReferenceError: foo is not defined
      at <anonymous>:8:1
>
```

Example

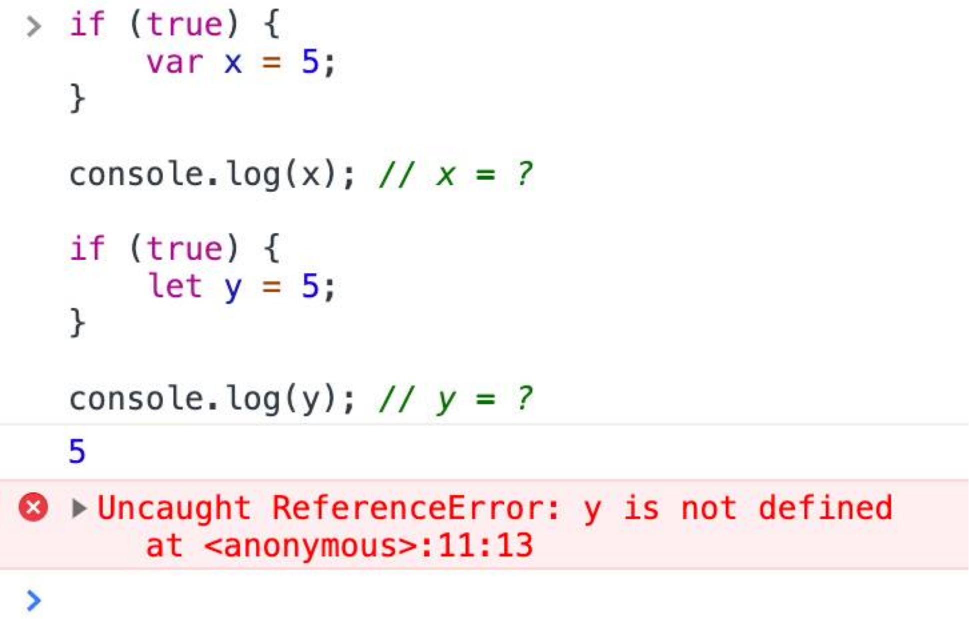
```
var i, $, p, q;

i = -1;
for (i = 0; i < 10; i += 1) {
    $ = -i;
}

if (true) {
    p = 'FOO';
} else {
    q = 'BAR';
}
```

What is the value of i, $, p, and q after executi

When the program runs, all variable declaratio are moved up to the top of the current scope.

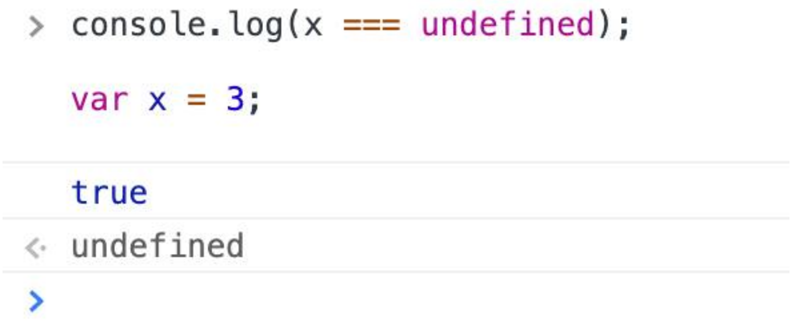
Variable scope

More about: Motivation for block scop



```
> if (true) {
      var x = 5;
  }

  console.log(x); // x = ?

  if (true) {
      let y = 5;
  }

  console.log(y); // y = ?
  5
⊗ ▶Uncaught ReferenceError: y is not defined
      at <anonymous>:11:13
>
```

Variable hoisting



```
> console.log(x === undefined);

  var x = 3;
  true
<· undefined
>
```

Variable declarations are hoisted, initializations / values are not!

What to use? var / let / const?

The answer is simple: usually you don’t even have a chance to c

The var will be forbidden on projects to prevent issues with hois

If you don’t change the value of a declared variable (let) afterw then the linter will warn you, and you must change to const.

Function hoisting

Function declarations will be hoisted.

```
foo(); // "bar"

function foo() {
  console.log('bar');
}
```

Function expressions will not.

More precisely, the baz variable will be hoisted, but its value will be undefined.

```
baz(); // TypeError: baz is not a function

var baz = function() {
  console.log('bar2');
};
```

IIFE

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
(function (param) {
    <statements>
})(param);
```

Some typical usecases:

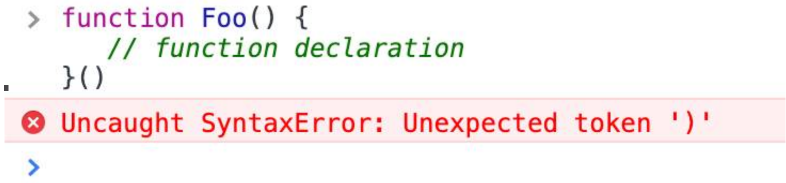
- library code
- as poor man’s block scope
- closures and private data
- capturing the global object

```
/** @license React v17.0.1
* react.development.js
*
* Copyright (c) Facebook, Inc. and its affiliates.
*
* This source code is licensed under the MIT license found in the
* LICENSE file in the root directory of this source tree.
*/
(function (global, factory) {
  typeof exports === 'object' && typeof module !== 'undefined’
    ? factory(exports)
    : typeof define === 'function' && define.amd
      ? define(['exports'], factory)
      : (global = global || self, factory(global.React = {}));
}(this, (function (exports) { 'use strict';
```

The first lines of the React code, do you recognise the pattern

IIFE – how does it work?

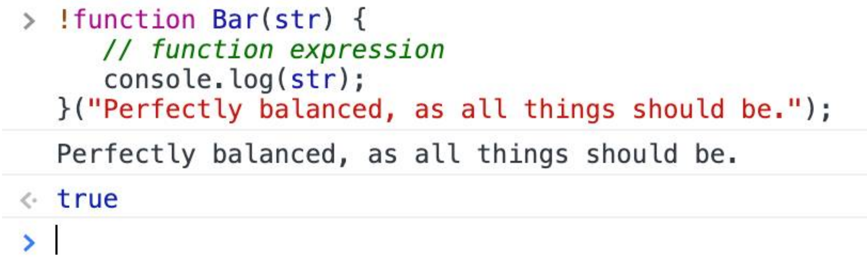
It does not work with function declarations.



```
> function Foo() {
      // function declaration
  }()
⊗ Uncaught SyntaxError: Unexpected token ')'
>
```

any operator will turn it to an expression,
still, use parentheses for clarity

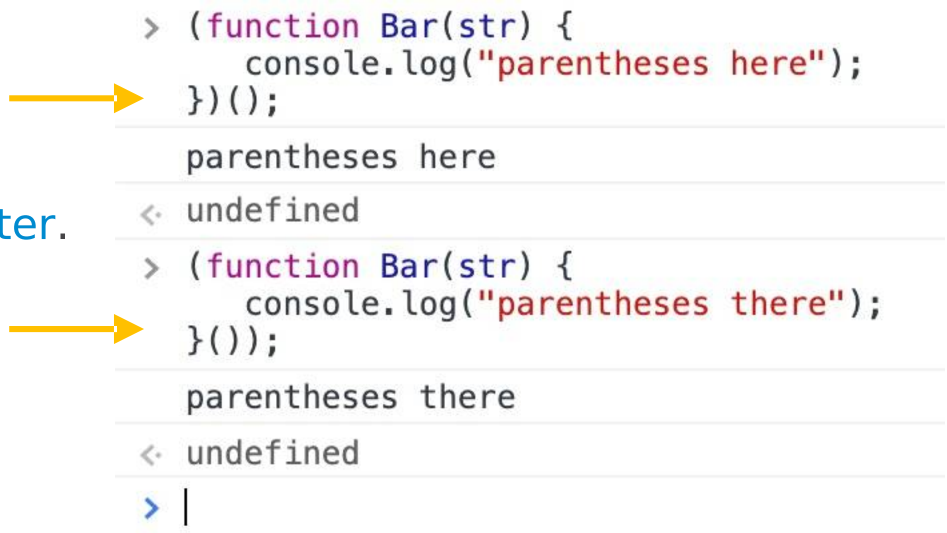
We need a function expression here.



```
> !function Bar(str) {
      // function expression
      console.log(str);
  }("Perfectly balanced, as all things should be.");
  Perfectly balanced, as all things should be.
<· true
> |
```

IIFE – variants

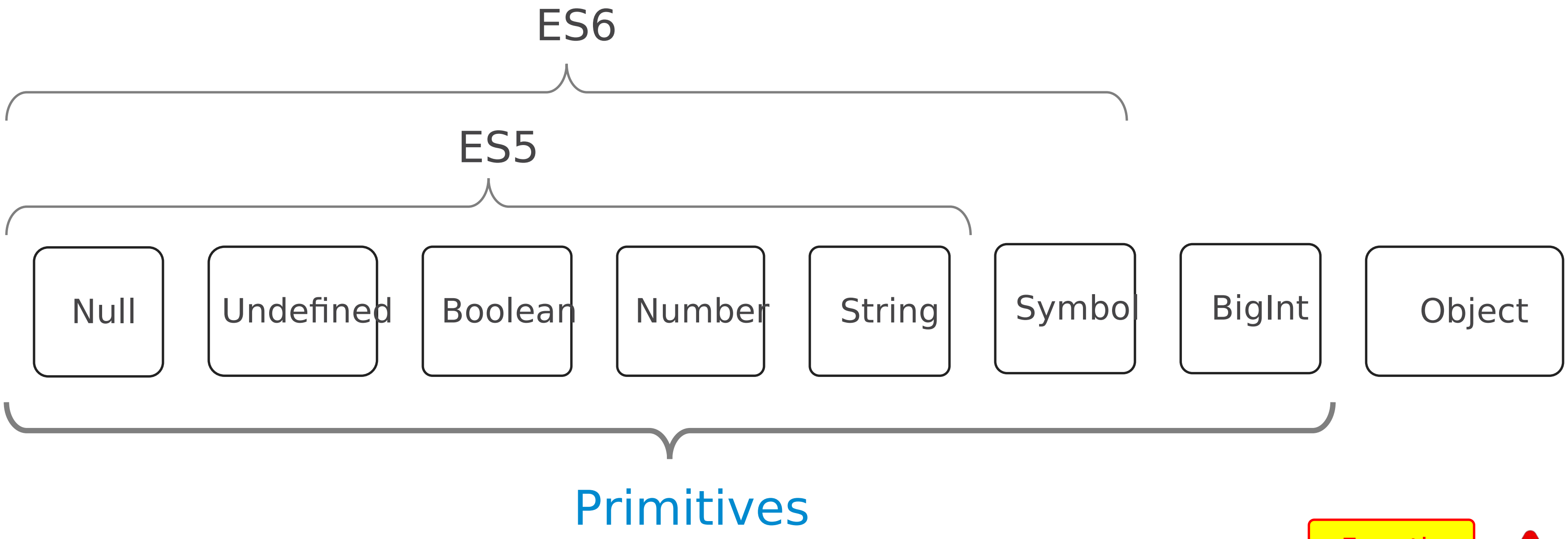
Even the placement of the parentheses does not matter.



```
> (function Bar(str) {
     console.log("parentheses here");
  })();
  parentheses here
<· undefined
> (function Bar(str) {
     console.log("parentheses there");
  }());
  parentheses there
<· undefined
> |
```

DATA TYPES

Types in JavaScript (ES11, ES2020)



ES6

ES5

Null

Undefined

Boolean

Number

String

Symbol

BigInt

Object

Primitives

Function

ECMA-262: ECMAScript Language Types

Function is **not a JavaScript type**, function is object!

typeof operator

The typeof operator evaluations, according to the standard.

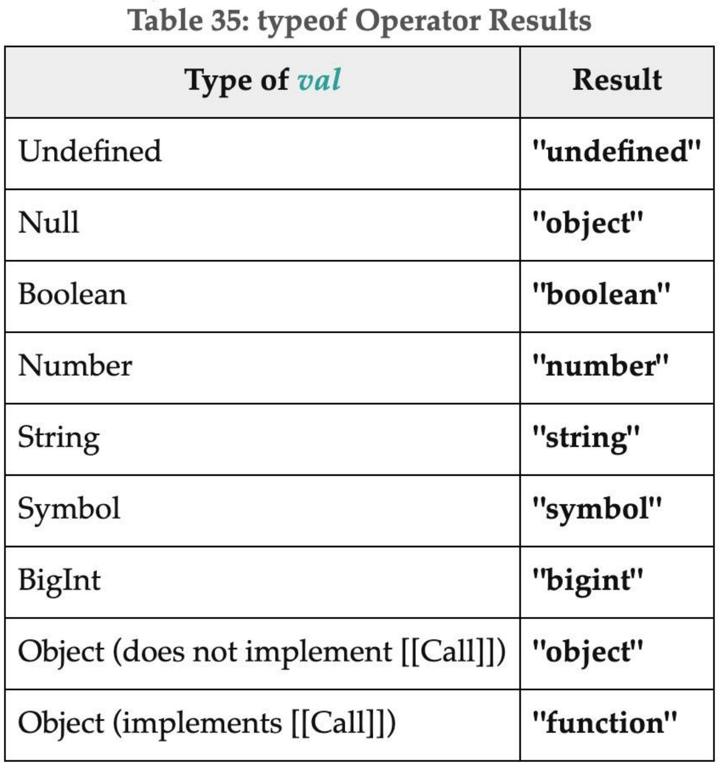


Table 35: typeof Operator Results

Type of *val*	Result
Undefined	**"undefined"**
Null	**"object"**
Boolean	**"boolean"**
Number	**"number"**
String	**"string"**
Symbol	**"symbol"**
BigInt	**"bigint"**
Object (does not implement [[Call]])	**"object"**
Object (implements [[Call]])	**"function"**

Function is a callable object.

[ECMA-262: The typeof Operator]

Primitive values

Null	null				
Undefined	undefined				
Boolean	true	false			
String	“”	“\n”	‘Hello’	“ℵℑℜ ”	
Number	01101	.45	-Infinity	NaN	5e-14

Symbol and BigInt will be discussed in the upcoming lectures.

Number

More about: IEEE 754

```
let n = 536;
n = 3.1415;
n = 0.1 + 0.1 + 0.1; // != 0.3

console.log(1/0, -1/0, 0/0);
```

The number type is represented as a 64-bit floating-point number (52 bit mantissa + 1
bit exponent + a sign bit).
Infinity represents the mathematical Infinity ∞.

What every (JavaScript) developer should not about floating numbers
IEEE 754 floating point calculator

NaN

How to end up with NaN?

- Division of zero by zero
- Dividing an infinity by an infinity
- Multiplication of an infinity by a zero
- Any operation in which NaN is an operand
- Converting a non-numeric string or undefined into a number

NaN is unordered

```
NaN < 1; // false
NaN > 1; // false
NaN == NaN; // false
// But we can still check for NaN:
isNaN(NaN); // true
isNaN(true); // false
isNaN(false); // false
```

isNaN converts the argument to a Number and returns true if the resulting value is Na
Number.isNaN does not convert the argument; it returns true when the argument is a
is NaN.

To Number

```
parseInt(string, radix);
```

```
parseInt(' 0xF', 16);
parseInt(' F', 16);
parseInt('17', 8);
parseInt(021, 8);
parseInt('015', 10);
parseInt(15.99, 10);
parseInt('1111', 2);
parseInt('15*3', 10);
parseInt('12px', 10);
```

```
parseFloat(value);
```

```
parseFloat(3.14);
parseFloat('3.14');
parseFloat('314e-2');
parseFloat('0.0314E+2');
parseFloat('3.14more');
```

```
Number(object);
```

```
Number('12.3');
Number('');
Number('0x11');
Number('0b11');
Number('0o11’);
Number();
Number(null);
Number(true);
```

Important: always use the radix*!

*the base in mathematical numeral systems: for decimal system the radix is 10; octal:

toString

```
const n = 255;
alert(n.toString(16));
2.toString(); // SyntaxError
2..toString(); // the second point is correctly recognized
2 .toString(); // note the space left to the dot
(2).toString(); // 2 is evaluated first
```

boolean, undefined, null

boolean, consists of the primitive values true and false.

```
const amIAlwaysRight = true;
let areYouAlwaysRight = false;
```

undefined, used when a variable has not been assigned a value

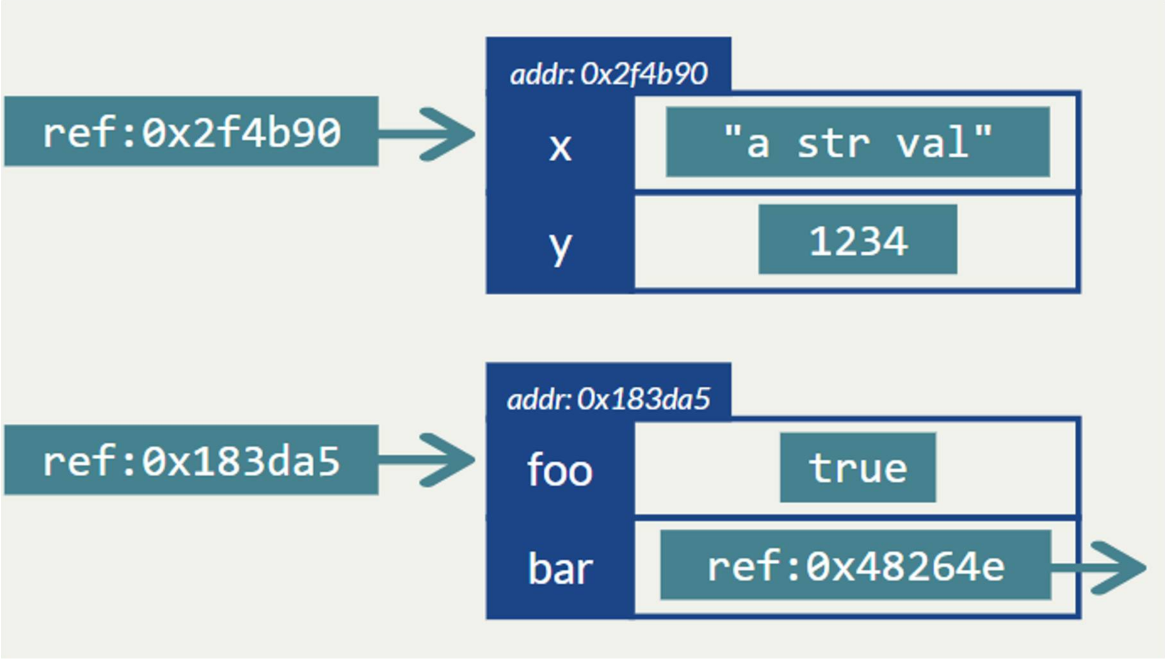
```
let foo;
console.log(foo);
console.log(window.bar);
console.log(bar);
```

null, represents the intentional absence of any object value

```
const age =null;
```

object

Any value of this type is a reference to some object (key-value pairs, associative arrays). Reference is in the stack, object data is in the heap.



Data type conversion

Converting to strings

Converting to numbers

Converting to boolean

To boolean conversion

It happens in logical operations, but also can be performed manually with the call of **Boolean(va**

Argument Type	**Result**
Undefined	false
Null	false
Boolean	The result equals the input argument (no conversion).
Number	The result is false if the argument is +0, −0, or NaN; otherwise, the res is true.
String	The result is false if the argument is the empty String (its length is zero otherwise, the result is true.
Object	true

To boolean conversion

```javascript
console.log(!!'0');
console.log(!!' ');
console.log(0 == '\n0\n');
console.log(!!undefined);
console.log(!!null);
console.log(!!'');
console.log(!!NaN);
console.log(!!{});
console.log(!![]);
```

To number conversion

Numeric conversion happens in mathematical functions and expressions automatically, or by call
Number(argument).

Argument Type	**Result**
Undefined	**NaN**
Null	**+0**
Boolean	The result is **1** if the argument is **true**. The result is **+0** if the argument is **false**.
Number	The result equals the input argument (no conversion).
String	Through parsing (parseInt/parseFloat and Number ruleset)
Object	Apply the following steps: - Let primValue be ToPrimitive(input argument, hint Number). - Return ToNumber(primValue).

To number conversion

```
let a = +'123';
console.log(+' \n  123 \n \n');
console.log(+true);
console.log(+false);
console.log('\n0 ' == 0);
console.log('\n' == false);
console.log('1' == true);
```

Equality comparision and sameness

To string conversion

String conversion happens when we need the string form of a value, or by calling String(argumer

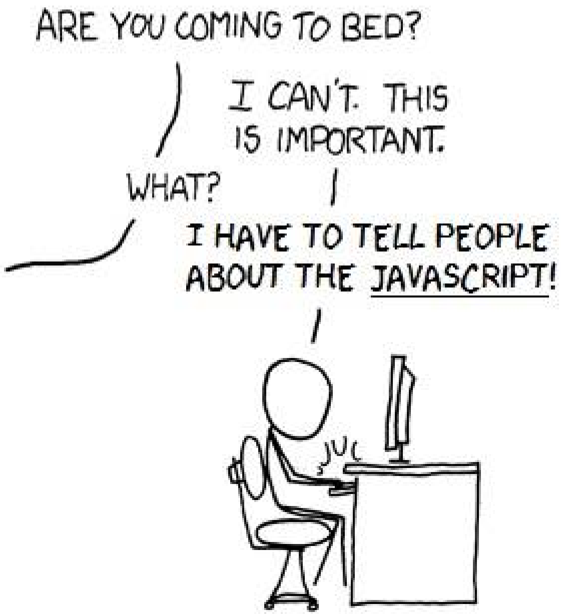
Argument Type	**Result**
Undefined	"undefined"
Null	"null"
Boolean	If the argument is **true**, then the result is "true". If the argument is **false**, then the result is **"**false**".**
Number	If m is **NaN**, return the String "NaN". If m is **+0** or **−0**, return the String "0". If m is less than zero, return the String concatenation of the String "-" and ToString(- If m is infinity, return the String "Infinity".
String	Return the input argument (no conversion)
Object	Apply the following steps: 1. Let primValue be ToPrimitive(input argument, hint String). 2. Return ToString(primValue).

To string conversion

```
console.log(true + 'test');
console.log('123' + undefined);
console.log('123' + {});
console.log(123 + 123);
console.log('123' + 123);
console.log([] + 1);
console.log([1] + 1);
console.log([1, 2] + 1);
console.log('\n' === false);
console.log('\n' == false);
console.log('Hi' == false);
```

Special values

```
console.log(null >= 0);      // true
console.log(null > 0);       // false
console.log(null == 0);      // false
console.log(undefined > 0); // false
console.log(undefined == 0);// false
console.log(undefined < 0); // false
console.log(NaN == NaN);     // false
console.log(NaN === NaN);    // false
console.log(undefined == null);// false
```


ARE YOU COMING TO BED?
I CAN'T. THIS IS IMPORTANT.
WHAT?
I HAVE TO TELL PEOPLE ABOUT THE JAVASCRIPT!

Refer to Abstract Relational Comparision Algo

Equality comparisions and samen

OPERATORS

Operators

- Arithmetic operators
- Assignment operators
- Logical operators
- Comparison operators
- Bitwise operators
- Bitwise logical operators
- Bitwise shift operators

Arithmetic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder of a division)
++	Increment
--	Decrement

Examples

```
console.log(100 + (4 * 11) / 2 - 1); //121

var i = 20;
console.log(++i); // 21
console.log(i); // 21
console.log(i--); // 21
console.log(i); // 20

var x = 25;
x += 20; //(x = x + 20);
console.log(x); //45
console.log(5 / 0); // Infinity
console.log(0 / 0); // NaN

typeof Infinity; // 'number'
typeof NaN; // 'number'
```

Assignment operators

Operator	Description
=	Assign
+=	Add and assign. For example, x+=y is the same as x=x+y.
-=	Subtract and assign. For example, x-=y is the same as x=x-y.
*=	Multiply and assign. For example, x*=y is the same as x=x*y.
/=	Divide and assign. For example, x/=y is the same as x=x/y.
%=	Modulus and assign. For example, x%=y is the same as x=x%y.

Comparison operators

Operator	Description
==	Is equal to
===	Is identical (is equal to and is of the same type)
!=	Is not equal to
!==	Is not identical
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Examples

Comparsions produce boolean values.

```
console.log(10 > 20); // false
console.log(10 < 20); // true
console.log(10 >= 20);
console.log(10 <= 20);
console.log(10 === 20);
console.log(10 !== 20);
console.log(10 == '10') // true
console.log(10 === '10'); // false
console.log(null == undefined) // true
console.log(null === undefined) // false
```

Logicaloperators

Operator	Description
&&	and
||	or
!	not

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

OR and Null coalesing operator

```
true || false;
true || true;
Infinity || true;
'\n' || false;
'' || 0 || false;
'' || 1 || 'hi';
null ?? 'hi';
0 ?? 'there'
```

AND

```
true && false;
1 && 2 && 3;
'Hi' && true && null && 1;
```

NOT

```
!true;
!!true;
!!'';
!!'false';
!!false;
!!{};
```

Mixed

```
!{} || ![1] && [] || !+true
0 || !!false && !{} || NaN || !null
```

Type conversion

String conversion

```
const a = true + "test"; // "truetest"
const b = "test" + undefined; // "testundefined"
const c = 123 + ""; // "123"
const d = String(55); // "55"
const e = 123;
e = e.toString(); // "123"
```

Type conversion

Numerical conversion

```
const a = +"123"; // 123
const b = Number("123"); // 123
const c = parseInt("123px", 10); // 123
const d = "5" * "4"; // 20
```

value	convert to
undefined	NaN
null	0
true/false	1/0
String	convert from string by special rules

Type conversion

Logical conversion

The transformation to the “true / false” is a boolean context, such as if (obj), while (obj) and the application of logical operators.

```
const a = Boolean(1); // true
const b = Boolean(0); // false
const c = !!5; // true
```

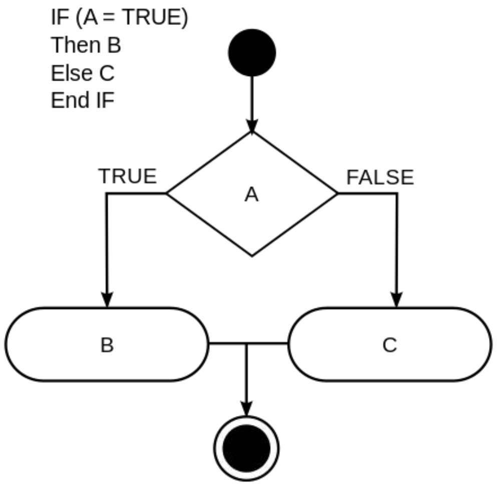
value	convert to
undefined, null	false
Number	all true, 0 - false, NaN - false
String	all true, empty string "" — false
Object	always true

CONDITIONAL EXECUTION

Conditional execution

In conditional execution we choose between two different routes based on a boolean value.

Conditional execution is written with the if keyword in JavaScript. In the simple case, we just want some code to be executed if, and only if, a certain condition holds.


IF (A = TRUE)
Then B
Else C
End IF

if...else

```
if (condition)
    statement1
[else
    statement2]
```

```
if (cipher_char === from_char) {
    result = result + to_char;
    x++;
} else {
    result = result + clear_char;
}
```

```
if (condition1)
    statement1
else if (condition2)
    statement2
...
else
    statementN
```

```
if (x > 5) {
    /* do the right thing */
} else if (x > 50) {
    /* do the right thing */
} else {
    /* do the right thing */
}
```

if

The if (…) statement evaluates the expression in parentheses.

false

The number 0, an empty string, null, undefined and NaN become false.

```
if (0) {
    // 0 is falsy ...
}
```

true

Other values become true, so they are called “truthy”.

```
if (1) {
    // 1 is truthy ...
}
```

If

```
if ([expression]) {
    [statement];
}

const a = 10;
if (a > 3) {
    console.log('it is true');
}

// Bad style, always use braces
if (a > 5) console.log('it is true’);

const user = false;
if (user) {
    console.log('hello dear user');
} else {
    console.log('please signIn');
}
```

No worries, a linter will take care of this, too.

Shorthand assigments

```
const i = 1;
const j = 5;
let result;

// bad approach
if (i !== j) {
    result = true;
} else {
    result = false;
}

// same result, but clean implementation
result = i !== j;
```

This is too complex and error prone.

Never do this! ⚠

else + if

Use the else + if statements to specify a new condition if the first condition is false.

```
if (city === 'Szeged') {
    console.log('Hi Szeged');
} else if (city === 'Debrecen') {
    console.log('Hi Debrecen');
} else if (city === 'Budapest') {
    console.log('Hi Budapest');
} else {
    console.log('Hi Unknown city');
}
```

Conditional (ternary) operator

```
condition ? expressionIfTrue : expressionIfFalse
```

```
const result = Math.PI > 4 ? 'yes' : 'no';
```

```
const salary = 90000;
let reaction = '';

// To define reaction, we can use a simple if statement
if (salary > 50000) {
    reaction = 'Not bad!';
} else {
    reaction = 'Not enough!’;
}

// But I'd rather use this one
reaction = salary > 50000 ? 'Not bad!' : 'Not enough!';
```

Please never do this, this is an expression as an intention, but this is a statement in implementation.

Never mismatch those!

Always do this. This is an expression in both ways.

Switch

The program will jump to the label that corresponds to the value that switch was given, or to default if no matching value is found.

```
switch (expression) {
  case value1:
    // The result of expression matches value1
    [break;]
  case value2:
    // The result of expression matches value2
    [break;]
  ...
  case valueN:
    // ... matches valueN
    [break;]
  [default:
    // Executed when none of the values match
    [break;]]
}
```

```
switch (city) {
    case 'Szeged':
        console.log('Hi Szeged');
        break;
    case 'Debrecen':
        console.log('Hi Debrecen');
        break;
    case 'Budapest':
        console.log('Hi Budapest');
        break;
    default:
        console.log('Hi Unknown city');
        break;
}
```

Switch – falling through

```
const animal = 'Giraffe';
switch (animal) {
    case 'Cow':
    case 'Giraffe':
    case 'Dog':
    case 'Pig':
        console.log("This animal will go on Noah's Ark.");
        break;
    case 'Dinosaur':
    default:
        console.log('This animal will not.');
}
```

fall throughs, this a common solution this way

this is not acceptable, because it contains code!

while default is not a mandatory part, it is mandatory in prod code!

Switch - true

```
switch (true) {
    case weekendDay(date):
        return "8:00am-12:00pm";

    case restDay(date):
        return "Closed";

    default:
        return "8:00am-20:00pm";
}
```

Maybe a bit surprising, however, a good solution f
replacing a long if/else chain.

This structure is much more clear and intention revealing than a complex if/else chain.

Sometimes clear code is not what you would consider as such.

Detecting real code smells requires a bit of learni

Albeit the lack of breaks here, this won’t fall through.

Joel Spolsky: Making Wrong Code Look W

Switch without default?

A common phenomena, that in some cases you feel that the de
is not right there: you just cannot add a meaningful default ca

Always listen to this suspicion, because it could be the clear s
that you should not use switch in that case in the first place

Also, it could be the indication that you are just
going to miss an important edge case.

Anyway, this is always a warning signal, that you must care

Switch vs dictionary object

When not to use switch?

```
switch (city) {
    case 'Szeged':
        console.log('Hi Szeged');
        break;
    case 'Debrecen':
        console.log('Hi Debrecen');
        break;
    case 'Budapest':
        console.log('Hi Budapest');
        break;
}
```

you would miss the default case, which is concerning

Use a dictionary object instead!

```
let welcomeMessages = {
  "Szeged": "Hi Szeged",
  "Debrecen": "Hi Debrecen",
  "Budapest": "Hi Budapest",
}

console.log(cities["Szeged"]);
```

Switch vs dictionary object

“Still not clear, please elaborate this!”

If our goal is to do something (=== statement) then it is a switch.

```
switch (city) {
    case 'Szeged':
        doSmtg();
        break;
    case 'Debrecen':
        doSmtgElse();
        break;
    case 'Budapest':
        doSmtgEvenMore ();
        break;
    default:
        doSmtgInAnyCase();
}
```

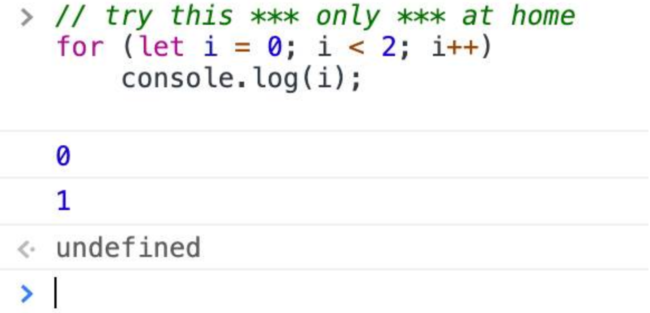
If we are interested in a value (=== expression) then it is a dictionary object.

```
let welcomeMessages = {
  "Szeged": "Hi Szeged",
  "Debrecen": "Hi Debrecen",
  "Budapest": "Hi Budapest",
}

console.log(cities["Szeged"]);
```

LOOPS

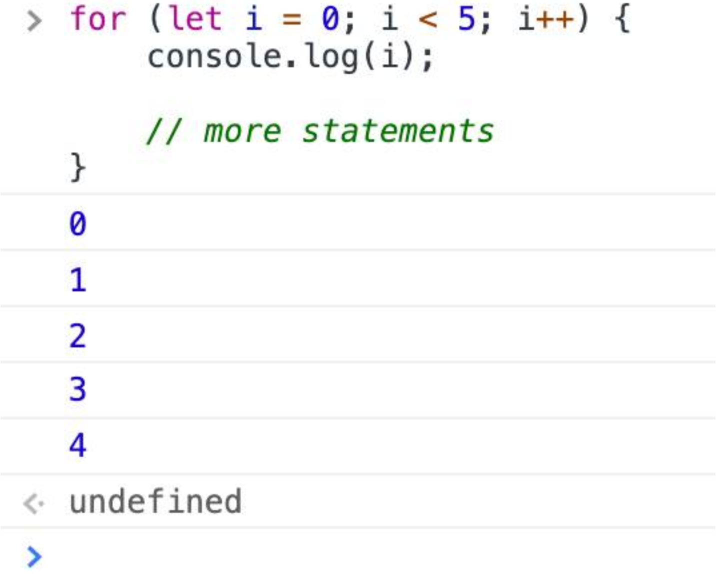
for

```
for ([initialization]; [condition]; [final-expression]) {
  <statement>
}
```



```
> // try this *** only *** at home
  for (let i = 0; i < 2; i++)
      console.log(i);

  0
  1
<· undefined
> |
```

Theoretically, for (and while, if, etc…) can be used without curly braces, however, in reality it can’t.



```
> for (let i = 0; i < 5; i++) {
      console.log(i);

      // more statements
  }
  0
  1
  2
  3
  4
<· undefined
>
```

for with missing parts

```
// we can skip initialization
let i = 5;
for (; i < 10; i++) {
    console.log(i);
}

// we can skip increment
for (; --i;) {
    console.log(i);
}

// finally, we can remove everything
for (; ;) {
    if (i == 0) {
        console.log(i);
        break;
    }
}
```

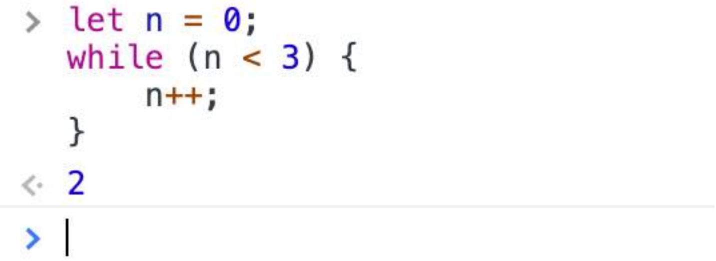
you can use for with missing parts in prod…

… but only once ;)

while, do...while

while

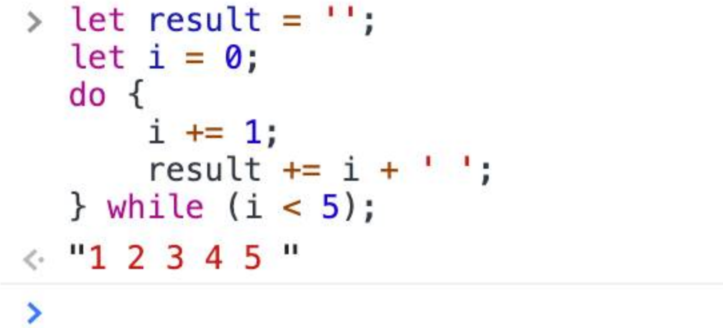
```
while (condition) {
    <statement>
}
```



```
> let n = 0;
  while (n < 3) {
      n++;
  }
<· 2
> |
```

do … while

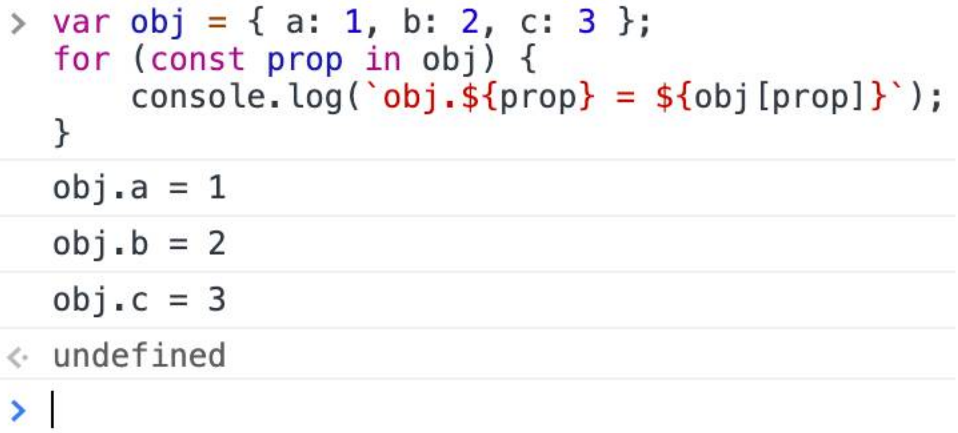
```
do {
    <statement>
} while (condition);
```



```
> let result = '';
  let i = 0;
  do {
      i += 1;
      result += i + ' ';
  } while (i < 5);
<· "1 2 3 4 5 "
>
```

for…in

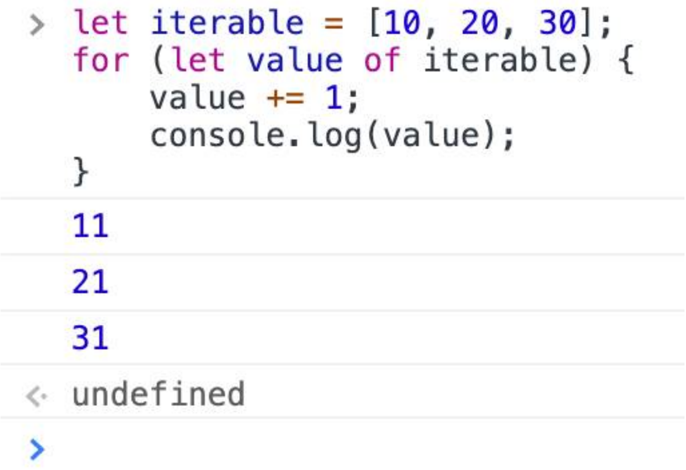
```
for (variable in object) {
    <statement>
}
```



```
> var obj = { a: 1, b: 2, c: 3 };
  for (const prop in obj) {
      console.log(`obj.${prop} = ${obj[prop]}`);
  }
  obj.a = 1
  obj.b = 2
  obj.c = 3
<· undefined
>
```

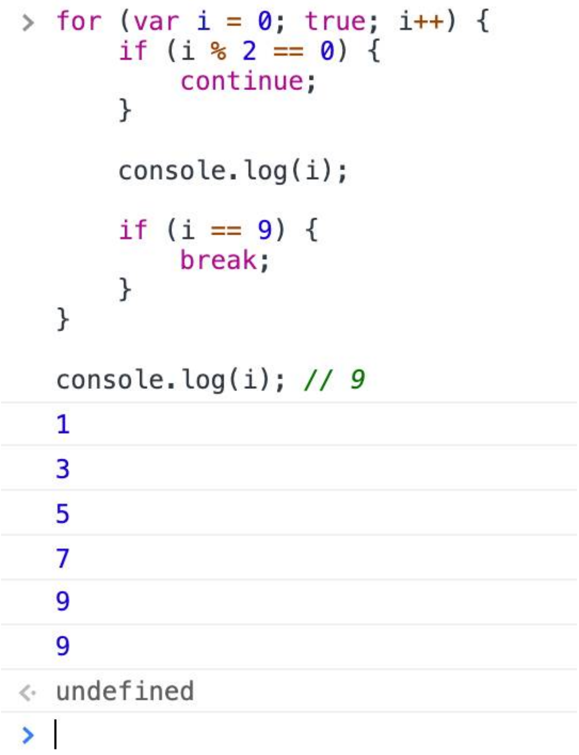
for…of

```
for (variable of iterable) {
    <statement>
}
```



```
> let iterable = [10, 20, 30];
  for (let value of iterable) {
      value += 1;
      console.log(value);
  }
  11
  21
  31
<· undefined
>
```

Break, continue (without a label)



```
> for (var i = 0; true; i++) {
      if (i % 2 == 0) {
          continue;
      }

      console.log(i);

      if (i == 9) {
          break;
      }
  }

  console.log(i); // 9
  1
  3
  5
  7
  9
  9
< undefined
>
```

The break statement terminates the current loop, switch, or label statement and transfers program control to the statement following the terminated statement.

Usually, these are not needed in a production code (in loops)

However, if you do need to quit from a loop, you can do it wi loop, but you cannot quit from a .forEach().

Also, there is a version of break and continue with a label, but there is zero c that you can use labels in prod.

Controlling loops: break and conti

APPENDIX

Keywords and reserved words

Back to the topic

A keyword is a token that has a syntactic use. The keywords of ECMAScript include if, while, async, await, and many others.

A reserved word cannot be used as an identifier. Many keywords are reserved words, but some are not, and some are reserved only in certain contexts. if and while are reserved words. await is reserved only inside async functions and modules. async is not reserved; it can be used as a variable name or statement label without restriction.

reserved words

```
await break case catch class const continue debugger default delete
do else enum export extends false finally for function if import in
instanceof new null return super switch this throw true try typeof
var void while with yield
```

ECMA-262: Keywords and Reserved W

Motivation for block scoping

Back to the topic

The main motivation for block scoped declarations was to eliminate the “closure in loop” bug hazard that may JavaScript programmer have encountered when they set event handlers within a loop.

```
function f(x) {
    for (var p in x) {
        var v = doSomething(x, p);

        element.addEventListener(function(args) {
            handle(v, p, args); // Every callback gets the same values for v and p
        });
    }
}
```

Some ECMAScript Explanations and Stories for Dave (wirfs-b

defer/async

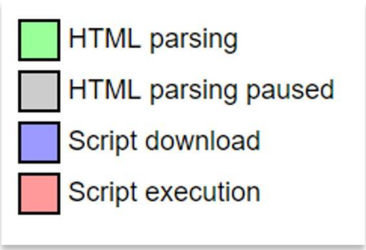
```html
<script src="1.js" async></script>
<script src="2.js" defer></script>
```

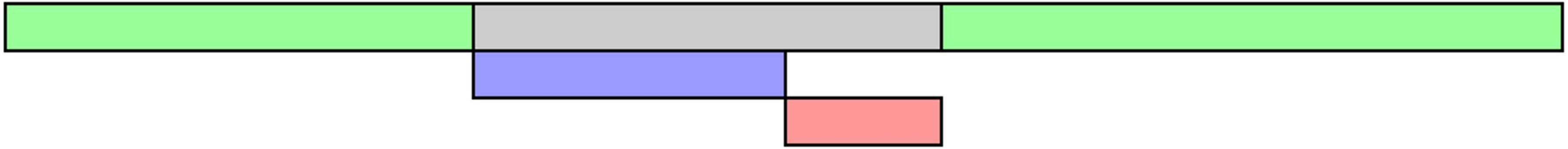
	Order	**DOMContentLoaded**
async	Load-first order. Their document order doesn’t matter – which loads first runs first	Irrelevant. May load and execute while the document has not yet been fully downloaded. That happens if scripts are small or cached, and the document is long enough.
defer	Document order (as they go in the document).	Execute after the document is loaded and parsed (th wait if needed), right before DOMContentLoaded.

<script>

<script> without any attributes

The HTML file will be parsed until the script file is hit, at that point parsing will stop and a request will be made to fetch the file (if it’s external). The script will then be executed before parsing is resumed.

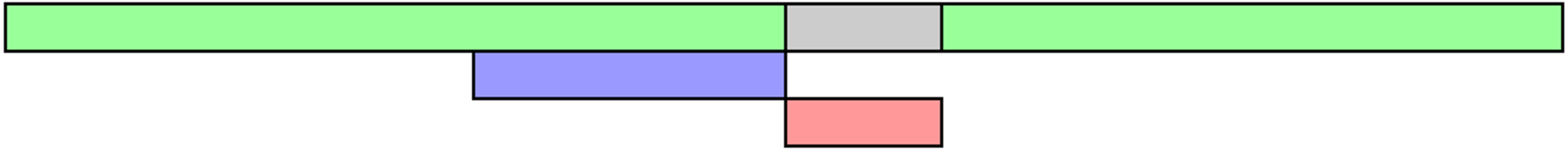




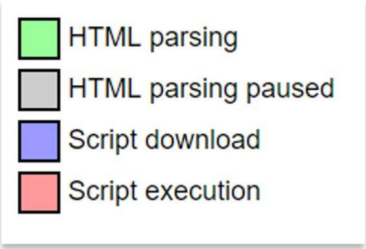
<script async>

- 🟩 HTML parsing
- ⬜ HTML parsing paused
- 🟦 Script download
- 🟥 Script execution

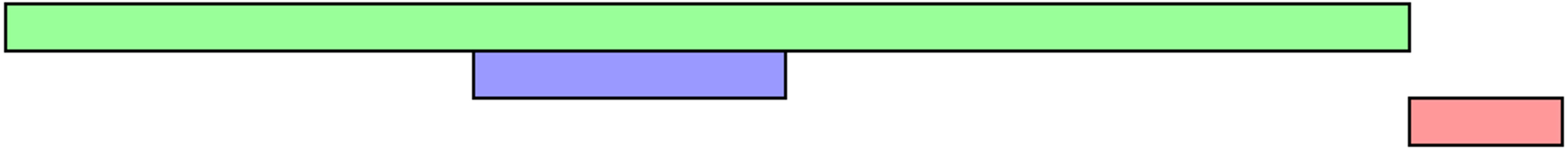
<script> with async attribute
Downloads the file during HTML parsing and will pause the HTML parser to execute it when it has finished downloading.



<script defer>

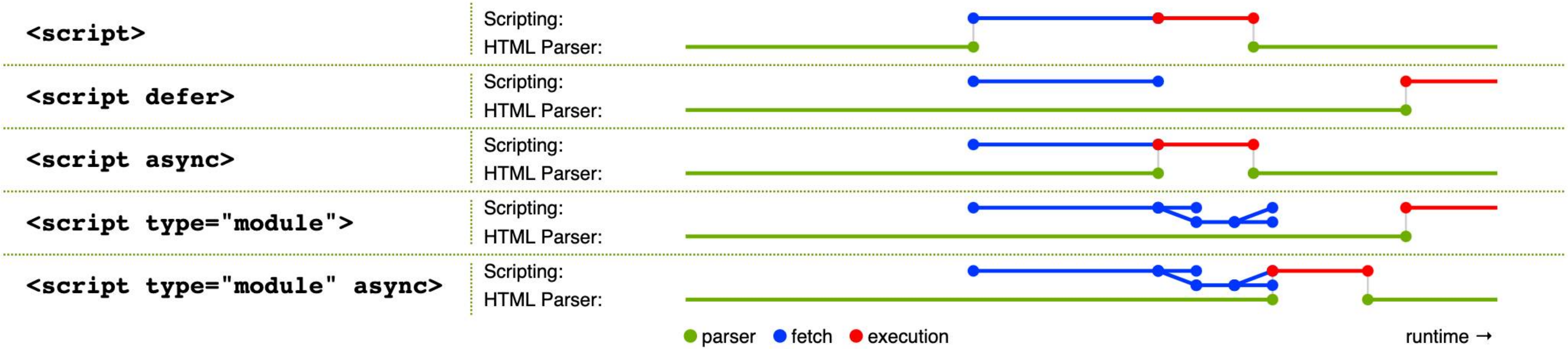


<script> with defer attribute
Downloads the file during HTML parsing and will only execute it after the parser has completed. Defer scripts are also guaranteed to execute in the order that they appear in the document.



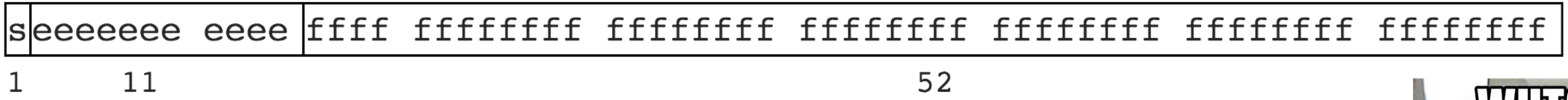
Parsing / fetching / execution

Back to the topic

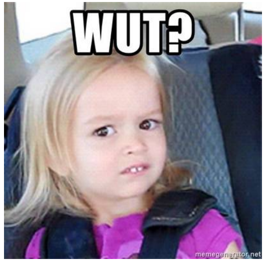


IEEE 754

Back to the topic



```
s eeeeeeee eeee ffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
1     11                                  52
```



```
0.1 + 0.2 === 0.3; // false
```

What can be represented finitely in base-10 is not necessarily can be in base-2.

Example: $0.1_{10} = 0.0011001100110011.._2$ which gives us mantissa $0.0001_2 = 1/16_{10}$ = 0.0625, a nice rounding error of 0.0375, assuming we have 5-point precision – fortunately, JavaScript uses a 52-bit mantissa, so rounding errors are small, but still there!

Result: floating points cause errors to build up over time, and even worse: arithmetic operations like *, /, +, -) are not guaranteed when dealing with floating points, even at high precision ones, e.g. ((x + y) + a + b) is not necessarily equal to ((x + y) + (a + b)).

This is not even unique to JavaScript!