Objetivos:

- Instalar Node.js e o gerenciador de pacotes JavaScript NPM (Node Package Manager). O npm roda como uma aplicação Node.js, então é necessário instalar o Node.js;
- Instalar o Visual Studio Code para editar os arquivos do projeto;
- ECMAScript - Versões da Linguagem JavaScript;
- Declaração de variáveis na linguagem JavaScript;
- Tipos de dados da Linguagem JavaScript;
- Tipos de Function: declaração de função, função anônima e Arrow Function;
- Arrays: métodos forEach, map e reduce;
- Objetos JavaScript;
- Classes: função de construtor, herança, getters e setters;
- JSON (JavaScript Object Notation);
- Instrução strict mode;
- Promise;
- Async e await;
- Require e exports;
- Import e export.

i. Instalar o Node.js e npm

A linguagem JavaScript é tipicamente suportada nos navegadores, isto é, do lado cliente. Porém para fazer uso dela no lado servidor teremos de instalar o Node.js. Acesse https://nodejs.org/en/download/ e faça o download do instalador na versão `.msi` para Windows. Após instalar ele criará a pasta `nodejs` (Figura 1) e na pasta `nodejs/node_modules/npm` estará a instalação do gerenciador de pacotes da linguagem JavaScript (npm – Node Package Manager), isto é, o npm será instalado juntamente com o Node.js.

Forneça os comandos da Figura 2 no CMD (prompt do DOS) para verificar as versões e, por consequência, verificar se as instalações foram efetivadas.

Diferenças entre os programas npx e npm:

- npx (eXecute - npm package runner) é uma ferramenta para executar pacotes Node, disponível no npm 5.2+. O npx será necessário para criarmos aplicações React. A seguir tem-se o comando para executar um pacote usando npx:

  ```
  npx packagename
  ```

- npm (manager) é uma ferramenta para ajudar na instalação de pacotes. Na prática ele não é capaz de fazer execuções, mas se você deseja executar algo usando npm, então é necessário especificar isso no arquivo package.json da sua aplicação Node. Para executar um pacote:

  ```
  npm run packagename
  ```

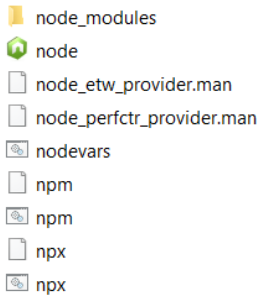

Figura 1 – Arquivos da pasta Node.js no computador. Deverá estar na pasta C:\Program Files\nodejs.

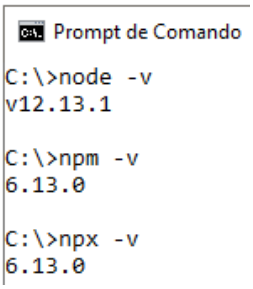

Figura 2 – Comandos para obter a versão do Node.js, npm e npx.

ii. Instalar o Visual Studio Code

O Visual Studio Code (VS Code) é um editor de código que nos ajuda a escrever arquivos de programas em diferentes linguagens de marcação e programação. Ele não é considerado uma IDE, pois inicialmente não integra debuggers e compiladores., mas é possível instalar plugins para auxiliar na edição do código (colorir e autocompletar).

Acesse https://code.visualstudio.com/ para baixar o instalador compatível com o seu SO. Após instalar o VS Code acesse o menu View > Extensions para instalar plugins, a seguir tem o nome de dois plugins que serão úteis:

- ESLint plugin para JavaScript (https://eslint.org/);
- Bootstrap v4 Snippets para auxiliar na digitação das classes Bootstrap 4.

Para testar o VS Code e o Node crie uma pasta qualquer no seu computador, aqui usaremos a pasta exemplos, e na sequência abra essa pasta no VS Code. A Figura 3 mostra como é possível criar arquivos usando a interface do VS Code e usar o terminal (prompt de comandos) para executar esses arquivos.

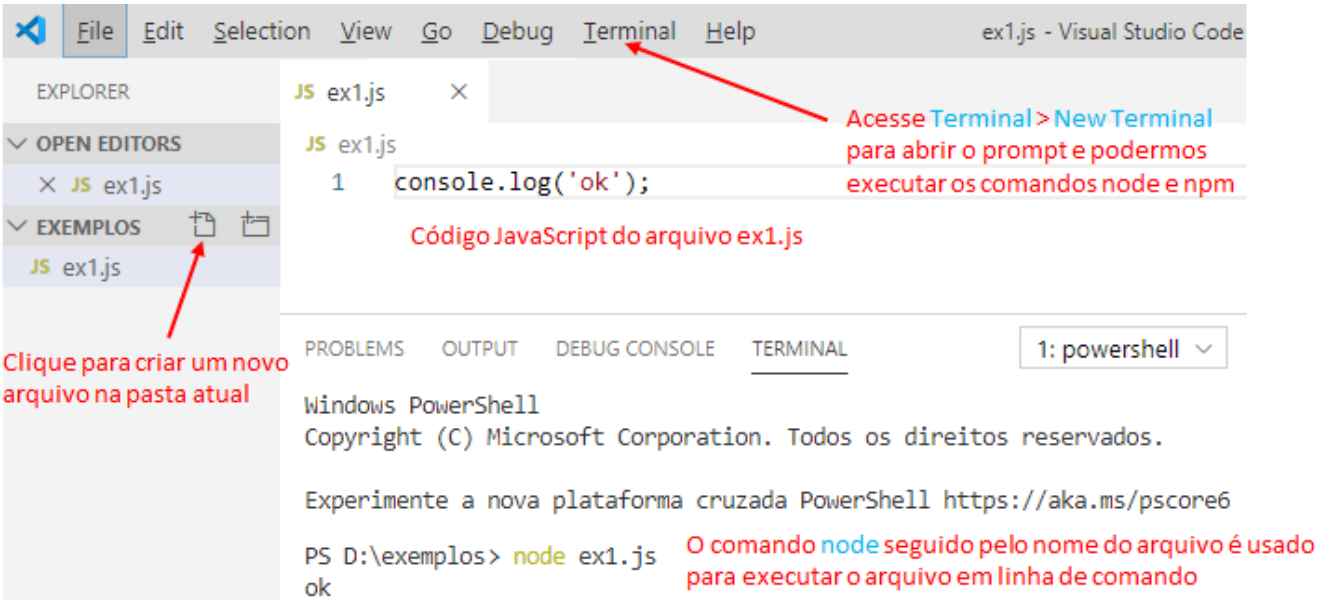

Figura 3 – Criar um arquivo JavaScript no VS Code e executar ele usando o comando node.

iii. Versões da Linguagem JavaScript

ECMAScript é uma especificação de linguagem de programação baseada em scripts, padronizada pela ECMA International (European Computer Manufactures Association). Apesar de já existir a 10ª versão – ECMAScript 2019, a maioria dos recursos suportados pelos navegadores vão até a ECMAScript 2016 (ES7) (fonte: https://en.wikipedia.org/wiki/ECMAScript).

Atualmente, padrões e normativas referentes à linguagem são mantidos pela ECMA-262 e ECMA-402, grupos criados na ECMA para a padronização do JavaScript e contam com participação de grandes empresas de tecnologia como Microsoft e Google, dentre outras.

O nome JavaScript ficou popular na comunidade, sendo o ECMAScript referenciado apenas para se determinar a versão da linguagem.

iv. Declaração de Variáveis na Linguagem JavaScript

A partir da versão ES6 podemos declarar uma variável usando const, let e var. As variáveis declaradas usando let e const possuem escopo de bloco e as variáveis declaradas usando var possuem escopo global. Uma variável declarada sem o termo const, let ou var, será considerada como se tivesse sido declarada usando var.

```
var a = 1;
let b = 2;
const c = 3;
aa = 10;
if( true ){
    var d = 4;
    let e = 5;
    const f = 6;
    g = 7; //é como se tivesse sido declarada usando var
    console.log(a + ' ' + b + ' ' + c + ' ' + d + ' ' + e + ' ' + f + ' ' + g);
}
//as variáveis 'e' e 'f' não estão disponíveis fora do bloco
console.log(a + ' ' + b + ' ' + c + ' ' + d + ' ' + g);
```

Escopo global e local (escopo de função)

As variáveis declaradas com var:

- Possuem escopo global se tiverem sido declaradas fora de uma função;
- Possuem escopo local se tiverem sido declaradas dentro de uma função.

As variáveis declaradas sem var, possuem escopo global independentemente de estarem dentro ou fora do bloco de uma função.

```
var a = 1; //possui escopo global
let b = 2;
const c = 3;
aa = 10;
function teste(){
    var d = 4; //possui o escopo da função
    let e = 5;
    const f = 6;
    g = 7; //possui escopo global
    //as variáveis declaradas globalmente podem ser acessadas dentro da função
    console.log(a + ' ' + b + ' ' + c + ' ' + d + ' ' + e + ' ' + f + ' ' + g);
}
```

```
teste();
//as variáveis 'd', 'e' e 'f' não estão disponíveis fora do bloco da função
console.log(a + ' ' + b + ' ' + c + ' ' + g);
```

Const

Como o nome sugere, const é usada para declarar variáveis cujo conteúdo não pode ser trocado. Porém, se a variável possuir o endereço/referência do conteúdo, assim como array ou objeto, daí os elementos do array e membros do objeto podem ser alterados. Por exemplo,

```
const vet = []; //a variável vet possui o endereço do array
//não gera erro, pois estamos adicionando um elemento no array que a variável vet está apontando
vet[0] = 10;
//gera erro, pois estaríamos trocando o conteúdo da variável vet
vet = [10,20,30,40];
console.log(vet); //não gera erro, pois é apenas a leitura da variável vet

const obj = {}; //a variável obj possui o endereço do objeto
obj.nome = 'Ana'; //o objeto passa a ter uma propriedade
obj = {}; //gera erro, pois estamos trocando o conteúdo da variável obj
console.log(obj);
```

v. Tipos de dados da Linguagem JavaScript

Cada linguagem de programação possui suas estruturas de dados internas (tipos de dados), mas geralmente elas diferem de uma linguagem para outra. Atualmente a última versão da ECMAScript define 8 tipos de dados (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures):

- 7 tipos de dados primitivos:
  - Number valores entre $-(2^{53}-1)$ e $(2^{53}-1)$
  - BigInt
  - String – cada elemento da string é um caractere representado por um inteiro sem sinal de 16 bits. Diferentemente de linguagens como C, as strings JavaScript são imutáveis, isto é, uma vez criada elas não podem ser modificadas:

    ```
    let a = 'oi';
    //a string anterior não será alterada,
    //mas será criada uma 3ª string e será atribuída a variável a
    a = 'oi' + 'e';
    ```
  - Boolean
  - Undefined
  - Null
  - Symbol – cada symbol retornado pela função Symbol é único. Esse valor pode ser utilizado em propriedades de objetos (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)
- Object é um conteúdo na memória que é referenciado por um endereço de memória.

A última versão da ECMAScript não considera o tipo de dado function, pois na prática function é um tipo de object que pode ser chamado para executar um bloco de código. Exemplos de tipos de dados:

```
console.log(typeof 1); //inteiro é tipo number
console.log(typeof 1.5); //real é tipo number
console.log(typeof NaN); //not-a-number é tipo number
console.log(typeof -Infinity); //+ ou - infinito é tipo number
console.log(typeof (2n ** 53n)); //9007199254740992n é tipo bigint
console.log(typeof 'oi'); //texto é tipo string
console.log(typeof true); //booleano é tipo boolean
console.log(typeof {}); //objeto é tipo object
console.log(typeof function(){}); //função é tipo function
console.log(typeof []); //array é tipo object
console.log(typeof undefined); //não definido é tipo undefined
console.log(typeof null); //endereço nulo é tipo object
console.log(typeof Symbol('oi')) //a função Symbol retorna um valor do tipo symbol
```

A linguagem JavaScript possui tipagem dinâmica ou *loosely typed*, desta forma, se o conteúdo da variável for alterado, o tipo de dado da variável será o tipo de dado do novo conteúdo da variável.

vi. Functions

Na linguagem JavaScript existem três formas de definir uma função. Para mais detalhes acesse https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions.

1 - Declaração de função

A forma mais tradicional é usando a palavra reservada function seguida pelo nome da função. A função somar a seguir recebe 2 argumentos e retorna a soma deles:

```
function somar(a, b){
    return a + b;
}
```

As variáveis a e b são locais ao corpo da função somar.

Constitui erro usar os termos let, var e const na declaração da variável que irá receber os valores passados como argumento:

```
function somar(let a, let b){
    return a + b;
}
```

A chamada da função é igual a de outras linguagens de programação:

```
console.log( somar(3,2) );
```

Parâmetro Default

Os parâmetros de uma função possuem valor padrão undefined, desta forma, a chamada a seguir não dará erro, mas irá retornar NaN (Not-a-Number):

```
console.log( somar() ); // retorna NaN
```

A partir da versão ES6 podemos atribuir valores default para os parâmetros da função. No exemplo a seguir a chamada da função irá retornar 5, pois o parâmetro a irá receber 5 e o parâmetro b ficará com valor 0.

```
function somar(a = 0, b = 0){
    return a + b;
```

```
}
console.log( somar(5) ); // retorna 5
```

Atribuir uma função como conteúdo de uma variável

Podemos atribuir uma função a uma variável, assim como no exemplo a seguir, porém o nome de função somar deixa de existir.

```
let sum = function somar(a, b){
    return a + b;
}
console.log( somar(3,2) ); // possui erro, somar não está definido
console.log( sum(3,2) ); // chamada correta da função
```

Variáveis declaradas dentro da função são locais, isto é, possuem como escopo apenas a própria função, porém variáveis declaradas fora da função podem ser acessadas dentro da função. No exemplo a seguir as variáveis a e b podem ser acessadas dentro da função somar:

```
let a = 5, b = 10;
function somar(){
    return a + b;
}
console.log( somar() ); // resultado será 15
```

Closure: uma função pode ser declarada dentro de outra função. A função interna pode usar as variáveis da função externa, mas o oposto não está disponível. No exemplo a seguir se mudarmos a instrução para `return soma() + w`, causará erro, pois a variável w está disponível apenas dentro da função soma.

```
let t = 2;
function somatorio(x){
    let y = 1.2;
    function soma(){
        let w = 5;
        return t + x + y + w;
    }
    return soma();
}
console.log(somatorio(1)); //resultado é 9.2
```

Rest parameter

A sintaxe rest parameter permite passar um número indefinido de parâmetros e eles são recebidos como se fossem um array. No exemplo a seguir, a variável g irá receber um número diferente de elementos em cada chamada da função.

```
function somar(...g){
    let r = 0;
    for(let i = 0; i < g.length; i++){
        r += g[i];
    }
    return r;
}
console.log( somar() );  // retorna 0
```

```
console.log( somar(3) ); // retorna 3
console.log( somar(3,2,5) ); // retorna 10
```

2- Função Anônima (function expression)

É uma função que não possui um nome, mas ela pode ser atribuída a uma variável e assim ser invocada através dessa variável. No exemplo a seguir, o conteúdo da variável diff será a função anônima. Desta forma, para chamar a função precisamos usar o nome da variável.

```
const diff = function(a, b){
    return a - b;
}
console.log( diff(3,2) );
```

É possível passar o conteúdo da variável para outra variável, mas lembre-se que o conteúdo da variável diff é apenas a referência (endereço) para a função, então na prática copiou-se apenas o endereço da função para a variável calc.

```
let calc = diff;
console.log( calc(5,1) );
```

Observação: as duas instruções a seguir são diferentes:

```
let a = diff(5,3);
let b = diff;
```

No 1º caso estamos invocando a função para ela ser executada e o resultado será colocado na variável a.

No 2º caso estamos apenas lendo o endereço da função na memória sem fazer qualquer execução, ou seja, a variável b irá receber o endereço da função na memória.

3 - Arrow Function

Possui uma sintaxe mais curta quando comparada com expression function. Arrow functions são sempre anônimas. A seguir tem-se quatro declarações distintas de arrow function:

```
const mult = (a, b) => {
    return a * b;
}

const div = (a, b) => a / b;

const pow = (a, b) => { return  a ** b };

const msg = txt => console.log(txt);

console.log( mult(3,2) ); //retorna 6
console.log( div(3,2) );  //retorna 1.5
console.log( pow(3,2) );  //retorna 9
msg('Bom dia'); //imprime no console Bom dia
```

Quando a função possui no corpo apenas a instrução return, então podemos retirar o return e as chaves, assim como fizemos na função div. Mas se adicionarmos as chaves, como fizemos na função pow, então a função precisará ter a instrução return.

Quando a função recebe somente 1 parâmetro, como fizemos na função msg, então não precisamos dos parênteses envolvendo o parâmetro.

Além da sintaxe mais curta, arrow function possui a vantagem de não fazer vinculação (bind) para this, mas isso será explicado posteriormente com objetos e classes.

vii. Arrays

Array é um objeto global do JavaScript usado na construção de arrays - objetos de alto nível semelhante a lista, pois eles podem ser redimensionados na linguagem JavaScript.

Um array pode ser criado vazio e posteriormente receber elementos. O array a seguir inicia vazio e é redimensionado, veja como exemplo o código a seguir e o seu resultado.

```
//criar um array vazio e coloca o endereço dele na variável v
let v = [];
v[0] = 10; //redimensiona o array para ter a 1a posição
//redimensiona o array, as posições não preenchidas
//recebem valor undefined
v[5] = 20;
console.log(v[1])
console.log(v);
```

```
undefined
[ 10, <4 empty items>, 20 ]
```

A forma tradicional de percorrer os elementos de um array é usando o for:

```
let w = [4, 2, 8, 5]; //cria um array e coloca os elementos nele
for(let i = 0; i < w.length; i++){
    console.log(w[i]);
}
```

Criar cópia do array

Para criar uma cópia do array:

```
let w = [4, 2, 8, 5];
let z = [...w]; //cria uma cópia do array w
w[1] = 20; //altera um elemento do array w sem alterar o array z
console.log(w); //resultado [ 4, 20, 8, 5 ]
console.log(z); //resultado [ 4, 2, 8, 5 ]
```

Remover elementos do array

O método splice(índice, quant) é usado para remover quant elementos a partir da posição índice do array. No exemplo a seguir serão removidos 3 elementos a partir da 3ª posição do array w.

```
let w = [4, 2, 8, 5, 1, 9, 7];
w.splice(2,3); //remove os elementos 8, 5 e 1
console.log(w);
```

Método forEach

O objeto Array possui métodos para iterar sobre os elementos do objeto array.

O método forEach itera sobre os elemento dos array, isto é, ele chama a operação callback para cada elemento do array ((https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)). O método forEach não modifica o array original. No exemplo a seguir a função callback será invocada para cada elemento do array w.

```
let w = [4, 2, 8, 5];
//O método forEach recebe como argumento uma função anônima
w.forEach(
    //Essa função anônima pode receber até 3 parâmetros que serão
    //fornecidos pelo forEach.
    //Essa função será invocada para cada elemento do array w
    function (item, indice, array) {
        console.log(indice + ':' + item + ' ' + array);
    }
);

w.forEach(
    //aqui foi passada uma arrow function com 2 parâmetros
    (item, indice) => console.log(indice + ':' + item)
);

w.forEach(
    //aqui foi passada uma arrow function com 1 parâmetro
    item => console.log(item)
);
```

```
0:4 4,2,8,5
1:2 4,2,8,5
2:8 4,2,8,5
3:5 4,2,8,5
0:4
1:2
2:8
3:5
4
2
8
5
```

Método map

O método map invoca a função callback, passada como argumento para cada elemento do array, e devolve um novo array como resultado (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/map). O método map não modifica o array original. No exemplo a seguir ele foi usado para duplicar cada elemento do array.

```
let w = [4, 2, 8, 5];
let r = w.map( function(item){
    return item * 2;
});
console.log(r); //resultado [ 8, 4, 16, 10 ]

let q = w.map( Math.sqrt ); //recebe uma função pronta do JavaScript
console.log(q); //resultado [ 2, 1.41, 2.82, 2.23 ]
```

Método reduce

O método reduce executa a função callback para cada elemento do array e retorna um único valor. O método reduce não modifica o array original. Nos exemplos a seguir será calculado o somatório.

```
let w = [4, 2, 8, 5];
let r = w.reduce( function(soma, item){
    return soma + item;
});
console.log(r); //retorna 19

// usando arrow function
let s = w.reduce( (soma, item) => soma + item );
console.log(s); //retorna 19
```

Os métodos forEach, map e reduce são comumente usados para operar sobre arrays, mas observe que eles são usados em situações distintas:

- forEach não possui retorno, pois opera sobre cada elemento do array atual;
- map retorna um novo array, onde cada elemento do array atual sofre a operação. Ele não altera o array atual;
- reduce retorna um único valor. Ele não altera o array atual.

Aqui foram apresentados alguns métodos do objeto Array, para mais detalhes acesse https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array.

viii. Objetos JavaScript

Em JavaScript, quase tudo é objeto. Desde as funcionalidades padrão, como strings e arrays, até as APIs para navegadores baseadas na linguagem. Para criar um objeto basta usar um par de chaves, no exemplo a seguir estamos criando um objeto sem membros e, na sequência, estamos adicionando uma propriedade/atributo e um método.

```
let obj = {}; //cria um objeto sem propriedades e métodos
console.log(obj); //resultado é {}

obj.nome = 'Ana'; //adiciona a propriedade nome no objeto
console.log(obj); //resultado é { nome: 'Ana' }

//adiciona o método print no objeto
obj.print = function(){
    console.log( this.nome ); //é obrigatório usar this para acessar os membros do objeto
};
console.log(obj); //resultado é { nome: 'Ana', print: [Function] }

obj.print(); //resultado é Ana
```

O objeto pode ser criado diretamente com as propriedades e métodos, isto é chamado de objeto literal:

```
let cad = {
    nome: 'Maria',
    print: function(){
        //resultado é { nome: 'Ana', print: [Function] }, pois this referência o próprio objeto
        console.log(this);
        console.log(this.nome);
    }
}
cad.print(); //resultado é Maria
```

As propriedades de um objeto podem ser acessadas usando a notação ponto ou colchetes:

```
console.log( cad.nome );
console.log( cad['nome'] );
```

É semelhante a maneira como acessamos itens de um array, e é basicamente o mesmo princípio, só que ao invés de usarmos um número de índice para selecionar um item, usamos o nome associado ao valor. Por esse motivo, objetos às vezes são chamados de arrays associativos - eles mapeiam strings a valores do mesmo modo que arrays mapeiam números a valores.

Se usarmos arrow function para criar um método, então não poderemos usar o this para referenciar o próprio objeto, pois arrow function não faz vinculação (bind) para this, isto é, ele não reconhece o próprio objeto.

```
let cad = {
    nome: 'Maria',
    print: () => {
        //arrow function não vincula (bind) this, então this não está vinculado ao próprio objeto
        console.log(this); //resultado é {}
        console.log(this.nome); //this é o objeto vazio {}, então não existe a propriedade nome
    }
}
cad.print(); //resultado é undefined
```

ix. Classes

As classes são usadas para criarmos objetos instanciados. No item anterior criamos objetos literais, isto é, escrevemos o conteúdo do objeto conforme o criamos. Os objetos literais são usados para representar apenas 1 objeto, já as classes funcionam como templates para a criação de vários objetos. Em JavaScript existem algumas formas de definir tipos de dados objetos, aqui serão apresentadas duas delas.

1 – Função de construtor: no exemplo a seguir a função Cliente é usada para definir um tipo de dado objeto e criar instâncias de objeto.

```
function Cliente(nome) {
    this.nome = nome;
    this.print = function () {
        console.log(nome); //acessa a propriedade sem o this
        console.log(this.nome);
    };
}

let a = new Cliente('Ana');
let b = new Cliente('Maria');
a.print(); //resultado é Ana
b.print(); //resultado é Maria
```

2 – Notação de classe: na ES6 foi introduzida a sintaxe de classe, semelhante a linguagem Java. No JavaScript o construtor possui o nome de constructor.

```
class Cliente{
    constructor(nome){
        this.nome = nome;
    }

    print() {
        console.log(this.nome);
    }
}

let a = new Cliente('Ana');
let b = new Cliente('Maria');
```

```
a.print(); //resultado é Ana
b.print(); //resultado é Maria
```

O método pode ser definido usando função anônima ou arrow function. Na arrow function o this não está vinculado ao código onde está a função, isto é, ele não está vinculado a classe. Ele está vinculado (bind) a quem chamou a função (método), no exemplo, a.print(), o this será o objeto a, pois foi o objeto a que chamou o método print.

```
class Cliente{
    constructor(nome){
        this.nome = nome;
    }

    print = () => {
        console.log(this.nome);
    }
}

let a = new Cliente('Ana');
let b = new Cliente('Maria');
a.print(); //resultado é Ana
b.print(); //resultado é Maria
```

Herança

Para criar uma subclasse, usamos a palavra reservada extends para informar ao JavaScript a classe na qual queremos basear nossa classe.

```
class Especial extends Cliente {
    constructor(nome){
        super(nome); //super refere-se ao construtor da super classe
    }

    categoria(){
        console.log('Especial');
    }
}
let c = new Especial('Clara');
c.print(); //resultado é Clara
c.categoria(); //resultado é Especial
```

Getters e Setters

Um getter retorna o valor atual da propriedade (leitura) e setter altera o valor da propriedade (escrita).

Os métodos getters e setters veem precedidos com a palavra reservada get e set, respectivamente. No exemplo a seguir esses métodos não podem ter o nome de nome, pois daria conflito com a propriedade nome. Além disso, os métodos get e set precisam ter o mesmo nome, pois na prática eles são acessados como variáveis, por exemplo, `a._nome = 'Mariane'`.

A vantagem de usar getters e setters é que podemos acessar uma propriedade encapsulada como se fosse variável.

```
class Cliente{
    constructor(nome){
```

```
        this.nome = nome;
    }

    get _nome(){
        return this.nome;
    }

    set _nome(nome){
        this.nome = nome;
    }

    print = function() {
        console.log(this.nome);
    }
}

let a = new Cliente('Maria');
a._nome = 'Mariane'; //escrita
```

x. JSON (JavaScript Object Notation)

É um formato de dados baseado em texto seguindo a sintaxe de objeto JavaScript. É comumente usado para transmitir dados em aplicações Web, do cliente para o servidor e vice-versa. Ele é uma alternativa a linguagem de marcação XML para o intercâmbio de dados na Web.

O JavaScript fornece um objeto JSON global que possui métodos disponíveis para conversão de texto para objeto nativo e vice-versa:

- parse: recebe como parâmetro uma string JSON e retorna o objeto JavaScript correspondente (deserialization);
- stringify: recebe como parâmetro um objeto nativo e retorna uma string JSON (stringfication).

A seguir tem-se um exemplo:

```
class Cliente{
    constructor(nome, idade){
        this.nome = nome;
        this.idade = idade;
    }

    print(){
        console.log(this.nome + ' ' + this.idade);
    }
}

let c = new Cliente('Ana',21);
//stringification do objeto nativo
let d = JSON.stringify(c); //converte de objeto para string no formato JSON
console.log(typeof c); //objeto
console.log(typeof d); //string
//veja que os métodos não são convertidos em string
console.log(d); // o resultado é {"nome":"Ana","idade":21}
```

```
//deserialization
console.log(JSON.parse(d)); //o resultado é { nome: 'Ana', idade: 21 }
```

Observações:

- JSON requer aspas duplas para serem usadas em torno de strings e nomes de propriedades. Aspas simples não são válidas:

```
let a = '{"nome":"Mara","idade":21}';
console.log(JSON.parse(a)); //correto
let b = '{nome:"Mara","idade":21}'; //erro a propriedade nome está sem aspas
let c = "{'nome':'Mara','idade':21}"; //erro a propriedade e valores envolvidos por aspas simples
```

- O JSON é puramente um formato de dados — contém apenas propriedades, sem métodos, veja que no exemplo anterior o método print não foi serializado.

xi. Strict mode

Strict mode transforma em erros alguns equívocos anteriormente aceitos. O JavaScript foi projetado para ser fácil para novos desenvolvedores, e algumas vezes ele dá semânticas de não-erros a operações que deveriam ser erros. Algumas vezes isso resolve o problema pontual, mas outras vezes cria problemas piores no futuro. Strict mode trata esses equívocos como erros para que sejam descobertos e consertados prontamente.

Para invocar strict mode para todo o script, coloque a declaração "use strict"; ou 'use strict'; antes de qualquer outra declaração.

O strict mode impossibilita criar variáveis globais acidentalmente. Em JavaScript cometer um erro de digitação ao digitar uma variável em uma atribuição cria uma propriedade no objeto global e continua a "*funcionar*" (embora falhas futuras sejam possíveis: provavelmente, em JavaScript moderno). Atribuições que acidentalmente criariam variáveis globais lançam exceções no strict mode, no exemplo a seguir se você comentar a instrução `'use strict'` o código não gera erro.

```
'use strict'; //sintaxe strict mode para todo o script
try{
    x = 4; //gera erro, pois toda variável precisa ser declarada usando let, var ou const
    console.log(x);
}
catch(e){
    console.log(e.message); //exceção: x is not defined
}
```

Para mais detalhes da cobertura do strict mode acesse (https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Strict_mode).

xii. Promise

Uma Promise (compromisso) é um objeto que representa a eventual conclusão ou falha de uma operação assíncrona. Essencialmente, uma promise é um objeto retornado para o qual adicionamos callbacks, em vez de passar callbacks para uma função. No exemplo a seguir, then, catch e finally são funções callback adicionadas a chamada da promise prom.

Os argumentos then, catch e finally são funções callback executadas de forma assíncrona. No exemplo a seguir tem-se o encadeamento de dois then, o 2º then está vinculado ao resultado do 1º then. Podemos encadear vários then.

Observe que ao rejeitar, nenhuma função then será executada.

```
//criação da promise
let prom = new Promise( (resolve, reject) => {
    //o código no construtor da promise é síncrono
    console.log('início do construtor');
    if (Math.random() > 0.5) {
        //irá invocar o 1o then
        resolve('Msg de sucesso');
    } else {
        //irá invocar o catch
        reject('Msg de erro');
    }
    console.log('fim do construtor');
});

//uso da promise
prom.then(
        //result possui o valor passado pela função resolve
        result => {
            console.log('1o then: ' + result);
            return 'resultado do 1o then';
        }
    )
    .then(
  //esse then somente é invocado se o return do 1o then for executado
        result => console.log('2o then: ' + result)
    )
    .catch( //err possui o valor passado pela função reject
        err => console.log('rejeitada: ' + err)
    ).finally( //executado em caso de sucesso ou falha
        () => console.log('finalizada')
    );

console.log('fim do código');
```

Resultado quando > 0.5

```
início do construtor
fim do construtor
fim do código
1o then: Msg de sucesso
2o then: resultado do 1o then
finalizada
```

Resultado quando <= 0.5

```
início do construtor
fim do construtor
fim do código
rejeitada: Msg de erro
finalizada
```

c

Para passarmos parâmetros para uma promise precisamos encapsular (wrapper) ela numa função e passar os parâmetros para a função. No exemplo a seguir os parâmetros x e y podem ser acessados dentro da promise.

```
function calculo(x, y) {
    return new Promise((resolve, reject) => {
        if (typeof x === 'number' && typeof y === 'number')
            resolve(x + y);
        else
            reject('Não são números');
    });
```

```
}


//uso da promise - resultado é 5
calculo(3, 2).then(
        result => console.log('Resultado: ' + result)
    ).catch(
        err => console.log('rejeitada: ' + err)
    );

//uso da promise - rejeitada
calculo('o', 'i').then(
        result => console.log('Resultado: ' + result)
    ).catch(
        err => console.log('rejeitada: ' + err)
    );
```

Uma promise pode ter três estados:

- Pendente: este é o estado inicial da promise, ela está aguardando para ser resolvida ou rejeitada. No exemplo anterior ocorre quando ele está gerando o número aleatório. Na prática, poderia ser quando estamos acessando a Web com uma solicitação AJAX. A promise ficará pendente até que a solicitação seja retornada;
- Cumprida: Quando a operação é concluída com êxito. No exemplo irá ocorrer quando o valor for > 0.5, isso irá disparar o 1º then;
- Rejeitada: Quando a operação falha. No exemplo irá ocorrer quando o valor for <= 0.5, isso irá disparar o catch.

Para mais detalhes acesse https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Usando_promises.

xiii. Async e await

Async function (função assíncrona) retorna por padrão uma promise. No exemplo a seguir a chamada teste(4) irá retornar uma promise.

Para obter o resultado/falha de uma promise temos de usar then ou catch.

```
async function teste(val){
    if( val !== undefined)
        return val;
    else
        throw new Error("Não definido");
}

let a = teste(4);
//o método then será invocado pelo resolve da promise
a.then(
    result => console.log('Then: ' + result)
)
.catch(
    result => console.log('Catch: ' + result.message)
);
```

Resultado do código ao lado:

```
Promise { 4 }
Then: 4
Catch: Não definido
```

```
console.log(a); //resultado é Promise { 4 }


let b = teste();
b.then(
    result => console.log('Then: ' + result)
)
.catch( //o método catch será invocado pelo reject da promise
    result => console.log('Catch: ' + result.message)
);
```

Podemos usar await somente no corpo de funções que são async. O operador await é utilizado para esperar por uma Promise, isto é, o operador await pausa a execução da função assíncrona e espera pela resolução da Promise. No exemplo a seguir observe que a 2ª chamada da função exec deveria terminar antes da 1ª, porém a instrução await fará com que a 2ª chamada só irá ocorrer após a 1ª ser concluída.

```
let a = await exec(10, 300); //espera 300 milissegundos
let b = await exec(20, 100); //só irá começar após terminar o anterior
```

No exemplo a seguir o resultado 30 só estará disponível após as duas chamadas da função exec serem concluídas.

A função calc retorna uma Promise, por esse motivo precisamos fazer a.then().

```
function exec(val, tempo) {
    return new Promise(resolve => {
        setTimeout(
            () => {
                console.log('terminou: '+ val);
                resolve(val) //irá chamar o then
            },
            tempo
        )
    });
}

//função async retorna uma Promise
async function calc() {
    let a = await exec(10, 300); //espera 300 milissegundos
    let b = await exec(20, 100); //só irá começar após terminar o anterior
    return a + b;
}

let a = calc();
//o then será a função resolve da Promise calc
a.then(result => console.log(result) );

//resultado é Promise { <pending> }, pois a promise está no estado pendente
console.log(a);
```

Resultado do código ao lado:

```
Promise { <pending> }
terminou: 10
terminou: 20
30
```

Para mais detalhes https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function e https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/await.

xiv. Require e exports

A função require é usada para importar recursos exportados de outros módulos (módulo é um arquivo JS que foi exportado). No Node cada módulo possui o seu próprio escopo, desta forma um módulo só pode acessar os recursos de outro módulo que foram expostos usando module.exports ou exports.

A seguir tem-se três formas de exportar os membros (variáveis, funções e classes) de um módulo. No 1º exemplo usou-se module.exports, e nos demais usou-se apenas exports, porém poderia ter sido usado também module.exports.

```
let x = 2;

function calc(){
  console.log(x*2);
}

class Cliente{
  constructor(nome){
    this.nome = nome;
  }

  print(){
    console.log(this.nome);
  }
}

//exports precisa estar
//no final do arquivo
module.exports = {
  x,
  calc,
  Cliente
}
```

```
let x = 2;

function calc(){
  console.log(x*2);
}

class Cliente{
  constructor(nome){
    this.nome = nome;
  }

  print(){
    console.log(this.nome);
  }
}

exports.x = x;
exports.calc = calc;
exports.Cliente = Cliente;
```

```
exports.x = 2;

exports.calc = function(){
    console.log(22);
}

exports.Cliente = class Cliente{
  constructor(nome){
    this.nome = nome;
  }

  print(){
    console.log(this.nome);
  }
}
```

Como exemplo crie os arquivos ex1.js e ex2.js dentro da mesma pasta e copie um dos códigos anteriores para o arquivo ex2.js. Para fazer a importação do módulo ex2 no arquivo ex1.js, usando a função require, precisamos fornecer o caminho+nome do módulo a ser importado. A extensão js é opcional, as duas instruções a seguir importam o mesmo módulo:

```
let ex = require('./ex2');
let ex = require('./ex2.js');
```

A seguir tem-se o código do módulo ex1. A função require carrega na variável ex aquilo que foi exportado do módulo ex2.

```
let ex = require('./ex2');

let c = new ex.Cliente('Mara');
```

```
c.print();
console.log(ex.x);
ex.calc();
```

xv. Import e export

Embora as instruções import e export façam parte do ES6, infelizmente elas ainda não são suportadas no Node por padrão. O Node segue a CommonJS, uma especificação de ecossistemas para o JavaScript. Especificações de transpiladores, tais como Traceur Compiler, Babel ou Rollup podem ser usados para converter códigos mais modernos para versões mais antigas. O React usa Babel, então lá usaremos import e export. Por enquanto, para testarmos essas instruções usaremos uma solução experimental, que consiste em salvar os nossos arquivos JS com a extensão .mjs (Module JS) e usar a flag de módulo experimental ao executar:

```
node --experimental-modules ex1.mjs
```

Há dois diferentes tipos de export: explícita (nomeada) e padrão (export default).

Exportação explícita: os membros exportados ficam entre chaves na exportação e importação.

```
//arquivo ex1.msj
import {calc, x, Cliente} from './ex2.mjs';

console.log(x);
calc();
let c = new Cliente('Ana');
c.print();
```

```
//arquivo ex2.msj
const x = 2;

function calc(){
    console.log(x * 2);
}

class Cliente{
    constructor(nome){
      this.nome = nome;
    }

    print(){
      console.log(this.nome);
    }
}

export {x, calc, Cliente};
```

Exportação padrão: só pode ter uma exportação padrão por módulo, as demais precisam ser explícitas. No exemplo a seguir apenas a variável x foi exportada por padrão. A exportação e importação por padrão não envolve o uso de chaves, e observe que elas precisam ser feitas em instruções distintas.

```
//arquivo ex1.msj
import {calc, Cliente} from './ex2.mjs';
import x from './ex2.mjs';

console.log(x);
calc();
let c = new Cliente('Ana');
```

```
//arquivo ex2.msj
const x = 2;

function calc() {
    console.log(x * 2);
}
```

```
c.print();
```

```
class Cliente {
    constructor(nome) {
        this.nome = nome;
    }

    print() {
        console.log(this.nome);
    }
}

export default x;
export { calc, Cliente };
```

No exemplo a seguir todos foram exportados explicitamente. Veja que a palavra reservada export foi adicionada antes de cada membro exportado.

```
//arquivo ex1.msj
import {x, calc, Cliente} from './ex2.mjs';

console.log(x);
calc();
let c = new Cliente('Ana');
c.print();
```

```
//arquivo ex2.msj
export const x = 2;

export function calc() {
    console.log(x * 2);
}

export class Cliente {
    constructor(nome) {
        this.nome = nome;
    }

    print() {
        console.log(this.nome);
    }
}
```

Podemos renomear ao exportar. No exemplo a seguir a função calc foi exportada com o nome de calcular, desta forma, ela precisará ser importada com o nome de calcular.

```
export { x, calc as calcular, Cliente };
```

Podemos renomear também ao importar. No exemplo a seguir a variável x foi renomeada para valor na importação.

```
import {x as valor, calc, Cliente} from './ex2.mjs';
console.log(valor);
```

Podemos definir um *namespace* na importação, veja que todos os membros importados precisam usar o namespace ex

```
import * as ex from './ex2.mjs';

console.log(ex.x);
ex.calc();
```

```
let c = new ex.Cliente('Ana');
c.print();
```

Para mais detalhes acesse https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/export e https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/import.