JavaScript

JavaScript is a high-level, object-oriented, multi-paradigm programming language.

- High-Level: We don't have to worry about complex stuff like memory management.
- Object-Oriented: Based on objects, for storing most kind of data
- Multi-paradigm: we can use different style of programming.

There is nothing we can't do with JS.

- Front-End Apps: Dynamic effect and web application for browser. E.g.: React, Angular
- Back-End Apps: Web application on servers. E.g.: Node Js
- Native Mobile Apps: We can develop mobile application too. E.g.: React Native
- Desktop Application: We can create desktop application too. E.g.: Ion, Atomic

JavaScript Releases:

ES5->ES6/ES2015->ES7/ES2016->ES8/ES2017->ES9/ES2018->ES10/ES2019->ES11/ES2020->…

ES6- Biggest Update to the language ever, and from ES6 Modern JS ERA Started.

JS DATA TYPES

Value->Object or Primitive Type(Everything Else).

7 Type of Primitive type:

1. Number: Floating Point Numbers. Used for decimal and integers. E.g.: let age = 23
2. String: Sequence of Characters. Used for text. E.g.: let firstName='Summy'
3. Boolean: Logical type that can only be true or false. Used for taking decision. E.g.: let fullAge=true;
4. Undefined: Value taken by variable that is yet to define.(empty value) E.g.: let children;
5. Null: Also means empty value
6. Symbol(ES2015): value that is unique and cannot be changed.
7. BigInt(ES2020): Larger Integers that the Number can hold.

JS has dynamic typing: We do not have to manually define the data type of the value stored in variable. Instead data type are determined automatically.

Boolean Logic:

1. AND- A AND B – true when all are true.
2. OR- A OR B – true when one is true.
3. NOT - !A- inverts true/false value.

Backward Compatibility: Don’t Break Web.

JS is designed in that way, even if we use modern JS it is backward compatible with old JS(1997) because then it won’t break any functionality and it would work fine. It is Backward Compatible.

- Old features are never removed
- Not really new version, just incremental version
- Websites keep working forever

```
let js = "amazing";
console.log(js)// amazing
console.log(40 + 8 + 23 - 10);


// Values and Variables
console.log("Jonas");
console.log(23);

let firstName = "Matilda";
console.log(firstName);        //matilda
console.log(firstName);        //matilda

let firstName="summy";
let lastName = "singh";
console.log("First name is: "+firstName);
console.log("Last Name is: "+lastName);
console.log("Full Name is: "+firstName +" "+lastName);

let jsBool = true;
console.log(jsBool);   //true

console.log(typeof(jsBool)); //boolean
console.log(typeof jsBool);  //boolean
console.log(typeof true);    //boolean
console.log(typeof 3);       //number
console.log(typeof(3));      //number
console.log(typeof NaN);    //number
console.log(typeof(NaN));   //number
console.log(typeof null);   //object
console.log(typeof(null));  //object
console.log(typeof string);     //undefined
console.log(typeof 'summy');    //string

let jsIs;
console.log(typeof '');          //string
console.log(typeof jsIs);       //undefined
console.log(typeof 3.14);       //number

let javascriptIsFun = 'YES!';
console.log(typeof javascriptIsFun);//string

// Basic Operators
// Math operators
```

```
const year=2020
const ageSummy = year-1996;
const ageSunny=year-1993;
console.log(ageSummy, ageSunny);

console.log(ageSummy*2, ageSummy+10, ageSummy/2, ageSummy%2);
console.log(ageSummy*2);
console.log(ageSummy+10);
console.log(ageSummy/2);
console.log(ageSummy%5);
console.log(2**3);//mean 2 to the power 3

//assignmnet operator

// let x=10+15;
// console.log(x);
// x+=10;
// console.log(x);
// x-=10;
// console.log(x);
// x*=10;
// console.log(x);
// x++;
// console.log(x);
// x--;
// console.log(x);
// ++x;
// console.log(x);
// --x;
// console.log(x)

//conparison operator >,<,>=,<= it will return true or false

// const now = 2020;
// const ageSummy = now-1996;
// const ageSunny = now-1994;
// console.log(now-1994>now-1996)       //true


// Operator Precedence

const now = 2037;
const ageJonas = now - 1991;
const ageSarah = now - 2018;
console.log(now - 1991 > now - 2018); //true

const averageAge = (ageJonas + ageSarah) / 2;
console.log(ageJonas, ageSarah, averageAge);  //46 19 32.5


let x, y;
x = y = 25 - 10 - 5; // x = y = 10, x = 10
console.log(x, y);


//string
```

```
const fName = 'summy';
const job = 'developer';
const birthYear = 1996;
const year = 2020;
const summy = "I'm "+fName+', a '+(year-birthYear)+' Year old '+job+' !';
console.log(summy);

//template literal
const summyNew = `I'm ${fName}, a ${year-birthYear} years old ${job}!`
console.log(summyNew);

console.log(`just a regular string`);

console.log('string \n\ with \n\ multiple lines');

console.log(`string
with multiple line
using literals`);

// Taking Decisions: if / else Statements

if (age >= 18) {
  console.log('Sarah can start driving license 🚗');
} else {
  const yearsLeft = 18 - age;
  console.log(`Sarah is too young. Wait another ${yearsLeft} years :)`);
}

const birthYear = 2012;

let century;
if (birthYear <= 2000) {
  century = 20;
} else {
  century = 21;
}
console.log(century);

// Type Conversion and Coercion

Type Conversion: Type conversion is when we manually convert one type to
another.

Type Coersion: Type coercion is when JS automatically convert one type to
another behind the scene for us.

// type conversion
const inputYear = '1991';
console.log(Number(inputYear), inputYear);    //1991(n),1991(s)
console.log(Number(inputYear) + 18);          //2009
console.log(+inputYear);                      //1991(number)

console.log(Number('Jonas'));                 //NaN
console.log(typeof NaN);                      //number
console.log(String(23), 23);                  //23(n), 23(s)
```

```
//type coercion
console.log('I am '+23+' Years old');
//whenever js sees a + operator it automatically convert number to string
console.log('23'-'10'-3);      //3            //string to number
console.log('23'-'10');        //13
console.log('23'+'10'+3);      //23103        //number to string
console.log('13'*'2');         //26

//5 falsy value---0,'',undefined,null,Nan
console.log(Boolean(0));                      //false
console.log(Boolean(''));                     //false
console.log(Boolean(undefined));              //false
console.log(Boolean(null));                   //false
console.log(Boolean(NaN));                    //false

console.log(Boolean({}));//empty object it is //true
console.log(Boolean(2));                      //true


//if else scenerio for falsy value
const money=0;
if(money){
        console.log("Don't spend all")
}else{
        console.log("Get a Job!");
}//return false because of 0

let height;                    //height is undefined here and it is falsy
if(height){
     console.log("It's true that you have height");
}else{
     console.log("Stop using Undefined, please define your height");
}

// Equality Operators: == vs. ===

= = = : Strict equality operator, it's strict because it does not perform
type coercion. It returns true when both values are exactly the same.

= = : Loose Equality Operator, actually does type coercion.
e.g. '18'==18 -> true.

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

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

const favourite = Number(prompt("What's your favourite number?"));
console.log(favourite);
console.log(typeof favourite);

if (favourite === 23) { // 22 === 23 -> FALSE
  console.log('Cool! 23 is an amzaing number!')
} else if (favourite === 7) {
  console.log('7 is also a cool number')
} else if (favourite === 9) {
  console.log('9 is also a cool number')
```

```
} else {
  console.log('Number is not 23 or 7 or 9')
}

if (favourite !== 23) console.log('Why not 23?');


//PROMPT- takes input from the browser and it return string by default.
const fav =  prompt("what is your fav number?");
console.log(fav);

//logical Operator
const hasDriversLicense = true; //A
const hasGoodVision = false;  //B

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

//switch case
const day = 'saturday';

switch(day){
     case 'monday':
         console.log("It's Monday");
         break;
     case 'tuesday':
         console.log("It's Tuesday");
         break;
     case 'wednesday':
         console.log("It's Wednesday");
         break;
     case 'thursday':
         console.log("It's Thursday");
         break;
     case 'friday':
         console.log("It's Friday");
         break;
     case 'saturday':
     case 'sunday':
         console.log("It's Weekend");
         break;
     default:
         console.log('Take a break!');
 }

//ternary operator

const age = 23;
age>=18?console.log("adult man"):console.log("you are a minor");

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

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

// **Statements and Expressions**

- An **Expression** is a piece of code that produces a value e.g. 3+4(Produces a value).
- An **Statements** is like a bigger piece of code that is executed and which does not produce a value on itself.(Are like full sentences, that translate our actions).

Declarations are like statements, and expressions are like the words.

```
3 + 4
1991
true && false && !false

if (23 > 10) {
  const str = '23 is bigger';
}

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

Let, Var and Const

1. **Let**: let age: number or let age=30 -> age=31
   - We can reassign a value to a variable, also we can say that we can mutate the age variable in this case.
   - When we need that kind of variable, then this is the perfect use case for let.
   - For empty variable also.
2. **Const**: To declare a variable that are not supposed to change the value. i.e. value can't be changed.
   - By default, always just use const, and let when you are sure that the value of the variable need to change at some point of the code.
3. **Var**: var job = 'Programmer';-> job='Developer'[Before ES6]

➢ Let is block-scoped and Var is function-scoped.

Strict Mode : It is a special mode which we can activate in JS, which makes it easier for us to write secure code.

'use strict' at the top of the JS file.
-to avoid accidental errors
-forbid us to do certain things
-create visible errors

Function: A piece of code, that we can use again and again according to the requirement.

- A variable can hold a value, but a function can hold one or more complete lines of code.
- We pass data to function and additionally, a function can return data as well
- Parameter are like variable that are specific only to the function and they will get defined once we call the function.
- We can think of these parameter of empty spaces that still need to be fill out when we are writing the function.

- And, when we call the function, we then fill these empty spaces by passing the real specific value. Which will then get assigned to the parameters.
- **If a function does not produce/return value, technically it does produce undefined. We don't capture undefined value**
- Parameter is like a local variable, that is only available inside the function.
- Parameter is the kind of placeholder in the function.
- Arguments is the actual value that we use to fill in the placeholder.

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

// calling / running / invoking function
logger();
logger();
logger();

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

const appleJuice = fruitProcessor(5, 0);
console.log(appleJuice);

const appleOrangeJuice = fruitProcessor(2, 4);
console.log(appleOrangeJuice);
```

Function Expression: Also called an anonymous function.

```
// Function Declarations vs. Expressions

// Function declaration
function calcAge1(birthYeah) {
  return 2037 - birthYeah;
}
const age1 = calcAge1(1991);

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

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

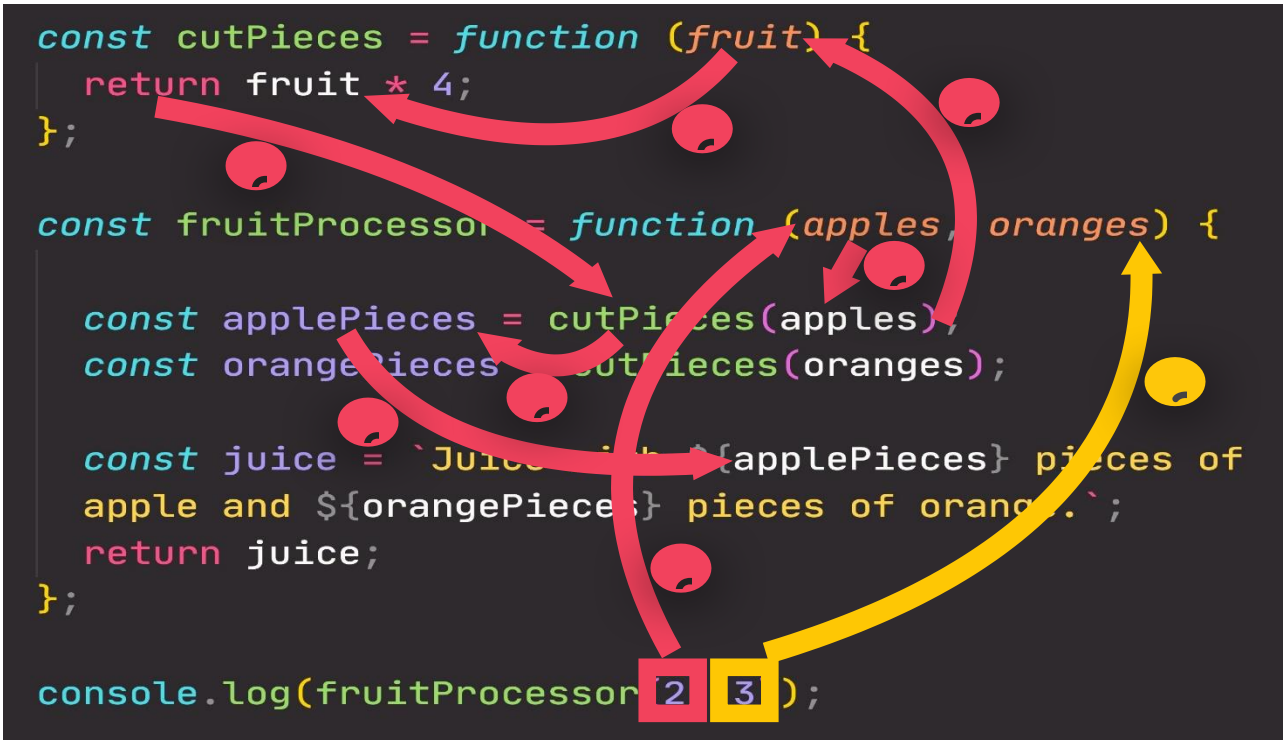
- And expression produces values, so we just assigned this whole expression to this variable, will then hold this function value basically.
- In JS, functions are just values.
- Function is not a type like number, string type. It is a value and we can store it in a variable.

Main difference between function declaration and expression is we can actually call function declaration before they are defined in the code. i.e. Function Hoisting.

Arrow Function(ES6): Arrow function is simply a special form of function expression that is shorter and therefore faster to write.

- We loose advantage of arrow function with complex code.
- Arrow function do not get so-called **this Keyword.**

```
// Arrow functions

const calcAge3 = birthYeah => 2037 - birthYeah;
const age3 = calcAge3(1991);
console.log(age3);

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'));
console.log(yearsUntilRetirement(1980, 'Bob'));


// 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));
```



FUNCTIONS REVIEW; 3 DIFFERENT FUNCTION TYPES

- **Function declaration**

  Function that can be used before it's declared

- **Function expression**

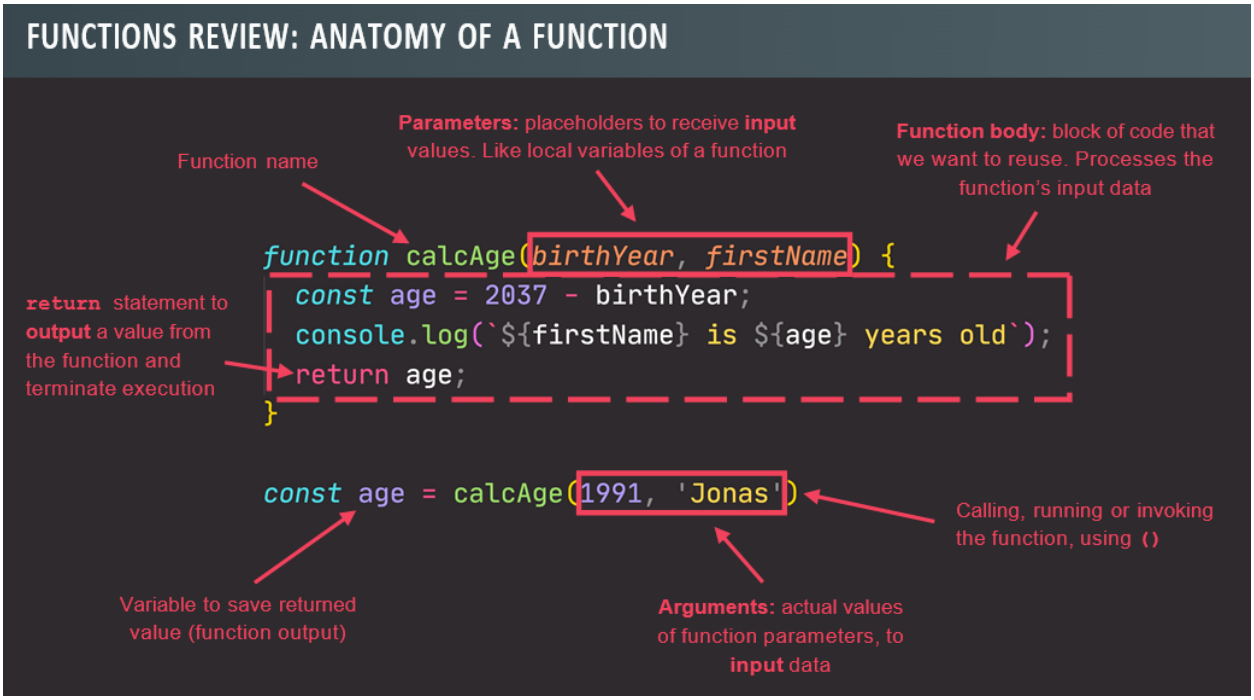
  Essentially a function *value* stored in a variable

- **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.



Array: We can change array even if it is declared as const.
- Because only primitive values are immutable. But array is not a primitive value and change it and that is why it is immutable.
- We cannot replace the entire array.

```
//Arrays
// const friends = ['summy','ranjeet','shubham','vishal','vicky']; //literal syntax to create array
// console.log(friends);

// const years = new Array(2018,2019,2020,2021,2022,2023); //array function to create array
// console.log(years);

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

const friends = ['Michael', 'Steven', 'Peter'];
console.log(friends);

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

console.log(friends[0]);
console.log(friends[2]);

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

friends[2] = 'Jay';
console.log(friends);                           //micheal,steven,jay
// friends = ['Bob', 'Alice']
```

```
const firstName = 'Jonas';
const jonas = [firstName, 'Schmedtmann', 2037 - 1991, 'teacher', friends];
console.log(jonas);
console.log(jonas.length);
```

Array Methods:

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

// Add elements
const newLength = friends.push('Jay');               //Add to the end
console.log(friends);
console.log(newLength);

friends.unshift('John');                             //Add to the start
console.log(friends);

// Remove elements
friends.pop();                                       // Remove from Last
const popped = friends.pop();
console.log(popped);
console.log(friends);

friends.shift();                                     // Remove from First
console.log(friends);


console.log(friends.indexOf('Steven'));       //Gives the index of element
console.log(friends.indexOf('Bob'));

friends.push(23);                      // check if element exist in the array
console.log(friends.includes('Steven'));
console.log(friends.includes('Bob'));
console.log(friends.includes(23));
```

Includes(ES6): Instead of returning the index of the element we simply return true if the element is in the array, and false if not.

- It check strict equality and it does not do type coercion.

Objects: Key-Value Pair
-Key is also called properties
-Order does not matter in object, to retrieve them unlike array
-Properties are ordered alphabetically
-Can retrieve data using two way:
1.Dot Notation
2.Bracket Notation- We can put expression and compute it in this.

```
// Introduction to Objects

const jonasArray = [
  'Jonas',
  'Schmedtmann',
  2037 - 1991,
  'teacher',
```

```
  ['Michael', 'Peter', 'Steven']
];
//cannot provide fname, lname,job and all details in array and that is why
object come and help
Console.log(jonasArray);

//object literal syntax and easiest way to write

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

Key is also called Properties, so we can say this object jonas have 5
properties.

// Dot vs. Bracket Notation

/dot notation
console.log(jonas.age);
console.log(jonas.fName);

//bracket notation
console.log(jonas ['lName']);

jonas.zodiac = 'Taurus';
jonas ['passion']='photography';
console.log(jonas);

// 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["calcAge"]())       //46
console.log(jonas.age);               //46
```

Loops: Loop are a fundamental aspect of every programming language because they allow us to automate repetitive task.

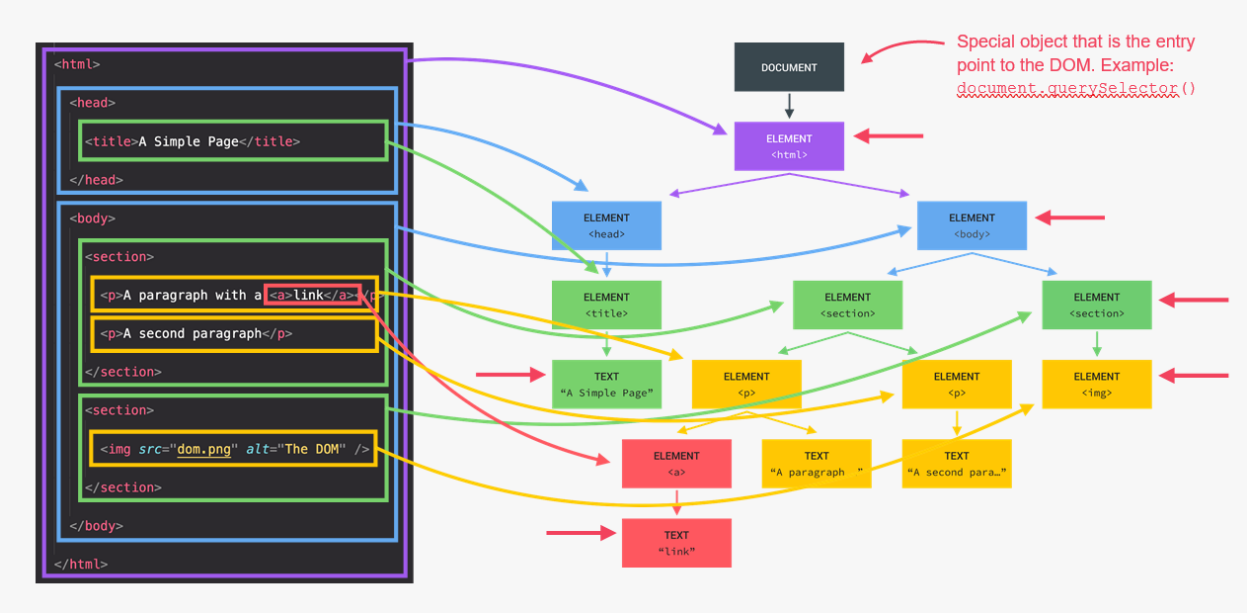
-**Continue**: Continue is to exit the current iteration of the loop and continue to the next one.
-**Break**: Break is used to completely terminate the whole loop.

```
//For Loop
1.for(let rep=1;rep<=15;rep++){
    console.log('Rep No: '+rep);
}

2.for(let rep=1;rep<=15;rep++){
     console.log(`Rep No: ${rep}`);
}

3. const summyArray = [
    'summy',
    'singh',
    1996,
    'software developer',
    2020-1996
];

const types=[];

for(let i=0;i<summyArray.length;i++){
    //reading data from summyArray
    console.log(summyArray[i]);

    //Reading data and type
    //console.log(summyArray[i], typeof summyArray[i]);

    //filling type array
    //types[i]=typeof summyArray[i];
    types.push(typeof summyArray[i]);
}
//console.log(types);

//continue and break

//Continue
// console.log("-----Only Strings------")
// for(let i=0;i<summyArray.length;i++){
//     if(typeof summyArray[i] !== 'string'){
//         continue;
//         //console.log(summyArray[i], typeof summyArray[i])
//     }
```

```
//     console.log(summyArray[i], typeof summyArray[i])
// }

//Break
// console.log("-----Break with Number------")
// for(let i=0;i<summyArray.length;i++){
//     if(typeof summyArray[i] === 'number'){
//         break;
//         //console.log(summyArray[i], typeof summyArray[i])
//     }
//     console.log(summyArray[i], typeof summyArray[i])
// }

//looping backward
// const summyArray = [
//     'summy',
//     'singh',
//     1996,
//     'software developer',
//     2020-1996
// ];

// for(let i = summyArray.length-1;i>=0;i--){
//     console.log(i,summyArray[i]);
// }

//While loop

// let rep=0;
// while(rep<=10){
//     console.log(`Rep No: ${rep}`);
//     rep++;
// }
```

DOM: Document Object Model: Structured representation of HTML Documents. Allows JS to access html elements and styles to manipulate them.

Tree structure, generated by browser on HTML Load.



-Makes JS Interact with Web Page is DOM Manipulation.

What is DOM?

Structured representation of html document. Allows JS to access html element and style to manipulate them.

-The DOM is automatically created by the browser as soon as the HTML Page loads and it stored in a tree structure.

-DOM is basically a connection point between HTML document and JS Code.

-Everything in DOM structure, every element has a node in tree, and we can access and interact with each of these nodes using JS.

-DOM stands with document object, and document is a special object we have access in JS. And this special object serves as a entry point in the DOM.

-DOM is not part of JS language, actually DOM & it's method(querySelector)are actually part of something called the WebAPIs.

-So the WebAPIs are like libraries that browser implement and that we can access from our JS Code.

```
console.log(document.querySelector('.message').textContent);
document.querySelector('.message').textContent = '🎉 Correct Number!';

document.querySelector('.number').textContent = 13;
document.querySelector('.score').textContent = 10;

document.querySelector('.guess').value = 23;
console.log(document.querySelector('.guess').value);

document.querySelector('.check').addEventListener('click', function () {
  const guess = Number(document.querySelector('.guess').value);
  console.log(guess, typeof guess);

const displayMessage = function (message) {
  document.querySelector('.message').textContent = message;
};

document.querySelector('body').style.backgroundColor = '#60b347';
document.querySelector('.number').style.width = '30rem';
```

Modal

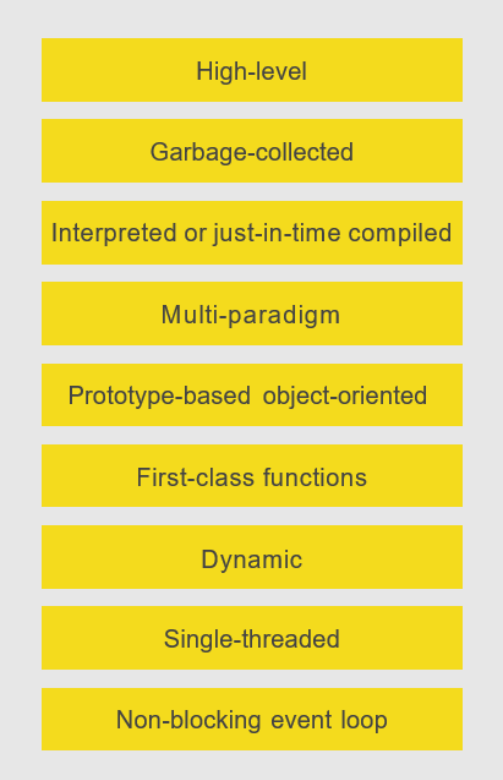
```
const modal = document.querySelector('.modal');
const overlay = document.querySelector('.overlay');
const btnCloseModal = document.querySelector('.close-modal');
const btnsOpenModal = document.querySelectorAll('.show-modal');

const openModal = function () {
  modal.classList.remove('hidden');
  overlay.classList.remove('hidden');
};

const closeModal = function () {
  modal.classList.add('hidden');
  overlay.classList.add('hidden');
};

for (let i = 0; i < btnsOpenModal.length; i++)
  btnsOpenModal[i].addEventListener('click', openModal);

btnCloseModal.addEventListener('click', closeModal);
overlay.addEventListener('click', closeModal);

document.addEventListener('keydown', function (e) {
  // console.log(e.key);

  if (e.key === 'Escape' && !modal.classList.contains('hidden')) {
    closeModal();
  }
});
```

```
document.getElementById(`current--${activePlayer}`).textContent = 0;

const switchPlayer = function () {
  document.getElementById(`current--${activePlayer}`).textContent = 0;
  currentScore = 0;
  activePlayer = activePlayer === 0 ? 1 : 0;
  player0El.classList.toggle('player--active');
  player1El.classList.toggle('player--active');
};
```

High-Level Overview of JavaScript

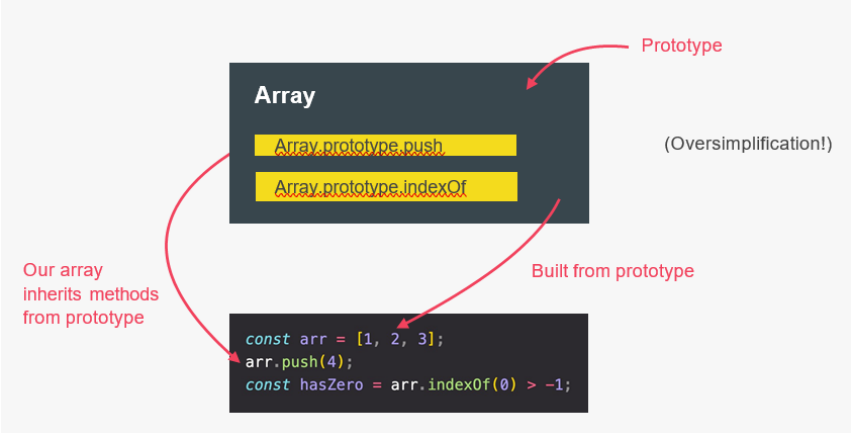
-It is a high level, object-oriented, Multi-Paradigm Programming Language.

-JavaScript is a high-level, Prototype-Based Object-Oriented, Multi-Paradigm, Interpreted or Just-In-Time Complied, Dynamic, Single Threaded, Garbage-Collected Programming language with first-class function and a non-blocking event loop concurrency model.



1. **High-Level**: Developer does not have to worry, everything happens automatically.
2. **Garbage Collected**: Cleaning the memory, so we don't have to.
3. **Interpreted or JIT**: Convert the code into machine by compiling.
4. **Paradigm**: An approach and mindset of structuring code, which will direct your coding style and technique.
   -3 Popular Paradigm are:
   - Procedural Programming
   - Object Oriented Programming
   - Functional Programming

   -Imperative and Declarative
5. **Prototype-Based Object-Oriented:**

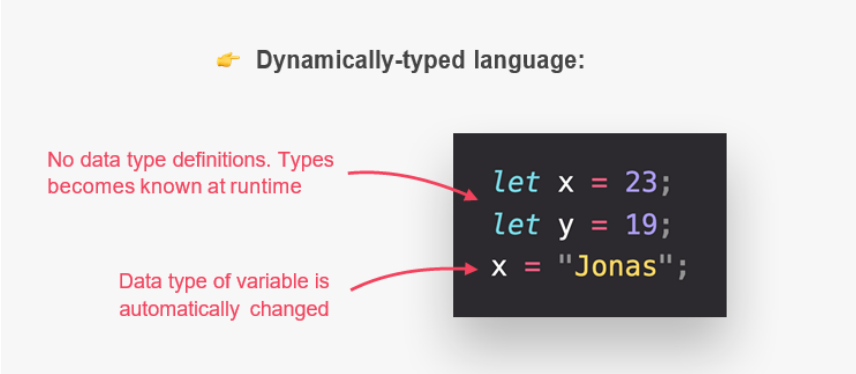


6. **First-Class Function:** Function are simply treated as variable. We can pass them into other function and return them from function.

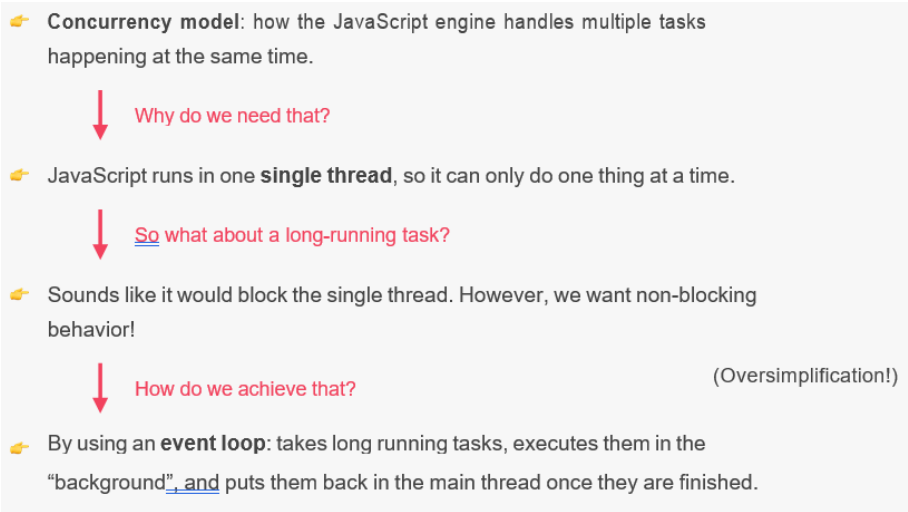
```
const closeModal = () => {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};

overlay.addEventListener("click", closeModal);
```

Passing a function into another function as an argument: First-class functions!

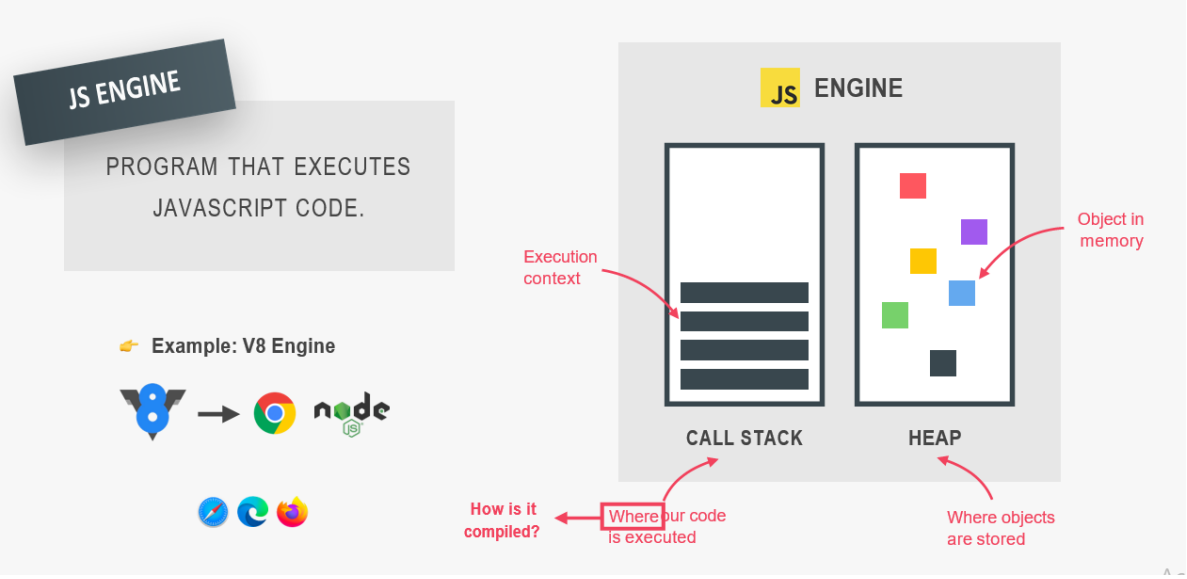
7. **Dynamic-Typed Function:**



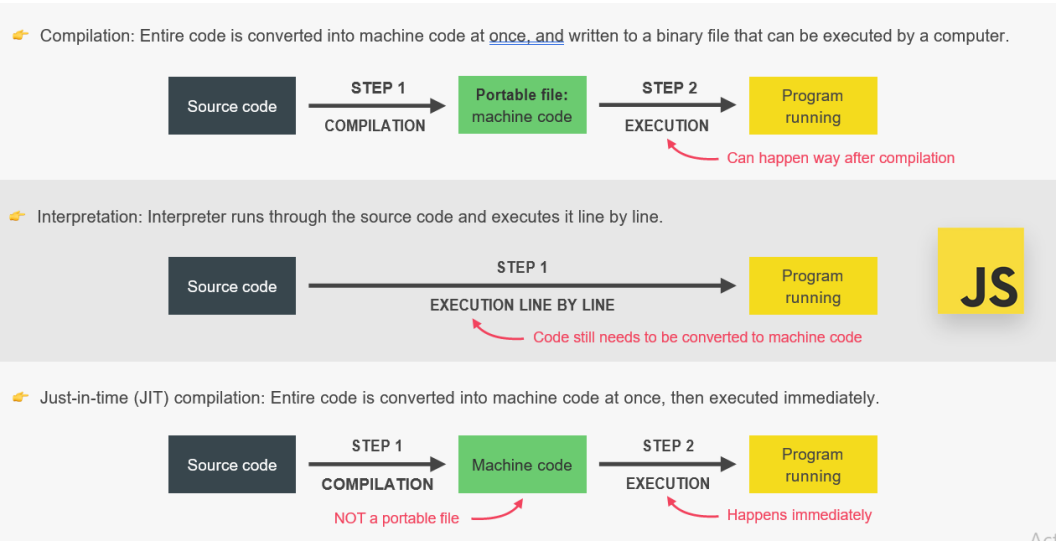
8. **Concurrency Model:**



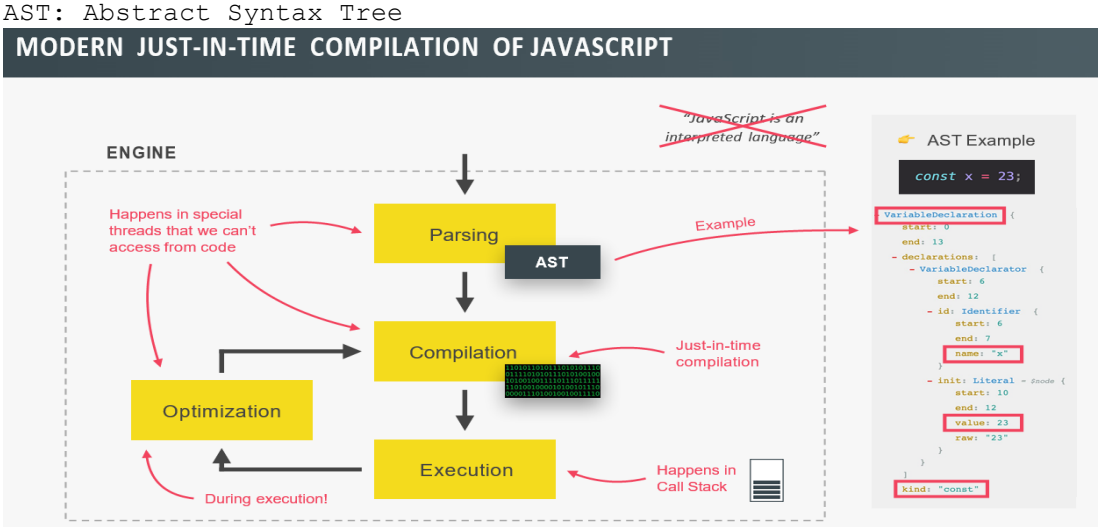
JavaScript Engine & Runtime:



Every Single computer program ultimately needs to be converted into this machine code, and this can happen using compilation or interpretation.



AST: Abstract Syntax Tree



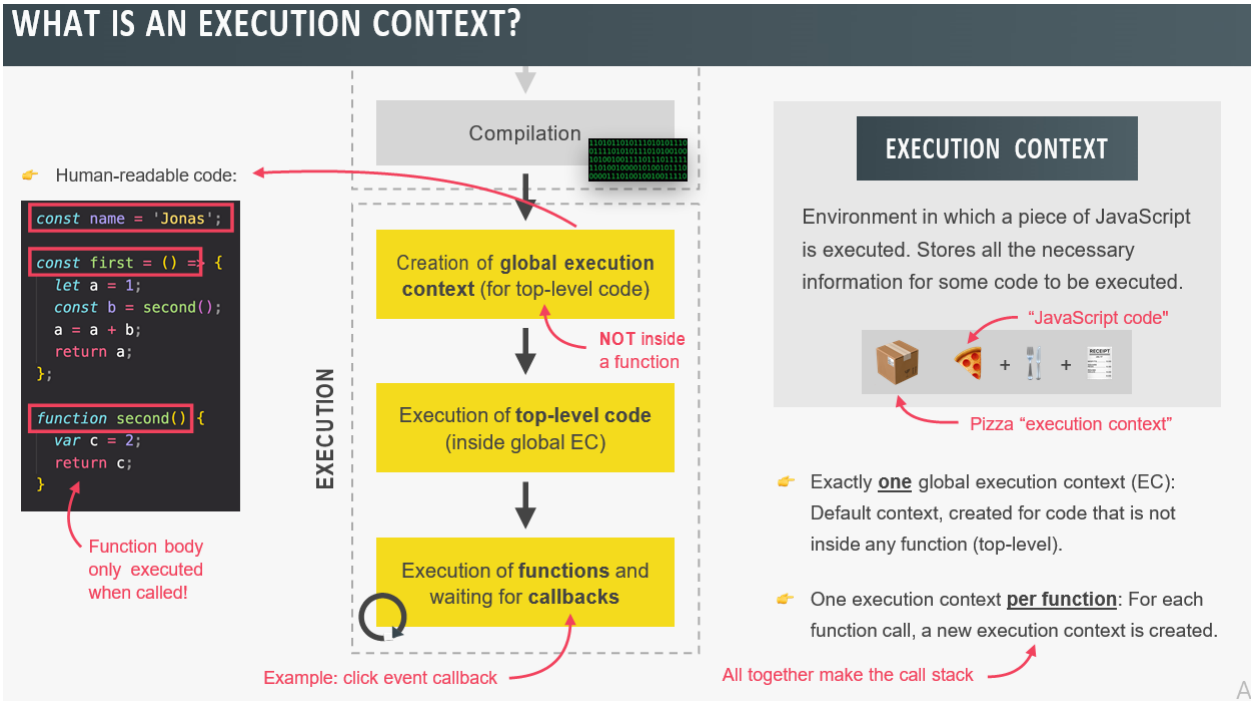
Execution Context and the Call Stack

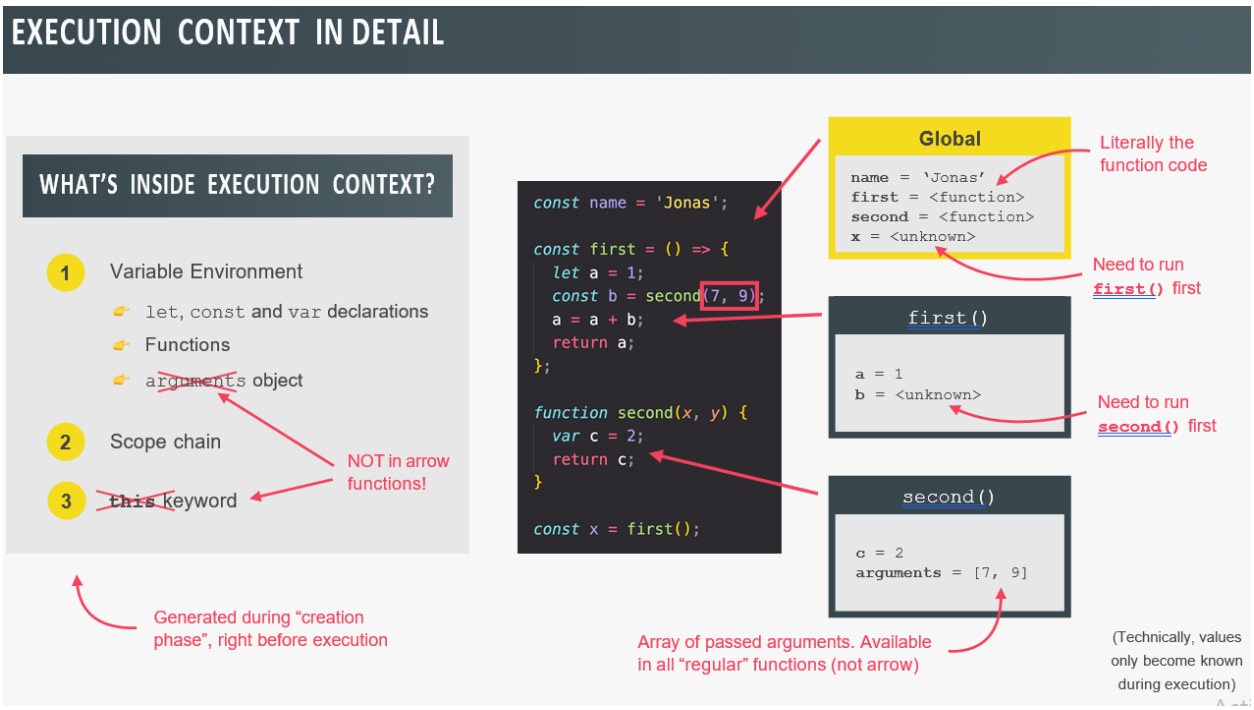
Execution Context: Environment in which a piece of JS is executed. Stores all the necessary information for some code to be executed.

-Exactly one global execution context(EC): Default context, created for code that is not inside any function.

-One execution context per function: For each function call, a new execution context is created.

-All together make the call stack.



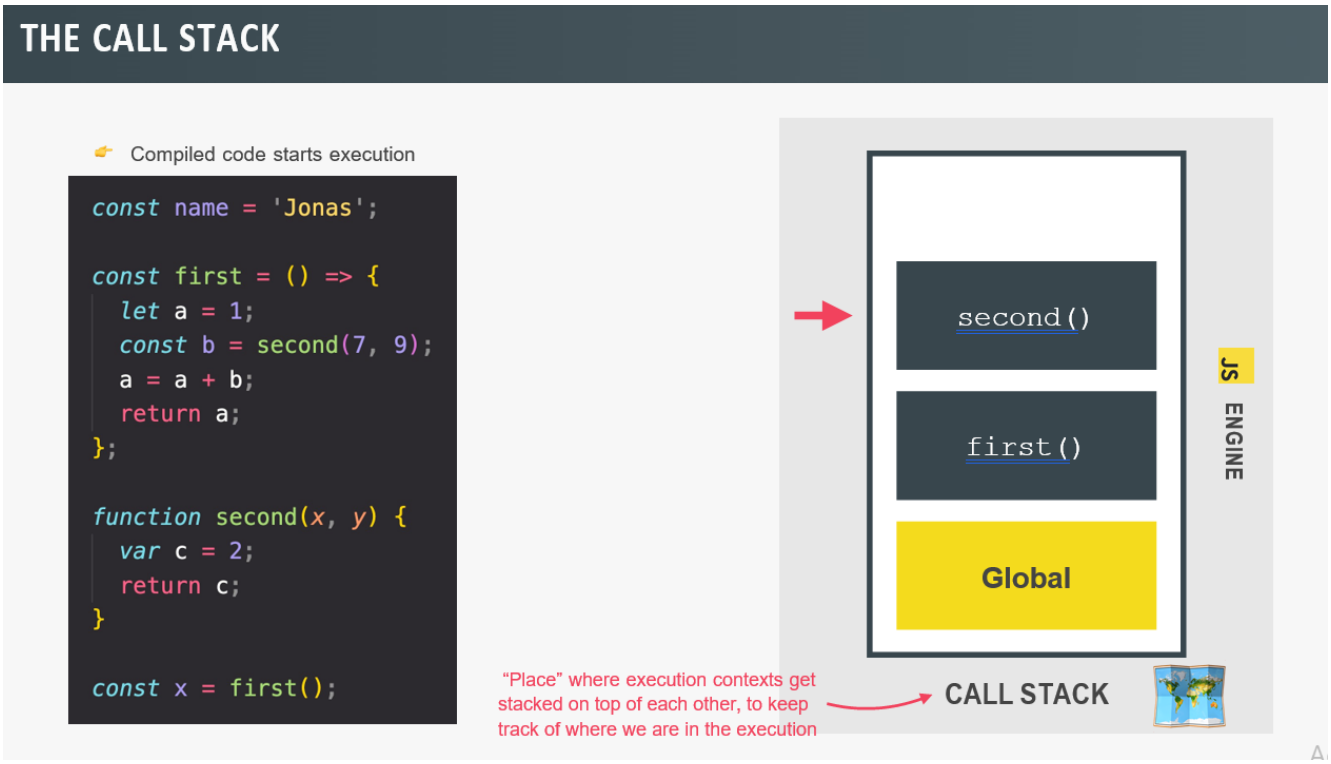


What's inside Execution Context?

1. Variable Environment
2. Scope Chain
3. This Keyword

All are generated during 'execution phase' right before execution.

->**Arrow function do not have argument object or this keyword**



Scoping and Scope in JS

- **Scoping:** How our program's variable are accessed and organized. "Where do variable lives"?
- **Lexical Scoping:** Scoping is controlled by placement of function and blocks in the code.
- **Scope:** Space or environment in which certain variable is declared. There is Global Scope, Functional Scope and Block Scope.
- **Scope of 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)

Act

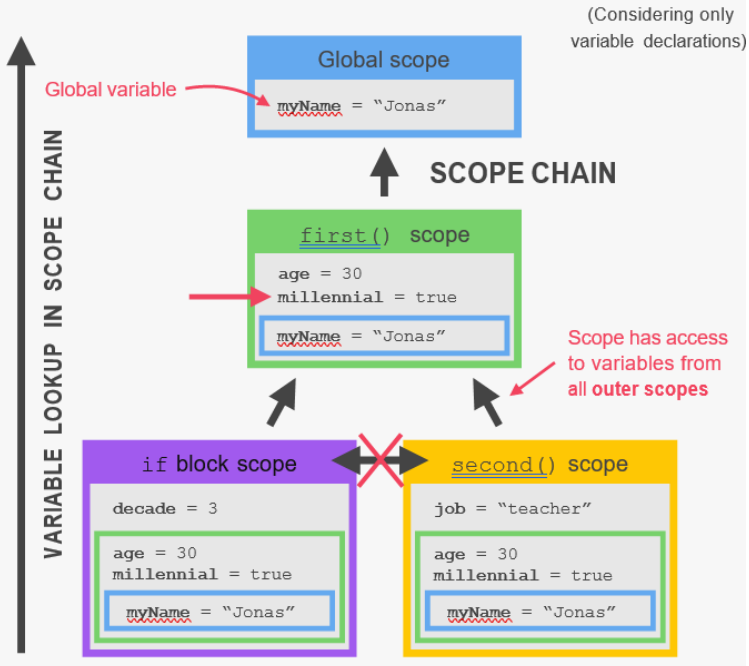
THE SCOPE CHAIN

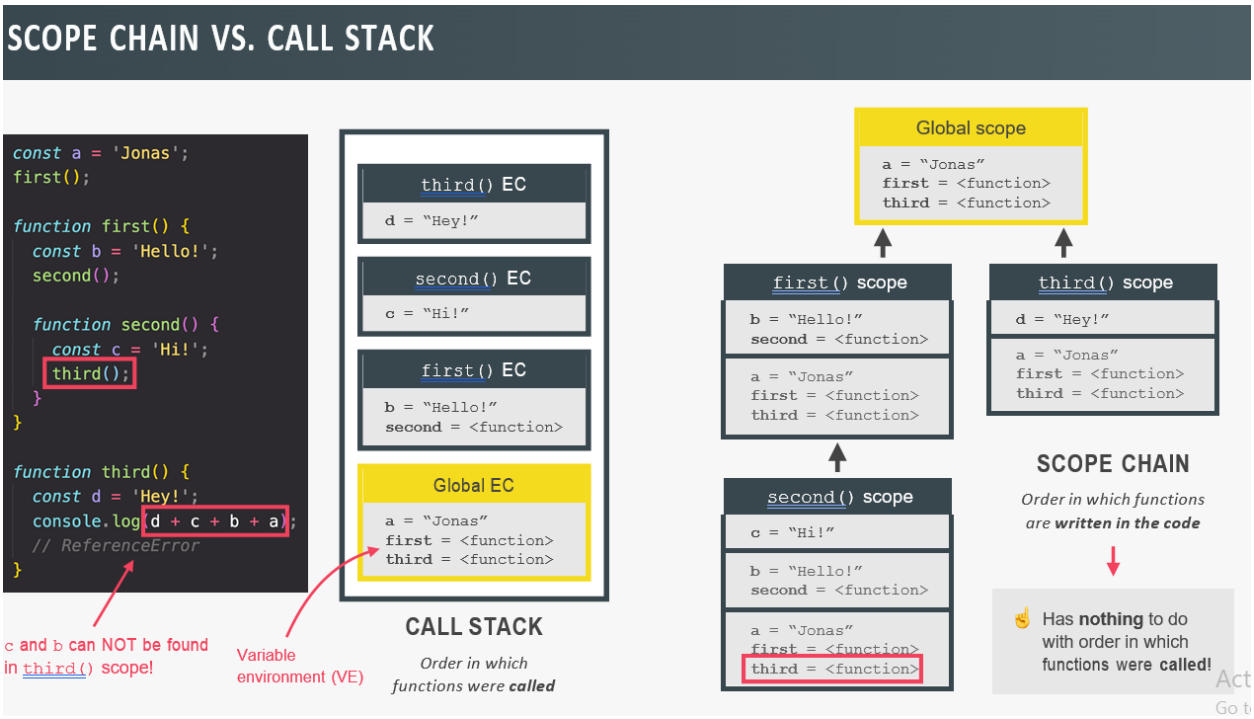
```
const myName = 'Jonas';

function first() {
  const age = 30;

  if (age >= 30) {  // true
    const decade = 3;
    var millenial = true;
  }

  function second() {
    const job = 'teacher';

    console.log(`${myName} is a ${age}-old ${job}`);
    // Jonas is a 30-old teacher
  }

  second();
}

first();
```

let and const are block-scoped

var is function-scoped

Variables not in current scope



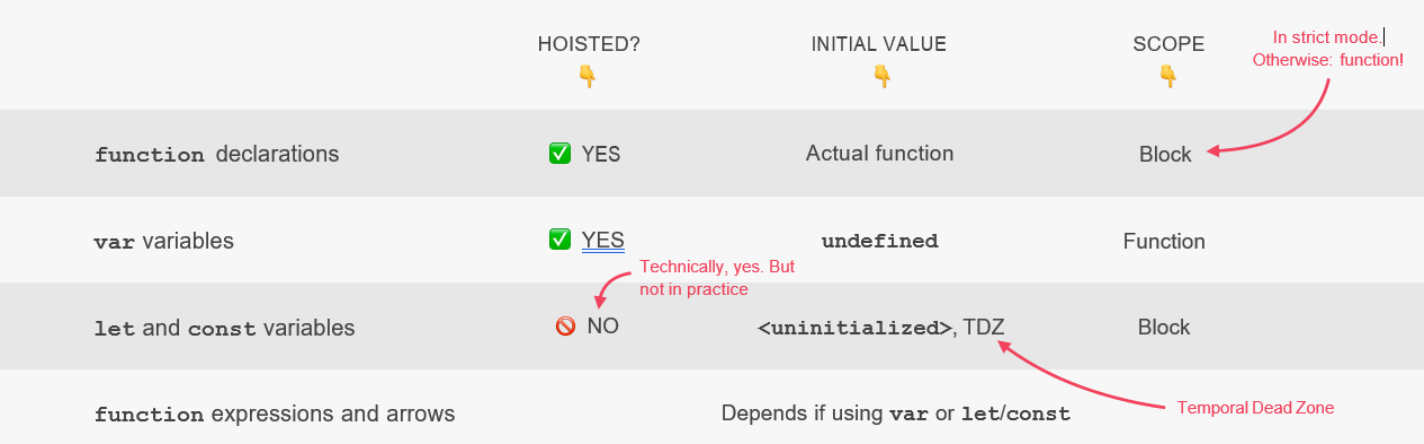


Summary:

- Only **let** and **const** variable are block-scoped. Variable declared with **Var** are end up or available for the closest function call.
- In JS, we have lexical scoping, so the rule of where we can access variables are based on exactly where in the code function and blocks are written.
- Every scope has access to all the variable from all its outer scope. This is the scope chain.
- When a variable is not in the current scope, the engine looks up in the scope until it finds the variable it is looking for. This is called **Variable Lookup**.
- The scope chain is a one way street: A scope will never, ever have access to the variable of an inner scope.
- The scope chain in a certain scope is equal to adding together all the variable environment of all the parent scope.
- The scope chain has nothing to do with the order in which function were called. It does not affect the scope chain at all.
- Variable declared with **var** keyword are function scoped.

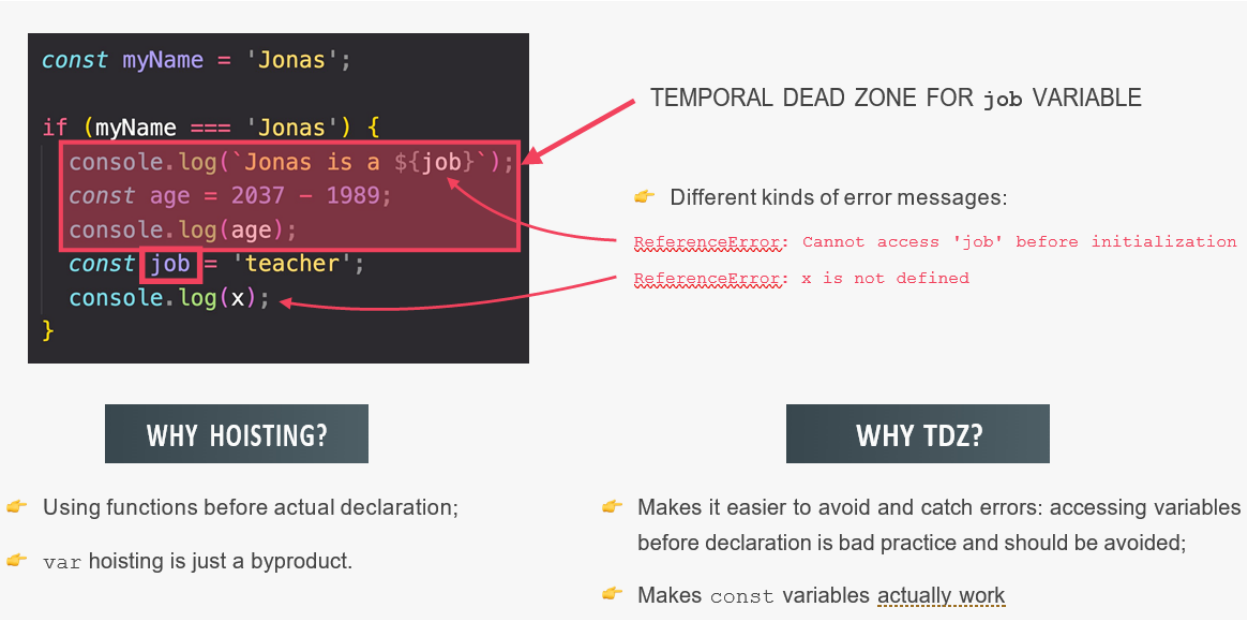
Hoisting: Makes some types of variable accessible/usable in the code before they are actually declared. "Variable lifted to top of their scope".

-(BTS)Before execution, code is scanned for variable declaration, and for each variable, a new property is created in the variable environment object.



Temporal Dead Zone(TDZ): Temporal Dead Zone, which makes it so that we can't access the variable between the beginning of the scope and the place where **var** is declared.

Why TDZ?
-Makes it easier to avoid and catch error: Accessing variable before declaration is bad practice and should be avoided.
-Makes const variable actually work.



Difference between let, var and const
The variable declared with **var** keyword creates windows properties while declared with **let & const** will not create window properties.

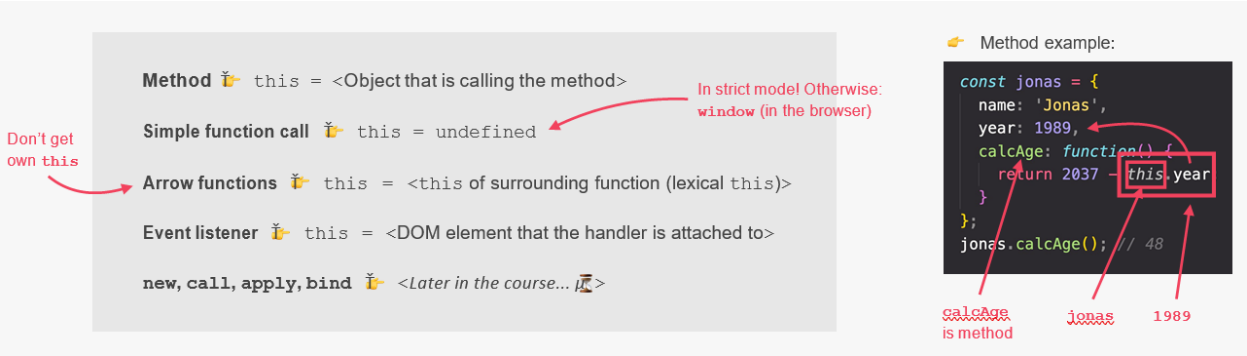
This Keyword: Special variable that is created for every execution context(every function).

-Takes the value of(point to) the "owner" of the function in which this keyword is used.

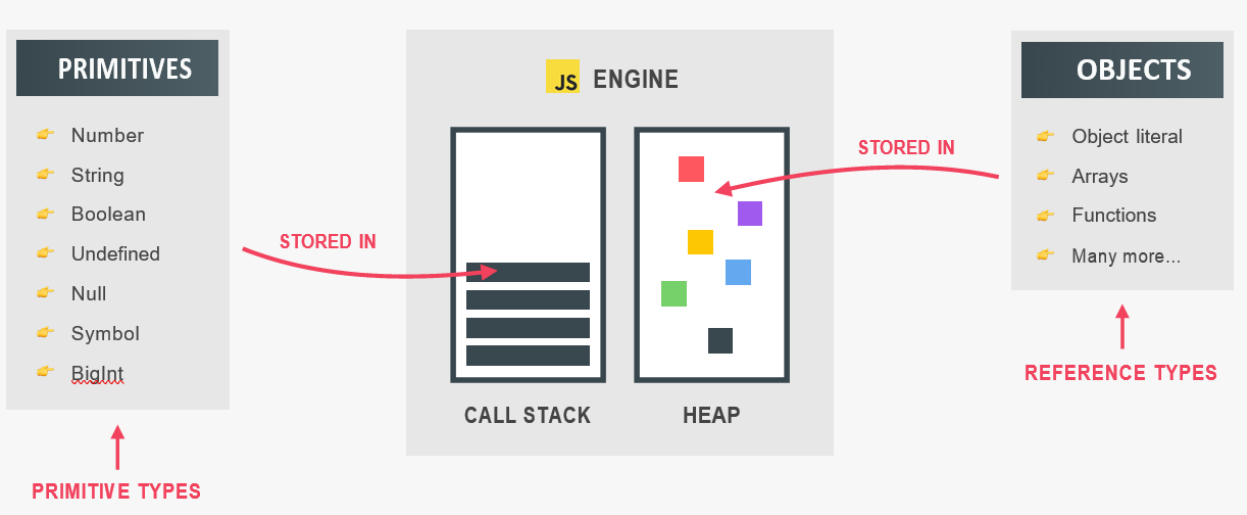
-this is not static. It depends on how the function is called, and it's value is only assigned when the function is actually called.

-Arrow function do not get their own 'this' keyword, it will point to the parent function/surrounding function.

-this does not point to the function itself, and also NOT the its variable environment.



Primitives vs Object(Primitive vs Reference types)



-Object might be too large to stored in stack, instead they store in heap, and give their memory address in stack to variable to point that.
-Stack just keeps the reference to where object stored in heap.

-We can change value of const, only when working with non-primitive type i.e. with object and not with primitive type.

-For Primitive re-assign of value is important to change the value.

-For Reference type we don't need to reassign value because it changes everything on memory address which point to the heap.

PRIMITIVE VS. REFERENCE VALUES

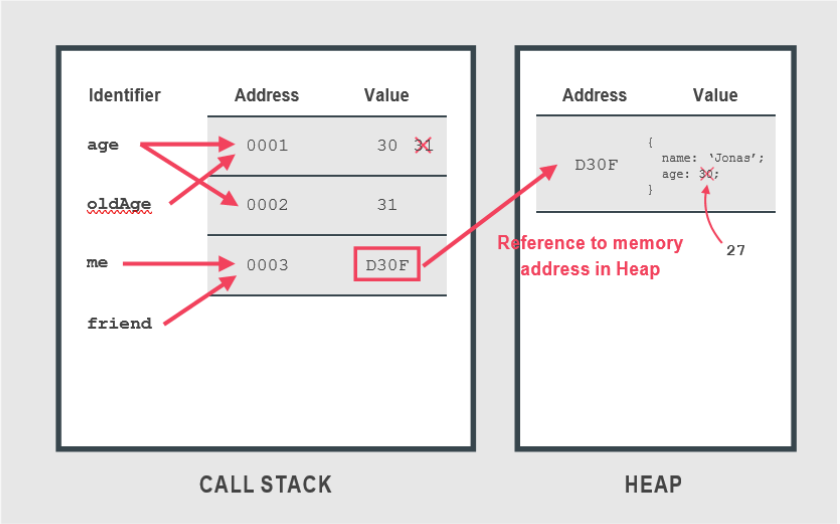
Primitive values example:

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

Reference values example:

```
const me = {
  name: 'Jonas'
  age: 30
};
const friend = me;
friend.age = 27;

console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }

console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```

No problem, because we're NOT changing the *value* at address 0003!




```
// Hoisting and TDZ in Practice

// Variables
console.log(me);
// console.log(job);
// console.log(year);

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

// Functions
console.log(addDecl(2, 3));
// console.log(addExpr(2, 3));
console.log(addArrow);
// console.log(addArrow(2, 3));

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);
if (!numProducts) deleteShoppingCart();

var numProducts = 10;
```

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

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

console.log(x === window.x);
console.log(y === window.y);
console.log(z === window.z);


// The this Keyword in Practice
console.log(this);

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

const calcAgeArrow = birthYear => {
  console.log(2037 - birthYear);
  console.log(this);
};
calcAgeArrow(1980);

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

const matilda = {
  year: 2017,
};

matilda.calcAge = jonas.calcAge;
matilda.calcAge();

const f = jonas.calcAge;
f();

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

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

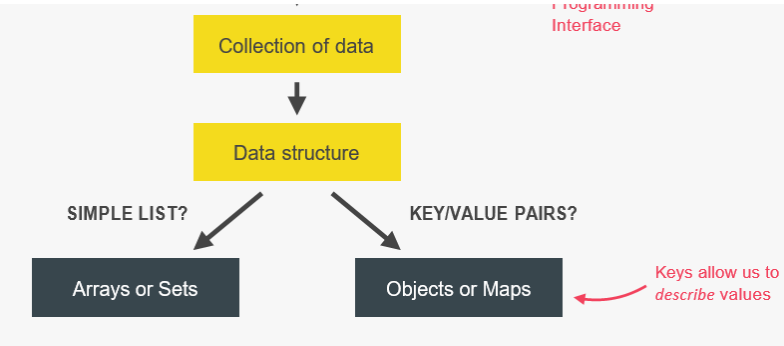
```
    // Solution 1
    // const self = this; // self or that
    // const isMillenial = function () {
    //   console.log(self);
    //   console.log(self.year >= 1981 && self.year <= 1996);
    // };

    // 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();
jonas.calcAge();

// arguments keyword
const addExpr = function (a, b) {
  console.log(arguments);
  return a + b;
};
addExpr(2, 5);
addExpr(2, 5, 8, 12);

var addArrow = (a, b) => {
  console.log(arguments);
  return a + b;
};
addArrow(2, 5, 8);

// Primitives vs. Objects in Practice

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

// Reference types
const jessica = {
  firstName: 'Jessica',
  lastName: 'Williams',
  age: 27,
};
const marriedJessica = jessica;
marriedJessica.lastName = 'Davis';
console.log('Before marriage:', jessica);     //Jessica Devis 27
console.log('After marriage: ', marriedJessica);      //Jessica Devis 27
// marriedJessica = {};
```

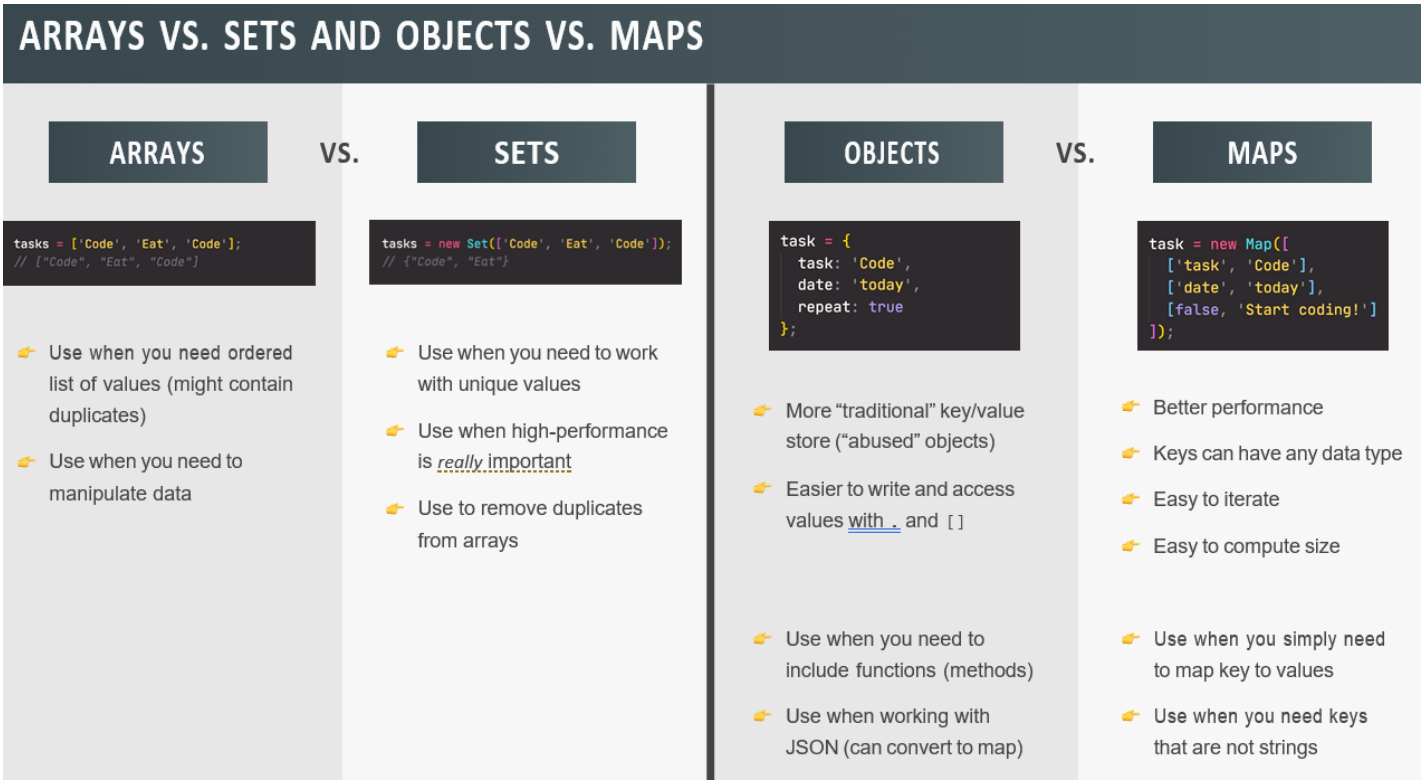
```
// Copying objects
const jessica2 = {
  firstName: 'Jessica',
  lastName: 'Williams',
  age: 27,
  family: ['Alice', 'Bob'],
};

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

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

console.log('Before marriage:', jessica2);    //Jessica Williams 27 array{4}
console.log('After marriage: ', jessicaCopy); //Jessica davis 27 array{4}
```

Data Structure and Overview

Source of Data:

1. From the Program itself: Directly written in source code
2. From the UI: Data input from user or from DOM
3. From the external source: From API





String

Split: allows us to split the string into different part.

```
console.log('a+very+nice+string'.split('+'));
["a", "very", "nice", "string"]

console.log('Jonas Schmedtmann'.split(' '));
["Jonas", "Schmedtmann"]
```

Join: join() method returns the array as a string.

```
The elements will be separated by a specific operator. The default separator
is comma(,)
Note: This method will not change the original array.

const [firstName, lastName] = 'Jonas Schmedtmann'.split(' ');

const newName = ['Mr.', firstName, lastName.toUpperCase()].join(' ');
console.log(newName);  //Mr. Jonas SCHMEDTMANN
```

Padding:

```
const message = 'Go to gate 23!';
console.log(message.padStart(20, '+').padEnd(30, '+'));
console.log('Jonas'.padStart(20, '+').padEnd(30, '+'));

++++++Go to gate 23!++++++++++
++++++++++++++Jonas++++++++++
```

Repeat: It will repeat the string. For the no. as passed in argument.

```
const message2 = 'Bad waether... All Departues Delayed... ';
console.log(message2.repeat(5));
```

Bad waether... All Departues Delayed... Bad waether... All Departues Delayed... Bad waether... All Departues Delayed... Bad waether... All Departues Delayed..

toUpperCase() & toLowerCase():

```
const airline = 'TAP Air Portugal';

console.log(airline.toLowerCase());
console.log(airline.toUpperCase());

// Fix capitalization in name
const passenger = 'jOnAS'; // Jonas
const passengerLower = passenger.toLowerCase();
const passengerCorrect =
  passengerLower[0].toUpperCase() + passengerLower.slice(1);
console.log(passengerCorrect);        //Jonas
```

passengerLower.slice(1)
"onas"

Replace: replace the character with a new character which is passed in argument

```
const priceGB = '288,97£';
const priceUS = priceGB.replace('£', '$').replace(',', '.');
console.log(priceUS);  //288.97$
```

Boolean:

```
const plane = 'Airbus A320neo';
console.log(plane.includes('A320'));          //true
console.log(plane.includes('Boeing'));        //false
console.log(plane.startsWith('Airb'));        //true
```

Working with String

```
const airline = 'TAP Air Portugal';
const plane = 'A320';

console.log(plane[0]);                                  //A
console.log(plane[1]);                                  //3
console.log(plane[2]);                                  //2
console.log('B737'[0]);                                 //B
console.log(airline.length);                            //16
console.log('B737'.length);                             //4
console.log(airline.indexOf('r'));                      //6
console.log(airline.lastIndexOf('r'));                  //10
console.log(airline.indexOf('portugal'));     //-1(because not available)
console.log(airline.slice(4));                          //Air Portugal
console.log(airline.slice(4, 7));                       //Air
console.log(airline.slice(0, airline.indexOf(' ')));         //TAP
console.log(airline.slice(airline.lastIndexOf(' ') + 1));    //Portugal
console.log(airline.slice(-2));                         //al
console.log(airline.slice(1, -1));                      //AP Air Portuga


console.log(new String('jonas'));                       //String {"jonas"}
console.log(typeof new String('jonas'));                //Object
console.log(typeof new String('jonas').slice(1));       //String
```

```
// Maps: Iteration

const question = new Map([
  ['question', 'What is the best programming language in the world?'],
  [1, 'C'],
  [2, 'Java'],
  [3, 'JavaScript'],
  ['correct', 3],
  [true, 'Correct 🎉'],
  [false, 'Try again!'],
]);
console.log(question);

//object
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};

// Convert object to map
console.log(Object.entries(openingHours));
const hoursMap = new Map(Object.entries(openingHours));
console.log(hoursMap);

//Iterate to Map
console.log(question.get('question'));
for (const [key, value] of question) {
  if (typeof key === 'number') console.log(`Answer ${key}: ${value}`);
}

// Convert map to array
console.log([...question]);
// console.log(question.entries());
console.log([...question.keys()]);
console.log([...question.values()]);


// Maps: Fundamentals
const rest = new Map();
rest.set('name', 'Classico Italiano');
rest.set(1, 'Firenze, Italy');
console.log(rest.set(2, 'Lisbon, Portugal'));

rest
  .set('categories', ['Italian', 'Pizzeria', 'Vegetarian', 'Organic'])
```

```
  .set('open', 11)
  .set('close', 23)
  .set(true, 'We are open :D')
  .set(false, 'We are closed :(');

console.log(rest.get('name'));
console.log(rest.get(true));
console.log(rest.get(1));

const time = 8;
console.log(rest.get(time > rest.get('open') && time < rest.get('close')));

console.log(rest.has('categories'));          //true
rest.delete(2);
rest.clear();

const arr = [1, 2];
rest.set(arr, 'Test');
console.log(rest);
console.log(rest.size);

console.log(rest.get(arr));

// Sets
const ordersSet = new Set([
  'Pasta',
  'Pizza',
  'Pizza',
  'Risotto',
  'Pasta',
  'Pizza',
]);
console.log(ordersSet);               //pasta,pizza,risotto
console.log(new Set('Jonas'));        //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');

//Set(4) {"Pasta", "Pizza", "Risotto", "Garlic Bread"}
ordersSet.delete('Risotto');
ordersSet.clear();
console.log(ordersSet);

//Iteration
for (const order of ordersSet) console.log(order);

//Assigning to another set/array
const staff = ['Waiter', 'Chef', 'Waiter', 'Manager', 'Chef', 'Waiter'];
const staffUnique = [...new Set(staff)];
console.log(staffUnique);

console.log(new Set('jonasschmedtmann').size);       //11

// Looping Objects: Object Keys, Values, and Entries
```

```
const weekdays = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun'];
const openingHours = {
  [weekdays[3]]: {
    open: 12,
    close: 22,
  },
  [weekdays[4]]: {
    open: 11,
    close: 23,
  },
  [weekdays[5]]: {
    open: 0, // Open 24 hours
    close: 24,
  },
};


// Property NAMES
const properties = Object.keys(openingHours);
console.log(properties);

let openStr = `We are open on ${properties.length} days: `;
for (const day of properties) {
  openStr += `${day}, `;
}
console.log(openStr);

// Property VALUES
const values = Object.values(openingHours);
console.log(values);

// 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}`);
}

// Optional Chaining
if (restaurant.openingHours && restaurant.openingHours.mon)
  console.log(restaurant.openingHours.mon.open);

// WITH optional chaining
console.log(restaurant.openingHours.mon?.open);
console.log(restaurant.openingHours?.mon?.open);

//object
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,

  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);
  },
};


// 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}`);
}

// Methods
console.log(restaurant.order?.(0, 1) ?? 'Method does not exist');
console.log(restaurant.orderRisotto?.(0, 1) ?? 'Method does not exist');

// Arrays
const users = [{ name: 'Jonas', email: 'hello@jonas.io' }];
// const users = [];

console.log(users[0]?.name ?? 'User array empty');

if (users.length > 0) console.log(users[0].name);
else console.log('user array empty');

// The for-of Loop
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];

for (const item of menu) console.log(item);

for (const [i, el] of menu.entries()) {
  console.log(`${i + 1}: ${el}`);
}
```

```
// console.log([...menu.entries()]);

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

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


// Short Circuiting (&& and ||)

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

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

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

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

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

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

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

restaurant.orderPizza && restaurant.orderPizza('mushrooms', 'spinach');

// 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);

const [pizza, , risotto, ...otherFood] = [
  ...restaurant.mainMenu,
  ...restaurant.starterMenu,
];
```

```
console.log(pizza, risotto, otherFood);

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

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

restaurant.orderPizza('mushrooms', 'onion', 'olives', 'spinach');
restaurant.orderPizza('mushrooms');


// The Spread Operator (...)

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

const newArr = [1, 2, ...arr];
console.log(newArr);

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

const newMenu = [...restaurant.mainMenu, 'Gnocci'];
console.log(newMenu);

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

// Join 2 arrays
const menu = [...restaurant.starterMenu, ...restaurant.mainMenu];
console.log(menu);

// Objects
const newRestaurant = { foundedIn: 1998, ...restaurant, founder: 'Guiseppe'
};
console.log(newRestaurant);

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

// Destructuring Objects
```

```
restaurant.orderDelivery({
  time: '22:30',
  address: 'Via del Sole, 21',
  mainIndex: 2,
  starterIndex: 2,
});

restaurant.orderDelivery({
  address: 'Via del Sole, 21',
  starterIndex: 1,
});

const { name, openingHours, categories } = restaurant;
console.log(name, openingHours, categories);

const {
  name: restaurantName,
  openingHours: hours,
  categories: tags,
} = restaurant;
console.log(restaurantName, hours, tags);

// Default values
const { menu = [], starterMenu: starters = [] } = restaurant;
console.log(menu, starters);

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

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

// Destructuring Arrays
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);
console.log(arr);

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

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

```
[main, secondary] = [secondary, main];
console.log(main, secondary);

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

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

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

Destructuring: Destructuring is an ES6 feature and it's basically a way of unpacking values from an array or an object into separate variables.

-or-Destructuring is a way of breaking down a complex data structure variable into a smaller data structure.

Spread Operator: Use basically to expand an array into all it's element.
-so basically unpacking all the array element at one.
-we can use this spread operator whenever we would otherwise write multiple values separated by commas.
-**USE**-1.Shallow copies of array
2.Merge two array together.
-Spread operator works on all arrays but not on all array just with array iterable.
-Iterable are things like all arrays, strings, maps or sets but not objects.
-We can only use spread operators when building an array, or when we pass values into a function.

Rest Syntax & Rest Parameter:(Opposite of spread)
-**Rest syntax** is taking multiple numbers or multiple values and then pack them up all into one array.
-**Rest pattern** can be used where we should write variable names, separated by commas and not values separated by commas.

Short-circuiting(&&, ||):
-In case of **OR** operator, short-circuiting means that if the first value is a truthy value, it will immediately return the first value.

-In case of **AND** operator short-circuiting, when the first value is falsy and then immediately returns that falsy value without even evaluating the second operand.

Optional Chaining: If a certain property does not exist, then undefined is returned immediately.

In ES6 two or more Data Structure are added that is **sets** and **maps.**

Sets: Just a collection of unique values. Can never have duplicate value.
-set can hold mixed data types

-set are iterable.
-set is different from array because.
        -contain unique value
        -order is irrelevant
-In set there is no indexes.
-There is no way of getting value out of set. That's because if all values are unique, and if the order does not matter, then there is no point of retrieving value from set.

Maps:
-It is a data structure to map values to key.
-So just like object, data is stored in key value pair in maps.
-Difference between object and map is that in map, a key can have any type.
-In Object, the keys are basically always string.

Strings: Strings are just primitive.
-so why do they have methods? Shouldn't be method only available to objects?
Ans: Whenever we call method on string, JS will convert the string primitive into string object behind the scene with the same content.
-And this process is called **Boxing.**

Default Parameters:

```
// Default Parameters
const bookings = [];

const createBooking = function (
  flightNum,
  numPassengers = 1,
  price = 199 * numPassengers
) {
  // ES5
  // numPassengers = numPassengers || 1;
  // price = price || 199;

  const booking = {
    flightNum,
    numPassengers,
    price,
  };
  console.log(booking);
  bookings.push(booking);
};

createBooking('LH123');
createBooking('LH123', 2, 800);
```

Argument works Value vs Reference:

```
// How Passing Arguments Works: Values vs. Reference
const flight = 'LH234';
const jonas = {
  name: 'Jonas Schmedtmann',
  passport: 24739479284,
};

const checkIn = function (flightNum, passenger) {
  flightNum = 'LH999';
  passenger.name = 'Mr. ' + passenger.name;
```

```
  if (passenger.passport === 24739479284) {
    alert('Checked in');
  } else {
    alert('Wrong passport!');
  }
};

// checkIn(flight, jonas);
// console.log(flight);
// console.log(jonas);

// Is the same as doing...
// const flightNum = flight;
// const passenger = jonas;
```

-**Val:** Just primitive type, as we passed that value into the function, then the flight num value here is basically a copy of that original value.
-so flightNum contains the copy and not simply the original value of the flight variable.
//flightNum=flight(it will not get reflected outside the function)

-**Ref:** When we pass a reference type to the function, what is copied is really just a reference to the object in the memory heap.

-**Summary:**

- Passing a primitive type to a function is really just the same as creating a copy like this, outside the function. So the value is simply copied.
- On the other hand, when we pass an object to a function, it is just like copying an object like this. So, whatever we change in copy will also happen in then original.

```
-const flightNum = flight;     //primitive
-const passenger = Jonas;      //object
```

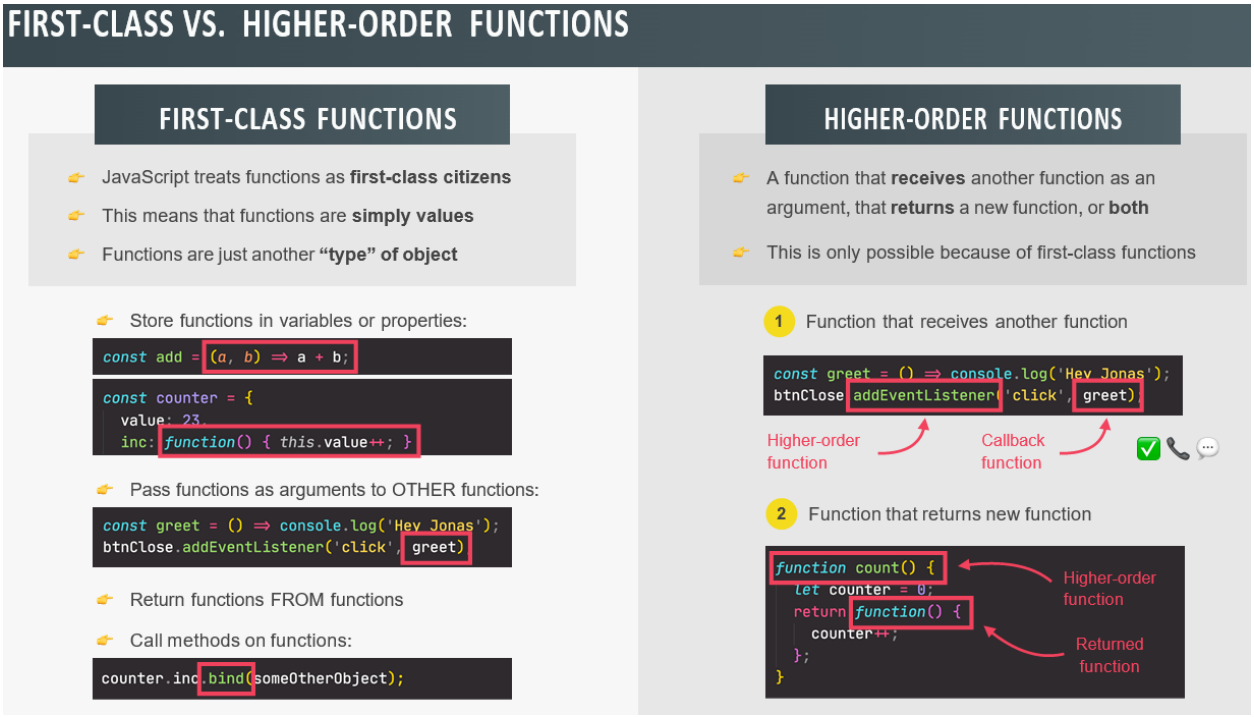
-JS, does not have passing by reference, only passing by value even though it's look like passing by reference.
-But as we look we passed object and that looks like pass by reference, but it's not. It's just simply a value that contains a memory address.
-Basically, we pass a reference to the function, but we do not pass by reference.

Why JS uses callback function all the time?
-Split of code into more reusable part
-callback function allows us to create abstraction

```
// Functions Accepting Callback Functions
const oneWord = function (str) {
  return str.replace(/ /g, '').toLowerCase();
};

const upperFirstWord = function (str) {
  const [first, ...others] = str.split(' ');
  return [first.toUpperCase(), ...others].join(' ');
```

```
};

// Higher-order function
const transformer = function (str, fn) {
  console.log(`Original string: ${str}`);
  console.log(`Transformed string: ${fn(str)}`);

  console.log(`Transformed by: ${fn.name}`);
};

transformer('JavaScript is the best!', upperFirstWord);
transformer('JavaScript is the best!', oneWord);

// JS uses callbacks all the time
const high5 = function () {
  console.log('✋');
};
document.body.addEventListener('click', high5);
['Jonas', 'Martha', 'Adam'].forEach(high5);
```

First-class Vs High-Order Function:



First-Class Function:

-JS treats function as first-class function
-This means that function are simply values
-Functions are just another "type" of objects

- Store function in variable or properties
- Pass function as arguments to OTHER function
- Return function FROM function
- Call methods on function

Higher-Order Function:

-A function that receives another function as an argument, that returns a new function, or both
-This is only possible because of first-class function

Note: Specifying part of arguments beforehand is actually a common pattern called **Partial Application.**

```
// Functions Returning Functions
const greet = function (greeting) {
  return function (name) {
    console.log(`${greeting} ${name}`);
  };
};

const greeterHey = greet('Hey');
greeterHey('Jonas');
greeterHey('Steven');

greet('Hello')('Jonas');

// Challenge
const greetArr = greeting => name => console.log(`${greeting} ${name}`);

greetArr('Hi')('Jonas');
```

Function Call:

In JavaScript all functions are object methods.

If a function is not a method of a JavaScript object, it is a function of the global object

The this Keyword

In a function definition, `this` refers to the "owner" of the function.

`call()` method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With `call()`, an object can use a method belonging to another object.

```
var person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
var person2 = {
  firstName:"Mary",
```

```
  lastName: "Doe"
}
person.fullName.call(person1);   //john doe

//call method with argument

var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.call(person1, "Oslo", "Norway");

// The call and apply Methods
const lufthansa = {
  airline: 'Lufthansa',
  iataCode: 'LH',
  bookings: [],
  // book: function() {}
  book(flightNum, name) {
    console.log(
      `${name} booked a seat on ${this.airline} flight
${this.iataCode}${flightNum}`
    );
    this.bookings.push({ flight: `${this.iataCode}${flightNum}`, name });
  },
};

lufthansa.book(239, 'Jonas Schmedtmann');
lufthansa.book(635, 'John Smith');

const eurowings = {
  airline: 'Eurowings',
  iataCode: 'EW',
  bookings: [],
};

const book = lufthansa.book;

// Does NOT work
// book(23, 'Sarah Williams');

// Call method
book.call(eurowings, 23, 'Sarah Williams');
console.log(eurowings);

book.call(lufthansa, 239, 'Mary Cooper');
console.log(lufthansa);

const swiss = {
  airline: 'Swiss Air Lines',
```

```
  iataCode: 'LX',
  bookings: [],
};

book.call(swiss, 583, 'Mary Cooper');
```

The `apply()` method is similar to the `call()` method

```
var person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
var person1 = {
  firstName: "Mary",
  lastName: "Doe"
}
person.fullName.apply(person1);  // Will return "Mary Doe"
```

The Difference Between call() and apply()

The difference is:

The `call()` method takes arguments **separately**.

The `apply()` method takes arguments as an **array**.

`apply()` method accepts arguments in an array

```
var person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," + country;
  }
}
var person1 = {
  firstName:"John",
  lastName: "Doe"
}
person.fullName.apply(person1, ["Oslo", "Norway"]);
```

```
// Apply method
const flightData = [583, 'George Cooper'];
book.apply(swiss, flightData);
console.log(swiss);

book.call(swiss, ...flightData);
```

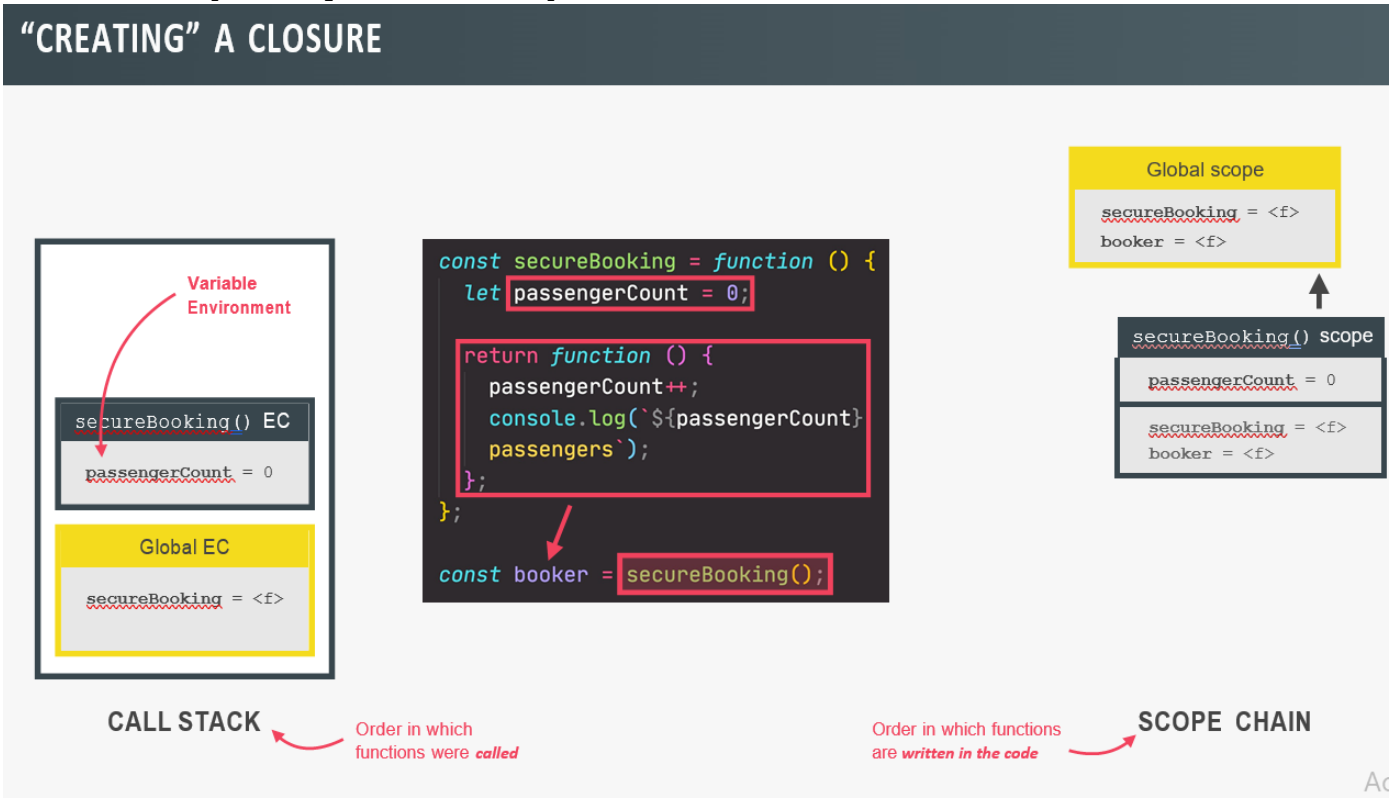
Bind Method: The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

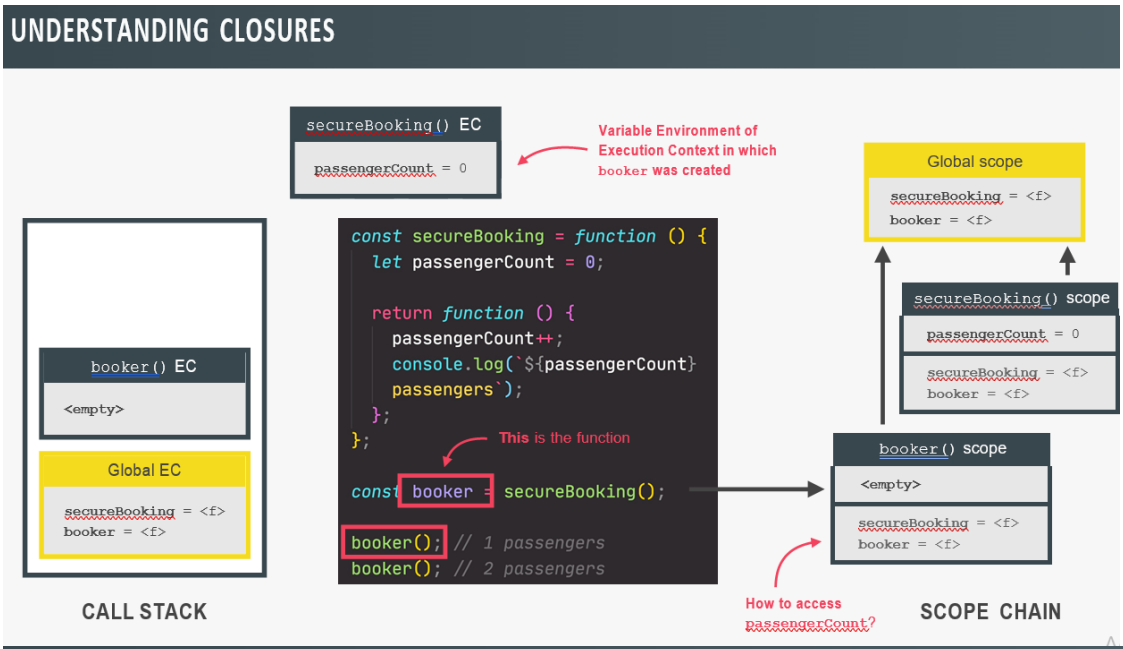
```
const module = {
```

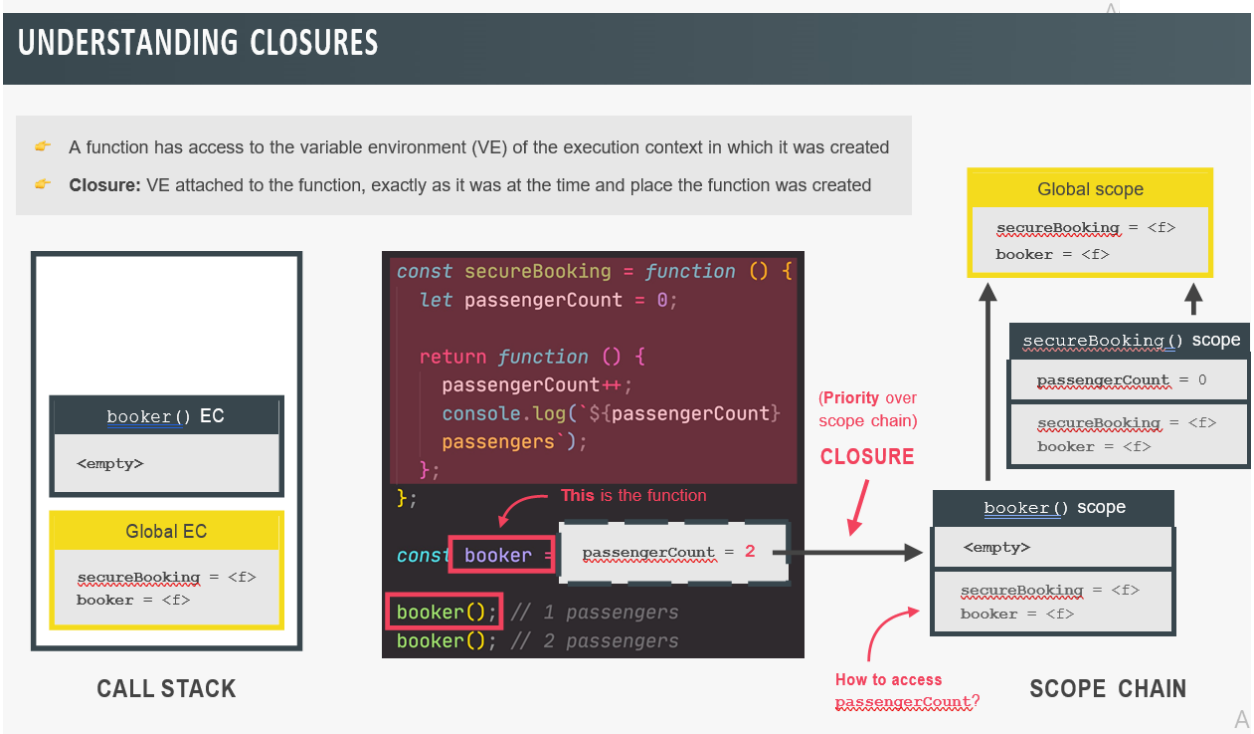
```
  x: 42,

  getX: function() {

    return this.x;

  }

};

const unboundGetX = module.getX;

console.log(unboundGetX()); // The function gets invoked at the global scope

// expected output: undefined

const boundGetX = unboundGetX.bind(module);

console.log(boundGetX());

// expected output: 42

// The bind Method
// book.call(eurowings, 23, 'Sarah Williams');

const bookEW = book.bind(eurowings);
const bookLH = book.bind(lufthansa);
const bookLX = book.bind(swiss);

bookEW(23, 'Steven Williams');

const bookEW23 = book.bind(eurowings, 23);
bookEW23('Jonas Schmedtmann');
bookEW23('Martha Cooper');

// With Event Listeners
lufthansa.planes = 300;
lufthansa.buyPlane = function () {
  console.log(this);

  this.planes++;
  console.log(this.planes);
};
// lufthansa.buyPlane();

document
  .querySelector('.buy')
  .addEventListener('click', lufthansa.buyPlane.bind(lufthansa));

// Partial application
const addTax = (rate, value) => value + value * rate;
console.log(addTax(0.1, 200));

const addVAT = addTax.bind(null, 0.23);
```

```
// addVAT = value => value + value * 0.23;

console.log(addVAT(100));
console.log(addVAT(23));

const addTaxRate = function (rate) {
  return function (value) {
    return value + value * rate;
  };
};
const addVAT2 = addTaxRate(0.23);
console.log(addVAT2(100));
console.log(addVAT2(23));
```

Closure: Closure makes a function remember all the variable that existed at the function birthplace essentially.
-Any function always has access to the variable environment of the execution context in which the function was created.
-Thanks to the closure, a function does not lose connection to variable that existed at the function birthplace.
-Closure has priority over the scope chain.








```
// Closures
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount} passengers`);
  };
};

const booker = secureBooking();
```

```
booker();
booker();
booker();

console.dir(booker);
```

-A closure is the closed-over variable environment of the execution context in which a function was created, even after that execution context is gone.
→(LF)A closure gives a function access to all the variable of its function, even after that parent function has returned. The function keeps a reference to its outer scope, which preserves the scope chain throughout time.

```
////////////////////////////////////////
// More Closure Examples
// Example 1
let f;

const g = function () {
  const a = 23;
  f = function () {
    console.log(a * 2);
  };
};

const h = function () {
  const b = 777;
  f = function () {
    console.log(b * 2);
  };
};

g();
f();
console.dir(f);

// Re-assigning f function
h();
f();
console.dir(f);

// Example 2
const boardPassengers = function (n, wait) {
  const perGroup = n / 3;

  setTimeout(function () {
    console.log(`We are now boarding all ${n} passengers`);
    console.log(`There are 3 groups, each with ${perGroup} passengers`);
  }, wait * 1000);

  console.log(`Will start boarding in ${wait} seconds`);
};

const perGroup = 1000;
boardPassengers(180, 3);
```

IIFE: Immediately Invoked Function Expression

> An **IIFE** (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. `(function () {`
>
> ```
>   statements
>
> })();
> ```

It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts:

1. The first is the anonymous function with lexical scope enclosed within the `Grouping Operator` `()`. This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
2. The second part creates the immediately invoked function expression `()` through which the JavaScript engine will directly interpret the function.

```
// Immediately Invoked Function Expressions (IIFE)
const runOnce = function () {
  console.log('This will never run again');
};
runOnce();

// IIFE
(function () {
  console.log('This will never run again');
  const isPrivate = 23;
})();

// console.log(isPrivate);

(() => console.log('This will ALSO never run again'))();

{
  const isPrivate = 23;
  var notPrivate = 46;
}
// console.log(isPrivate);
console.log(notPrivate);
```

Array:

Basic Array Method:

```
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];
```

Slice: Extract part of any array, but without changing the original array.

```
let arr = ['a', 'b', 'c', 'd', 'e'];

// SLICE
console.log(arr.slice(2));
console.log(arr.slice(2, 4));
console.log(arr.slice(-2));
console.log(arr.slice(-1));
console.log(arr.slice(1, -2));
```

```
console.log(arr.slice());
console.log([...arr]);
```

Splice: Works exactly same as slice, but fundamental difference is it does change the original array.
-It mutate the array.
-It doesn't interest us, it just interest us when we want to delete one or more element from an array using splice.

```
// SPLICE
// console.log(arr.splice(2));
arr.splice(-1);
console.log(arr);
arr.splice(1, 2);
console.log(arr);
```

Reverse: Reverse the array and does actually mutate the array.

```
// REVERSE
arr = ['a', 'b', 'c', 'd', 'e'];
const arr2 = ['j', 'i', 'h', 'g', 'f'];
console.log(arr2.reverse());
console.log(arr2);
```

Concat: Concatenate two array and does not mutate original array.

```
// CONCAT
const letters = arr.concat(arr2);
console.log(letters);
console.log([...arr, ...arr2]);
```

Join:

```
// JOIN
console.log(letters.join(' - '));
```

ForEach: movements.forEach(func(){})
-What forEach method does is to loop over the array, and in each iteration it will execute this callback function here.

```
// Looping Arrays: forEach
const movements = [200, 450, -400, 3000, -650, -130, 70, 1300];

// for (const movement of movements) {
for (const [i, movement] of movements.entries()) {
  if (movement > 0) {
    console.log(`Movement ${i + 1}: You deposited ${movement}`);
  } else {
    console.log(`Movement ${i + 1}: You withdrew ${Math.abs(movement)}`);
  }
}

console.log('---- FOREACH ----');
movements.forEach(function (mov, i, arr) {
  if (mov > 0) {
    console.log(`Movement ${i + 1}: You deposited ${mov}`);
  } else {
    console.log(`Movement ${i + 1}: You withdrew ${Math.abs(mov)}`);
```

```
  }
});
// 0: function(200)
// 1: function(450)
// 2: function(400)
// ...


/////////////////////////////////////
// forEach With Maps and Sets
// Map
const currencies = new Map([
  ['USD', 'United States dollar'],
  ['EUR', 'Euro'],
  ['GBP', 'Pound sterling'],
]);

currencies.forEach(function (value, key, map) {
  console.log(`${key}: ${value}`);
});

// Set
const currenciesUnique = new Set(['USD', 'GBP', 'USD', 'EUR', 'EUR']);
console.log(currenciesUnique);
currenciesUnique.forEach(function (value, _, map) {
  console.log(`${value}: ${value}`);
});
```

Note: You cannot break out of forEach loop, but can of for-of loop.

Map: Map creates a brand new array based on the original array.
-Map method takes an array, loop over that array and in each iteration, it applies an callback function that we specify in our code to the current array element.
-Better than forEach
-Map return a new array containing the result of applying an operation on all original array element.

```
// The map Method
const eurToUsd = 1.1;

// const movementsUSD = movements.map(function (mov) {
//   return mov * eurToUsd;
// });

const movementsUSD = movements.map(mov => mov * eurToUsd);

console.log(movements);
console.log(movementsUSD);

const movementsUSDfor = [];
for (const mov of movements) movementsUSDfor.push(mov * eurToUsd);
console.log(movementsUSDfor);

const movementsDescriptions = movements.map(
  (mov, i) =>
    `Movement ${i + 1}: You ${mov > 0 ? 'deposited' : 'withdrew'} ${Math.abs(
      mov
```

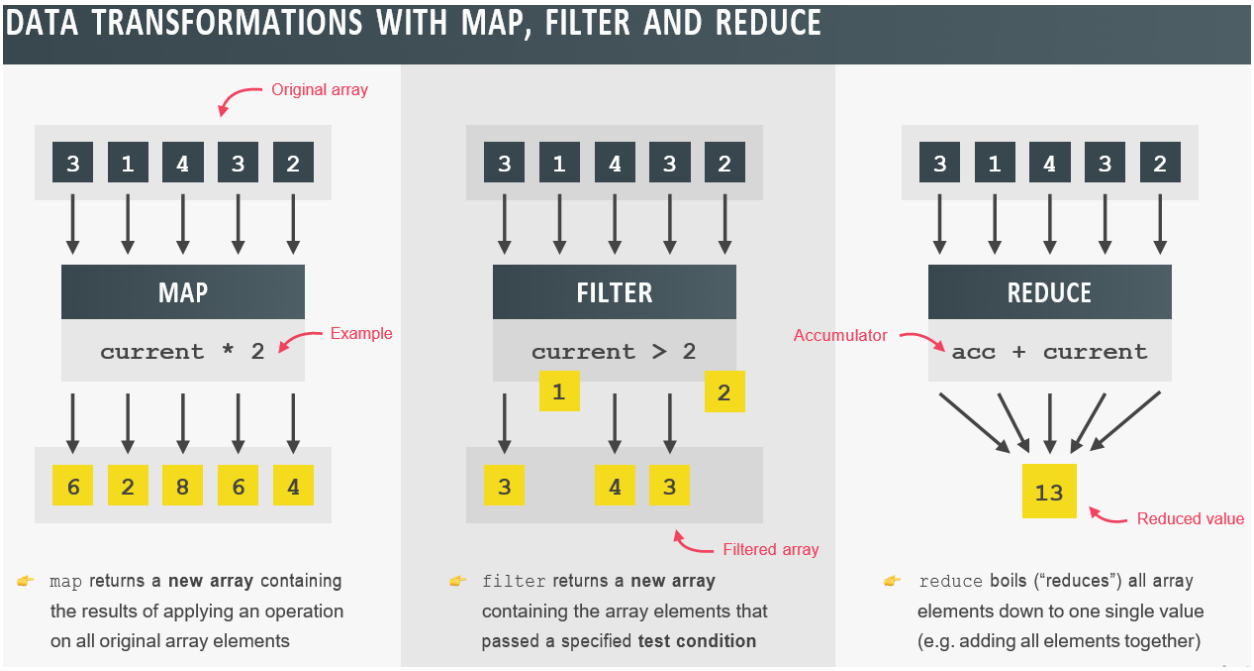
```
    )}`
);
console.log(movementsDescriptions);
```

Filter: Used to filter for element in the original array which satisfy certain condition.
-Filter returns a new array containing the array element that passes a specific test condition.

```
// The filter Method
const deposits = movements.filter(function (mov, i, arr) {
  return mov > 0;
});
console.log(movements);
console.log(deposits);

const depositsFor = [];
for (const mov of movements) if (mov > 0) depositsFor.push(mov);
console.log(depositsFor);

const withdrawals = movements.filter(mov => mov < 0);
console.log(withdrawals);
```

Reduce: Used to boil down all the elements of the original array into one single value. E.g.: adding all the elements together.

```
// The reduce Method
console.log(movements);

// accumulator -> SNOWBALL
// const balance = movements.reduce(function (acc, cur, i, arr) {
//   console.log(`Iteration ${i}: ${acc}`);
//   return acc + cur;
// }, 0);
const balance = movements.reduce((acc, cur) => acc + cur, 0);
console.log(balance);

let balance2 = 0;
for (const mov of movements) balance2 += mov;
console.log(balance2);

// Maximum value
const max = movements.reduce((acc, mov) => {
  if (acc > mov) return acc;
  else return mov;
}, movements[0]);
console.log(max);
```




```
// The Magic of Chaining Methods
const eurToUsd = 1.1;
console.log(movements);

// PIPELINE
const totalDepositsUSD = movements
  .filter(mov => mov > 0)
  .map((mov, i, arr) => {
    // console.log(arr);
    return mov * eurToUsd;
  })
  // .map(mov => mov * eurToUsd)
  .reduce((acc, mov) => acc + mov, 0);
console.log(totalDepositsUSD);
```

Find: Use find method to retrieve one element, of an array based on a condition.
-It will not return the array, but will return the first element that satisfy the condition.

```
// The find Method
const firstWithdrawal = movements.find(mov => mov < 0);
console.log(movements);
console.log(firstWithdrawal);

console.log(accounts);

const account = accounts.find(acc => acc.owner === 'Jessica Davis');
console.log(account);
```

Some: The some() method checks if any of the elements in an array pass a test

The some() method executes the function once for each element present in the array:

- If it finds an array element where the function returns a *true* value, some() returns true (and does not check the remaining values)
- Otherwise it returns false

Note: some() does not execute the function for array elements without values.

Note: some() does not change the original array.

Every: The every() method checks if all elements in an array pass a test

```
// some and every
console.log(movements);

// EQUALITY
console.log(movements.includes(-130));

// SOME: CONDITION
console.log(movements.some(mov => mov === -130));

const anyDeposits = movements.some(mov => mov > 0);
console.log(anyDeposits);

// EVERY
console.log(movements.every(mov => mov > 0));
console.log(account4.movements.every(mov => mov > 0));

// Separate callback
const deposit = mov => mov > 0;
console.log(movements.some(deposit));
console.log(movements.every(deposit));
console.log(movements.filter(deposit));
```

Flat: The **flat()** method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

```
const arr1 = [0, 1, 2, [3, 4]];

console.log(arr1.flat());
// expected output: [0, 1, 2, 3, 4]

const arr2 = [0, 1, 2, [[[3, 4]]]];

console.log(arr2.flat(2));
// expected output: [0, 1, 2, [3, 4]]

console.log(arr2.flat(3));
// expected output: [0, 1, 2, 3, 4]
```

FlatMap: The **flatMap()** method returns a new array formed by applying a given callback function to each element of the array, and then flattening the result by one level. It is identical to a map() followed by a flat() of depth 1, but slightly more efficient than calling those two methods separately.

```
// flat and flatMap
const arr = [[1, 2, 3], [4, 5, 6], 7, 8];
console.log(arr.flat());
```

```
const arrDeep = [[[1, 2], 3], [4, [5, 6]], 7, 8];
console.log(arrDeep.flat(2));

// flat
const overalBalance = accounts
  .map(acc => acc.movements)
  .flat()
  .reduce((acc, mov) => acc + mov, 0);
console.log(overalBalance);

// flatMap
const overalBalance2 = accounts
  .flatMap(acc => acc.movements)
  .reduce((acc, mov) => acc + mov, 0);
console.log(overalBalance2);

// Sorting Arrays

// Strings
const owners = ['Jonas', 'Zach', 'Adam', 'Martha'];
console.log(owners.sort());
console.log(owners);

// Numbers
console.log(movements);

// return < 0, A, B (keep order)
// return > 0, B, A (switch order)

// Ascending
// movements.sort((a, b) => {
//   if (a > b) return 1;
//   if (a < b) return -1;
// });
movements.sort((a, b) => a - b);
console.log(movements);

// Descending
// movements.sort((a, b) => {
//   if (a > b) return -1;
//   if (a < b) return 1;
// });
movements.sort((a, b) => b - a);
console.log(movements);
```

Fill: The `fill()` method fills the specified elements in an array with a static value.

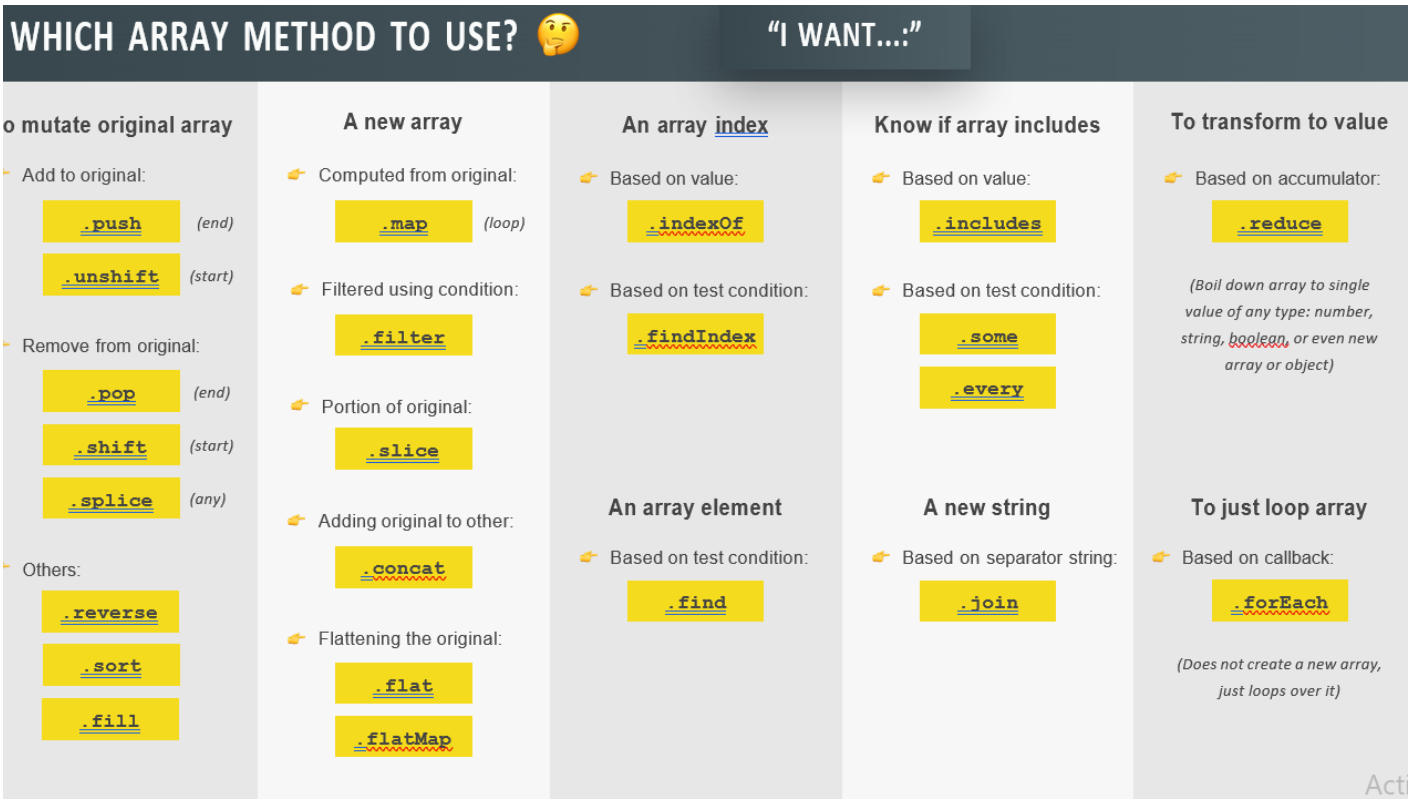
You can specify the position of where to start and end the filling. If not specified, all elements will be filled.

Note: this method overwrites the original array.

```
// More Ways of Creating and Filling Arrays
const arr = [1, 2, 3, 4, 5, 6, 7];
console.log(new Array(1, 2, 3, 4, 5, 6, 7));

// Empty arrays + fill method
const x = new Array(7);
console.log(x);
// console.log(x.map(() => 5));
x.fill(1, 3, 5);
x.fill(1);
console.log(x);

arr.fill(23, 2, 6);
console.log(arr);

// Array.from
const y = Array.from({ length: 7 }, () => 1);
console.log(y);

const z = Array.from({ length: 7 }, (_, i) => i + 1);
console.log(z);

labelBalance.addEventListener('click', function () {
  const movementsUI = Array.from(
    document.querySelectorAll('.movements__value'),
    el => Number(el.textContent.replace('€', ''))
  );
  console.log(movementsUI);

  const movementsUI2 = [...document.querySelectorAll('.movements__value')];
});
```

From: The **`Array.from()`** static method creates a new, shallow-copied `Array` instance from an array-like or iterable object.

The Array.from() method returns an Array object from any object with a length property or an iterable object.
`Array.from(`*`object, mapFunction, thisValue`*`)`




```
// Converting and Checking Numbers
console.log(23 === 23.0);                       //true

// Base 10 - 0 to 9. 1/10 = 0.1. 3/10 = 3.3333333
// Binary base 2 - 0 1
console.log(0.1 + 0.2);                         // 0.30000000000000004
console.log(0.1 + 0.2 === 0.3);                 //false

// Conversion
console.log(Number('23'));
console.log(+'23');

// Parsing
console.log(Number.parseInt('30px', 10));       //30
console.log(Number.parseInt('e23', 10));        //NaN

console.log(Number.parseInt('  2.5rem  '));     //2
console.log(Number.parseFloat('  2.5rem  '));   //2.5

// console.log(parseFloat('  2.5rem  '));

// Check if value is NaN
console.log(Number.isNaN(20));                  //false
console.log(Number.isNaN('20'));                //false
console.log(Number.isNaN(+'20X'));              //true
console.log(Number.isNaN(23 / 0));              //false

// Checking if value is number
console.log(Number.isFinite(20));               //true
```

```
console.log(Number.isFinite('20'));          //false
console.log(Number.isFinite(+'20X'));        //false
console.log(Number.isFinite(23 / 0));        //false

console.log(Number.isInteger(23));           //true
console.log(Number.isInteger(23.0));         //true
console.log(Number.isInteger(23 / 0));       //false

// Math and Rounding
console.log(Math.sqrt(25));
console.log(25 ** (1 / 2));
console.log(8 ** (1 / 3));

console.log(Math.max(5, 18, 23, 11, 2));
console.log(Math.max(5, 18, '23', 11, 2));
console.log(Math.max(5, 18, '23px', 11, 2));

console.log(Math.min(5, 18, 23, 11, 2));

console.log(Math.PI * Number.parseFloat('10px') ** 2);

console.log(Math.trunc(Math.random() * 6) + 1);

const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min) + 1) + min;
// 0...1 -> 0...(max - min) -> min...max
// console.log(randomInt(10, 20));

// Rounding integers
console.log(Math.round(23.3));
console.log(Math.round(23.9));

console.log(Math.ceil(23.3));
console.log(Math.ceil(23.9));

console.log(Math.floor(23.3));
console.log(Math.floor('23.9'));

console.log(Math.trunc(23.3));

console.log(Math.trunc(-23.3));
console.log(Math.floor(-23.3));

// Rounding decimals
console.log((2.7).toFixed(0));
console.log((2.7).toFixed(3));
console.log((2.345).toFixed(2));
console.log(+(2.345).toFixed(2));


////////////////////////////////////
// The Remainder Operator
console.log(5 % 2);
console.log(5 / 2); // 5 = 2 * 2 + 1

console.log(8 % 3);
console.log(8 / 3); // 8 = 2 * 3 + 2
```

```
console.log(6 % 2);
console.log(6 / 2);

console.log(7 % 2);
console.log(7 / 2);

const isEven = n => n % 2 === 0;
console.log(isEven(8));
console.log(isEven(23));
console.log(isEven(514));

labelBalance.addEventListener('click', function () {
  [...document.querySelectorAll('.movements__row')].forEach(function (row, i)
{
    // 0, 2, 4, 6
    if (i % 2 === 0) row.style.backgroundColor = 'orangered';
    // 0, 3, 6, 9
    if (i % 3 === 0) row.style.backgroundColor = 'blue';
  });
});


/////////////////////////////////////
// Working with BigInt
console.log(2 ** 53 - 1);
console.log(Number.MAX_SAFE_INTEGER);
console.log(2 ** 53 + 1);
console.log(2 ** 53 + 2);
console.log(2 ** 53 + 3);
console.log(2 ** 53 + 4);

console.log(4838430248342043823408394839483204n);
console.log(BigInt(48384302));

// Operations
console.log(10000n + 10000n);
console.log(362863726372637263762372637263726372632n * 10000000n);
// console.log(Math.sqrt(16n));

const huge = 20289830237283728378237n;
const num = 23;
console.log(huge * BigInt(num));

// Exceptions
console.log(20n > 15);
console.log(20n === 20);
console.log(typeof 20n);
console.log(20n == '20');

console.log(huge + ' is REALLY big!!!');

// Divisions
console.log(11n / 3n);
console.log(10 / 3);

// Creating Dates
```

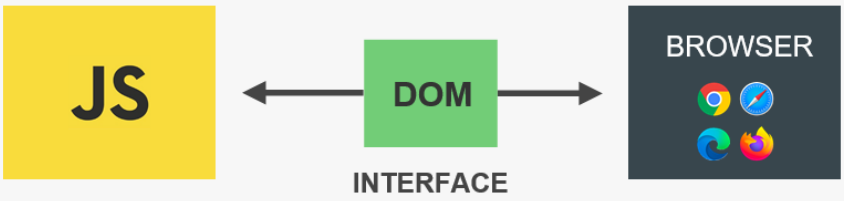
```
// Create a date

const now = new Date();
console.log(now);

console.log(new Date('Aug 02 2020 18:05:41'));
console.log(new Date('December 24, 2015'));
console.log(new Date(account1.movementsDates[0]));

console.log(new Date(2037, 10, 19, 15, 23, 5));
console.log(new Date(2037, 10, 31));

console.log(new Date(0));
console.log(new Date(3 * 24 * 60 * 60 * 1000));


// Working with dates
const future = new Date(2037, 10, 19, 15, 23);
console.log(future);
console.log(future.getFullYear());
console.log(future.getMonth());
console.log(future.getDate());
console.log(future.getDay());
console.log(future.getHours());
console.log(future.getMinutes());
console.log(future.getSeconds());
console.log(future.toISOString());
console.log(future.getTime());

console.log(new Date(2142256980000));

console.log(Date.now());

future.setFullYear(2040);
console.log(future);


/////////////////////////////////////
// Operations With Dates
const future = new Date(2037, 10, 19, 15, 23);
console.log(+future);

const calcDaysPassed = (date1, date2) =>
  Math.abs(date2 - date1) / (1000 * 60 * 60 * 24);

const days1 = calcDaysPassed(new Date(2037, 3, 4), new Date(2037, 3, 14));
console.log(days1);


/////////////////////////////////////
// Internationalizing Numbers (Intl)
const num = 3884764.23;

const options = {
  style: 'currency',
  unit: 'celsius',
  currency: 'EUR',
```

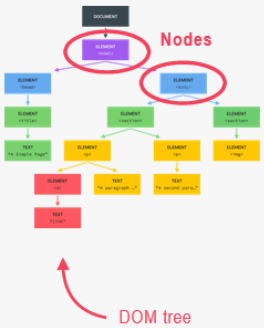
```
  // useGrouping: false,
};

console.log('US:      ', new Intl.NumberFormat('en-US',
options).format(num));
console.log('Germany: ', new Intl.NumberFormat('de-DE',
options).format(num));
console.log('Syria:   ', new Intl.NumberFormat('ar-SY',
options).format(num));
console.log(
  navigator.language,
  new Intl.NumberFormat(navigator.language, options).format(num)
);


////////////////////////////////////
// Timers

// setTimeout
const ingredients = ['olives', 'spinach'];
const pizzaTimer = setTimeout(
  (ing1, ing2) => console.log(`Here is your pizza with ${ing1} and ${ing2}
🍕`),
  3000,
  ...ingredients
);
console.log('Waiting...');

if (ingredients.includes('spinach')) clearTimeout(pizzaTimer);

// setInterval
setInterval(function () {
  const now = new Date();
  console.log(now);
}, 1000);
```

Advance DOM Events:

-Allows us to make JS interact with Browser
-We can write JS to create, modify and delete HTML elements, set styles, classes and attributes and listen and respond to events.
-DOM Tree is generated from an HTML document which we can then interact with.
-DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree.

REVIEW: WHAT IS THE DOM?



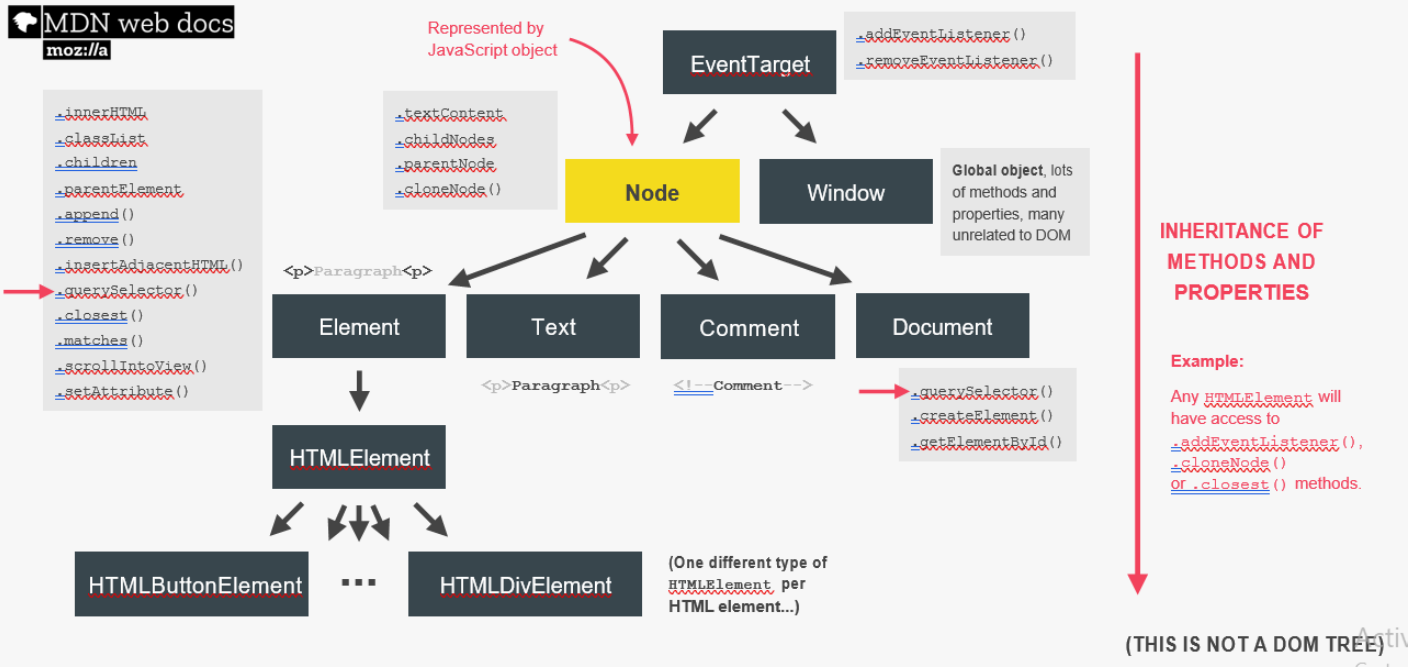


- Allows us to make JavaScript interact with the browser;
- We can write JavaScript to create, modify and delete HTML elements; set styles, classes and attributes; and listen and respond to events;
- DOM tree is generated from an HTML document, which we can then interact with;
- DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree

Application Programming Interface

```
.querySelector() / .addEventListener() / .createElement() /
.innerHTML / .textContent / .children / etc ...
```

HOW THE DOM API IS ORGANIZED BEHIND THE SCENES

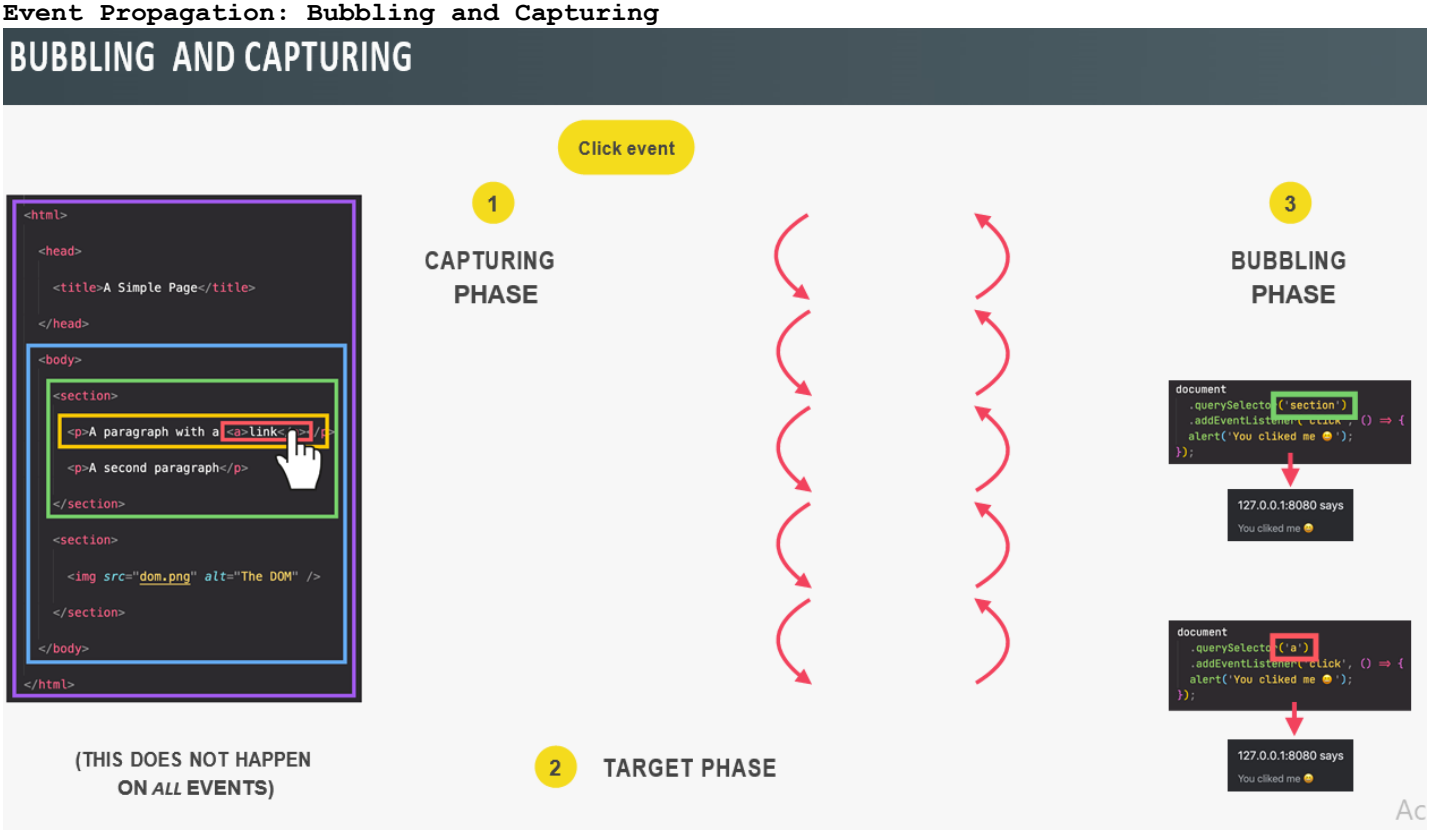



```
// Selecting, Creating, and Deleting Elements

// Selecting elements
console.log(document.documentElement);
console.log(document.head);
```

```
console.log(document.body);

const header = document.querySelector('.header');
const allSections = document.querySelectorAll('.section');
console.log(allSections);

document.getElementById('section--1');
const allButtons = document.getElementsByTagName('button');
console.log(allButtons);

console.log(document.getElementsByClassName('btn'));

// Creating and inserting elements
const message = document.createElement('div');
message.classList.add('cookie-message');
// message.textContent = 'We use cookied for improved functionality and
analytics.';
message.innerHTML =
  'We use cookied for improved functionality and analytics. <button
class="btn btn--close-cookie">Got it!</button>';

// header.prepend(message);
header.append(message);
// header.append(message.cloneNode(true));

// header.before(message);
// header.after(message);

// Delete elements
document
  .querySelector('.btn--close-cookie')
  .addEventListener('click', function () {
    // message.remove();
    message.parentElement.removeChild(message);
  });


////////////////////////////////////
// Styles, Attributes and Classes

// Styles
message.style.backgroundColor = '#37383d';
message.style.width = '120%';

console.log(message.style.color);
console.log(message.style.backgroundColor);

console.log(getComputedStyle(message).color);
console.log(getComputedStyle(message).height);

message.style.height =
  Number.parseFloat(getComputedStyle(message).height, 10) + 30 + 'px';

document.documentElement.style.setProperty('--color-primary', 'orangered');

// Attributes
const logo = document.querySelector('.nav__logo');
```

```
console.log(logo.alt);
console.log(logo.className);

logo.alt = 'Beautiful minimalist logo';

// Non-standard
console.log(logo.designer);
console.log(logo.getAttribute('designer'));
logo.setAttribute('company', 'Bankist');

console.log(logo.src);
console.log(logo.getAttribute('src'));

const link = document.querySelector('.nav__link--btn');
console.log(link.href);
console.log(link.getAttribute('href'));

// Data attributes
console.log(logo.dataset.versionNumber);

// Classes
logo.classList.add('c', 'j');
logo.classList.remove('c', 'j');
logo.classList.toggle('c');
logo.classList.contains('c'); // not includes

// Don't use
logo.clasName = 'jonas';
```

Event Propagation: Bubbling and Capturing



-JS Events has a important property, which capture and bubbling phase.
-this keyword and e.currenttarget are exactly gonna be same in any event handler

DOM Traversing: DOM Traversing is basically walking through the DOM, Which means we can select an element based on another element.

-Query Selector all it will go down as deep as it can for child element, but not for the indirect child element

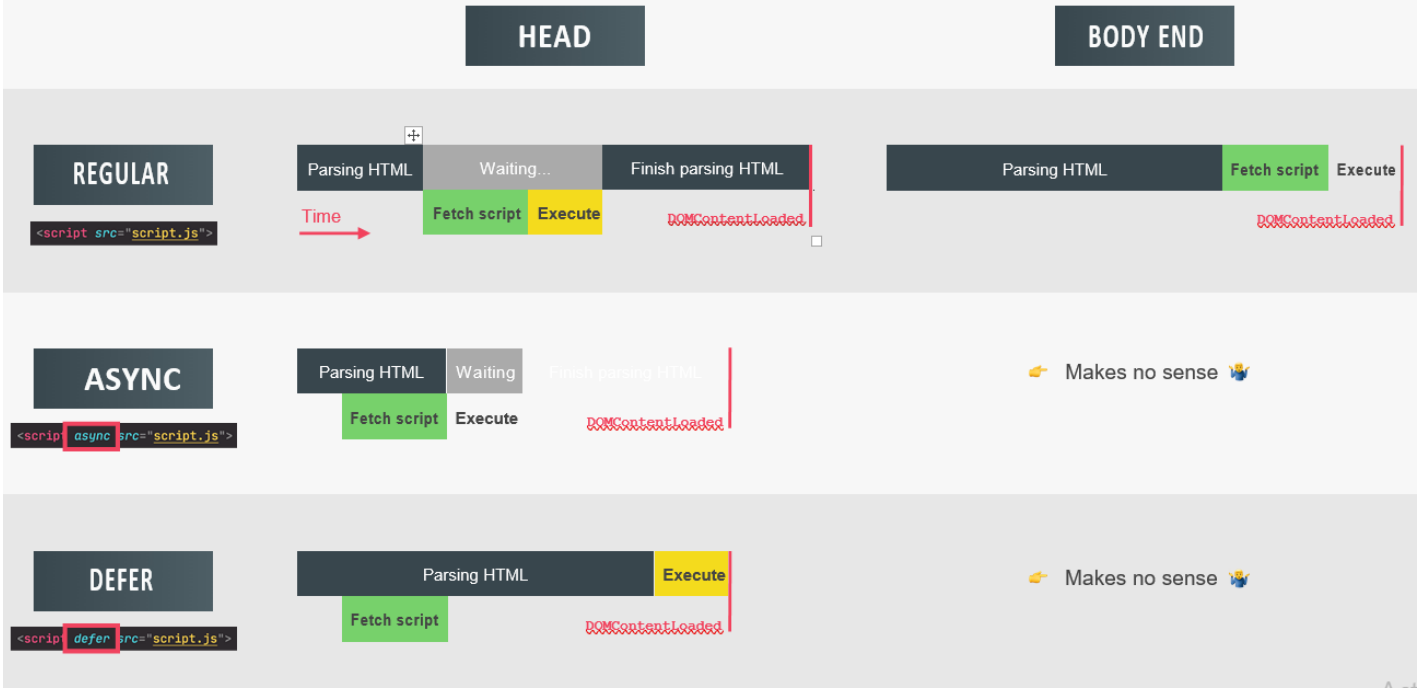
Guard Clause: It is basically an if statement which will return early if some condition is matched.
if(!clicked) return;

Bind: Bind method creates a copy of the function that it's called on and it will set this keyword in the function call, to whatever value we pass in the bind.

```
// Types of Events and Event Handlers
const h1 = document.querySelector('h1');

const alertH1 = function (e) {
  alert('addEventListener: Great! You are reading the heading :D');
};

h1.addEventListener('mouseenter', alertH1);

setTimeout(() => h1.removeEventListener('mouseenter', alertH1), 3000);

// h1.onmouseenter = function (e) {
//   alert('onmouseenter: Great! You are reading the heading :D');
// };

// Event Propagation in Practice
const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min + 1) + min);
const randomColor = () =>
  `rgb(${randomInt(0, 255)},${randomInt(0, 255)},${randomInt(0, 255)})`;

document.querySelector('.nav__link').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('LINK', e.target, e.currentTarget);
  console.log(e.currentTarget === this);

  // Stop propagation
  // e.stopPropagation();
});

document.querySelector('.nav__links').addEventListener('click', function (e)
{
  this.style.backgroundColor = randomColor();
  console.log('CONTAINER', e.target, e.currentTarget);
});

document.querySelector('.nav').addEventListener('click', function (e) {
  this.style.backgroundColor = randomColor();
  console.log('NAV', e.target, e.currentTarget);
```

```
});

// DOM Traversing
const h1 = document.querySelector('h1');

// Going downwards: child
console.log(h1.querySelectorAll('.highlight'));
console.log(h1.childNodes);
console.log(h1.children);
h1.firstElementChild.style.color = 'white';
h1.lastElementChild.style.color = 'orangered';

// Going upwards: parents
console.log(h1.parentNode);
console.log(h1.parentElement);

h1.closest('.header').style.background = 'var(--gradient-secondary)';

h1.closest('h1').style.background = 'var(--gradient-primary)';

// Going sideways: siblings
console.log(h1.previousElementSibling);
console.log(h1.nextElementSibling);

console.log(h1.previousSibling);
console.log(h1.nextSibling);

console.log(h1.parentElement.children);
[...h1.parentElement.children].forEach(function (el) {
  if (el !== h1) el.style.transform = 'scale(0.5)';
});

// Sticky navigation
const initialCoords = section1.getBoundingClientRect();
console.log(initialCoords);

window.addEventListener('scroll', function () {
  console.log(window.scrollY);

  if (window.scrollY > initialCoords.top) nav.classList.add('sticky');
  else nav.classList.remove('sticky');
});

// Lifecycle DOM Events
document.addEventListener('DOMContentLoaded', function (e) {
  console.log('HTML parsed and DOM tree built!', e);
});

window.addEventListener('load', function (e) {
  console.log('Page fully loaded', e);
});

window.addEventListener('beforeunload', function (e) {
  e.preventDefault();
  console.log(e);
  e.returnValue = '';
});
```

DEFER AND ASYNC SCRIPT LOADING



REGULAR VS. ASYNC VS. DEFER

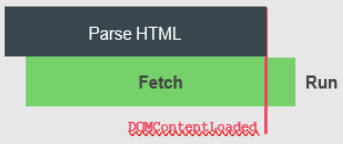
END OF BODY

- Scripts are fetched and executed ***after the HTML is completely parsed***
- **Use if you need to support old browsers**

You can, of course, use **different strategies for different scripts**. Usually a complete web applications includes more than just one script

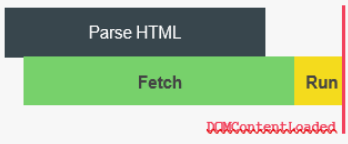
ASYNC IN HEAD

- Scripts are fetched ***asynchronously*** and executed ***immediately***
- Usually the `DOMContentLoaded` event waits for ***all*** scripts to execute, except for `async` scripts. So, `DOMContentLoaded` does ***not*** wait for an `async` script
- Scripts ***not*** guaranteed to execute in order
- **Use for 3rd-party scripts where order doesn't matter (e.g. Google Analytics)**



DEFER IN HEAD

- Scripts are fetched ***asynchronously*** and executed ***after the HTML is completely parsed***
- `DOMContentLoaded` event fires ***after*** `defer` script is executed
- Scripts are executed ***in order***
- **This is overall the best solution! Use for your own scripts, and when order matters (e.g. including a library)**



OOPS :

WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

Style of code, "how" we write and organize code

- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;
- We use objects to **model** (describe) real-world or abstract features;
- Objects may contain data (properties) and code (methods). By using objects, we pack **data and the corresponding behavior** into one block;
- In OOP, objects are **self-contained** pieces/blocks of code;
- Objects are **building blocks** of applications, and **interact** with one another;
- Interactions happen through a **public interface** (API): methods that the code **outside** of the object can access and use to communicate with the object;
- OOP was developed with the goal of **organizing** code, to make it **more flexible and easier to maintain** (avoid "spaghetti code").

E.g. user or todo list item

E.g. HTML component or data structure

Data

```
const user = {
  user: 'jonas',
  password: 'dk23s',

  login(password) {
    // Login logic
  },
  sendMessage(str) {
    // Sending logic
  }
}
```

Behaviour

CLASSES AND INSTANCES (TRADITIONAL OOP)

Like a blueprint from which we can create **new objects**

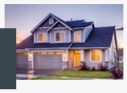
CLASS

```
User {
  user
  password
  email

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

Just a representation, **NOT** actual JavaScript syntax!

JavaScript does **NOT** support *real* classes like represented here

Instance

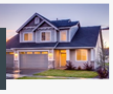


```
{
  user = 'jonas'
  password = 'dk23s'
  email = 'hello@jonas.io'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

New object created from the class. Like a *real* house created from an *abstract* blueprint

`new User('jonas')`

Instance



```
{
  user = 'mary'
  password = 'qwerty23'
  email = 'mary@test.com'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

`new User('mary')`

Instance

```
{
  user = 'steven'
  password = '5p8dz32dd'
  email = 'steven@tes.co'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

`new User('steven')`

☝ Conceptual overview: it works a bit **differently** in JavaScript. Still important to understand!

THE 4 FUNDAMENTAL OOP PRINCIPLES

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

The 4 fundamental principles of Object-Oriented Programming

🤔 *"How do we actually design classes? How do we model real-world data into classes?"*

PRINCIPLE 1: ABSTRACTION

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

```
Phone {
  charge
  volume
  voltage
  temperature

  homeBtn() {}
  volumeBtn() {}
  screen() {}
  verifyVolt() {}
  verifyTemp() {}
  vibrate() {}
  soundSpeaker() {}
  soundEar() {}
  frontCamOn() {}
  frontCamOff() {}
  rearCamOn() {}
  rearCamOff() {}
}
```

Real phone

Abstracted phone

```
Phone {
  charge
  volume

  homeBtn() {}
  volumeBtn() {}
  screen() {}
}
```

Details have been abstracted away

Do we *really need* all these low-level details?

👉 **Abstraction:** Ignoring or hiding details that **don't matter**, allowing us to get an **overview** perspective of the *thing* we're implementing, instead of messing with details that don't really matter to our implementation.

PRINCIPLE 2: ENCAPSULATION

Abstraction

Encapsulation

Inheritance

Polymorphism

Encapsulation: Keeping properties and methods **private** inside the class, so they are **not accessible from outside the class**. Some methods can be **exposed** as a public interface (API).

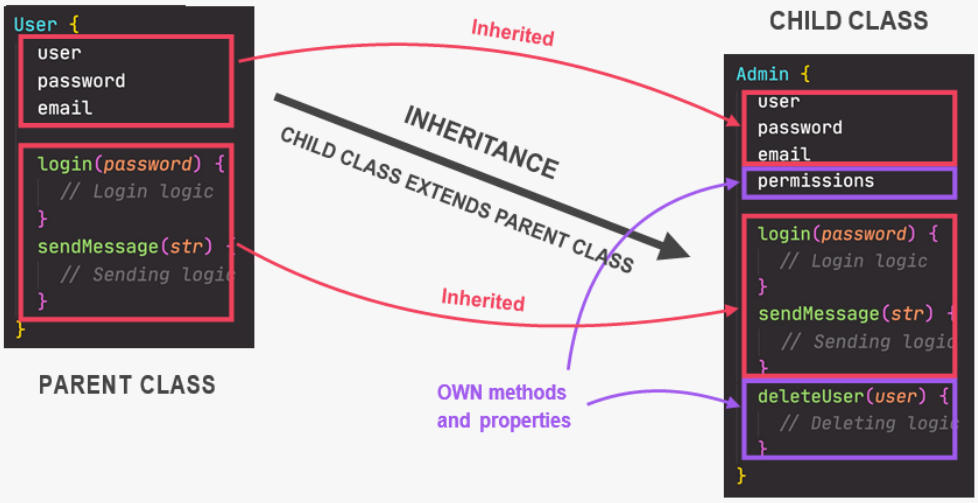
PRINCIPLE 3: INHERITANCE

Abstraction

Encapsulation

Inheritance

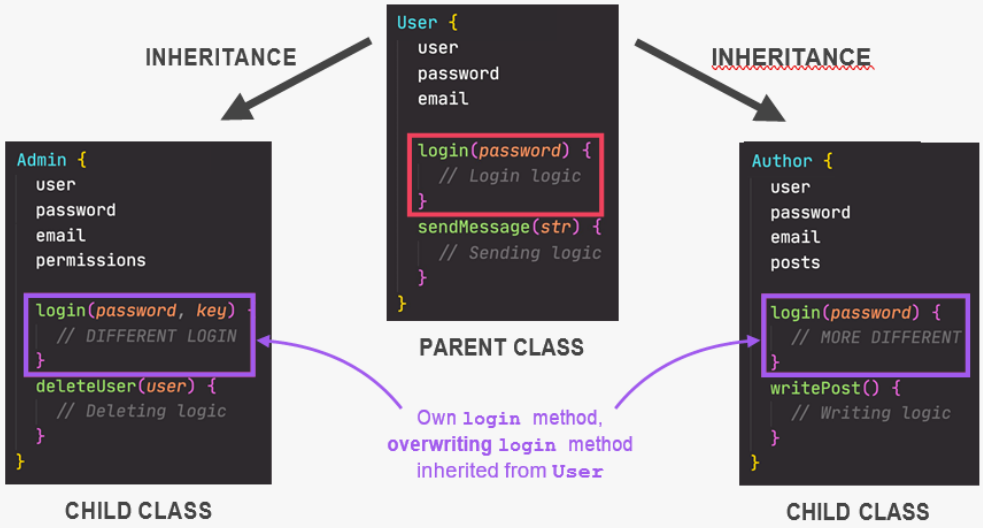
Polymorphism



Inheritance: Making all properties and methods of a certain class **available to a child class**, forming a hierarchical relationship between classes. This allows us to **reuse common logic** and to model real-world relationships.

PRINCIPLE 4: POLYMORPHISM

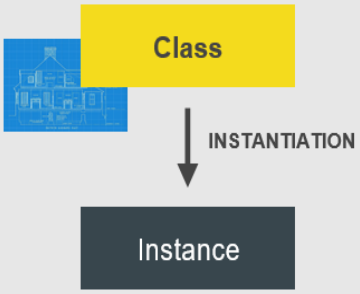
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **Polymorphism:** A child class can **overwrite** a method it inherited from a parent class [it's more complex that that, but enough for our purposes].

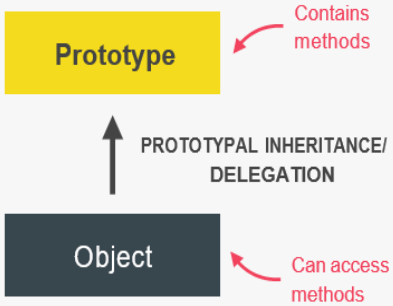
OOP IN JAVASCRIPT: PROTOTYPES

"CLASSICAL OOP": CLASSES



- 👉 Objects (instances) are **instantiated** from a class, which functions like a blueprint;
- 👉 Behavior (methods) is **copied** from class to all instances.

OOP IN JS: PROTOTYPES



- 👉 Objects are **linked** to a prototype object;
- 👉 **Prototypal inheritance:** The prototype contains methods (behavior) that are **accessible to all objects linked to that prototype;**
- 👉 Behavior is **delegated** to the linked prototype object.

👉 **Example: Array**

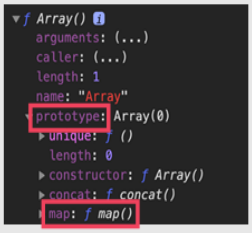
```
const num = [1, 2, 3];
num.map(v => v * 2);
```

MDN web docs

```
Array.prototype.keys()
Array.prototype.lastIndexOf()
Array.prototype.map()
```

Array.prototype is the **prototype** of all array objects we create in JavaScript

Therefore, **all** arrays have access to the map method!



3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JAVASCRIPT

🤔 *"How do we actually create prototypes? And how do we link objects to prototypes? How can we create new objects, without having classes?"*

☝ **The 4 pillars of OOP are still valid!**

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

1 Constructor functions

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

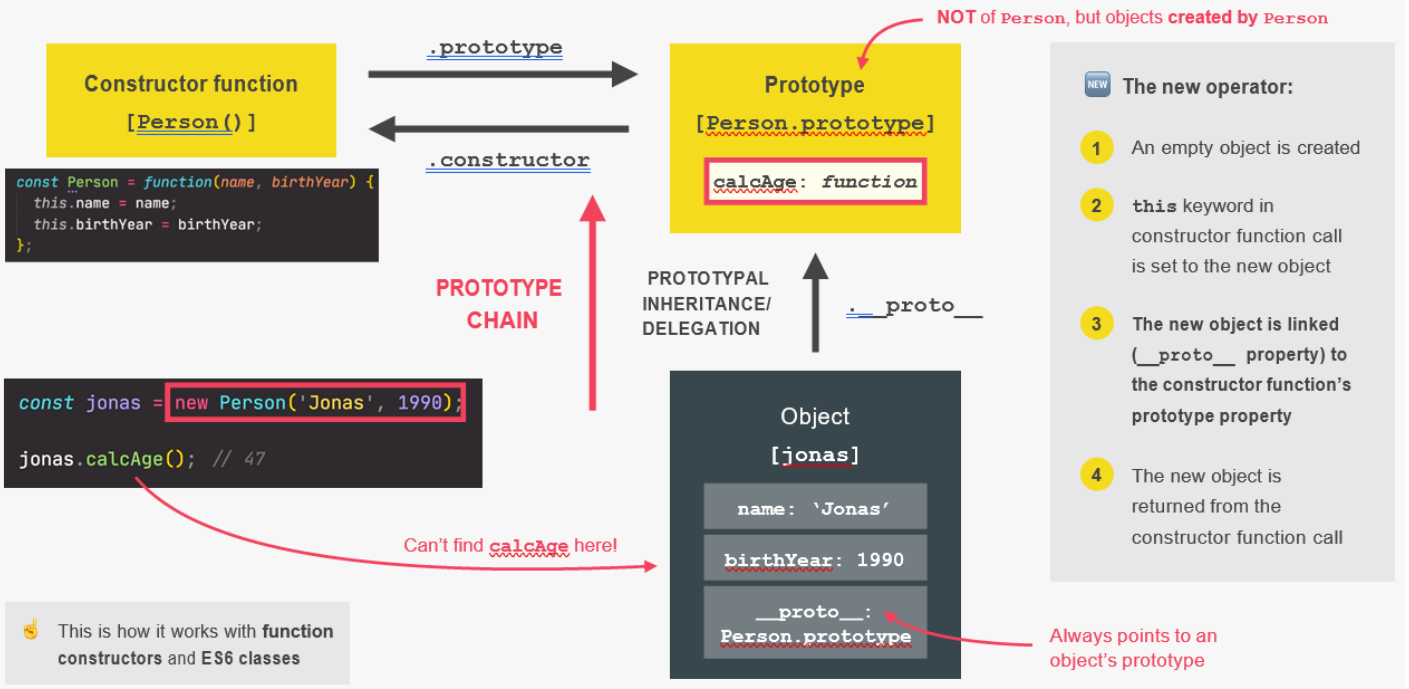
2 ES6 Classes

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

3 `Object.create()`

- The easiest and most straightforward way of linking an object to a prototype object.

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS

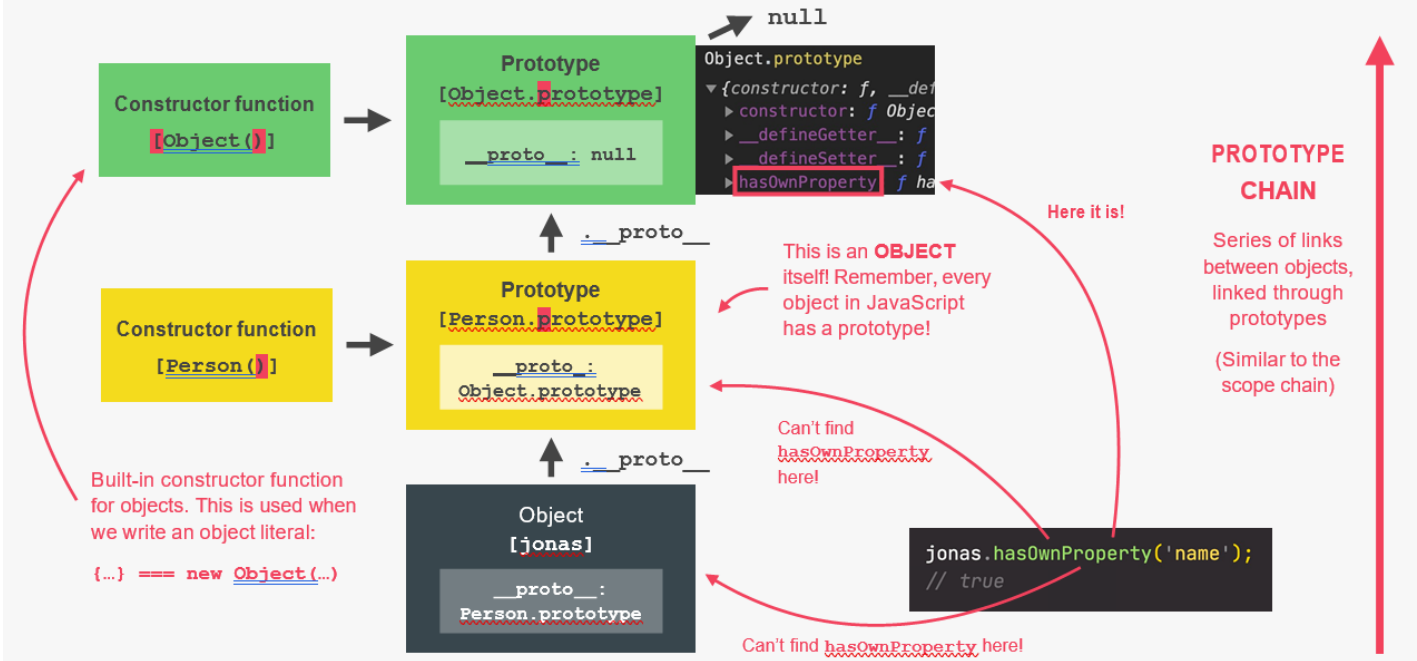


☝ This is how it works with **function constructors** and **ES6 classes**

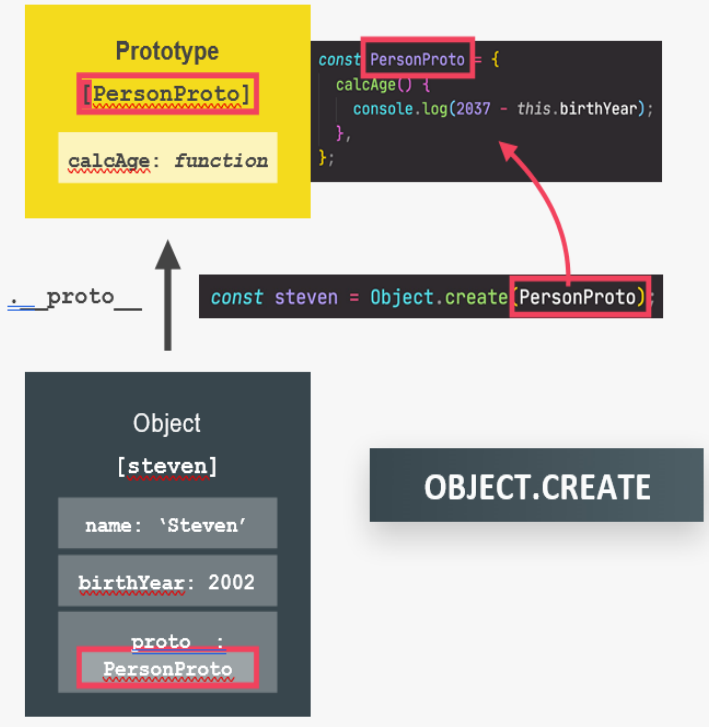
🆕 **The new operator:**

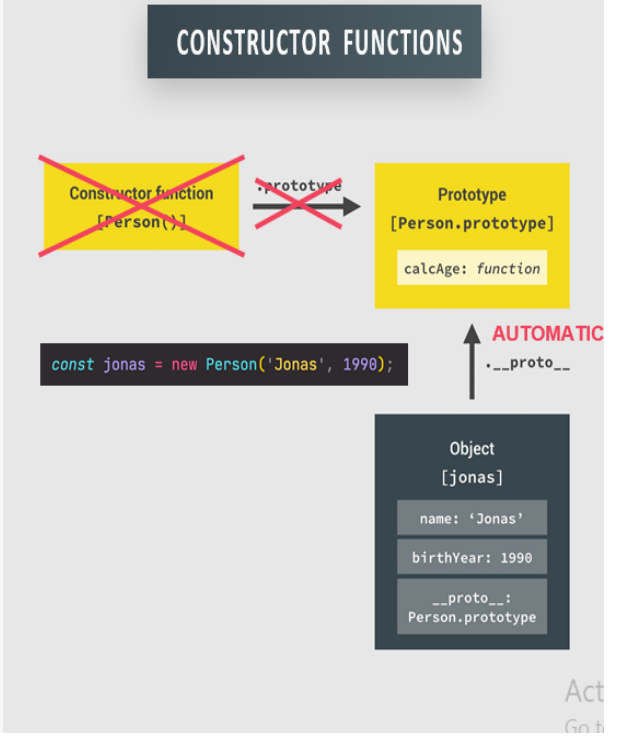
1. An empty object is created
2. `this` keyword in constructor function call is set to the new object
3. **The new object is linked (`__proto__` property) to the constructor function's prototype property**
4. The new object is returned from the constructor function call

THE PROTOTYPE CHAIN

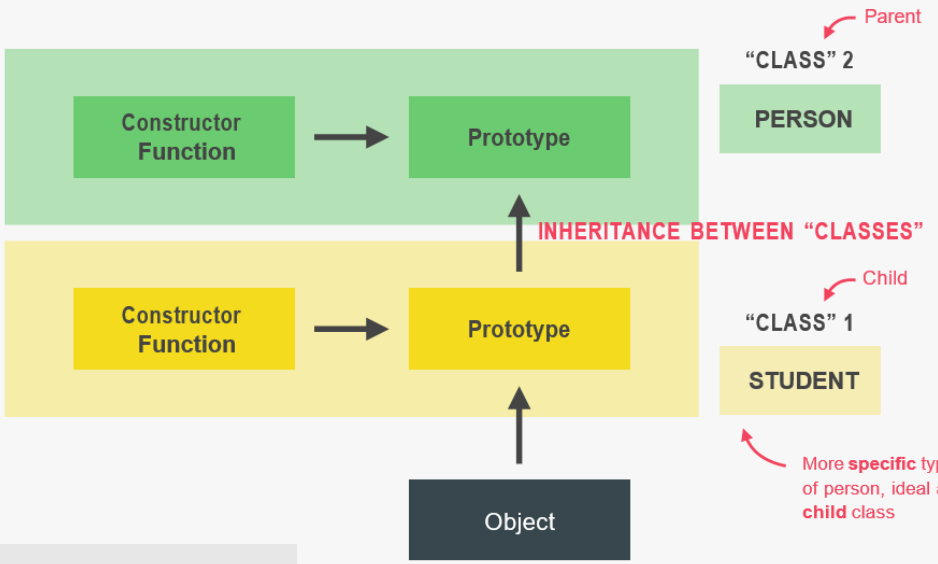


HOW OBJECT.CREATE WORKS





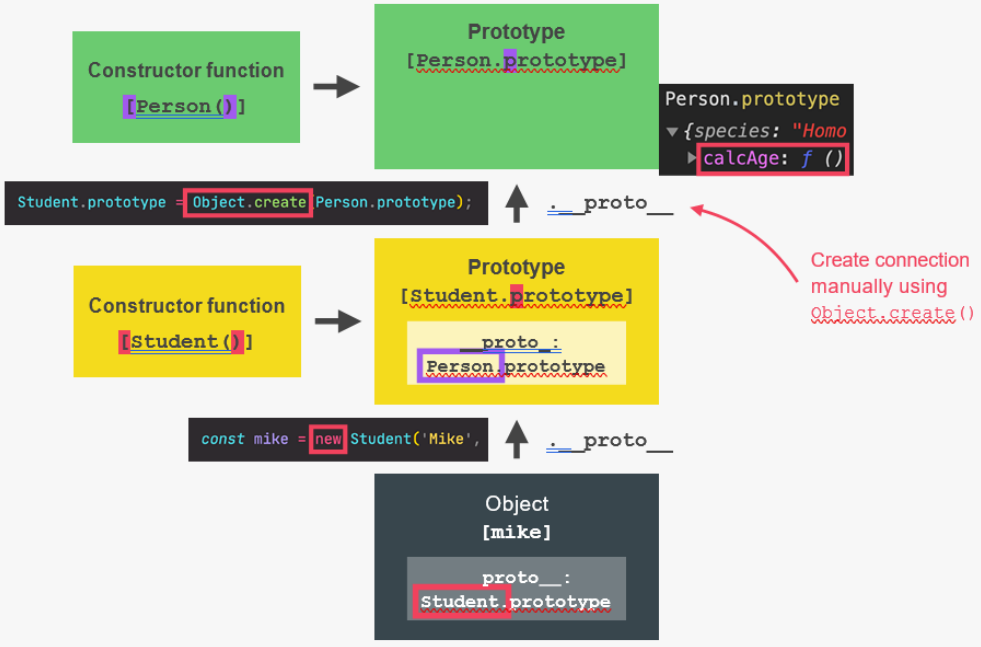
INHERITANCE BETWEEN "CLASSES"



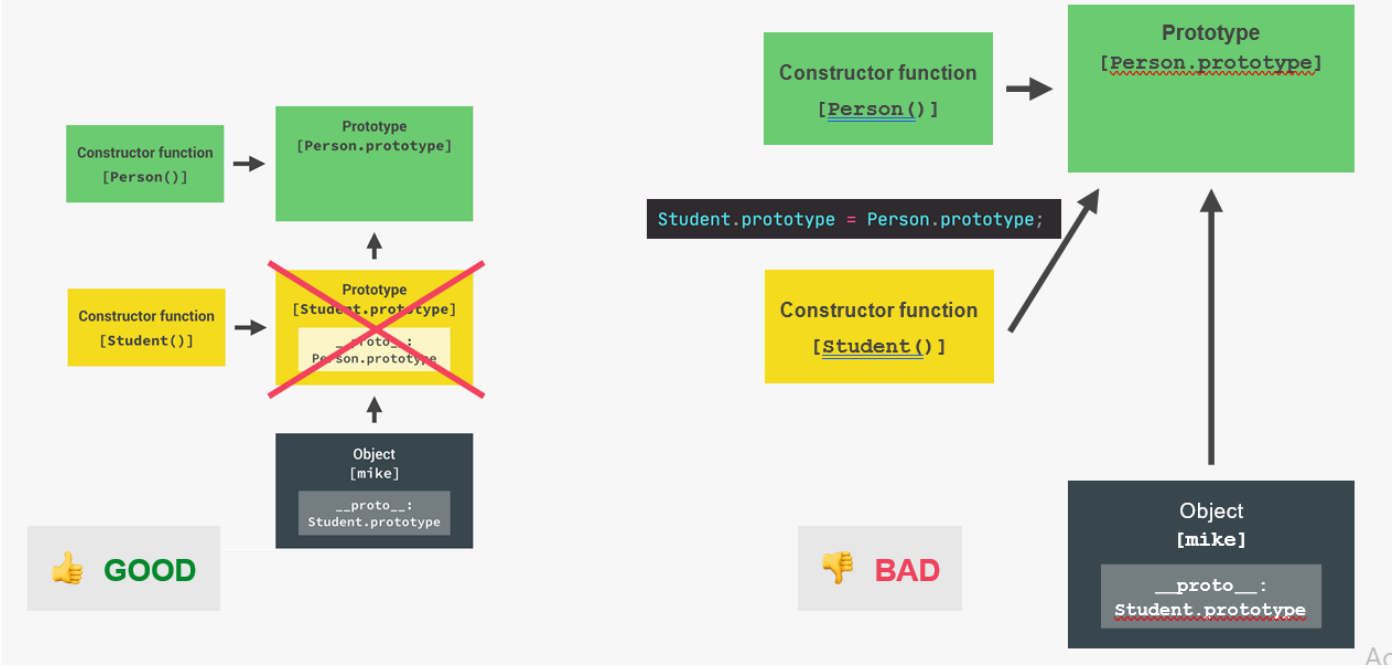
☝ Using class terminology here to make it easier to understand.

1. Constructor functions
2. ES6 Classes
3. `Object.create()`

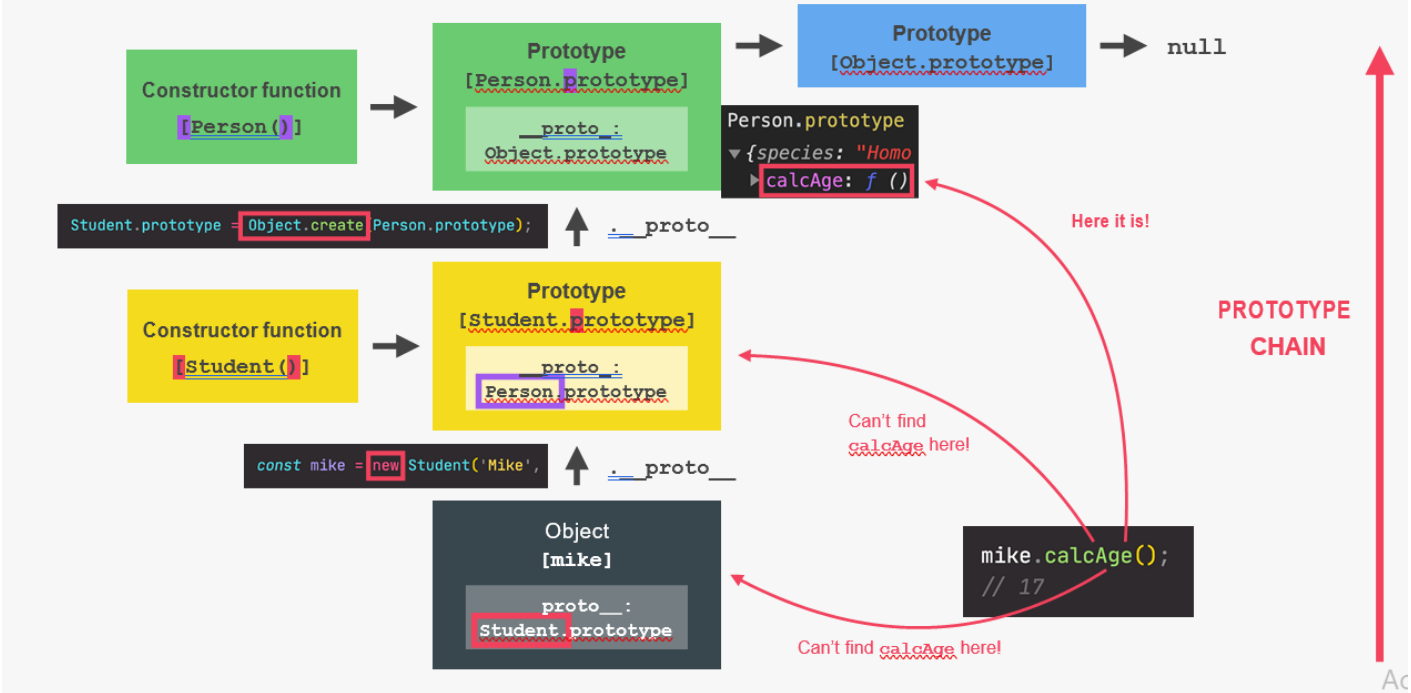
INHERITANCE BETWEEN "CLASSES"



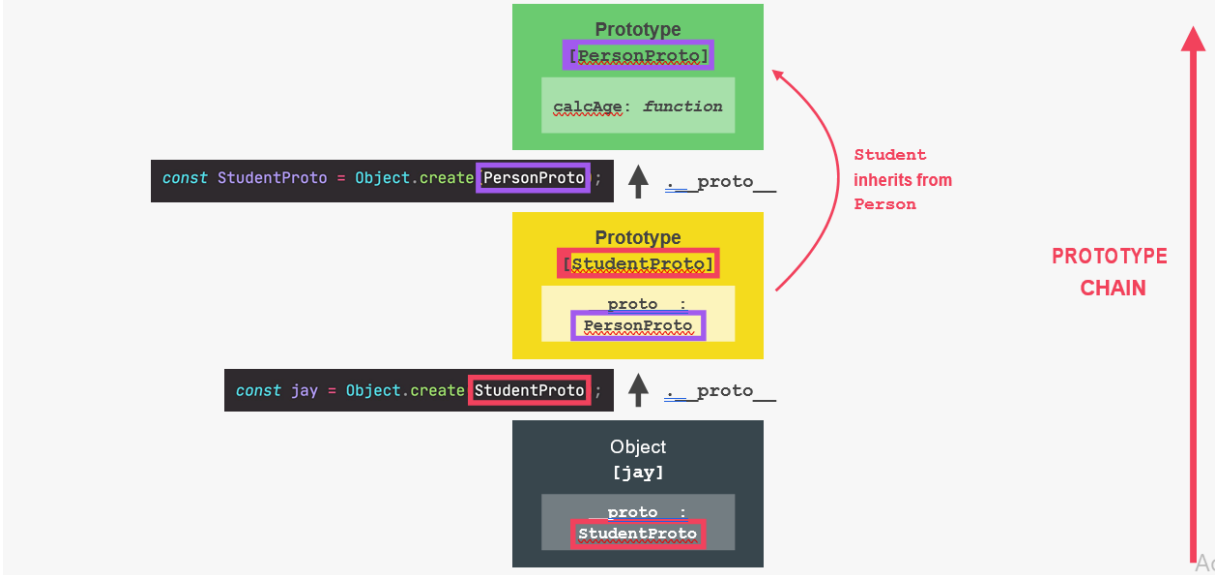
INHERITANCE BETWEEN "CLASSES"



INHERITANCE BETWEEN "CLASSES"



INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE



Public field (similar to property, available on created object)

Private fields (not accessible **outside** of class)

Static public field (available **only** on **class**)

Call to parent (super) class (necessary with **extend**). Needs to happen before accessing **this**

Instance property (available on created object)

Redefining private field

Public method

Referencing private field and method

Private method (÷ Might not yet work in your browser. "Fake" alternative: _ instead of #)

Getter method

Setter method (use _ to set property with same name as method, and also add getter)

Static method (available **only** on **class**. Can not access instance properties nor methods, only static ones)

Creating new object with **new** operator

```
class Student extends Person {
  university = 'University of Lisbon';
  #studyHours = 0;
  #course;
  static numSubjects = 10;

  constructor(fullName, birthYear, startYear, course) {
    super(fullName, birthYear);

    this.startYear = startYear;

    this.#course = course;
  }

  introduce() {
    console.log(`I study ${this.#course} at ${this.university}`);
  }

  study(h) {
    this.#makeCoffe();
    this.#studyHours += h;
  }

  #makeCoffe() {
    return 'Here is a coffe for you ';
  }

  get testScore() {
    return this._testScore;
  }

  set testScore(score) {
    this._testScore = score <= 20 ? score : 0;
  }

  static printCurriculum() {
    console.log(`There are ${this.numSubjects} subjects`);
  }
}

const student = new Student('Jonas', 2020, 2037, 'Medicine');
```

Parent class

Inheritance between classes, automatically sets prototype

Child class

Constructor method, called by **new** operator. Mandatory in regular class, might be omitted in a **child** class

- Classes are just "**syntactic sugar**" over constructor functions
- Classes are **not** hoisted
- Classes are **first-class** citizens
- Class body is always executed in **strict mode**

```
// Constructor Functions and the new Operator
const Person = function (firstName, birthYear) {
  // Instance properties
  this.firstName = firstName;
  this.birthYear = birthYear;

  // Never to this!
  // this.calcAge = function () {
  //   console.log(2037 - this.birthYear);
  // };
};

const jonas = new Person('Jonas', 1991);
console.log(jonas);

// 1. New {} is created
// 2. function is called, this = {}
// 3. {} linked to prototype
// 4. function automatically return {}

const matilda = new Person('Matilda', 2017);
const jack = new Person('Jack', 1975);

console.log(jonas instanceof Person);

Person.hey = function () {
  console.log('Hey there 👋');
  console.log(this);
};
Person.hey();

// Prototypes
console.log(Person.prototype);

Person.prototype.calcAge = function () {
  console.log(2037 - this.birthYear);
};

jonas.calcAge();
matilda.calcAge();

console.log(jonas.__proto__);
console.log(jonas.__proto__ === Person.prototype);

console.log(Person.prototype.isPrototypeOf(jonas));
console.log(Person.prototype.isPrototypeOf(matilda));
console.log(Person.prototype.isPrototypeOf(Person));

// .prototyeOfLinkedObjects

Person.prototype.species = 'Homo Sapiens';
console.log(jonas.species, matilda.species);

console.log(jonas.hasOwnProperty('firstName'));
console.log(jonas.hasOwnProperty('species'));
```

```
// Prototypal Inheritance on Built-In Objects
console.log(jonas.__proto__);
// Object.prototype (top of prototype chain)
console.log(jonas.__proto__.__proto__);
console.log(jonas.__proto__.__proto__.__proto__);

console.dir(Person.prototype.constructor);

const arr = [3, 6, 6, 5, 6, 9, 9]; // new Array === []
console.log(arr.__proto__);
console.log(arr.__proto__ === Array.prototype);

console.log(arr.__proto__.__proto__);

Array.prototype.unique = function () {
  return [...new Set(this)];
};

console.log(arr.unique());

const h1 = document.querySelector('h1');
console.dir(x => x + 1);

// ES6 Classes

// Class expression
// const PersonCl = class {}

// Class declaration
class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods
  // Methods will be added to .prototype property
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  greet() {
    console.log(`Hey ${this.fullName}`);
  }

  get age() {
    return 2037 - this.birthYear;
  }

  // Set a property that already exists
  set fullName(name) {
    if (name.includes(' ')) this._fullName = name;
    else alert(`${name} is not a full name!`);
  }

  get fullName() {
    return this._fullName;
```

```
  }

  // Static method
  static hey() {
    console.log('Hey there 👋');
    console.log(this);
  }
}

const jessica = new PersonCl('Jessica Davis', 1996);
console.log(jessica);
jessica.calcAge();
console.log(jessica.age);

console.log(jessica.__proto__ === PersonCl.prototype);

// PersonCl.prototype.greet = function () {
//   console.log(`Hey ${this.firstName}`);
// };
jessica.greet();

// 1. Classes are NOT hoisted
// 2. Classes are first-class citizens
// 3. Classes are executed in strict mode

const walter = new PersonCl('Walter White', 1965);
// PersonCl.hey();

// Setters and Getters
const account = {
  owner: 'Jonas',
  movements: [200, 530, 120, 300],

  get latest() {
    return this.movements.slice(-1).pop();
  },

  set latest(mov) {
    this.movements.push(mov);
  },
};

console.log(account.latest);

account.latest = 50;
console.log(account.movements);

// Object.create
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
```

```
  },
};

const steven = Object.create(PersonProto);
console.log(steven);
steven.name = 'Steven';
steven.birthYear = 2002;
steven.calcAge();

console.log(steven.__proto__ === PersonProto);

const sarah = Object.create(PersonProto);
sarah.init('Sarah', 1979);
sarah.calcAge();

// Inheritance Between "Classes": Constructor Functions

const Person = function (firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
};

Person.prototype.calcAge = function () {
  console.log(2037 - this.birthYear);
};

const Student = function (firstName, birthYear, course) {
  Person.call(this, firstName, birthYear);
  this.course = course;
};

// Linking prototypes
Student.prototype = Object.create(Person.prototype);

Student.prototype.introduce = function () {
  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};

const mike = new Student('Mike', 2020, 'Computer Science');
mike.introduce();
mike.calcAge();

console.log(mike.__proto__);
console.log(mike.__proto__.__proto__);

console.log(mike instanceof Student);
console.log(mike instanceof Person);
console.log(mike instanceof Object);

Student.prototype.constructor = Student;
console.log(Student.prototype.constructor);

// Inheritance Between "Classes": ES6 Classes

class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
```

```
    this.birthYear = birthYear;
  }

  // Instance methods
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  greet() {
    console.log(`Hey ${this.fullName}`);
  }

  get age() {
    return 2037 - this.birthYear;
  }

  set fullName(name) {
    if (name.includes(' ')) this._fullName = name;
    else alert(`${name} is not a full name!`);
  }

  get fullName() {
    return this._fullName;
  }

  // Static method
  static hey() {
    console.log('Hey there 👋');
  }
}

class StudentCl extends PersonCl {
  constructor(fullName, birthYear, course) {
    // Always needs to happen first!
    super(fullName, birthYear);
    this.course = course;
  }

  introduce() {
    console.log(`My name is ${this.fullName} and I study ${this.course}`);
  }

  calcAge() {
    console.log(
      `I'm ${
        2037 - this.birthYear
      } years old, but as a student I feel more like ${
        2037 - this.birthYear + 10
      }`
    );
  }
}

const martha = new StudentCl('Martha Jones', 2012, 'Computer Science');
martha.introduce();
martha.calcAge();
```

```
// Inheritance Between "Classes": Object.create

const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  },
};

const steven = Object.create(PersonProto);

const StudentProto = Object.create(PersonProto);
StudentProto.init = function (firstName, birthYear, course) {
  PersonProto.init.call(this, firstName, birthYear);
  this.course = course;
};

StudentProto.introduce = function () {
  // BUG in video:
  // console.log(`My name is ${this.fullName} and I study ${this.course}`);

  // FIX:
  console.log(`My name is ${this.firstName} and I study ${this.course}`);
};

const jay = Object.create(StudentProto);
jay.init('Jay', 2010, 'Computer Science');
jay.introduce();
jay.calcAge();

// Encapsulation: Protected Properties and Methods
// Encapsulation: Private Class Fields and Methods

// 1) Public fields
// 2) Private fields
// 3) Public methods
// 4) Private methods
// (there is also the static version)

class Account {
  // 1) Public fields (instances)
  locale = navigator.language;

  // 2) Private fields (instances)
  #movements = [];
  #pin;

  constructor(owner, currency, pin) {
    this.owner = owner;
    this.currency = currency;
    this.#pin = pin;

    // Protected property
```

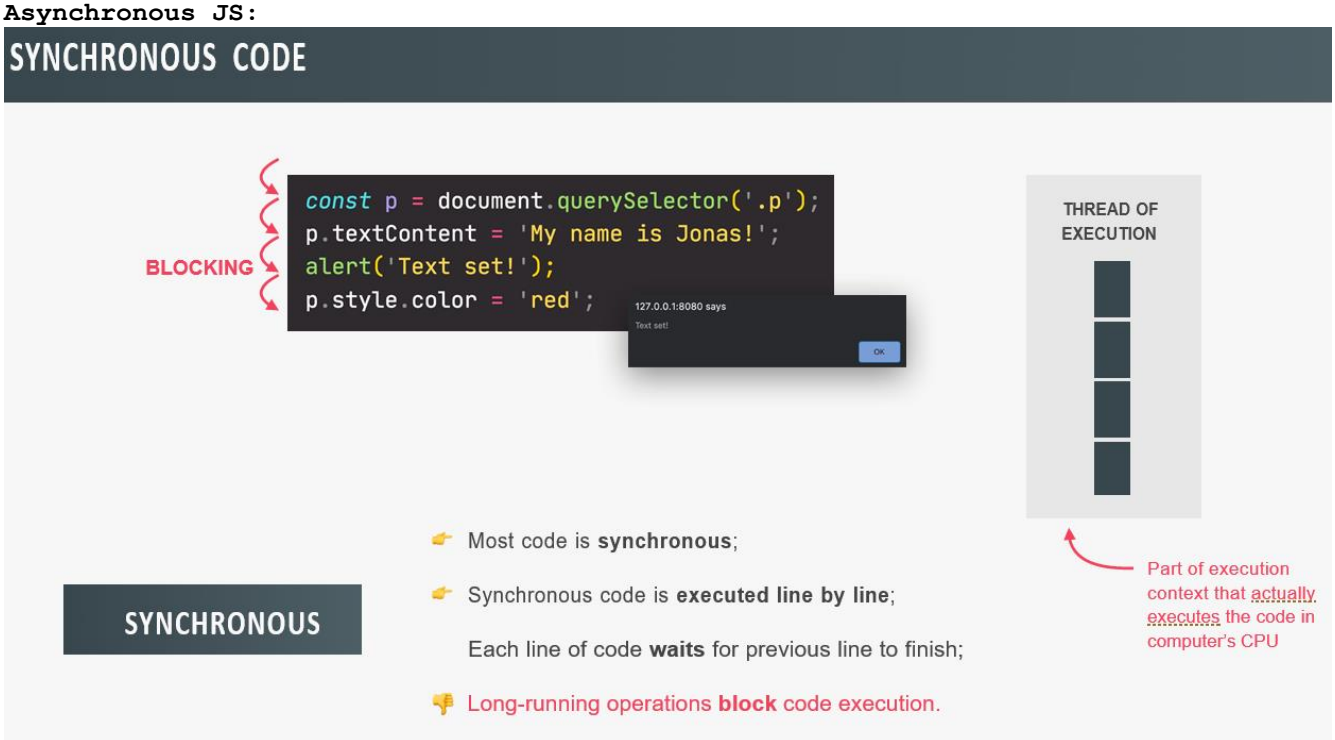
```
    // this._movements = [];
    // this.locale = navigator.language;

    console.log(`Thanks for opening an account, ${owner}`);
  }

  // 3) Public methods

  // Public interface
  getMovements() {
    return this.#movements;
  }

  deposit(val) {
    this.#movements.push(val);
    return this;
  }

  withdraw(val) {
    this.deposit(-val);
    return this;
  }

  requestLoan(val) {
    // if (this.#approveLoan(val)) {
    if (this._approveLoan(val)) {
      this.deposit(val);
      console.log(`Loan approved`);
      return this;
    }
  }

  static helper() {
    console.log('Helper');
  }

  // 4) Private methods
  // #approveLoan(val) {
  _approveLoan(val) {
    return true;
  }
}

const acc1 = new Account('Jonas', 'EUR', 1111);

// acc1._movements.push(250);
// acc1._movements.push(-140);
// acc1.approveLoan(1000);

acc1.deposit(250);
acc1.withdraw(140);
acc1.requestLoan(1000);
console.log(acc1.getMovements());
console.log(acc1);
Account.helper();

// console.log(acc1.#movements);
```

```
// console.log(acc1.#pin);
// console.log(acc1.#approveLoan(100));

// Chaining
acc1.deposit(300).deposit(500).withdraw(35).requestLoan(25000).withdraw(4000)
;
console.log(acc1.getMovements());
```

Local Storage:

```
  _setLocalStorage() {
    localStorage.setItem('workouts', JSON.stringify(this.#workouts));
  }

  _getLocalStorage() {
    const data = JSON.parse(localStorage.getItem('workouts'));

    if (!data) return;

    this.#workouts = data;

    this.#workouts.forEach(work => {
      this._renderWorkout(work);
    });
  }

  reset() {
    localStorage.removeItem('workouts');
    location.reload();
  }
}
```

Asynchronous JS:



ASYNCHRONOUS CODE

Asynchronous

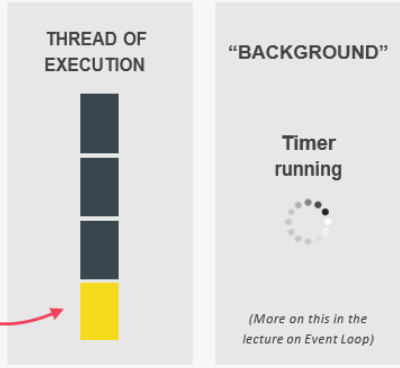
CALLBACK WILL RUN AFTER TIMER

```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

👉 **Example:** Timer with callback

```
[1, 2, 3].map(v => v * 2);
```

Callback does NOT automatically make code asynchronous!



Executed after all other code

ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

ASYNCHRONOUS CODE

Asynchronous

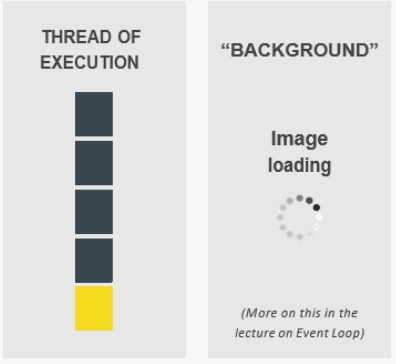
CALLBACK WILL RUN AFTER IMAGE LOADS

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

👉 **Example:** Asynchronous image loading with event and callback

👉 **Other examples:** Geolocation API or **AJAX calls**

`addEventListener` does NOT automatically make code asynchronous!



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 Asynchronous code is executed **after a task that runs in the "background" finishes**;
- 👍 Asynchronous code is **non-blocking**;
- 👉 Execution doesn't wait for an asynchronous task to finish its work;
- 👉 Callback functions alone do **NOT** make code asynchronous!

WHAT ARE AJAX CALLS?

AJAX

Asynchronous **J**avaScript **A**nd **X**ML: Allows us to communicate with remote web servers in an **asynchronous way**. With AJAX calls, we can **request data** from web servers dynamically.

WHAT IS AN API?

API

- **A**pplication **P**rogramming **I**nterface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;
- There are be many types of APIs in web development:
  - DOM API
  - Geolocation API
  - Own Class API
  - "Online" API ← Just "API"
- **"Online" API:** Application running on a server, that receives requests for data, and sends data back as response;
- We can build **our own** web APIs (requires back-end development, e.g. with node.js) or use **3rd-party** APIs.

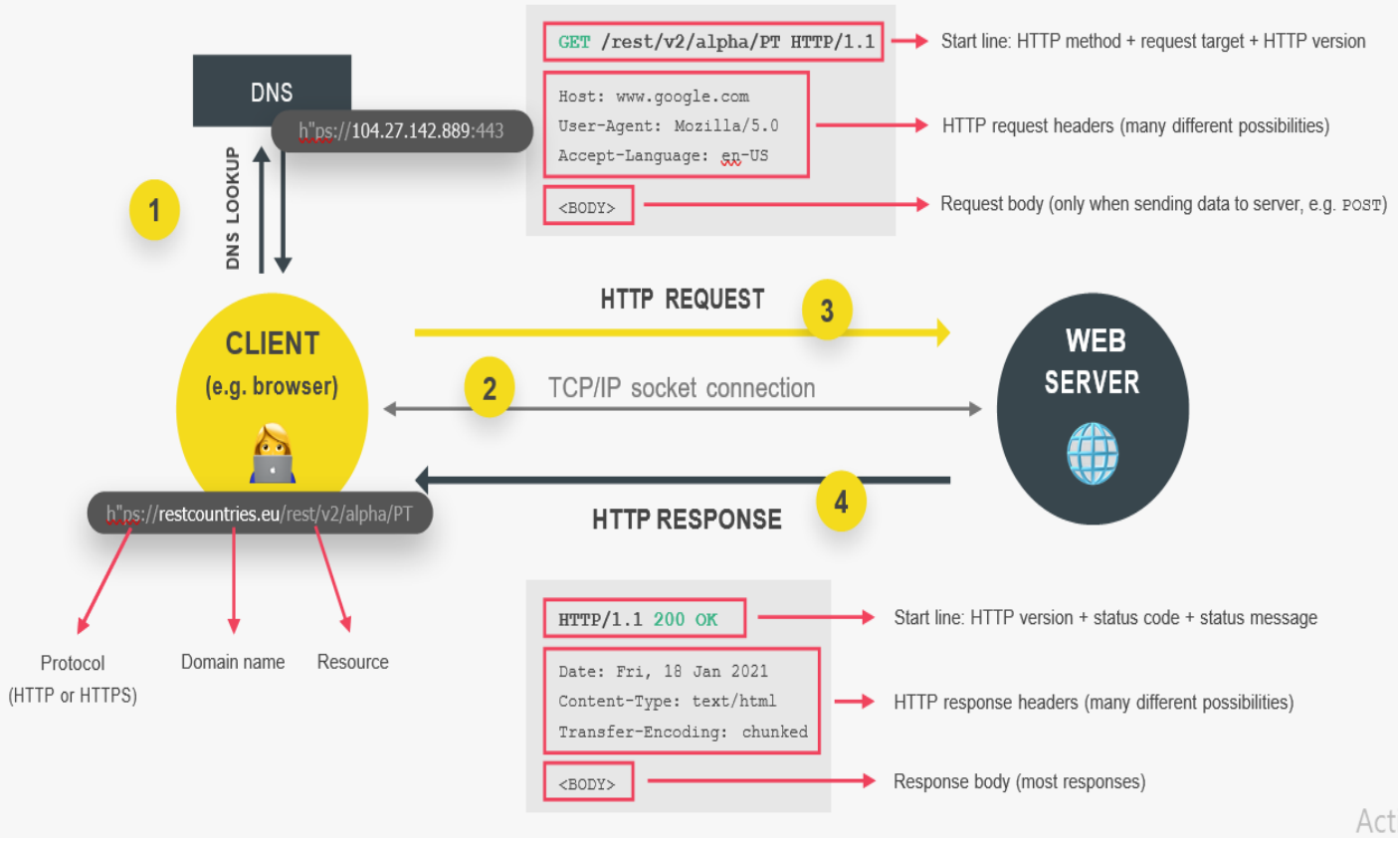
There is an API for *everything*

- Weather data
- Data about countries
- Flights data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps
- *Millions of possibilities...*

WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

👉 **Request-response model** or **Client-server architecture**

WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



WHAT ARE PROMISES?

PROMISE

- **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.
  - Less formal
- **Promise:** A container for an asynchronously delivered value.
  - Less formal
- **Promise:** A container for a future value.
  - **Example:** Response from AJAX call
- We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results;
- Instead of nesting callbacks, we can **chain promises** for a sequence of asynchronous operations: **escaping callback hell** 🎉

Promise that I will receive money if I guess correct outcome

🤞 I buy lottery ticket (promise) right now

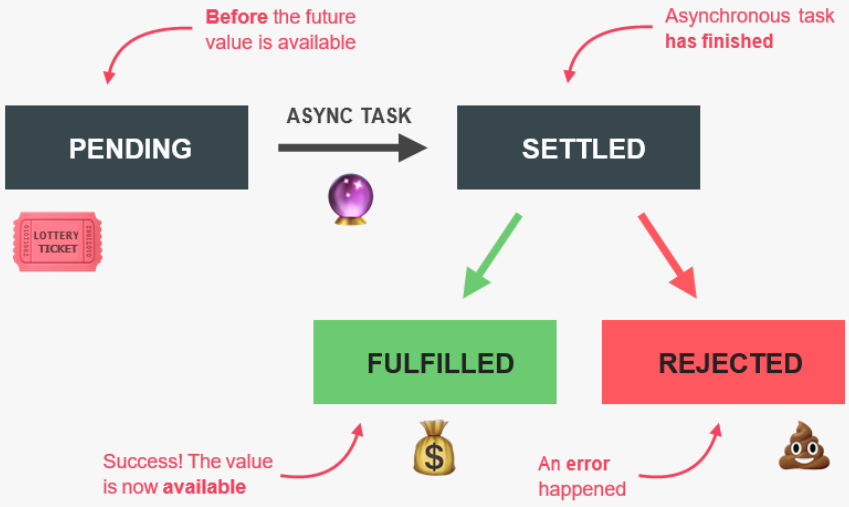
↓

🔮 Lottery draw happens asynchronously

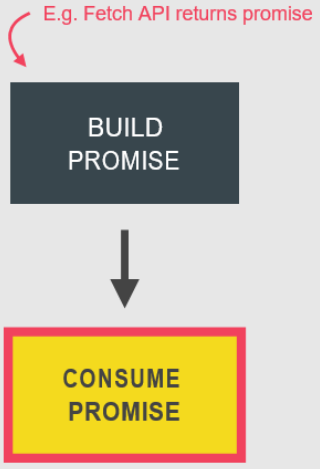
↓

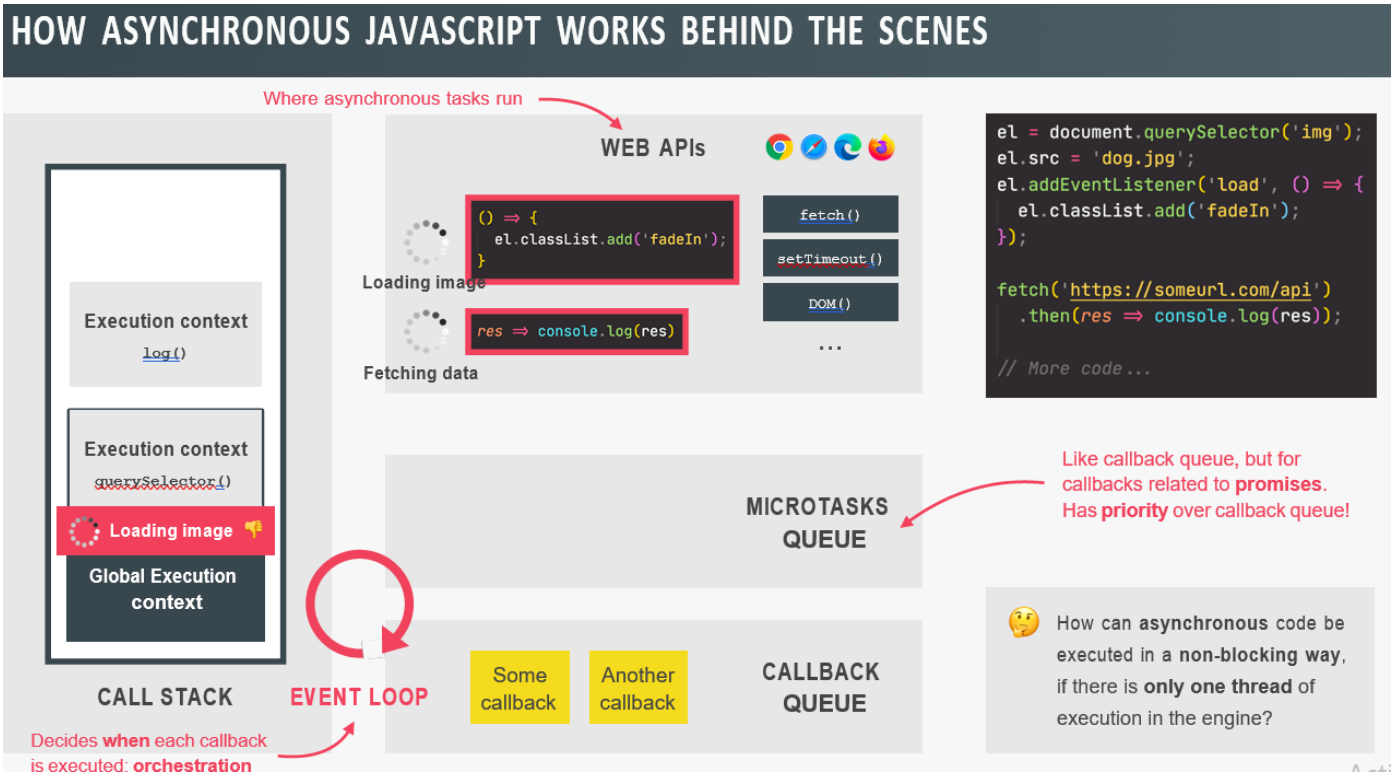
💰 If correct outcome, I receive money, because it was promised

THE PROMISE LIFECYCLE



- We are able **handle** these different states in our code!





Promisifying: Promisifying means to convert callback based asynchronous behavior to promise based.

Async & Await:

```
Const whereAmI = async function(){
        //consume promise(can have one or more await)
}
```

-This function is now an asynchronous function. So, a function that will basically keep running in the background while performing the code that is inside of it, then when this function is done it automatically returns a promise.

-In an async function we can use the await keyword to basically await for the result of the promise.

-So, basically await will stop code execution at this point of the function until the promise is fulfilled and so until the data is been fetched.

// **Our First AJAX Call: XMLHttpRequest**

```
const getCountryData = function (country) {
  const request = new XMLHttpRequest();
  request.open('GET', `https://restcountries.eu/rest/v2/name/${country}`);
  request.send();

  request.addEventListener('load', function () {
    const [data] = JSON.parse(this.responseText);
    console.log(data);

    const html = `
```

```
  <article class="country">
    <img class="country__img" src="${data.flag}" />
    <div class="country__data">
      <h3 class="country__name">${data.name}</h3>
      <h4 class="country__region">${data.region}</h4>
      <p class="country__row"><span>👫</span>${(
        +data.population / 1000000
      ).toFixed(1)} people</p>
      <p class="country__row"><span>🗣️</span>${data.languages[0].name}</p>
      <p class="country__row"><span>💰</span>${data.currencies[0].name}</p>
    </div>
  </article>
  `;
    countriesContainer.insertAdjacentHTML('beforeend', html);
    countriesContainer.style.opacity = 1;
  });
};

getCountryData('portugal');
getCountryData('usa');
getCountryData('germany');

// Welcome to Callback Hell

/*
const getCountryAndNeighbour = function (country) {
  // AJAX call country 1
  const request = new XMLHttpRequest();
  request.open('GET', `https://restcountries.eu/rest/v2/name/${country}`);
  request.send();

  request.addEventListener('load', function () {
    const [data] = JSON.parse(this.responseText);
    console.log(data);

    // Render country 1
    renderCountry(data);

    // Get neighbour country (2)
    const [neighbour] = data.borders;

    if (!neighbour) return;

    // AJAX call country 2
    const request2 = new XMLHttpRequest();
    request2.open('GET',
`https://restcountries.eu/rest/v2/alpha/${neighbour}`);
    request2.send();

    request2.addEventListener('load', function () {
      const data2 = JSON.parse(this.responseText);
      console.log(data2);

      renderCountry(data2, 'neighbour');
    });
  });
```

```
};

// getCountryAndNeighbour('portugal');
getCountryAndNeighbour('usa');

// Consuming Promises
// Chaining Promises
// Handling Rejected Promises
// Throwing Errors Manually

// const getCountryData = function (country) {
//   fetch(`https://restcountries.eu/rest/v2/name/${country}`)
//     .then(function (response) {
//       console.log(response);
//       return response.json();
//     })
//     .then(function (data) {
//       console.log(data);
//       renderCountry(data[0]);
//     });
// };

// const getCountryData = function (country) {
//   // Country 1
//   fetch(`https://restcountries.eu/rest/v2/name/${country}`)
//     .then(response => {
//       console.log(response);

//       if (!response.ok)
//         throw new Error(`Country not found (${response.status})`);

//       return response.json();
//     })
//     .then(data => {
//       renderCountry(data[0]);
//       // const neighbour = data[0].borders[0];
//       const neighbour = 'dfsdfdef';

//       if (!neighbour) return;

//       // Country 2
//       return fetch(`https://restcountries.eu/rest/v2/alpha/${neighbour}`);
//     })
//     .then(response => {
//       if (!response.ok)
//         throw new Error(`Country not found (${response.status})`);

//       return response.json();
//     })
//     .then(data => renderCountry(data, 'neighbour'))
//     .catch(err => {
//       console.error(`${err} 💥💥💥`);
//       renderError(`Something went wrong 💥💥 ${err.message}. Try again!`);
//     })
//     .finally(() => {
//       countriesContainer.style.opacity = 1;
```

```
//     });
// };

const getCountryData = function (country) {
  // Country 1
  getJSON(
    `https://restcountries.eu/rest/v2/name/${country}`,
    'Country not found'
  )
    .then(data => {
      renderCountry(data[0]);
      const neighbour = data[0].borders[0];

      if (!neighbour) throw new Error('No neighbour found!');

      // Country 2
      return getJSON(
        `https://restcountries.eu/rest/v2/alpha/${neighbour}`,
        'Country not found'
      );
    })

    .then(data => renderCountry(data, 'neighbour'))
    .catch(err => {
      console.error(`${err} 💥💥💥`);
      renderError(`Something went wrong 💥💥 ${err.message}. Try again!`);
    })
    .finally(() => {
      countriesContainer.style.opacity = 1;
    });
};

btn.addEventListener('click', function () {
  getCountryData('portugal');
});

// getCountryData('australia');
*/



// Promisifying the Geolocation API
const getPosition = function () {
  return new Promise(function (resolve, reject) {
    // navigator.geolocation.getCurrentPosition(
    //   position => resolve(position),
    //   err => reject(err)
    // );
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
};
// getPosition().then(pos => console.log(pos));

const whereAmI = function () {
  getPosition()
    .then(pos => {
```

```
      const { latitude: lat, longitude: lng } = pos.coords;

      return fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
    })
    .then(res => {
      if (!res.ok) throw new Error(`Problem with geocoding ${res.status}`);
      return res.json();
    })
    .then(data => {
      console.log(data);
      console.log(`You are in ${data.city}, ${data.country}`);

      return fetch(`https://restcountries.eu/rest/v2/name/${data.country}`);
    })
    .then(res => {
      if (!res.ok) throw new Error(`Country not found (${res.status})`);

      return res.json();
    })
    .then(data => renderCountry(data[0]))
    .catch(err => console.error(`${err.message} 💥`));
};

btn.addEventListener('click', whereAmI);


// Consuming Promises with Async/Await
// Error Handling With try...catch

const getPosition = function () {
  return new Promise(function (resolve, reject) {
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
};

// fetch(`https://restcountries.eu/rest/v2/name/${country}`).then(res =>
console.log(res))

const whereAmI = async function () {
  try {
    // Geolocation
    const pos = await getPosition();
    const { latitude: lat, longitude: lng } = pos.coords;

    // Reverse geocoding
    const resGeo = await
fetch(`https://geocode.xyz/${lat},${lng}?geoit=json`);
    if (!resGeo.ok) throw new Error('Problem getting location data');

    const dataGeo = await resGeo.json();
    console.log(dataGeo);

    // Country data
    const res = await fetch(
      `https://restcountries.eu/rest/v2/name/${dataGeo.country}`
    );
```

```
    // BUG in video:
    // if (!resGeo.ok) throw new Error('Problem getting country');

    // FIX:
    if (!res.ok) throw new Error('Problem getting country');

    const data = await res.json();
    console.log(data);
    renderCountry(data[0]);
  } catch (err) {
    console.error(`${err} 💥`);
    renderError(`💥 ${err.message}`);
  }
};
whereAmI();


// Running Promises in Parallel
const get3Countries = async function (c1, c2, c3) {
  try {
    // const [data1] = await getJSON(
    //   `https://restcountries.eu/rest/v2/name/${c1}`
    // );
    // const [data2] = await getJSON(
    //   `https://restcountries.eu/rest/v2/name/${c2}`
    // );
    // const [data3] = await getJSON(
    //   `https://restcountries.eu/rest/v2/name/${c3}`
    // );
    // console.log([data1.capital, data2.capital, data3.capital]);

    const data = await Promise.all([
      getJSON(`https://restcountries.eu/rest/v2/name/${c1}`),
      getJSON(`https://restcountries.eu/rest/v2/name/${c2}`),
      getJSON(`https://restcountries.eu/rest/v2/name/${c3}`),
    ]);
    console.log(data.map(d => d[0].capital));
  } catch (err) {
    console.error(err);
  }
};
get3Countries('portugal', 'canada', 'tanzania');


// Other Promise Combinators: race, allSettled and any
// Promise.race
(async function () {
  const res = await Promise.race([
    getJSON(`https://restcountries.eu/rest/v2/name/italy`),
    getJSON(`https://restcountries.eu/rest/v2/name/egypt`),
    getJSON(`https://restcountries.eu/rest/v2/name/mexico`),
  ]);
  console.log(res[0]);
})();
```

```
const timeout = function (sec) {
  return new Promise(function (_, reject) {
    setTimeout(function () {
      reject(new Error('Request took too long!'));
    }, sec * 1000);
  });
};

Promise.race([
  getJSON(`https://restcountries.eu/rest/v2/name/tanzania`),
  timeout(5),
])
  .then(res => console.log(res[0]))
  .catch(err => console.error(err));

// Promise.allSettled
Promise.allSettled([
  Promise.resolve('Success'),
  Promise.reject('ERROR'),
  Promise.resolve('Another success'),
]).then(res => console.log(res));

Promise.all([
  Promise.resolve('Success'),
  Promise.reject('ERROR'),
  Promise.resolve('Another success'),
])
  .then(res => console.log(res))
  .catch(err => console.error(err));

// Promise.any [ES2021]
Promise.any([
  Promise.resolve('Success'),
  Promise.reject('ERROR'),
  Promise.resolve('Another success'),
])
  .then(res => console.log(res))
  .catch(err => console.error(err));
```