Javascript (Part I)

Pau Fernández Rafa Genés Jose L. Muñoz

Universitat Politècnica de Catalunya (UPC)

Table of Contents

The Javascript Language

Javascript in the browser

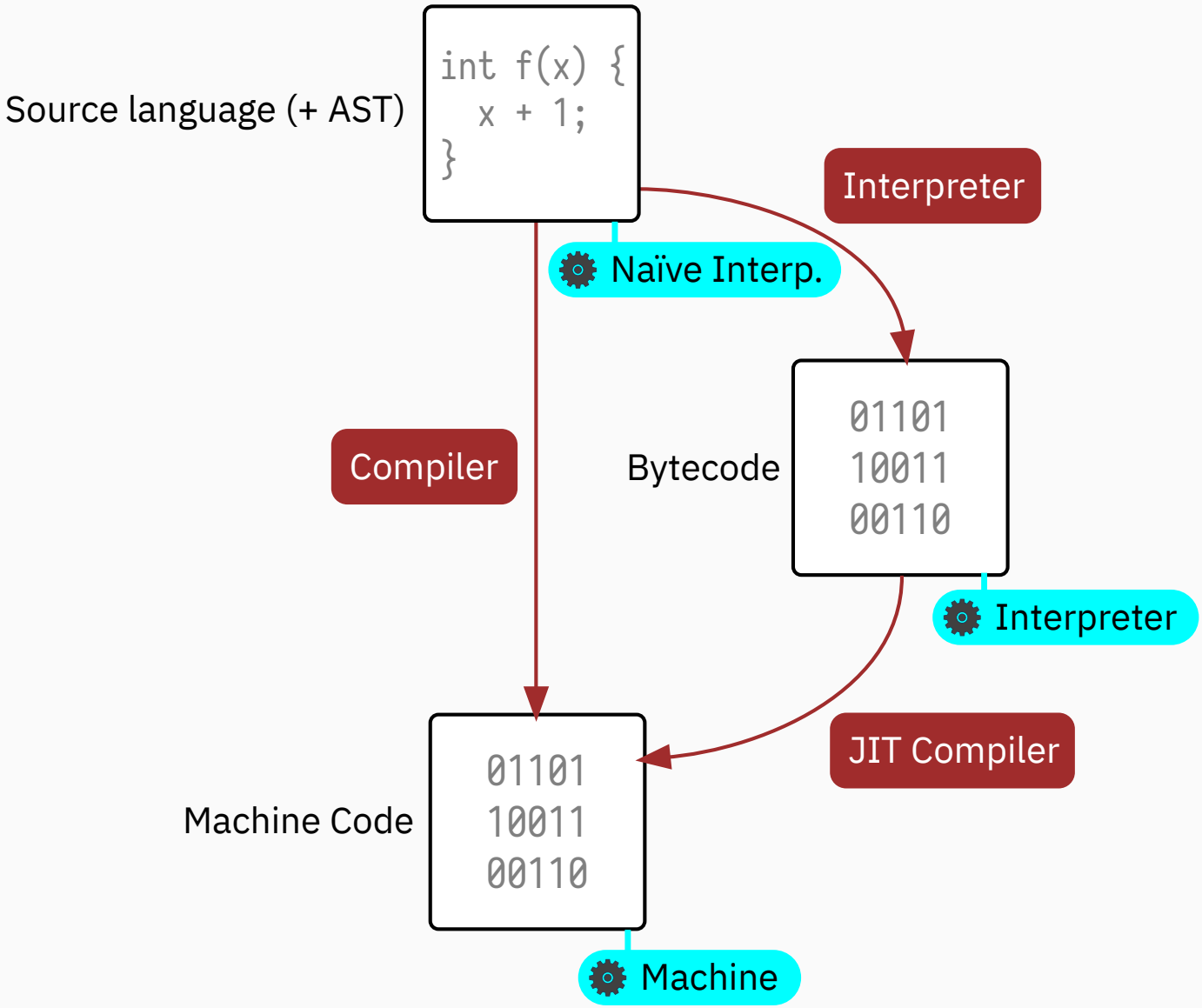
- It is one of the three pillars of the Web Platform (along with HTML and CSS).
- Has no relation with Java at all (it was going to be called LiveScript but a marketing decision changed it).
- Started as a browser scripting language (in 1995).
- Was ”unleashed” from the browser in 2009 by Ryan Dahl (NodeJS), who took the V8 engine and combined it with an event loop and a low-level IO API.
- Classification:
  - Imperative
  - Dynamic
  - Object-oriented (prototype-based)
  - Functional (first-class functions)
  - Interpreted (but with incredibly good JITs)
  - Garbage Collected
- Now managed by the Mozilla Foundation.
- Standardized by Ecma International in ECMA-262, and ISO/IEC 16262.

Compilation, Intrepretation and JITs

Languages can be implemented as:

- **Compilers**: they translate source code directly to the target machine language (the final program doesn't need the language implementation).
  Examples: C, C++, Rust, Go, D, ...
- **Interpreters**: they execute programs by interpreting the meaning of instructions at run time (the final program *needs* the language implementation).
  Examples: Python, Ruby, Perl, PHP, Lua, Haxe, ...
- **JIT compilers**: they interpret programs and compile parts of them on the fly (the final program *needs* the language implementation).
  Examples: Java, C#, Javascript, LuaJIT, ...

Compilation, Intrepretation and JITs (2)



Javascript Versions

Ver	Official Name	Changes
1	ECMAScript 1 (1997)	First Edition.
2	ECMAScript 2 (1998)	Editorial changes.
3	ECMAScript 3 (1999)	Regular Expressions, `try`/`catch`.
4	ECMAScript 4	(Never released.)
5	ECMAScript 5 (2009)	`"strict mode"`, JSON, `String.trim`, `Array.isArray`, iterators.
5.1	ECMAScript 5.1 (2011)	Editorial changes.
6	ECMAScript 2015	`let`, `const`, default parameters, classes, modules, destructuring, arrow functions, templates, Symbol type, generators, iterators, ...
7	ECMAScript 2016	Exponentiation (**), `Array.prototype.includes`.
8	ECMAScript 2017	String padding, new `Object` properties, `async` functions, shared memory.
9	ECMAScript 2018	Rest / spread properties, Asynchronous, iteration, `Promise.finally()`, additions to RegExp.

Can I Use?

https://caniuse.com

Babel

Babel is a **Javascript compiler**

https://babeljs.io/

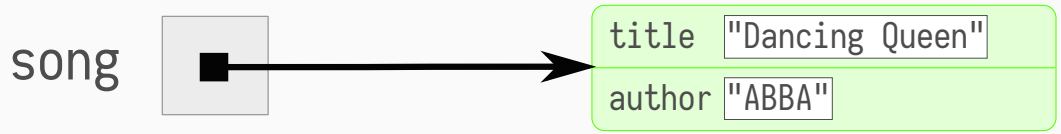
It can:

- Compile from one version of Javascript to another.
- Polyfill features that are missing in your implementation.
- Convert JSX syntax (for React).
- Allow type annotations (for Flow and TypeScript).
- Be configured and extended using plugins.

Garbage Collection: Example

First, we create a **song**:

```
let song = {
  title: "Dancing Queen",
  author: "ABBA",
}
```



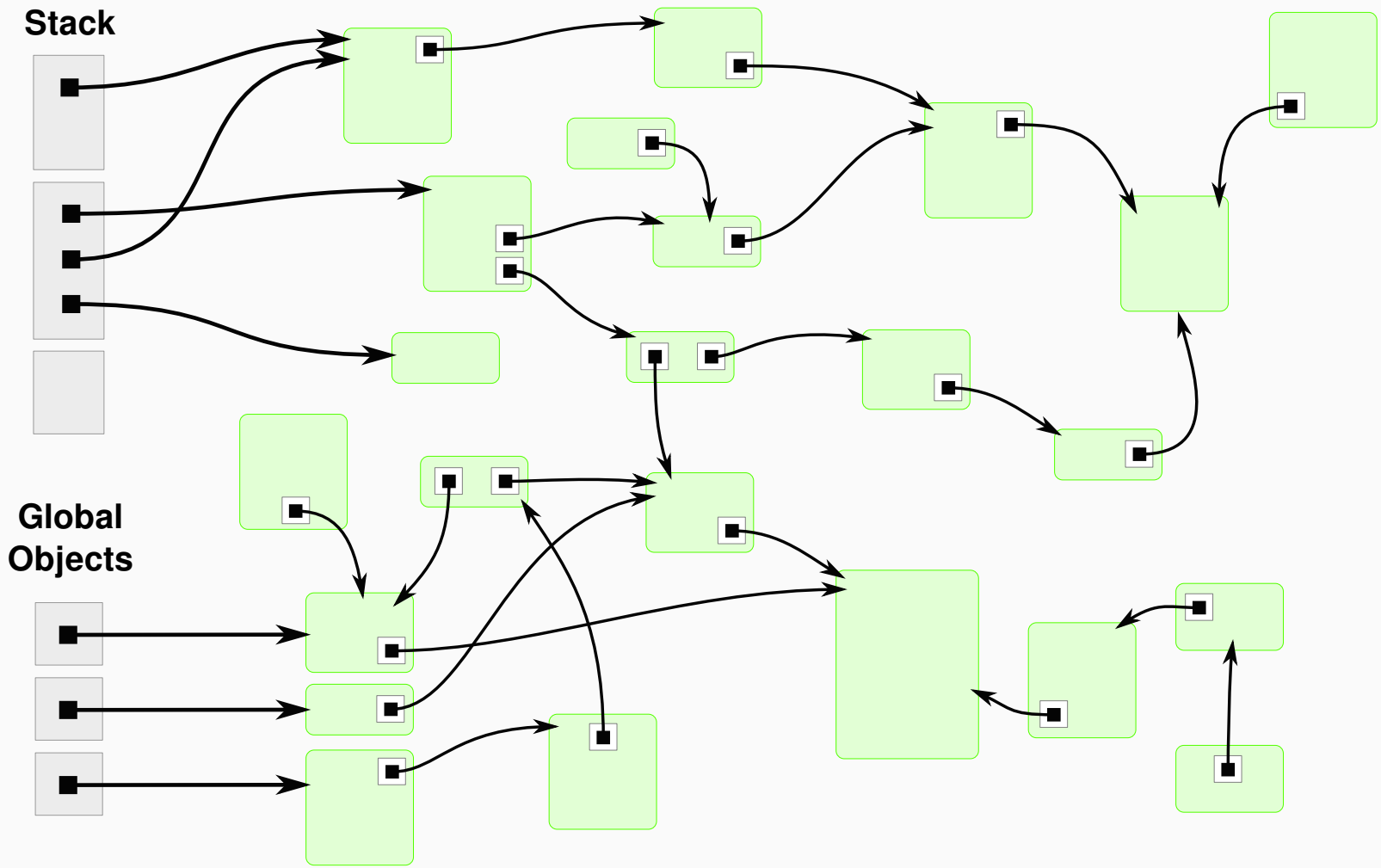
Now we set **song** to **null**:

```
song = null
```

The song is now garbage (and will be collected in the next cycle).

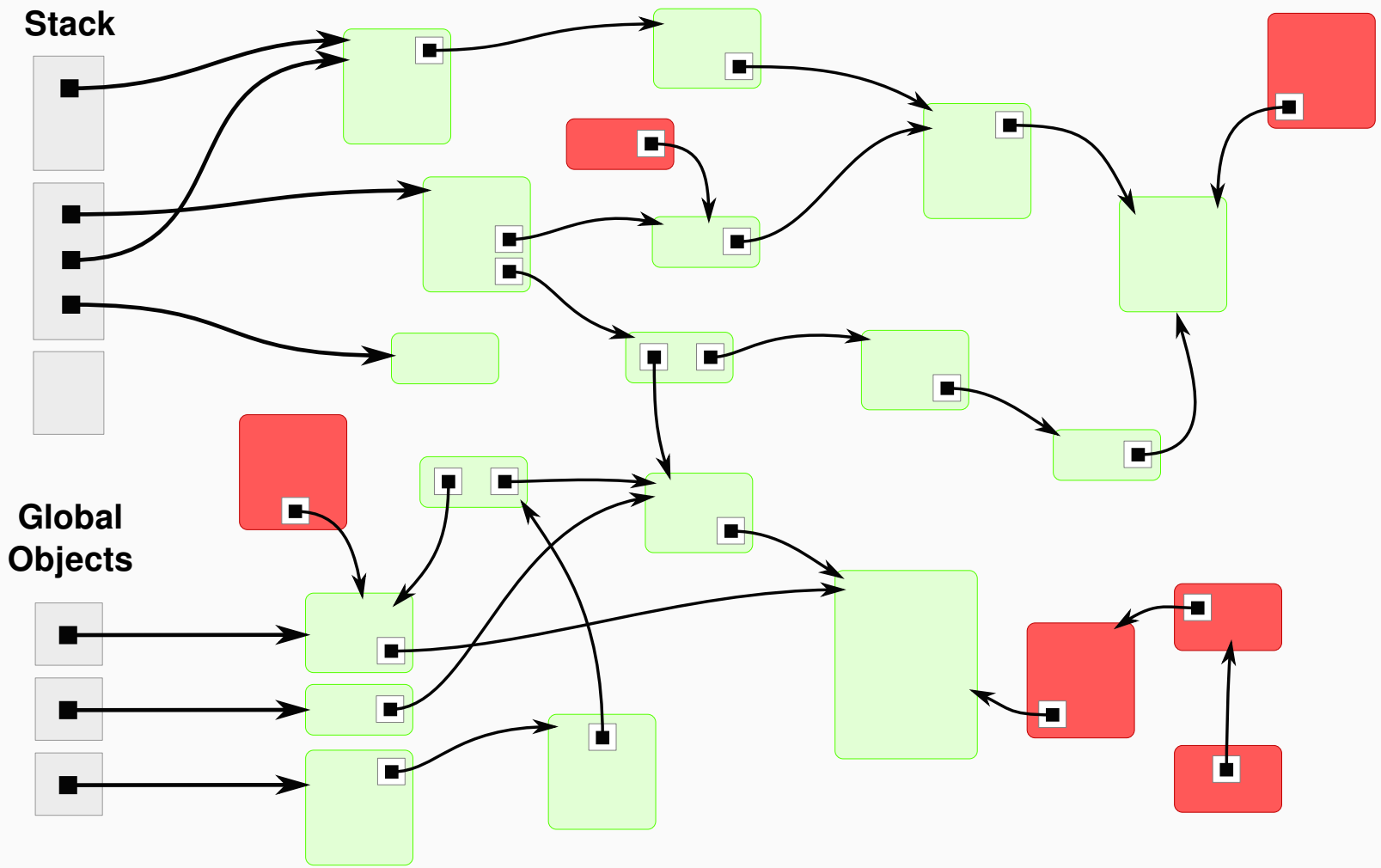
Garbage Collection: Reachability 1

Any object that doesn't have a pointer ultimately connecting it to the stack or any global variable is "garbage" (it is not being used).



Garbage Collection: Reachability 2

Any object that doesn't have a pointer ultimately connecting it to the stack or any global variable is "garbage" (it is not being used).



Hello, Browser

Put this into a file called **hello.html** and open it with a browser:

```
<!doctype html>
<html>
  <body>
    <p>This document will salute you!</p>
    <script>
      alert('Hello from Javascript!')
    </script>
  </body>
</html>
```

Table of Contents

NodeJS Installation

How to install NodeJS with "n"

- We will use the "n" package to install NodeJS.
  https://github.com/tj/n
- Local installation for our user (no root access required).
- To install "n":

```
$ curl -L https://git.io/n-install | bash
```

- Installs nodejs in $HOME/n/
- In the configuration file of our shell (e.g. .bashrc, .zshrc, etc) it adds the necessary paths:

```
export N_PREFIX="$HOME/n";
[[ :$PATH: == *":$N_PREFIX/bin:"* ]] || PATH+=":$N_PREFIX/bin"
```

- Also convenient to create a soft link for node called nodejs:

```
$ ln -s ~/n/bin/node ~/n/bin/nodejs
```

Install Global Packages (**nodejs** Apps)

- Installing **nodejs** standalone applications using **npm**:

```
$ npm install -g weather-cli
$ weather -c Barcelona -C spain
Barcelona, spain
Condition: Clouds
Temperature: 7.98°C
```

- List global packages installed:

```
$ npm list -g --depth=0
/home/jlmunoz/n/lib
  |---npm@6.5.0
  |---weather-cli@1.3.2
```

- If you want to know where global modules are installed:

```
$ npm root -g
/home/user/n/lib/node_modules
```

Installing Versions of **nodejs** with "**n**"

- Install the latest **nodejs** version:

```
$ n latest
```

- Install **nodejs** version 11.x.y:

```
$ n 11
```

- Install **nodejs** version 11.13.y:

```
$ n 11.13
```

- Versions are stored in **$HOME/n/n/versions/node/**
- We can list versions (available and installed):

```
$ n ls
```

Switching and Removing Versions of **nodejs** with "**n**"

When running multiple versions of **nodejs**, we can use the binaries stored by "**n**".

- To ask about the path of a **nodejs** binary:

```
$ n bin 11.9.0
/home/user/n/n/versions/node/11.9.0/bin/node
```

- To use a specific version:

```
$ n use 11.9.0 some.js
```

- To remove a specific **nodejs** version:

```
$ n rm 11.9.0
```

- To remove all versions except the current:

```
$ n prune
```

Hello, Node

Put this into a file called `hello.js`:

```
console.log("Hello from Javascript!")
```

and execute it from the command line:

```
$ node hello.js
```

Table of Contents

Javascript Fundamentals

Statements

Javascript is an *imperative language*, where you issue statements one after another:

```
console.log('Hi, there...')
console.log('...Javascript!')
```

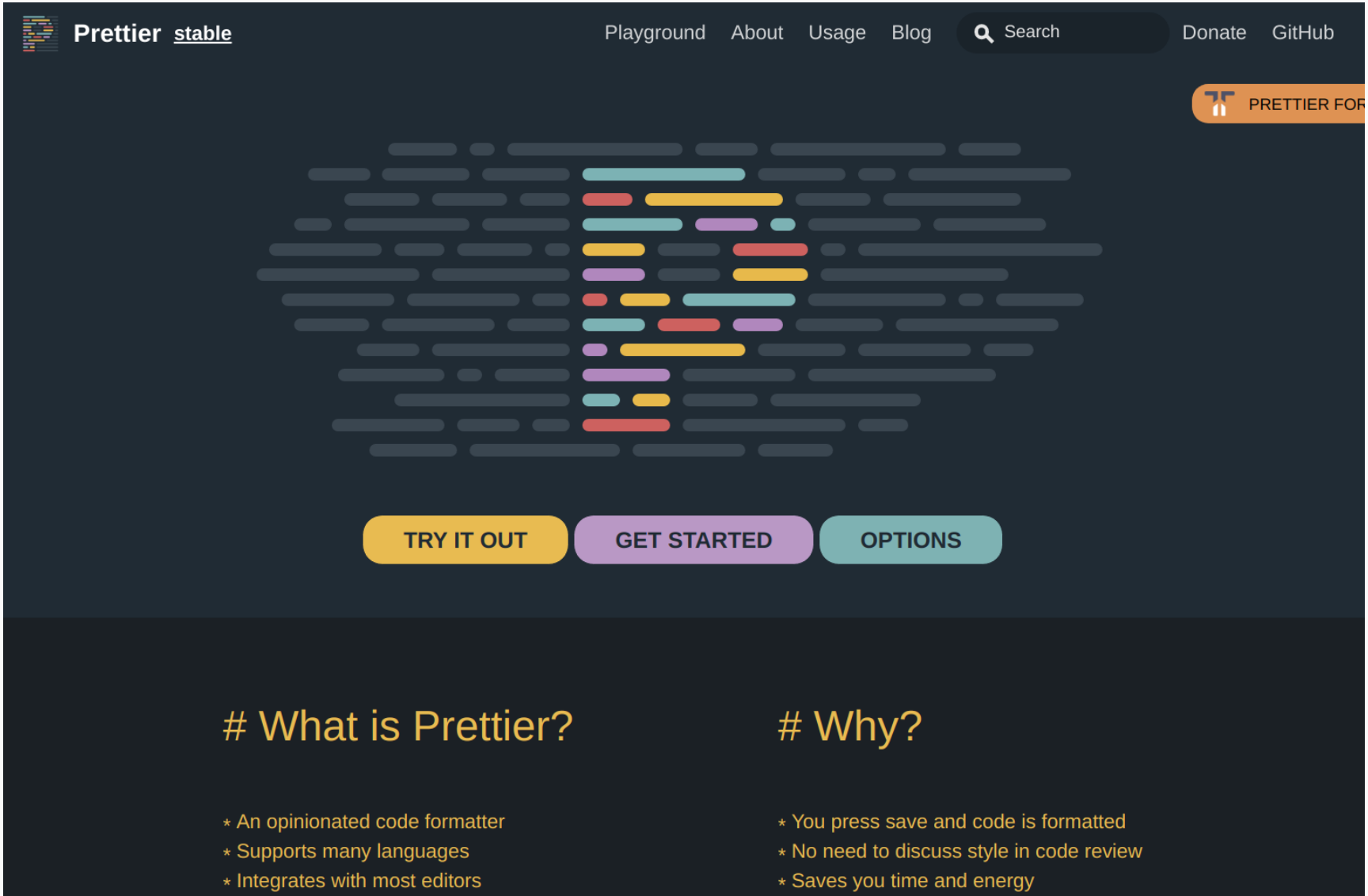
A statement is an instruction to perform a specific action.

A program is just a list of statements.

Semicolons

- In Javascript, a semicolon can be omitted when a linebreak exists (unlike in C or Java)
- But this could give rise to difficult to diagnose errors:
  - https://javascript.info/structure#semicolon
  - https://standardjs.com/rules.html#semicolons
- Don't use them if your are comfortable with the exceptions.
- Use them if you need an easy to remember rule that doesn't give surprises.

https://prettier.io/



Expressions

An expression is a piece of code that has a value (and type):

- The value you assign to a variable.
- The condition in an `if` statement.
- The parameters in a function.
- ...

For functions, expression or statement depends on the context:

```
const f1 = function () { }     // f1 assigned to a function expression
const f2 = function foo() { }  // f2 assigned to a named function expression
function foo() { }             // function declaration (this is a statement and it is hoisted)
```

The compiler determines if the code is an expression or a statement.

Comments

- Single-line comments begin with //
- Multi-line comments begin with /* and end with */.
- You cannot **nest** multi-line comments (just like in C).

```
console.log('hi') // This function call prints 'hi'

/* This other function call
   prints 'ho' */
console.log('ho')

/* Nesting /* comments */ is == "not a good idea" */
```

Strict Mode

```
"use strict" // totally vacuous expression with the side-effect
             // of enabling strict mode in the language
             // -> from this point on <-
```

This expression enables ”Strict Mode”.

”Strict Mode” only allows standardized Javascript (ECMAScript 5 and later) and ”breaks” (very) old code.

Babel and other compilers always output `"use strict"`.

Since ECMAScript 5, modules enter ”Strict Mode” by default.

Table of Contents

Variables

Variables

A variable is a **named storage** for an object.

You create a variable with `let`:

```
let a
```

And change its value freely:

```
a = 'the quick brown fox'
a = 1
a = true
```

Variables don't have types, **values have types**. This is in contrast to languages like C or Java.

You can declare many variables and assign values at once:

```
let user = 'pauek', password = 'fr34ky', now = '2018-10-07 13:12'
```

Old Javascript used to have `var` instead of `let`, which subtly changes the behavior of the declaration.

const

`const` declares variables that can't be changed:

```
const pizza = true
pizza = false        // TypeError: Assignment to a constant variable
```

The object itself can be changed, though:

```
const list = [5, 4, 3, 2, 1]
list.sort()
console.log(list) // -> [1, 2, 3, 4, 5]
```

Lexical scoping

`let` introduces *lexical scoping*: a variable lives within the block that contains it (also overriding any variable with the same name in an outer scope)

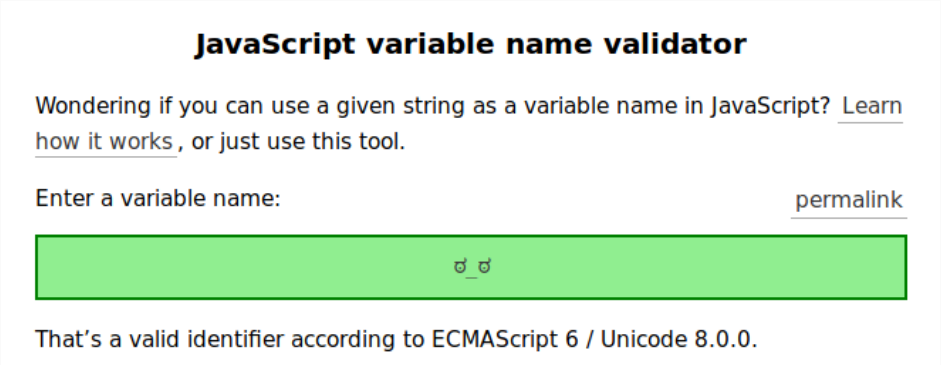
```
let lexical = true
let js = "ES6"

if (lexical) {
  let js = "Javascript"
  console.log(js) // -> Javascript
}

console.log(js)   // -> ES6
```

Variable names

Variable names are quite different from other languages. The rules are (https://mathiasbynens.be/notes/javascript-identifiers):

- They cannot be reserved words: **break**, **case**, **catch**, **class**, **continue**, **const**, **debugger**, **default**, ...
- Or literals like **true**, **false**, and **null**.
- Or things that act like reserved words: **NaN**, **Infinity**, **undefined**.
- They can start with **$**, **_**, or **Unicode Letters**.
- The rest of the name can use Unicode characters that are non-spaces.

Ok, that's too difficult, just use:



https://mothereff.in/js-variables

Table of Contents

Types

Basic types

Number

- For both integers and real numbers.
- There are three special values: `Infinity`, `-Infinity` and `NaN`.
- Doing math with numbers is safe (it generates no exceptions or errors). At most you get values like `NaN` or `Infinity`.

Boolean

- For truth values: `true` and `false`.
- The result of expressions like:

```
age > 18
```

String

- Sequences of characters

Quotes

Three types of quotes:

```
let q1 = 'Single quotes'
let q2 = "Double quotes"
let qb = `Back quotes`
```

Double and single quotes are **equivalent**. Single quotes seem to be preferred by the community.

Back quotes have two important properties:

- They implement interpolation (expression embedding with ${}).
- They represent *raw strings* that can span multiple lines.

Backquotes

Interpolation (expression embedding) allows us to produce a string which includes computed values:

```
let home = "cruel world";
console.log(`Bye, ${home}!`);
console.log(`The result is ${25 * 9 + 3 / 2}`);
```

Strings spanning multiple lines are also useful in many situations:

```
let usage = `usage: cat [options] args...
Meaw on the screen.

options:
  -1   One time.
  -n   N times.
`
console.log(usage)
```

Special values: **undefined** and **null**

Special values

- **undefined**
  Variables have this value if *uninitialized*. Means something like ”variable not assigned”.

```
let a
console.log(a) // undefined
```

- **null**
  A value indicating that a variable doesn’t have any value. This value is similar to Java’s `null`, it means that the variable doesn’t point to any object, it is *empty*.

```
let b = null
console.log(a)
```

Type Conversions

to String

```
String(true)          // -> 'true'
String(1)             // -> '1'
```

to Number

```
Number('123')         // -> 123
Number(true)          // -> 1
Number(false)         // -> 0
Number('wtf')         // -> NaN
Number(null)          // -> 0
Number(undefined)     // -> NaN
```

to Boolean

```
Boolean(1)            // -> true
Boolean("something") // -> true
Boolean("")           // -> false
```

typeof

It returns the type of an expression as a string:

```
typeof 13           // -> 'number'
typeof 3.141592     // -> 'number'
typeof 'asdf'       // -> 'string'

typeof true         // -> 'boolean'
typeof false        // -> 'boolean'
typeof undefined    // -> 'undefined'
typeof null         // -> 'object'
typeof {}           // -> 'object'
typeof Symbol()     // -> 'symbol'
typeof ((x) => x+1) // -> 'function'
```

Table of Contents

Operators

Operators: String Concatenation

Strings can be joined with the `+` operator:

```
let s = "Conde" + "mor"
console.log(s) // -> Condemor
```

One very important thing to remember is that if any of the operands is a string, the other operand is converted to a string too:

```
let twelve = '1' + 2
let not_ten = 5 + '5'
console.log(twelve)    // -> 12
console.log(not_ten)   // -> 55
```

What is the output of this?

```
let a = 2 + 3 + "4"
console.log(a)         // -> ???
```

Operators: Arithmetic and Logical

Arithmetic operators (+, -, *, / and %)

logical operators (and &&, or ||, not !) and

relational operators (>, <, >=, <=, ==, !=)

have the same behavior as in C or Java:

```
let x = 0, y = 5
let inside  = (x > -2 && x < 2)
let outside = (y < 0 || y > 10)
let c = "i"
let vowel = (c == "a" || c == "e" || c == "i" || c == "o" || c == "u")
let consonant = !vowel
let year = 2014
let leap_year = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

What distinguishes Javascript from the previous (for better or worse) is the "strict equality" operator ===.

Operators: Numeric Conversion

The unary + operator has the effect of calling the `Number` conversion:

```
let x = 1, y = -5.5
console.log(+x)      // -> 1    (no effect on numbers)
console.log(+y)      // -> -5.5 (no effect on numbers)
console.log(+true)   // -> 1
console.log(+"")     // -> 0     (who knew...)
let a = "2", b = "3"
console.log(+a + +b) // -> 5
```

Operators: Assignment

Assignment can also be seen as an operator because assignments return the value of the assigned variable, therefore they can be chained:

```
let a, b, c
a = b = c = 2 * 2
console.log(a)     // -> 4
console.log(b)     // -> 4
console.log(c)     // -> 4
```

You should not write code like this, but...

```
let a = 1
let b = 2
let c = 3 - (a = b + 1)
console.log(a)           // -> 3
console.log(c)           // -> 0
```

Operators: Remainder

This operator does not have anything to do with percents, it is the remainder of the integer division (as in C, C++, Java, and many other languages):

```
let a = 5, b = 2, c = 7, d = 13
console.log(a % b)   // -> 1
console.log(b % c)   // -> 0
console.log(c % a)   // -> 2
console.log(d % c)   // -> ???
console.log(d % a)   // -> ???
```

Operators: Exponentiation

Exponentiation is a recent addition to the language (ES6).

```
console.log(2 ** 3)       // -> 8
console.log(5 ** 4)       // -> 625
console.log(2 ** 8)       // -> 256
console.log(3 ** 5)       // -> 243
console.log(2 ** 1/2)     // -> 1
console.log(2 ** (1/2))   // -> 1.4142135623730951
console.log(8 ** (1/3))   // -> 2
```

Operators: Bitwise Operations

Javascript inherits bitwise operators from C:

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`!`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO-FILL RIGHT SHIFT (`>>>`)

These operators are used quite rarely, but they can be useful.

Operators: Increment and Decrement

Increment works exactly like in C:

- Increments and decrements in one unit:

```
let i = 0, j = 10
i++     // -> increment by 1
j--     // -> decrement by 1
```

- Pre-increment acts **before** evaluation (same with decrement):

```
let x = 5
if (++x > 5) {
  console.log("Boom!")
}
```

- Post-increment acts **after** evaluation (same with decrement):

```
let a = 0, b = 3
a = b++
console.log(a)   // -> 3
```

Operators: Modify in Place

Lets suppose you have a variable with a very long name:

```
let variableWithAVeryLongName = 5
```

and you want to multiply it by 2:

```
variableWithAVeryLongName = variableWithAVeryLongName * 2
```

For the purpose of abbreviating these instructions, there are special operators that let you apply both the operation and the assignment at once:

```
variableWithAVeryLongName += 1
variableWithAVeryLongName *= 3
variableWithAVeryLongName /= 2
variableWithAVeryLongName -= 4
// Also: %=, **=, <<=, >>=, >>>=, &=, |=, ^=
```

Implicit Conversions

It is very important to have in mind these implicit conversions:

- Addition with strings implicitly converts to a string:

```
1 + '2'             // -> '12'
'1' + 2             // -> '12'
'1' + null          // -> '1null'
'2' + undefined     // -> '2undefined'
null + 'able'       // -> 'nullable'
undefined + 'ness' // -> 'undefinedness'
```

- Mathematical expressions involving things which are not numbers auto-convert to numbers:

```
true + false        // -> 1
'57' / '8'          // -> 7
1111 / 'x'          // -> NaN
null + 1            // -> 1
undefined + 1       // -> NaN
```

Comparisons: Loose Equality

The == operator (two equals) is ”loose” in Javascript because prior to the comparison, **it auto-converts values to numbers**:

```
let num = 0
let obj = new String('0')
let str = '0'

console.log(num == num)        // -> true
console.log(obj == obj)        // -> true
console.log(str == str)        // -> true

console.log(num == obj)        // -> true
console.log(num == str)        // -> true
console.log(obj == str)        // -> true
console.log(null == undefined) // -> true

// both false, except in rare cases
console.log(obj == null)       // -> false
console.log(obj == undefined)  // -> false
```

Comparisons: Strict Equality

"===" is the strict version of "==". (It does not auto-convert.)

Always returns false with operands of different types.

```
var num = 0
var obj = new String('0')
var str = '0'

console.log(num === num)        // true
console.log(obj === obj)        // true
console.log(str === str)        // true

console.log(num === obj)        // false
console.log(num === str)        // false
console.log(obj === str)        // false
console.log(null === undefined) // false

console.log(obj === null)       // false
console.log(obj === undefined)  // false
```

Checking if a value is **NaN**

In Javascript something curious happens

```
NaN == NaN   // -> false (?)
NaN === NaN  // -> false (????)
```

To check if a certain value is NaN

```
Number.isNaN(NaN)           // -> true
Number.isNaN(1)             // -> false
Number.isNaN([])            // -> false
Number.isNaN('NaN')         // -> false
Number.isNaN({ NaN: true }) // -> false
```

`Number.isNaN` can be called with any object and returns the expected result. (Do not confuse with `isNaN` which is a standalone function.)

Table of Contents

Control: Alternatives

The `if` statement

The `if` statement evaluates the condition between parentheses, and if the resulting value is `true`, it executes the code inside de braces:

```
let password = prompt("Enter your password:");
if (password == "z0rk") {
  alert("You are now logged in");
}
```

With an `else` clause, when the condition evaluates to `false`, the code in the `else` branch is executed:

```
let password = prompt("Enter your password:")
if (password == "z0rk") {
  alert("You are now logged in")
} else {
  alert("Wrong password. This computer will now self-destruct.")
}
```

Boolean Conversion

The condition inside the `if`'s parentheses gets converted to a boolean if it is of another type:

This values are converted to `false`:

- the number `0`,
- an empty string `""`,
- `null`,
- `undefined`, and
- `NaN`.

The rest of values become `true`.

More than One Condition: `else if`

Whenever we have another `if` statement inside an `else` branch:

```
if (weight < 3) {
  word = "small"
} else {
  if (weight < 5) {
    word = "medium"
  } else {
    word = "big"
  }
}
```

It is usually written in a different form to better show the different cases:

```
if (weight < 3) {
  word = "small"
} else if (weight < 5) {
  word = "medium"
} else {
  word = "big"
}
```

The Ternary Operator

Whenever we have to produce a value depending on a condition:

```
let veredict
if (mark < 5) {
  veredict = "disapproved"
} else {
  veredict = "approved"
}
```

we can use a special syntax "`<cond> ? <if-true> : <if-false>`" but only if the two alternatives are of the same type:

```
let veredict = (mark < 5 ? "disapproved" : "approved")
```

Other examples:

```
let x = 33, y = "33"
let light = (color > 128 ? "light" : "dark")
let result = (x === y ? "strictly equal" : "different")
```

The `switch` Statement

A switch statement can replace multiple `if` conditions when they check for equality with different values:

```
switch (<expression>) {
case <value1>:
   // ...
   break;
case <value2>:
   // ...
   break;
case <value3>:
   // ...
   break;
default:
   // ...
}
```

`switch` Evaluation

1. The `<expression>` is evaluated.
2. The resulting value is compared **using strict equality** with the `<value>` in each `case`.
3. If a match is found, the code starting at that `case` is executed, until the `break` instruction.
4. If a case doesn't end with a `break` instruction, we will enter the next `case` and keep executing.
5. If there is a `default` clause and no other `case` has matched, the `default` clause is executed.

Table of Contents

Control: Loops

The `while` Statement

The `while` statement wraps in braces a piece of code (the `<body>`) that has to be executed many times, ”while” a condition holds:

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

The statement starts by evaluating the condition, and if it is `true`, it enters the body of the loop and executes it. After that, it checks the condition again, and keeps executing until the condition turns false, in which case it continues with the code below.

```
let i = 0
while (i < 5) {
  console.log(i)
  i++
}
```

The `for` statement

The `for` statement groups parts of the control of a loop that are dispersed in the `while` statement, such as the initialization, the condition, and the increment:

```
for (<init>; <cond>; <incr>) {
  <body>
}
```

The initialization (`<init>`) usually marks the starting point of the loop (usually controlled by an iterator). The condition (`<cond>`) determines the ending point, and the increment (`<incr>`) how the loop progresses.

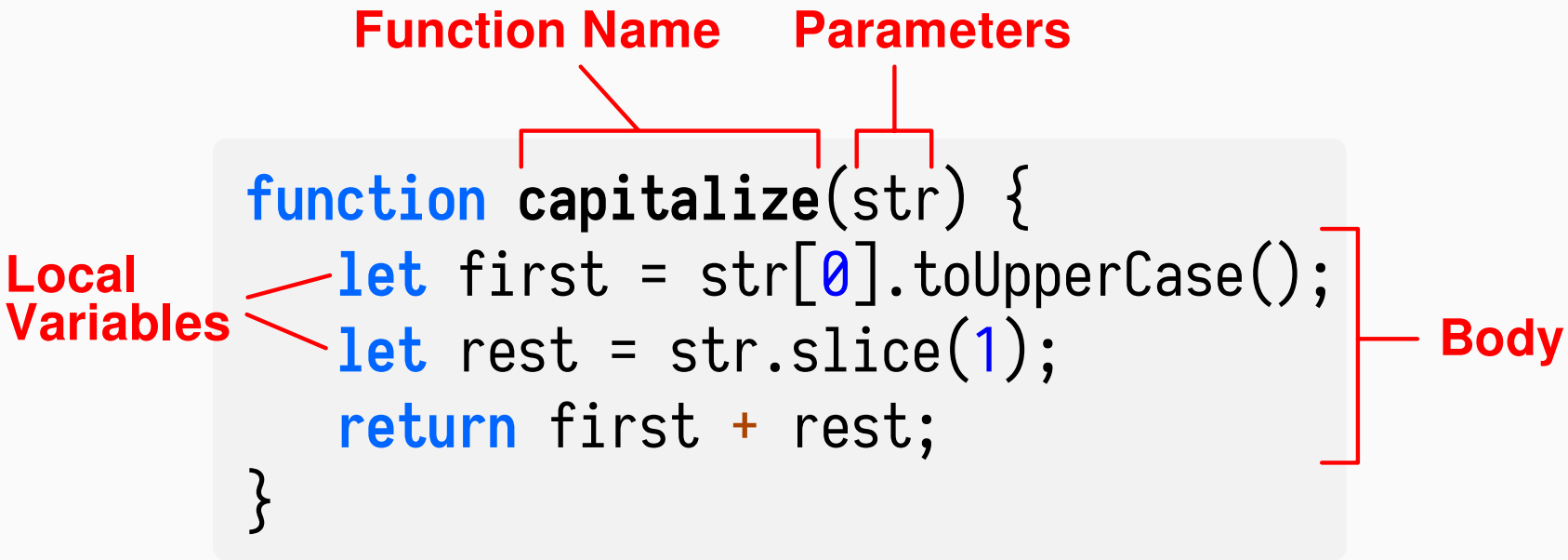
```
for (let i = 0; i < 5; i++) {
  console.log(i)
}
```

Table of Contents

Functions

Declaring functions

To create a function we can use a *function declaration*



A *local variable* is a variable declared inside a function and is only visible inside that function.

Returning Results

A function uses an explicit **return** statement to return results:

```
function sum(a, b) {
   return a + b
}
```

A function can also **return** without an expression:

```
function maybeSayHi(person) {
   if (person === "Adolf Hitler") {
      return
   }
   console.log("Hi, " + person + "!")
}
```

If you get the value returned in this case, it is **undefined**.

Function Arguments i

We can pass arbitrary data to functions using arguments:

```
function formatDate(day, month, year) {
   return day + "/" + month + "/" + year...
}
```

```
function veclen(x, y) {
   return Math.sqrt(x*x + y*y)
}
```

```
function repeat(str, n) {
   let result = ''
   for (let i = 0; i < n; i++) {
      result += str
   }
   return result
}
```

But how arguments are exactly assigned?

Function Arguments ii

When we call a function, arguments are copied into function parameters:

```
function f(num, obj, array) {
  // Do the stuff
}
f(2, {name: "joe", age: 29}, [1, 2, 3])
```

Important Notes

- Argument copy works **exactly like assignment**.
- For objects, the copied argument will be a new reference to the object, **not a deep copy**.
- Functions and arrays are objects too.
- Arguments can be used as local variables inside the function.
- Local variables disappear when the function terminates.

Function Arguments iii

Examples:

```
function addOne(x) { x++ }
function incAge(x) { x.age++ }
function pushOne(x) { x.push(1) }

let a = 5
let b = { name: "joe", age: 29 }
let c = []

addOne(a)   // Does not change "a" (useless function)
incAge(b)   // Increments the age property of "b"
pushOne(c)  // Pushes a 1 to "c"

console.log(a, b, c)   // 5   { name: 'joe', age: 30 }   [ 1 ]
```

Default values for Arguments

When parameters are missing, they take the value **undefined**.

```
function range(from, to, step) {
    let A = []
    for (let i = from; i < to; i += step) { A.push(i) }
    return A
}
console.log(range(1, 10))  // -> [1] ?????
```

To indicate the default value for parameters we can assign to a parameter:

```
function range(from, to, step = 1) {
    let A = []
    for (let i = from; i < to; i += step) { A.push(i) }
    return A
}
console.log(range(1, 10))     // -> [1, 2, 3, 4, 5, 6, 7, 8, 9]
console.log(range(1, 10, 2))  // -> [1, 3, 5, 7, 9]
```

First class functions i

Functions are ”first class” in Javascript, they are **values**.

Declaring a function like

```
function bomb() {
   console.log("Kaboom!")
}
```

is equivalent to doing it like

```
let bomb = function() {
   console.log("Kaboom!")
}
```

The meaning is the same: assign to **boom** the value of a function printing ”Kaboom!”.

Note: click here to read about the differences between them.

First class functions ii

Things you can do with functions:

- Assign them to variables.

```
let add1 = function(x) { return x + 1 }
```

- Copy them.

```
let addone = add1
```

- Pass them as parameters.

```
console.log(function() { console.log("what?") })
```

- Put them as elements of arrays.

```
let F = [function(x) { return x }, function(x) { return x + 1 }]
```

- Put them as fields of objects.

```
let obj = { a: 1, b: function() { console.log("Hi!") } }
```

Arrow functions

A new syntax lets us specify function literals more concisely:

```
function () { console.log("blah"); }   ⟶   () => console.log("blah")
```

```
function (x) { return x + 1; }   ⟶   x => x + 1
```

```
function (x, y) { return x + y; }   ⟶   (x, y) => x + y
```

In multiline functions, use braces and `return`:

```
function (a, b) {
   let dx = a.x - b.x;
   let dy = a.y - b.y;
   return dx*dx + dy*dy;
}
```

⟶

```
(a, b) => {
   let dx = a.x - b.x;
   let dy = a.y - b.y;
   return dx*dx + dy*dy;
}
```

Arrow functions: returning objects

Returning an object from an arrow function could be confused with a multiline function.

```
(a, b, c) => {
   name: a,        // is this an instruction?
   lastname: b,    // is this an instruction?
   age: c,         // is this an instruction?
}
```

Use parentheses in that case:

```
(a, b, c) => ({
   name: a,
   lastname: b,
   age: c,
})
```