INTRODUCTION TO JS

ABOUT US

Enrique Álvarez Mesa
alu0101142104

Sergio de la Barrera García
alu0100953275

TABLE OF CONTENTS

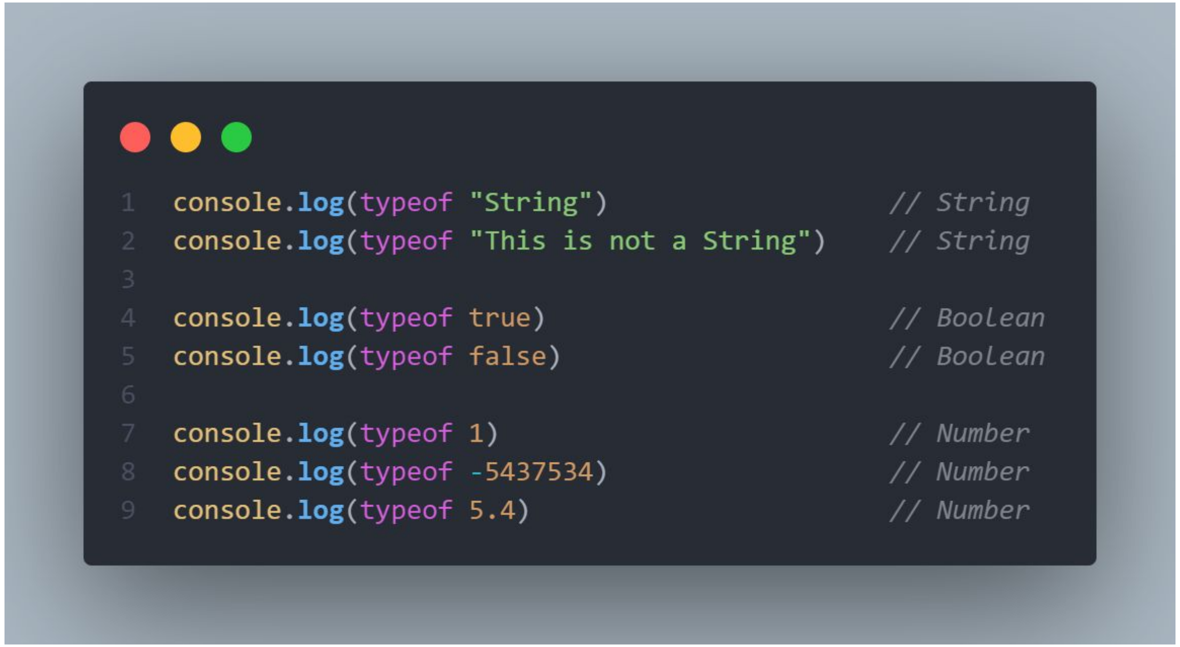
01

TYPES

Primitives, objects, curiosities, …

PRIMITIVES 1

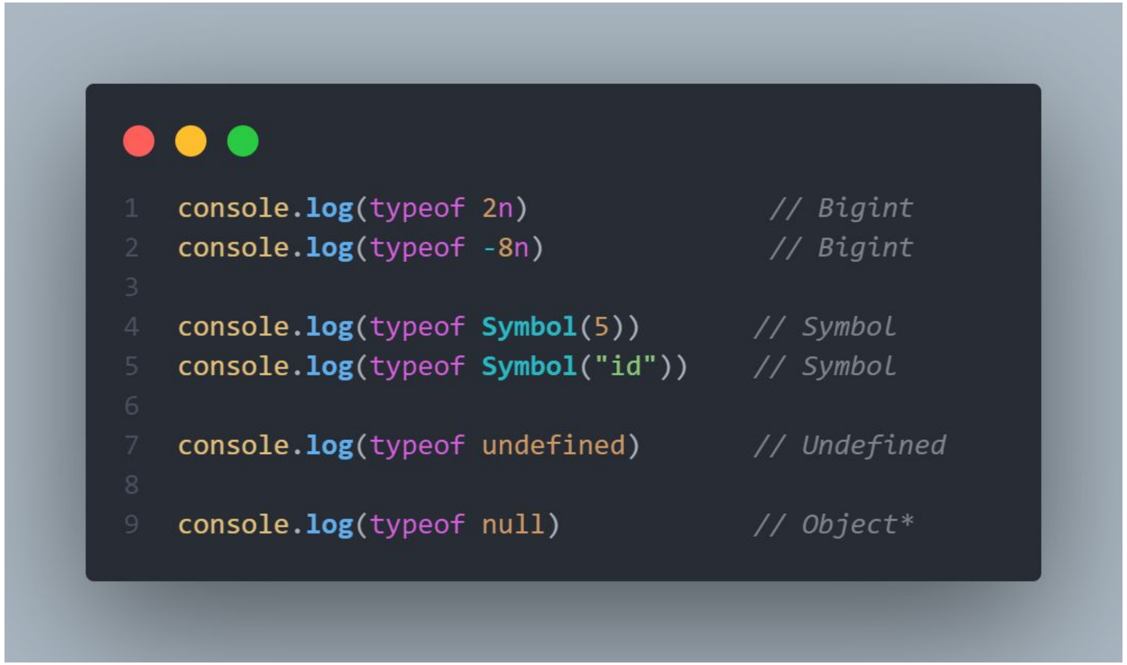
- String
- Boolean
- Number



```
console.log(typeof "String")                     // String
console.log(typeof "This is not a String")       // String

console.log(typeof true)                         // Boolean
console.log(typeof false)                        // Boolean

console.log(typeof 1)                            // Number
console.log(typeof -5437534)                     // Number
console.log(typeof 5.4)                          // Number
```

PRIMITIVES 2

- Bigint
- Symbol
- Undefined
- Null*

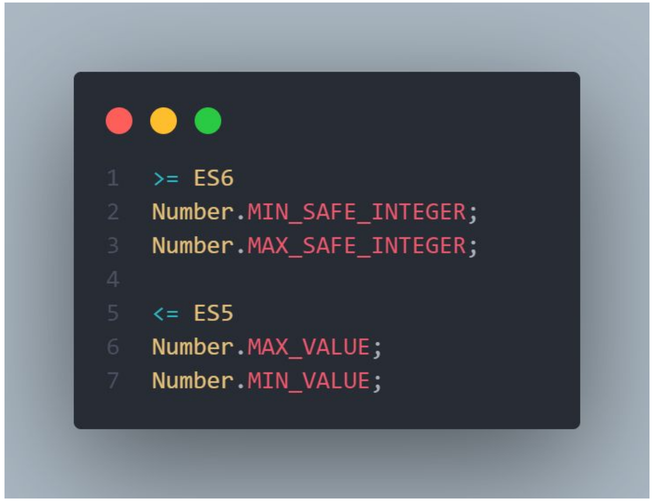


```
console.log(typeof 2n)              // Bigint
console.log(typeof -8n)             // Bigint

console.log(typeof Symbol(5))      // Symbol
console.log(typeof Symbol("id"))   // Symbol

console.log(typeof undefined)      // Undefined

console.log(typeof null)           // Object*
```

CURIOSITIES (NUMBER)

The maximum storable safe default value in Javascript is 9007199254740991.

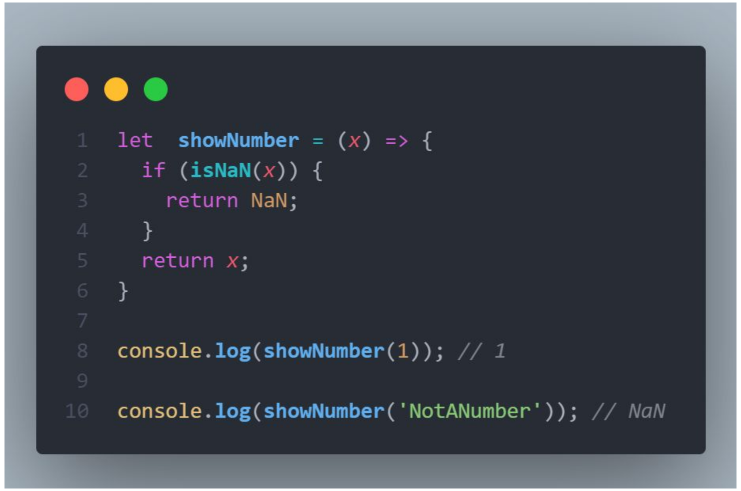
This can be modify carefully by doing the following:



```
>= ES6
Number.MIN_SAFE_INTEGER;
Number.MAX_SAFE_INTEGER;

<= ES5
Number.MAX_VALUE;
Number.MIN_VALUE;
```

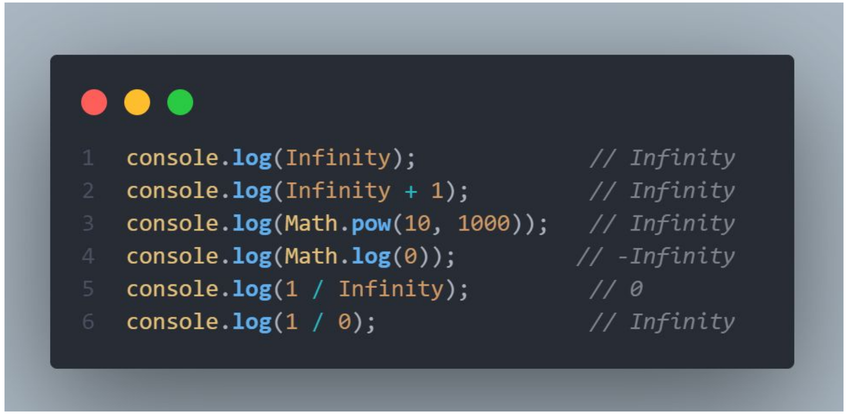
CURIOSITIES (NAN-INFINITY)

- NaN is a value representing Not-A-Number



```javascript
let  showNumber = (x) => {
  if (isNaN(x)) {
    return NaN;
  }
  return x;
}

console.log(showNumber(1)); // 1

console.log(showNumber('NotANumber')); // NaN
```

- Infinity is a numeric value representing infinity

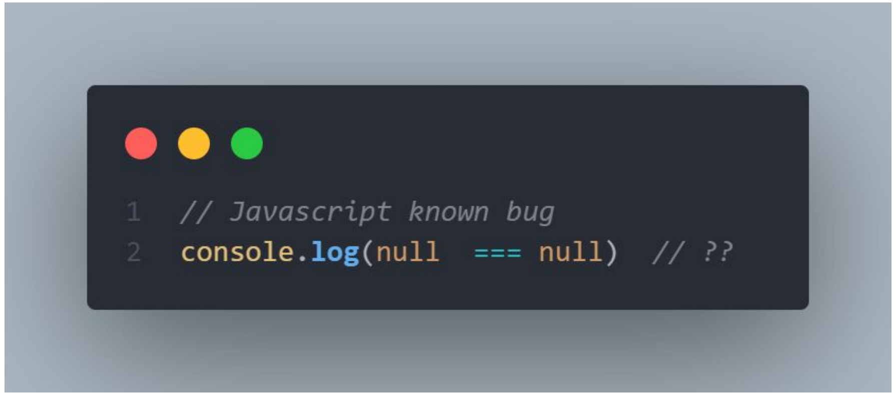


```javascript
console.log(Infinity);                  // Infinity
console.log(Infinity + 1);              // Infinity
console.log(Math.pow(10, 1000));        // Infinity
console.log(Math.log(0));              // -Infinity
console.log(1 / Infinity);              // 0
console.log(1 / 0);                     // Infinity
```

MAX_VALUE: 1.7976931348623157e+308

CURIOSITIES (NULL)

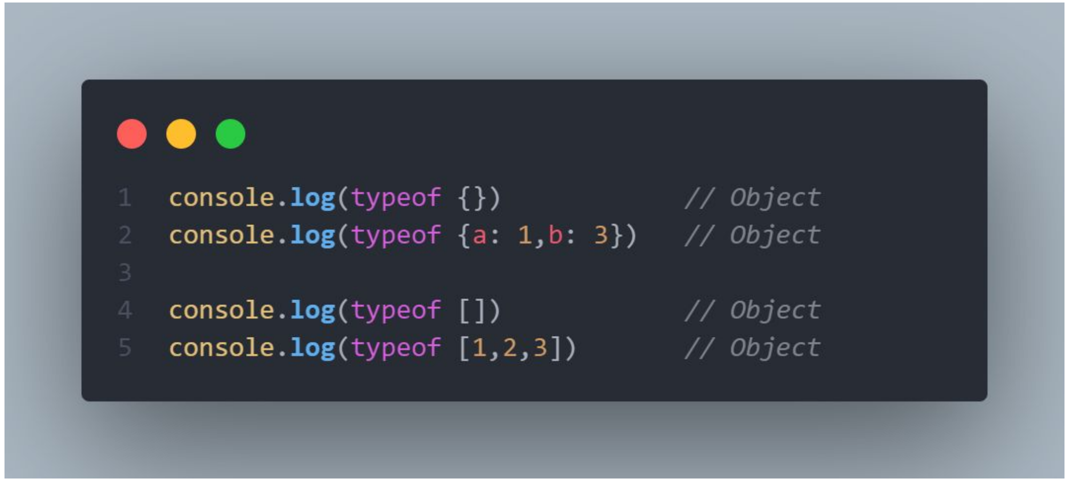
This is a known Javascript bug that only appears when you do the “type of” it. However it will act as a primitive type:



```
// Javascript known bug
console.log(null  === null)  // ??
```

OBJECTS 1

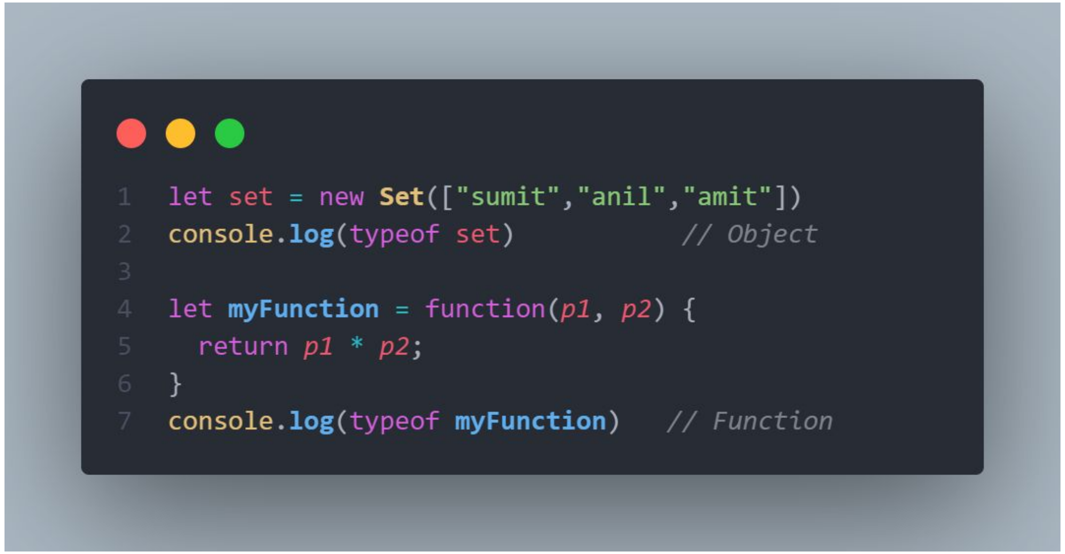
- Objects
- Arrays



```
console.log(typeof {})            // Object
console.log(typeof {a: 1,b: 3})   // Object

console.log(typeof [])            // Object
console.log(typeof [1,2,3])       // Object
```

OBJECTS 2

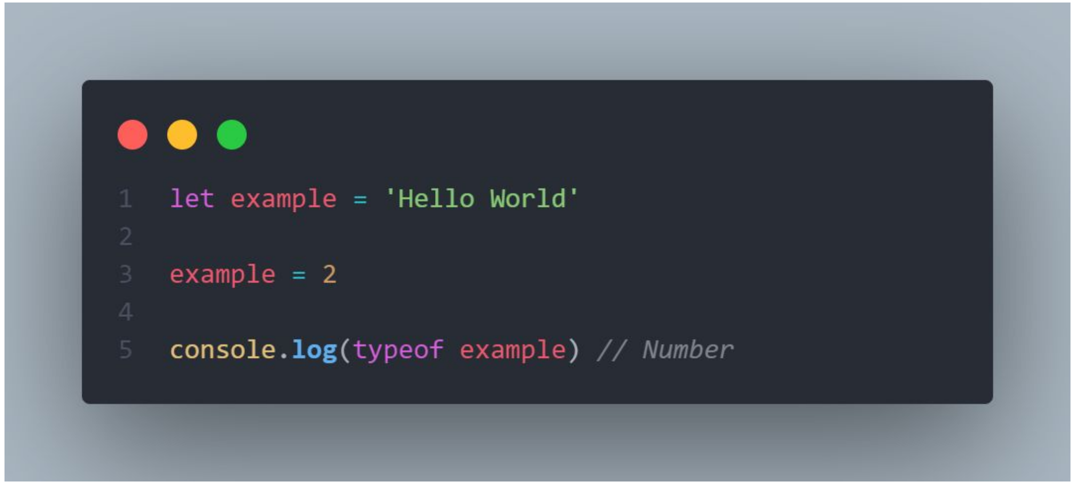
- Sets
- Functions*



```
let set = new Set(["sumit","anil","amit"])
console.log(typeof set)            // Object

let myFunction = function(p1, p2) {
  return p1 * p2;
}
console.log(typeof myFunction)   // Function
```

WHAT TYPE OF LANGUAGE IS JS? 1

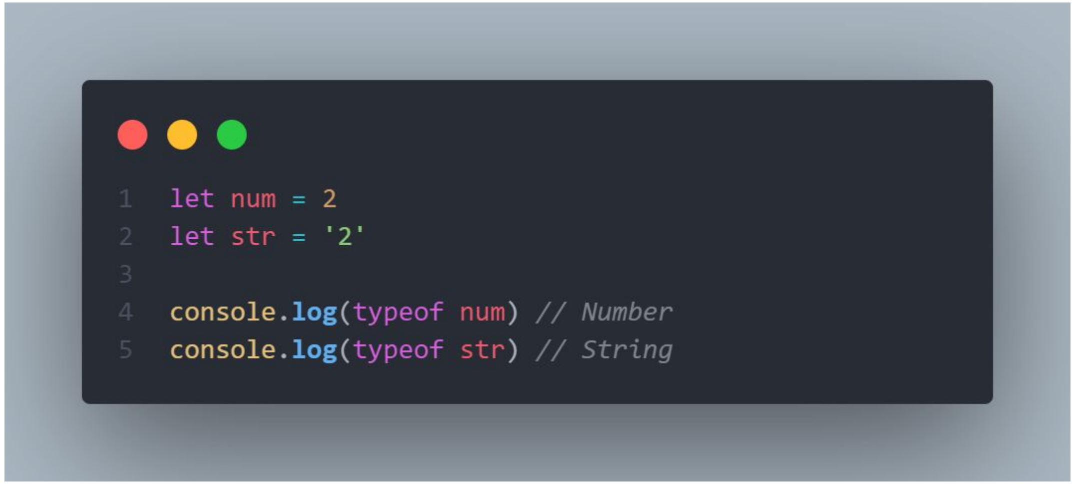
It is a dynamic and weakly typed language. This means, as you should know from LPP, that we can change the type of a variable at runtime:



```javascript
let example = 'Hello World'

example = 2

console.log(typeof example) // Number
```

WHAT TYPE OF LANGUAGE IS JS? 2

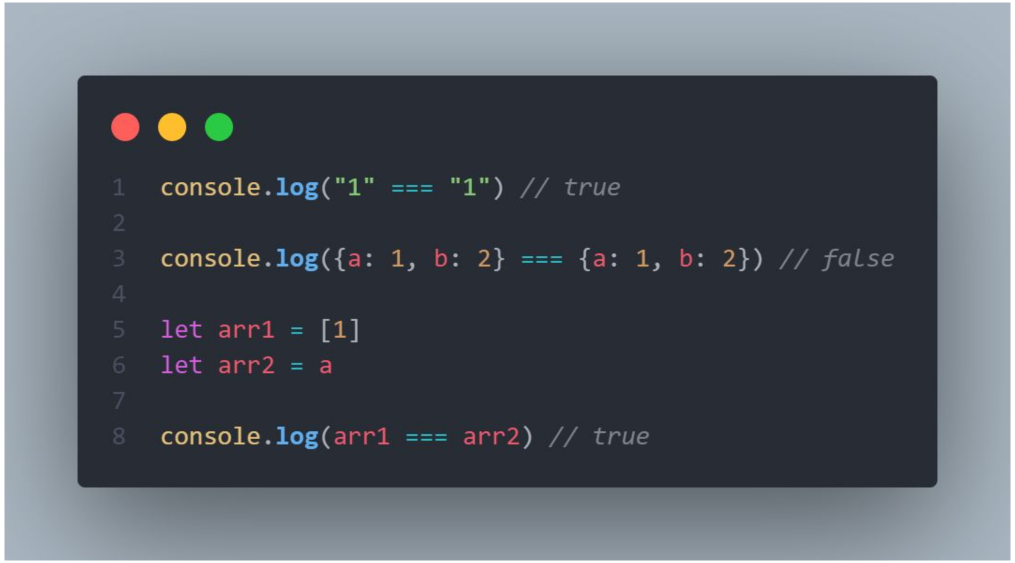
It is weakly typed since it is not necessary to specify the type of a variable (unlike Typescript):



```
let num = 2
let str = '2'

console.log(typeof num) // Number
console.log(typeof str) // String
```

DIFFERENCE BETWEEN THE TWO

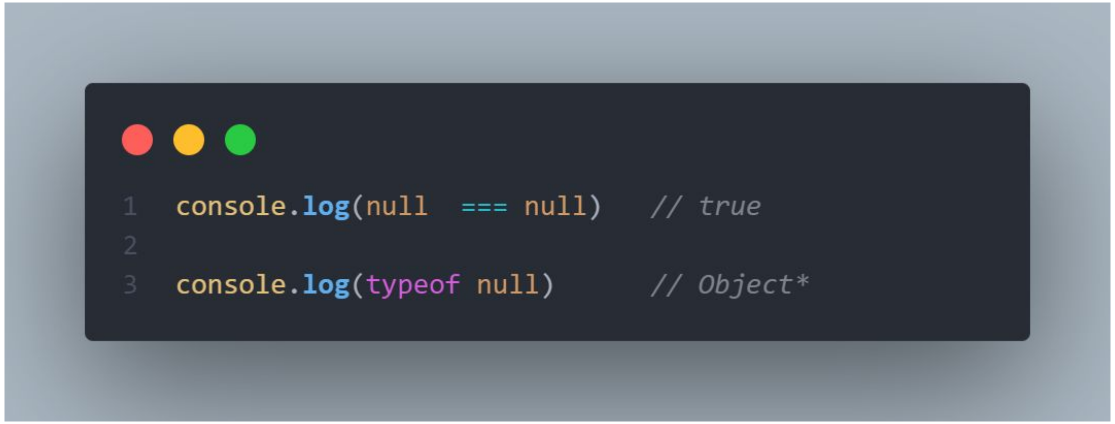
Primitives are compared by value, while objects are compared by reference (by memory address):



```
console.log("1" === "1") // true

console.log({a: 1, b: 2} === {a: 1, b: 2}) // false

let arr1 = [1]
let arr2 = a

console.log(arr1 === arr2) // true
```

SPECIAL CASE (NULL)

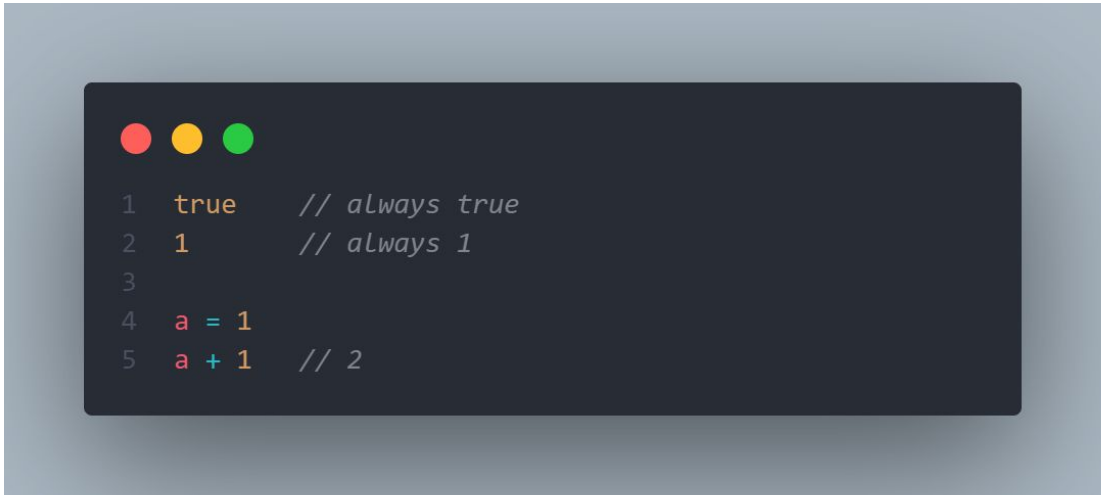
In the particular case of null, it behaves like a primitive but when doing the typeof it returns object:



```
console.log(null  === null)   // true

console.log(typeof null)      // Object*
```

IMMUTABLE

Primitives are immutable, that is, you cannot change their value, unlike objects:



```
true     // always true
1        // always 1

a = 1
a + 1    // 2
```

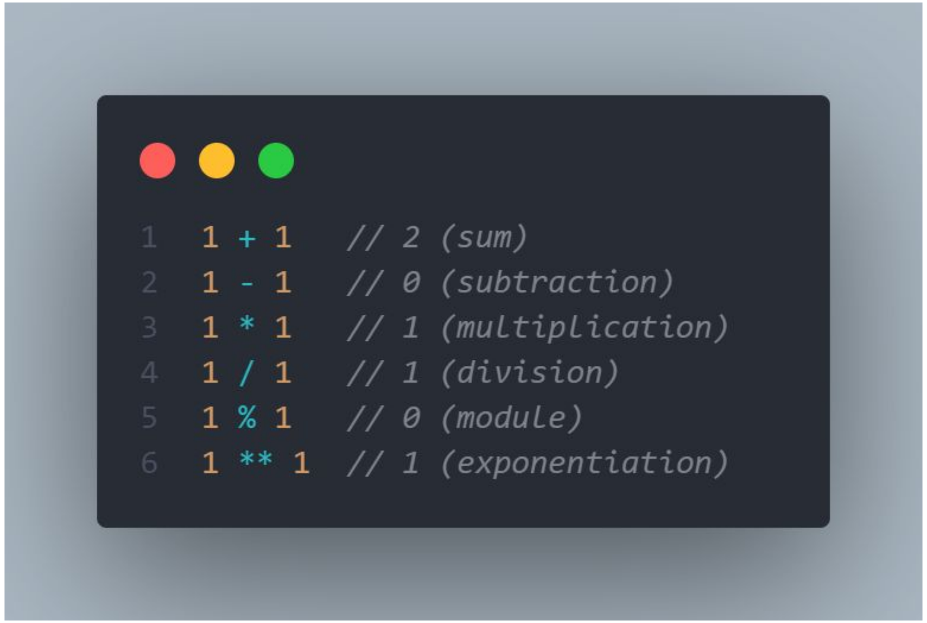
01

OPERATORS

Binary, unary, ternary, type coercion, …

BINARY OPERATORS 1

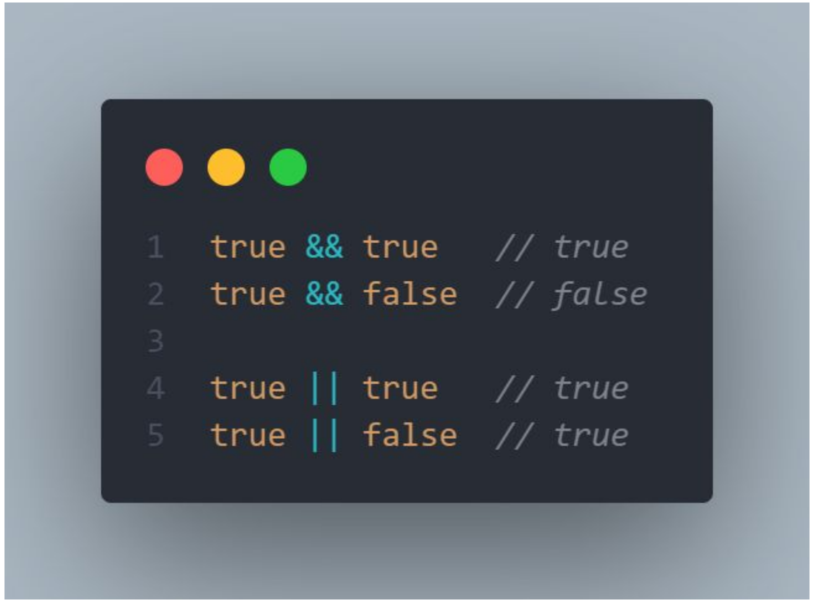
Mathematical



```
1 + 1   // 2 (sum)
1 - 1   // 0 (subtraction)
1 * 1   // 1 (multiplication)
1 / 1   // 1 (division)
1 % 1   // 0 (module)
1 ** 1  // 1 (exponentiation)
```

BINARY OPERATORS 2

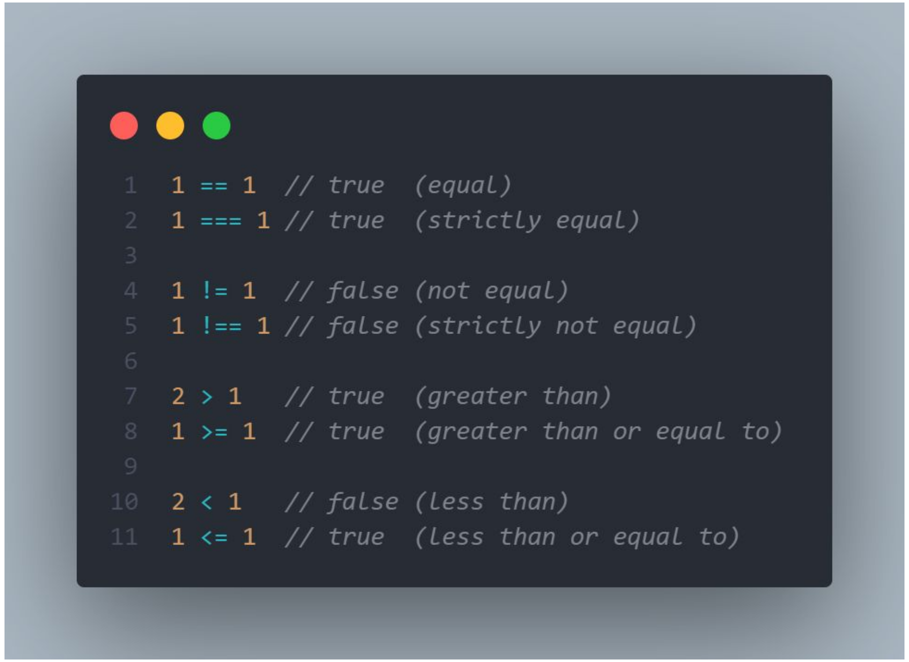
Logic



```
true && true   // true
true && false  // false

true || true   // true
true || false  // true
```

BINARY OPERATORS 3

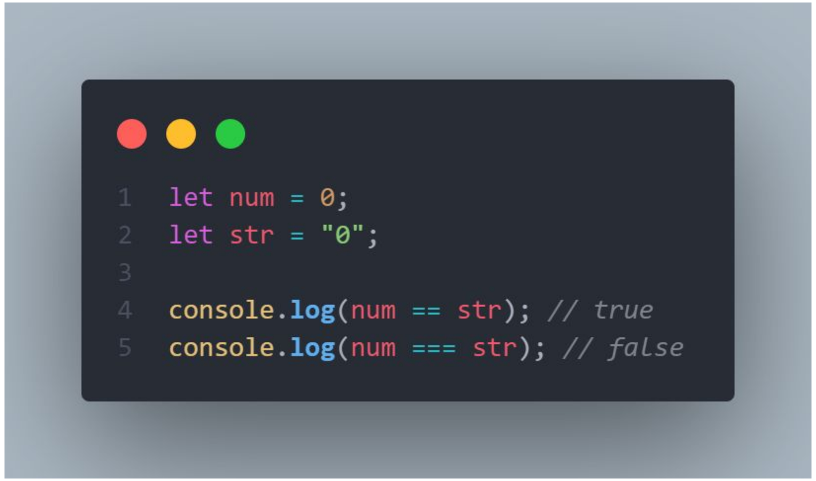
Comparison



```
1 == 1  // true  (equal)
1 === 1 // true  (strictly equal)

1 != 1  // false (not equal)
1 !== 1 // false (strictly not equal)

2 > 1   // true  (greater than)
1 >= 1  // true  (greater than or equal to)

2 < 1   // false (less than)
1 <= 1  // true  (less than or equal to)
```

DIFFERENCES BETWEEN == AND ===

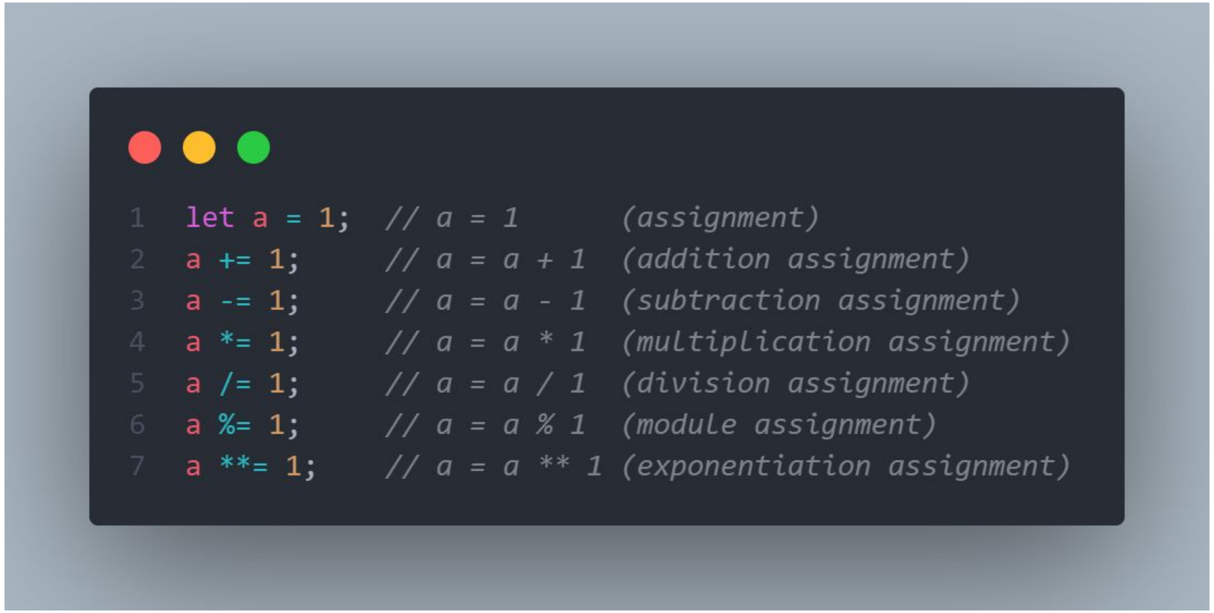
The difference is that while with the == before making the comparison, both data are converted to a common type:



```javascript
let num = 0;
let str = "0";

console.log(num == str); // true
console.log(num === str); // false
```

BINARY OPERATORS 4

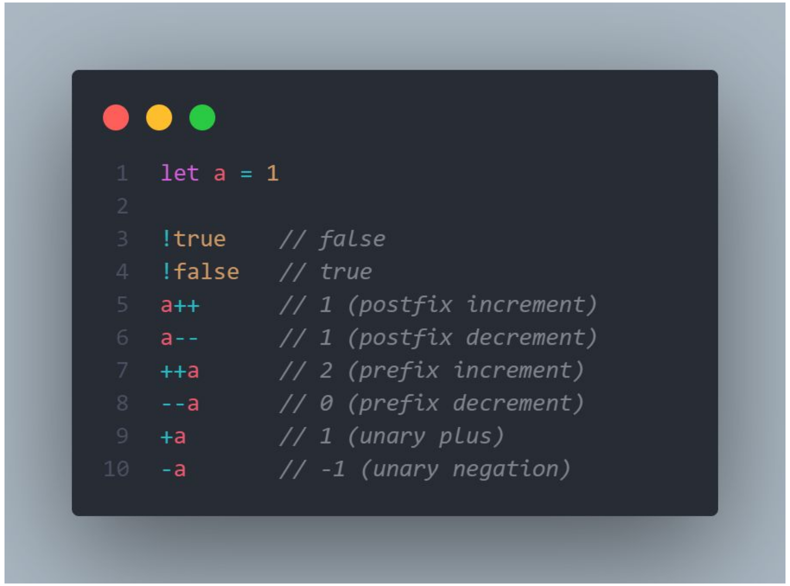
Assignment



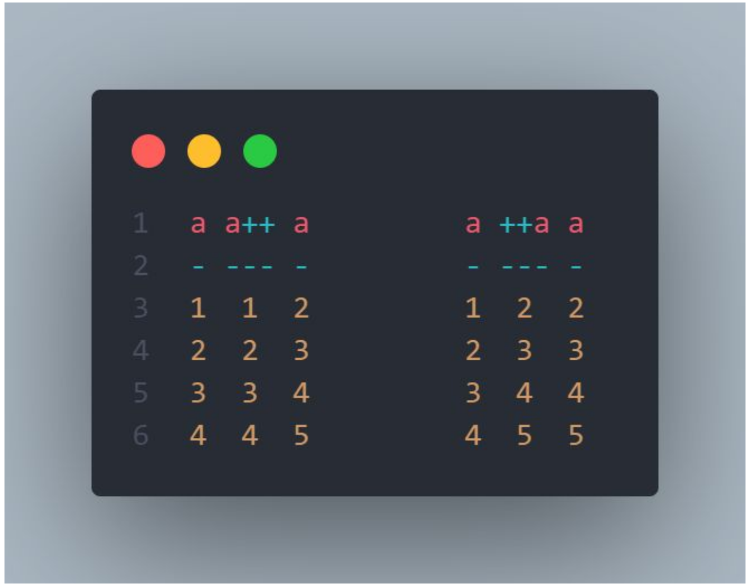
```
let a = 1;  // a = 1      (assignment)
a += 1;     // a = a + 1  (addition assignment)
a -= 1;     // a = a - 1  (subtraction assignment)
a *= 1;     // a = a * 1  (multiplication assignment)
a /= 1;     // a = a / 1  (division assignment)
a %= 1;     // a = a % 1  (module assignment)
a **= 1;    // a = a ** 1 (exponentiation assignment)
```

UNARY OPERATORS

Increment, decrement, etc:

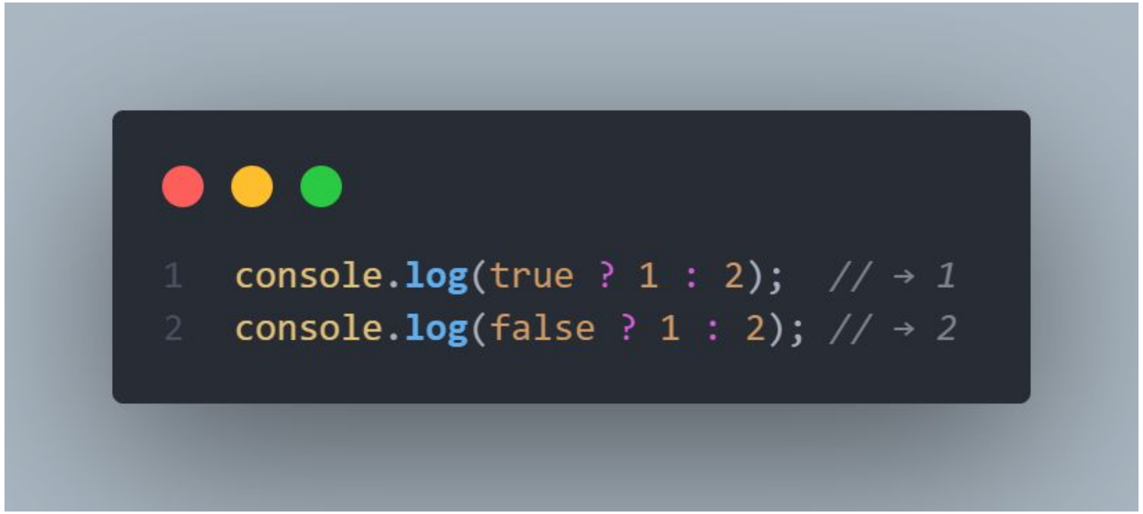


```
let a = 1

!true    // false
!false   // true
a++      // 1 (postfix increment)
a--      // 1 (postfix decrement)
++a      // 2 (prefix increment)
--a      // 0 (prefix decrement)
+a       // 1 (unary plus)
-a       // -1 (unary negation)
```



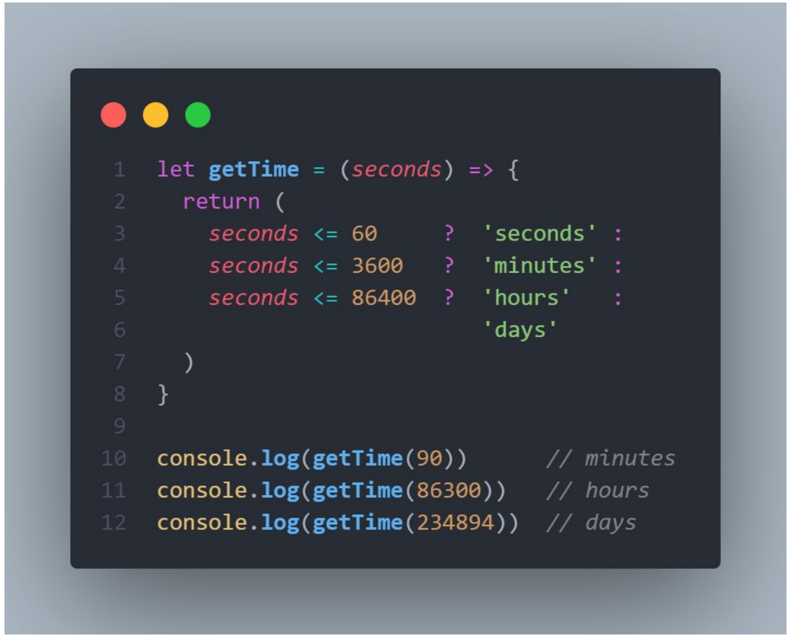
```
a a++ a      a ++a a
- --- -      - --- -
1  1  2      1  2  2
2  2  3      2  3  3
3  3  4      3  4  4
4  4  5      4  5  5
```

TERNARY OPERATOR

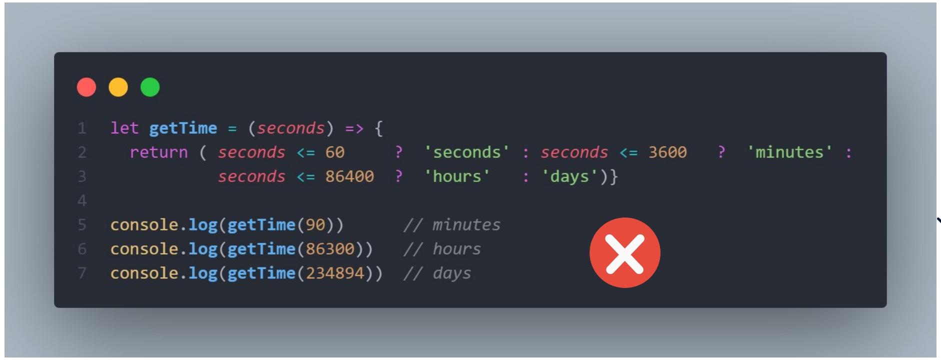

```
console.log(true ? 1 : 2);  // → 1
console.log(false ? 1 : 2); // → 2
```

NESTED TERNARY OPERATOR

Not recommended unless:



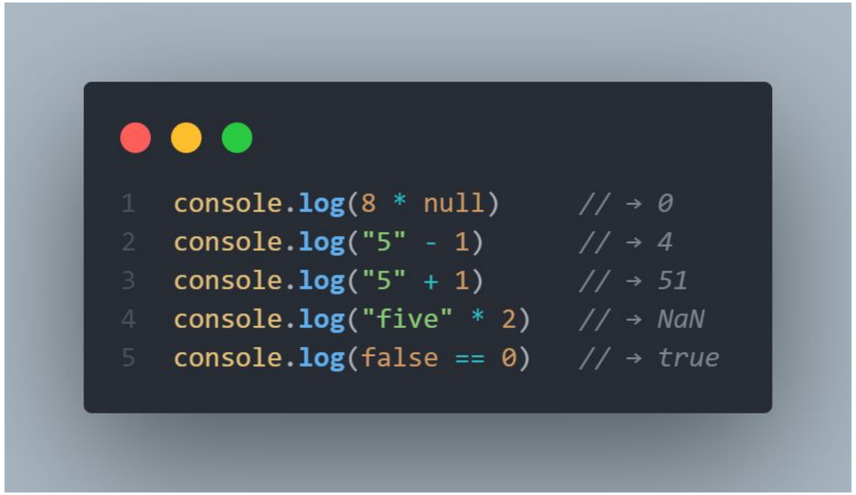
```
let getTime = (seconds) => {
  return (
    seconds <= 60     ?  'seconds' :
    seconds <= 3600   ?  'minutes' :
    seconds <= 86400  ?  'hours'   :
                         'days'
  )
}

console.log(getTime(90))      // minutes
console.log(getTime(86300))   // hours
console.log(getTime(234894))  // days
```



```
let getTime = (seconds) => {
  return ( seconds <= 60     ?  'seconds' : seconds <= 3600   ?  'minutes' :
           seconds <= 86400  ?  'hours'   : 'days')}

console.log(getTime(90))      // minutes
console.log(getTime(86300))   // hours
console.log(getTime(234894))  // days
```

TYPE COERCION (JS IS WEIRD) 1

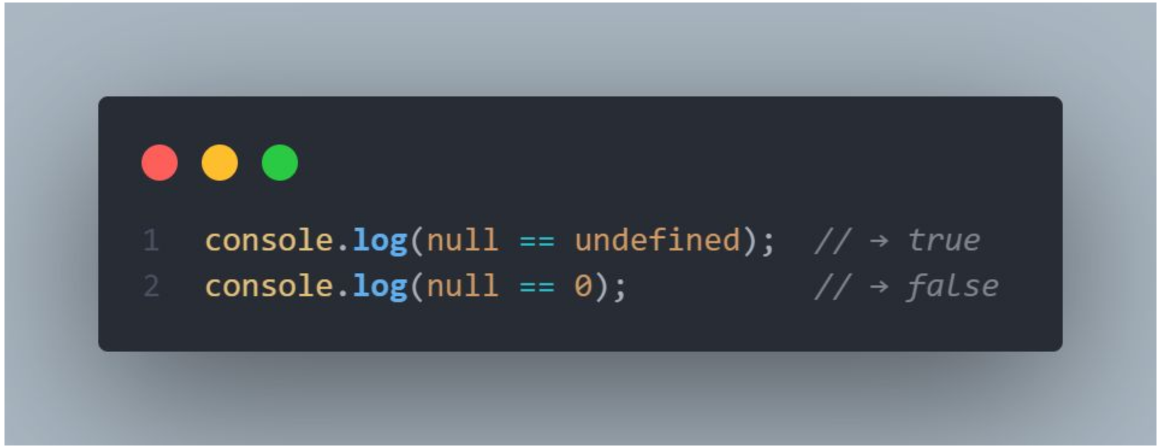
When an operator is applied to the "wrong" value type, JavaScript will silently cast that value to the type it needs, using a set of rules that are often not what you want or expect.



```
console.log(8 * null)       // → 0
console.log("5" - 1)        // → 4
console.log("5" + 1)        // → 51
console.log("five" * 2)     // → NaN
console.log(false == 0)     // → true
```

TYPE COERCION (JS IS WEIRD) 2

When null or undefined appears on either side of the == operator, it produces true only if both sides are one of null or undefined.

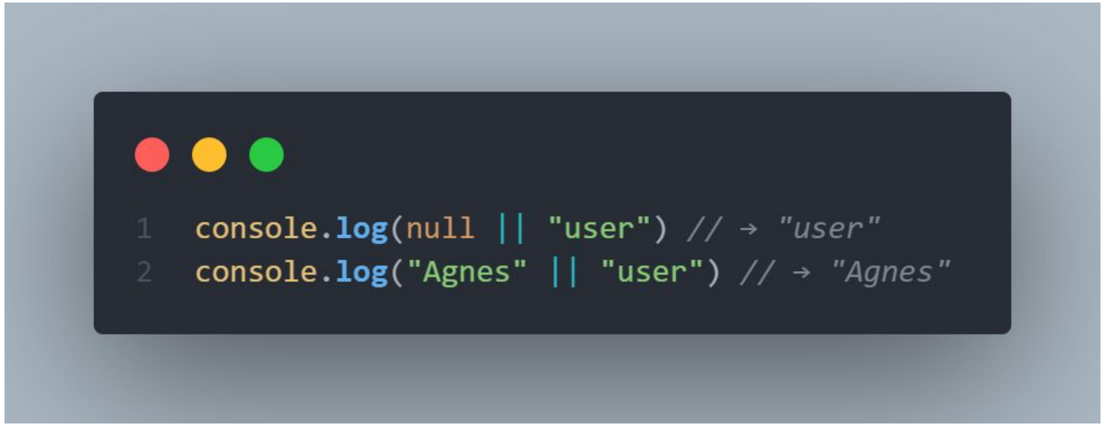


```
console.log(null == undefined);  // → true
console.log(null == 0);          // → false
```

TYPE COERCION (JS IS WEIRD) 3

Logic operators:

- || -> It will return the operand on the left if it can be converted to true, otherwise it will return the operand on the right.

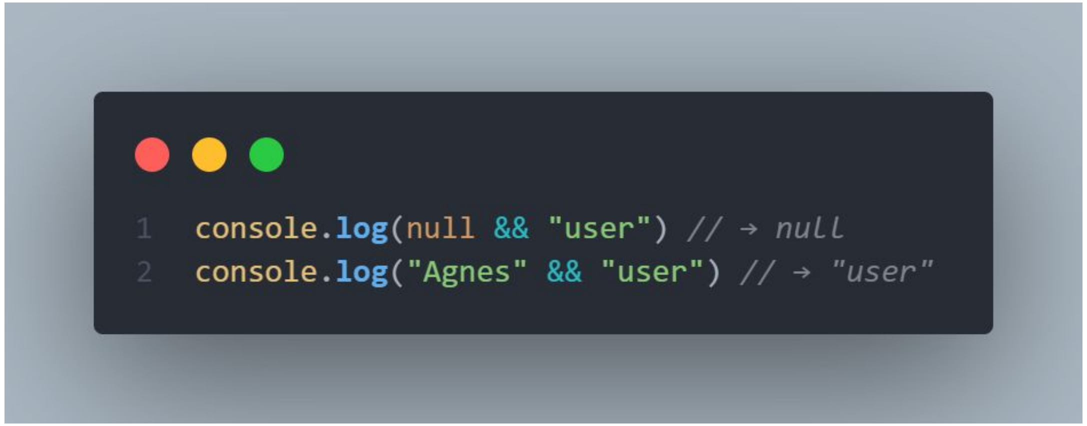


```
console.log(null || "user") // → "user"
console.log("Agnes" || "user") // → "Agnes"
```

TYPE COERCION (JS IS WEIRD) 4

Logic operators:

- && -> It will return the operand on the left if it can be converted to false, otherwise it will return the operand on the right.

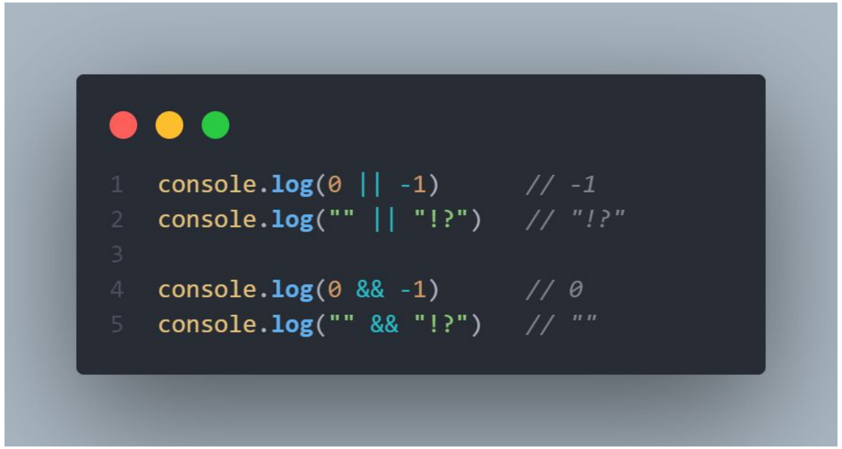


```
console.log(null && "user") // → null
console.log("Agnes" && "user") // → "user"
```

WHAT ARE THESE VALUES?

Values that can NOT be converted to booleans and therefore return false:

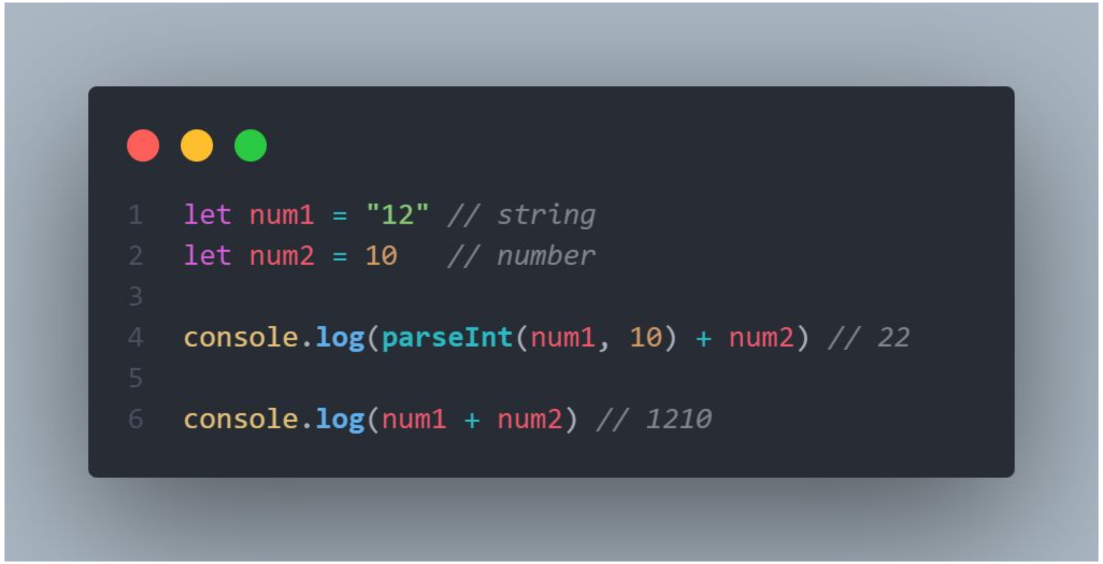
- "" → Empty string
- 0
- NaN



```
console.log(0 || -1)      // -1
console.log("" || "!?")   // "!?"

console.log(0 && -1)      // 0
console.log("" && "!?")   // ""
```

AVOID TYPE COERCION 1

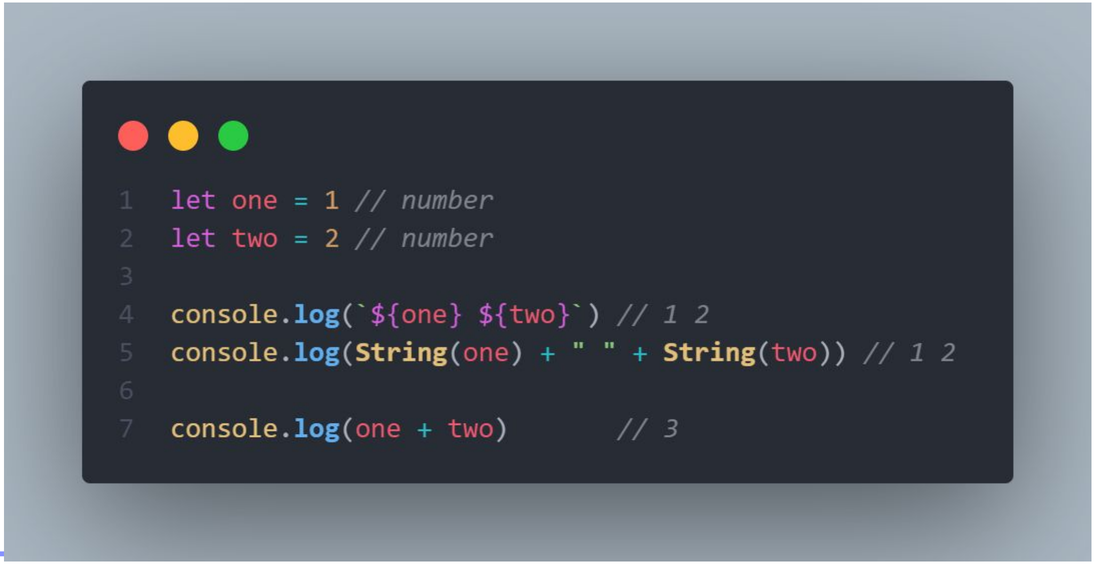
If you do not want to use this automatic type conversion, you must use explicit conversions with mathematical operations:



```
let num1 = "12" // string
let num2 = 10   // number

console.log(parseInt(num1, 10) + num2) // 22

console.log(num1 + num2) // 1210
```

AVOID TYPE COERCION 2

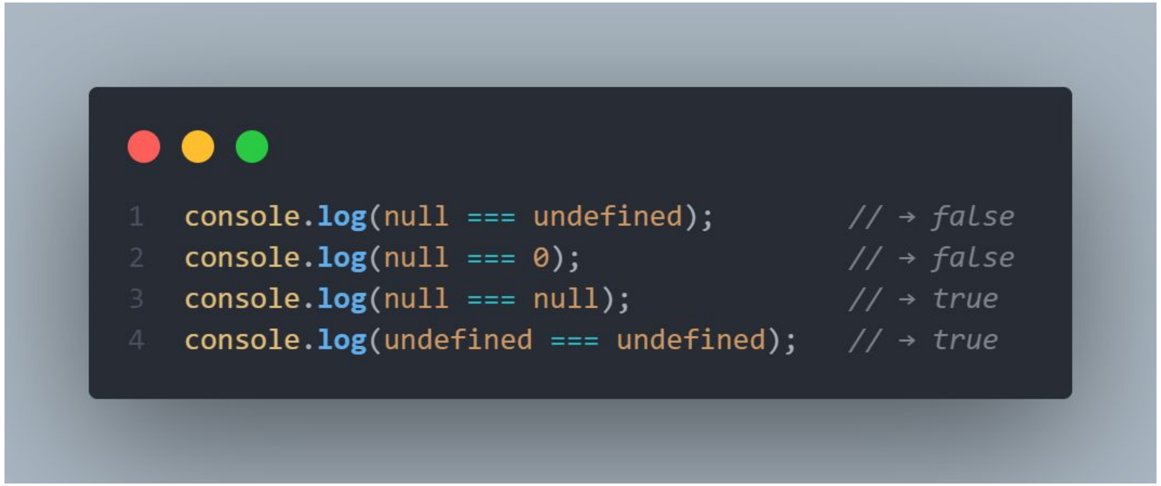
Concatenate string using template literals instead of +:



```javascript
let one = 1 // number
let two = 2 // number

console.log(`${one} ${two}`) // 1 2
console.log(String(one) + " " + String(two)) // 1 2

console.log(one + two)       // 3
```

AVOID TYPE COERCION 3

Use strict comparison (===) when comparing values:



```
console.log(null === undefined);            // → false
console.log(null === 0);                    // → false
console.log(null === null);                 // → true
console.log(undefined === undefined);       // → true
```

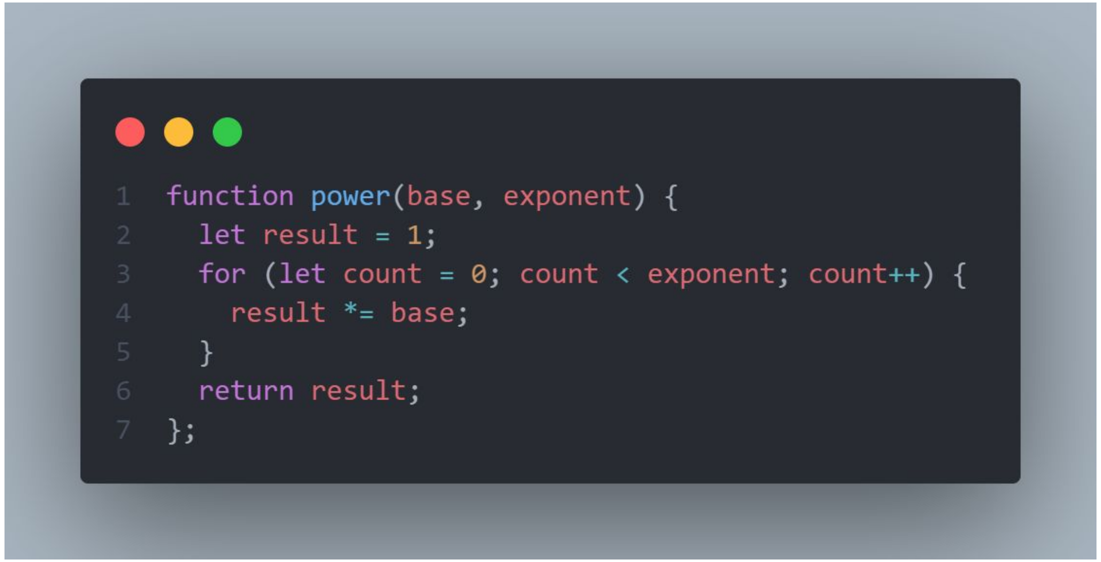
02

FUNCTIONS

Function declaration, function expression and arrow functions

FUNCTION DECLARATION 1

The *function* keyword goes first, then goes the name of the function, then a list of parameters between the parentheses and finally the code of the function, also named “the function body”, between curly braces.
Example:

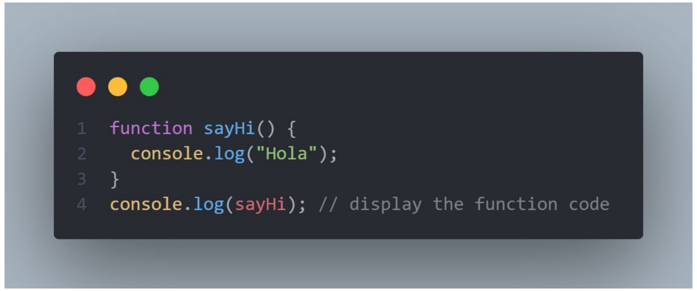


```
function power(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

FUNCTION DECLARATION 2

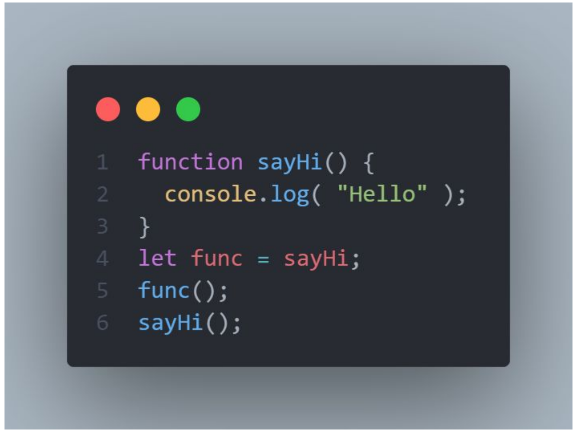
In JavaScript, a **function** is not a “magical language structure”, but a **special kind of value**.

Print out



```
function sayHi() {
  console.log("Hola");
}
console.log(sayHi); // display the function code
```

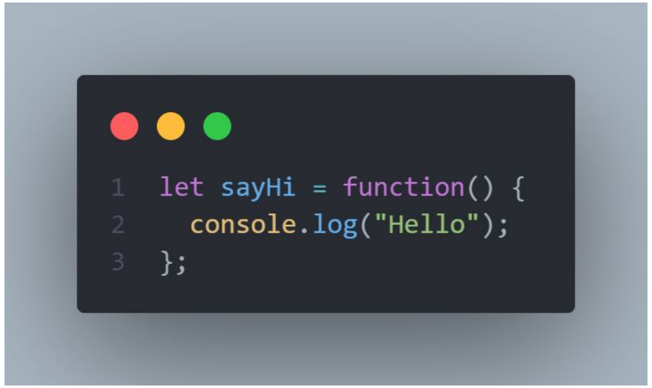
Copy into a variable



```
function sayHi() {
  console.log( "Hello" );
}
let func = sayHi;
func();
sayHi();
```

FUNCTION EXPRESSION

There is another syntax for creating a function that is called a *Function Expression*.
It allows us to create a new function **in the middle** of any expression.

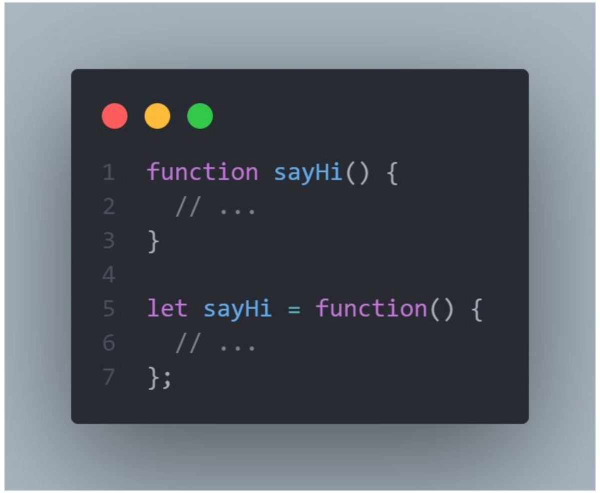


```
let sayHi = function() {
  console.log("Hello");
};
```

Here we immediately assign it to the variable, so the meaning of these code samples is the same: "create a function and put it into the variable sayHi".

WHY IS THERE A SEMICOLON AT THE END?

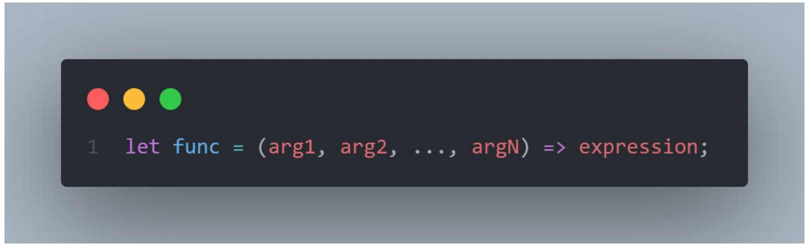
Google style guide say the semicolon is recommended at the end of the statement and since this is basically an assignment to a variable, it should be put. So, it’s not a part of the function syntax.



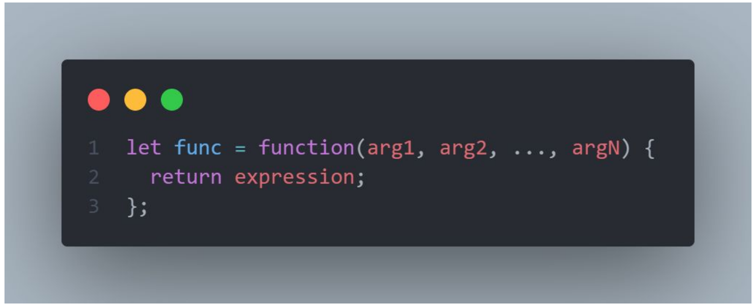
```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

ARROW FUNCTION 1

Instead of the *function* keyword, it uses an arrow (`=>`). The arrow comes **after** the list of parameters and is followed by the function's body. It expresses something like "this input (the parameters) produces this result (the body)".
Syntax example:



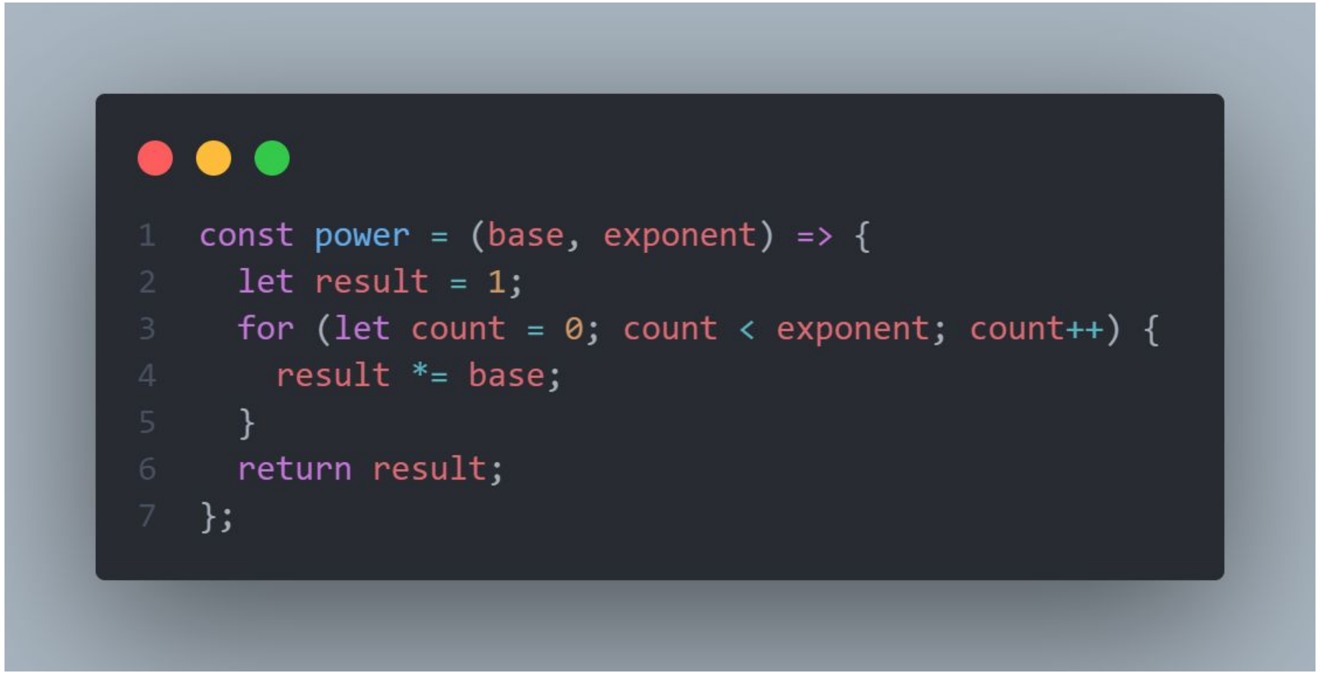
```
let func = (arg1, arg2, ..., argN) => expression;
```



```
let func = function(arg1, arg2, ..., argN) {
  return expression;
};
```

ARROW FUNCTION 2

Example:



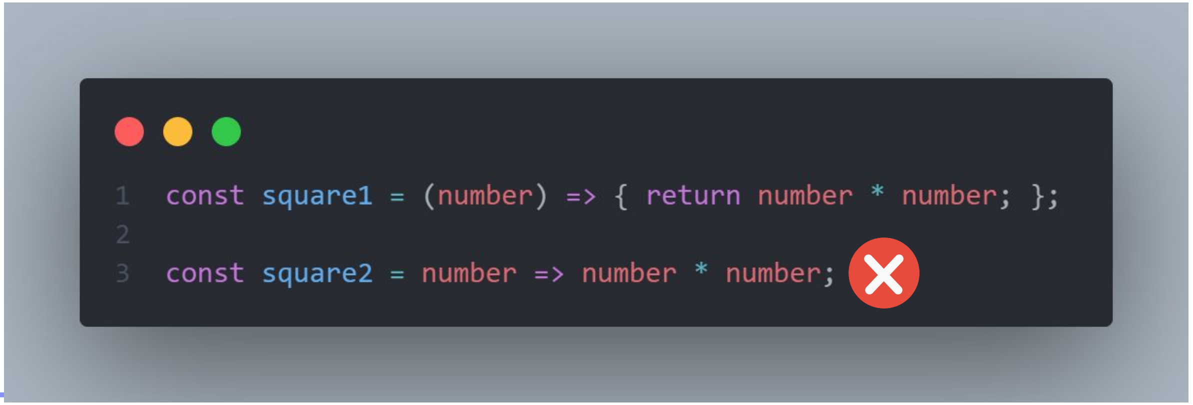
```javascript
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

ARROW FUNCTION 3

Notably:

- When there is only **one parameter name**, you can **omit the parentheses** around the parameter list.
- If the body is a **single expression**, rather than a block in braces, that expression will be **returned from the function**.

Example:



```
const square1 = (number) => { return number * number; };

const square2 = number => number * number;
```

FUNCTIONS SUMMARY

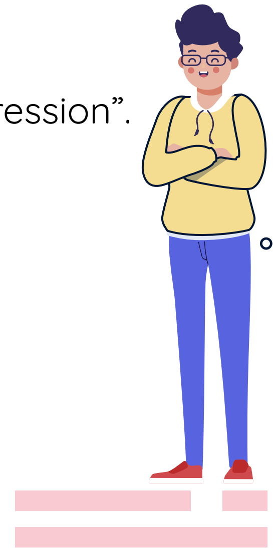
- Functions are values.They can be assigned, copied or declared in any place of the code.

- If the function is declared as a separate statement in the main code flow, that’s called a “Function Declaration”.

- If the function is created as a part of an expression, it’s called a “Function Expression”.

- Function Declarations are processed before the code block is executed.

- Function Expressions are created when the execution flow reaches them.



03

OOP

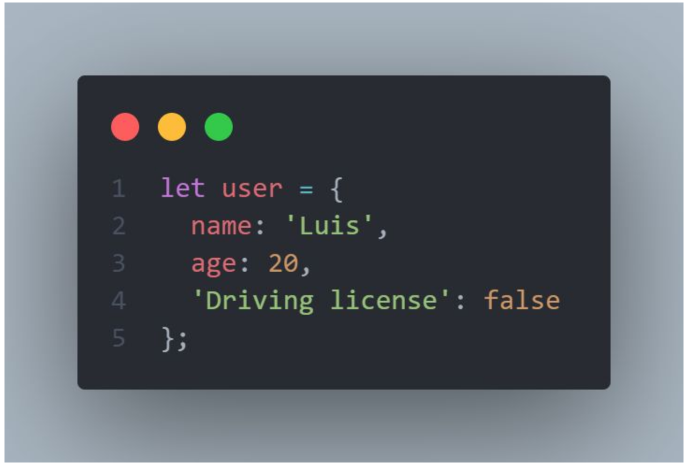
Object, method and classes

OBJECTS 1

An *object* can be created with figure brackets {…} with an optional list of properties.

A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

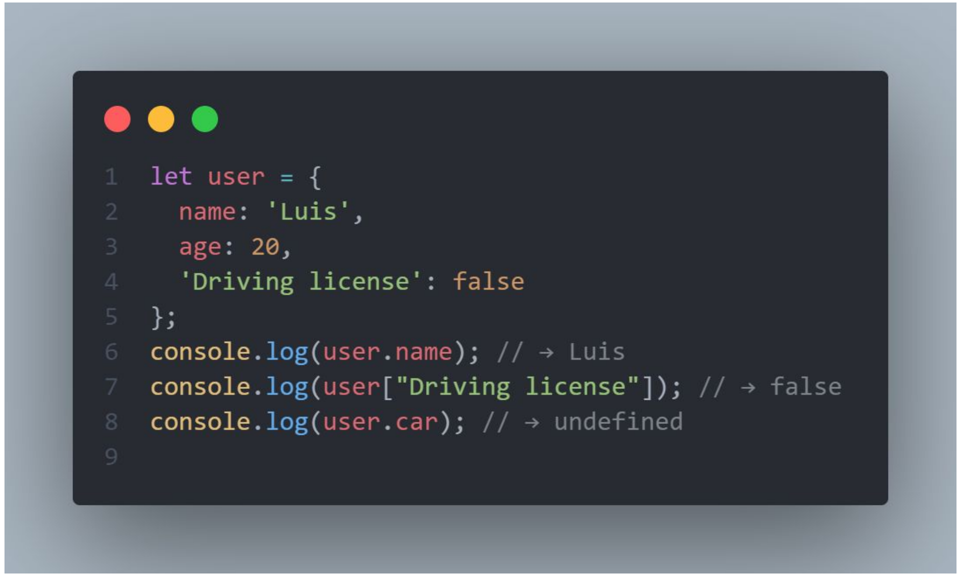
Example:



```
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
```

OBJECTS 2

- Square brackets for multiword properties.
- Reading a property that doesn't exist will give you the value *undefined*.

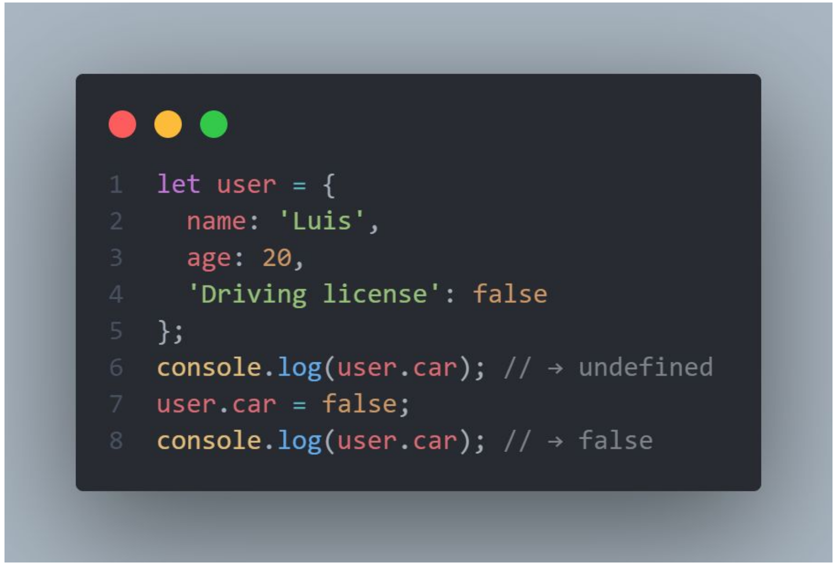


```
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
console.log(user.name); // → Luis
console.log(user["Driving license"]); // → false
console.log(user.car); // → undefined
```

OBJECTS 3

= operator

This will replace the property's value if it already existed or create a new property on the object if it didn't.

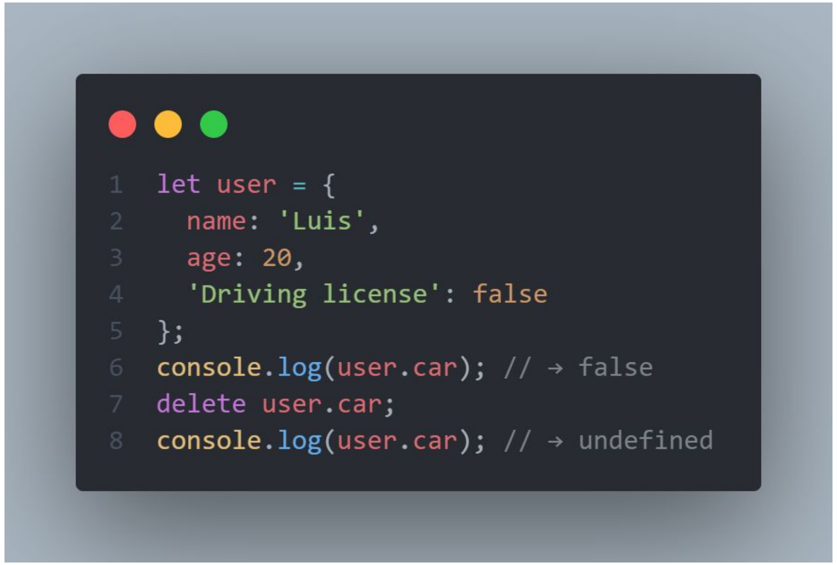


```
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
console.log(user.car); // → undefined
user.car = false;
console.log(user.car); // → false
```

OBJECTS 4

Delete operator

It is a unary operator that, when applied to an object property, will remove the named property from the object. This is not a common thing to do, but it is possible.

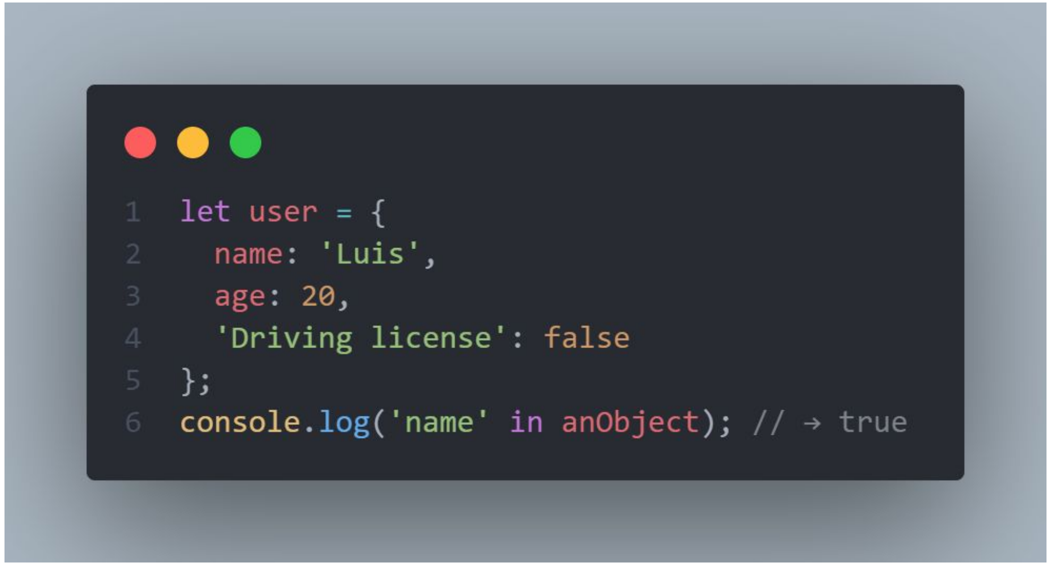


```javascript
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
console.log(user.car); // → false
delete user.car;
console.log(user.car); // → undefined
```

OBJECTS 5

In operator

It is a binary operator that, when applied to a string and an object, tells you whether that object has a property with that name.

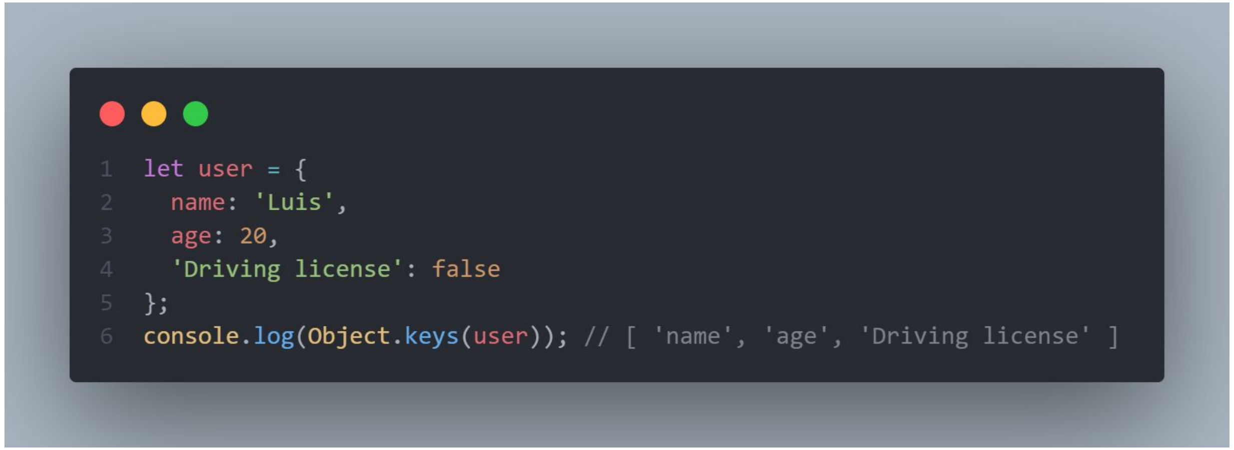


```
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
console.log('name' in anObject); // → true
```

OBJECTS 6

Keys function

To find out what properties an object has, you can use the *Object.keys* function. You give it an object, and it returns an array of strings—the object's property names.

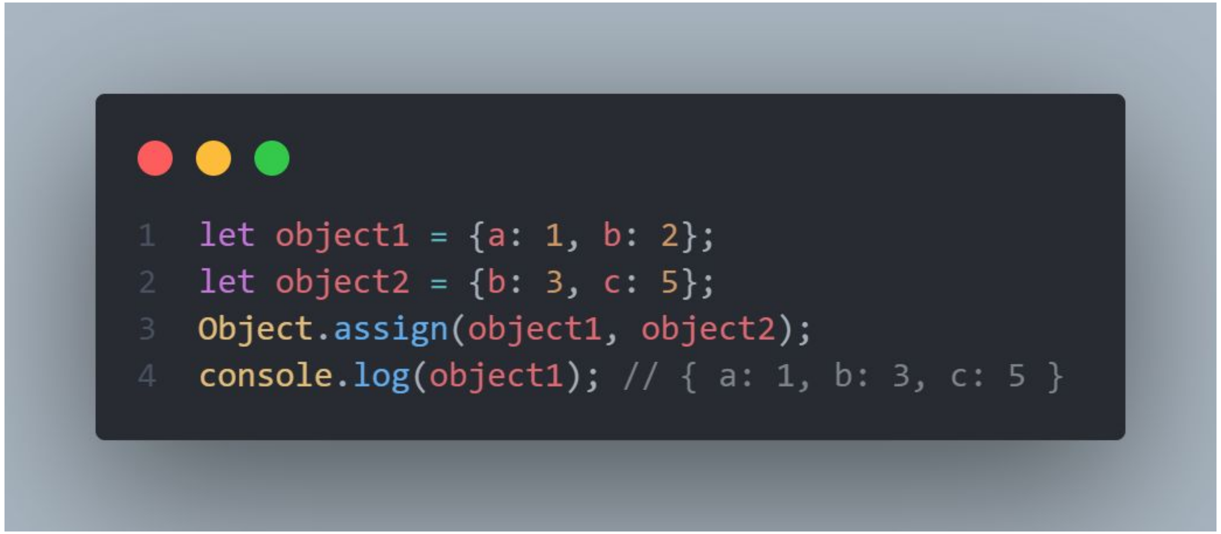


```
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
console.log(Object.keys(user)); // [ 'name', 'age', 'Driving license' ]
```

OBJECTS 7

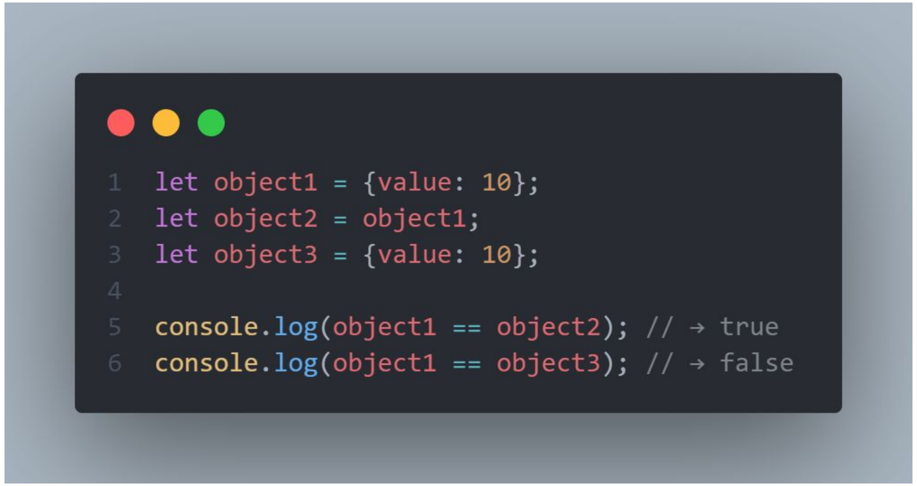
Object.assign

Function that copies all properties from one object into another.


```
let object1 = {a: 1, b: 2};
let object2 = {b: 3, c: 5};
Object.assign(object1, object2);
console.log(object1); // { a: 1, b: 3, c: 5 }
```

Mutability

With objects, there is a difference between having two references to the same object and having two different objects that contain the same properties. Consider the following code:

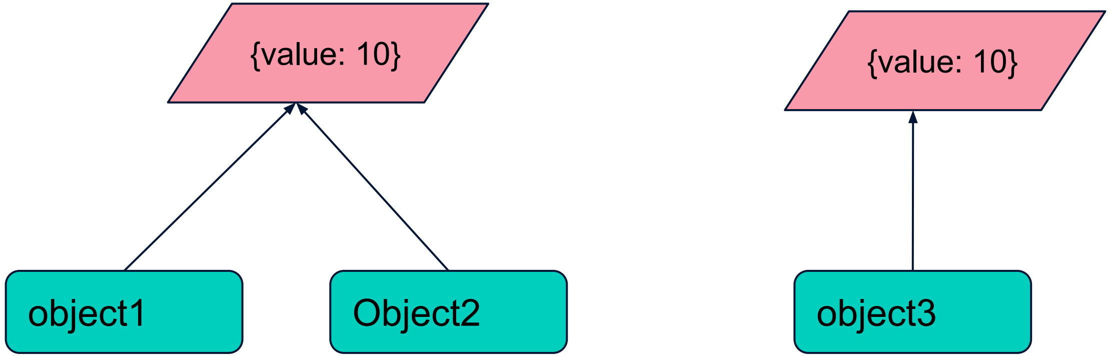


```javascript
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2); // → true
console.log(object1 == object3); // → false
```

OBJECTS 9

Mutability

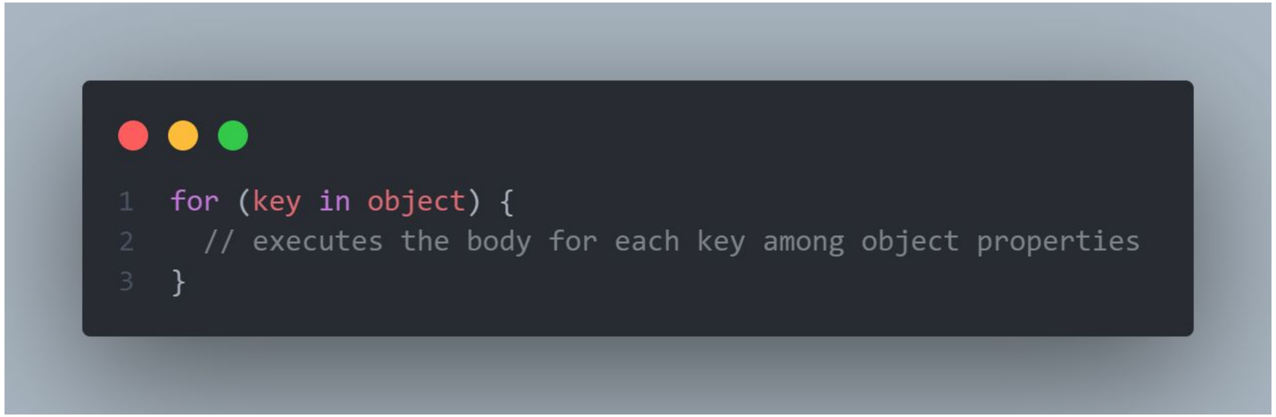
Explanation: The *object1* and *object2* bindings grasp the **same** object, which is why changing *object1* also changes the value of *object2*. They are said to have the same **identity**. The binding *object3* points to a different object, which initially contains the same properties as *object1* but lives a separate life.



OBJECTS 10

For … in loop

To walk over all keys of an object, there exists a special form of the loop: `for..in`. The syntax:

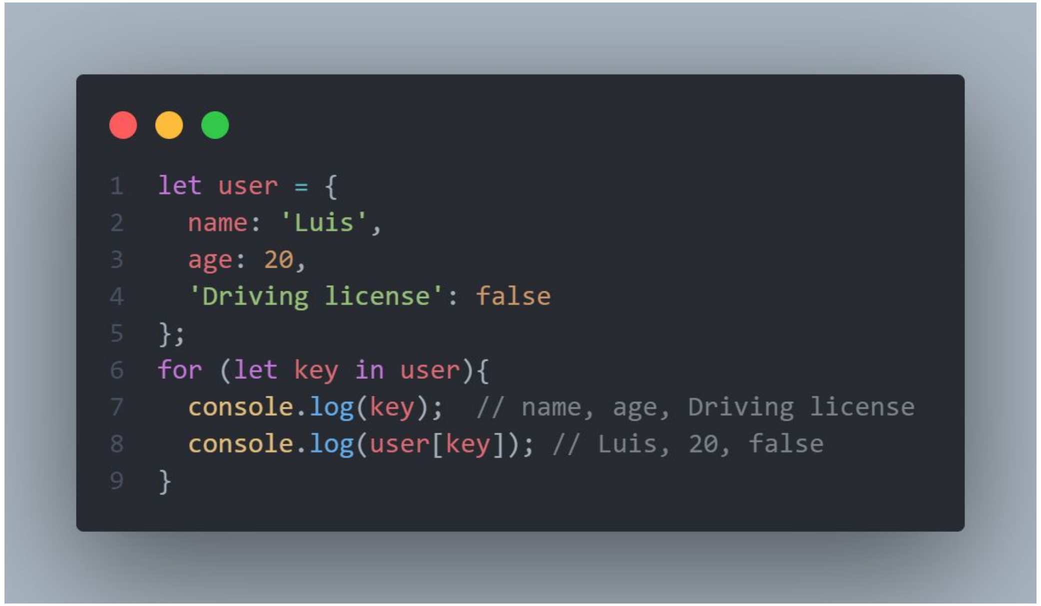


```
for (key in object) {
  // executes the body for each key among object properties
}
```

OBJECTS 11

For … in loop

Example:

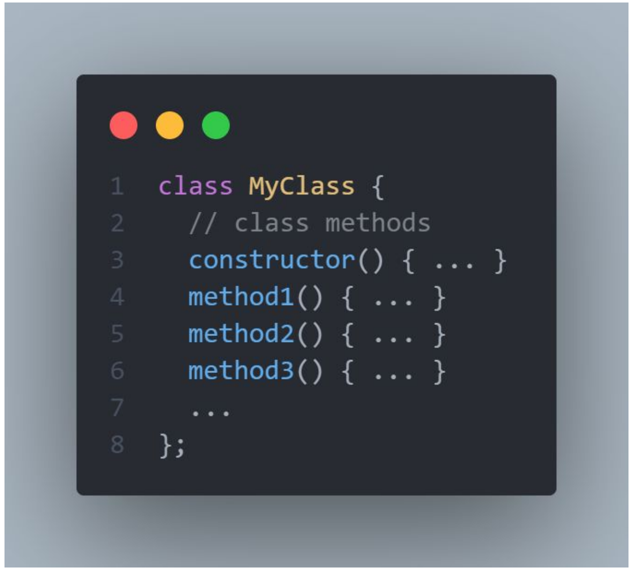


```
let user = {
  name: 'Luis',
  age: 20,
  'Driving license': false
};
for (let key in user){
  console.log(key);  // name, age, Driving license
  console.log(user[key]); // Luis, 20, false
}
```

CLASS 1

A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.

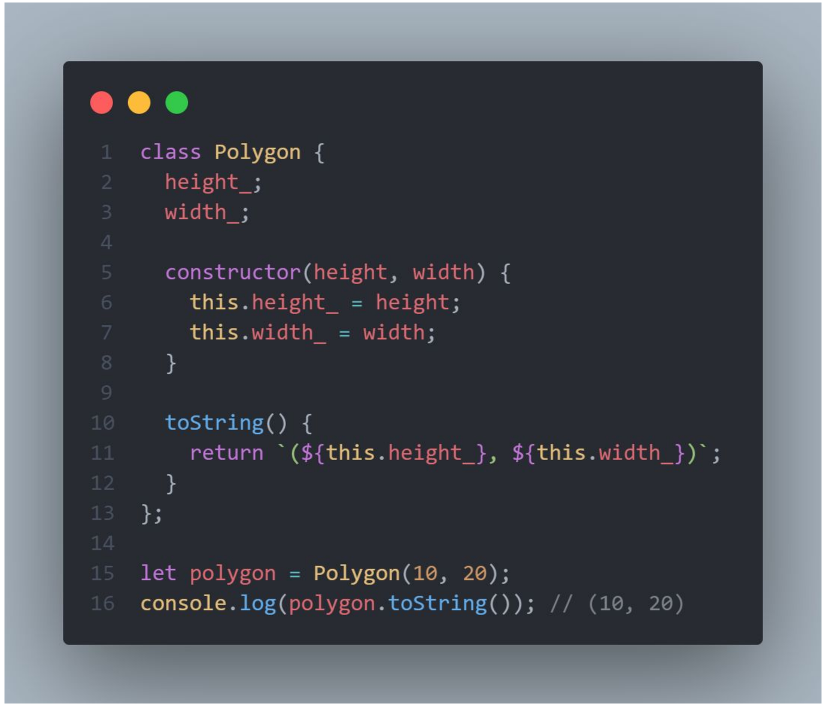
The basic syntax is:



```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
};
```

CLASS 2

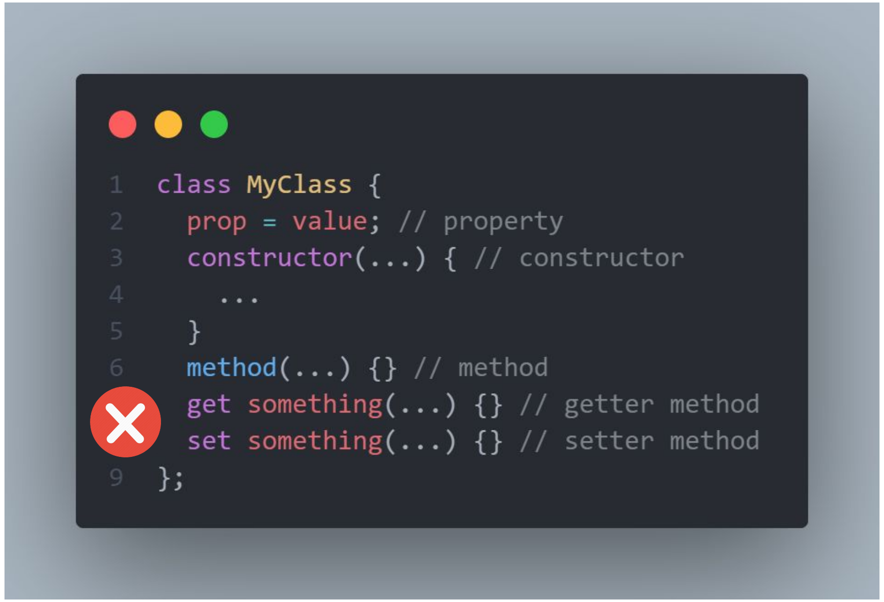
Example:



```
class Polygon {
  height_;
  width_;

  constructor(height, width) {
    this.height_ = height;
    this.width_ = width;
  }

  toString() {
    return `(${this.height_}, ${this.width_})`;
  }
};

let polygon = Polygon(10, 20);
console.log(polygon.toString()); // (10, 20)
```

CLASS 3

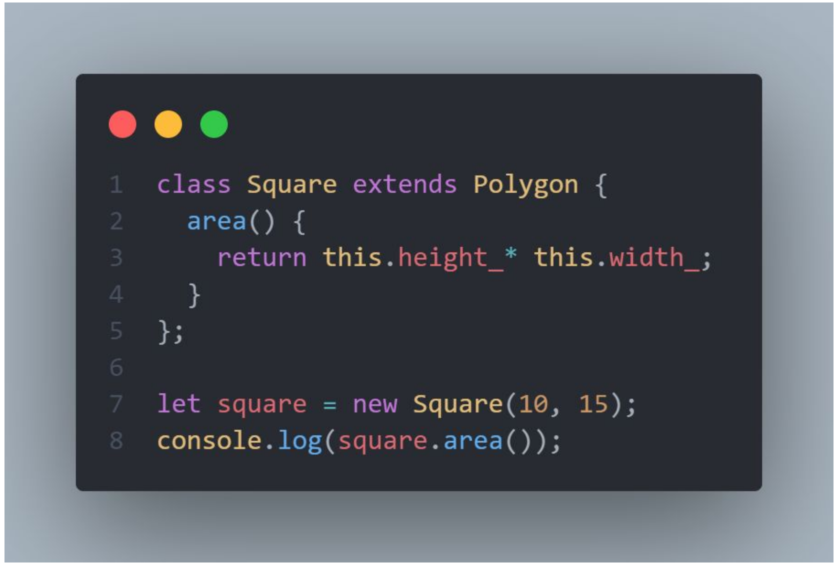
Do not use JavaScript getter and setter properties. They are potentially surprising and difficult to reason about, and have limited support in the compiler. Provide ordinary methods instead.



```
class MyClass {
  prop = value; // property
  constructor(...) { // constructor
    ...
  }
  method(...) {} // method
  get something(...) {} // getter method
  set something(...) {} // setter method
};
```

CLASS INHERITANCE

Class inheritance is a way for one class to extend another class. So we can create new functionality on top of the existing.
Example: Square class should be based on Polygon, have access to polygon methods, so that squares can do what “generic” polygons can do.



```
class Square extends Polygon {
  area() {
    return this.height_* this.width_;
  }
};

let square = new Square(10, 15);
console.log(square.area());
```

04

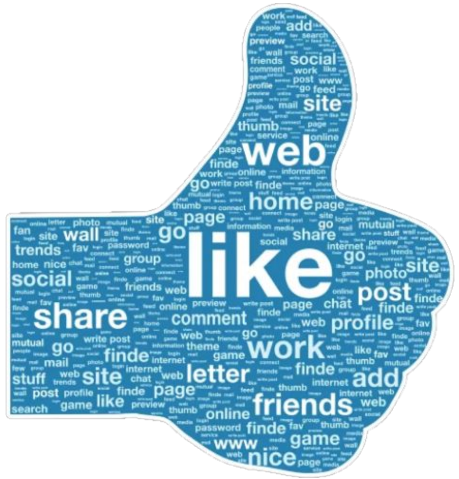
MODERN JS

Many fancy and cool things ...

WHAT IS IT?

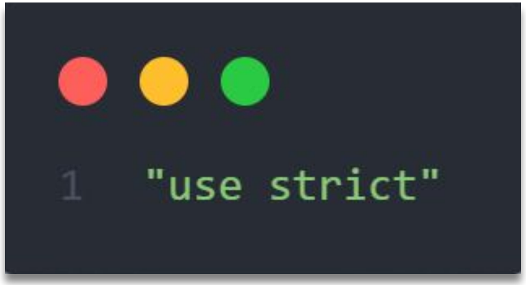
Modern JavaScript is a safe, latest update way of coding in JS.

In general, modern JavaScript is considered to be more concise, expressive, and efficient than older versions, and is designed to make it easier for developers to create dynamic, high-performing web applications.



WHERE DO WE START?

We should start by adding this at the first line of your code:



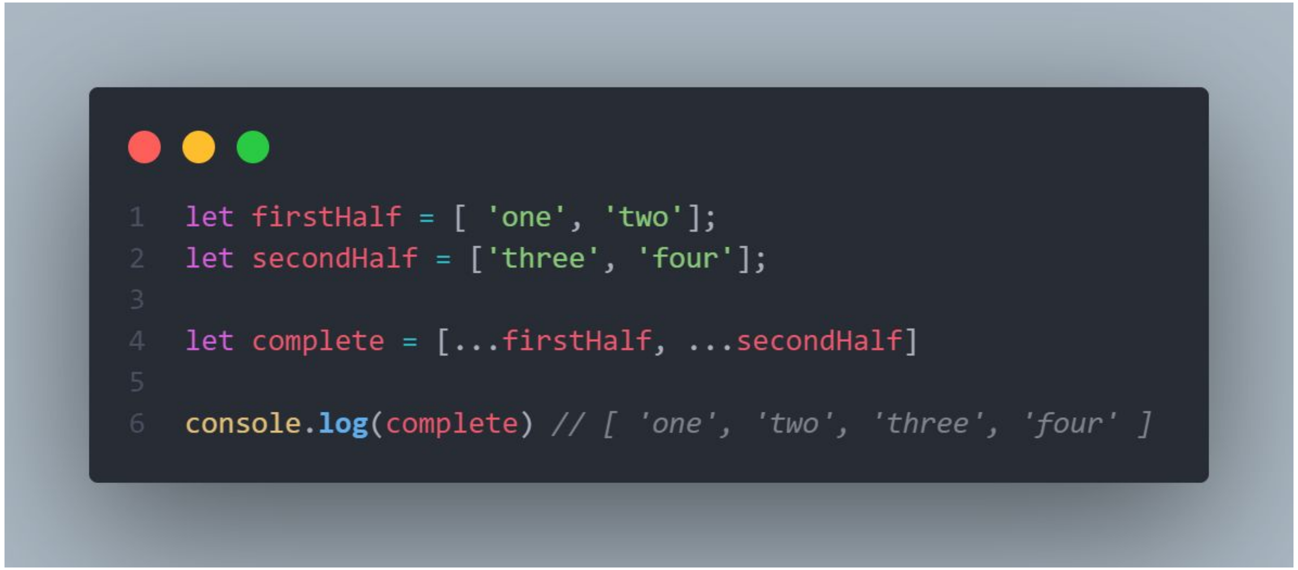
```
"use strict"
```

This way, we enable Strict Mode that changes the semantics of the JavaScript language and force you to code in a modern style

SOMES FEATURES 1

There are a lot of cool features of Modern JS. But we can't cover them all, so here are the ones we like the most:

- Spread operator for arrays:

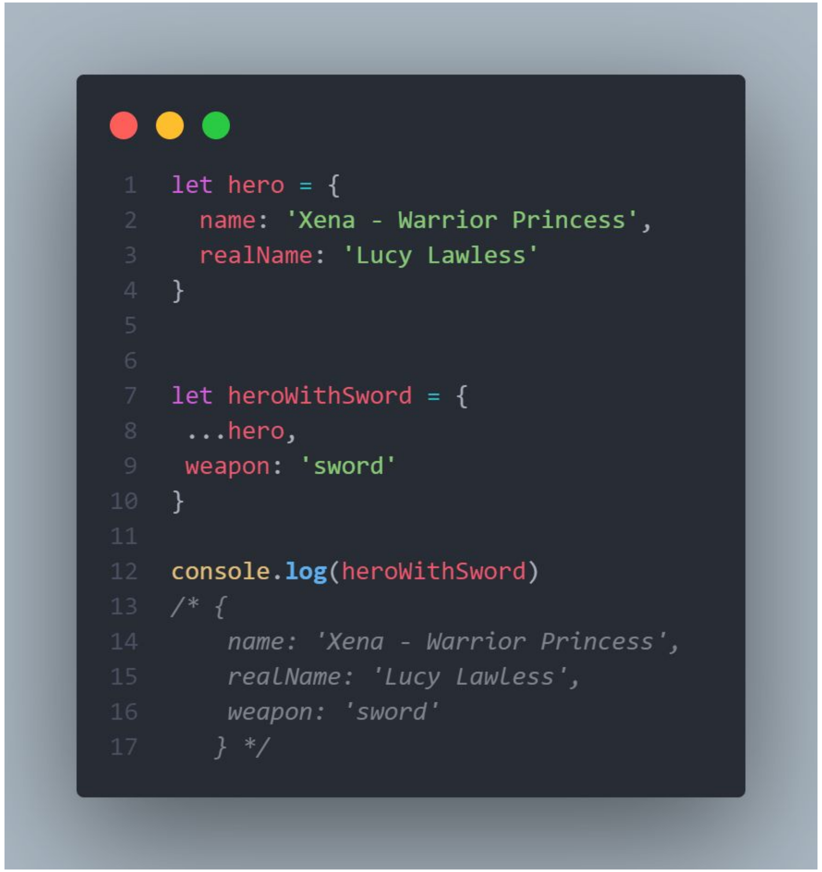


```
let firstHalf = [ 'one', 'two'];
let secondHalf = ['three', 'four'];

let complete = [...firstHalf, ...secondHalf]

console.log(complete) // [ 'one', 'two', 'three', 'four' ]
```

SOMES FEATURES 2

- Spread operator for objects:

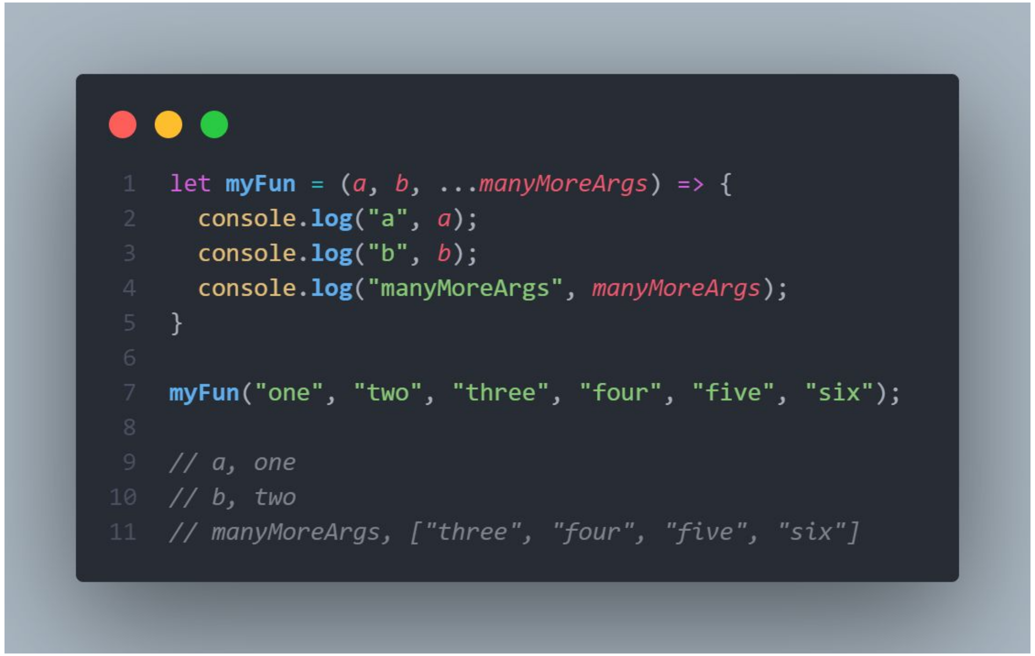
There's an Object.assign but this way it is more visual



```
let hero = {
  name: 'Xena - Warrior Princess',
  realName: 'Lucy Lawless'
}


let heroWithSword = {
 ...hero,
 weapon: 'sword'
}

console.log(heroWithSword)
/* {
    name: 'Xena - Warrior Princess',
    realName: 'Lucy Lawless',
    weapon: 'sword'
   } */
```

SOMES FEATURES 3

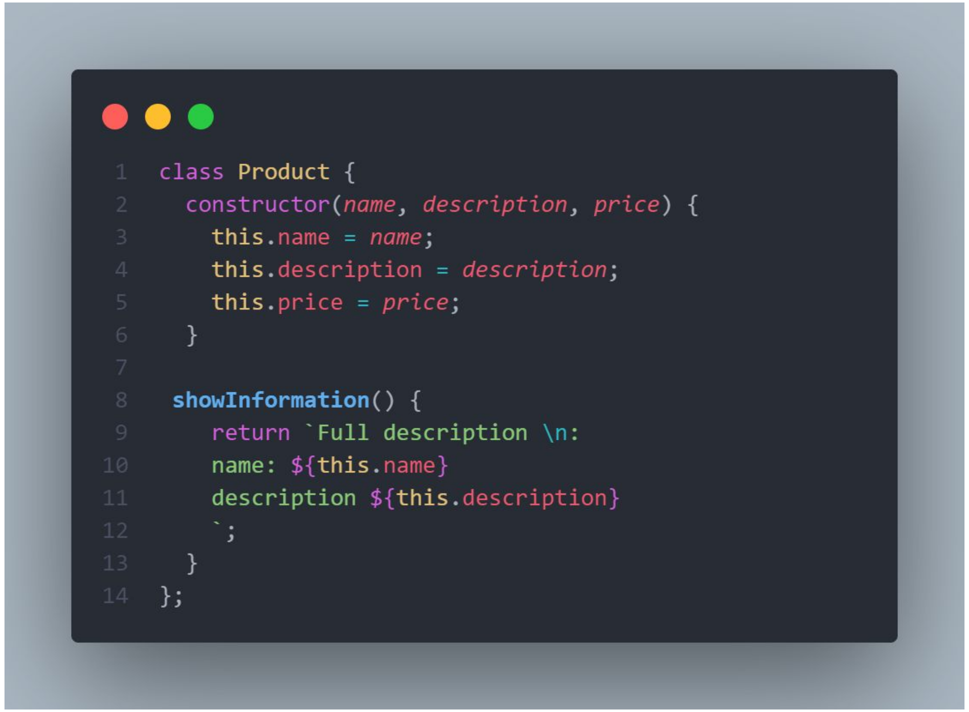
- Rest parameter (just one):



```
let myFun = (a, b, ...manyMoreArgs) => {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");

// a, one
// b, two
// manyMoreArgs, ["three", "four", "five", "six"]
```

SOMES FEATURES 4

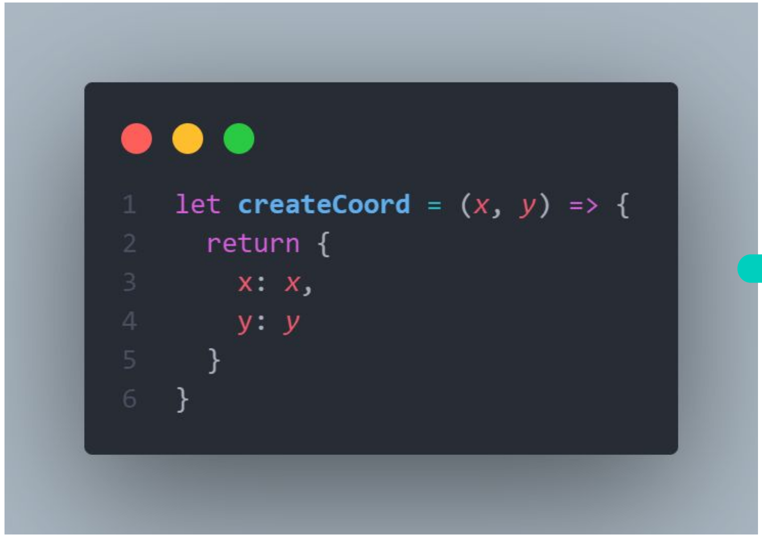
- String interpolation:



```
class Product {
  constructor(name, description, price) {
    this.name = name;
    this.description = description;
    this.price = price;
  }

 showInformation() {
    return `Full description \n:
    name: ${this.name}
    description ${this.description}
    `;
  }
};
```

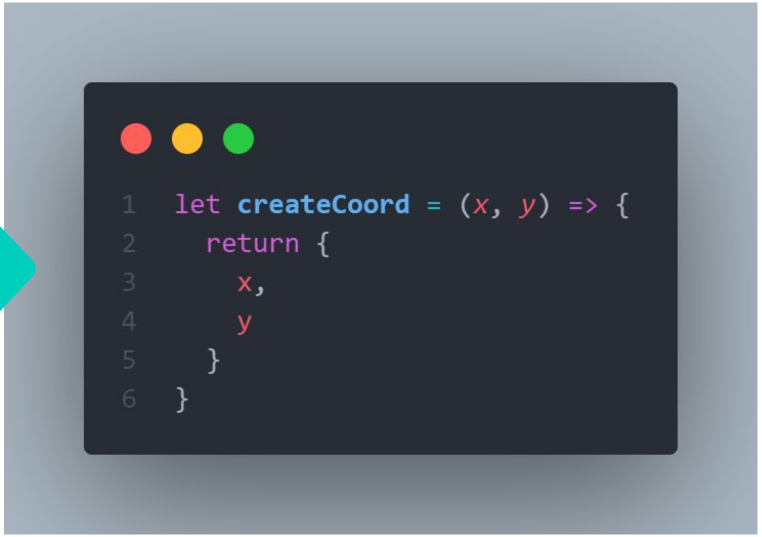
SOMES FEATURES 5

- Shorthand properties:



```
let createCoord = (x, y) => {
  return {
    x: x,
    y: y
  }
}
```

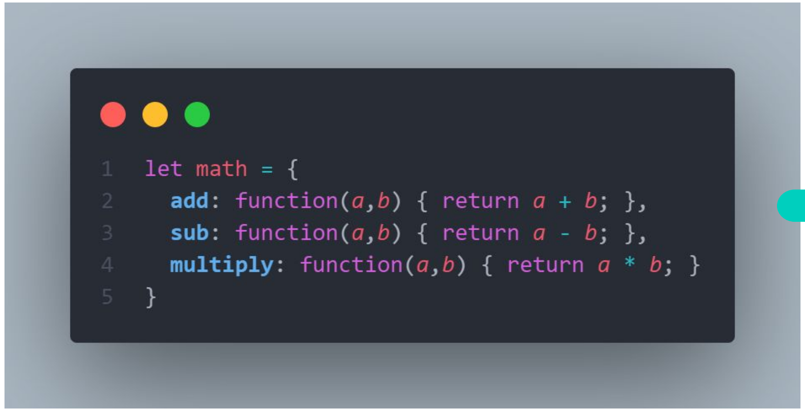
ES6



```
let createCoord = (x, y) => {
  return {
    x,
    y
  }
}
```

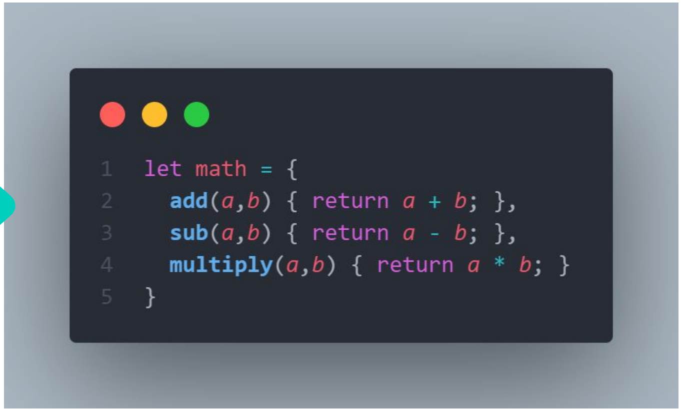
SOMES FEATURES 6

- Method properties:



```
let math = {
  add: function(a,b) { return a + b; },
  sub: function(a,b) { return a - b; },
  multiply: function(a,b) { return a * b; }
}
```

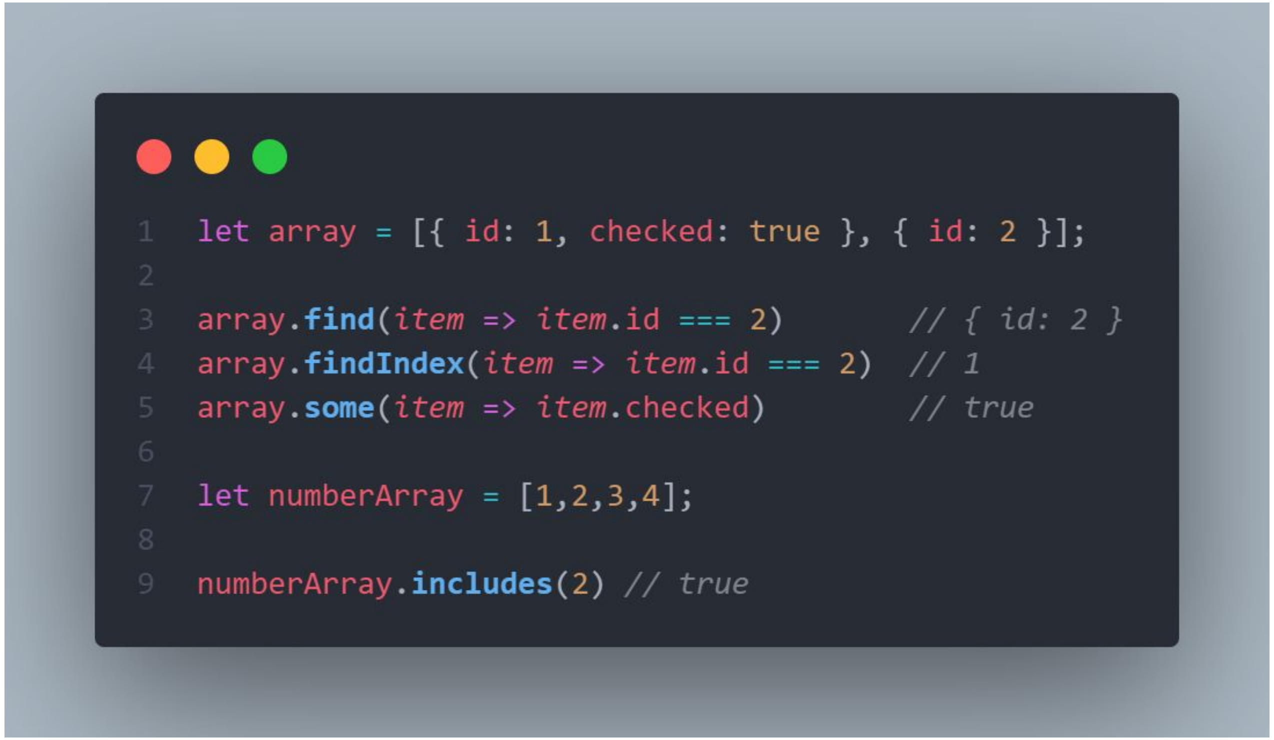
ES6



```
let math = {
  add(a,b) { return a + b; },
  sub(a,b) { return a - b; },
  multiply(a,b) { return a * b; }
}
```

SOMES FEATURES 7

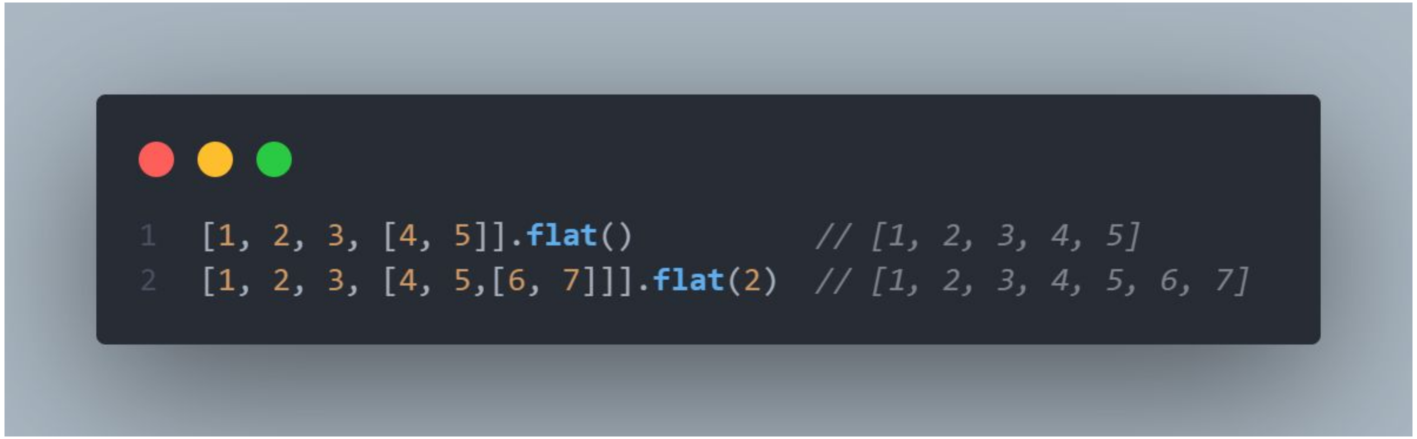
- Array methods("find()", "findIndex()", "some()", "includes()"):



```
let array = [{ id: 1, checked: true }, { id: 2 }];

array.find(item => item.id === 2)         // { id: 2 }
array.findIndex(item => item.id === 2)  // 1
array.some(item => item.checked)          // true

let numberArray = [1,2,3,4];

numberArray.includes(2) // true
```

SOMES FEATURES 8

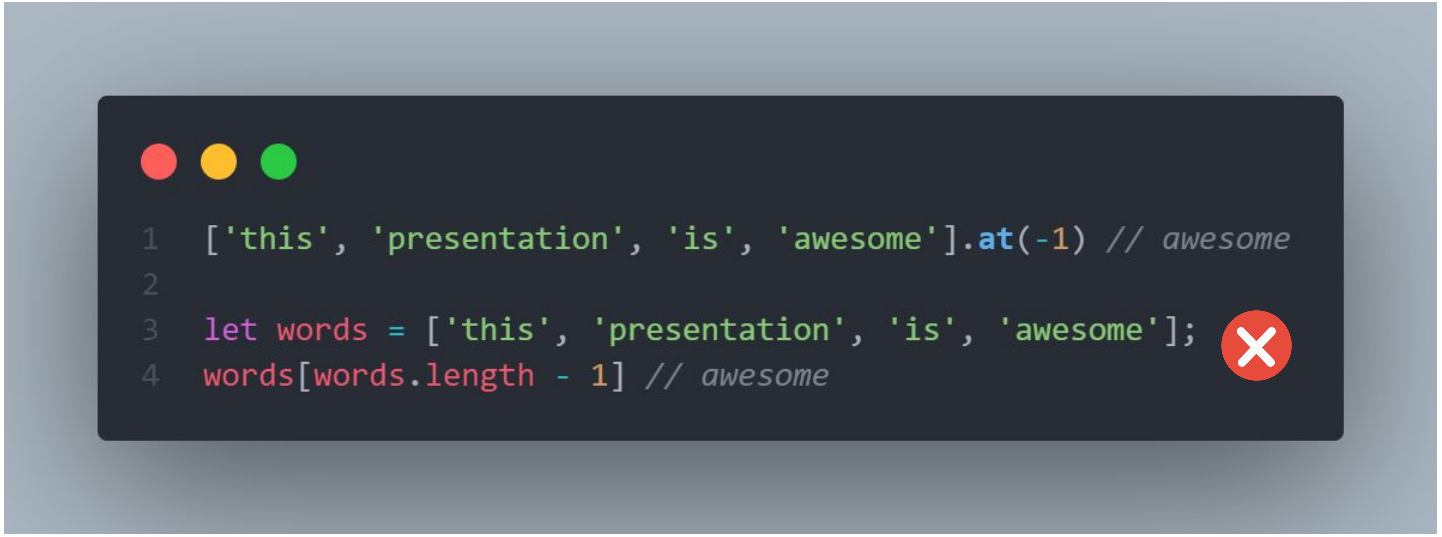
- Array method(“flat()”):



```
[1, 2, 3, [4, 5]].flat()          // [1, 2, 3, 4, 5]
[1, 2, 3, [4, 5,[6, 7]]].flat(2)  // [1, 2, 3, 4, 5, 6, 7]
```

SOMES FEATURES 9

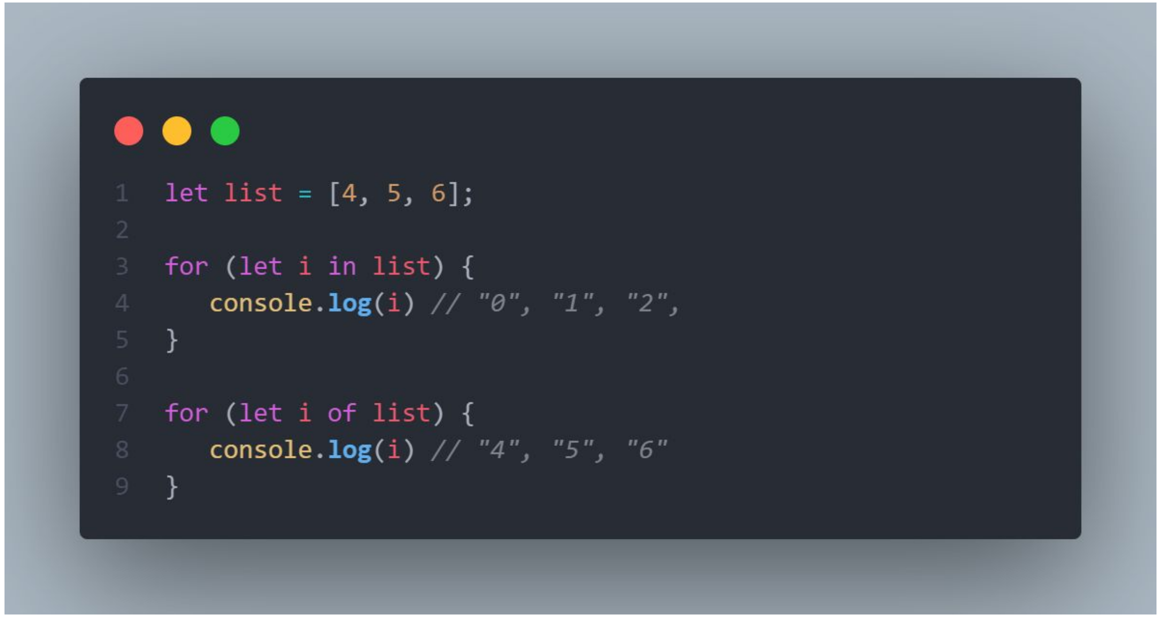
- Array and string method (“at()”):



```
['this', 'presentation', 'is', 'awesome'].at(-1) // awesome

let words = ['this', 'presentation', 'is', 'awesome'];
words[words.length - 1] // awesome
```

SOMES FEATURES 10

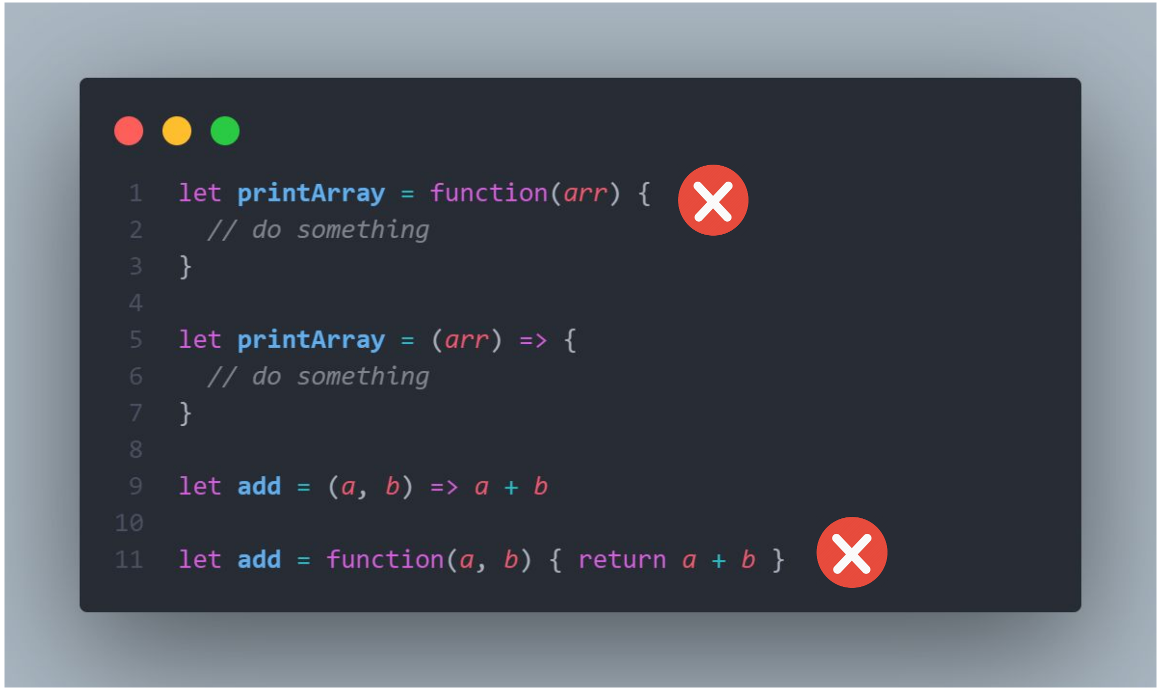
- For of:



```
let list = [4, 5, 6];

for (let i in list) {
   console.log(i) // "0", "1", "2",
}

for (let i of list) {
   console.log(i) // "4", "5", "6"
}
```

SOMES FEATURES 11

- Arrow functions and let usage:



```
let printArray = function(arr) {
  // do something
}

let printArray = (arr) => {
  // do something
}

let add = (a, b) => a + b

let add = function(a, b) { return a + b }
```

THANKS!

Do you have any questions?

Enrique Álvarez Mesa (alu0101142104)
Sergio de la Barrera García (alu0100953275)

REFERENCES

- Eloquent JavaScript
- Modern JavaScript tutorial
- Mozilla JavaScript guide
- W3 JavaScript Tutorial
- Manz: Lenguaje JS
- Midudev