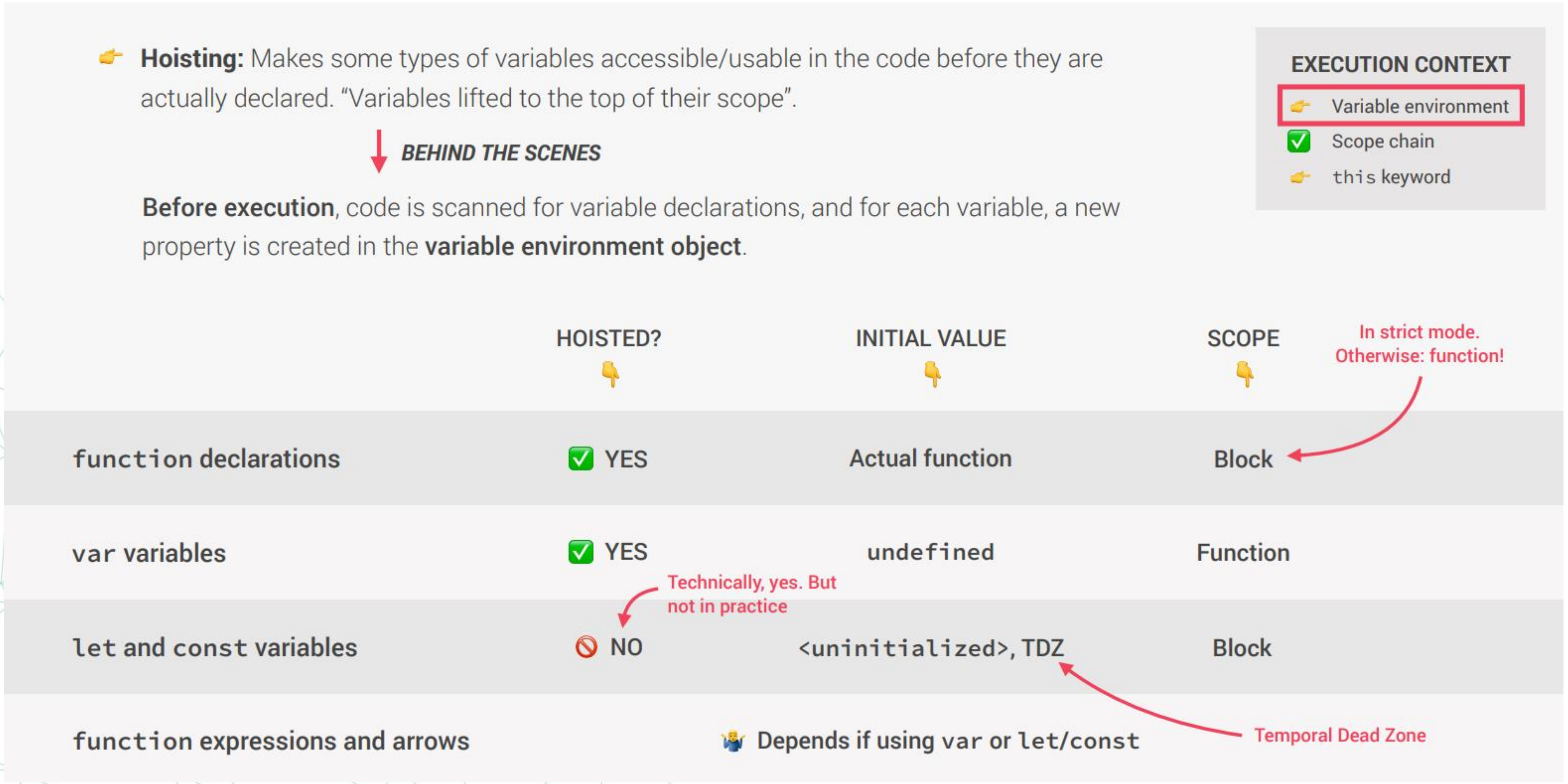
JavaScript Part 3

Table of Contents

❑ What are ES6 Features?

- Hoisting



👉 **Hoisting:** Makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".

↓ ***BEHIND THE SCENES***

Before execution, code is scanned for variable declarations, and for each variable, a new property is created in the **variable environment object**.

EXECUTION CONTEXT

- 👉 Variable environment
- ✅ Scope chain
- 👉 `this` keyword

	HOISTED? 👇	INITIAL VALUE 👇	SCOPE 👇
`function` declarations	✅ YES	Actual function	Block
`var` variables	✅ YES	`undefined`	Function
`let` and `const` variables	🚫 NO	`<uninitialized>`, TDZ	Block
`function` expressions and arrows		🤷 Depends if using `var` or `let/const`	

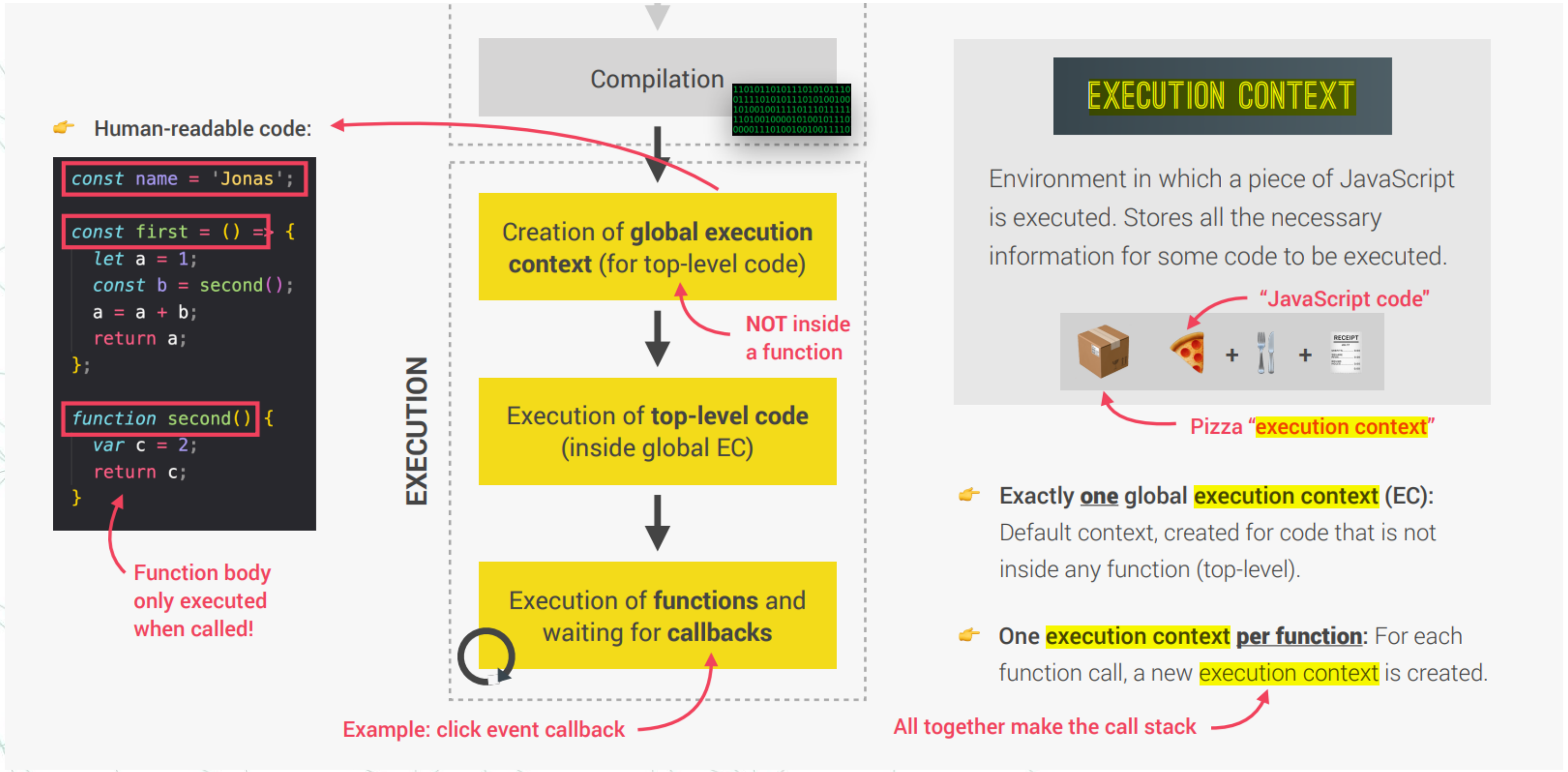
In strict mode. Otherwise: function!

Technically, yes. But not in practice

Temporal Dead Zone

❑ What is Execution Context??

- Global Execution Context.
- Local Execution Context.



Human-readable code:

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second();
  a = a + b;
  return a;
};

function second() {
  var c = 2;
  return c;
}
```

Function body only executed when called!

Compilation

EXECUTION

Creation of **global execution context** (for top-level code)

NOT inside a function

Execution of **top-level code** (inside global EC)

Execution of **functions** and waiting for **callbacks**

Example: click event callback

EXECUTION CONTEXT

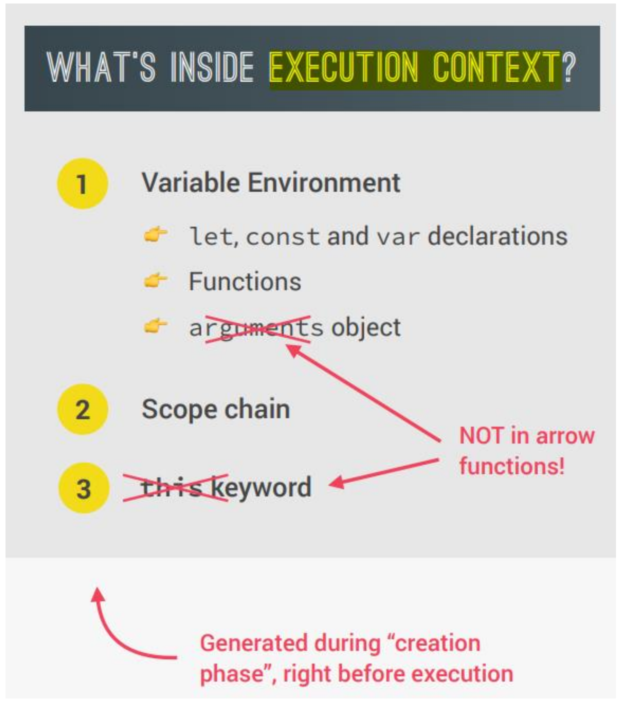
Environment in which a piece of JavaScript is executed. Stores all the necessary information for some code to be executed.

"JavaScript code"

Pizza "execution context"

- Exactly **one** global **execution context** (EC): Default context, created for code that is not inside any function (top-level).
- One **execution context per function**: For each function call, a new execution context is created.

All together make the call stack



WHAT'S INSIDE EXECUTION CONTEXT?

1. Variable Environment
   - `let`, `const` and `var` declarations
   - Functions
   - ~~`arguments`~~ object
2. Scope chain
3. ~~`this`~~ keyword

NOT in arrow functions!

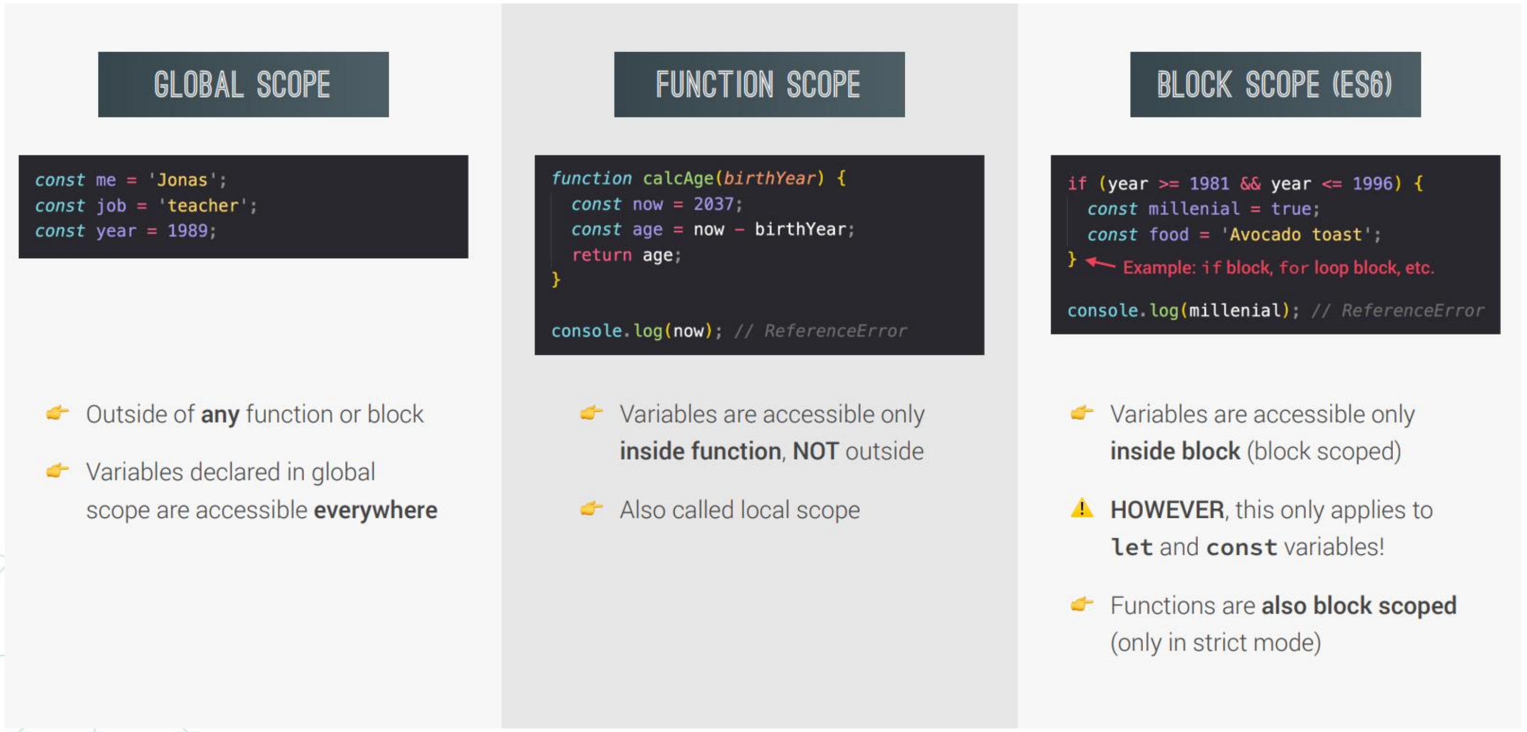
Generated during "creation phase", right before execution

❑ Scopes

- Global Scope.
- Local Scope.
- Lexical Scope.
- Block Scope. NEW!

👉 **Lexical scoping:** Scoping is controlled by **placement** of functions and blocks in the code;

Scoping: How our program's variables are organized and accessed. "Where do variables live?" or "Where can we access a certain variable, and where not?"



GLOBAL SCOPE

```
const me = 'Jonas';
const job = 'teacher';
const year = 1989;
```

👉 Outside of **any** function or block

👉 Variables declared in global scope are accessible **everywhere**

FUNCTION SCOPE

```
function calcAge(birthYear) {
  const now = 2037;
  const age = now - birthYear;
  return age;
}

console.log(now); // ReferenceError
```

👉 Variables are accessible only **inside function**, **NOT** outside

👉 Also called local scope

BLOCK SCOPE (ES6)

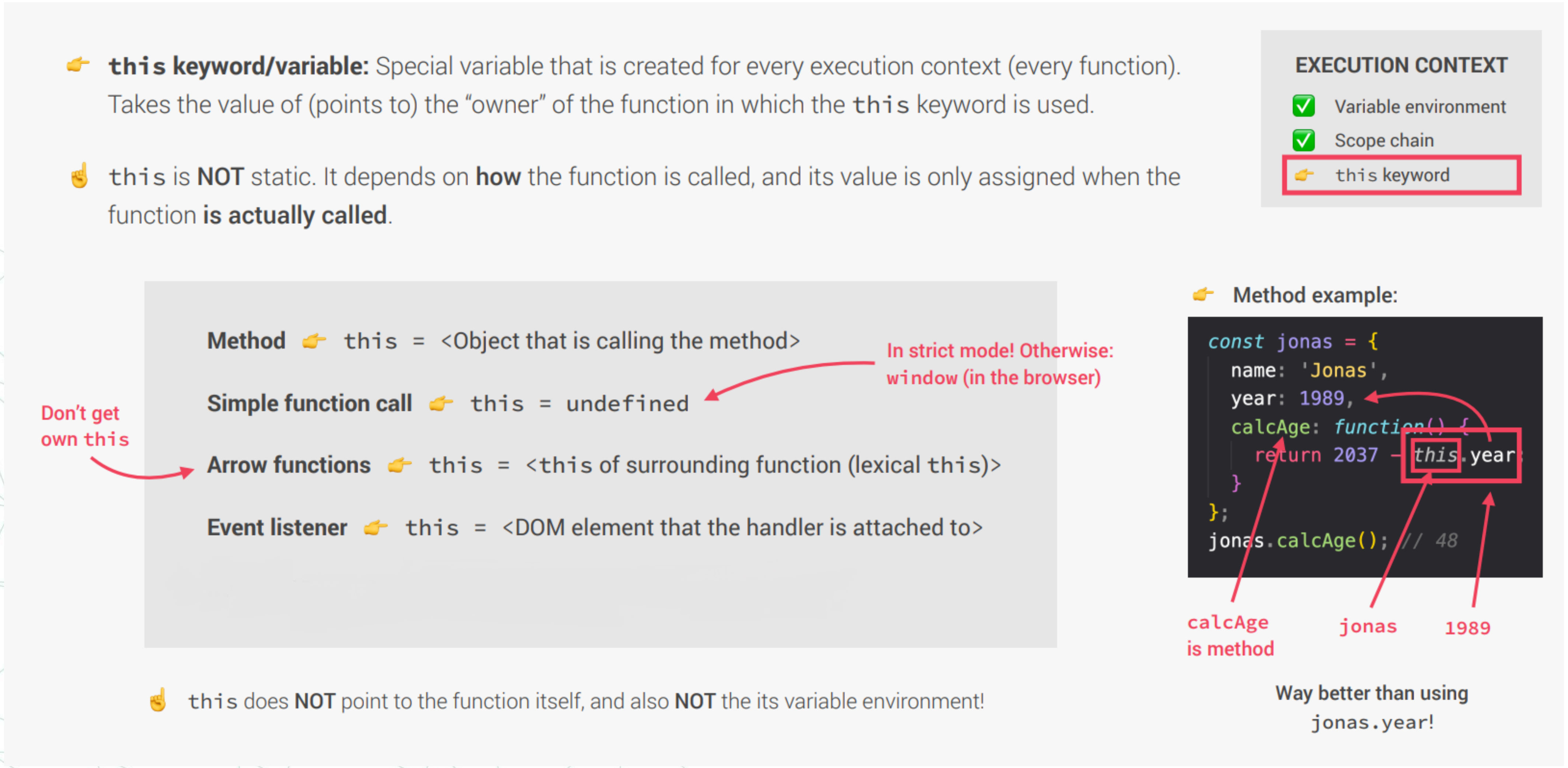
```
if (year >= 1981 && year <= 1996) {
  const millenial = true;
  const food = 'Avocado toast';
} ← Example: if block, for loop block, etc.

console.log(millenial); // ReferenceError
```

👉 Variables are accessible only **inside block** (block scoped)

⚠️ **HOWEVER**, this only applies to `let` and `const` variables!

👉 Functions are **also block scoped** (only in strict mode)

❑ This Keyword



👉 **this keyword/variable:** Special variable that is created for every execution context (every function). Takes the value of (points to) the "owner" of the function in which the this keyword is used.

☝ this is **NOT** static. It depends on **how** the function is called, and its value is only assigned when the function **is actually called**.

EXECUTION CONTEXT

- ✓ Variable environment
- ✓ Scope chain
- 👉 this keyword

Method 👉 this = <Object that is calling the method>

Simple function call 👉 this = undefined

Arrow functions 👉 this = <this of surrounding function (lexical this)>

Event listener 👉 this = <DOM element that the handler is attached to>

In strict mode! Otherwise: window (in the browser)

Don't get own this

👉 Method example:

```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year;
  }
};
jonas.calcAge(); // 48
```

Way better than using jonas.year!

☝ this does **NOT** point to the function itself, and also **NOT** the its variable environment!

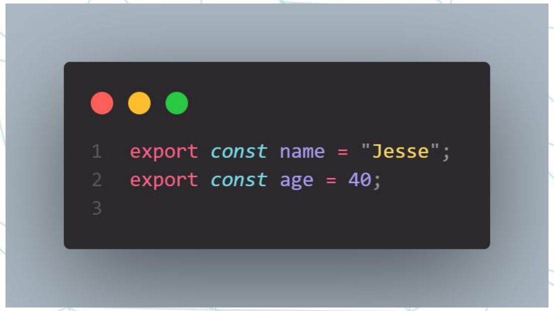
Modules -> Import / Export

JavaScript modules allow you to break up your code into separate files.

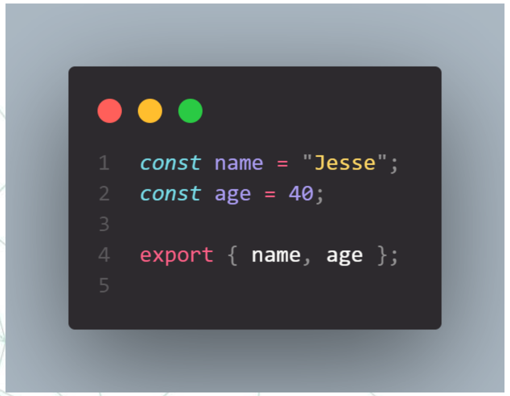
Modularity: The basic principle of Modularity is that “Systems should be built from cohesive, loosely coupled components (modules)” which means s system should be made up of different components that are united and work together in an efficient way and such components have a well-defined function.

There’s Two ways to export Modules

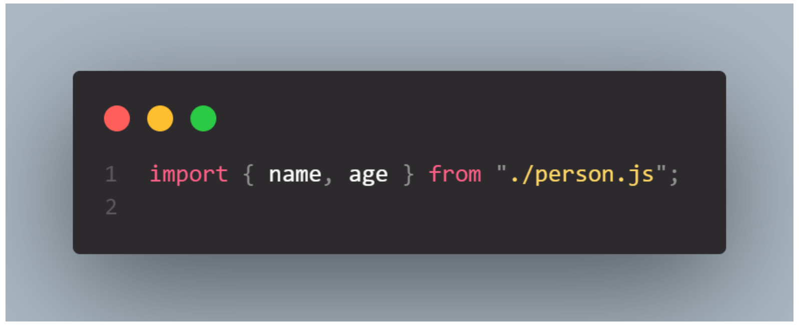
- Named Export & Import



```
export const name = "Jesse";
export const age = 40;
```

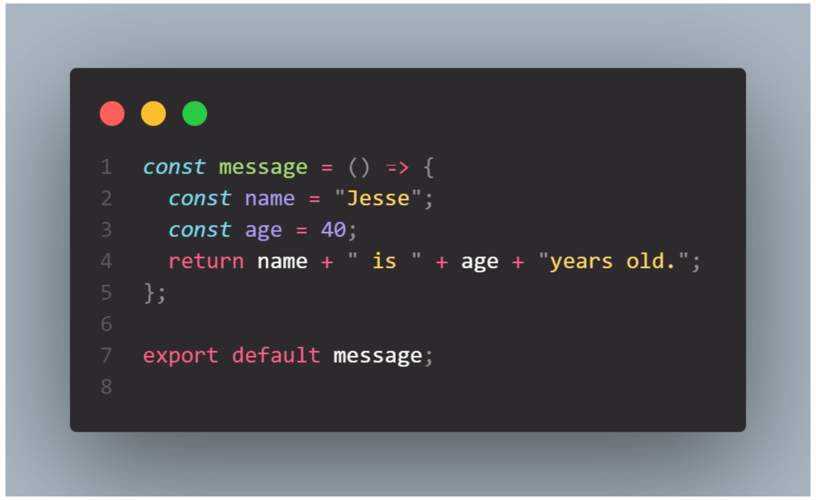


```
const name = "Jesse";
const age = 40;

export { name, age };
```

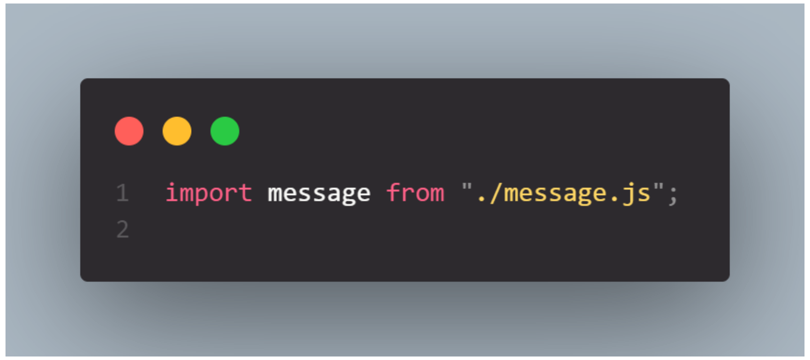


```
import { name, age } from "./person.js";
```

- Default Export & Import

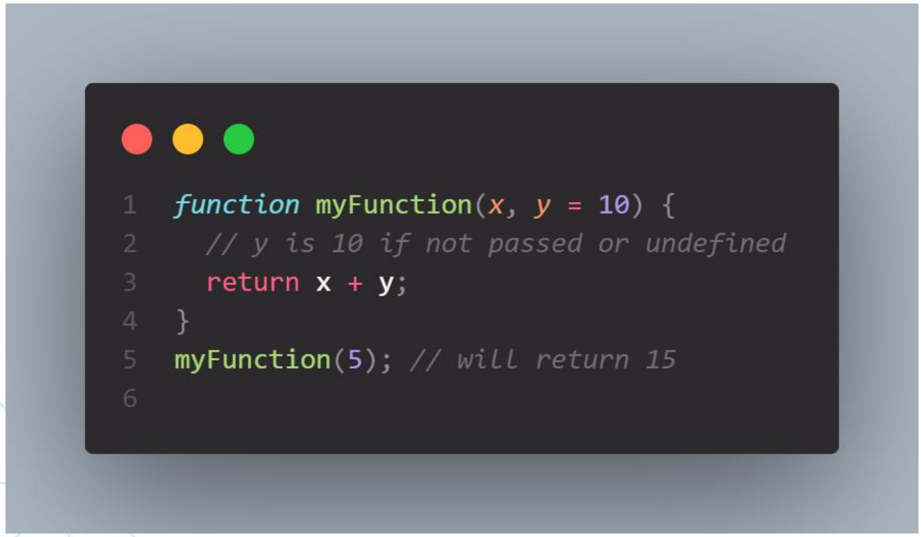


```javascript
const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + " is " + age + "years old.";
};

export default message;
```



```javascript
import message from "./message.js";
```

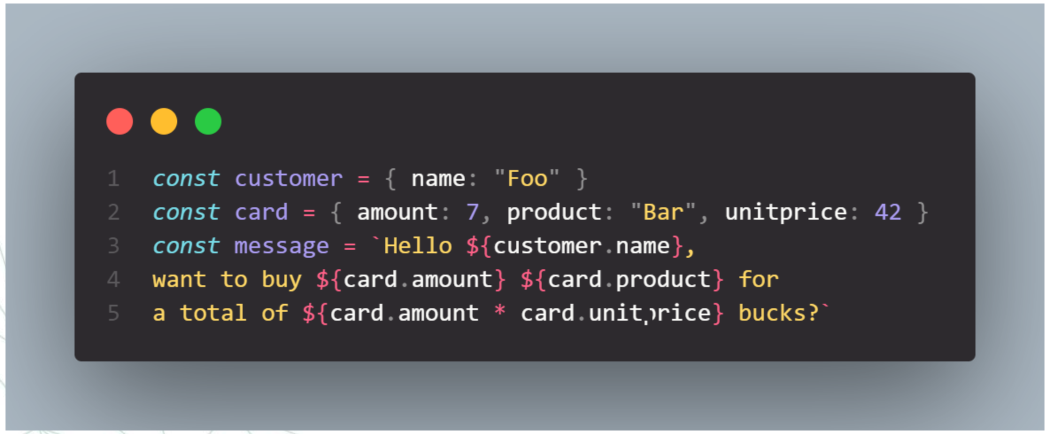
Default Parameter Values



```javascript
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

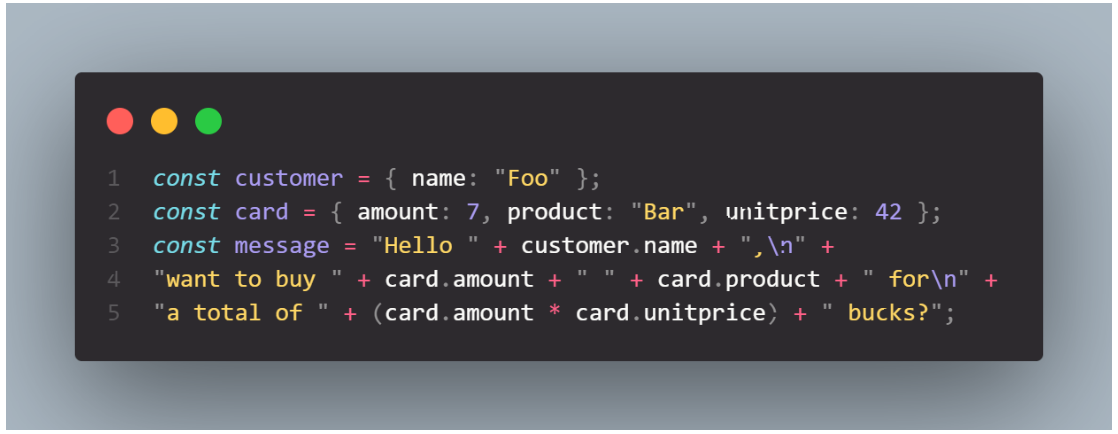
Template Literals

Template Literal



```
const customer = { name: "Foo" }
const card = { amount: 7, product: "Bar", unitprice: 42 }
const message = `Hello ${customer.name},
want to buy ${card.amount} ${card.product} for
a total of ${card.amount * card.unitprice} bucks?`
```

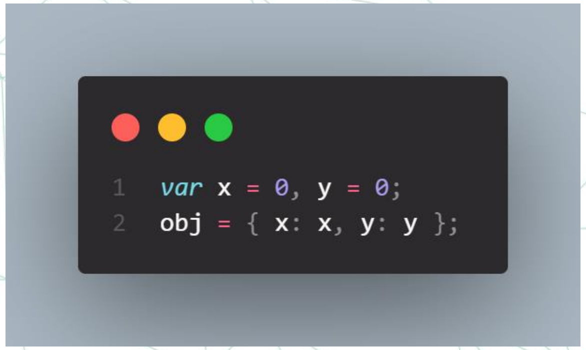
Normal String



```
const customer = { name: "Foo" };
const card = { amount: 7, product: "Bar", unitprice: 42 };
const message = "Hello " + customer.name + ",\n" +
"want to buy " + card.amount + " " + card.product + " for\n" +
"a total of " + (card.amount * card.unitprice) + " bucks?";
```

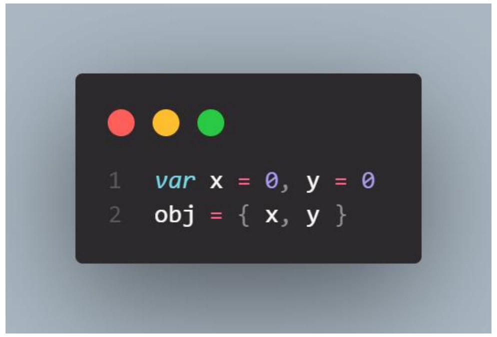
Object Enhancement

Old Way



```
var x = 0, y = 0;
obj = { x: x, y: y };
```

New Way

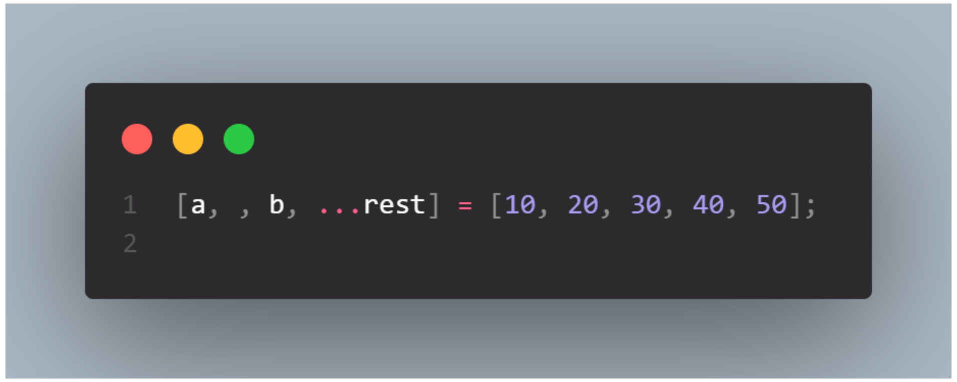


```
var x = 0, y = 0
obj = { x, y }
```

DE-structuring Assignment

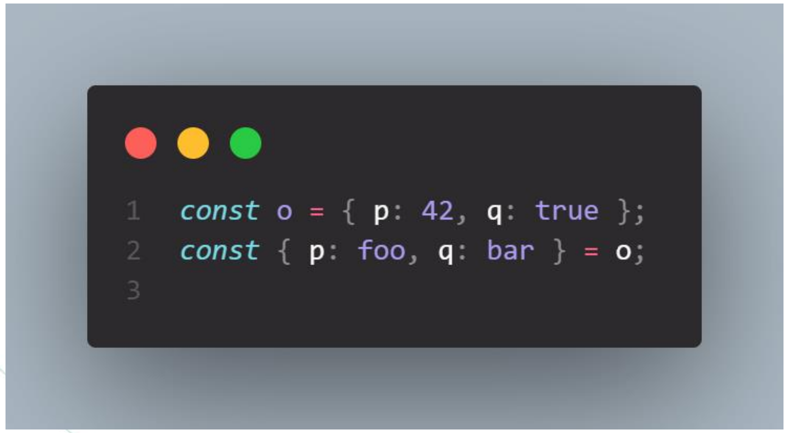
DE-structuring is a feature in programming languages that enables the extraction of values from complex data structures like arrays, objects, or maps and assigning them to individual variables.

DE-structuring Array



```
[a, , b, ...rest] = [10, 20, 30, 40, 50];
```

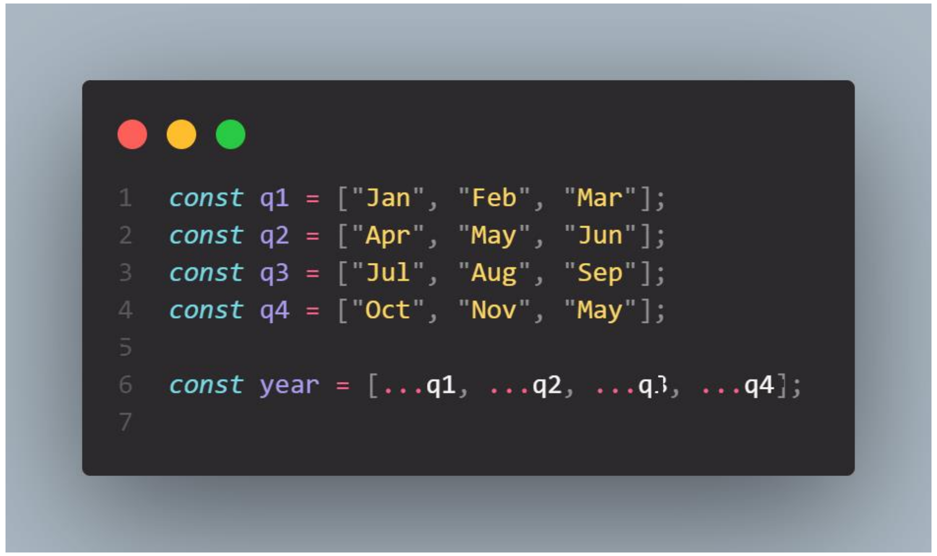
DE-structuring Object



```
const o = { p: 42, q: true };
const { p: foo, q: bar } = o;
```

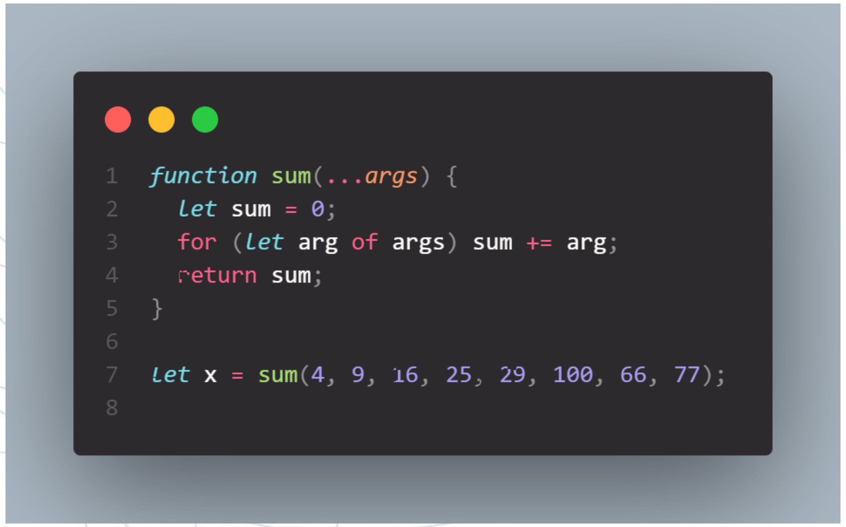
Arrow Functions

Spread Operator ...



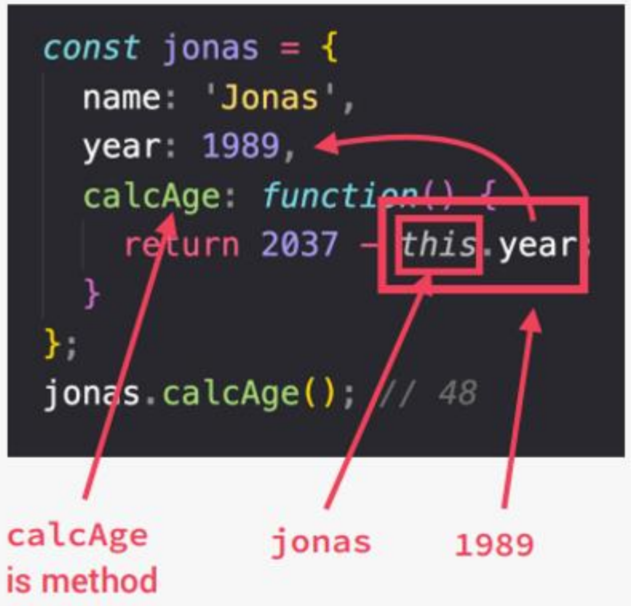
```
const q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const q3 = ["Jul", "Aug", "Sep"];
const q4 = ["Oct", "Nov", "May"];

const year = [...q1, ...q2, ...q3, ...q4];
```

Rest Operator ...



```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

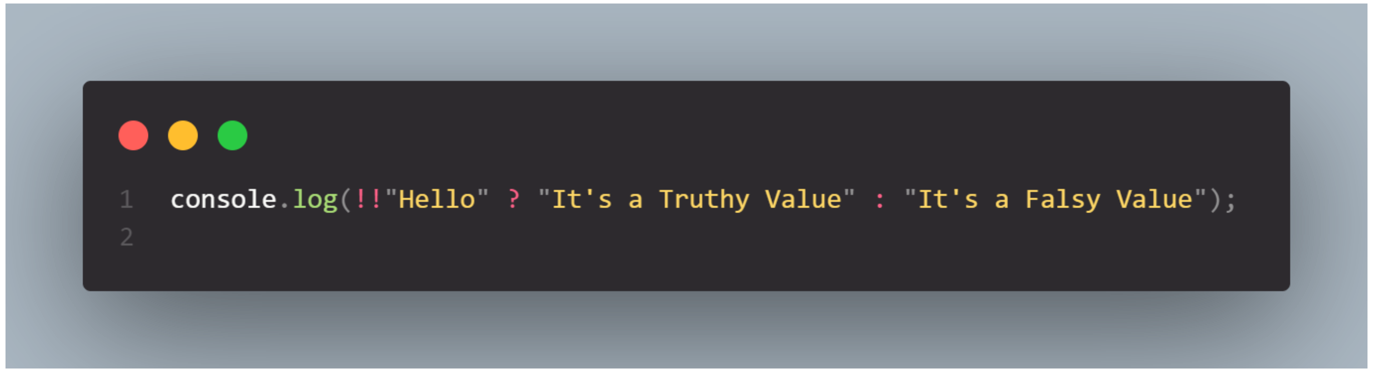
👉 Method example:



```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year;
  }
};
jonas.calcAge(); // 48
```

Way better than using `jonas.year`!

❑ Ternary Operator (? :)



```
console.log(!!"Hello" ? "It's a Truthy Value" : "It's a Falsy Value");
```

❑ Expression Vs Statement

In JavaScript, an expression is any valid unit of code that resolves to a value. An expression can be used as part of a statement. A statement is a unit of code that performs an action. Statements are executed for their side effects, while expressions are evaluated for their value

For Example: If else Statement & Ternary Operator

❑ Short Circuiting

In JavaScript, short-circuiting is a way of writing code that can save time and resources by not evaluating the second operand if the first operand is enough to determine the value of the expression. This is done using logical operators such as && and ||.

For example, in an AND expression (&&), if the first operand is false, JavaScript will short-circuit and not even look at the second operand.

Similarly, in an OR expression (||) & (??), if the first operand is true, then a short circuit occurs, evaluation stops, and true is returned.

❑ Classes

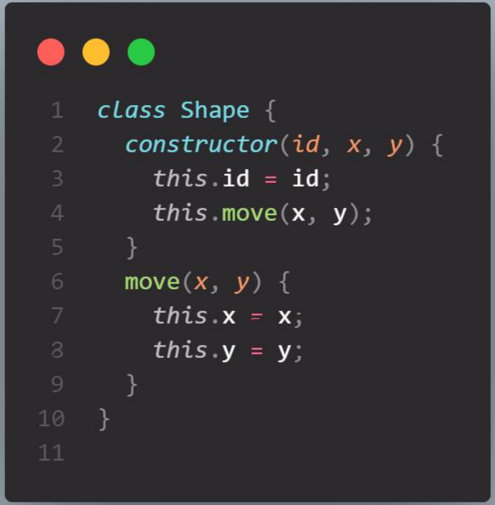
ES6 introduced classes in JavaScript. A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method

1 Constructor functions

- Technique to create objects from a function;
- This is how built-in objects like Arrays, Maps or Sets are actually implemented.

2 ES6 Classes

- Modern alternative to constructor function syntax;
- “Syntactic sugar”: behind the scenes, ES6 classes work **exactly** like constructor functions;
- ES6 classes do **NOT** behave like classes in “classical OOP” (last lecture).



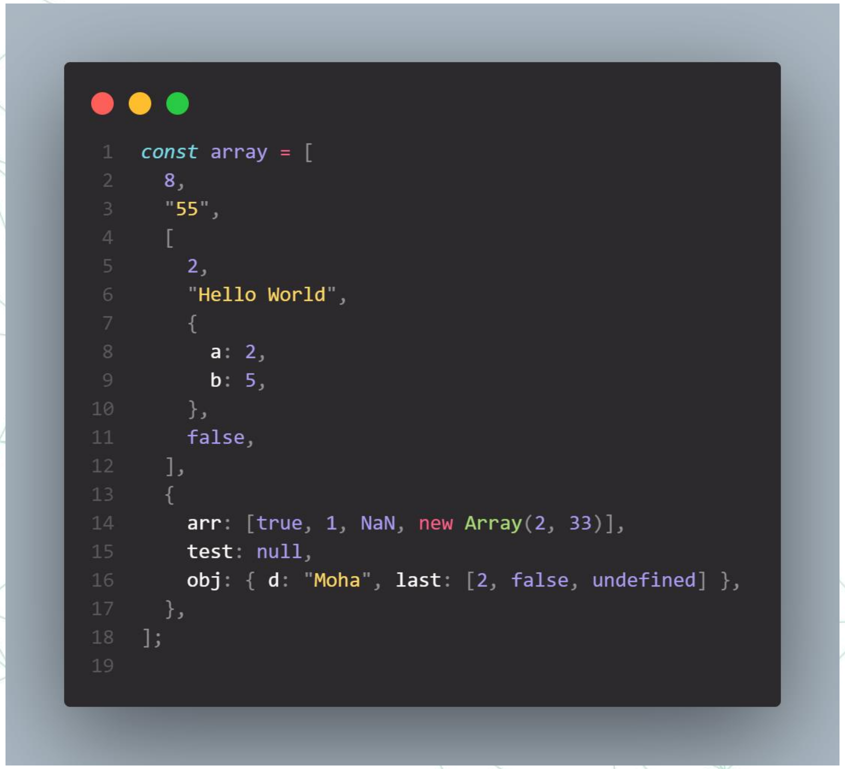
```
class Shape {
  constructor(id, x, y) {
    this.id = id;
    this.move(x, y);
  }
  move(x, y) {
    this.x = x;
    this.y = y;
  }
}
```

Any Questions

Time To Code

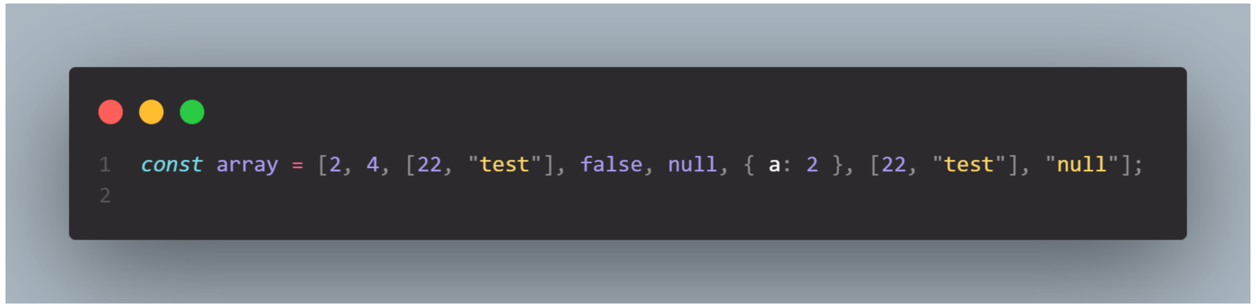
Using JavaScript do the following

De-structure the following Variable and extract 33 and “moha”



```javascript
const array = [
  8,
  "55",
  [
    2,
    "Hello World",
    {
      a: 2,
      b: 5,
    },
    false,
  ],
  {
    arr: [true, 1, NaN, new Array(2, 33)],
    test: null,
    obj: { d: "Moha", last: [2, false, undefined] },
  },
];
```

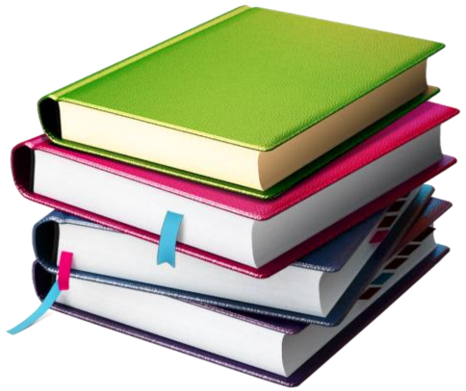
Find and Search for the best way to detect if the array has duplicates or not



```javascript
const array = [2, 4, [22, "test"], false, null, { a: 2 }, [22, "test"], "null"];
```

❑ References

- ES6 Features (W3schools)
- ES6 Features (Dev.to)



❑ Further Readings

- Closures.
- Higher Order Functions.
- Numeric Separators.