Javascript

JAVASCRIPT IS A HIGH-LEVEL,OBJECT-ORIENTED, MULTI-PARADIGM, PROGRAMMING LANGUAGE.

A programming language is basically just a tool that allows us to write code that will instruct a computer to do something.

JavaScript is a high-level language,which means that we don't have to think about a lot of complex stuff such as managing the computer's memory while it runs or program. So in JavaScript, there are a lot of so-called abstractions over all these small details that we don't want to worry about. And this makes the language a lot easier to write and to learn.

JavaScript is object oriented. And all that means is that the language is mostly based on the concept of objects for storing most kinds of data.

Finally, **JavaScript is also a multi-paradigm language**, meaning that it's so flexible and versatile, that we can use all kinds of different programming styles, such as imperative and declarative programming. And these different styles are just different ways of structuring our code, basically.

Now let's start building some stuff
Copy the code from this index.html link
https://github.com/jonasschmedtmann/complete-javascript-course/blob/master/01-Fundamentals-Part-1/starter/index.html

Or simply paste this code into a html file and get started

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
```

```
    <title>JavaScript Fundamentals - Part 1</title>
    <style>
      body {
        height: 100vh;
        display: flex;
        align-items: center;
        background: linear-gradient(to top left, #28b487, #7dd56f);
      }
      h1 {
        font-family: sans-serif;
        font-size: 50px;
        line-height: 1.3;
        width: 100%;
        padding: 30px;
        text-align: center;
        color: white;
      }
    </style>
  </head>
  <body>
    <h1>JavaScript Fundamentals - Part 1</h1>
  </body>
</html>
```

Now make a script.js and link that script to this html file between head tags

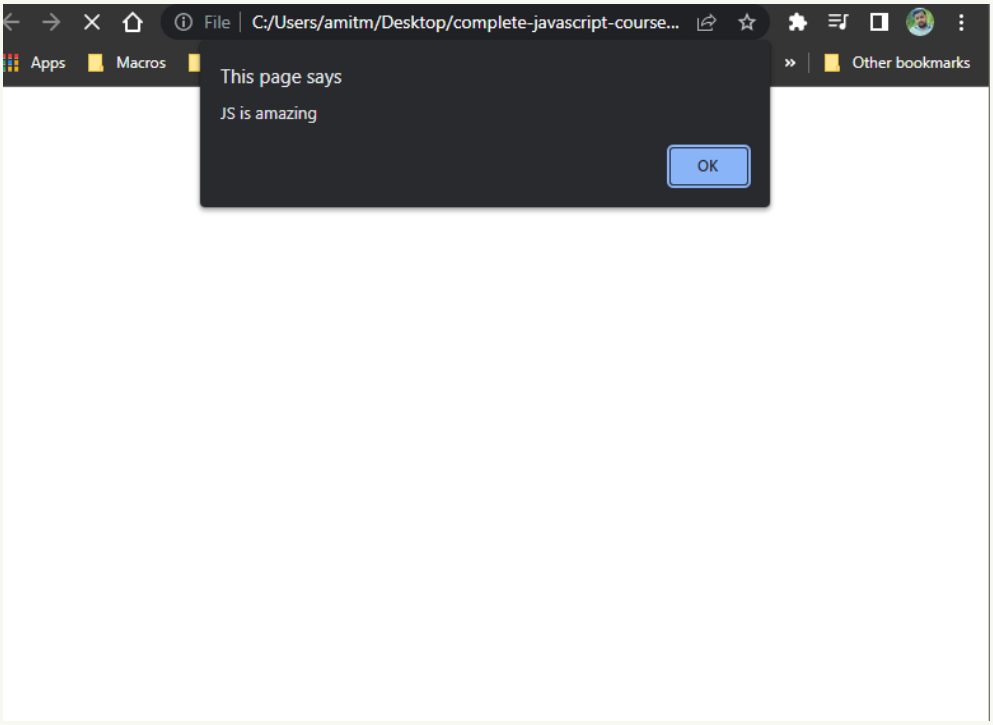
Index.html

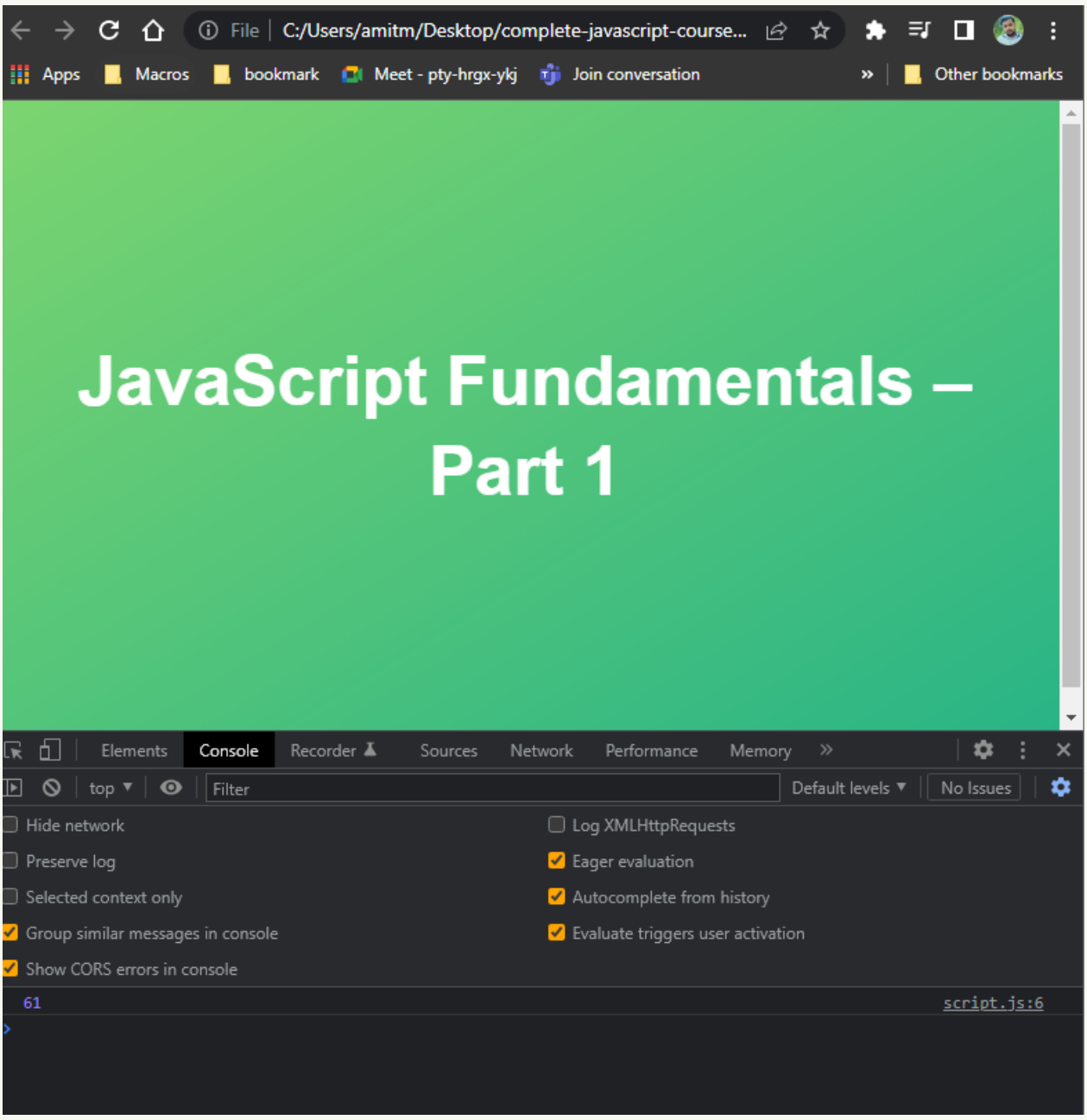
```
    <script src="script.js"></script>
```

script.js

```
let js = "amazing";
if(js==="amazing") alert("JS is amazing")
console.log(40 + 8 + 23 - 10);
```

Now when you open the index.html it will show this output


File | C:/Users/amitm/Desktop/complete-javascript-course...
Apps
Macros
Other bookmarks
This page says
JS is amazing
OK


File | C:/Users/amitm/Desktop/complete-javascript-course...
Apps
Macros
bookmark
Meet - pty-hrgx-ykj
Join conversation
Other bookmarks
JavaScript Fundamentals – Part 1
Elements
Console
Recorder
Sources
Network
Performance
Memory
top
Filter
Default levels
No Issues
Hide network
Preserve log
Selected context only
Group similar messages in console
Show CORS errors in console
Log XMLHttpRequests
Eager evaluation
Autocomplete from history
Evaluate triggers user activation
61
script.js:6

JavaScript Variables

4 Ways to Declare a JavaScript Variable:

- Using var
- Using let
- Using const
- Using nothing

Variables are containers for storing data (storing data values).

In this example, x, y, and z, are variables, declared with the let keyword:

```
let x = 5;
let y = 6;
let z = x + y;
```

When to Use JavaScript const?

If you want a general rule: always declare variables with const.
If you think the value of the variable can change, use let.

JS variable naming rules

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter
- Names can also begin with $ and _ (but we will not use it in this tutorial)
- Names are case sensitive (y and Y are different variables)
- Reserved words (like JavaScript keywords) cannot be used as variable names
- Also do not use caps as first letter of variable its just a convention in js that variables beginning with caps letter are class names like Person
- Write all variable names in caps if they are constant like PI
- Variable names should be descriptive like myFirstJob, myCurrentJob instead of job1, job2

Re-Declaring JavaScript Variables
If you re-declare a JavaScript variable declared with var, it will not lose its value.

The variable carName will still have the value "Volvo" after the execution of these statements:

```
var carName = "Volvo";
var carName;
```

You cannot re-declare a variable declared with let or const.

This will not work: it will throw an error

```
let carName = "Volvo";
let carName;
```

Let's practice variables

Open script.js and try creating variables and data type you learnet above
Here is one such example

```
console.log("Jonas");    // logging output to console
console.log(23);  // logging output to console
```

```
let firstName = "Matilda";  // create a varible to hold your first name

console.log(firstName);  // logging firstName output to console
console.log(firstName);  // logging firstName output to console
console.log(firstName);  // logging firstName output to console

// you see above if we chnage first name value in let line then all occurance value will be changed
// thats the reason we use varibles to hold some value so that we can use it many a times

// Variable name conventions
let jonas_matilda = "JM"; // you can use alphabets, numbers, underscore and dollar in the name of variable
let $function = 27;

let person = "jonas";
let PI = 3.1415;

let myFirstJob = "Coder";   // always try to give a meaningful name like this
let myCurrentJob = "Teacher";  // always try to give a meaningful name like this

let job1 = "programmer";  // this is not a meaningful variable name
let job2 = "teacher";  // this is not a meaningful variable name

console.log(myFirstJob);
```

Try to experiment with the code the more you experiment the more you learn

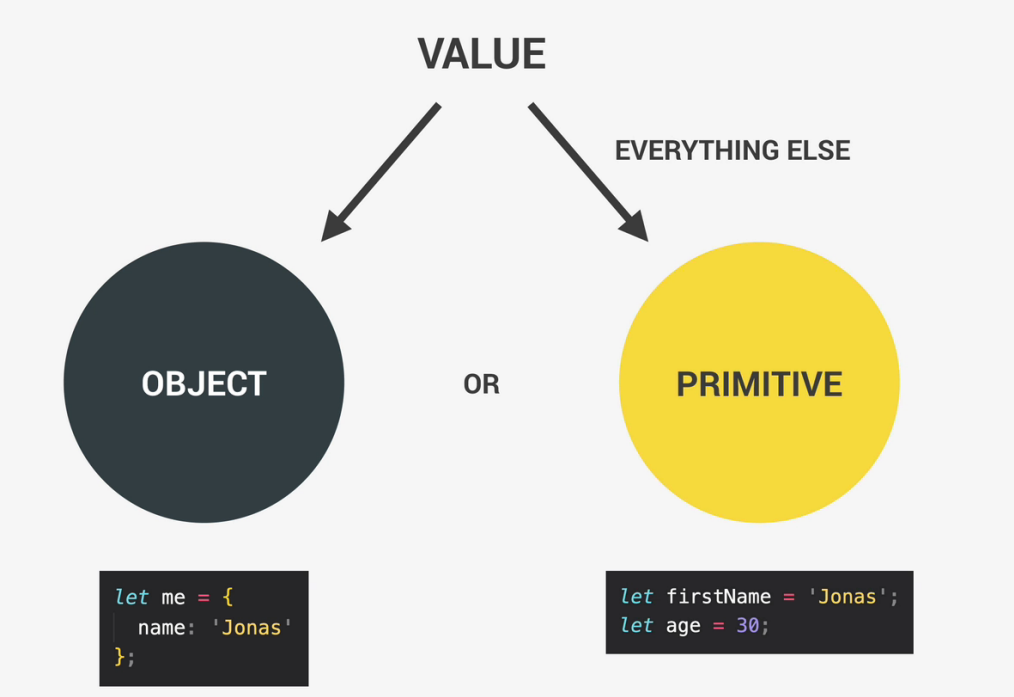
Now time for some assignment

Assignment1 :
1. Declare variables called 'country', 'continent' and 'population' and assign their values according to your own country (population in millions)
2. Log their values to the console

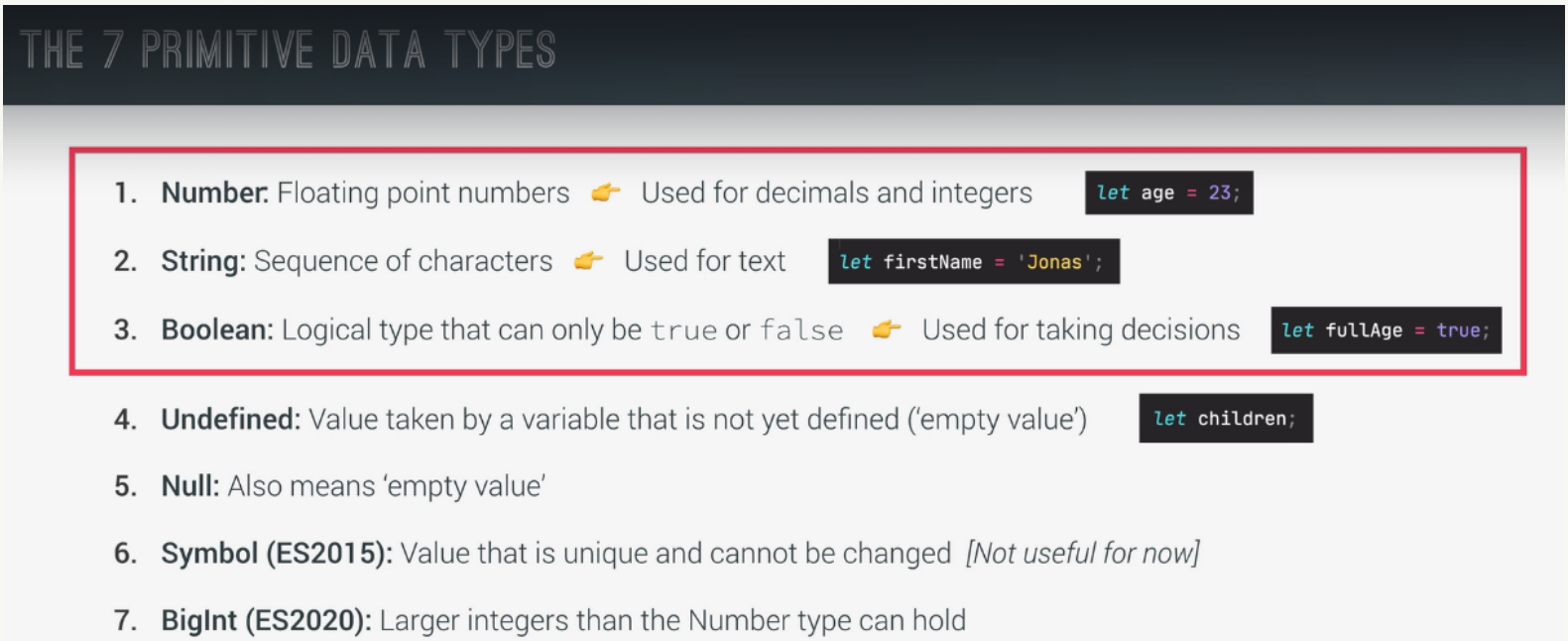
Be true to yourself do the assignments even if it is easy, then only you will learn.

Data Types

In js every value is either object or primitive data type



Different types of data types



JavaScript variables can hold different data types: numbers, strings, objects and more:
JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

```
let x;           // Now x is undefined
x = 5;           // Now x is a Number
x = "John";    // Now x is a String
```

Strings

```
let carName1 = "Volvo XC60";   // Using double quotes
let carName2 = 'Volvo XC60';   // Using single quotes

let answer1 = "It's alright";            // Single quote inside double quotes
let answer2 = "He is called 'Johnny'";   // Single quotes inside double quotes
```

```
let answer3 = 'He is called "Johnny"';    // Double quotes inside single quotes
```

Numbers

```
let x1 = 34.00;      // Written with decimals
let x2 = 34;         // Written without decimals

let y = 123e5;       // 12300000
let z = 123e-5;      // 0.00123
```

Booleans

```
let x = 5;
let y = 5;
let z = 6;
let a = (x == y)       // Returns true
let b = (x == z)       // Returns false
```

Undefined
Here both a and b are having a special value known as undefined

```
let a;
let b=undefined;
```

Null

```
// foo is known to exist now but it has no type or value:
var foo = null;
foo; //null
```

BigInt

BigInt is a primitive wrapper object used to represent and manipulate primitive bigint values — which are too large to be represented by the number primitive.

```
const previouslyMaxSafeInteger = 9007199254740991n // by ending a number with n

const alsoHuge = BigInt(9007199254740991) // by calling BigInt function
```

Arrays

```
const cars = ["Saab", "Volvo", "BMW"];
```

Objects

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue",
};
```

typeof Operator

```
typeof 0              // Returns "number"
```

```
typeof 314             // Returns "number"
typeof 3.14            // Returns "number"
typeof (3)             // Returns "number"
typeof (3 + 4)         // Returns "number"
typeof ""              // Returns "string"
typeof "John"          // Returns "string"
typeof "John Doe"      // Returns "string"
```

Difference between null and undefined
When checking for null or undefined, beware of the differences between equality (==) and identity (===) operators, as the former performs type-conversion.

```
typeof null           // "object" (not "null" for legacy reasons)
typeof undefined      // "undefined"
null === undefined    // false
null  == undefined    // true
null === null         // true
null  == null         // true
!null                 // true
isNaN(1 + null)       // false
isNaN(1 + undefined)  // true
```

Assignment:

1. Declare a variable called 'isIsland' and set its value according to your country. The variable should hold a Boolean value. Also declare a variable 'language', but don't assign it any value yet
2. Log the types of 'isIsland', 'population', 'country' and 'language' to the console

let, const and var

Var
Before the advent of ES6, var declarations ruled. There are issues associated with variables declared with var, though. That is why it was necessary for new ways to declare variables to emerge.

Scope of var
Scope essentially means where these variables are available for use. var declarations are globally scoped or function/locally scoped.

The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.

var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

To understand further, look at the example below.

```
var greeter = "hey hi";

function newFunction() {
    var hello = "hello";
}
```

Here, greeter is globally scoped because it exists outside a function while hello is function scoped. So we cannot access the variable hello outside of a function. So if we do this:

```
var tester = "hey hi";

function newFunction() {
    var hello = "hello";
}
console.log(hello); // error: hello is not defined
```

We'll get an error which is as a result of hello not being available outside the function.

var variables can be re-declared and updated
This means that we can do this within the same scope and won't get an error.

```
var greeter = "hey hi";
var greeter = "say Hello instead";
//and this also
var greeter = "hey hi";
greeter = "say Hello instead";
```

Hoisting of var
Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that if we do this:

```
console.log (greeter);
var greeter = "say hello"
```

it is interpreted as this:

```

var greeter;
console.log(greeter); // greeter is undefined
greeter = "say hello"
```

So var variables are hoisted to the top of their scope and initialized with a value of undefined.

Let
let is now preferred for variable declaration. It's no surprise as it comes as an improvement to var declarations. It also solves the problem with var that we just covered. Let's consider why this is so.

let is block scoped
A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.

So a variable declared in a block with let is only available for use within that block. Let me explain this with an example:

```
let greeting = "say Hi";
let times = 4;

if (times > 3) {
     let hello = "say Hello instead";
     console.log(hello);// "say Hello instead"
```

```
 }
console.log(hello) // hello is not defined
```

We see that using hello outside its block (the curly braces where it was defined) returns an error. This is because let variables are block scoped .

let can be updated but not re-declared.
Just like var, a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope. So while this will work:

```
let greeting = "say Hi";
greeting = "say Hello instead";
```

this will return an error:

```
let greeting = "say Hi";
let greeting = "say Hello instead"; // error: Identifier 'greeting' has already been declared
```

However, if the same variable is defined in different scopes, there will be no error:

```
let greeting = "say Hi";
if (true) {
    let greeting = "say Hello instead";
    console.log(greeting); // "say Hello instead"
}
console.log(greeting); // "say Hi"
```

Why is there no error? This is because both instances are treated as different variables since they have different scopes.

This fact makes let a better choice than var. When using let, you don't have to bother if you have used a name for a variable before as a variable exists only within its scope.

Also, since a variable cannot be declared more than once within a scope, then the problem discussed earlier that occurs with var does not happen.

Hoisting of let
Just like var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to use a let variable before declaration, you'll get a Reference Error.

Const
Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.

const declarations are block scoped
Like let declarations, const declarations can only be accessed within the block they were declared.

const cannot be updated or re-declared
This means that the value of a variable declared with const remains the same within its scope. It cannot be updated or re-declared. So if we declare a variable with const, we can neither do this:

```
const greeting = "say Hi";
    greeting = "say Hello instead";// error: Assignment to constant variable.
```

nor this:

```
const greeting = "say Hi";
const greeting = "say Hello instead";// error: Identifier 'greeting' has already been declared
```

Every const declaration, therefore, must be initialized at the time of declaration.

So we must give value to const at the time of declaration

```
const greeting // error: missing initializer in const declaration
```

This behavior is somehow different when it comes to objects declared with const. While a const object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a const object as this:

```
const greeting = {
    message: "say Hi",
    times: 4
}

greeting.message = "say Hello instead";
```

Hoisting of const
Just like let, const declarations are hoisted to the top but are not initialized.

Do not do like this ; doing like this will declare the variables in the global scope and not in the current scope

```
lastName = 'Schmedtmann';
console.log(lastName);
```

Assignment:

1. Set the value of 'language' to the language spoken where you live (some countries have multiple languages, but just choose one)
2. Think about which variables should be const variables (which values will never change, and which might change?). Then, change these variables to const.
3. Try to change one of the changed variables now, and observe what happens

JavaScript Operators

JavaScript Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

JavaScript String Operators
The + operator can also be used to add (concatenate) strings.

```
let text1 = "John";
let text2 = "Doe";
let text3 = text1 + " " + text2; //John Doe
```

Adding Strings and Numbers

Adding two numbers, will return the sum, but adding a number and a string will return a string:

```
let x = 5 + 5; // 10
let y = "5" + 5; //55
let z = "Hello" + 5; //Hello5
```

JavaScript Comparison Operators

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

JavaScript Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

```
// Logical Operators
const hasDriversLicense = true; // A
const hasGoodVision = true; // B

console.log(hasDriversLicense && hasGoodVision); // true
console.log(hasDriversLicense || hasGoodVision); // true
```

```
console.log(!hasDriversLicense); // false

if (hasDriversLicense && hasGoodVision) {
  console.log('Sarah is able to drive!');
} else {
  console.log('Someone else should drive...');
} // Sarah is able to drive!

const isTired = false; // C
console.log(hasDriversLicense && hasGoodVision && isTired); // false

if (hasDriversLicense && hasGoodVision && !isTired) {
  console.log('Sarah is able to drive!');
} else {
  console.log('Someone else should drive...');
} // Sarah is able to drive!
```

Assignment:

1. Comment out the previous code so the prompt doesn't get in the way

2. Let's say Sarah is looking for a new country to live in. She wants to live in a country that speaks english, has less than 50 million people and is not an island.

3. Write an if statement to help Sarah figure out if your country is right for her. You will need to write a condition that accounts for all of Sarah's criteria. Take your time with this, and check part of the solution if necessary.

4. If yours is the right country, log a string like this: 'You should live in Portugal :)'. If not, log 'Portugal does not meet your criteria :('

5. Probably your country does not meet all the criteria. So go back and temporarily change some variables in order to make the condition true (unless you live in Canada :D)

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

JavaScript Bitwise Operators
Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description
&	AND
|	OR
~	NOT
^	XOR
<<	left shift
>>	right shift
>>>	unsigned right shift

Assignement:
1. If your country split in half, and each half would contain half the population, then how many people would live in each half?

2. Increase the population of your country by 1 and log the result to the console
3. Finland has a population of 6 million. Does your country have more people than Finland?
4. The average population of a country is 33 million people. Does your country have less people than the average country?
5. Based on the variables you created, create a new variable 'description' which contains a string with this format: 'Portugal is in Europe, and its 11 million people speak portuguese

JavaScript Operator Precedence Values

Precedence	Operator type	Associativity	Individual operators
19	Grouping	n/a	(…)
18	Member Access	left-to-right	… . …
	Computed Member Access	left-to-right	… […]
	new (with argument list)	n/a	new … (…)
	Function Call	left-to-right	… (…)
	Optional chaining	left-to-right	?.
17	new (without argument list)	right-to-left	new …
16	Postfix Increment	n/a	… ++
	Postfix Decrement		… --
15	Logical NOT (!)	right-to-left	! …
	Bitwise NOT (~)		~ …
	Unary plus (+)		+ …
	Unary negation (-)		- …
	Prefix Increment		++ …

	Prefix Decrement		-- …
	typeof		typeof …
	void		void …
	delete		delete …
	await		await …
14	Exponentiation (**)	right-to-left	… ** …
13	Multiplication (*)	left-to-right	… * …
	Division (/)		… / …
	Remainder (%)		… % …
12	Addition (+)	left-to-right	… + …
	Subtraction (-)		… - …
11	Bitwise Left Shift (<<)	left-to-right	… << …
	Bitwise Right Shift (>>)		… >> …
	Bitwise Unsigned Right Shift (>>>)		… >>> …
10	Less Than (<)	left-to-right	… < …
	Less Than Or Equal (<=)		… <= …
	Greater Than (>)		… > …
	Greater Than Or Equal (>=)		… >= …
	in		… in …
	instanceof		… instanceof …
9	Equality (==)	left-to-right	… == …

	Inequality (!=)		… != …
	Strict Equality (===)		… === …
	Strict Inequality (!==)		… !== …
8	Bitwise AND (&)	left-to-right	… & …
7	Bitwise XOR (^)	left-to-right	… ^ …
6	Bitwise OR (|)	left-to-right	… | …
5	Logical AND (&&)	left-to-right	… && …
4	Logical OR (||)	left-to-right	… || …
	Nullish coalescing operator (??)	left-to-right	… ?? …
3	Conditional (ternary) operator	right-to-left	… ? … : …
2	Assignment	right-to-left	… = …
			… += …
			… -= …
			… **= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …

			… |= …
			… &&= …
			… ||= …
			… ??= …
	yield	right-to-left	yield …
	yield*		yield* …
1	Comma / Sequence	left-to-right	… , …

Coding Challenge #1

Mark and John are trying to compare their BMI (Body Mass Index), which is calculated using the formula:
BMI = mass / height ** 2 = mass / (height * height) (mass in kg and height in meter).

Your tasks:
1. Store Mark's and John's mass and height in variables
2. Calculate both their BMIs using the formula (you can even implement both versions)
3. Create a Boolean variable 'markHigherBMI' containing information about whether Mark has a higher BMI than John.

Test data:

Data 1: Marks weights 78 kg and is 1.69 m tall. John weights 92 kg and is 1.95 m tall.
Data 2: Marks weights 95 kg and is 1.88 m tall. John weights 85 kg and is 1.76 m tall.

JavaScript Strings

You can use single or double quotes:

```
let carName1 = "Volvo XC60";  // Double quotes
let carName2 = 'Volvo XC60';  // Single quotes
```

String Length
To find the length of a string, use the built-in length property:

Example

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let length = text.length;
```

Escape Character

```
let text= 'It\'s alright.';
```

JavaScript Strings as Objects
Normally, JavaScript strings are primitive values, created from literals:

```
let x = "John";
```

But strings can also be defined as objects with the keyword new:

```
let y = new String("John");
```

Do not create Strings objects.
The new keyword complicates the code and slows down execution speed.
String objects can produce unexpected results:

```
//When using the == operator, x and y are equal:

let x = "John";
let y = new String("John");

//When using the === operator, x and y are not equal:

let x = "John";
let y = new String("John");


//Comparing two JavaScript objects always returns false.
//(x == y) true or false? the result will be false

let x = new String("John");
let y = new String("John");

//(x === y) true or false? the result will be false

let x = new String("John");
let y = new String("John");
```

JavaScript Template Literals

Back-Tics Syntax
Template Literals use back-ticks (``) rather than the quotes ("") to define a string:

```
let text = `Hello World!`;
```

Quotes Inside Strings
With template literals, you can use both single and double quotes inside a string:

```
let text = `He's often called "Johnny"`;
```

Multiline Strings
Template literals allows multiline strings:

```
let text =
`The quick
brown fox
jumps over
the lazy dog`;
```

Interpolation
Template literals provide an easy way to interpolate variables and expressions into strings.

The method is called string interpolation.

Variable Substitutions
Template literals allow variables in strings:

```
let firstName = "John";
let lastName = "Doe";

let text = `Welcome ${firstName}, ${lastName}!`;
```

Inside the curly braces you can write variables, javascript expression, and html templates

Assignment:
1. Recreate the 'description' variable from the last assignment, this time using the template literal syntax

JavaScript if, else, and else if

Conditional statements are used to perform different actions based on different conditions.

The if Statement

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

```
if (hour < 18) {
    greeting = "Good day";
  }
```

The else Statement
Use the else statement to specify a block of code to be executed if the condition is false.

```
if (hour < 18) {
    greeting = "Good day";
  } else {
    greeting = "Good evening";
  }
```

The else if Statement
Use the else if statement to specify a new condition if the first condition is false.

```
if (time < 10) {
    greeting = "Good morning";
  } else if (time < 20) {
    greeting = "Good day";
  } else {
    greeting = "Good evening";
  }
```

Assignment:

1. If your country's population is greater that 33 million, log a string like this to the console: 'Portugal's population is above average'. Otherwise, log a string like 'Portugal's population is 22 million below average' (the 22 is the average of 33 minus the country's population)

2. After checking the result, change the population temporarily to 13 and then to 130. See the different results, and set the population back to original

Coding Challenge #2

Use the BMI example from Challenge #1, and the code you already wrote, and improve it.

Your tasks:
1. Print a nice output to the console, saying who has the higher BMI. The message is either "Mark's BMI is higher than John's!" or "John's BMI is higher than Mark's!"
2. Use a template literal to include the BMI values in the outputs. Example: "Mark's BMI (28.3) is higher than John's (23.9)!"

Hint: Use an if/else statement

Type conversion and Coercion

```
const inputYear = '1991';
console.log(Number(inputYear), inputYear);  // this will output 1991 '1991' as number and
string
console.log(inputYear + 18); // 199118

console.log(Number(inputYear) + 18); // 2009

console.log(Number('Jonas')); // NaN
console.log(typeof NaN); // number

console.log(String(23), 23); // 23 23   as string and number

// type coercion
console.log('I am ' + 23 + ' years old'); //  I am 23 years old
console.log('23' - '10' - 3); // 10
console.log('23' / '2'); // 11.5

let n = '1' + 1; // '11'
n = n - 1;
console.log(n); //  10
```

Here we can see that we can convert any numerical string into number using a function Number()
We can not add directly numerical string and number, doing this will concatenate the values
And if we try to convert non numerical string to number using Number() function it will give NaN which is not a number
And we can convert any number to string by using String() function

Type Coercion
Only + operator will do type coercion i.e it will convert number to string if that number is added to any sting (numbers converted to string)
On the other hand
Other operator like - * / will not work like this if you apply these operators between any numerical string then result will be evaluation of the expression mathematically (string converted to numbers)

Assignment :

1. Predict the result of these 5 operations without executing them:

'9' - '5';
'19' - '13' + '17';
'19' - '13' + 17;
'123' < 57;
5 + 6 + '4' + 9 - 4 - 2;

2. Execute the operations to check if you were right

Truthy and Falcy values

```
// 5 falsy values: 0, '', undefined, null, NaN
console.log(Boolean(0));  //  false
console.log(Boolean(undefined));  // false
console.log(Boolean('Jonas'));  //  true
console.log(Boolean({}));  //  true
console.log(Boolean(''));  //  false

const money = 100;
if (money) {
  console.log("Don't spend it all");
} else {
  console.log('You should get a job!');
}  //  Don't spend it all

let height = 0;
if (height) {
  console.log('YAY! Height is defined');
} else {
  console.log('Height is UNDEFINED');
}  //   Height is UNDEFINED
```

5 falsy values: 0, '', undefined, null, NaN

These 5 values will be converted to false if we convert to boolean

Assignment:

1. Declare a variable 'numNeighbours' based on a prompt input like this: prompt('How many neighbour countries does your country have?');

2. If there is only 1 neighbour, log to the console 'Only 1 border!' (use loose equality == for now)

3. Use an else-if block to log 'More than 1 border' in case 'numNeighbours' is greater than 1

4. Use an else block to log 'No borders' (this block will be executed when 'numNeighbours' is 0 or any other value)

5. Test the code with different values of 'numNeighbours', including 1 and 0.

6. Change == to ===, and test the code again, with the same values of 'numNeighbours'. Notice what happens when there is exactly 1 border! Why is this happening?

7. Finally, convert 'numNeighbours' to a number, and watch what happens now when you input 1

8. Reflect on why we should use the === operator and type conversion in this situation

Equality operator == and ====

```
// Equality Operators: == vs. ===
const age = '18';
if (age === 18) console.log('You just became an adult :D (strict)');  // this will not log as this is strict checking

if (age == 18) console.log('You just became an adult :D (loose)');  // You just became an adult :D (loose)

const favorite = Number(prompt("What's your favorite number?"));  // it will give prompt and ask for an input
console.log(favorite);  // 19
console.log(typeof favorite);  // number

if (favorite === 23) { // 22 === 23 -> FALSE
  console.log('Cool! 23 is an amazing number!')
} else if (favorite === 7) {
  console.log('7 is also a cool number')
} else if (favorite === 9) {
  console.log('9 is also a cool number')
} else {
  console.log('Number is not 23 or 7 or 9')
}
  // Number is not 23 or 7 or 9

if (favorite !== 23) console.log('Why not 23?');
  //  Why not 23?
```

=== will check for stict checking means value and type of both lhs and rhs should match , it will not do type coercion
== will check only for value and behind the scene it will do type coercion for comparing the lhs and rhs

Coding Challenge #3

There are two gymnastics teams, Dolphins and Koalas. They compete against each other 3 times. The winner with the highest average score wins a trophy!

Your tasks:

1. Calculate the average score for each team, using the test data below

2. Compare the team's average scores to determine the winner of the competition, and print it to the console. Don't forget that there can be a draw, so test for that as well (draw means they have the same average score)

3. Bonus 1: Include a requirement for a minimum score of 100. With this rule, a team only wins if it has a higher score than the other team, and the same time a score of at least 100 points. Hint: Use a logical operator to test for minimum score, as well as multiple else-if blocks

4. Bonus 2: Minimum score also applies to a draw! So a draw only happens when both teams have the same score and both have a score greater or equal 100 points. Otherwise, no team wins the trophy

Test data:
§ Data 1: Dolphins score 96, 108 and 89. Koalas score 88, 91 and 110
§ Data Bonus 1: Dolphins score 97, 112 and 101. Koalas score 109, 95 and 123
§ Data Bonus 2: Dolphins score 97, 112 and 101. Koalas score 109, 95 and 106

```
// const scoreDolphins = (96 + 108 + 89) / 3;
// const scoreKoalas = (88 + 91 + 110) / 3;
// console.log(scoreDolphins, scoreKoalas);

// if (scoreDolphins > scoreKoalas) {
//   console.log('Dolphins win the trophy 🏆');
// } else if (scoreKoalas > scoreDolphins) {
//   console.log('Koalas win the trophy 🏆');
// } else if (scoreDolphins === scoreKoalas) {
//   console.log('Both win the trophy!');
// }

// BONUS 1
const scoreDolphins = (97 + 112 + 80) / 3;
const scoreKoalas = (109 + 95 + 50) / 3;
console.log(scoreDolphins, scoreKoalas);

if (scoreDolphins > scoreKoalas && scoreDolphins >= 100) {
  console.log('Dolphins win the trophy 🏆');
} else if (scoreKoalas > scoreDolphins && scoreKoalas >= 100) {
  console.log('Koalas win the trophy 🏆');
} else if (scoreDolphins === scoreKoalas && scoreDolphins >= 100 && scoreKoalas >= 100) {
  console.log('Both win the trophy!');
} else {
  console.log('No one wins the trophy 😭');
```

The switch Statement

```


////////////////////////////////////
// The switch Statement
const day = 'friday';
switch (day) {
  case 'monday': // day === 'monday'
```

```
    console.log('Plan course structure');
    console.log('Go to coding meetup');
    break;
  case 'tuesday':
    console.log('Prepare theory videos');
    break;
  case 'wednesday':
  case 'thursday':
    console.log('Write code examples');
    break;
  case 'friday':
    console.log('Record videos');
    break;
  case 'saturday':
  case 'sunday':
    console.log('Enjoy the weekend :D');
    break;
  default:
    console.log('Not a valid day!');
} // Record videos


if (day === 'monday') {
  console.log('Plan course structure');
  console.log('Go to coding meetup');
} else if (day === 'tuesday') {
  console.log('Prepare theory videos');
} else if (day === 'wednesday' || day === 'thursday') {
  console.log('Write code examples');
} else if (day === 'friday') {
  console.log('Record videos');
} else if (day === 'saturday' || day === 'sunday') {
  console.log('Enjoy the weekend :D');
} else {
  console.log('Not a valid day!');
} // Record videos
```

Assignment :

1. Use a switch statement to log the following string for the given 'language':

chinese or mandarin: 'MOST number of native speakers!'

spanish: '2nd place in number of native speakers'

english: '3rd place'

hindi: 'Number 4'

arabic: '5th most spoken language'

for all other simply log 'Great language too :D'

Statement and Expression

an expression is a piece of code that produces a value.
Eg. 3+4
1991
True && false && !false

Statement is a bigger piece of code which does not produce a value on it own
Eg. sequence of statements in if else block
Or switch statement

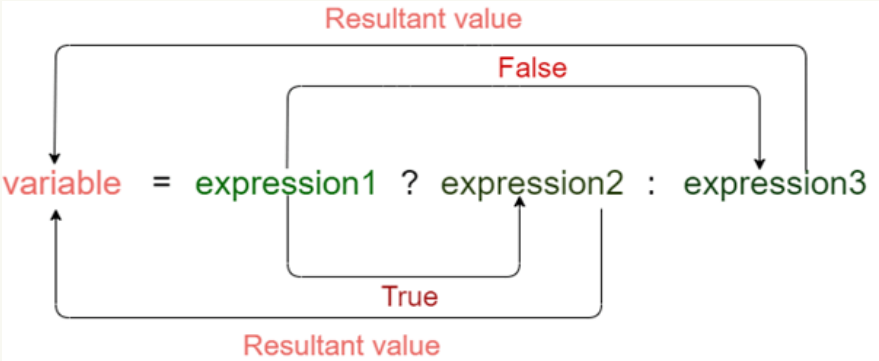
```
// Statements and Expressions
3 + 4
1991
true && false && !false

if (23 > 10) {
  const str = '23 is bigger';     // ‘23 is bigger’ is an expression and this complete line is an statement
}

const me = 'Jonas';
console.log(`I'm ${2037 - 1991} years old ${me}`); // I'm 46 years old Jonas
```

So here in template literal only allows expression in curly braces { } so here we can use anything which produces value but we cant use statement inside { }. Try placing and statement in { } and you will see it will throw error

The conditional operator

```
// The Conditional (Ternary) Operator
const age = 23;
 age >= 18 ? console.log('I like to drink wine 🍷') : console.log('I like to drink water 💧');  // I like to drink wine 🍷

const drink = age >= 18 ? 'wine 🍷' : 'water 💧';
console.log(drink);   // wine 🍷

let drink2;
```

```
if (age >= 18) {
  drink2 = 'wine 🍷';
} else {
  drink2 = 'water 💧';
}   //
console.log(drink2);    // wine 🍷

console.log(`I like to drink ${age >= 18 ? 'wine 🍷' : 'water 💧'}`);    // I like to drink
wine 🍷
```



Conditional operator is an operator , also operator produces a value, so it is an expression so we can use it in template literal
So instead of if else we can use conditional operator

Assignment:

1. If your country's population is greater than 33 million, use the ternary operator to log a string like this to the console: 'Portugal's population is above average'. Otherwise, simply log 'Portugal's population is below average'. Notice how only one word changes between these two sentences!

2. After checking the result, change the population temporarily to 13 and then to 130. See the different results, and set the population back to original

Coding Challenge #4

Steven wants to build a very simple tip calculator for whenever he goes eating in a restaurant. In his country, it's usual to tip 15% if the bill value is between 50 and 300. If the value is different, the tip is 20%.

Your tasks:

1. Calculate the tip, depending on the bill value. Create a variable called 'tip' for this. It's not allowed to use an if/else statement 🙂 (If it's easier for you, you can start with an if/else statement, and then try to convert it to a ternary operator!)

2. Print a string to the console containing the bill value, the tip, and the final value (bill + tip). Example: “The bill was 275, the tip was 41.25, and the total value 316.25”

Test data:

§ Data 1: Test for bill values 275, 40 and 430

Hints:
§ To calculate 20% of a value, simply multiply it by 20/100 = 0.2
§ Value X is between 50 and 300, if it's >= 50 && <= 300 ✅

```
const bill = 430;
const tip = bill <= 300 && bill >= 50 ? bill * 0.15 : bill * 0.2;
console.log(`The bill was ${bill}, the tip was ${tip}, and the total value ${bill + tip}`);
```

Activating Strict mode

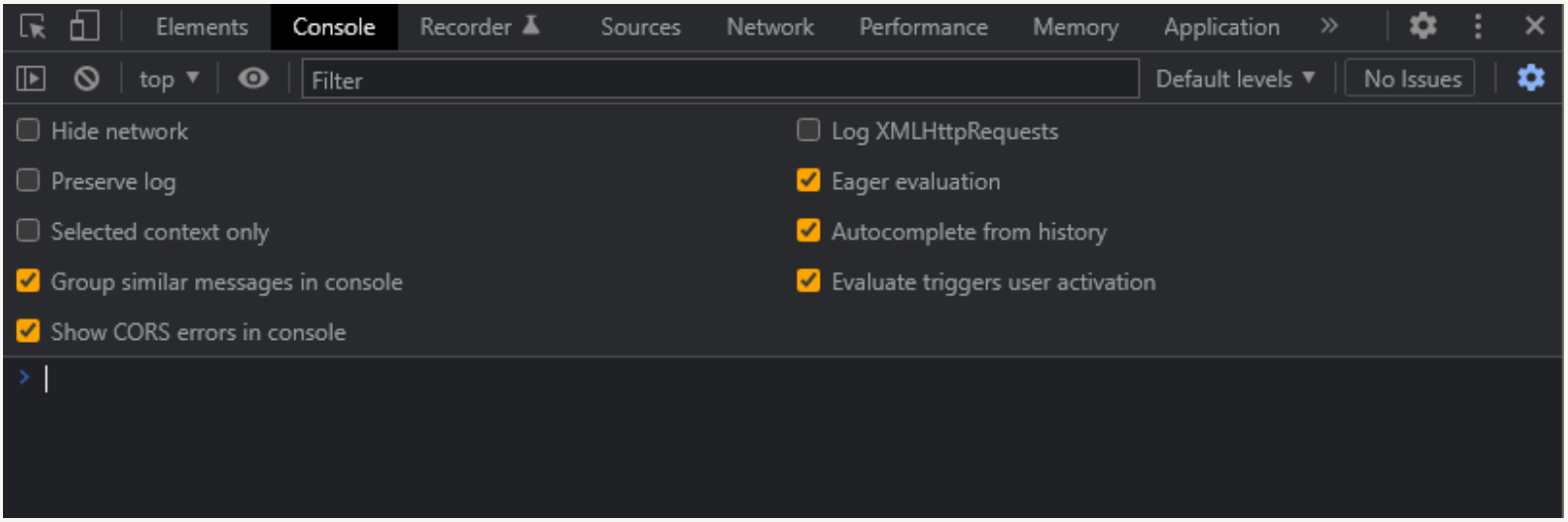
To active strict mode in js
Just type below line in the first line of your js file
Strict mode makes several changes to normal JavaScript semantics:

Eliminates some JavaScript silent errors by changing them to throw errors.
Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
Prohibits some syntax likely to be defined in future versions of ECMAScript.

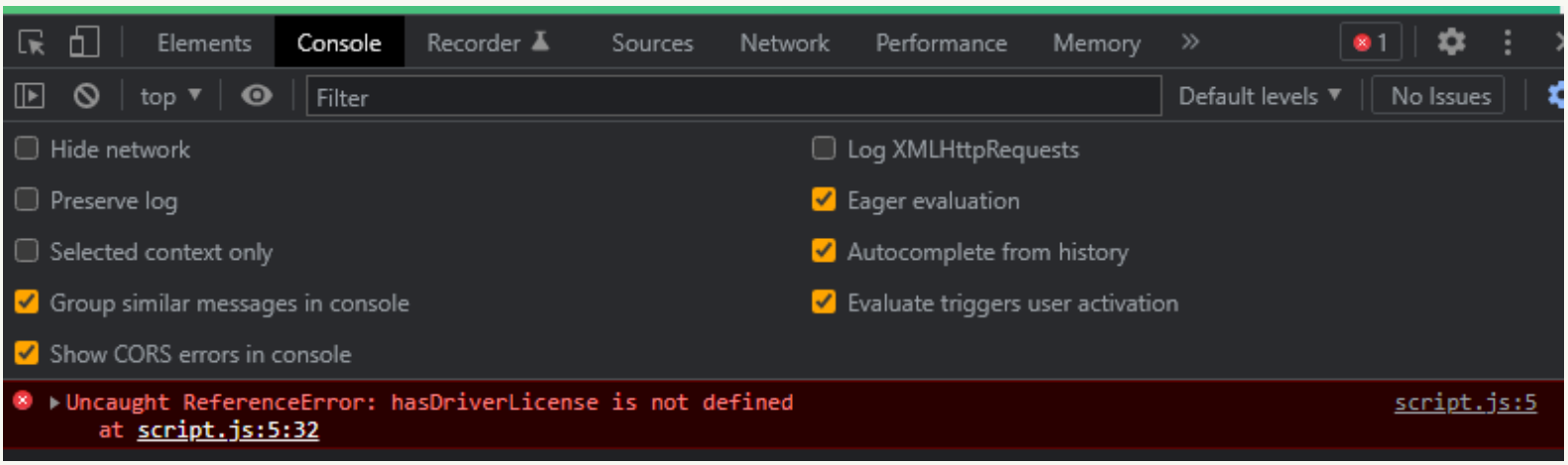
```
'use strict';
```

It Will throw err so that you will not fall in bug later in the development

```
//'use strict';
let hasDriversLicense = false;
const passTest = true;

if (passTest) hasDriverLicense = true;   // here we have used wrong varibale name we missed s in the variable name so if we dont use use Strict console will not throw any err when when we enable use Strict then it will shrow an err
if (hasDriversLicense) console.log('I can drive :D');

const interface = 'Audio';  // will throw err in strict mode as it is reserved word by js for future use
const private = 534;  // will throw err in strict mode as it is reserved word by js for future use
```



Now let's active strict mode and see



So using strict mode is beneficial

Functions

```
'use strict';
// Functions
function logger() {
  console.log('My name is Jonas');
}

// calling / running / invoking function
logger();  // My name is Jonas
logger();  // My name is Jonas
logger();  // My name is Jonas
```

```
function fruitProcessor(apples, oranges) {
  const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
  return juice;
}

const appleJuice = fruitProcessor(5, 0);
console.log(appleJuice);  // Juice with 5 apples and 0 oranges.

const appleOrangeJuice = fruitProcessor(2, 4);
console.log(appleOrangeJuice);  // Juice with 2 apples and 4 oranges.
```

Functions are the statements which can be executed any number of times as per the need without writing multiple times

Assignment:

1. Write a function called 'describeCountry' which takes three parameters: 'country', 'population' and 'capitalCity'. Based on this input, the function returns a string with this format: 'Finland has 6 million people and its capital city is Helsinki'

2. Call this function 3 times, with input data for 3 different countries. Store the returned values in 3 different variables, and log them to the console

Function Declarations vs. Expressions

```
'use strict';
// Function Declarations vs. Expressions

// Function declaration
function calcAge1(birthYeah) {  // here in this line birthYeah is parameter
  return 2037 - birthYeah;
}
const age1 = calcAge1(1991); // here the value which are passing to the function is know as argument

// Function expression
const calcAge2 = function (birthYeah) {
  return 2037 - birthYeah;
}
const age2 = calcAge2(1991);

console.log(age1, age2);  // 46 46
```

The only difference in both are

We can call function declaration before we have defined the code
But we can not call a function expression before we have defined it
Remember hoisting

Assignment:

1. The world population is 7900 million people. Create a function declaration called 'percentageOfWorld1' which receives a 'population' value, and returns the percentage of the world population that the given population represents. For example, China has 1441 million people, so it's about 18.2% of the world population

2. To calculate the percentage, divide the given 'population' value by 7900 and then multiply by 100

3. Call 'percentageOfWorld1' for 3 populations of countries of your choice, store the results into variables, and log them to the console

4. Create a function expression which does the exact same thing, called 'percentageOfWorld2', and also call it with 3 country populations (can be the same populations)

Arrow Functions

```
'use strict';
// Function expression
const calcAge2 = function (birthYeah) {
  return 2037 - birthYeah;
}
// Arrow functions
const calcAge3 = birthYeah => 2037 - birthYeah;
const age3 = calcAge3(1991);
console.log(age3);   //  46

const yearsUntilRetirement = (birthYeah, firstName) => {
  const age = 2037 - birthYeah;
  const retirement = 65 - age;
  // return retirement;
  return `${firstName} retires in ${retirement} years`;
}
console.log(yearsUntilRetirement(1991, 'Jonas'));    //  Jonas retires in 19 years
console.log(yearsUntilRetirement(1980, 'Bob'));     //  Bob retires in 8 years
```

Arrow function is simple way to write the function

An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

There are differences between arrow functions and traditional functions, as well as some limitations:

Arrow functions don't have their own bindings to this, arguments or super, and should not be used as methods.
Arrow functions don't have access to the new.target keyword.
Arrow functions aren't suitable for call, apply and bind methods, which generally rely on establishing a scope.
Arrow functions cannot be used as constructors.
Arrow functions cannot use yield, within its body.

Assignment
1. Recreate the last assignment, but this time create an arrow function called 'percentageOfWorld3'

Functions Calling Other Functions

```
'use strict';
// Functions Calling Other Functions
function cutFruitPieces(fruit) {
  return fruit * 4;
}

function fruitProcessor(apples, oranges) {
  const applePieces = cutFruitPieces(apples);
  const orangePieces = cutFruitPieces(oranges);

  const juice = `Juice with ${applePieces} piece of apple and ${orangePieces} pieces of
orange.`;
  return juice;
}
console.log(fruitProcessor(2, 3)); //  Juice with 8 piece of apple and 12 pieces of orange.
```

Here we have called one function inside another

Assignment :

1. Create a function called 'describePopulation'. Use the function type you like the most. This function takes in two arguments: 'country' and 'population', and returns a string like this: 'China has 1441 million people, which is about 18.2% of the world.'

2. To calculate the percentage, 'describePopulation' call the 'percentageOfWorld1' you created earlier

3. Call 'describePopulation' with data for 3 countries of your choice

👉 **Function declaration**

Function that can be used before it's declared

👉 **Function expression**

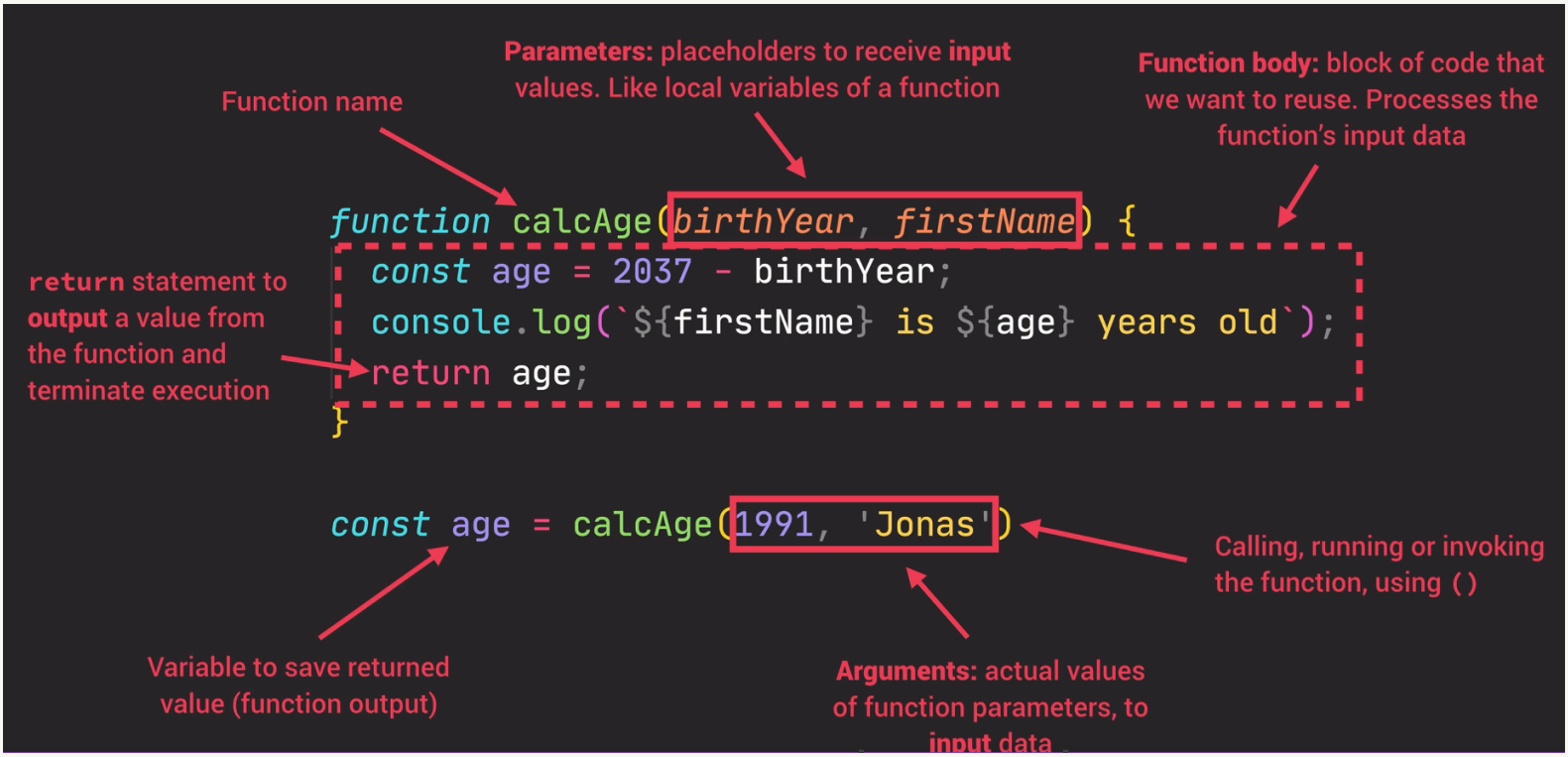
Essentially a function *value* stored in a variable

👉 **Arrow function**

Great for a quick one-line functions. Has no `this` keyword (more later...)

```
function calcAge(birthYear) {
  return 2037 - birthYear;
}

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

const calcAge = birthYear ⇒ 2037 - birthYear;
```

☝ Three different ways of writing functions, but they all work in a similar way: receive **input** data, **transform** data, and then **output** data.



Coding Challenge #1

Back to the two gymnastics teams, the Dolphins and the Koalas! There is a new gymnastics discipline, which works differently. Each team competes 3 times, and then the average of the 3 scores is calculated (so one average score per team). A team only wins if it has at least double the average score of the other team. Otherwise, no team wins!

Your tasks:

1. Create an arrow function 'calcAverage' to calculate the average of 3 scores

2. Use the function to calculate the average for both teams

3. Create a function 'checkWinner' that takes the average score of each team as parameters ('avgDolhins' and 'avgKoalas'), and then logs the winner to the console, together with the victory points, according to the rule above. Example: "Koalas win (30 vs. 13)"

4. Use the 'checkWinner' function to determine the winner for both Data 1 and Data 2

5. Ignore draws this time

Test data:

§ Data 1: Dolphins score 44, 23 and 71. Koalas score 65, 54 and 49

§ Data 2: Dolphins score 85, 54 and 41. Koalas score 23, 34 and 27

Hints:
§ To calculate average of 3 values, add them all together and divide by 3
§ To check if number A is at least double number B, check for A >= 2 * B. Apply this to the team's average scores

Reviewing Functions

Go though the example just for understanding

```
'use strict';

// Reviewing Functions
const calcAge = function (birthYeah) {
  return 2037 - birthYeah;
}

const yearsUntilRetirement = function (birthYeah, firstName) {
  const age = calcAge(birthYeah);
  const retirement = 65 - age;

  if (retirement > 0) {
    console.log(`${firstName} retires in ${retirement} years`);
    return retirement;
  } else {
    console.log(`${firstName} has already retired 🎉`);
    return -1;
  }
}

console.log(yearsUntilRetirement(1991, 'Jonas')); // Jonas retires in 19 years    // 19
console.log(yearsUntilRetirement(1950, 'Mike'));  // Mike has already retired 🎉    // -1
```

Arrays

An array is a special variable, which can hold more than one value:

Ways to declare array

```
const cars = ["Saab", "Volvo", "BMW"];
```

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

```
const cars = new Array("Saab", "Volvo", "BMW");
```

Accessing Array Elements
You access an array element by referring to the index number:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];  // Saab
```

Changing array elements
We can change the value of the elements of the array, although the cars is a constant variable.

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";     // leagle
```

But we can not change the value of cars as complete as it is constant

```
const cars = ["Saab", "Volvo", "BMW"];
cars =['Audi','Porsche']   //Illegal
```

Arrays are objects

Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.

But, JavaScript arrays are best described as arrays.

Arrays use numbers to access its "elements". In this example, person[0] returns John:

```
const person = ["John", "Doe", 46];
```

Objects use names to access its "members". In this example, person.firstName returns John:

```
const person = {firstName:"John", lastName:"Doe", age:46};
```

WARNING !!
If you use named indexes, JavaScript will redefine the array to an object.

After that, some array methods and properties will produce incorrect results.

```
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
person.length;     // Will return 0
```

```
person[0];          // Will return undefined
```

The Difference Between Arrays and Objects
In JavaScript, arrays use numbered indexes.

In JavaScript, objects use named indexes.

Arrays are a special kind of objects, with numbered indexes.

Array Properties and Methods
The real strength of JavaScript arrays are the built-in array properties and methods:

```
cars.length   // Returns the number of elements
cars.sort()   // Sorts the array
```

Looping through the array

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fLen = fruits.length;

for (let i = 0; i < fLen; i++) {
    console.log(fruits[i]);
}
```

You can also use the Array.forEach() function:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.forEach(myFunction);

function myFunction(value) {
  console.log(value);
}
```

Adding Array Elements
The easiest way to add a new element to an array is using the push() method:

```
const fruits = ["Banana", "Orange", "Apple"];
fruits.push("Lemon");  // Adds a new element (Lemon) to fruits
```

New element can also be added to an array using the length property:

```
const fruits = ["Banana", "Orange", "Apple"];
fruits[fruits.length] = "Lemon";  // Adds "Lemon" to fruits
```

WARNING !
Adding elements with high indexes can create undefined "holes" in an array:

```
Example
const fruits = ["Banana", "Orange", "Apple"];
fruits[6] = "Lemon";  // Creates undefined "holes" in fruits
```

JavaScript new Array()
JavaScript has a built in array constructor new Array().

But you can safely use [] instead.

These two different statements both create a new empty array named points:

const points = new Array();
const points = [];

But try to use [] for creating arrays as its safe and gives proper result

An Array exercise, just go though the example

```
'use strict';
// Introduction to Arrays
const friend1 = 'Michael';
const friend2 = 'Steven';
const friend3 = 'Peter';

const friends = ['Michael', 'Steven', 'Peter'];
console.log(friends);  //(3) ['Michael', 'Steven', 'Peter']0: "Michael"1: "Steven"2:
"Jay"length: 3[[Prototype]]: Array(0)

const y = new Array(1991, 1984, 2008, 2020);

console.log(friends[0]);  //Michael
console.log(friends[2]);  //Peter

console.log(friends.length);  //3
console.log(friends[friends.length - 1]);  //Peter

friends[2] = 'Jay';
console.log(friends);  // (3) ['Michael', 'Steven', 'Jay']
// friends = ['Bob', 'Alice']

const firstName = 'Jonas';
const jonas = [firstName, 'Schmedtmann', 2037 - 1991, 'teacher', friends];
console.log(jonas);  // (5) ['Jonas', 'Schmedtmann', 46, 'teacher', Array(3)]
console.log(jonas.length); // 5

// Exercise
const calcAge = function (birthYeah) {
```

```
  return 2037 - birthYeah;
}
const years = [1990, 1967, 2002, 2010, 2018];

const age1 = calcAge(years[0]);
const age2 = calcAge(years[1]);
const age3 = calcAge(years[years.length - 1]);
console.log(age1, age2, age3);  // 47 70 19

const ages = [calcAge(years[0]), calcAge(years[1]), calcAge(years[years.length - 1])];
console.log(ages);  // (3) [47, 70, 19]
```

Assignment:

1. Create an array containing 4 population values of 4 countries of your choice. You may use the values you have been using previously. Store this array into a variable called 'populations'

2. Log to the console whether the array has 4 elements or not (true or false)

3. Create an array called 'percentages' containing the percentages of the world population for these 4 population values. Use the function 'percentageOfWorld1' that you created earlier to compute the 4 percentage values

Arrays methods

We will study about various array methods like push, pop , shift, unshift, indexof, includes

```
'use strict';
// Basic Array Operations (Methods)
const friends = ['Michael', 'Steven', 'Peter'];

// Add elements to the array
const newLength = friends.push('Jay');
console.log(friends);  // (4) ['Michael', 'Steven', 'Peter', 'Jay']
console.log(newLength);  // 4

friends.unshift('John');
console.log(friends);  // (5) ['John', 'Michael', 'Steven', 'Peter', 'Jay']

// Remove elements from the array
friends.pop(); // Last
const popped = friends.pop();
console.log(popped);  // Peter
console.log(friends);  // (3) ['John', 'Michael', 'Steven']

friends.shift(); // First
console.log(friends);  // (2) ['Michael', 'Steven']
```

```
// checking the index of the array if it is present it will give index if not it will give -1

console.log(friends.indexOf('Steven'));  //  1
console.log(friends.indexOf('Bob'));  // -1

friends.push(23);
// checking if the value is present in the array or not using includes
console.log(friends.includes('Steven'));  // true
console.log(friends.includes('Bob'));  // false
console.log(friends.includes(23));  // true

if (friends.includes('Steven')) {
  console.log('You have a friend called Steven');  // You have a friend called Steven
}
```

Converting Arrays to Strings
The JavaScript method toString() converts an array to a string of (comma separated) array values.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();
```

JOIN
The join() method also joins all array elements into a string.
It behaves just like toString(), but in addition you can specify the separator:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.join(" * ");
```

JavaScript Array length
The length property provides an easy way to append a new element to an array:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[fruits.length] = "Kiwi";
```

Merging (Concatenating) Arrays
The concat() method creates a new array by merging (concatenating) existing arrays:
The concat() method does not change the existing arrays. It always returns a new array.

```
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];
const arr3 = ["Robin", "Morgan"];

const myChildren = myGirls.concat(myBoys);
const myChildren = myGirls.concat(myBoys, arr3);    // it can take any number of arguments
const myChildren = myBoys.concat("Peter"); // it can also take string arguments
                                                    // Emil,Tobias,Linus,Peter
```

Splicing and Slicing Arrays
The splice() method adds new items to an array.

The slice() method slices out a piece of an array.

JavaScript Array splice()
The splice() method can be used to add new items to an array:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");   // Banana,Orange,Lemon,Kiwi,Apple,Mango
```

The first parameter (2) defines the position where new elements should be added (spliced in).
The second parameter (0) defines how many elements should be removed.
The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be added.

The splice() method returns an array with the deleted items:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let removed = fruits.splice(2, 2, "Lemon", "Kiwi");
console.log(removed)    // Apple,Mango
console.log(fruits)  // Banana,Orange,Lemon,Kiwi
```

Using splice() to Remove Elements
With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);  // Orange,Apple,Mango
```

The first parameter (0) defines the position where new elements should be added (spliced in).
The second parameter (1) defines how many elements should be removed.
The rest of the parameters are omitted. No new elements will be added.

JavaScript Array slice()
The slice() method slices out a piece of an array into a new array.
The slice() method creates a new array.
The slice() method does not remove any elements from the source array.
This example slices out a part of an array starting from array element 1 ("Orange"):

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1); // Orange,Lemon,Apple,Mango
```

This example slices out a part of an array starting from array element 3 ("Apple"):

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(3);
console.log(fruits)  // Banana,Orange,Lemon,Apple,Mango
console.log(citrus)  // Apple,Mango
```

The slice() method can take two arguments like slice(1, 3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3);
console.log(fruits)  // Banana,Orange,Lemon,Apple,Mango
console.log(citrus)  // Orange,Lemon
```

If the end argument is omitted, like in the first examples, the slice() method slices out the rest of the array.

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(2);
console.log(fruits)  // Banana,Orange,Lemon,Apple,Mango
console.log(citrus)  // Lemon,Apple,Mango
```

Assignment:

1. Create an array containing all the neighbouring countries of a country of your choice. Choose a country which has at least 2 or 3 neighbours. Store the array into a variable called 'neighbours'

2. At some point, a new country called 'Utopia' is created in the neighbourhood of your selected country. So add it to the end of the 'neighbours' array

3. Unfortunately, after some time, the new country is dissolved. So remove it from the end of the array

4. If the 'neighbours' array does not include the country ‘Germany’, log to the console: 'Probably not a central European country :D'

5. Change the name of one of your neighbouring countries. To do that, find the index of the country in the 'neighbours' array, and then use that index to change the array at that index position. For example, you can search for 'Sweden' in the array, and then replace it with 'Republic of Sweden'.

Coding Challenge #2

Steven is still building his tip calculator, using the same rules as before: Tip 15% of the bill if the bill value is between 50 and 300, and if the value is different, the tip is 20%.

Your tasks:

1. Write a function 'calcTip' that takes any bill value as an input and returns the corresponding tip, calculated based on the rules above (you can check out the code from first tip calculator challenge if you need to). Use the function type you like the most. Test the function using a bill value of 100

2. And now let's use arrays! So create an array 'bills' containing the test data below

3. Create an array 'tips' containing the tip value for each bill, calculated from the function you created before

4. Bonus: Create an array 'total' containing the total values, so the bill + tip

Test data:
125, 555 and 44

Hint:
Remember that an array needs a value in each position, and that value can actually be the returned value of a function! So you can just call a function as array values (so don't store the tip values in separate variables first, but right in the new array)

Introduction to objects

```
'use strict';
// Introduction to Objects
const jonasArray = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
  ['Michael', 'Peter', 'Steven']
];

const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 2037 - 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven']
};

console.log(jonasArray);  // {firstName: 'Jonas', lastName: 'Schmedtmann', age: 46, job:
'teacher', friends: ['Michael', 'Peter', 'Steven']}
console.log(jonas); // ['Jonas', 'Schmedtmann', 46, 'teacher', ['Michael', 'Peter',
'Steven']]
```

The first difference is that we can access arrays with numerical indexes but we need to access objects with named indexes as numerical indexes wont work

Assignment:

1. Create an object called 'myCountry' for a country of your choice, containing properties 'country', 'capital', 'language', 'population' and 'neighbours' (an array like we used in previous assignments)

Dot vs Bracket notation

We can access all the properties from the object using either dot notation or with square brackets

Accessing JavaScript Properties
The syntax for accessing the property of an object is:

```
objectName.property       // person.age
```
or

```
objectName["property"]    // person["age"]
```
or

```
objectName[expression]    // x = "age"; person[x]
```

```
'use strict';
// Dot vs. Bracket Notation
const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 2037 - 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven']
};
console.log(jonas);  // {firstName: 'Jonas', lastName: 'Schmedtmann', age: 46, job:
'teacher', friends: Array(3)}

console.log(jonas.lastName);  // Schmedtmann
console.log(jonas['lastName']);  // Schmedtmann

const nameKey = 'Name';
console.log(jonas['first' + nameKey]);  // Jonas
console.log(jonas['last' + nameKey]);  // Schmedtmann

// console.log(jonas.'last' + nameKey)   // this will not work as with dot notation we have
to only user real property name and not the expression

const interestedIn = prompt('What do you want to know about Jonas? Choose between firstName,
lastName, age, job, and friends');

if (jonas[interestedIn]) {
  console.log(jonas[interestedIn]);
} else {
  console.log('Wrong request! Choose between firstName, lastName, age, job, and friends');
}  // teacher

jonas.location = 'Portugal';
jonas['twitter'] = '@jonasschmedtman';
console.log(jonas);  // {firstName: 'Jonas', lastName: 'Schmedtmann', age: 46, job:
'teacher', friends: ['Michael', 'Peter', 'Steven'], location: "Portugal", twitter:
"@jonasschmedtman"}

// Challenge
// "Jonas has 3 friends, and his best friend is called Michael"
console.log(`${jonas.firstName} has ${jonas.friends.length} friends, and his best friend is
called ${jonas.friends[0]}`);  // Jonas has 3 friends, and his best friend is called Michael
```

Assignment:

1. Using the object from the previous assignment, log a string like this to the console: 'Finland has 6 million finnish-speaking people, 3 neighbouring countries and a capital called Helsinki.'

2. Increase the country's population by two million using dot notation, and then decrease it by two million using brackets notation.

Object Methods

```
'use strict';
// Object Methods

const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  birthYeah: 1991,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven'],
  hasDriversLicense: true,

  // calcAge: function (birthYeah) {
  //   return 2037 - birthYeah;
  // }

  // calcAge: function () {
  //   // console.log(this);
  //   return 2037 - this.birthYeah;
  // }

  calcAge: function () {
    this.age = 2037 - this.birthYeah;
    return this.age;
  },

  getSummary: function () {
    return `${this.firstName} is a ${this.calcAge()}-year old ${jonas.job}, and he has
${this.hasDriversLicense ? 'a' : 'no'} driver's license.`
  }
};

console.log(jonas.calcAge());  // 46

console.log(jonas.age);  // 46
console.log(jonas.age);  // 46
console.log(jonas.age);  // 46
```

```
// Challenge
// "Jonas is a 46-year old teacher, and he has a driver's license"
console.log(jonas.getSummary());  // Jonas is a 46-year old teacher, and he has a driver's
license.
```

Assignments
1. Add a method called 'describe' to the 'myCountry' object. This method will log a string to the console, similar to the string logged in the previous assignment, but this time using the 'this' keyword.

2. Call the 'describe' method

3. Add a method called 'checkIsland' to the 'myCountry' object. This method will set a new property on the object, called 'isIsland'. 'isIsland' will be true if there are no neighbouring countries, and false if there are. Use the ternary operator to set the property.

Coding Challenge #3

Let's go back to Mark and John comparing their BMIs! This time, let's use objects to implement the calculations! Remember: BMI = mass / height ** 2 = mass / (height * height) (mass in kg and height in meter)

Your tasks:
1. For each of them, create an object with properties for their full name, mass, and height (Mark Miller and John Smith)

2. Create a 'calcBMI' method on each object to calculate the BMI (the same method on both objects). Store the BMI value to a property, and also return it from the method

3. Log to the console who has the higher BMI, together with the full name and the respective BMI. Example: "John's BMI (28.3) is higher than Mark's (23.9)!"

Test data:
Marks weights 78 kg and is 1.69 m tall.
John weights 92 kg and is 1.95 m tall.

Loops

Loops can execute a block of code a number of times.

For Loop

```
for (let i = 0; i < cars.length; i++) {
  console.log(cars[i])
}
```

Assignment :

1. There are elections in your country! In a small town, there are only 50 voters. Use a for loop to simulate the 50 people voting, by logging a string like this to the console (for numbers 1 to 50): 'Voter number 1 is currently voting'

for loop on arrays, using break and continue

```
'use strict';

// Looping Arrays, Breaking and Continuing
```

```
const jonas = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
  ['Michael', 'Peter', 'Steven'],
  true
];
const types = [];

// console.log(jonas[0])
// console.log(jonas[1])
// ...
// console.log(jonas[4])
// jonas[5] does NOT exist

for (let i = 0; i < jonas.length; i++) {
  // Reading from jonas array
  console.log(jonas[i], typeof jonas[i]);

  // Filling types array
  // types[i] = typeof jonas[i];
  types.push(typeof jonas[i]);
}
// output of above for loop
// Jonas string
// Schmedtmann string
// 46 'number'
// teacher string
// (3) ['Michael', 'Peter', 'Steven'] 'object'
// true 'boolean'


console.log(types); //['string', 'string', 'number', 'string', 'object', 'boolean']

const years = [1991, 2007, 1969, 2020];
const ages = [];

for (let i = 0; i < years.length; i++) {
  ages.push(2037 - years[i]);
}
console.log(ages); // (4) [46, 30, 68, 17]

// continue and break
console.log('--- ONLY STRINGS ---')  //  --- ONLY STRINGS ---
for (let i = 0; i < jonas.length; i++) {
  if (typeof jonas[i] !== 'string') continue;
```

```
  console.log(jonas[i], typeof jonas[i]);
}
// ouput of above for loop
//  Jonas string
//  Schmedtmann string
//  teacher string

console.log('--- BREAK WITH NUMBER ---')
for (let i = 0; i < jonas.length; i++) {
  if (typeof jonas[i] === 'number') break;

  console.log(jonas[i], typeof jonas[i]);
}
// ouput of above for loop
//  Jonas string
//  Schmedtmann string
```

Assignment:

1. Let's bring back the 'populations' array from a previous assignment

2. Use a for loop to compute an array called 'percentages2' containing the percentages of the world population for the 4 population values. Use the function 'percentageOfWorld1' that you created earlier

3. Confirm that 'percentages2' contains exactly the same values as the 'percentages' array that we created manually in the previous assignment, and reflect on how much better this solution is

Loop inside another loop and looping backwards

```
'use strict';
// Looping Backwards and Loops in Loops
const jonas = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
  ['Michael', 'Peter', 'Steven'],
  true
];

// 0, 1, ..., 4
// 4, 3, ..., 0

for (let i = jonas.length - 1; i >= 0; i--) {
  console.log(i, jonas[i]);
}
// output of above for loop
// 5 true
// 4 (3) ['Michael', 'Peter', 'Steven']
```

```
// 3 'teacher'
// 2 46
// 1 'Schmedtmann'
// 0 'Jonas'

for (let exercise = 1; exercise < 4; exercise++) {
  console.log(`-------- Starting exercise ${exercise}`);

  for (let rep = 1; rep < 6; rep++) {
    console.log(`Exercise ${exercise}: Lifting weight repetition ${rep} 🏋`);
  }
}
// output of above for loop
// script.js:20 -------- Starting exercise 1
// script.js:23 Exercise 1: Lifting weight repetition 1 🏋
// script.js:23 Exercise 1: Lifting weight repetition 2 🏋
// script.js:23 Exercise 1: Lifting weight repetition 3 🏋
// script.js:23 Exercise 1: Lifting weight repetition 4 🏋
// script.js:23 Exercise 1: Lifting weight repetition 5 🏋
// script.js:20 -------- Starting exercise 2
// script.js:23 Exercise 2: Lifting weight repetition 1 🏋
// script.js:23 Exercise 2: Lifting weight repetition 2 🏋
// script.js:23 Exercise 2: Lifting weight repetition 3 🏋
// script.js:23 Exercise 2: Lifting weight repetition 4 🏋
// script.js:23 Exercise 2: Lifting weight repetition 5 🏋
// script.js:20 -------- Starting exercise 3
// script.js:23 Exercise 3: Lifting weight repetition 1 🏋
// script.js:23 Exercise 3: Lifting weight repetition 2 🏋
// script.js:23 Exercise 3: Lifting weight repetition 3 🏋
// script.js:23 Exercise 3: Lifting weight repetition 4 🏋
// script.js:23 Exercise 3: Lifting weight repetition 5 🏋
```

Assignment:

1. Store this array of arrays into a variable called 'listOfNeighbours' [['Canada', 'Mexico'], ['Spain'], ['Norway', 'Sweden', 'Russia']];

2. Log only the neighbouring countries to the console, one by one, not the entire arrays. Log a string like 'Neighbour: Canada' for each country

3. You will need a loop inside a loop for this. This is actually a bit tricky, so don't worry if it's too difficult for you! But you can still try to figure this out anyway 🙂

While loop

We will use while loop when we don't know in advanced that how many times to loop

```
'use strict';

// The while Loop
for (let rep = 1; rep <= 10; rep++) {
  console.log(`Lifting weights repetition ${rep} 🏋`);
}
// output of above for loop
// Lifting weights repetition 1 🏋
// Lifting weights repetition 2 🏋
// Lifting weights repetition 3 🏋
// Lifting weights repetition 4 🏋
// Lifting weights repetition 5 🏋
// Lifting weights repetition 6 🏋
// Lifting weights repetition 7 🏋
// Lifting weights repetition 8 🏋
// Lifting weights repetition 9 🏋
// Lifting weights repetition 10 🏋

let rep = 1;
while (rep <= 10) {
  // console.log(`WHILE: Lifting weights repetition ${rep} 🏋`);
  rep++;
}

let dice = Math.trunc(Math.random() * 6) + 1;

while (dice !== 6) {
  console.log(`You rolled a ${dice}`);
  dice = Math.trunc(Math.random() * 6) + 1;
  if (dice === 6) console.log('Loop is about to end...');
}
// output of above for loop
// script.js:17 You rolled a 2
// script.js:19 Loop is about to end...
```

Assignment:

1. Recreate the challenge from the lecture 'Looping Arrays, Breaking and Continuing', but this time using a while loop (call the array 'percentages3')

2. Reflect on what solution you like better for this task: the for loop or the while loop?

Coding Challenge #4

Let's improve Steven's tip calculator even more, this time using loops!

Your tasks:

1. Create an array 'bills' containing all 10 test bill values

2. Create empty arrays for the tips and the totals ('tips' and 'totals')

3. Use the 'calcTip' function we wrote before (no need to repeat) to calculate tips and total values (bill + tip) for every bill value in the bills array. Use a for loop to perform the 10 calculations!

Test data: 22, 295, 176, 440, 37, 105, 10, 1100, 86 and 52

Hints: Call ‘calcTip ‘in the loop and use the push method to add values to the tips and totals arrays

Bonus:

4. Bonus: Write a function 'calcAverage' which takes an array called 'arr' as an argument. This function calculates the average of all numbers in the given array. This is a difficult challenge (we haven't done this before)! Here is how to solve it:

4.1. First, you will need to add up all values in the array. To do the addition, start by creating a variable 'sum' that starts at 0. Then loop over the array using a for loop. In each iteration, add the current value to the 'sum' variable. This way, by the end of the loop, you have all values added together

4.2. To calculate the average, divide the sum you calculated before by the length of the array (because that's the number of elements)

4.3. Call the function with the 'totals' array

Developer skill and editor setup

Try to learn debugger in chrome developer tool
And try to solve problems by googling, the more problem you solve on your own the more interested you will be

So now let's solve a challenge \

Coding Challenge #1

Given an array of forecasted maximum temperatures, the thermometer displays a string with the given temperatures. Example: [17, 21, 23] will print "... 17ºC in 1 days ... 21ºC in 2 days ... 23ºC in 3 days ..."

Your tasks:

1. Create a function 'printForecast' which takes in an array 'arr' and logs a string like the above to the console. Try it with both test datasets.

2. Use the problem-solving framework: Understand the problem and break it up into sub-problems!

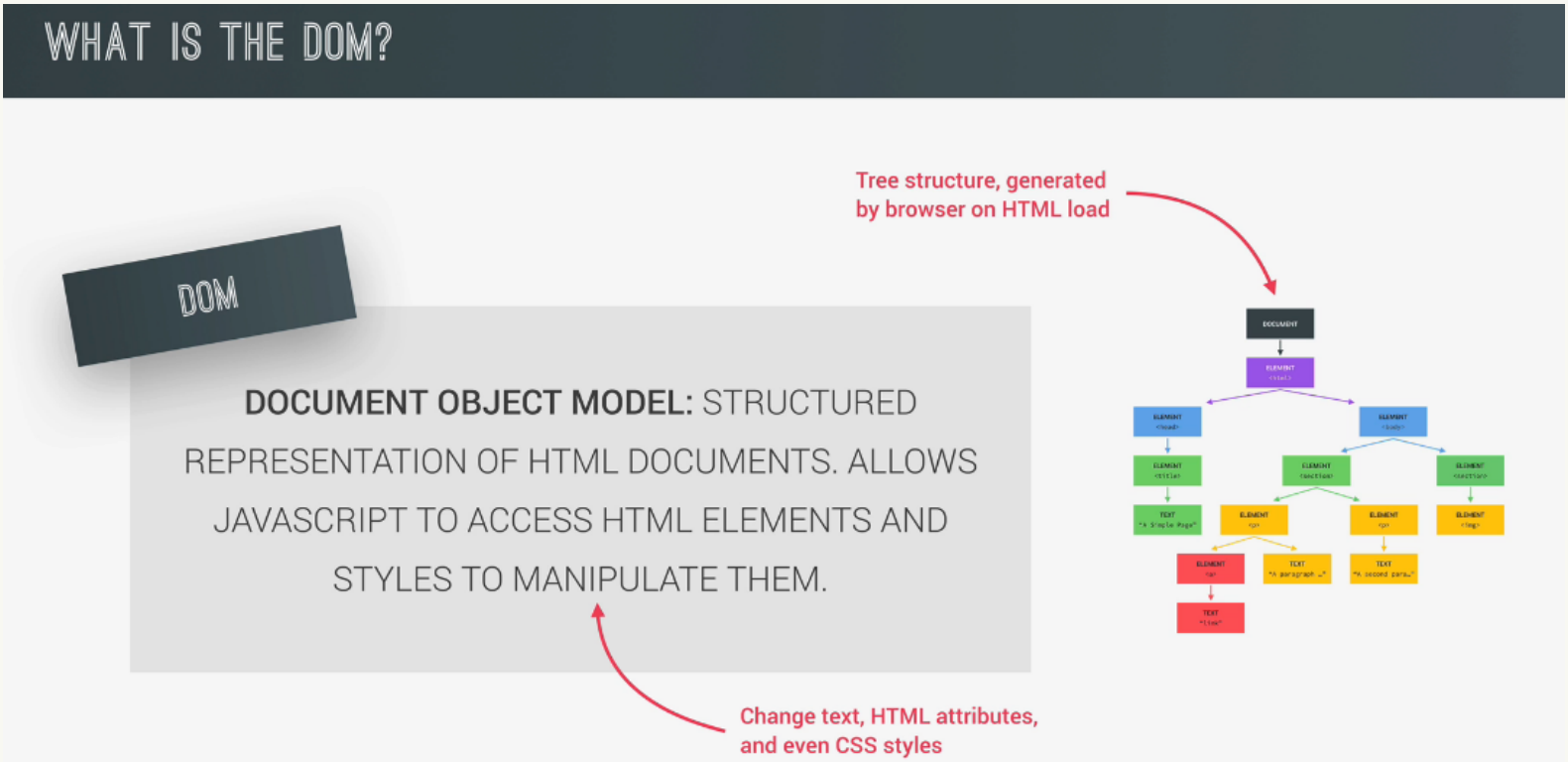
Test data:

§ Data 1: [17, 21, 23]

§ Data 2: [12, 5, -5, 0, 4]

Javascript DOM and events

What is DOM



The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

DOM tree

The backbone of an HTML document is tags.

According to the Document Object Model (DOM), every HTML tag is an object. Nested tags are “children” of the enclosing one. The text inside a tag is an object as well.

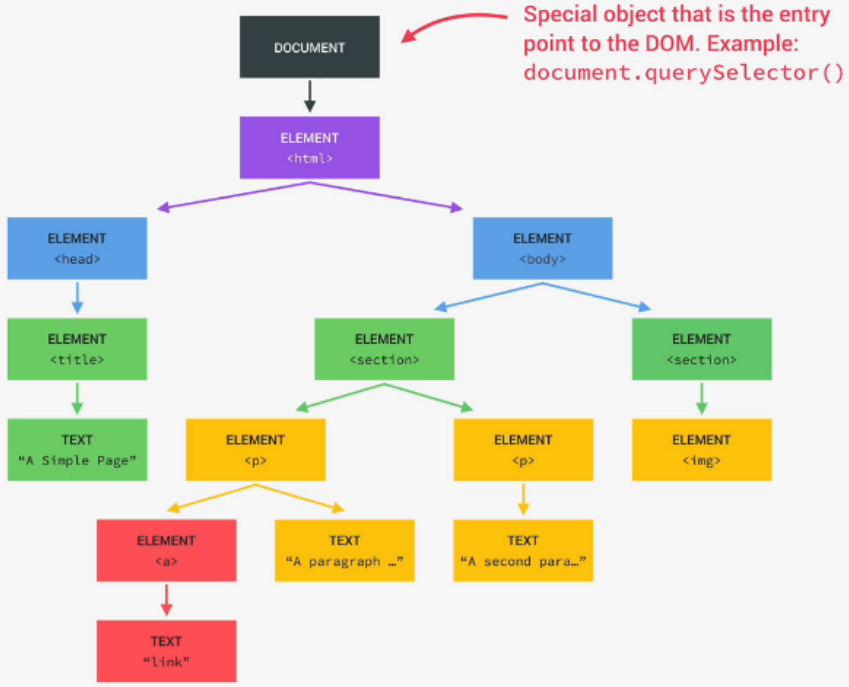
All these objects are accessible using JavaScript, and we can use them to modify the page.

Every tree node is an object.

Tags are element nodes (or just elements) and form the tree structure: <html> is at the root, then <head> and <body> are its children, etc.

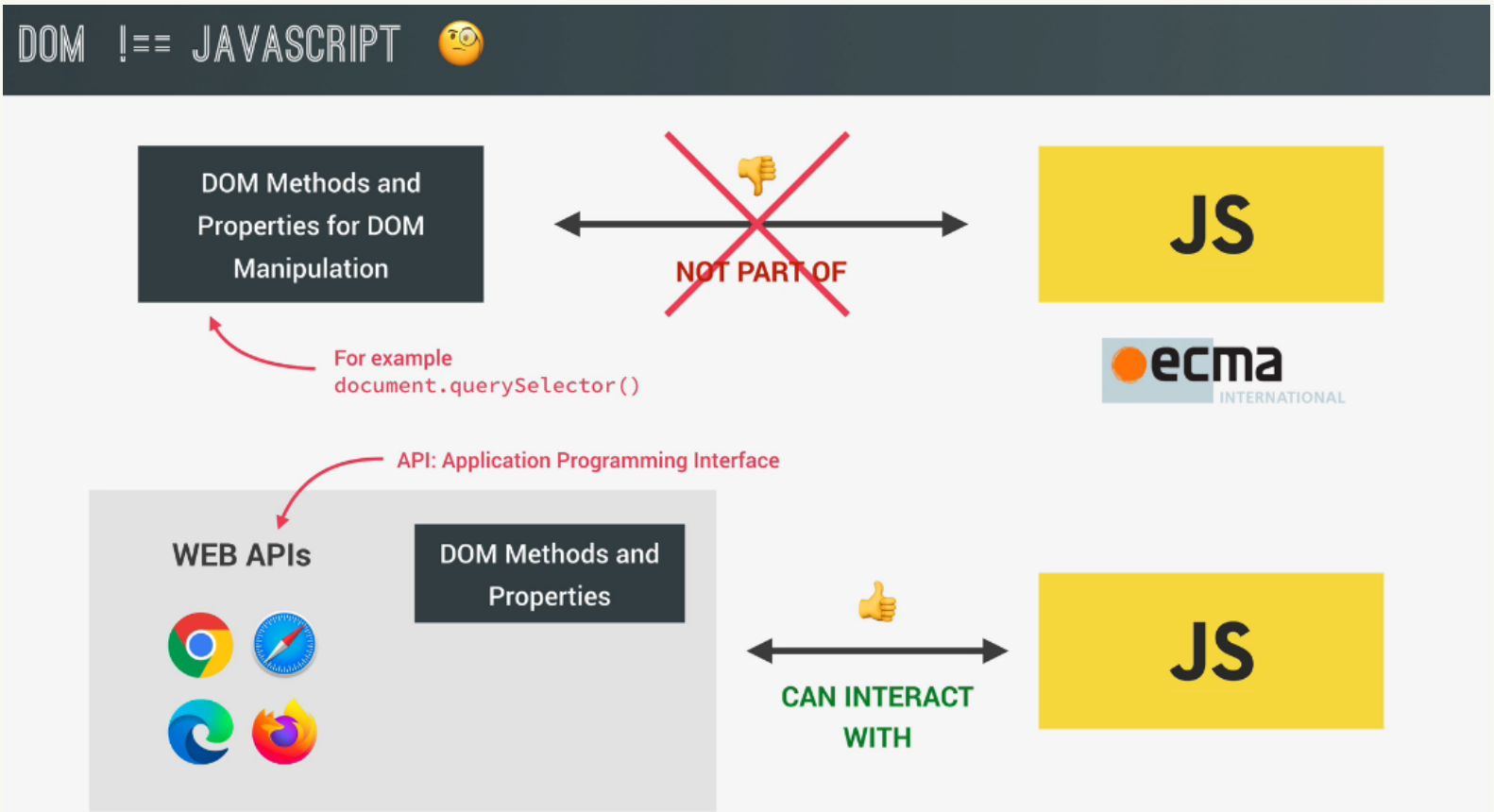
The text inside elements forms text nodes, labelled as #text. A text node contains only a string. It may not have children and is always a leaf of the tree.

```html
<html>
  <head>
    <title>A Simple Page</title>
  </head>
  <body>
    <section>
      <p>A paragraph with a <a>link</a></p>
      <p>A second paragraph</p>
    </section>
    <section>
      <img src="dom.png" alt="The DOM" />
    </section>
  </body>
</html>
```



DOM is not javascript but it is WEB APIs

Dom and methods are not a part of JavaScript there a part of Web API
So the web API APIs are like libraries that browsers implement and that we can access from our JavaScript code.



DOM APIs

Finding HTML Elements

getElementById Method
The most common way to access an HTML element is to use the id of the element.

This example finds the element with id="intro":

```
const element = document.getElementById("intro");
```

If the element is found, the method will return the element as an object (in element).

If the element is not found, element will contain null.

Finding HTML Elements by Tag Name
This example finds all <p> elements:

```
const element = document.getElementsByTagName("p");
```

Finding HTML Elements by Class Name

If you want to find all HTML elements with the same class name, use getElementsByClassName().
This example returns a list of all elements with class="intro".

```
const x = document.getElementsByClassName("intro");
```

Finding HTML Elements by CSS Selectors

If you want to find all HTML elements that match a specified CSS selector (id, class names, types, attributes, values of attributes, etc), use the querySelectorAll() method.
This example returns a list of all <p> elements with class="intro".

```
const x = document.querySelectorAll("p.intro");
```

Finding HTML Elements by HTML Object Collections

This example finds the form element with id="frm1", in the forms collection, and displays all element values:

```
const x = document.forms["frm1"];
let text = "";
for (let i = 0; i < x.length; i++) {
  text += x.elements[i].value + "<br>";
}
document.getElementById("demo").innerHTML = text;
```

Changing HTML

innerHTML property

The easiest way to modify the content of an HTML element is by using the innerHTML property.

To change the content of an HTML element, use this syntax:
```
document.getElementById("p1").innerHTML = "New text!";
```

Changing the Value of an Attribute

To change the value of an HTML attribute, use this syntax:
```
document.getElementById(id).attribute = new value
document.getElementById("myImage").src = "landscape.jpg";
```

document.write()

In JavaScript, document.write() can be used to write directly to the HTML output stream:
```
document.write(Date());
```

```
Never use document.write() after the document is loaded. It will overwrite the document.
```

JavaScript Form Validation

HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

```
<script>
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
</script>

<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>
```

JavaScript Can Validate Numeric Input

JavaScript is often used to validate numeric input:

```
<p>Please input a number between 1 and 10:</p>

<input id="numb">
```

```
<button type="button" onclick="myFunction()">Submit</button>

<p id="demo"></p>

<script>
function myFunction() {
  // Get the value of the input field with id="numb"
  let x = document.getElementById("numb").value;
  // If x is Not a Number or less than one or greater than 10
  let text;
  if (isNaN(x) || x < 1 || x > 10) {
    text = "Input not valid";
  } else {
    text = "Input OK";
  }
  document.getElementById("demo").innerHTML = text;
}
</script>
```

HTML Constraint Validation

HTML Input Attributes

Attribute	Description
disabled	Specifies that the input element should be disabled
max	Specifies the maximum value of an input element
min	Specifies the minimum value of an input element
pattern	Specifies the value pattern of an input element
required	Specifies that the input field requires an element
type	Specifies the type of an input element

CSS Pseudo Selectors

Selector	Description
:disabled	Selects input elements with the "disabled" attribute specified
:invalid	Selects input elements with invalid values
:optional	Selects input elements with no "required" attribute specified
:required	Selects input elements with the "required" attribute specified
:valid	Selects input elements with valid values

HTML DOM - Changing CSS

Changing HTML Style
To change the style of an HTML element, use this syntax:

```
document.getElementById(id).style.property = new style
```

```
document.getElementById("p2").style.color = "blue";
```

HTML DOM Events

Reacting to Events
A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.

To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:
`onclick=JavaScript`

Examples of HTML events:

When a user clicks the mouse
When a web page has loaded
When an image has been loaded
When the mouse moves over an element
When an input field is changed
When an HTML form is submitted
When a user strokes a key

1.
```
<h1 onclick="this.innerHTML = 'Ooops!'">Click on this text!</h1>
```

2.
```
<h2 onclick="changeText(this)">Click on this text!</h2>

<script>
function changeText(id) {
  id.innerHTML = "Ooops!";
}
</script>
```

3.
```
<button onclick="displayDate()">The time is?</button>

<script>
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>
```

Assign Events Using the HTML DOM
The HTML DOM allows you to assign events to HTML elements using JavaScript:

```
<button id="myBtn">Try it</button>

<p id="demo"></p>

<script>
document.getElementById("myBtn").onclick = displayDate;
```

```
function displayDate() {
  document.getElementById("demo").innerHTML = Date();
}
</script>
```

The onload and onunload Events

The onload and onunload events are triggered when the user enters or leaves the page.

The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

The onload and onunload events can be used to deal with cookies.

Example

```
<body onload="checkCookies()">
```

The onchange Event

The onchange event is often used in combination with validation of input fields.

Below is an example of how to use the onchange. The upperCase() function will be called when a user changes the content of an input field.

Example

```
<input type="text" id="fname" onchange="upperCase()">
```

The onmouseover and onmouseout Events

The onmouseover and onmouseout events can be used to trigger a function when the user mouses over, or out of, an HTML element:

```
<div onmouseover="mOver(this)" onmouseout="mOut(this)"
style="background-color:red">
Mouse Over Me</div>

<script>
function mOver(obj) {
  obj.innerHTML = "Thank You"
}

function mOut(obj) {
  obj.innerHTML = "Mouse Over Me"
}
</script>
```

The onmousedown, onmouseup and onclick Events

The onmousedown, onmouseup, and onclick events are all parts of a mouse-click. First when a mouse-button is clicked, the onmousedown event is triggered, then, when the mouse-button is released, the onmouseup event is triggered, finally, when the mouse-click is completed, the onclick event is triggered.

```
<div onmousedown="mDown(this)" onmouseup="mUp(this)"
style="background-color:red">
Click Me</div>
```

```
<script>
function mDown(obj) {
  obj.style.backgroundColor = "#1ec5e5";
  obj.innerHTML = "Release Me";
}

function mUp(obj) {
  obj.style.backgroundColor="#D94A38";
  obj.innerHTML="Thank You";
}
</script>
```

Events

Event	Description
abort	The event occurs when the loading of a media is aborted
afterprint	The event occurs when a page has started printing, or if the print dialogue box has been closed
animationend	The event occurs when a CSS animation has completed
animationiteration	The event occurs when a CSS animation is repeated
animationstart	The event occurs when a CSS animation has started
beforeprint	The event occurs when a page is about to be printed
beforeunload	The event occurs before the document is about to be unloaded
blur	The event occurs when an element loses focus
canplay	The event occurs when the browser can start playing the media (when it has buffered enough to begin)
canplaythrough	The event occurs when the browser can play through the media without stopping for buffering
change	The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>)
click	The event occurs when the user clicks on an element
contextmenu	The event occurs when the user right-clicks on an element to open a context menu
copy	The event occurs when the user copies the content of an element
cut	The event occurs when the user cuts the content of an element
dblclick	The event occurs when the user double-clicks on an element
drag	The event occurs when an element is being dragged
dragend	The event occurs when the user has finished dragging an element
dragenter	The event occurs when the dragged element enters the drop target
dragleave	The event occurs when the dragged element leaves the drop target
dragover	The event occurs when the dragged element is over the drop target
dragstart	The event occurs when the user starts to drag an element
drop	The event occurs when the dragged element is dropped on the drop target
durationchange	The event occurs when the duration of the media is changed
ended	The event occurs when the media has reach the end (useful for messages like "thanks for listening")
error	The event occurs when an error occurs while loading an external file
focus	The event occurs when an element gets focus
focusin	The event occurs when an element is about to get focus
focusout	The event occurs when an element is about to lose focus
fullscreenchange	The event occurs when an element is displayed in fullscreen mode
fullscreenerror	The event occurs when an element can not be displayed in fullscreen mode
hashchange	The event occurs when there has been changes to the anchor part of a URL
input	The event occurs when an element gets user input
invalid	The event occurs when an element is invalid
keydown	The event occurs when the user is pressing a key
keypress	The event occurs when the user presses a key
keyup	The event occurs when the user releases a key
load	The event occurs when an object has loaded
loadeddata	The event occurs when media data is loaded
loadedmetadata	The event occurs when meta data (like dimensions and duration) are loaded
loadstart	The event occurs when the browser starts looking for the specified media

message	The event occurs when a message is received through the event source
mousedown	The event occurs when the user presses a mouse button over an element
mouseenter	The event occurs when the pointer is moved onto an element
mouseleave	The event occurs when the pointer is moved out of an element
mousemove	The event occurs when the pointer is moving while it is over an element
mouseover	The event occurs when the pointer is moved onto an element, or onto one of its children
mouseout	The event occurs when a user moves the mouse pointer out of an element, or out of one of its children
mouseup	The event occurs when a user releases a mouse button over an element
mousewheel	Deprecated. Use the wheel event instead
offline	The event occurs when the browser starts to work offline
online	The event occurs when the browser starts to work online
open	The event occurs when a connection with the event source is opened
pagehide	The event occurs when the user navigates away from a webpage
pageshow	The event occurs when the user navigates to a webpage
paste	The event occurs when the user pastes some content in an element
pause	The event occurs when the media is paused either by the user or programmatically
play	The event occurs when the media has been started or is no longer paused
playing	The event occurs when the media is playing after having been paused or stopped for buffering
popstate	The event occurs when the window's history changes
progress	The event occurs when the browser is in the process of getting the media data (downloading the media)
ratechange	The event occurs when the playing speed of the media is changed
resize	The event occurs when the document view is resized
reset	The event occurs when a form is reset
scroll	The event occurs when an element's scrollbar is being scrolled
search	The event occurs when the user writes something in a search field (for <input="search">)
seeked	The event occurs when the user is finished moving/skipping to a new position in the media
seeking	The event occurs when the user starts moving/skipping to a new position in the media
select	The event occurs after the user selects some text (for <input> and <textarea>)
show	The event occurs when a <menu> element is shown as a context menu
stalled	The event occurs when the browser is trying to get media data, but data is not available
storage	The event occurs when a Web Storage area is updated
submit	The event occurs when a form is submitted
suspend	The event occurs when the browser is intentionally not getting media data
timeupdate	The event occurs when the playing position has changed (like when the user fast forwards to a different point in the media)
toggle	The event occurs when the user opens or closes the <details> element
touchcancel	The event occurs when the touch is interrupted
touchend	The event occurs when a finger is removed from a touch screen
touchmove	The event occurs when a finger is dragged across the screen
touchstart	The event occurs when a finger is placed on a touch screen
transitionend	The event occurs when a CSS transition has completed
unload	The event occurs once a page has unloaded (for <body>)
volumechange	The event occurs when the volume of the media has changed (includes setting the volume to "mute")
waiting	The event occurs when the media has paused but is expected to resume (like when the media pauses to buffer more data)
wheel	The event occurs when the mouse wheel rolls up or down over an element

HTML DOM Event Properties and Methods

Property/Method	Description
altKey	Returns whether the "ALT" key was pressed when the mouse event was triggered
altKey	Returns whether the "ALT" key was pressed when the key event was triggered
animationName	Returns the name of the animation
bubbles	Returns whether or not a specific event is a bubbling event
button	Returns which mouse button was pressed when the mouse event was triggered
buttons	Returns which mouse buttons were pressed when the mouse event was triggered
cancelable	Returns whether or not an event can have its default action prevented
charCode	Returns the Unicode character code of the key that triggered the onkeypress event
changeTouches	Returns a list of all the touch objects whose state changed between the previous touch and this touch
clientX	Returns the horizontal coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
clientY	Returns the vertical coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
clipboardData	Returns an object containing the data affected by the clipboard operation
code	Returns the code of the key that triggered the event
composed	Returns whether the event is composed or not
ctrlKey	Returns whether the "CTRL" key was pressed when the mouse event was triggered

ctrlKey Returns whether the "CTRL" key was pressed when the key event was triggered
currentTarget Returns the element whose event listeners triggered the event
data Returns the inserted characters
dataTransfer Returns an object containing the data being dragged/dropped, or inserted/deleted
defaultPrevented Returns whether or not the preventDefault() method was called for the event
deltaX Returns the horizontal scroll amount of a mouse wheel (x-axis)
deltaY Returns the vertical scroll amount of a mouse wheel (y-axis)
deltaZ Returns the scroll amount of a mouse wheel for the z-axis
deltaMode Returns a number that represents the unit of measurements for delta values (pixels, lines or pages)
detail Returns a number that indicates how many times the mouse was clicked
elapsedTime Returns the number of seconds an animation has been running
elapsedTime Returns the number of seconds a transition has been running
eventPhase Returns which phase of the event flow is currently being evaluated
getTargetRanges() Returns an array containing target ranges that will be affected by the insertion/deletion
getModifierState() Returns an array containing target ranges that will be affected by the insertion/deletion
inputType Returns the type of the change (i.e "inserting" or "deleting")
isComposing Returns whether the state of the event is composing or not
isTrusted Returns whether or not an event is trusted
key Returns the key value of the key represented by the event
key Returns the key of the changed storage item
keyCode Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event
location Returns the location of a key on the keyboard or device
lengthComputable Returns whether the length of the progress can be computable or not
loaded Returns how much work has been loaded
metaKey Returns whether the "META" key was pressed when an event was triggered
metaKey Returns whether the "meta" key was pressed when the key event was triggered
MovementX Returns the horizontal coordinate of the mouse pointer relative to the position of the last mousemove event
MovementY Returns the vertical coordinate of the mouse pointer relative to the position of the last mousemove event
newValue Returns the new value of the changed storage item
newURL Returns the URL of the document, after the hash has been changed
offsetX Returns the horizontal coordinate of the mouse pointer relative to the position of the edge of the target element
offsetY Returns the vertical coordinate of the mouse pointer relative to the position of the edge of the target element
oldValue Returns the old value of the changed storage item
oldURL Returns the URL of the document, before the hash was changed
onemptied The event occurs when something bad happens and the media file is suddenly unavailable (like unexpectedly disconnects)
pageX Returns the horizontal coordinate of the mouse pointer, relative to the document, when the mouse event was triggered
pageY Returns the vertical coordinate of the mouse pointer, relative to the document, when the mouse event was triggered
persisted Returns whether the webpage was cached by the browser
preventDefault() Cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur
propertyName Returns the name of the CSS property associated with the animation or transition
pseudoElement Returns the name of the pseudo-element of the animation or transition
region
relatedTarget Returns the element related to the element that triggered the mouse event
relatedTarget Returns the element related to the element that triggered the event
repeat Returns whether a key is being hold down repeatedly, or not
screenX Returns the horizontal coordinate of the mouse pointer, relative to the screen, when an event was triggered
screenY Returns the vertical coordinate of the mouse pointer, relative to the screen, when an event was triggered
shiftKey Returns whether the "SHIFT" key was pressed when an event was triggered
shiftKey Returns whether the "SHIFT" key was pressed when the key event was triggered
state Returns an object containing a copy of the history entries
stopImmediatePropagation()
stopPropagation() Prevents further propagation of an event during event flow
storageArea Returns an object representing the affected storage object
target Returns the element that triggered the event
targetTouches Returns a list of all the touch objects that are in contact with the surface and where the touchstart event occured on the same target element as the current target element
timeStamp Returns the time (in milliseconds relative to the epoch) at which the event was created
total Returns the total amount of work that will be loaded
touches Returns a list of all the touch objects that are currently in contact with the surface
transitionend The event occurs when a CSS transition has completed
type Returns the name of the event

url Returns the URL of the changed item's document
which Returns which mouse button was pressed when the mouse event was triggered
which Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event
view Returns a reference to the Window object where the event occurred

HTML DOM EventListener

The addEventListener() method attaches an event handler to the specified element.

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.

You can add many event handlers to one element.

You can add many event handlers of the same type to one element, i.e two "click" events.

You can add event listeners to any DOM object, not only HTML elements. i.e the window object.

The addEventListener() method makes it easier to control how the event reacts to bubbling.

When using the addEventListener() method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.

You can easily remove an event listener by using the removeEventListener() method.

Syntax

```
element.addEventListener(event, function, useCapture);
```

he first parameter is the type of the event (like "click" or "mousedown" or any other HTML DOM Event. **see above list**)

The second parameter is the function we want to call when the event occurs.

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".

Add an Event Handler to an Element

1.

```
<button id="myBtn">Try it</button>

<script>
document.getElementById("myBtn").addEventListener("click", function() {
  alert("Hello World!");
});
</script>
```

2.

```
<button id="myBtn">Try it</button>

<script>
document.getElementById("myBtn").addEventListener("click", myFunction);

function myFunction() {
```

```
  alert ("Hello World!");
}
</script>
```

Add Many Event Handlers to the Same Element
The addEventListener() method allows you to add many events to the same element, without overwriting existing events:

```
<button id="myBtn">Try it</button>

<p id="demo"></p>

<script>
var x = document.getElementById("myBtn");
x.addEventListener("mouseover", myFunction);
x.addEventListener("click", mySecondFunction);
x.addEventListener("mouseout", myThirdFunction);

function myFunction() {
  document.getElementById("demo").innerHTML += "Moused over!<br>";
}

function mySecondFunction() {
  document.getElementById("demo").innerHTML += "Clicked!<br>";
}

function myThirdFunction() {
  document.getElementById("demo").innerHTML += "Moused out!<br>";
}
</script>
```

Add an Event Handler to the window Object

The addEventListener() method allows you to add event listeners on any HTML DOM object such as HTML elements, the HTML document, the window object, or other objects that support events, like the xmlHttpRequest object.

```
<p id="demo"></p>

<script>
window.addEventListener("resize", function(){
  document.getElementById("demo").innerHTML = Math.random();
});
</script>
```

Passing Parameters
When passing parameter values, use an "anonymous function" that calls the specified function with the parameters:

```
<button id="myBtn">Try it</button>
```

```
<p id="demo"></p>

<script>
let p1 = 5;
let p2 = 7;
document.getElementById("myBtn").addEventListener("click", function() {
  myFunction(p1, p2);
});

function myFunction(a, b) {
  document.getElementById("demo").innerHTML = a * b;
}
</script>
```

Event Bubbling or Event Capturing?
There are two ways of event propagation in the HTML DOM, bubbling and capturing.

Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In bubbling the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the element's click event.

In capturing the outer most element's event is handled first and then the inner: the element's click event will be handled first, then the <p> element's click event.

With the addEventListener() method you can specify the propagation type by using the "useCapture" parameter:

```
addEventListener(event, function, useCapture);
```

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

```
<!DOCTYPE html>
<html>
<head>
<style>
#myDiv1, #myDiv2 {
  background-color: coral;
  padding: 50px;
}

#myP1, #myP2 {
  background-color: white;
  font-size: 20px;
  border: 1px solid;
  padding: 20px;
}
</style>
<meta content="text/html; charset=utf-8" http-equiv="Content-Type">
</head>
```

```
<body>

<h2>JavaScript addEventListener()</h2>

<div id="myDiv1">
  <h2>Bubbling:</h2>
  <p id="myP1">Click me!</p>
</div><br>

<div id="myDiv2">
  <h2>Capturing:</h2>
  <p id="myP2">Click me!</p>
</div>

<script>
document.getElementById("myP1").addEventListener("click", function() {
  alert("You clicked the white element!");
}, false);

document.getElementById("myDiv1").addEventListener("click", function() {
  alert("You clicked the orange element!");
}, false);

document.getElementById("myP2").addEventListener("click", function() {
  alert("You clicked the white element!");
}, true);

document.getElementById("myDiv2").addEventListener("click", function() {
  alert("You clicked the orange element!");
}, true);
</script>

</body>
</html>
```

The removeEventListener() method

The removeEventListener() method removes event handlers that have been attached with the addEventListener() method:

```
<div id="myDIV">
    <p>This div element has an onmousemove event handler that displays a random number every time you move your mouse inside this orange field.</p>
    <p>Click the button to remove the div's event handler.</p>
    <button onclick="removeHandler()" id="myBtn">Remove</button>
  </div>

  <p id="demo"></p>
```

```
<script>
document.getElementById("myDIV").addEventListener("mousemove", myFunction);

function myFunction() {
  document.getElementById("demo").innerHTML = Math.random();
}

function removeHandler() {
  document.getElementById("myDIV").removeEventListener("mousemove", myFunction);
}
</script>
```

Creating New HTML Elements (Nodes)

To add a new element to the HTML DOM, you must create the element (element node) first, and then append it to an existing element.

```
<div id="div1">
    <p id="p1">This is a paragraph.</p>
    <p id="p2">This is another paragraph.</p>
</div>

    <script>
    const para = document.createElement("p");
    const node = document.createTextNode("This is new.");
    para.appendChild(node);
    const element = document.getElementById("div1");
    element.appendChild(para);
    </script>
```

Creating new HTML Elements - insertBefore()
The appendChild() method in the previous example, appended the new element as the last child of the parent.

If you don't want that you can use the insertBefore() method:

```
<div id="div1">
    <p id="p1">This is a paragraph.</p>
    <p id="p2">This is another paragraph.</p>
    </div>

    <script>
    const para = document.createElement("p");
    const node = document.createTextNode("This is new.");
    para.appendChild(node);

    const element = document.getElementById("div1");
    const child = document.getElementById("p1");
    element.insertBefore(para,child);
    </script>
```

Removing Existing HTML Elements

To remove an HTML element, use the remove() method:

```
<div>
    <p id="p1">This is a paragraph.</p>
    <p id="p2">This is another paragraph.</p>
    </div>

    <button onclick="myFunction()">Remove Element</button>

    <script>
    function myFunction() {
    document.getElementById("p1").remove();
    }
    </script>
```

The remove() method does not work in older browsers, see the example below on how to use removeChild() instead.

Removing a Child Node

For browsers that does not support the remove() method, you have to find the parent node to remove an element:

```
<div id="div1">
    <p id="p1">This is a paragraph.</p>
    <p id="p2">This is another paragraph.</p>
    </div>

    <script>
    const parent = document.getElementById("div1");
    const child = document.getElementById("p1");
    parent.removeChild(child);

    //other method
    // const child = document.getElementById("p1");
    // child.parentNode.removeChild(child);
    </script>
```

Replacing HTML Elements

To replace an element to the HTML DOM, use the replaceChild() method:

```
<div id="div1">
    <p id="p1">This is a paragraph.</p>
    <p id="p2">This is a paragraph.</p>
    </div>

    <script>
    const parent = document.getElementById("div1");
    const child = document.getElementById("p1");
    const para = document.createElement("p");
    const node = document.createTextNode("This is new.");
    para.appendChild(node);
```

```
parent.replaceChild(para,child);
</script>
```

The HTMLCollection Object

The getElementsByTagName() method returns an HTMLCollection object.

An HTMLCollection object is an array-like list (collection) of HTML elements.

The following code selects all <p> elements in a document:

```
const myCollection = document.getElementsByTagName("p");
```

The elements in the collection can be accessed by an index number.

To access the second <p> element you can write:

```
myCollection[1]
```

HTML HTMLCollection Length

The length property defines the number of elements in an HTMLCollection:

```
myCollection.length
```

The length property is useful when you want to loop through the elements in a collection:

Change the text color of all <p> elements:

```
const myCollection = document.getElementsByTagName("p");
for (let i = 0; i < myCollection.length; i++) {
  myCollection[i].style.color = "red";
}
```

An HTMLCollection is NOT an array!

An HTMLCollection may look like an array, but it is not.
You can loop through the list and refer to the elements with a number (just like an array).
However, you cannot use array methods like valueOf(), pop(), push(), or join() on an HTMLCollection.

HTML DOM NodeList Object

A NodeList object is a list (collection) of nodes extracted from a document.

A NodeList object is almost the same as an HTMLCollection object.

Some (older) browsers return a NodeList object instead of an HTMLCollection for methods like getElementsByClassName().

All browsers return a NodeList object for the property childNodes.

Most browsers return a NodeList object for the method querySelectorAll().

The following code selects all <p> nodes in a document:

```
const myNodeList = document.querySelectorAll("p");
```

The elements in the NodeList can be accessed by an index number.

To access the second <p> node you can write:

```
myNodeList[1]
```

HTML DOM Node List Length
The length property defines the number of nodes in a node list:

```
myNodelist.length
```

The length property is useful when you want to loop through the nodes in a node list:

```
const myNodelist = document.querySelectorAll("p");
for (let i = 0; i < myNodelist.length; i++) {
  myNodelist[i].style.color = "red";
}
```

The Difference Between an HTMLCollection and a NodeList
A NodeList and an HTMLcollection is very much the same thing.

Both are array-like collections (lists) of nodes (elements) extracted from a document. The nodes can be accessed by index numbers. The index starts at 0.

Both have a length property that returns the number of elements in the list (collection).

An HTMLCollection is a collection of document elements.

A NodeList is a collection of document nodes (element nodes, attribute nodes, and text nodes).

HTMLCollection items can be accessed by their name, id, or index number.

NodeList items can only be accessed by their index number.

An HTMLCollection is always a live collection. Example: If you add a <li> element to a list in the DOM, the list in the HTMLCollection will also change.

A NodeList is most often a static collection. Example: If you add a <li> element to a list in the DOM, the list in NodeList will not change.

The getElementsByClassName() and getElementsByTagName() methods return a live HTMLCollection.

The querySelectorAll() method returns a static NodeList.

The childNodes property returns a live NodeList.

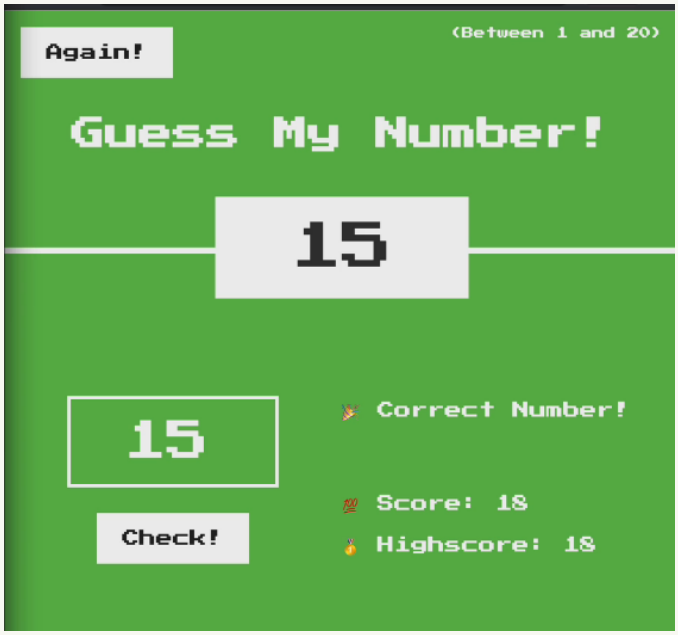
Not an Array!
A NodeList may look like an array, but it is not.

You can loop through a NodeList and refer to its nodes by index.

But, you cannot use Array methods like push(), pop(), or join() on a NodeList.

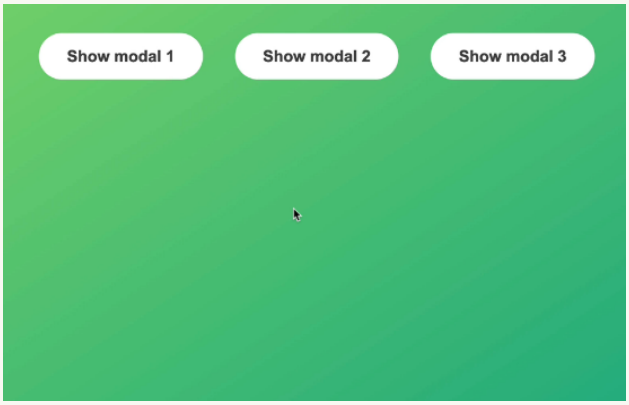
Assignment :

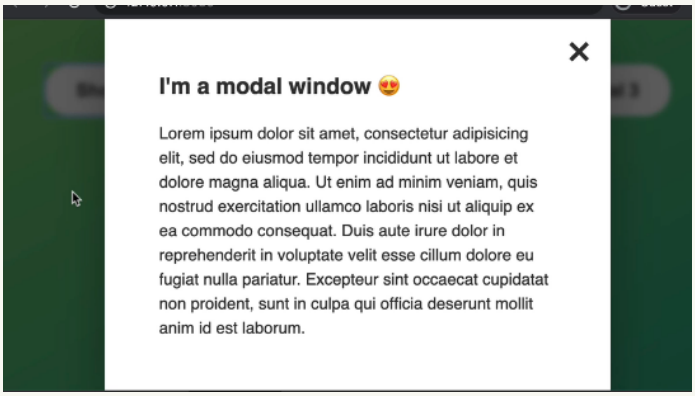
Make guess the game and upload code link here of github



Assignment:

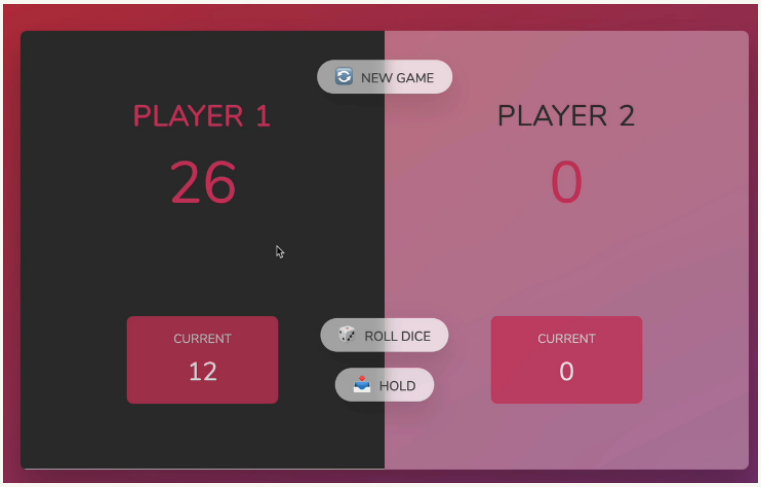
Make display model project and upload code link here for github



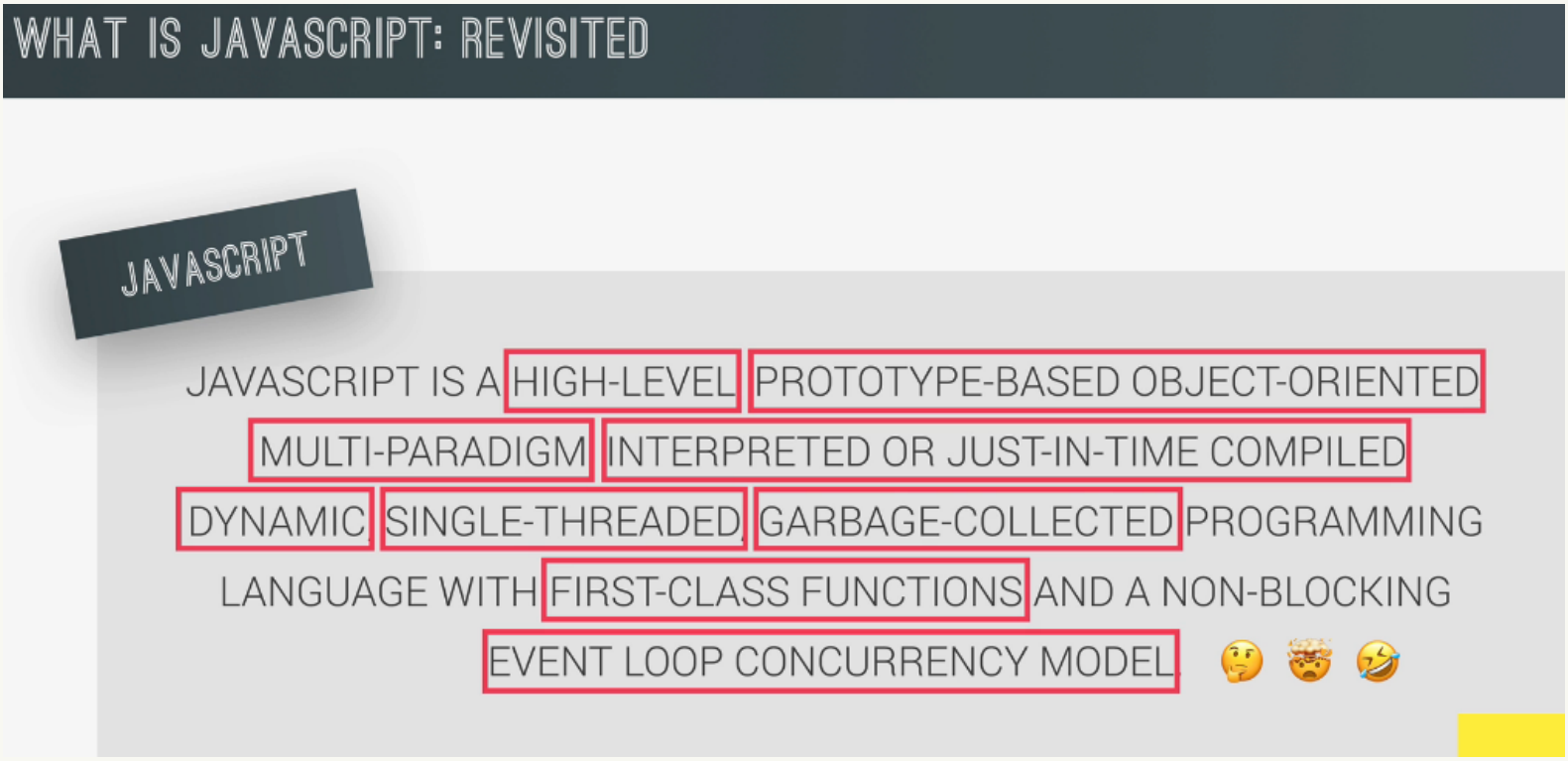


Assignment:

Make roll the dice and upload code link here for github



Javascript on a high level



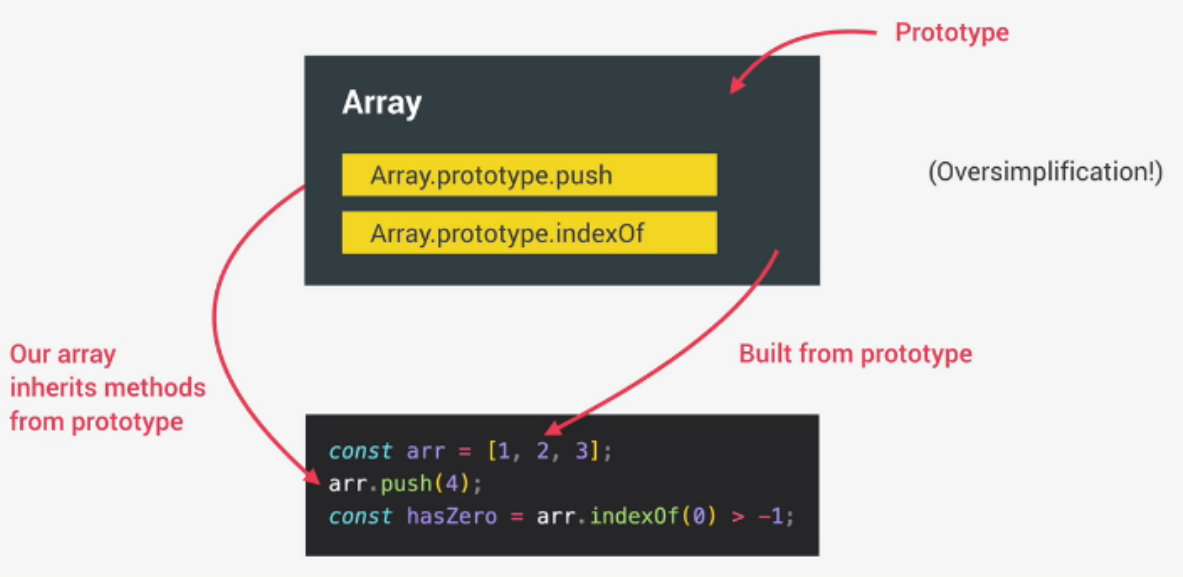
High level language means you do not have to manage resources, the resources will be easily managed by the the language itself.

Garbage collection means JavaScript will automatically remove the variables and object which are no longer referenced

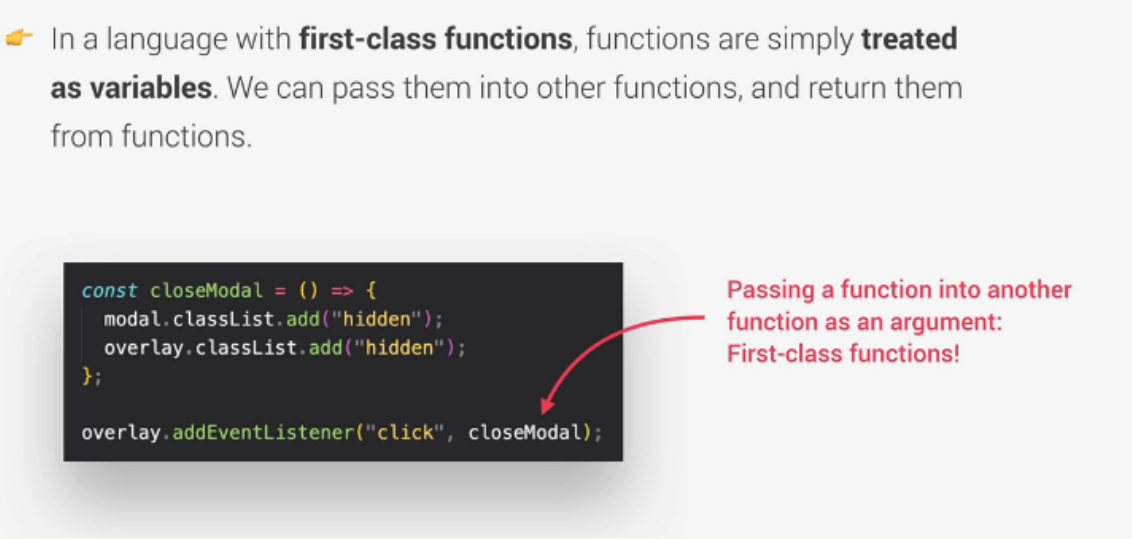
Computer understands only zeros and ones JavaScript engine converts are are instructions in anto 0s and 1s This is called interpreted or **Just in time compiled language**

JavaScript is a **multi-paradigm language**

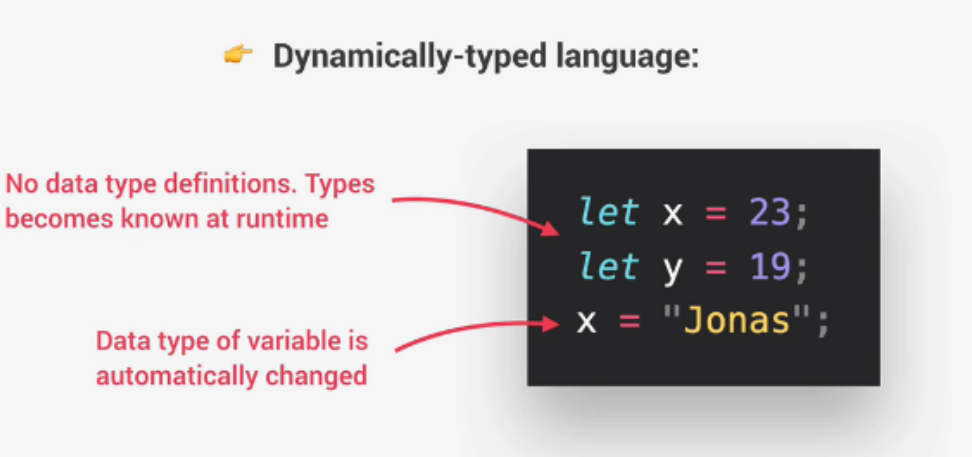
JavaScript is a **prototype based language**
Means the child inherits the functionality of the parent



JavaScript has a **first class functions**



JavaScript is **dynamically typed language**



JavaScript has **non-blocking event loops**

- **Concurrency model**: how the JavaScript engine handles multiple tasks happening at the same time.

  ↓ Why do we need that?

- JavaScript runs in one **single thread**, so it can only do one thing at a time.

  ↓ So what about a long-running task?

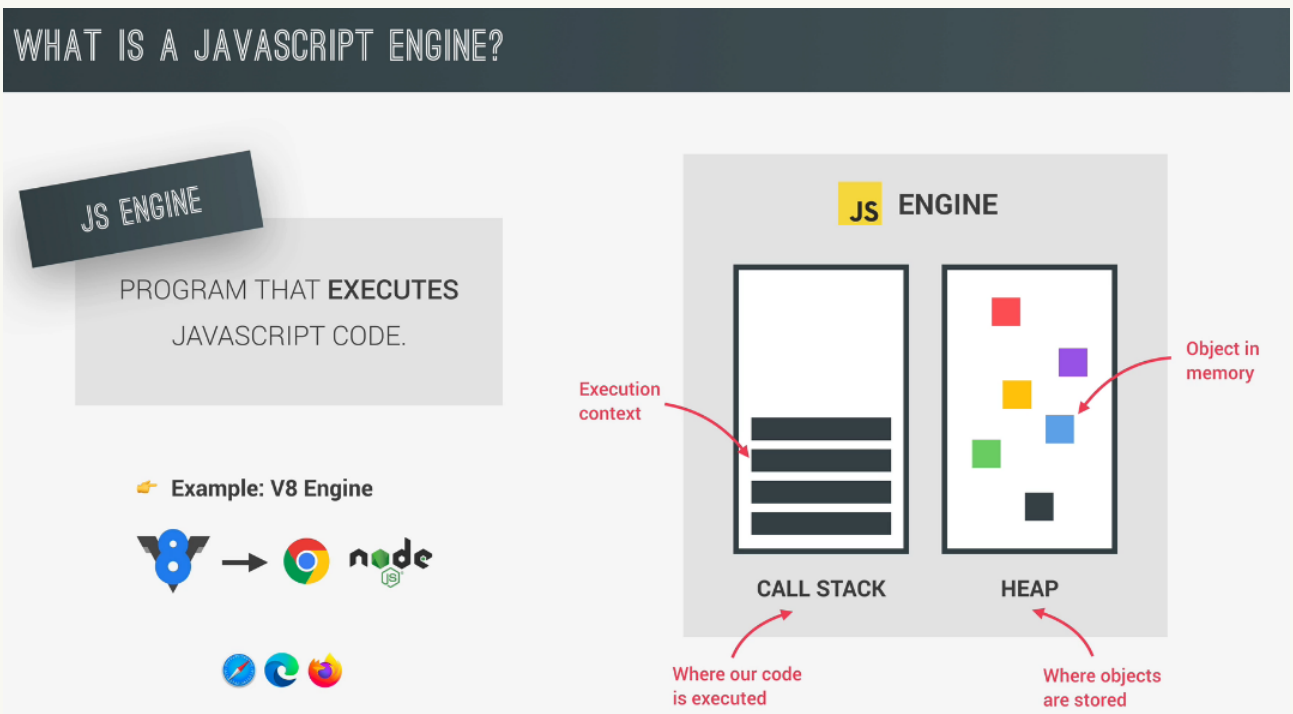
- Sounds like it would block the single thread. However, we want non-blocking behavior!

  ↓ How do we achieve that? (Oversimplification!)

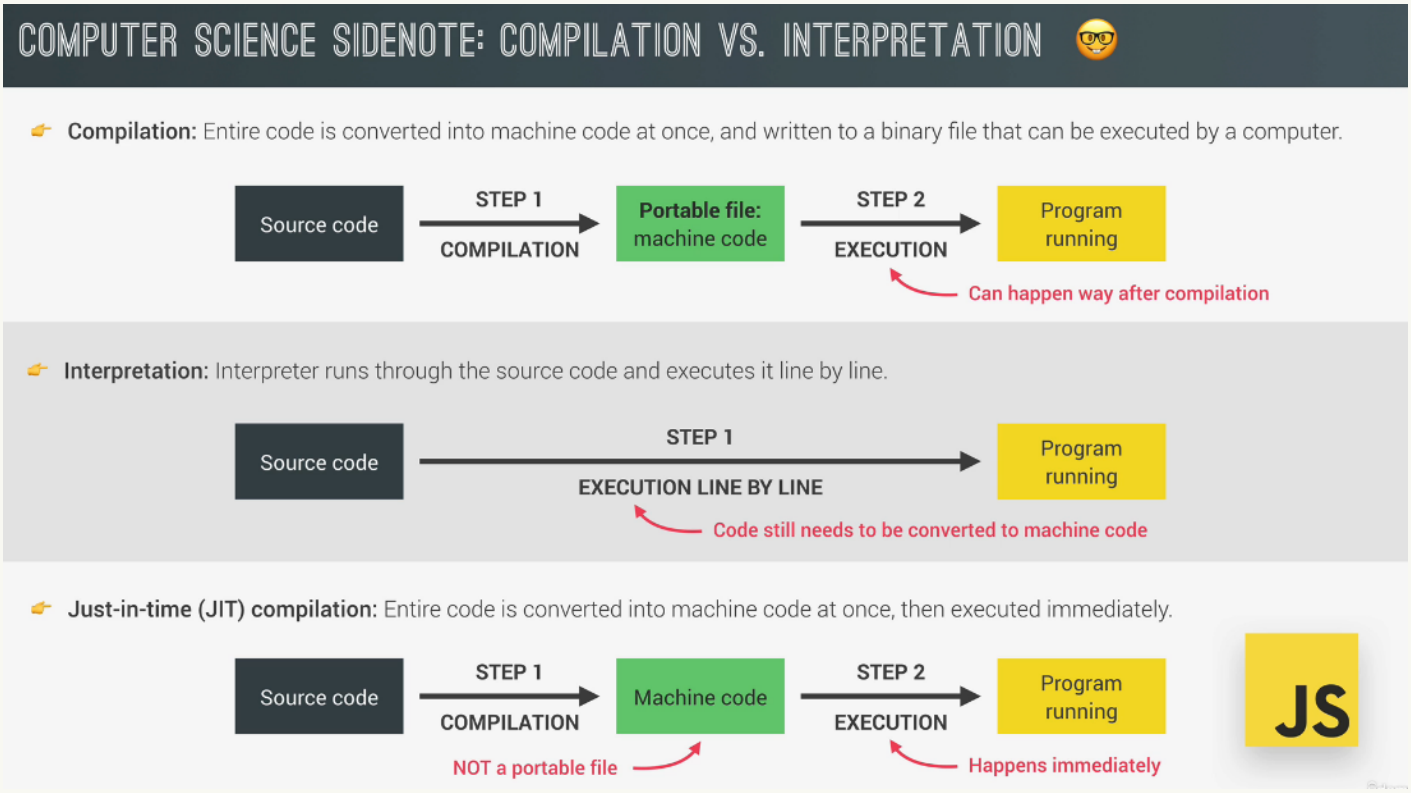
- By using an **event loop**: takes long running tasks, executes them in the "background", and puts them back in the main thread once they are finished.

What is js engine

Js engine contains call stack and heap
Call stack is used for execution by using execution context
And heap is where objects are stored

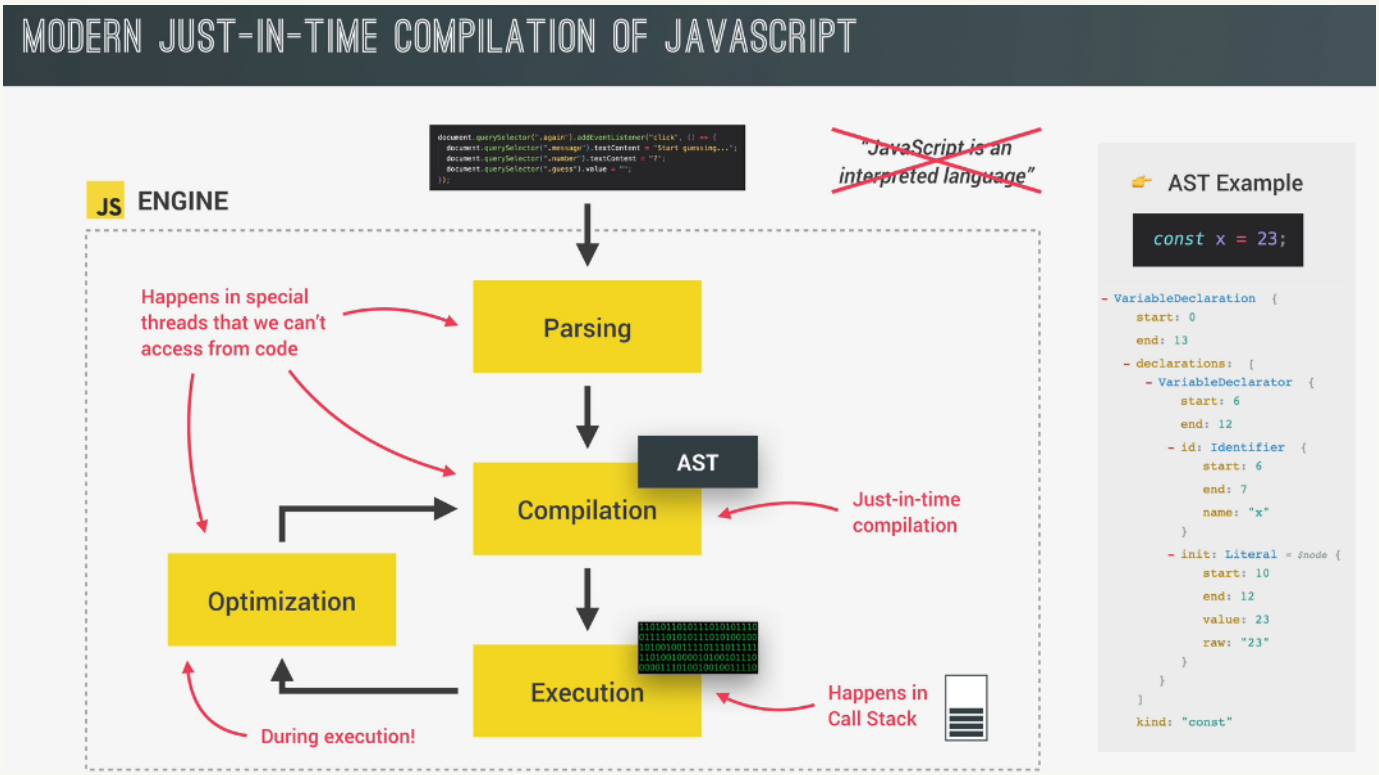


JavaScript compiled or interpreted language?
Answer is both. Its just-in-time compiled language

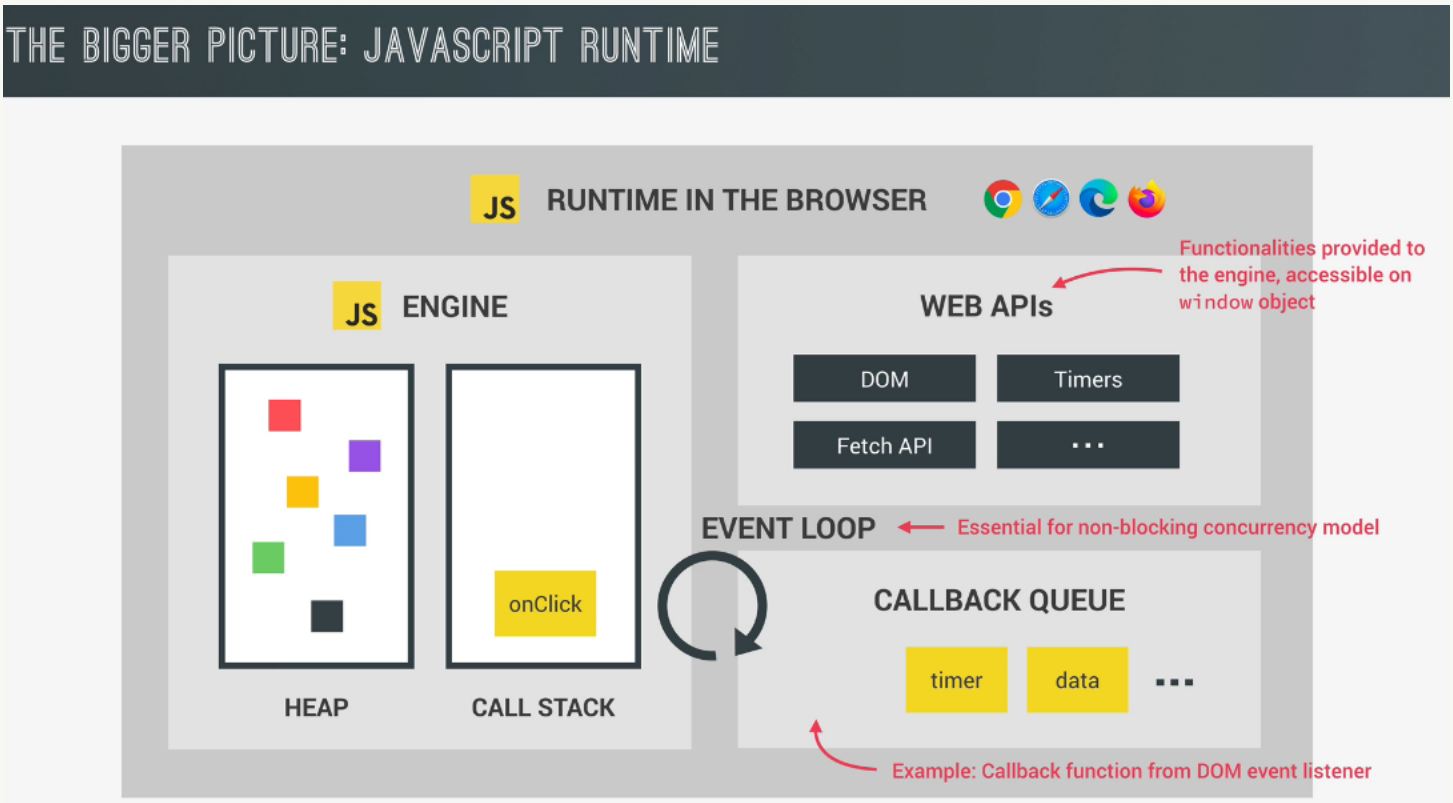


How compilation in js works

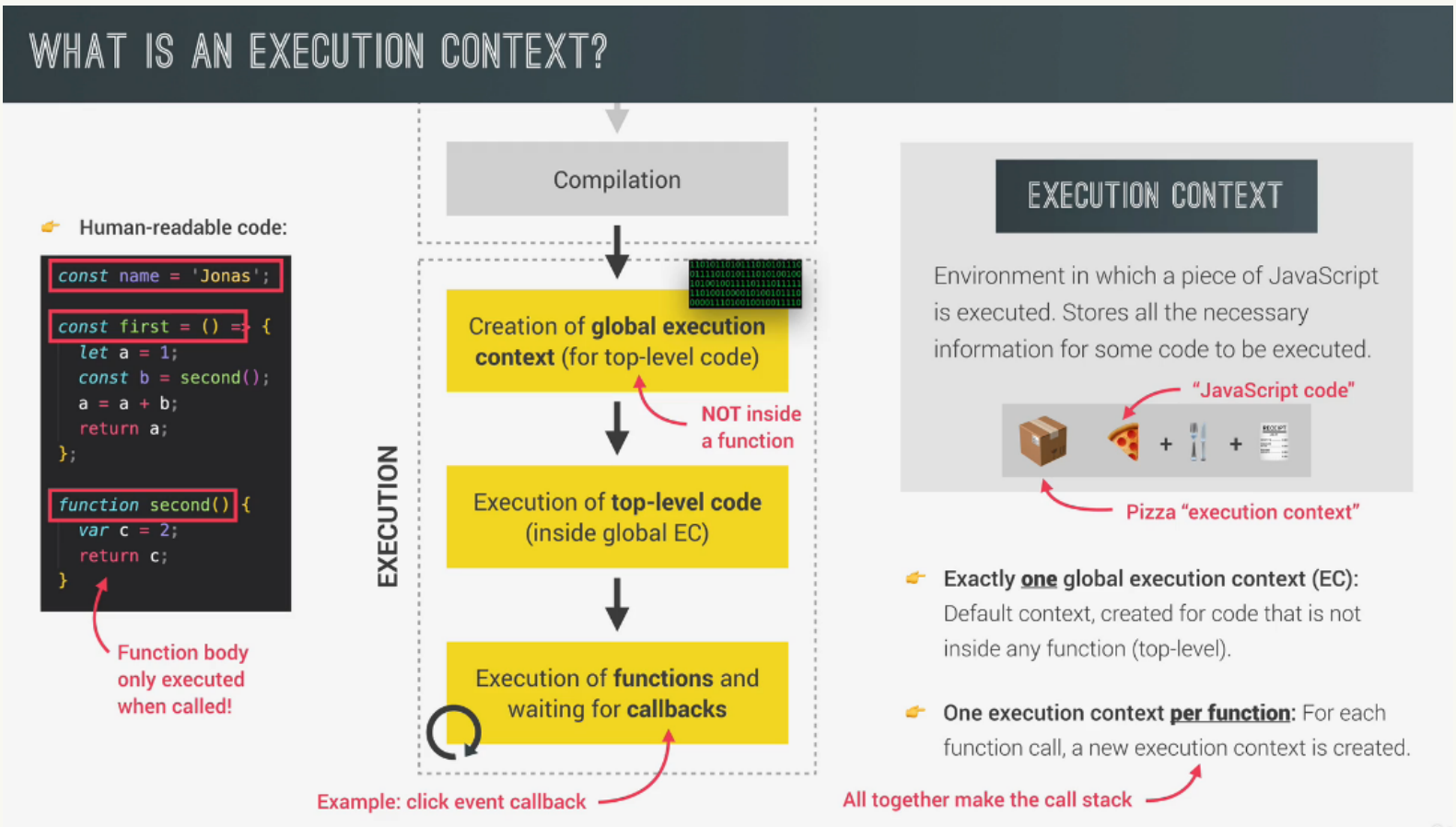
AST is abstract syntax tree



Js runtime



Execution context



Execution context in detail

EXECUTION CONTEXT IN DETAIL

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

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

Global

```
name = 'Jonas'
first = <function>
second = <function>
x = <unknown>
```

Literally the function code

Need to run `first()` first

first()

```
a = 1
b = <unknown>
```

Need to run `second()` first

second()

```
c = 2
arguments = [7, 9]
```

Array of passed arguments. Available in all "regular" functions (not arrow)

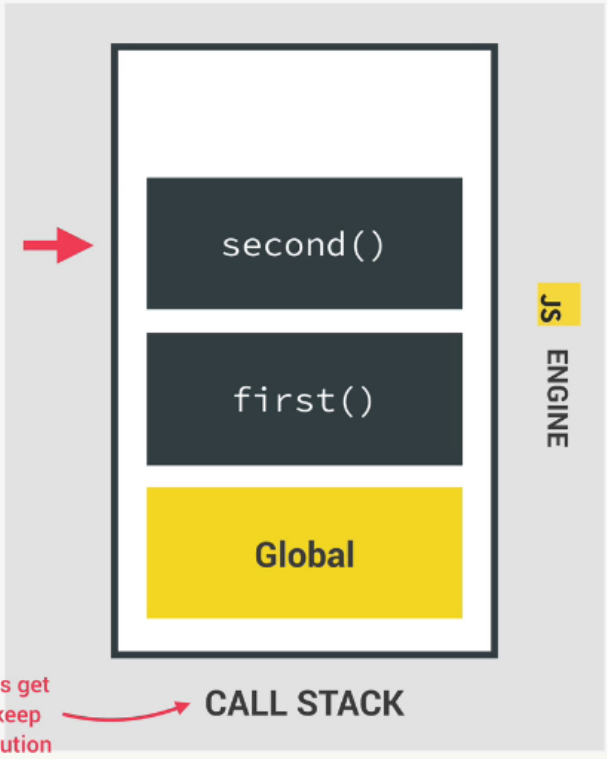
(Technically, values only become known during execution)

JavaScript uses call stack for the execution

THE CALL STACK

- Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

CALL STACK

Scope and scope chain

SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

EXECUTION CONTEXT

- Variable environment
- Scope chain
- `this` keyword

SCOPE CONCEPTS

- **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?"*;
- **Lexical scoping:** Scoping is controlled by **placement** of functions and blocks in the code;
- **Scope:** Space or environment in which a certain variable is **declared** (*variable environment in case of functions*). There is **global** scope, **function** scope, and **block** scope;
- **Scope of a variable:** Region of our code where a certain variable can be **accessed**.

THE 3 TYPES OF SCOPE

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
```

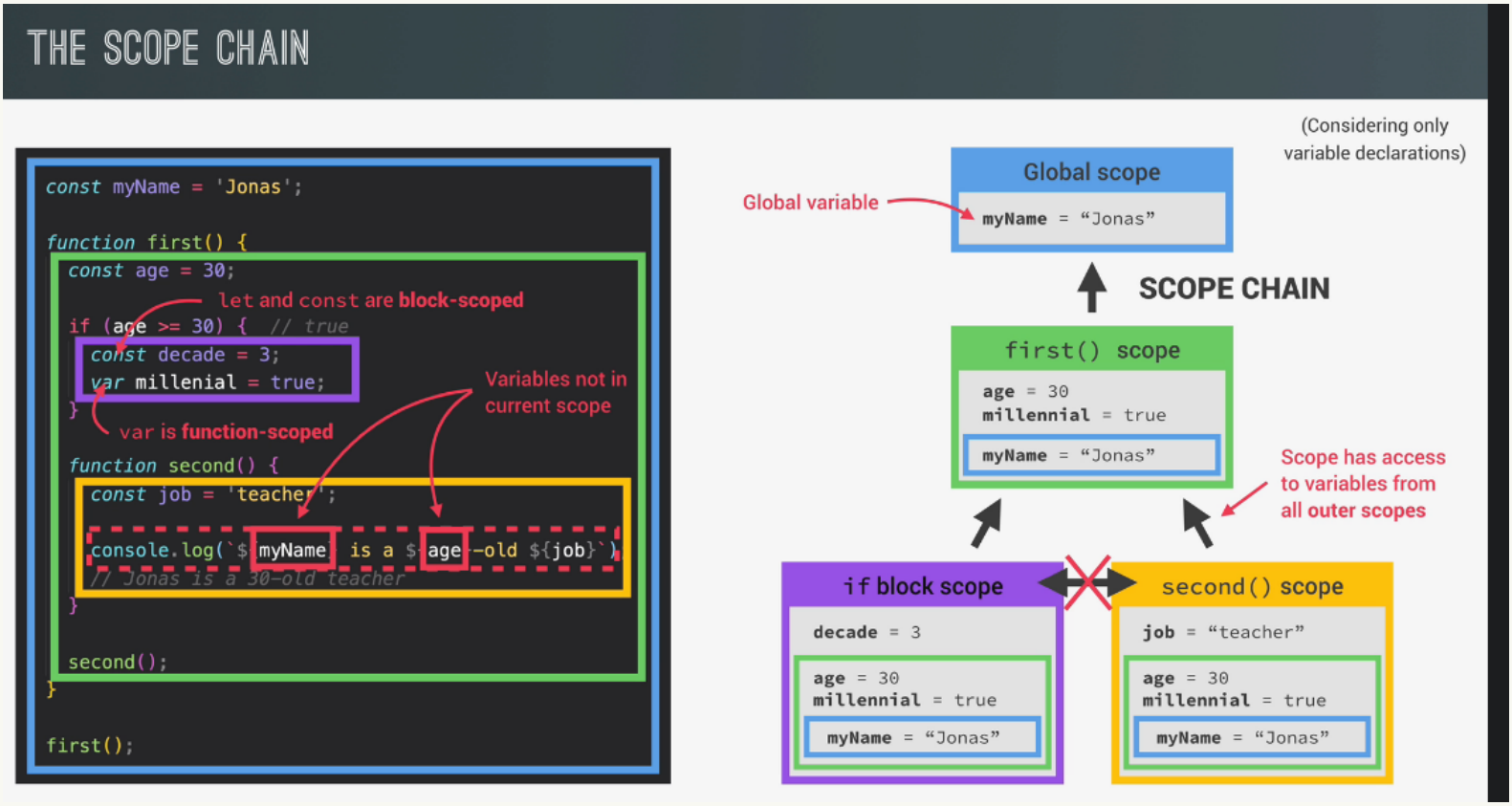
- Variables are accessible only **inside function**, **NOT** outside
- Also called local scope

BLOCK SCOPE (ES6)

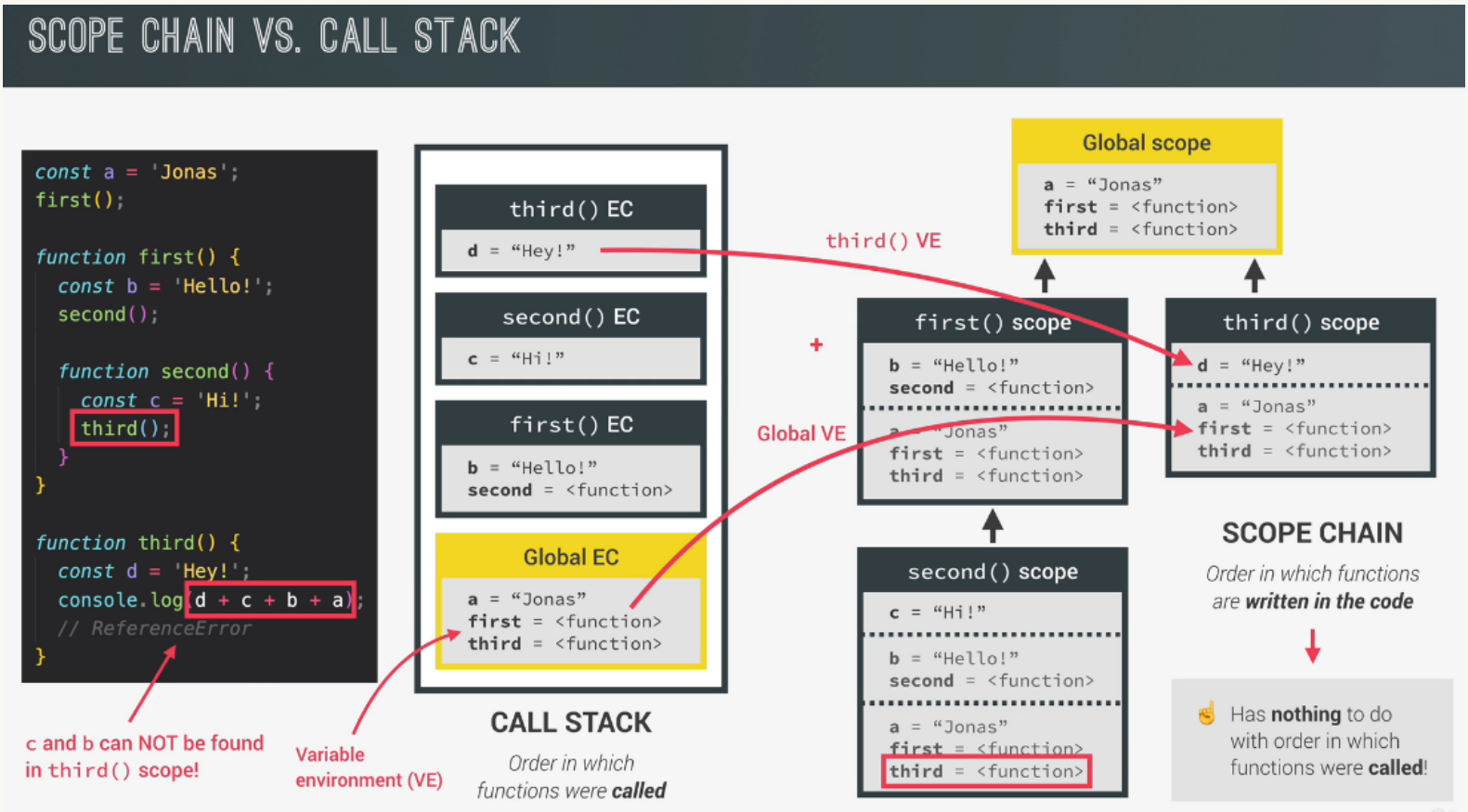
```
if (year >= 1981 && year <= 1996) {
  const millenial = true;
  const food = 'Avocado toast';
} ← Example: if block, for loop block, etc.

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)

Scope Chain



Scope chain example



SUMMARY 🥳

- Scoping asks the question *"Where do variables live?"* or *"Where can we access a certain variable, and where not?"*;
- There are 3 types of scope in JavaScript: the global scope, scopes defined by functions, and scopes defined by blocks;
- Only `let` and `const` variables are block-scoped. Variables declared with `var` end up in the closest function scope;
- In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written;
- Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it's looking for. This is called variable lookup;
- The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope;
- The scope chain in a certain scope is equal to adding together all the variable environments of the all parent scopes;
- The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!

Scoping in practice

Variable environment hoisting

HOISTING IN JAVASCRIPT

- **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

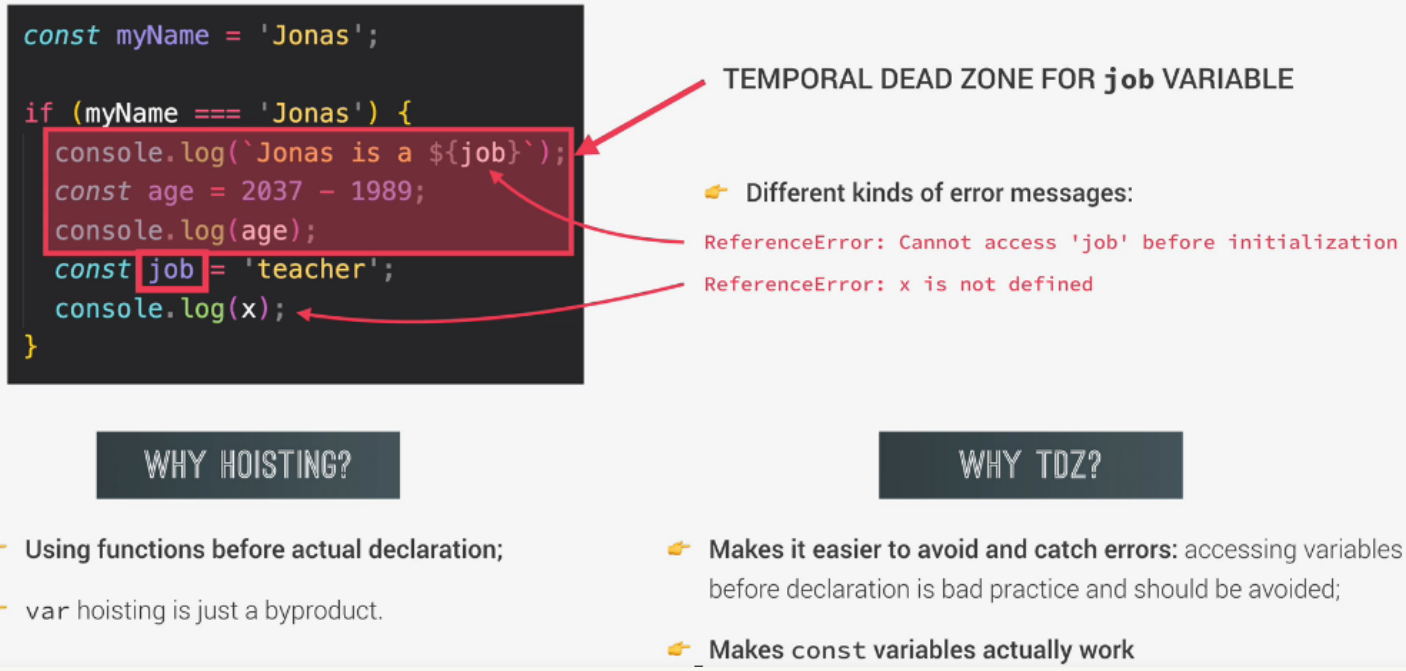
Temporal Dead Zone

An example

```
'use strict';

// Scoping in Practice


function calcAge(birthYear) {
  const age = 2037 - birthYear;

  function printAge() {
    let output = `${firstName}, you are ${age}, born in ${birthYear}`;
    console.log(output); // Jonas, you are 46, born in 1991

    if (birthYear >= 1981 && birthYear <= 1996) {
      var millenial = true;
      // Creating NEW variable with same name as outer scope's variable
      const firstName = 'Steven';

      // Reasssigning outer scope's variable
      output = 'NEW OUTPUT!';

      const str = `Oh, and you're a millenial, ${firstName}`;
      console.log(str); //Oh, and you're a millenial, Steven

      function add(a, b) {
        return a + b;
      }
    }
    // console.log(str);  // str is not defined
    console.log(millenial); // true
    // console.log(add(2, 3));  // add is not defined
    console.log(output); // NEW OUTPUT!
  }
  printAge();

  return age;
}

const firstName = 'Jonas';
calcAge(1991);
// console.log(age);  // age is not defined
// printAge();  // printAge is not defined
```

Temporal dead zone

TEMPORAL DEAD ZONE, LET AND CONST



WHY HOISTING?

- Using functions before actual declaration;
- `var` hoisting is just a byproduct.

WHY TDZ?

- Makes it easier to avoid and catch errors: accessing variables before declaration is bad practice and should be avoided;
- Makes `const` variables actually work

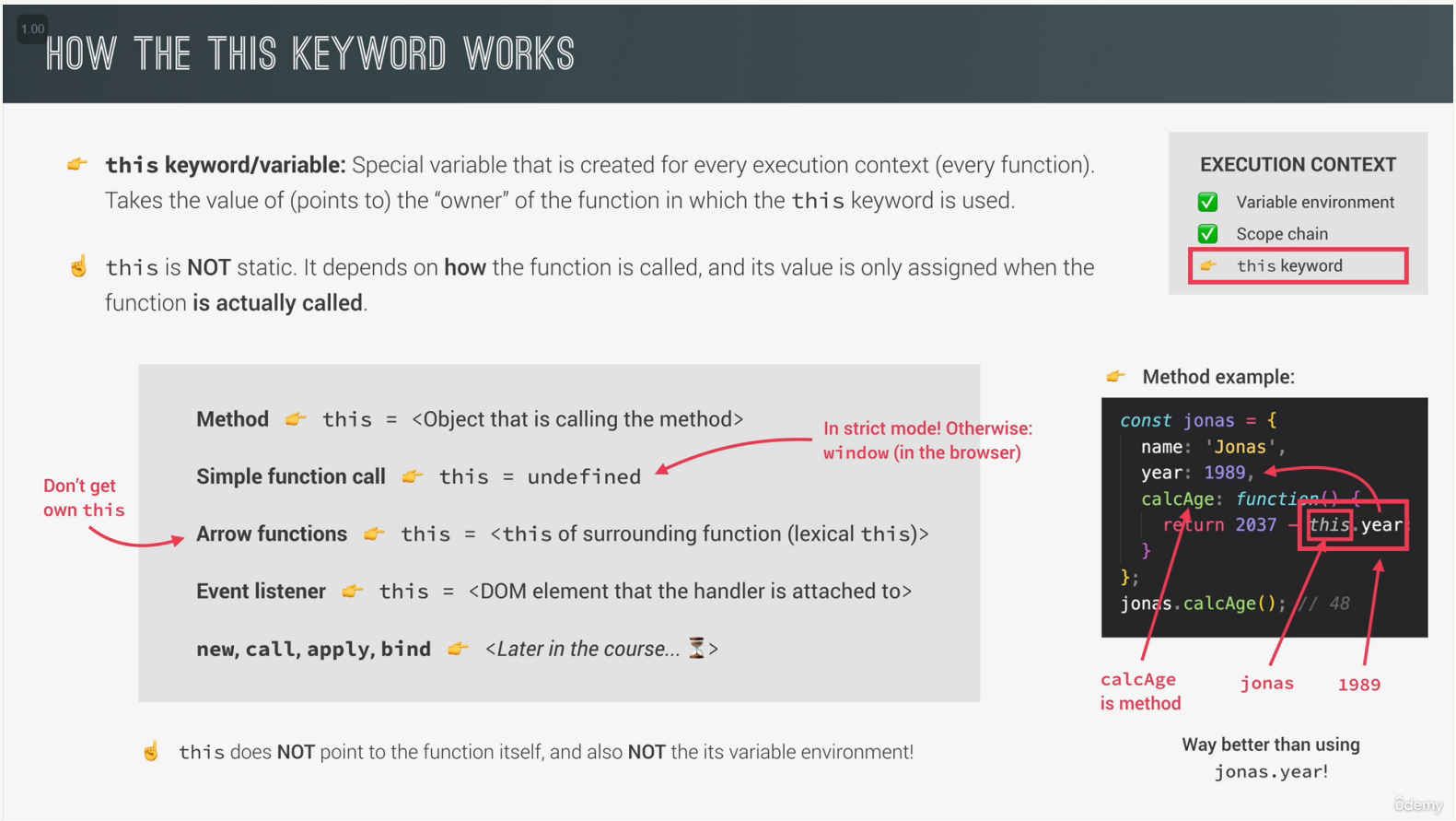
temporal dead zone is a zone where const and let variables are inaccessible or whenever we use will get a reference error because we have assigned it later and we are using it earlier

but the same do not happens with var keyword as because var has a function scope and it is hoisted but const and let are not hoisted

```
'use strict';
// Hoisting and TDZ in Practice

// Variables
console.log(me); // undefined
// console.log(job); // Cannot access 'job' before initialization
// console.log(year); // Cannot access 'year' before initialization

var me = 'Jonas';
let job = 'teacher';
const year = 1991;

// Functions
console.log(addDecl(2, 3)); // 5
// console.log(addExpr(2, 3)); // Cannot access 'addExpr' before initialization
console.log(addArrow); // undefined
// console.log(addArrow(2, 3)); // Cannot access 'addArrow' before initialization

function addDecl(a, b) {
  return a + b;
}

const addExpr = function (a, b) {
```

```
  return a + b;
};

var addArrow = (a, b) => a + b;

// Example
console.log(undefined); // undefined
if (!numProducts) deleteShoppingCart(); //  All products deleted!

var numProducts = 10;

function deleteShoppingCart() {
  console.log('All products deleted!');
} //

var x = 1;
let y = 2;
const z = 3;

console.log(x === window.x); // true
console.log(y === window.y); // false
console.log(z === window.z); // false
```

The this keyword



,

```
'use strict';
// The this Keyword in Practice
console.log(this); // Window {window: Window, self: Window, document: document, name: '',
location: Location, …}

const calcAge = function (birthYear) {
  console.log(2037 - birthYear);
  console.log(this);
};
calcAge(1991); // 46  // undefined

const calcAgeArrow = birthYear => {
  console.log(2037 - birthYear);
  console.log(this);
};
calcAgeArrow(1980); // 57 //  Window {window: Window, self: Window, document: document, name:
'', location: Location, …}

const jonas = {
  year: 1991,
  calcAge: function () {
    console.log(this);
    console.log(2037 - this.year);
  },
};
jonas.calcAge(); // {year: 1991, calcAge: ƒ}  // 46

const matilda = {
  year: 2017,
};

matilda.calcAge = jonas.calcAge;
matilda.calcAge(); // {year: 2017, calcAge: ƒ}  // 20

const f = jonas.calcAge;
f(); // undefined script.js:21 Uncaught TypeError: Cannot read properties of undefined
(reading 'year') at calcAge
```

Regular Function vs Arrow Function

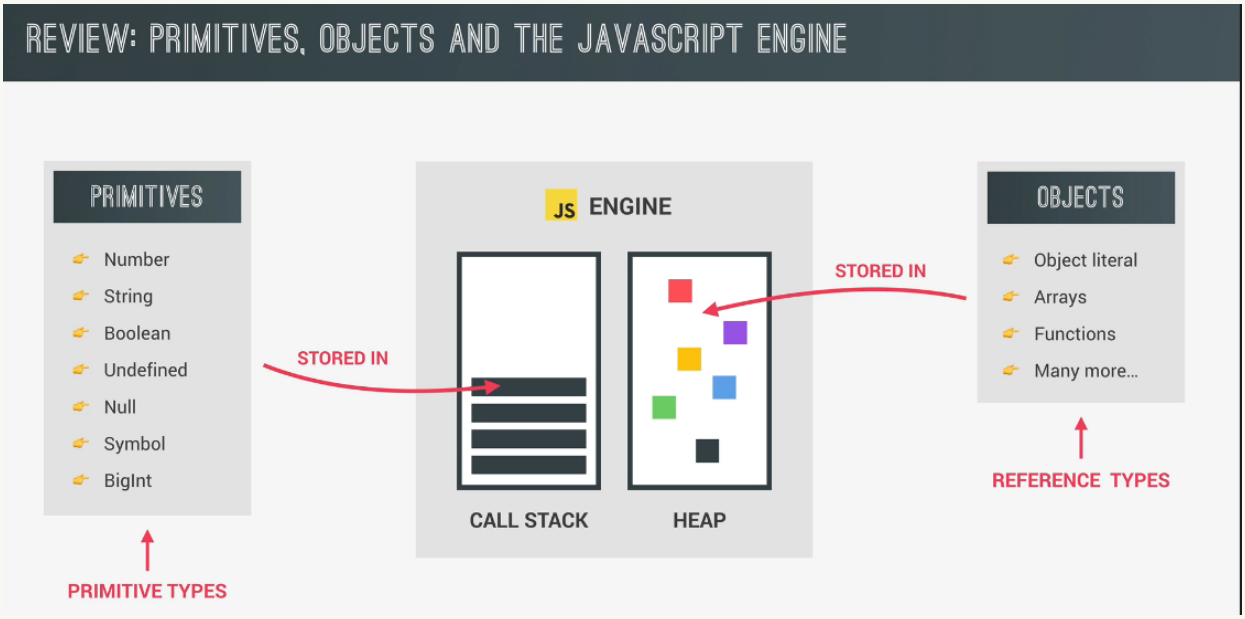
```
'use strict';
// Regular Functions vs. Arrow Functions
// var firstName = 'Matilda';

const jonas = {
```

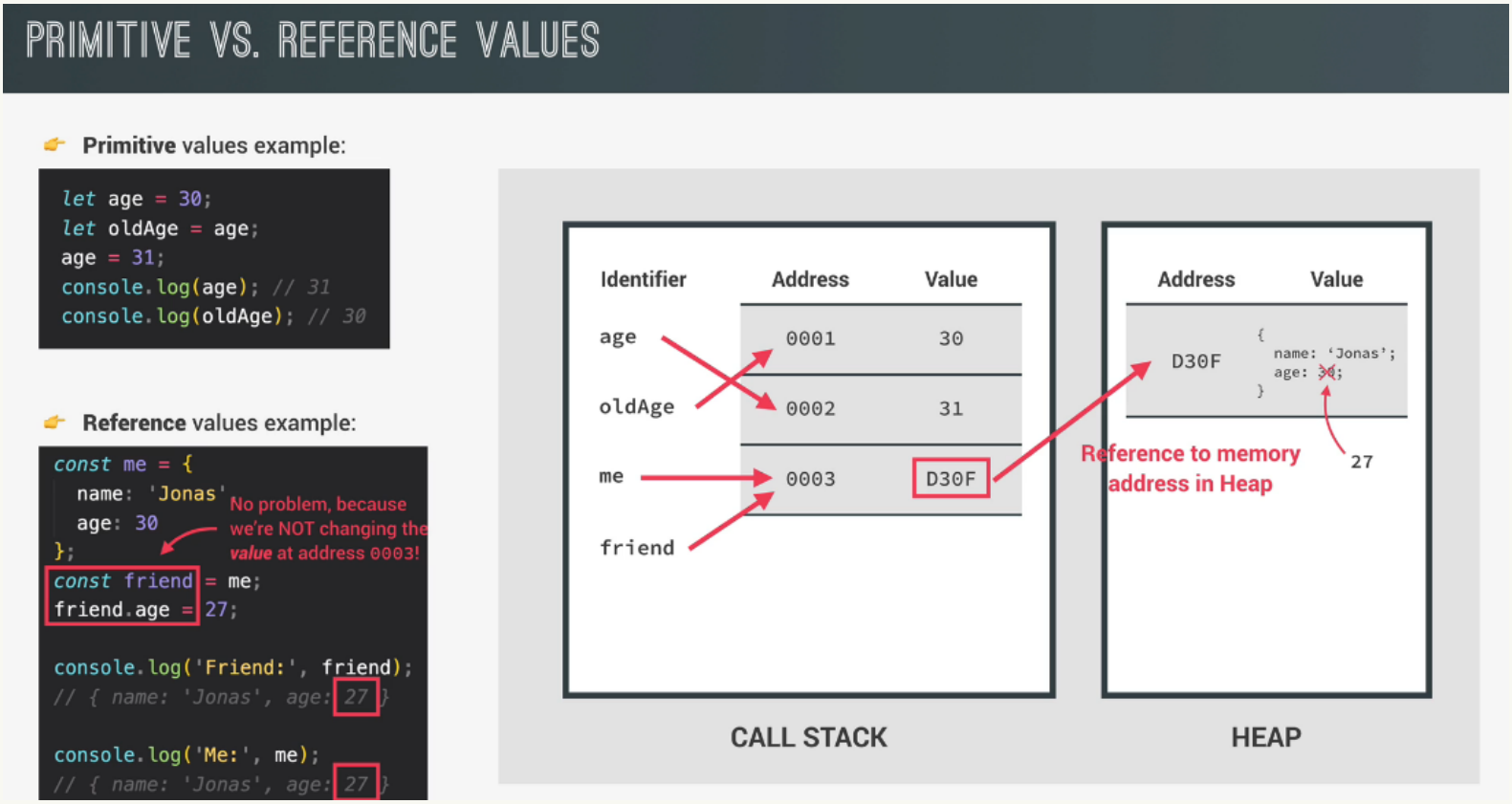
```
  firstName: 'Jonas',
  year: 1991,
  calcAge: function () {
    // console.log(this);
    console.log(2037 - this.year);

    //Solution 1
    // const self = this; // self or that
    // const isMillenial = function () {
    //   console.log(self);
    //   console.log(self.year >= 1981 && self.year <= 1996);
    // };

    // // Solution 2
    const isMillenial = () => {
      console.log(this);
      console.log(this.year >= 1981 && this.year <= 1996);
    };
    isMillenial();
  },

  greet: () => {
    console.log(this);
    console.log(`Hey ${this.firstName}`);
  },
};
jonas.greet(); // Window {window: Window, self: Window, document: document, name: '',
location: Location, …}  // Hey undefined
jonas.calcAge(); // if Solution 1 implemented :::  // 46  // {firstName: 'Jonas', year: 1991,
calcAge: ƒ, greet: ƒ}   // true
jonas.calcAge(); // if Solution 2 implemented :::  // 46  // {firstName: 'Jonas', year: 1991,
calcAge: ƒ, greet: ƒ}   // true

// arguments keyword
const addExpr = function (a, b) {
  console.log(arguments);
  return a + b;
};
addExpr(2, 5);  // Arguments(2) [2, 5, callee: (...), Symbol(Symbol.iterator): ƒ]
addExpr(2, 5, 8, 12);  // Arguments(4) [2, 5, 8, 12, callee: (...), Symbol(Symbol.iterator):
ƒ]

var addArrow = (a, b) => {
  console.log(arguments);
  return a + b;
};
addArrow(2, 5, 8); // Uncaught ReferenceError: arguments is not defined at addArrow
(script.js:44:15)
```

This keyword inside a regular function is undefined
this keyword inside Arrow function depends on its parent element

arguments keyboard is only available in regular function and not in Arrow function

Primitives versus objects





```
'use strict';
// Objects vs. primitives
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30

const me = {
  name: 'Jonas',
  age: 30,
};
const friend = me;
friend.age = 27;
console.log('Friend:', friend); // Friend: {name: 'Jonas', age: 27}
console.log('Me', me); // Me {name: 'Jonas', age: 27}
```

Primitives versus objects In practice

```
'use strict';
// Primitives vs. Objects in Practice

// Primitive types
let lastName = 'Williams';
let oldLastName = lastName;
lastName = 'Davis';
console.log(lastName, oldLastName);  // Davis Williams

// Reference types
const jessica = {
  firstName: 'Jessica',
  lastName: 'Williams',
  age: 27,
};
const marriedJessica = jessica;
marriedJessica.lastName = 'Davis';
console.log('Before marriage:', jessica); // Before marriage: Object
// Object
// age: 27
// firstName: "Jessica"
// lastName: "Davis"
// [[Prototype]]: Object


console.log('After marriage: ', marriedJessica); // After marriage:  Object
```

```
// Object
// age: 27
// firstName: "Jessica"
// lastName: "Davis"
// [[Prototype]]: Object

// marriedJessica = {};

// Copying objects
const jessica2 = {
  firstName: 'Jessica',
  lastName: 'Williams',
  age: 27,
  family: ['Alice', 'Bob'],
};

const jessicaCopy = Object.assign({}, jessica2);
jessicaCopy.lastName = 'Davis';

jessicaCopy.family.push('Mary');
jessicaCopy.family.push('John');

console.log('Before marriage:', jessica2); // Before marriage: Object
// Object
// age: 27
// family: (4) ['Alice', 'Bob', 'Mary', 'John']
// firstName: "Jessica"
// lastName: "Williams"
// [[Prototype]]: Object


console.log('After marriage: ', jessicaCopy); // After marriage:  Object

// Object
// age: 27
// family: (4) ['Alice', 'Bob', 'Mary', 'John']
// firstName: "Jessica"
// lastName: "Davis"
// [[Prototype]]: Object
```

Destructuring arrays

```
'use strict';

// Destructuring Arrays
const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
```

```
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  }

};

const arr = [2, 3, 4];
const a = arr[0];
const b = arr[1];
const c = arr[2];

const [x, y, z] = arr;
console.log(x, y, z);  //  2 3 4
console.log(arr);  //  (3) [2, 3, 4]

let [main, , secondary] = restaurant.categories;
console.log(main, secondary); //  Italian Vegetarian

// swapping variables
// const temp = main;
// main = secondary;
// secondary = temp;
// console.log(main, secondary);

// swapping variables using destructuring
[main, secondary] = [secondary, main];
console.log(main, secondary);  //  Vegetarian Italian

// Receive 2 return values from a function
const [starter, mainCourse] = restaurant.order(2, 0);
console.log(starter, mainCourse);  //  Garlic Bread Pizza

// Nested destructuring
const nested = [2, 4, [5, 6]];
// const [i, , j] = nested;
const [i, , [j, k]] = nested;
console.log(i, j, k);  //  2 5 6

// Default values
const [p = 1, q = 1, r = 1] = [8, 9];
console.log(p, q, r);  //  8 9 1
```

Destructuring object

```
'use strict';
// Destructuring Objects
const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },
  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  }

};


restaurant.orderDelivery({
  time: '22:30',
  address: 'Via del Sole, 21',
  mainIndex: 2,
  starterIndex: 2,
});  // Order received! Garlic Bread and Risotto will be delivered to Via del Sole, 21 at
22:30
```

```
restaurant.orderDelivery({
  address: 'Via del Sole, 21',
  starterIndex: 1,
});  // Order received! Bruschetta and Pizza will be delivered to Via del Sole, 21 at 20:00

// destructuring object
const { name, openingHours, categories } = restaurant;
console.log(name, openingHours, categories); // Classico Italiano {thu: {…}, fri: {…}, sat:
{…}} (4) ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']

// destructuring object while renaming the variable
const {
  name: restaurantName,
  openingHours: hours,
  categories: tags,
} = restaurant;
console.log(restaurantName, hours, tags); // Classico Italiano {thu: {…}, fri: {…}, sat: {…}}
(4) ['Italian', 'Pizzeria', 'Vegetarian', 'Organic']

// giving Default values, if the property do not exist then default gets applied
const { menu = [], starterMenu: starters = [] } = restaurant;
console.log(menu, starters);  // [] (4) ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese
Salad']

// Mutating variables
let a = 111;
let b = 999;
const obj = { a: 23, b: 7, c: 14 };
({ a, b } = obj);
console.log(a, b);  // 23 7

// Nested objects destructuring
const {
  fri: { open: o, close: c },
} = openingHours;
console.log(o, c);  // 11 23
```

The spread operator

```
'use strict';

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
```

```
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta
', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },
  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },

};

// The Spread Operator (...)

const arr = [7, 8, 9];
const badNewArr = [1, 2, arr[0], arr[1], arr[2]];
console.log(badNewArr); // (5) [1, 2, 7, 8, 9]

const newArr = [1, 2, ...arr];
console.log(newArr); // (5) [1, 2, 7, 8, 9]

console.log(...newArr); // 1 2 7 8 9
console.log(1, 2, 7, 8, 9); // 1 2 7 8 9
```

```
const newMenu = [...restaurant.mainMenu, 'Gnocci'];
console.log(newMenu); // (4) ['Pizza', 'Pasta', 'Risotto', 'Gnocci']

// Copy array
const mainMenuCopy = [...restaurant.mainMenu];

// Join 2 arrays
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];
console.log(menu); // (7) ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad',
'Pizza', 'Pasta', 'Risotto']

// Iterables: arrays, strings, maps, sets. NOT objects
const str = 'Jonas';
const letters = [...str, ' ', 'S.'];
console.log(letters); // (7) ['J', 'o', 'n', 'a', 's', ' ', 'S.']
console.log(...str); // J o n a s
// console.log(`${...str} Schmedtmann`);

// Real-world example
const ingredients = [
  // prompt("Let's make pasta! Ingredient 1?"),
  // prompt('Ingredient 2?'),
  // prompt('Ingredient 3'),
];
console.log(ingredients); // []

restaurant.orderPasta(ingredients[0], ingredients[1], ingredients[2]); // Here is your
declicious pasta with undefined, undefined and undefined
restaurant.orderPasta(...ingredients);  // Here is your declicious pasta with undefined,
undefined and undefined

// Objects
const newRestaurant = { foundedIn: 1998, ...restaurant, founder: 'Guiseppe' };
console.log(newRestaurant); // {foundedIn: 1998, name: 'Classico Italiano', location: 'Via
Angelo Tavanti 23, Firenze, Italy', categories: Array(4), starterMenu: Array(4), …}

const restaurantCopy = { ...restaurant };
restaurantCopy.name = 'Ristorante Roma';
console.log(restaurantCopy.name); // Ristorante Roma
console.log(restaurant.name); // Classico Italiano
```

Rest operator

```
'use strict';
```

```
const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },
  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },
  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },

};


// Rest Pattern and Parameters
// 1) Destructuring
```

```
// SPREAD, because on RIGHT side of =
const arr = [1, 2, ...[3, 4]];

// REST, because on LEFT side of =
const [a, b, ...others] = [1, 2, 3, 4, 5];
console.log(a, b, others);  // 1 2 (3) [3, 4, 5]

const [pizza, , risotto, ...otherFood] = [
  ...restaurant.mainMenu,
  ...restaurant.starterMenu,
];
console.log(pizza, risotto, otherFood);  //  Pizza Risotto (4) ['Focaccia', 'Bruschetta',
'Garlic Bread', 'Caprese Salad']

// Objects
const { sat, ...weekdays } = restaurant.openingHours;
console.log(weekdays);  //

// 2) Functions
const add = function (...numbers) {
  let sum = 0;
  for (let i = 0; i < numbers.length; i++) sum += numbers[i];
  console.log(sum);
};

add(2, 3);  //  5
add(5, 3, 7, 2);  //  17
add(8, 2, 5, 3, 2, 1, 4);  // 25

const x = [23, 5, 7];
add(...x);  //  35

restaurant.orderPizza('mushrooms', 'onion', 'olives', 'spinach');  //mushrooms    //  (3)
['onion', 'olives', 'spinach']
restaurant.orderPizza('mushrooms');  //mushrooms    //  []
```

Short circuiting operator

```
'use strict';

const restaurant = {
  name: 'Classico Italiano',
```

```
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },
  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your delicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },
  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },

};

// Short Circuiting (&& and ||)

console.log('---- OR ----');  //  ---- OR ----
// Use ANY data type, return ANY data type, short-circuiting
console.log(3 || 'Jonas');  //  3
console.log('' || 'Jonas');  //  Jonas
console.log(true || 0);  //  true
```

```
console.log(undefined || null);  //  null

console.log(undefined || 0 || '' || 'Hello' || 23 || null);  //  Hello

restaurant.numGuests = 0;
const guests1 = restaurant.numGuests ? restaurant.numGuests : 10;
console.log(guests1);  //  10

const guests2 = restaurant.numGuests || 10;
console.log(guests2);  //  10

console.log('---- AND ----');  //  ---- AND ----
console.log(0 && 'Jonas');  //  0
console.log(7 && 'Jonas');  //  Jonas

console.log('Hello' && 23 && null && 'jonas');  //  null

// Practical example
if (restaurant.orderPizza) {    // mushrooms  // ['spinach']
  restaurant.orderPizza('mushrooms', 'spinach');
}

restaurant.orderPizza && restaurant.orderPizza('mushrooms', 'spinach');  // mushrooms  //
['spinach']
```

nullish coalescing operator

```
'use strict';

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
```

```
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },
  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },
  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },

};

// The Nullish Coalescing Operator
restaurant.numGuests = 0;
const guests = restaurant.numGuests || 10;
console.log(guests); // 10

// Nullish: null and undefined (NOT 0 or '')
const guestCorrect = restaurant.numGuests ?? 10;
console.log(guestCorrect);  // 0
```

Logical assignment operator

```
'use strict';
```

```
const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  openingHours: {
    thu: {
      open: 12,
      close: 22,
    },
    fri: {
      open: 11,
      close: 23,
    },
    sat: {
      open: 0, // Open 24 hours
      close: 24,
    },
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },
  order: function (starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },
  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },
  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },

};

// Logical Assignment Operators
const rest1 = {
  name: 'Capri',
  // numGuests: 20,
```

```
  numGuests: 0,
};

const rest2 = {
  name: 'La Piazza',
  owner: 'Giovanni Rossi',
};

// OR assignment operator
console.log(rest1.numGuests = rest1.numGuests || 10);  //  10
console.log(rest2.numGuests = rest2.numGuests || 10);  //  10
console.log(rest1.numGuests ||= 10);  //  10
console.log(rest2.numGuests ||= 10);  //  10

// nullish assignment operator (null or undefined));
console.log(rest1.numGuests ??= 10);  //  10
console.log(rest2.numGuests ??= 10);  //  10

// AND assignment operator);
console.log(rest1.owner = rest1.owner && '<ANONYMOUS>');  //  undefined
console.log(rest2.owner = rest2.owner && '<ANONYMOUS>');  //  <ANONYMOUS>
console.log(rest1.owner &&= '<ANONYMOUS>');  //  undefined
console.log(rest2.owner &&= '<ANONYMOUS>');  //  <ANONYMOUS>

console.log(rest1);  //  {name: 'Capri', numGuests: 10, owner: undefined}
console.log(rest2);  //  {name: 'La Piazza', owner: '<ANONYMOUS>', numGuests: 10}
```

// Coding Challenge #1

/*
We're building a football betting app (soccer for my American friends 😅)!

Suppose we get data from a web service about a certain game (below). In this challenge we're gonna work with the data. So here are your tasks:

1. Create one player array for each team (variables 'players1' and 'players2')
2. The first player in any player array is the goalkeeper and the others are field players. For Bayern Munich (team 1) create one variable ('gk') with the goalkeeper's name, and one array ('fieldPlayers') with all the remaining 10 field players
3. Create an array 'allPlayers' containing all players of both teams (22 players)
4. During the game, Bayern Munich (team 1) used 3 substitute players. So create a new array ('players1Final') containing all the original team1 players plus 'Thiago', 'Coutinho' and 'Perisic'
5. Based on the game.odds object, create one variable for each odd (called 'team1', 'draw' and 'team2')
6. Write a function ('printGoals') that receives an arbitrary number of player names (NOT an array) and prints each of them to the console, along with the number of goals that were scored in total (number of player names passed in)
7. The team with the lower odd is more likely to win. Print to the console which team is more likely to win, WITHOUT using an if/else statement or the ternary operator.

TEST DATA FOR 6: Use players 'Davies', 'Muller', 'Lewandowski' and 'Kimmich'. Then, call the function again with players from game.scored

GOOD LUCK 😀

```
const game = {
  team1: 'Bayern Munich',
  team2: 'Borrussia Dortmund',
  players: [
    [
      'Neuer',
      'Pavard',
      'Martinez',
      'Alaba',
      'Davies',
      'Kimmich',
      'Goretzka',
      'Coman',
      'Muller',
      'Gnarby',
      'Lewandowski',
    ],
    [
      'Burki',
      'Schulz',
      'Hummels',
      'Akanji',
      'Hakimi',
      'Weigl',
      'Witsel',
      'Hazard',
      'Brandt',
      'Sancho',
      'Gotze',
    ],
  ],
  score: '4:0',
  scored: ['Lewandowski', 'Gnarby', 'Lewandowski', 'Hummels'],
  date: 'Nov 9th, 2037',
  odds: {
    team1: 1.33,
    x: 3.25,
    team2: 6.5,
  },
};
```

Looking a for of loop

```
'use strict';

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

};

// The for-of Loop
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];

for (const item of menu) console.log(item);
//output goes here...../

// Focaccia
// Bruschetta
// Garlic Bread
// Caprese Salad
// Pizza
// Pasta
// Risotto

for (const [i, el] of menu.entries()) {
  console.log(`${i + 1}: ${el}`);
}
//output goes here...../
// 1: Focaccia
// 2: Bruschetta
// 3: Garlic Bread
// 4: Caprese Salad
// 5: Pizza
// 6: Pasta
// 7: Risotto
console.log([...menu.entries()]);
//output goes here...../
// Array(7)
// 0: (2) [0, 'Focaccia']
// 1: (2) [1, 'Bruschetta']
// 2: (2) [2, 'Garlic Bread']
// 3: (2) [3, 'Caprese Salad']
// 4: (2) [4, 'Pizza']
// 5: (2) [5, 'Pasta']
```

```
// 6: (2) [6, 'Risotto']
// length: 7
```

Enhanced object literals

```
'use strict';
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
   // ES6 enhanced expression in property
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  // ES6 enhanced object literals
  openingHours,
 // ES6 enhanced function
  order(starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
```

```
  },

  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },

  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },
};
```

Optional chaining

```
'use strict';
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
   // ES6 enhanced expression in property
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  // ES6 enhanced object literals
  openingHours,
 // ES6 enhanced function
  order(starterIndex, mainIndex) {
```

```
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },

  orderPasta(ing1, ing2, ing3) {
    console.log(
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },

  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },
};

// Optional Chaining
if (restaurant.openingHours && restaurant.openingHours.mon)
  console.log(restaurant.openingHours.mon.open);

// console.log(restaurant.openingHours.mon.open);

// WITH optional chaining
console.log(restaurant.openingHours.mon?.open);  // undefined
console.log(restaurant.openingHours?.mon?.open);  // undefined

// Example
const days = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];

for (const day of days) {
  const open = restaurant.openingHours[day]?.open ?? 'closed';
  console.log(`On ${day}, we open at ${open}`);
}
  // output goes here...../
  // On mon, we open at closed
  // On tue, we open at closed
  // On wed, we open at closed
  // On thu, we open at 12
  // On fri, we open at 11
  // On sat, we open at 0
  // On sun, we open at closed
```

```
// Methods
console.log(restaurant.order?.(0, 1) ?? 'Method does not exist');  // (2) ['Focaccia',
'Pasta']
console.log(restaurant.orderRisotto?.(0, 1) ?? 'Method does not exist');  // Method does not
exist

// Arrays
const users = [{ name: 'Jonas', email: 'hello@jonas.io' }];
// const users = [];

console.log(users[0]?.name ?? 'User array empty');  // Jonas

if (users.length > 0) console.log(users[0].name);
else console.log('user array empty');
  // Jonas
```

Looping objects

```
'use strict';

const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];

const openingHours = {

   // ES6 enhanced expression in property

  [weekdays[3]]: {

    open: 12,

    close: 22,

  },

  [weekdays[4]]: {

    open: 11,

    close: 23,

  },

  [weekdays[5]]: {

    open: 0, // Open 24 hours

    close: 24,

  },

};



const restaurant = {

  name: 'Classico Italiano',

  location: 'Via Angelo Tavanti 23, Firenze, Italy',

  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
```

```
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],

  mainMenu: ['Pizza', 'Pasta', 'Risotto'],



  // ES6 enhanced object literals

  openingHours,

 // ES6 enhanced function

  order(starterIndex, mainIndex) {

    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];

  },



  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {

    console.log(

      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`

    );

  },



  orderPasta(ing1, ing2, ing3) {

    console.log(

      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`

    );

  },
```

```
  orderPizza(mainIngredient, ...otherIngredients) {

    console.log(mainIngredient);

    console.log(otherIngredients);

  },

};



// Looping Objects: Object Keys, Values, and Entries



// Property NAMES

const properties = Object.keys(openingHours);

console.log(properties);  // (3) ['thu', 'fri', 'sat']0: "thu"1: "fri"2: "sat"length: 3[[Prototype]]: Array(0)



let openStr = `We are open on ${properties.length} days: `;

for (const day of properties) {

  openStr += `${day}, `;

}

console.log(openStr);  // We are open on 3 days: thu, fri, sat,



// Property VALUES

const values = Object.values(openingHours);

console.log(values);  // (3) [{…}, {…}, {…}]0: {open: 12, close: 22}1: {open: 11, close: 23}2: {open: 0, close: 24}length: 3[[Prototype]]: Array(0)
```

```
// Entire object

const entries = Object.entries(openingHours);

// console.log(entries);



// [key, value]

for (const [day, { open, close }] of entries) {

  console.log(`On ${day} we open at ${open} and close at ${close}`);

}

//output goes here...../

// On thu we open at 12 and close at 22

// On fri we open at 11 and close at 23

// On sat we open at 0 and close at 24
```

Coding challenge

//////////////////////////////////////

// Coding Challenge #2

/*
Let's continue with our football betting app!

1. Loop over the game.scored array and print each player name to the console, along with the goal number (Example: "Goal 1: Lewandowski")
2. Use a loop to calculate the average odd and log it to the console (We already studied how to calculate averages, you can go check if you don't remember)
3. Print the 3 odds to the console, but in a nice formatted way, exaclty like this:
    Odd of victory Bayern Munich: 1.33
    Odd of draw: 3.25
    Odd of victory Borrussia Dortmund: 6.5

Get the team names directly from the game object, don't hardcode them (except for "draw"). HINT: Note how the odds and the game objects have the same property names 😉

BONUS: Create an object called 'scorers' which contains the names of the players who scored as properties, and the number of goals as the value. In this game, it will look like this:

```
{
  Gnarby: 1,
  Hummels: 1,
  Lewandowski: 2
}
```

GOOD LUCK 😀

```
const game = {
  team1: 'Bayern Munich',
  team2: 'Borrussia Dortmund',
  players: [
    [
      'Neuer',
      'Pavard',
      'Martinez',
      'Alaba',
      'Davies',
      'Kimmich',
      'Goretzka',
      'Coman',
      'Muller',
      'Gnarby',
      'Lewandowski',
    ],
    [
      'Burki',
      'Schulz',
      'Hummels',
      'Akanji',
      'Hakimi',
      'Weigl',
      'Witsel',
      'Hazard',
      'Brandt',
      'Sancho',
      'Gotze',
    ],
  ],
  score: '4:0',
  scored: ['Lewandowski', 'Gnarby', 'Lewandowski', 'Hummels'],
  date: 'Nov 9th, 2037',
  odds: {
    team1: 1.33,
    x: 3.25,
    team2: 6.5,
  },
```

};

Sets

```
'use strict';
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
   // ES6 enhanced expression in property
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};

const restaurant = {
  name: 'Classico Italiano',
  location: 'Via Angelo Tavanti 23, Firenze, Italy',
  categories: ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'],
  starterMenu: ['Focaccia', 'Bruschetta', 'Garlic Bread', 'Caprese Salad'],
  mainMenu: ['Pizza', 'Pasta', 'Risotto'],

  // ES6 enhanced object literals
  openingHours,
 // ES6 enhanced function
  order(starterIndex, mainIndex) {
    return [this.starterMenu[starterIndex], this.mainMenu[mainIndex]];
  },

  orderDelivery({ starterIndex = 1, mainIndex = 0, time = '20:00', address }) {
    console.log(
      `Order received! ${this.starterMenu[starterIndex]} and ${this.mainMenu[mainIndex]} will
be delivered to ${address} at ${time}`
    );
  },

  orderPasta(ing1, ing2, ing3) {
    console.log(
```

```
      `Here is your declicious pasta with ${ing1}, ${ing2} and ${ing3}`
    );
  },

  orderPizza(mainIngredient, ...otherIngredients) {
    console.log(mainIngredient);
    console.log(otherIngredients);
  },
};

// Sets
const ordersSet = new Set([
  'Pasta',
  'Pizza',
  'Pizza',
  'Risotto',
  'Pasta',
  'Pizza',
]);
console.log(ordersSet);  //  Set(3) {'Pasta', 'Pizza', 'Risotto'}

console.log(new Set('Jonas'));  //  Set(5) {'J', 'o', 'n', 'a', 's'}

console.log(ordersSet.size);  //  3
console.log(ordersSet.has('Pizza'));  //  true
console.log(ordersSet.has('Bread'));  //  false
ordersSet.add('Garlic Bread');
ordersSet.add('Garlic Bread');
ordersSet.delete('Risotto');
// ordersSet.clear();
console.log(ordersSet);  //  Set(3) {'Pasta', 'Pizza', 'Garlic Bread'}

for (const order of ordersSet) console.log(order);
  //  output goes here///
  // Pasta
  // Pizza
  // Garlic Bread
// Example
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];
const staffUnique = [...new Set(staff)];
console.log(staffUnique);  //  (3) ['Waiter', 'Chef', 'Manager']

console.log(
  new Set(['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter']).size
);  //  3

console.log(new Set('jonasschmedtmann').size);  //  11
```

Map fundamentals

Map iteration