# Introduction to JavaScript

29/01/2024

# Our Team

Thomas Edward Bradley (alu0101408248)

Daniel David Sarmiento Barrera (alu0101499208)

# Index

# What is JS?

# History

Brendan Eich
(1995)

- Not related to Java, named similarly for marketing
- Standardized under the name ECMAScript
- We'll be using ES6+
- Famous for always pushing ahead, despite code quality

# Usage

| Jan 2024 | Jan 2023 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Python | 13.97% | -2.39% |
| 2 | 2 | | C | 11.44% | -4.81% |
| 3 | 3 | | C++ | 9.96% | -2.95% |
| 4 | 4 | | Java | 7.87% | -4.34% |
| 5 | 5 | | C# | 7.16% | +1.43% |
| 6 | 7 | ^ | JavaScript | 2.77% | -0.11% |

# Running Code

There are two ways to run JS:

- In browser
- In terminal (using Node.js)

```
node example.js
```

Remember:

- JS is an interpreted language ( 
is compiled)

# Values

# Values

Some values:


We store these in bindings with the **let** keyword:

```
let exampleString = 'Hello World!';
let exampleNumber = 1;
let exampleBool = true;
let emptyValue = null;
```

# Values

**Bindings** (also known as **variables**) can also be declared using the **var** or **const** keywords:

```
var exampleVar = 'Hello World!';
const CONST_VAR = 'Hello World!';
```

Note:

- **const** creates an unchanging, **constant** variable
- **var** modifies the **scope** of the variable

# Values

- **var** was mainly used in pre-2015 JS, not anymore
- If **in** function, gives binding **function scope**
- If **not in** function, gives binding **global scope**

```
var example = 'I\'m global variable!';
function test() {
  var example = 'I\'m function variable!';
  {
    var example = 'I\'m block variable!';
  }
  console.log('Inside Function - ' + example);
  // -> Inside Function - I'm block variable!
}
test();
console.log('Outside Function - ' + example);
// -> Outside Function - I'm global variable!
```

# Values

JS is:

- **Dynamic** - A variable can change its type during runtime
- **Weakly typed** - When declaring a variable, we don’t need to specify its type

Remember:

Is static and strongly typed

# Binding Names

According to google's style guide:

- Binding names should use **lower*CamelCase***
  - Same as functions
- Constant names should use ***CONSTANT_CASE***

```
let example = 'Hello World!';
var exampleVar = 'Hello World!';
const CONST_EXAMPLE = 'Hello World!';
```

# Binding Names

Names also **must not**:

- Start with a digit
- Contain special characters (other than ‘$’ and ‘_’)
- Be a keyword
- Be on the list of reserved future keywords

# Expressions & Statements

- Expression: A fragment of code that produces a value

- Statement: A full instruction for the computer
  - Has to end in a semicolon (;)

# Data Types

There are 7 primitive types (contains a single thing):

- Number
- BigInt
- String
- Symbol
- Boolean
- Null
- Undefined

# *typeof* unary operator

| typeof | Result |
| --- | --- |
| Undefined | “undefined” |
| Null | “object” |
| Number | “number” |
| String | “string” |
| BigInt | “bigint” |
| Symbol | “symbol” |
| Boolean | “boolean” |
| Function | “function” |

# typeof null is “object”?

- That's an officially recognized error in typeof
- Kept for compatibility reasons
- **Null is not an object**, it is a special value with a separate type of its own

```
1   console.log(typeof undefined);  // -> "undefined"
2   console.log(typeof null);   // -> "object"
3   console.log(typeof false);  // -> "boolean"
4   console.log(typeof 5);  // -> "number"
5   console.log(typeof 'foo');  // -> "string"
6   console.log(typeof console.log);    // -> "function"
```

# Number

JavaScript does not differentiate between integers, floats, doubles, ...

```
let integerValue = 1234;
console.log(typeof(integerValue));
// -> number
let floatValue = 12.34;
console.log(typeof(floatValue));
// -> number
```

# Number

- Numbers are stored in 64 bits
- 2^64 possible numbers
- This means we can represent: ±(2^(53)-1)
  - 1 bit for sign
  - 11 bits for exponent

# Number

We can use scientific notation

```
let scientificNotation = 2e5; // 2 * 10^5
console.log(scientificNotation);
// -> 200000
```

# Number

Special values:

- NaN
  - NaN is toxic

```
console.log('Hello World!' / 2); // -> NaN
console.log(Math.sqrt(-1)); // -> NaN
console.log(Number('Impossible')); // -> NaN
```

```
console.log(NaN * 10);  // -> NaN
console.log(NaN / 10 + (5 * 10));  // -> NaN
console.log(NaN + 'Hello'); // -> NaNHello
```

# Number

Special values:

- ±Infinity

```
console.log(10 / 0); // -> Infinity
console.log(-10 / 0); // -> -Infinity
console.log(Number.MAX_VALUE * 2); // -> Infinity
console.log(Infinity + 1); // -> Infinity
console.log(Infinity - 1); // -> Infinity
console.log(Infinity + Infinity); // -> Infinity
```

# BigInt

- Append **n** at the end of the number
- Extended range
- Solve precision errors
- Rarely needed

```
let smallNumber = 123;
let bigNumber = 123456789012345678901234567890n;
let result = bigNumber * BigInt(smallNumber);
console.log(result); // -> 24691357802469135780246913578024691357780n
```

# String

- Double (“ “), single quotes (‘ ‘) or backticks (` `)
- No char type
- Special character ‘\’
  - New Line: \n
  - Tab: \t
- String interpolation with backticks
- Use single quotes (google style guide)
- Additional String Properties

# String

- Template literals:
  - Enclosed by backticks
- Backticks allow **multiline** strings

```
let name = 'Juan';
let age = 25;
let message = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(message); // -> Hello, my name is Juan and I am 25 years old.
```

```
let name = 'Juan';
let age = 25;
let message = 'Hello, my name is ${name} and I am ${age} years old.';
console.log(message);   // -> Hello, my name is ${name} and I am ${age} years old.
```

# Symbol

- Unique identifiers for objects
- Useful for creating shared constants
- Not to be confused with char
- Needs to be specifically declared
  - Takes a string as a parameter

```
let symbolExample = Symbol('mySymbol', {global: true}, {constant: false});
console.log(symbolExample);
// -> Symbol(mySymbol)
```

# Null / Undefined

- **Null**: special value which represents “empty”
- **Undefined**: represents “value is not assigned”

```
let notDefined;
let exampleOfNull = null;
console.log(notDefined);    // -> "undefined"
console.log(exampleOfNull);   // -> "null"
```

Remember:

- Null **is not** a reference to a non-existent object
- Null **is not** a null pointer

# Null / Undefined

Most operations that don’t produce a meaningful value return undefined

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

Useful to test if a value is different from null or undefined (has a real value)

# Wrapped Objects

- Primitives are immutable
- Object wrapper types (except null and undefined)
- Object wrappers are temporary

```
let welcomeMessage = 'Hello, World!';
let messageLength = welcomeMessage.length;  // Access length
console.log(messageLength);                 // Print length
// -> 13
```

# Wrapped Objects

1. Create a string
2. Create a temporary wrapped object
3. Access the length property of our new variable
4. Print the length of the string

```
let welcomeMessage = 'Hello, World!';
let wrappedObject = new String(welcomeMessage);
let messageLength = wrappedObject.length;
console.log(messageLength);
// -> 13
```

# Type Conversion

- Called **coercion** when performed automatically
- String conversion: String(value)
- Numeric conversion: Number(value)
  - Undefined → NaN
  - Null → 0
  - True/false → 1 / 0
- Boolean conversion: Boolean(value)
  - 0, null, undefined, NaN, “” → false

# Type Conversion

```
let userAge = '25';
let userAgeNumber = Number(userAge);
console.log(userAgeNumber); // -> 25
let userAges = [25, 26, 27];
let userAgesString = String(userAges);
console.log(userAgesString); // -> 25,26,27
```

Basic Conversions

# Type Conversion

```
console.log(Boolean(0));
console.log(Boolean(null));
console.log(Boolean(undefined));
console.log(Boolean(NaN));
console.log(Boolean(""));
console.log(Boolean(''));
console.log(Boolean(``));
// -> false
```

False Values

```
console.log(Number(undefined));
// -> NaN
console.log(Number(null));
// -> 0
console.log(Number(true));
// -> 1
console.log(Number('1990w'));
// -> NaN
```

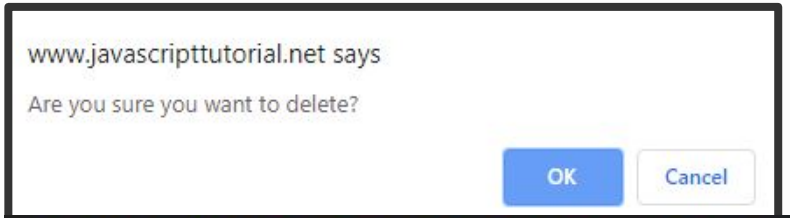
Number Conversions

# JS Basics

# prompt, alert, confirm


```
prompt('What is your name?');
```


```
alert('Default Alert');
```




```
confirm('Are you sure you want to delete?');
```

# Familiar Statements

- Conditional Statement: if
- Loops: while, do, for
- Internal Loop Statements: break, continue
- Construct: switch

Same as 

Remember: leave a space between the keyword and parentheses

# for (... in ...)

However we can use an alternate form of **for( ; ; )** :

```
const numbers = [1, 2, 3, 4, 5];
let text = '';
for (let position in numbers) {
  text += numbers[position];
}
console.log(text);
// -> 12345
```

# for (... of ...)

Or we can simplify further:

```
const numbers = [1, 2, 3, 4, 5];
let text = '';
for (let digit of numbers) {
  text += digit;
}
console.log(text);
// -> 12345
```

# Comments

2 types of comments

- Single line
- Multiline

```
// Coment
```

```
/*
  Multiline Comment
*/
```

Same as 

# Indentation

Our code should be indented using 2 spaces

```
function indetationExample() {
  // code here
  if (true) {
    while (false) {
      // code here
    }
    // code here
  }
}
// code here
```

```
function indetationExample() {
    // code here
    if (true) {
        while (false) {
            // code here
        }
    // code here
    }
}
// code here
```

# Semicolon

- Our code will work fine without semicolons (;)
- We should use them regardless

```
const numbers = [1, 2, 3, 4, 5];
let text = "";
for (let position in numbers) {
  text += numbers[position];
}
console.log(text);
// -> 12345
```

=

```
const numbers = [1, 2, 3, 4, 5]
let text = ""
for (let position in numbers) {
  text += numbers[position]
}
console.log(text)
// -> 12345
```

# Operators

# Familiar Operators

- Unary: !, +, -, ++, --
- Binary: <, >, <=, >=, ==, !=, +, -, *, /, %
- Ternary: ( ? : )

```
let randomBool = true;
console.log(randomBool ? "It's True!" : "It's... False");
// -> It's True!
```

Same as 

# String Concatenation

```
let firstWord = 'I';
let secondWord = 'like';
let sentence = '';
sentence = sentence + ' ' + firstWord + ' ' + secondWord;
sentence += ' turtles';
console.log(sentence);
// -> I like turtles
```

Same as 

# OR (||)

- Evaluates from left to right
- Converts operands to boolean
- Has the lowest precedence
- If left operand is true → return this value
- If left operand is false → return right operand
- For **>2 operands**: return the **first true** value
  - If none is found, return last value

# OR (||)

```
console.log(5 - 5 || 2);
// -> 2
console.log(undefined || null || 0);
// -> 0
console.log('hello' || 'goodbye');
// -> 'hello'

let userName = '';
let lastName = '';
let nickName = 'SuperMan';
console.log(lastName || userName || nickName || 'Anonymous');
// -> SuperMan
```

# AND (&&)

- Evaluates from left to right
- Converts operands to boolean
- Second lowest precedence
- If left operand is false → return this value
- If left operand is true → return right operand
- For **>2 operands**: returns the **first false** value
  - If none is found, returns last value

# AND (&&)

```
console.log(5 - 5 && 2);
// -> 0
console.log(undefined && null && 0);
// -> undefined
console.log('hello' && 'goodbye');
// -> 'goodbye'

let userName = '';
let lastName = '';
let nickName = 'SuperMan';
console.log(nickName && 'Anonymous' && userName && lastName);
// -> ''
```

# String Comparison

- JavaScript compares by alphabetical order
  - Being more specific, Unicode order
- Compared letter-by-letter

```
console.log('Z' > 'A');                  // -> true
console.log('Lee' > 'Lea');              // -> true
console.log('Bee' > 'Be');               // -> true
console.log('Leonardo' < 'Leonardo ');   // -> true
```

# Different types

- Converts the values to numbers
- Consequences

```
let cookieCount1 = 0;
let cookieCount2 = '0';
console.log(cookieCount1 == cookieCount2);  // -> true
console.log(Boolean(cookieCount1) == Boolean(cookieCount2));   // -> false
```

# Regular Equality Check

- Equality check (==) cannot differentiate 0 from false
- It happens the same with a string
- JavaScripts == and != are broken

```
console.log(0 == false);    // -> true
console.log('' == false);   // -> true
```

Remember:

- Different types are converted to numbers

# Strict equality

- Checks the equality without type conversion
- If the operands are of different types → immediately returns false

```
console.log(0 === false);    // -> false
console.log('' === false);   // -> false
console.log(5 === '5');      // -> false
```

Note:

- There is also a “strict non-equality” operator !==

# Comparison: null and undefined

- Results are different for a strict and a non-strict equality check
- Special rule

```
console.log(null === undefined);    // -> false
console.log(null == undefined);     // -> true
```

Note:

- undefined == null → without conversion they are equal to each other

# Comparison: null and undefined

- Equality check (==) and comparisons (> < >= <=) work different
- Comparisons converts null to a number(0)

```
console.log(null > 0);  // -> false
console.log(null == 0); // -> false
console.log(null >= 0); // -> true
```

Note:

- undefined == null → without conversion they are equal to each other and don't equal anything else

# Functions

# Functions as Values

- Can store function in a variable
- Can pass as an argument
- Should be const (google style guide)
- Should end in ‘;’ as any other variable

```
const functionValue = function(argument) {
  // do something
};

exampleFunction(functionValue);
```

# Declaration Notation

Alternatively we can do:

```
function declarationNotation(argument) {
  // do something
}
```

In this case we can call the function before reaching that part of the code

# Wrong number of Arguments

- Too **many** arguments → Extras are ignored
- Too **few** arguments → Assigned undefined

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}
console.log(minus(10));     // -> -10
console.log(minus(10, 5));  // -> 5
```

# Optional Arguments

- Can give default value to arguments
- Also used when undefined is passed

```
function power(base, exponent = 2) {
  let result = 1;
  for (let counter = 0; counter < exponent; counter++) {
    result *= base;
  }
  return result
}
console.log(power(4));    // -> 16
console.log(power(2, 6)); // -> 64
```

# Optional Arguments

Unlike in  , it doesn't need to be exclusive to the end arguments

```
function power(base = 2, exponent) {
  let result = 1;
  for (let counter = 0; counter < exponent; counter++) {
    result *= base;
  }
  return result
}
console.log(power(undefined, 6)); // -> 64
console.log(power(2, 6));         // -> 64
```

# Return

The following 3 functions are equivalent:

```
function firstReturn() {
  // code
  return undefined;
}
```

```
function secondReturn() {
  // code
  return;
}
```

```
function thirdReturn() {
  // code
}
```

# Nested Scope

```
const timesTable = function(multiplicand) {
  const printMultiplication = function(multiplier) {
    console.log(`${multiplicand} * ${multiplier} = ${multiplicand * multiplier}`);
    // -> 4 * 0 = 0
    // -> 4 * 1 = 4
    // -> ...
  }
  for (let counter = 0; counter < 10; counter++) {
    printMultiplication(counter);
  }
}
timesTable(4);
```

# Closure

Referencing an instance of a local binding in an enclosing scope:

```
function storeValue(number) {
  return () => number;
}
let stored1 = storeValue(1);
let stored2 = storeValue(2);
console.log(stored1()); // -> 1
console.log(stored2()); // -> 2
```

# Closure

The function body sees the environment in which it was created, not the one in which it is called

```
function multiplier(factor) {
  return number => number * factor;
}
let twice = multiplier(2);
console.log(twice(5));
// -> 10
```

# Arrow Functions

- No function keyword
- Reads as: “This input produces this result”
- Recommended by google style guide

```
const arrowFunction = (argument) => {
  // code
}
```

# Arrow Functions

- We can simplify single line arrow functions:

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

- We can use them with no arguments:

```
const sayHello = () => { console.log("Hi! :)"); }
```

# Arrow Functions

However an arrow function that has no arguments, no return value and produces no side effects is illegal

```
const someFunction = () => anotherFunction();
```

# Objects (OOP)

# Objects

- Primitives:
  - Compared by value
  - Simple, immutable
- Objects:
  - Compared by reference
  - Complex, mutable
  - Can contain properties

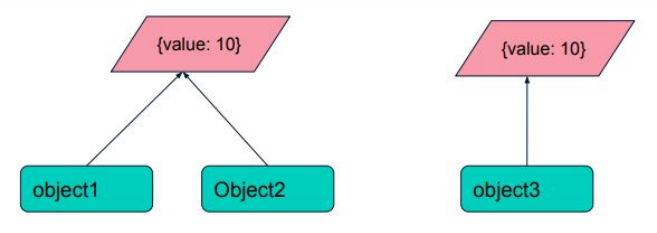
# Mutability

```
let number1 = 12;
let number2 = 12;
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(number1 == number2);  // -> true
console.log(object1 == object2);  // -> true
console.log(object1 == object3);  // -> false

object1.value = 15;
console.log(object2.value);       // → 15
console.log(object3.value);       // → 10
```

- '==' Compares identities, not properties



# Definition

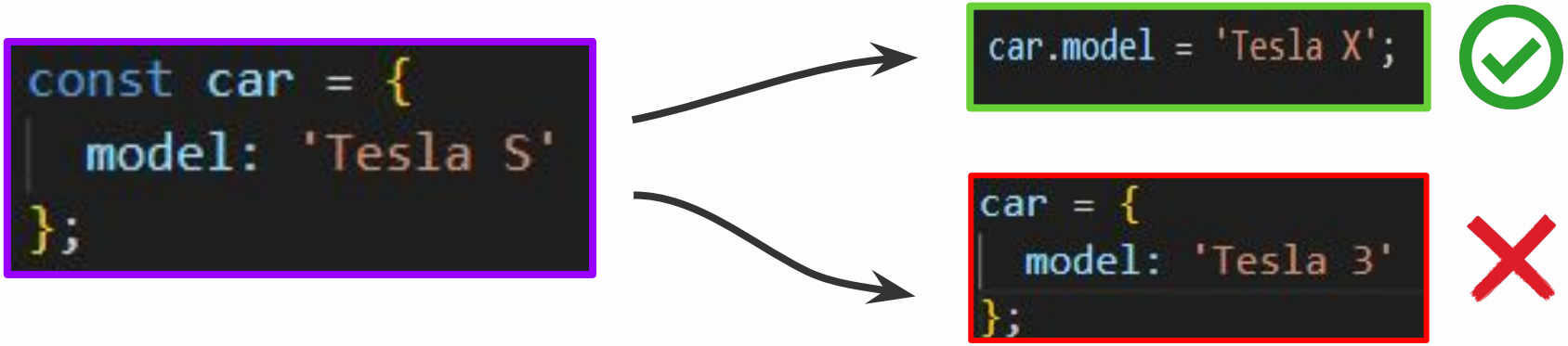
- Definition: Associated arrays with special features
- They store properties (‘key: value’ pairs)
- Property key must be strings or symbols

```
let car = new Object(); // "Object constructor"
```

```
let airplane = {};      // "Object literal"
```

# const Objects

- Objects are stored and copied by reference
- A variable assigned to an object stores its “address in memory”



# Properties

- Listed one after another, separated by commas
- Accessible using dot notation
- We can add and remove properties
- Reading a non-existent property returns undefined

```
let car = {
  model: 'Tesla S',
  releaseYear: 2012,
  price: 95970
};
```

```
delete car.price;      // -> Delete price property
car.color = 'blue';    // -> Add color property
```

# Properties

- Deleting a property is not equal to setting it to undefined
- **Object.keys(...)** is used to see an objects properties

```
let car = {
  model: 'Tesla S',
  releaseYear: 2012
};

delete car.releaseYear;
console.log(Object.keys(car));
// -> [ 'model' ]
```

≠

```
let car = {
  model: 'Tesla S',
  releaseYear: 2012
};

car.releaseYear = undefined;
console.log(Object.keys(car));
// -> [ 'model', 'releaseYear' ]
```

# Properties

- Syntax: **Object.assign(targetObject, sources)**
- Copies the properties of all **sources** objects into **targetObject**
- If property already exists → overwrite

```
let car = {
  model: 'Tesla S',
  releaseYear: 2012
};
Object.assign(car, {releaseYear: 1990, color: 'blue'});
console.log(car);
// -> { model: 'Tesla S', releaseYear: 1990, color: 'blue' }
```

# Modern JS

# What is it?

- Latest way of coding in JS
- Safe
- More concise, expressive and efficient
- Easier for developers

# How to use it?

We must include `'use strict';` at the start of our program to enable **Strict Mode**.

This changes the semantics of JS and forces us to code in a modern style

# Features

Spread:

```
let cake = {
  flavor: 'chocolate',
  color: 'brown'
}
let cakeWithIcing = {
  ...cake,
  icing: 'buttercream'
}
console.log(cakeWithIcing);
/*  {
      flavor: 'chocolate',
      color: 'brown',
      icing: 'buttercream'
    } */
```

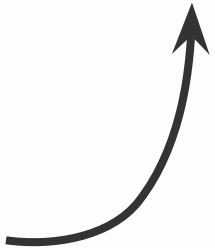
```
let firstHalf = [1, 2];
let secondHalf = [3, 4];
let complete = [...firstHalf, ...secondHalf];
console.log(complete);  // -> [ 1, 2, 3, 4 ]
```

# Features

Shorthand:

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

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

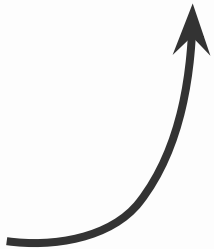


# Features

Method Properties:

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

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



# Features

Array Methods:

```
let array = [{id: 1, checked: true}, {id:2}];
console.log(array.find(item => item.id == 2));
// -> {id: 2}
console.log(array.findIndex(item => item.id == 2));
// -> 1
console.log(array.some(item => item.checked));
// -> true

let numberArray = [1, 2, 3, 4];
console.log(numberArray.includes(2));
// -> true
```

# Features

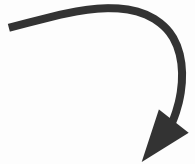
Array Methods:

```
console.log([1, 2, 3, [4, 5]].flat());
// -> [ 1, 2, 3, 4, 5 ]
console.log([1, 2, 3, [4, 5, [6, 7]]].flat());
// -> [ 1, 2, 3, 4, 5, [ 6, 7 ] ]
console.log([1, 2, 3, [4, 5, [6, 7]]].flat(2));
// -> [ 1, 2, 3, 4, 5, 6, 7 ]
```

# Features

Array/String - at():

```
let words = ['I', 'like', 'turtles'];
console.log(words[words.length - 1]);
// -> turtles
```



```
console.log(['I', 'like', 'turtles'].at(-1));
// -> turtles
```

# Bibliography


- Eloquent JavaScript, Third Edition - Marijn Haverbeke
- The Modern JavaScript Tutorial
- Virtual Classroom Unit 1 Notes
- Introduction to JavaScript 22/23 Repository
- Additional String Properties
- JavaScript Reserved Future Keywords
- Google JavaScript Style Guide
- Presentation Slides Template

# Any questions?

# Thank you for your attention

Sarmiento Barrera, Daniel David (alu0101499208@ull.edu.es)
Bradley, Thomas Edward (alu0101408248@ull.edu.es)