



Sumário

Sumário	1
1. Introdução	2
1.0 console.log	2
1.1 html JavaScript	3
1.2 Variáveis let e const	4
1.3 Variáveis let e const - Parte 2	5
1.4 Navegador - window	6
1.5 Strings	7
1.6 Numbers	9
1.7 Objeto Math	10
1.8 Array	11
1.9 Funções - parte 1	14
1.10 Funções - parte 2	17
1.11 Objetos	21
2. Lógica de Programação	23
2.0 If...Else	23
2.1 Operador Ternário	24
2.2 Date	25
2.3 Switch/Case	26
2.4 Atribuição via destruturação	27
2.5 Estruturas de Repetição	27
2.6 Document Object Model (DOM)	30
Tipos de dados (interface)	31
Document	31
Node	35
Element	39
Elementos e funcionalidades	45
2.7 Try ... Catch	46
2.8 SetInterval e SetTimeout	48
3. Funções Avançadas	50
3.0 Função de Callback	50
3.1 Função imediata	51

3.2 Função fábrica (Factory functions)	51
3.3 Função Construtora	52
3.4 Função Recursiva	53
3.5 Função Geradora	54
4. Arrays Avançados	56
4.0 Filter	56
4.1 Map	58
4.2 Reduce	59
5. Objetos prototypes avançados	60
5.0 this	60
5.1 Protótipos de objetos	63
5.2 Herança	69
5.3 Polimorfismo	73
5.4 Objeto Map	75
6. Classes POO	77
6.0 Getter / Setter	83
6.0.0 Getter	83
6.0.1 Setter	85
7. JS Assíncrono	86
7.0 Promises	86
7.1 Async/Await	93
8. JS Tooling	95
8.0 Import	95
8.1 Export	97

1. Introdução

1.0 console.log

Mostra informações na tela.

exemplo: console.

```
console.log('hello world!');  
  
console.log("Diogo Dias");  
  
console.log(`'diogo' "mello"`)  
  
console.log(155, 16.86, "Diogo Dias")
```

Caso seja necessário colocar aspas simples dentro do `console.log()`, é importante que o texto ou qualquer tipo de dados seja colocado dentro de aspas duplas e vice versa.

1.1 html JavaScript

Dentro do HTML pode ser colocado um alerta utilizando o JS, de modo que ao abrir a página o alerta será emitido.

Exemplo:

```
// comando nao suportado pelo visual code
alert("testando alerta")
```

Esse tipo de alerta não é suportado pelo vs code.

Resultado do alerta:



Para isso é importante que o comando JS esteja linkado com uma documento html.

Exemplo:

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>New order</title>
</head>
<body>

  <!-- parte de javascript -->
  <!-- primeira forma -->
  <!-- <script>
    console.log("ola mundo")
    console.log("exibido no trecho do navegador")
  </script> -->

  <!-- segunda forma (mais usada) -->
  <script src="js/p3_html_javascript.js"></script>
</body>
</html>
```

1.2 Variáveis let e const

No JS as variáveis são escritas de forma maiúsculas e minúsculas sem ser consideradas diferentes.

JS possui três tipos de variáveis básicas, sendo que uma já não está mais em uso, são elas: **var** (obs: está obsoleta), **let** e **const**.

- **Var** - Declara uma variável, opcionalmente, inicializando-a com um valor.

exemplo:

```
var nome = "Joao";
```

- **Let** - Declara uma variável local de escopo do bloco, opcionalmente, inicializando-a com um valor.

exemplo:

```
let nome = "Joao";
```

```
let idade;  
console.log(idade);  
idade = 19;  
console.log(idade);
```

1º idade - undefined. (nota: variáveis declaradas com var ou let, se não for colocado um valor inicial, elas irão retornar undefined)

2º idade - 19.

- **Const** - Declare uma constante de escopo de bloco, apenas de leitura.

formato:

```
const nome2 = "Tatiana";  
console.log(nome2);
```

1º nome2 - Tatiana (obs: o valor de variável nome2 não pode ser mudado ao longo do código, caso contrário irá gerar um erro de código).

Formas de dar nomes às variáveis: **nome**, **nome_aluno**, **nomeAluno**, **nome1**, **_nome**, **NOME**, **NoMe**. obs: os nomes de variáveis não podem ser inicializados com números.

OBS: O escopo de bloco se refere ao meio onde a variável está inserida, por exemplo, caso a variável seja criada dentro de uma função, ela não poderá ser chamada dentro do escopo global, já o inverso irá funcionar.

INFORMAÇÃO EXTRA:

- **typeof**: retorna o tipo de dado.

Exemplo: `console.log(typeof(resultado));` .

- **Undefined** se comporta como valor falso.

Os nomes de constantes não podem ter o mesmo nome das funções presentes no código pois isso irá gerar erros, assim como, uma **variável const** não pode ser redeclarada como **let** ou **var** posteriormente.

1.3 Variáveis let e const - Parte 2

Em JS as variáveis podem receber 6 tipos de dados, são eles: **Booleano** (true e false), **nulo**, **undefined**, **número** (int - 42 e float - 42.50), **cadeia de caracteres** ("Olá mundo"), **símbolos**, **objetos** (recipientes para valores) e **Funções** (métodos que suas aplicações podem executar).

não precisa ser especificado qual o tipo de dado de uma variável quando for declarada, pois o **JS é uma linguagem dinamicamente tipada**.

Exemplo: Redecaração de variável.

```
// pode ser redeclarada
var nome = "Diogo";
var nome = "Tatiana";
console.log(nome);
```

Nota: Ao combinarmos um valor numérico com um valor do tipo string, teremos o valor numérico transformado em string.

Tabela de caracteres especiais que podem ser colocados dentro de uma string em JS:

<code>\b</code>	Backspace
<code>\n</code>	Nova linha
<code>\t</code>	Tabulação

INFORMAÇÃO EXTRA:

- **parseInt** - retornar apenas valores numéricos inteiros (obs: restrito para a casa dos decimais).

exemplo: `const num2 = parseInt('5');`

- **parseFloat** - retornar apenas valores numéricos de ponto flutuante.

exemplo: `const num3 = parseFloat('5.3');`

1.4 Navegador - window

O objeto window representa uma **janela que contém um elemento DOM**.

funcionalidades de window:

- **window.console** (**debug**, **error**, **info**, **log**, **warn**) - Retorna uma referência para o objeto console fornecendo acesso ao console debugging do navegador.
- **window.crypto** (SubtleCrypto) - Retorna o objeto de criptografia do navegador.
- **window.defaultStatus** - Obtém/define o texto da barra de status para determinada janela.
- **window.document** - Retorna a referência à propriedade document que a janela contém.

1.5 Strings

Dentro de JS as strings são consideradas uma cadeia de caracteres.

exemplo:

```
let umaString = "um texto";
```

- **se comportam da mesma forma que um array** tendo as palavras que a compõem acessadas por referência.

exemplo: acessando string pelo índice.

```
console.log(umaString[4], umaString[5]);
```

onde será exibido “e” e “x” que estão presentes dentro da palavra “texto”.

Os valores dentro da string podem ser acessados utilizando **charAt**.

exemplo:

```
console.log(umaString.charAt(1));
```

Uma forma que há de **concatenar diferentes strings criadas seria usar a função concat** que se encontra dentro do próprio JS.

exemplo:

```
console.log(umaString.concat(' ', 'em um lindo dia'));
```

Nota: podemos utilizamos “+” para concatenar strings.

As strings também podem ser comparadas da mesma forma que se compara valores numéricos utilizando “<”/”>”.

Strings podem ser tratadas como **strings primitivas e objetos strings**.

```
let umaString = "um texto"; // primitiva  
let outraString = new String(umaString); //objeto string
```

INFORMAÇÕES EXTRAS:

- **indexOf** - retorna o índice da primeira ocorrência do valor fornecido.

Exemplo:

```
// mostrar o índice de início
console.log('testando o index: ', umaString.indexOf('texto'));
```

- **lastIndexOf** - retorna o índice da primeira ocorrência do valor fornecido, só que de forma inversa ao indexOf, ou seja, começa do final.
- **eval()** - computa um código JS representado como uma string.

Exemplo:

```
contaString = eval(contaString);
console.log(contaString);
```

- **substring()** - retorna a parte da string entre os índices inicial e final, ou até o final da string.

Exemplo:

```
console.log('utilizando o subString: ', umaString.substring(umaString.length -5, umaString.length -1));
```

- **length** - retorna o tamanho de uma string.
- **replace()** - troca uma palavra pela outra dentro da string.

Exemplo:

```
console.log('utilizando o replace: ', umaString.replace('um', 'outro'));
```

- **slice()** - pega partes específicas dentro de uma string, colocando um valor de início e fim.

Exemplo:

```
console.log('slice: |', umaString.slice(3, 8));
```

- **split()** - separa todos os valores da string dentro um array.

Exemplo:

```
console.log(umaString.split(' '));
```

No exemplo os valores estão sendo separados a cada espaço encontrado dentro da frase.

1.6 Numbers

Os números em JS podem ser classificados principalmente entre valores **inteiros (int - 7)** e **valores com ponto flutuante (float - 7.5)**. Logo após, valores de ponto de flutuante podem apresentar muitos números após a vírgula, uma forma de determinar essa quantidade seria usar **toFixed()**.

Exemplo:

```
console.log(num1.toFixed(3));
```

Isso significa que os valores após a vírgula só podem ter no máximo 3 casas.

Caso se deseje transformar pontos flutuantes em valores reais, uma ideia seria a utilização de `Number()`.

Exemplo:

```
num1 = Number(num1.toFixed(2));  
console.log('valor de num1: ', num1);
```

Valor exibido será “1”, isto é, ocorre pelo fato da variável conter o valor “0.7” que quando passado para valor real o JS considera que ele está mais próximo de 1.

OBS: Em alguns casos pode ocorrer do objeto em questão não poder ser transformado para um número, por isso o JS irá retornar à NaN.

A duas forma de utilizar o `Number()`, a primeira seria utilizando `new Number()` como se fosse um construtor e a segunda seria colocando apenas `Number()`, que nesse iria servir como se fosse um tipo de conversão de tipo.

INFORMAÇÕES EXTRAS:

- `Number.NaN`: valor especial que não é número.

exemplo: `console.log(Number.NaN);`, tem como retorno NaN.

- `Number.isNaN()`: determina se o valor passado é NaN.

exemplo: `temp = num3 * num2;
console.log(Number.isNaN(temp));`

retorna falso (false), pois os valores que estão presentes dentro das variáveis passadas para “temp” são números.

- `Number.isInteger()`: determina se o valor que foi passado é inteiro.
- `Number.parseFloat()`: converte a string recebida como argumento e a retorna como um número de ponto flutuante.
- `Number.parseInt()`: converte a string recebida como argumento e a retorna como um número inteiro.

1.7 Objeto Math

Math é um objeto embutido que tem **propriedades e métodos para constantes e funções matemáticas**, não é um objeto de função. Além disso, **todas as propriedades de Math são estáticas**, diferente dos construtores que também apresentam propriedades não estáticas.

Alguns exemplos em relação a utilização de Math:

- ```
pi = Math.PI;
console.log(pi);
```

Neste caso a variável criada “pi” irá receber o valor de real por meio da utilização do método Math.PI, gerando o valor de 3.14.

- ```
x = 90;  
const sen = (valor) => {  
    return Math.sin(valor);  
};  
  
console.log(sen(x));
```

Nesta situação duas variáveis são criadas, uma para receber um valor numérico qualquer e a outra para receber o método sin() oriundo de Math.

Um ponto importante de salientar, é que a variável “sen” também pode ser considerada uma **Arrow Function**.

- Após isso, o valor retornado em sen(x) será em radianos, portanto, 0.894.
 - Uma forma de converter esse valor em radiano para graus, seria por meio da divisão por 180, caso contrário basta multiplicar.

INFORMAÇÕES EXTRAS:

- **Math.floor()**: irá arredondar o valor numérico para baixo.
 - ```
let num2 = Math.floor(num1);
```
- **Math.ceil()**: irá arredondar o valor numérico para cima.
  - ```
let num3 = Math.ceil(num1);
```
- **Math.max()**: retorna o maior valor dentro dos parâmetros.
 - ```
let num4 = Math.max(1, 2, 4, 8, 16, 32, -16, -8, -4, -2, -1);
```

- **Math.min()**: retorna o menor valor dentro dos parâmetros.
  - `let num5 = Math.min(1, 2, 4, 8, 16, 32, -16, -8, -4, -2, -1);`
- **Math.pow(base, expoente)**: retorna a base elevada ao expoente.
  - `console.log(Math.pow(2, 2));`
- **Math.random()**: retorna um número pseudo aleatório entre 0 e 1.
  - `console.log(Math.random() * (10 - 5) + 5);`, neste caso o número aleatório ficará entre 10 e 5.
- **Math.round()**: retorna o valor arredondado de x, para o valor inteiro mais próximo.
  - `console.log(Math.round(0.7));`
- **Math.sign()**: retorna o tipo de sinal de x, se é negativo, positivo ou zero.
  - `console.log(Math.sign(x));`
- **Math.sqrt()**: retorna a raiz quadrada.

## 1.8 Array

O objeto Array do JavaScript é um objeto global usado na construção de 'arrays':  
**objetos de alto nível semelhantes a listas.**

### Conceitos:

- Nem o tamanho de um array JavaScript nem os tipos de elementos são fixos.
- As propriedades JavaScript que começam com um dígito não podem ser referenciadas com notação de ponto.

### exemplo:

- ```
const nome = ['Maria', 'Diogo', 'Tatiana', 'Cristian', 'Joao Lucas', 'Valeria'];
console.log(nome);
```

Outras sintaxe:

- **new Array** (elemento1, elemento2, ..., elementoN)

O array pode guardar uma grande quantidade de informações dependendo de quem o criou.

Uma forma de saber a **quantidade de elementos dentro do array seria por meio da utilização de “length”**, onde será retornado o tamanho do array.

Exemplo:

```
console.log(nome.length);
```

Cada item dentro do array pode ser acessado de forma individual, sem necessariamente passar por todos os itens presentes, basta utilizar o **índice de cada elemento**.

- **A contagem dos índices do array começam de 0**. Ou seja, considerando que temos um array de 12 elementos, sua contagem irá de 0 à 11, sendo 0 o seu primeiro elemento.

O array apresenta **formas de iterar sobre os seus itens** com a utilização de **forEach()**. Esse método tem por objetivo executar uma função em cada elemento de um array.

Exemplo:

```
// iterando dentro do array
const estados = ['Pará', 'Maranhã', 'Piauí', 'Ceara'];
estados.forEach(function (itemEstado, indice, array){
    console.log(`itens array: ${itemEstado}, \tíndices dos itens: ${indice}, \tarray: ${array}\n`);
});
```

é possível notar na figura que **forEach()** itera somente sobre os valores, índice e o próprio array.

indexOf() - procura um determinado índice dentro do array.

exemplo:

```
const localPara = estados.indexOf('Pará');
console.log(`índice onde se encontra o Pará: ${localPara}`);
```

splice() - remove um item utilizando seu índice como vetor que recebe apenas um valor informando em que índice começar e outro dizendo quantos valores eliminar a partir do início dado.

Exemplo:

```
let inicio = 2;
let qtdRemovido = 1;
const estadosRemovidos = estados.splice(inicio, qtdRemovido);
console.log(estadosRemovidos);
```

slice() - Copia grandes quantidades de valores de um array para outro array. basicamente irá fatiar o array entre seus valores, podendo copiar valores com um início e fim determinado.

Exemplo:

```
let valorInicio = 0;
let valorFim = 4;
const outrosNomes = nome.slice(valorInicio, valorFim);
console.log(outrosNomes);
```

Métodos interessantes:

- **Array.of()**: Cria uma nova instância Array com um número variável de argumentos, independentemente do número ou tipo dos argumentos.

Exemplo:

```
const ar = Array.of(1, 3, 4, 5);
console.log(ar);
```

o valor retornado será [1, 2, 3, 4, 5]. Dessa forma, é possível notar que não necessariamente é um método importante, pois pode facilmente ser substituído pelo método normal ou pelo construtor de array.

INFORMAÇÕES EXTRAS:

- **keys**: uma forma de acessar somente as chaves do array.

```
console.log(Object.keys(estados));
```

- **push()**: adiciona no final, `nome.push('Dulcila');`
- **unshift()**: adiciona no início, `nome.unshift('Eduardo');`
- **pop()**: remove do final, `nome.pop();`
- **shift()**: remove do início `const primeiroElemento = nome.shift();`

1.9 Funções - parte 1

Uma função é um procedimento de JavaScript - um conjunto de instruções que executa uma tarefa ou calcula um valor.

Exemplo:

```
function Saudacao(nome) {  
  // console.log(`boa tarde ${nome}!`);  
  return `BOM DIA ${nome}!`;  
}
```

(função chamada)

```
const raiz = function (n) {  
  return Math.sqrt(n);  
};
```

(função anônima - melhor).

De modo geral, **função é um "subprograma" que pode ser chamado por código externo (ou interno no caso de recursão) à função.**

Em JavaScript funções são **objetos de primeira classe**, pois elas podem ter propriedades e métodos como qualquer outro objeto.

- Os parâmetros de uma função são chamados de **argumentos da função**, argumentos são passados para a função por valor.
- **Se uma função muda o valor de um argumento, esta mudança não é refletida globalmente ou na chamada da função.**

Nota: referência de objetos são valores também, e eles são especiais: se a função muda as propriedades do objeto referenciado, estas mudanças são visíveis fora da função.

Outra forma de criar uma função é por meio das **arrow function**, que são nada mais que expressões com **sintaxe curtas e que conectam seu valor lexicalmente**.

Exemplo:

```
const marcas = () =>{  
  return ['adidas', 'nike', 'olimpicos'];  
};
```

Nessas funções a presença dos parênteses só se dá importante quando há mais de um argumento, caso contrário não há necessidade.

dois argumentos

```
const marcas = (marca1, marca2) => {  
  return marca1 + ' ' + marca2;  
};
```

um argumento

```
const marcas1 = marca1 => {  
  return marca1;  
};
```

new function - serve para criar novas funções, o uso do construtor Function para criar funções não é recomendado uma vez que é requerido o corpo da função como string, o que pode impedir algumas otimizações por parte do motor JS e pode também causar outros problemas.

Uma função pode receber alguns tipos de parâmetros:

- **undefine/default** (permitem que parâmetros nomeados sejam inicializados com valores padrão se nenhum valor ou undefined for passado)
- **rest** (permite que uma função aceite um número indefinido de argumentos como uma matriz).

Exemplo: parâmetros com valor padrão

```
function Soma(x, y) {  
  return x + y;  
}
```

Exemplo: parâmetros rest

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

Podemos definir métodos **getters** (métodos de acesso) e **setters** (métodos de alteração) dentro de objetos.

O método get - vincula uma propriedade de objeto a uma função que será chamada quando essa propriedade for pesquisada.

Exemplo:

```
const obj = {
  log: ['a', 'b', 'c', 'd'],
  get latest() {
    return this.log[this.log.length - 1];
  }
}

console.log(obj.latest);
```

O método set - vincula uma propriedade de objeto a uma função a ser chamada quando houver uma tentativa de definir essa propriedade.

formato:

```
const language = {
  set current(name){
    this.log.push(name);
  },
  log: []
};

language.current = 'en';
language.current = 'br';

console.log(language.log);
```

Podemos criar métodos mais curtos dentro de objetos da mesma forma que getters e setters

Exemplo:


```
const nomeCompleto = {  
  nome(){  
    return "Diogo";  
  },  
  sobrenome(){  
    return "Mello";  
  }  
}
```

INFORMAÇÕES EXTRAS:

- **arguments**: uma forma de acessar os argumentos de função. formato:

```
function valores(a, b, c){  
  console.log(arguments[0], arguments[1], arguments[2]);  
}
```

1.10 Funções - parte 2

Definição de função consiste na palavra chave function, seguida:

- Nome da função.
- Lista de argumentos para função entre parênteses e separados por vírgulas em caso onde a mais de um argumento.
- Lista de instruções colocadas entre chaves.

Normalmente, são passados parâmetros primitivos para função.

por exemplo: imaginando que temos uma function nome com parâmetros nome e sobrenome sendo argumentos da função, os valores a serem passados seriam apenas strings, depois os valores passados para função seriam alterados sem refletir globalmente no código ou na função chamada. Agora, supondo que seja passado um objeto para função e esse mesmo objeto for alterado pela função, esta alteração será refletida em todo o código.

Variáveis quando criadas dentro do escopo de uma função não podem ser chamadas em nenhum lugar fora da função, porque ela foi definida somente dentro do escopo da função.

funções podem acessar todas as variáveis que foram definidas fora dela. Então, supondo que a função seja criada dentro de um escopo global dentro do código, ela poderá acessar todas as variáveis que foram definidas dentro desse escopo.

Recursividade:

- Quando uma função chama a si mesma.
- **Executa o código várias vezes** por isso é importante que haja uma condição para parar o laço, caso contrário ficaríamos em laço infinito.

Exemplo:

```
function loop(n){  
  if (n > 10) return 1;  
  else return console.log(loop(n+1));  
}  
  
console.log(loop(1));
```

Podemos aninhar funções dentro de outras, onde a função que se encontra interna só poderá ser chamada pela função pai, ou seja, a função mais externa. Dessa forma, é constituído um **closure que é uma expressão que pode ter variáveis livres em conjunto com o ambiente que conecta essas variáveis**. Sendo uma função aninhada, ela pode herdar todos os argumentos e variáveis presentes na sua função de contenção (pai).

Exemplo:

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

Uma vez formado um closure, **podemos especificar argumentos para função externa e interna.**

exemplo:

```
function nome (a){
  function sobrenome (c){
    return a + c;
  }

  return sobrenome;
}
```

criamos um closure e logo em seguida determinamos tanto o nome quanto o sobrenome, `b = nome('Diogo')('Mello')`.

Funções podem formar múltiplos aninhamentos onde a função A contém uma função B, que contém uma função C, assim podemos dizer que A possui dois closures e que A pode ser acessada tanto por B quanto por C, isso é chamado de **encadeamento de escopo**.

Exemplo:

```
function A(){
  const a = 'Diogo';

  function B(){
    const b = 'Eu sou o';

    function C(){
      const c = 'Olá, ';
      return c;
    }

    return C() + b;
  }

  return B() + a;
}
```

Closures dentro do javascript, basicamente são funções (mais interna) que se encontram dentro de outra (mais externas) **fornecendo mais segurança aos argumentos, variáveis e funções que se encontram dentro do closure**, além disso, esses valores irão durar mais na memória do que a função externa, isso claro se a função interna durar mais tempo na memória.

Quando dentro de uma closure houver dois argumentos ou variáveis que possuem o mesmo nome, **o mais interno é que terá maior prioridade em relação aos**

outros, enquanto o mais externo ficará com a menor prioridade, isto é chamado de cadeia do escopo.

Podemos colocar valores padrão dentro dos parâmetros em javascript quando não for dado um valor, assim o valor do parâmetro não irá mais retornar undefined e sim o valor que foi dado na sua criação.

exemplo:

```
let pessoa = (nome="Diogo") => {  
  return nome;  
}
```

Além disso, podemos também utilizar **rest em casos que há um número indefinido de parâmetros na função**, normalmente esse tipo de argumento é colocado após argumentos normais ou padrões simbolizado com **"..."** e o seu nome.

exemplo:

```
const multiplicador = (multi, ...args) => {  
  return args.map(x => multi * x);  
}  
  
let conta = multiplicador(2, 1, 3, 4);  
console.log(conta);
```

Há casos em que podemos utilizar um tipo de função chamada de **"função seta ou arrow functions"** que apresenta uma sintaxe pequena em comparação com a expressão de uma função normal e **lexicalmente vincula o valor this, além de serem normalmente anônimas**.

Ademais, arrow functions **não podem ser usadas quando se deseja criar construtores ou métodos e usar a palavra chave this, assim como utilizar prototype, new function e yield**.

Esses tipos de funções podem apresentar dois **tipos de corpos**:

- **concise body** - o primeiro seria quando há uma expressão especificada com retorno implícito.

- **block body** - é necessário explicitamente usar uma expressão de retorno.

Podemos usar a palavra `this` dentro da função de duas formas.

Exemplo:

```
function Pessoa2(){
  this.idade = 0;

  setInterval(function crescer(){
    this.idade++;
  }, 200)
}
```

```
function Pessoa2(){
  self = this;
  self.idade = 0;

  setInterval(function crescer(){
    self.idade++;
  }, 200)
}
```

1.11 Objetos

Objetos podem ser considerados manipuladores de dados. Além disso, objetos são nada mais nada menos que uma coleção de dados e funcionalidades relacionadas, apresentando **variáveis (propriedades)** e **funções (métodos)** dentro do seu escopo.

A cada **chave (nome do membro)/valor (valor do membro)** que for sendo colocado dentro de um objeto é importante que eles sejam separados por vírgulas,

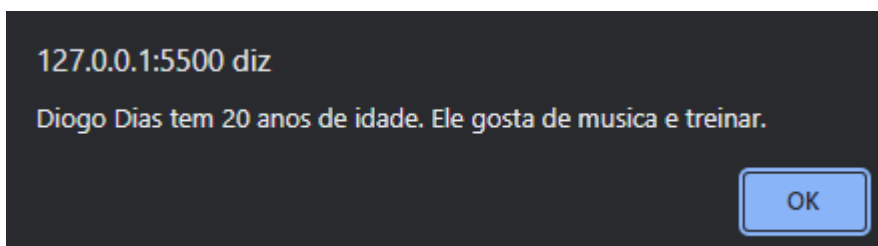
assim como chave e valor devem ser separados por “:”, **Exemplo:**

```
let pessoa = {  
  nome: ['Diogo', 'Dias'],  
  idade: 20,  
  sexo: 'masculino',  
  interesses: ['musica', 'treinar'],  
  bio: function(){  
    alert(this.nome[0] + ' ' + this.nome[1] + ' tem ' + this.idade  
    interesses[1] + '.');  
  },  
  saudacoes: function(){  
    alert('Oi! Eu sou ' + this.nome[0] + '.');  
  }  
};
```

Em seguida, é possível notar que podemos acrescentar qualquer tipo de valor de membro para o objeto literal como: **funções, arrays, strings, outros objetos, números e dentro outros tipos.**

Todos esses valores poderão ser acessados por meio da notação de pontos onde é importante que coloquemos primeiramente o nome do objeto que desejamos acessar seguido de um ponto + o nome da propriedade ou método encapsulado.

Exemplo: acessando o método “bio” do namespace pessoa



resposta do sistema.

De mesma forma, também podemos acessar os valores membros dentro do objeto por meio dos colchetes (**chamamos isso de arrays associativos**), ou seja, eles mapeiam strings a valores ao invés de números a valores como são feitos com arrays normais.

exemplo:

```
> pessoa['idade']  
< 20
```

A princípio, podemos setar (atualizar) os valores membros de um objeto simplesmente declarando a variável que desejamos setar.

exemplo:

```
> pessoa.nome['segundoNome'] = 'Mello';  
< 'Mello'  
  
> pessoa.nome['primeiroNome'] + ' ' + pessoa.nome['segundoNome'];  
< 'Diogo Mello'
```

Analogamente, é possível criar outros membros.

exemplo:

```
pessoa.terceiroNome = 'Mello'
```

na figura criamos um novo membro chamado “terceiroNome” a partir da notação de pontos e definimos seu valor membro como “Mello”.

A palavra `this` presente dentro do objeto `pessoa` tem como finalidade referenciar o objeto atual em que o código está escrito, ou seja, “`pessoa`”. Depois, utilizando o `this` é possível assegurar que valores corretos que estão sendo usados quando o contexto de um membro muda.

2. Lógica de Programação

2.0 If...Else

O condicional `if` tem como **funcionalidade executar códigos que se encontrem dentro do seu escopo quando a condição que foi imposta for verdadeira**, caso contrário o bloco `if` não irá ser executado e o **Else que entrará em execução**.

exemplo:

```
const hora = 13;

if (hora < 12) {
  console.log('Bom Dia Rapariga');
} else if (hora >= 12 && hora < 18) {
  console.log('Boa tarde filho da mae');
} else console.log('vai dormir baitola');
```

É possível notar que há 3 condicionais e somente o segundo condicional será executado porque a variável hora possui valor 13, se encaixando no segundo condicional).

Assim sendo, o formato básico de if/else é:

if(condicional){instrução}else{instrução},

- dentro das chaves será possível executar tanto instruções como outras condições.

Se houver necessidade há também uma forma de colocar uma outra condição if para acrescentar uma nova regra ao código.

exemplo:

```
if (hora < 12) {
  console.log('Bom Dia Rapariga');
} else if (hora >= 12 && hora < 18) {
  console.log('Boa tarde filho da mae');
```

notamos que a outro if após o primeiro condicional precedido de um else (aninhamento).

2.1 Operador Ternário

Este tipo de **operador é o único que possui 3 operando** em sua composição e pode ser considerado um atalho para instrução if.

exemplo:

```
// segundo modo condicional - condicao ? valor para verdadeiro : valor para falso
const nivelUsuario = pontuacaoUsuario >= 1000 ? 'usuario vip' : 'usuario normal';
```

a mensagem “usuário vip” somente será executado caso a condição

“pontuacaoUsuario” for >= a 1000, caso contrário “usuário normal” que será efetuado.

As **avaliações ternárias também pode ser atribuídas a variáveis** fazendo com que seu valor seja salvo dentro da mesma.

exemplo:

```
const idade = 20;  
  
console.log(idade >= 20 ? 'pode entrar' : 'não pode entrar');
```

2.2 Date

Este objeto é baseado no **valor de tempo que é em milissegundos** desde de 1º de janeiro de 1970 e representa um único momento de tempo.

Exemplo:

```
console.log(new Date());
```

Após, os parâmetros que estão presentes são: ano, mês (começa em 0), dia, hora, minuto, segundo e milissegundos.

forma de criação de um objeto Date:

- ```
const todoDia = new Date();
```
- ```
const data2 = new Date('2022-04-21 20:20:59');
```
- ```
const data = new Date(2019, 3, 20, 15, 14, 27);
```

sequência: ano, mês, dia, hora, minutos, segundos e milisegunds.

## 2.3 Switch/Case

A instrução case avalia uma expressão correspondente ao valor de uma expressão a uma série de cases (cláusulas) e executa instruções após a primeira case com um valor correspondente até que um break seja encontrado.

exemplo:

```
switch(diaSemana) {
 case 0:
 diaSemanaTexto = 'domingo';
 break;
 case 1:
 diaSemanaTexto = 'segunda';
 break;
 case 2:
 diaSemanaTexto = 'terca';
 break;
 case 3:
 diaSemanaTexto = 'quarta';
 break;
 case 4:
 diaSemanaTexto = 'quinta';
 break;
 case 5:
 diaSemanaTexto = 'sexta';
 break;
 case 6:
 diaSemanaTexto = 'sabado';
 break;
 default:
 diaSemanaTexto = '';
}
```

Após, caso nenhum dos cases sejam selecionados a cláusula default será executada.

**A execução desse tipo de instrução se dá pela avaliação de uma expressão**

que após analisada será comparado aos diversos cases que houver dentro do switch, se for igual então todas as instruções que se encontram na parte interna do case serão executadas.

Porém, se **nenhuma das cláusulas forem encontradas** o programa irá passar o controle para default onde por convenção se encontra na final do switch.

**Obs: caso não seja colocado nenhum break dentro da cláusula, o programa irá rodar outra cláusula mesmo que não esteja dentro dos critérios.**

## 2.4 Atribuição via desestruturação

**Expressão que possibilita a retirada de valores ou dados de arrays e objetos em variáveis distintas.**

exemplo:

```
const pessoa = {
 nome: 'Diogo',
 sobrenome: 'Dias',
 idade: 20,
 endereco: {
 rua: 'WE 84',
 numero: 881
 }
};

const { nome, ...sobras } = pessoa;

const { nome: n } = pessoa;
```

Dessa maneira, a possibilidade de definirmos quais elementos devem ser extraídos da variável de origem com elementos do lado esquerdo (variáveis que irão pegar os elementos).

A também como ignorar alguns dados que não são importantes no momento, para isso é importante que se utilize **'vírgula'** para sinalizar que está pulando um elemento dentro do array ou objeto.

exemplo:

```
const [a, , b] = hobbies;
```

## 2.5 Estruturas de Repetição

Os laços de repetição nada mais servem para a execução de determinada ação repetidas vezes.

Também, há diferentes modos de se usar o laço dependendo somente da necessidade do programador escolher o melhor modo para resolução de determinada situação.

**Tipos de mecanismos de repetição:**

- **for**
  - O laço será repetido até que a expressão seja falsa e em casos que a condição seja omitida, a mesma será considerada verdadeira.
  - formato: **for (expressaoInicial; condicao; incremento) {intrucao}**

### exemplo:

```
for (let i = 0; i < 6; i++) {
 console.log(`valor ${i}`);
}
```

```
for (let i = 0; i < frutas.length; i++) {
 console.log(`nome fruta: ${frutas[i]}`);
}
```

- Caso as instruções dentro do laço sejam grandes ocupando mais de uma linha, é necessário que seja colocado chaves “{ }”.

### • do while

- Repetirá até que a expressão seja falsa. Após, a instrução será executada apenas uma vez e depois ocorrerá a verificação da condição, se verdadeira o laço se repetirá novamente

○ **formato: do {instrucao} while(condicao).**

### exemplo:

```
do {
 // executando o código
 rand = random(min, max);
 console.log(rand);
 // chutando a condição
} while (rand !== 10)
```

### • while

- Esse tipo de instrução diferentemente da “do while” irá primeiramente verificar a condição e logo após executar a instrução. É importante destacar que se a condição não for explicitada como falsa o bloco de instrução será executado diversas vezes gerando um loop infinito.

○ **formato: while (condicao) { instrucao }.**

### • label

- Gera um identificador que permite que esse seja identificado em outro lugar no programa, onde se utilizará um break ou continue para sinalizar a finalização do laço.

- formato: **label : declaracao.**
- pouco uso.
- **break**
  - Normalmente utilizado para finalizar laços.

exemplo:

```
for (i = 0; i < a.length; i++){
 console.log(`valor de i: ${i}`);
 if (a[i] == theValue){
 console.log('Parou mano');
 break;
 }
 console.log(`valor: ${a[i]}`)
}
```

- **continue**
  - Encerra a interação atual mais interna de uma instrução e continua a interação do laço a partir da próxima interação. Além disso, quando este tipo de comando for aplicado em um while ele irá voltar para condição, de mesmo modo que se for aplicado em um for será mandado para incrementação.

exemplo:

```
while (i < 5){
 i++;
 if (i == 3){
 continue;
 }

 n += i;
 console.log(`valor de i: ${i}`);
 console.log(`valor de n: ${n}`);
}
```

- **for in**
  - Percorre todas as propriedades do objeto de uma variável específica.
  - interage com o nome das chaves de objeto e índices para arrays
  - formato: **for (variavel in objeto) { declaracoes };**

exemplo:

```
for (let i in frutas){
 console.log(frutas[i]);
}
```

- **for of**

- Cria laços com objetos iterativos, executando interação para o valor de cada propriedade distinta.
- seus objetos iterativos incluem: **array, map, set**
- formato: **for (variavel of objeto){ declaracoes }**
- interage com o valor das propriedades

exemplo:

```
for (let i of frutas){
 console.log(i);
}
```

## 2.6 Document Object Model (DOM)

**Representação de dados dos objetos que compõem a estrutura e o conteúdo de um documento Web**, ou seja, uma interface de programação com documentos HTML e XML. Deste modo, podemos alterar a estrutura, estilo e o conteúdo do documento.

O DOM por definição determina os objetos que descrevem fundamentalmente um documento e os objetos dentro dele, além disso é independente de qualquer linguagem de programação.

### Tipos de dados (interface)

Document

- Ponto de entrada para o conteúdo da página, provendo funcionalidades globais ao documento
- Todo objeto document implementa a interface Document, Node e EventTarget

#### ○ Propriedades:

- **Document.characterSet:** retorna codificação de caracteres usados no documento.

exemplo:

```
alert(document.characterSet);
```

retorna um alert com a codificação.

- **Document.documentURI:** retorna a URL do documento.

Exemplo:

```
const URI = document.documentURI;
```

retorna um alerta com a URL.

#### ○ Propriedades (Extensão):

- **Document.activeElement:** retorna o elemento atualmente focado, ou seja, o elemento que receberá os eventos do usuário em caso de digitação. Além disso, este tipo de propriedade geralmente retorna um input ou textarea e pode obter mais informações com as propriedades: `selectionStart` e `selectionEnd`. Assim, em casos onde não estiver nada selecionado o `activeElement` será o próprio body da página.

Exemplo:

```
let selectTextArea = document.activeElement;
```

```
let selection = selectTextArea.value.substring(
 selectTextArea.selectionStart,
 selectTextArea.selectionEnd);
```

- **Document.body:** Retorna o `<body>` elemento do documento atual.

- **Document.forms:** Retorna uma lista de <form>, formato:  
**document.forms[index]** - acessa o forms completo ou  
**document.forms[index].elements[index]** - acessa um elemento específico.

#### Exemplo:

```
const loginForms = document.forms.login;
loginForms.elements.email.placeholder += "diogoeng19@gmail.com";
loginForms.elements.password.placeholder = "password";
```

- **Document.head:** Retorna o head
- **Document.readyState:** retorna o status de carregamento do documento, como por exemplo: **“loading”** (carregando), **“interactive”** (quase carregado) e **“complete”** (completamente carregado).

#### Exemplo:

```
switch (document.readyState){
 case "loading":
 let p = document.createElement('p');
 p.textContent = "carregando.";
 document.body.appendChild(p);
 case "interactive":
 let p1 = document.createElement('p');
 p1.textContent = "finalizando.";
 document.body.appendChild(p1);
 case "complete":
 let p2 = document.createElement('p');
 p2.textContent = "documento carregado.";
 document.body.appendChild(p2);
}
```

#### ○ Métodos:

- **Documento.createAttribute(nomeAtributo):** cria um nó de atributo e o retorna (ex de atributo: id, class, lang. tudo que se encontra dentro de uma tag).

#### Exemplo:



```
const att = document.createAttribute("class");
att.value = "democlass";

const h1 = document.getElementsByTagName("h1")[0];
h1.setAttributeNode(att);
```

- **Document.createDocumentFragment():** cria um novo vazio onde nós DOM pode ser adicionados para construir uma árvore DOM fora da tela

### Exemplo:

```
const element = document.getElementsByClassName('teste')[0];
const fragment = document.createDocumentFragment();

const browser = ['chrome', 'firefox', 'opera', 'safari'];

browser.forEach((valoresBrowser) => {
 const li = document.createElement('li');

 li.textContent = valoresBrowser;

 fragment.appendChild(li);
});
```

- **Document.createElement(nome):** cria novo elemento com o nome da tag fornecida.

### Exemplo:

```
function adcElement(){
 let divNova = document.createElement('div');

 let conteudo = document.createTextNode("Olá, cumprimentos");

 divNova.appendChild(conteudo);

 let divAtual = document.getElementsByClassName("teste")[0];
 document.body.insertBefore(divNova, divAtual);
}
```

**obs: insertBefore está inserindo “divNova” em “divAtual”.**

- **Document.createTextNode(texto de string):** cria um nó de texto.

**Exemplo:**

```
let divNova = document.createElement('div');

let conteudo = document.createTextNode("Olá, cumprimentos");

divNova.appendChild(conteudo);
```

- **Document.getElementsByClassName(nome da classe em string):** retorna lista de elementos com o nome de classe fornecido.

**Exemplo:**

```
let divAtual = document.getElementsByClassName("teste")[0];
```

quando invocado em um documento, o mesmo será examinado por completo.

- **Document.getElementsByTagName(nome da tag em string):** retorna lista de elementos com o nome de tag fornecido.

**Exemplo:**

```
let allPass = document.getElementsByTagName('p');
```

- **Document.getElementById(string\_id):** retorna uma referência de objeto ao elemento identificado.

**Exemplo:**

```
let div1 = document.getElementById('teste');
```

- **Document.querySelector(string\_selector):** retorna o primeiro nó element no document que corresponde ao grupo especificado.

### Exemplo:

```
let idTeste = document.querySelector('#teste');
```

**obs: pode ser adicionados outros tipos de elementos ou atributos.**

- pode conter um ou mais seletores separados por vírgula.
  - **Document.querySelectorAll(string\_selector):** retorna uma lista de todos os nós elements. Obs: tem a mesma ideia de “querySelector()”

### Node

- Interface onde diversos tipos de DOM herdam, permitindo que os mesmo sejam tratados de forma similar
  - **Propriedades:**
    - **Node.childNodes:** retorna todos os filhos vivos do nó a medida que ele vai atualizando. Ou seja, childNodes retorna uma coleção ordenada de objetos node que são filhos do elemento corrente.

### Exemplo:

```
const filhos = content.childNodes;
console.log(filhos);
```

- **formato: variável = noReferencia.childNodes;**
- **tudo que for retornado serão apenas objetos.**
  - **Node.nodeName:** propriedade de leitura que retorna o nome do elemento atual como uma string.

### Exemplo:

```
// *** NODE.NODENAME ***
const div = document.querySelector('.content');
const nomeNode = div.nodeName;
```

- **Node.nodeType**: retorna um unsigned short representando o tipo do nodo.

#### Exemplo:

```
const div = document.querySelector('.content');
console.log(div.nodeType);
```

- **Node.nodeValue**: retorna ou define o valor em formato de string do nó atual. Ademais, quando retornado nós de texto, comentário e CDATA o conteúdo do nó é retornado, agora quando nos referimos a um atributo o valor de retorno é o conteúdo do atributo.

#### Exemplo:

```
// *** NODE.NODEVALUE ***
let div = document.querySelector('div').lastChild;
div.nodeValue += "<p>fazendo teste</p>";
```

**obs: para o devido funcionamento inserir lastchild ou firstchild dependendo da onde deseja inserir o código.**

- **Node.parentNode**: retorna node pai de um nó. Caso esse mesmo nó seja o pai de todos ou não tenha filhos, retorna null.

#### Exemplo:

```
// *** NODE.PARENTNODE ***
let li = document.querySelector('.li1');
console.log(li);
const parentLi = li.parentNode;
console.log(parentLi);
```

**Propriedade somente de leitura.**

- **Node.textContent**: retorna ou define o conteúdo de texto de um nó e dos seus descendentes.

### Exemplo:

```
// *** NODE.TEXTCONTENT ***
let li = document.querySelector('.li1');
const texto = li.textContent;
console.log(texto);
li.textContent = "mudança de texto";
const texto2 = li.textContent;
console.log(texto2);
```

- formato: `let text = node.textContent; node.textContent = string.`

#### ○ Métodos:

- **node.appendChild:** adiciona um nó ao final da lista de filhos de um nó pai especificado, caso esse nó já exista em outro pai, ele será removido e colocado no pai atual.

### Exemplo:

```
// *** NODE.APPENDCHILD ***
const li = document.createElement('li');
const ul2 = document.querySelector('.ul2');
li.setAttribute('class', 'class:"li13"');
li.textContent = 'teste 13';
ul2.appendChild(li);
```

- formato: `elemento.appendChild(elementChild)`
- Esse tipo de método não permite mover o nó entre diferentes documentos.
- **Node.cloneNode():** duplica um elemento node da estrutura de um documento DOM.

### Exemplo:

```
// *** NODE.CLONENODE ***
const ul = document.querySelector('.ul1');
const ulclone = ul.cloneNode(true);
let body = document.body;
body.appendChild(ulclone);
```

- formato: `node.cloneNode(deep)`, `deep` será `true` quando se desejar clonar os elementos filhos do nó clonado e `false` para clonar apenas o elemento pai.
- este tipo de método não faz a clonagem da formatação de estilo do elemento.

- **Node.contains():** indica se um nó é um descendente de um dado nó.

#### Exemplo:

```
// *** NODE.CONTAINS ***
const ul = document.querySelector('.ul1');
const body = document.body;
console.log(body.contains(ul));
```

- formato: `node.contains( otherNode )`.

- **Node.insertBefore:** insere um nó antes do nó de referência como um filho de um nó pai especificado, caso o filho especificado já exista, será movido da sua posição atual para a nova posição. Além disso, se o nó especificado for null, ele será posicionado no final da lista de filhos do nó pai especificado.

#### Exemplo:

```
// *** NODE.INSERTBEFORE ***
const ulPai = document.querySelector('.ul1');
const liChild = document.createElement('li');
const liRefe = document.querySelector('li4');

ulPai.insertBefore(liChild, liRefe);
```

```
// *** NODE.INSERTBEFORE ***
const ulPai = document.querySelector('.ul1');
const liMove = document.querySelector('li6');
const liRefe = document.querySelector('li4');

ulPai.insertBefore(liMove, liRefe);
```

- **Node.removeChild():** remove um nó filho do DOM.

#### Exemplo:

```
// *** NODE.REMOVECHILD ***
const div = document.querySelector('.content');
console.log(div);
const ul = document.getElementsByClassName('ul1')[0];
div.removeChild(ul);
```

- **Node.replaceChild():** substitui o elemento filho especificado por outro.

#### Exemplo:

```
// *** NODE.REPLACECHILD ***
const liNew = document.createElement('li');
liNew.textContent = 'teste de um teste';
const ul = document.querySelector('.ul2');
const liExisting = document.querySelector('.li9');

ul.replaceChild(liNew, liExisting);
```

## Element

- Classe de base mais geral onde todos os objetos que um document herda. Portanto, possuindo métodos e propriedades comuns para todos os tipos de elementos.

- **Propriedades:**

- **element.attributes:** lista todos os atributos associados ao elemento.

### Exemplo:

```
let p = document.getElementsByTagName("p")[0];
let attr = p.attributes;
```

- **element.children:** retorna um live HTMLcollection que contém todos os filhos elements do elemento que foi marcado.

### exemplo:

```
const elemento = document.getElementById("bloco-teste");
for (const child of elemento.children){
 console.log(child.tagName);
}
```

- utilize item() para acessar cada um dos filhos individualmente.

- **element.classList:** retorna lista de atributos de classe.

### exemplo:

```
//ELEMENT.CLASSLIST
const div = document.getElementsByTagName("div")[0];
div.classList.remove("teste");
div.classList.add("teste3");
div.classList.toggle("done");
```

- somente para leitura, mas pode ser modificado com **add(string)** - adiciona valores de classe especificado e **remove(string)** - remove valores de classes especificos.
- para o ler o valor de uma classe especifica utilizasse item(number).
- **toggle (string)** - com apenas um argumento: se existir uma classe remova-o e retorne false, caso contrário adicione e coloque true. com dois argumentos - se o segundo argumento for avaliado como true adiciona o valor especificado da classe, em contrapartida, se for false remova-o
- **contains(string)** - verifica se o valor do atributo de classe existe.

- **element.className:** retorna e define o valor do atributo classe do elemento especificado.

#### exemplo:

```
//ELEMENT.CLASSNAME
const div = document.querySelector(".teste");

div.className = "teste4";

console.log(div.className);
```

- **element.id:** representa um id do elemento.

#### Exemplo:

```
// *** ELEMENT.ID ***
const div = document.querySelector("div");
console.log(div.id);
```

, resposta - **content-test-ID**

- **element.innerHTML:** representa a marcação do conteúdo do elemento, além disso define ou obtém a sintaxe html descrevendo os elementos descendentes.

#### Exemplo:



```
document.getElementById("txt").innerHTML;
,
const contentEst = document.getElementsByClassName
("content-test")[0];
contentEst.innerHTML += "<form>5</form>";
```

obs: Se um nó <div>, <span> e <noembed> possuir um filho que tenha (&), (<) e (>) no seu interior, o mesmo nó pai irá retornar &amp;, &lt; e &gt; respectivamente.

- **ParentNode.firstChild**: primeiro elemento filho direto de um elemento.

Exemplo:

```
// *** ELEMENT.FIRSTELEMENTCHILD
const contentEst = document.getElementsByClassName
("content-test")[0];
console.log(contentEst.firstChild);
```

Propriedade somente de leitura.

- **ParentNode.lastElementChild**: último elemento filho direto de um elemento.

Exemplo:

```
// *** ELEMENT.LASTELEMENTCHILD ***
const contentEst = document.getElementsByClassName
("content-test")[0];
console.log(contentEst.lastElementChild);
```

- **Element.outerHTML**: descreve o elemento incluindo os seus descendentes.

Exemplo:

```
// *** ELEMENT.OUTERHTML ***
const contentEst = document.getElementsByClassName(
 "content-test")[0];
console.log(contentEst.outerHTML);
```

```
// *** ELEMENT.OUTERHTML ***
const contentEst = document.getElementsByClassName(
 "content-test")[0];
console.log(contentEst.outerHTML);

const form = document.querySelector(".form-test2");
console.log(form);

form.outerHTML += '<li class="li-test">Valéria'
```

- Há possibilidade de substituição de nós obtidos através da análise de uma string.
- se tentar alterar um elemento raiz do documento há possibilidade de ser gerado um erro `DOMException`.

- **Element.tagName:** retorna o nome do elemento.

### Exemplo:

```
// *** ELEMENTO.TAGNAME ***
const div = document.getElementById("content-test-id");
console.log(div.tagName);
```

#### ○ Métodos:

- **EventTarget.addEventListener():** registra um manipulador de evento para um tipo específico de evento do elemento, mas a possibilidade de capturar diferentes tipos de eventos, basta registrar mais de uma espera de evento como alvo.

### Exemplo:

```
let el = document.getElementById("t");
el.addEventListener("click", modifyText, false);
```

```
el.addEventListener('click', this, false);
el.addEventListener('dblclick', this, false);
```

- `formato: elemento.addEventListener(typeEvent, listener{Capture Method})` // `captureMethod` - indica o início da captura do evento, `listener` - recebe notificação quando ocorre o evento e `type` - tipo de evento.

- **`Element.getAttribute()`**: Recupera um valor de atributo nomeado e o retorno como um objeto.

### Exemplo:

```
let div = document.getElementsByClassName('div-test')[0];
let align = div.getAttribute("align");
```

```
let div = document.getElementsByClassName('div-test')[0];
let align = div.getAttribute("align");
let id = div.getAttribute("id");
```

- `formato: let atributo = element.getAttribute(nomeDoAtributo).`
- **`Element.getElementsByTagName()`**: retorna todos os elementos descendentes da classe ou das classes especificadas.

### Exemplo:

```
// *** ELEMENT.GETELEMENTSBYCLASSNAME() ***
let div = document.getElementsByClassName('div-test')[0];
let divEl = div.getElementsByTagName('table-test')[0];
console.log(divEl);
```

- `formato: let elements = element.getElementsByTagName(names);.`
- O mesmo uso se tem para `getElementsByTagName()`.
- **`Element.querySelector()`**: ...
- **`Element.querySelectorAll()`**: ...
- **`Element.remove()`**: remove um elemento do DOM.

### Exemplo:

```
// *** ELEMENT.REMOVE ***
let div = document.getElementsByClassName('div-test')[0];
let div_td = div.getElementsByTagName('td')[1];
div_td.remove();
```

- **Element.removeAttribute():** remove um atributo de um elemento específico.

### Exemplo:

```
// *** ELEMENT.REMOVEATTRIBUTE ***
let div = document.getElementsByTagName('div')[0];
console.log(div);
div.removeAttribute('id');
console.log(div);
```

- formato: `element.removeAttribute(attrName);`.
- **EventTarget.removeEventListener():** remove o event listener registrado anteriormente. Ademais, se removido enquanto processa um evento, este não será disparado pelas current actions.

### Exemplo:

```
// *** EVENTTARGET.REMOVEEVENTLISTENER ***
let div = document.getElementsByTagName('div')[0];
let listener = function(evente){
 const div = document.createElement('div');
 const body = document.body
 body.appendChild(div);
};
div.addEventListener('click', listener, false);
setTimeout(() => {
 div.removeEventListener('click', listener, false);
}, 10000)
```

- formato: `target.removeEventListener(type, listener[, useCapture])`.
- **Element.setAttribute():** define um valor de um atributo nomeado do nó atual.

### Exemplo:

```
// *** ELEMENT.SETATTRIBUTE ***
let div2 = document.getElementsByClassName('div-2')[0];
console.log(div2);
div2.addEventListener('click', e =>{
 div2.setAttribute("style", "background: pink;")
});
```

- formato: `element.setAttribute(name, value);`

### Nota: outros tipos.

- NodeList
- Attribute
- NamedNodeMap

### Elementos e funcionalidades

- **getElementsByTagName:** retorna lista de todos os elementos.

### exemplo:

```
let paragrafos = document.getElementsByTagName("p");
```

- **alert:** exibe alerta quando o documento for carregado.

### exemplo:

```
alert(paragrafos.nodeName)
```

nesse caso pode tanto ser usado `document.alert` ou somente `alert`.

- **createElement:** cria elementos html.

### exemplo:

```
let heading = document.createElement("h1");
```

- **createTextNode:** cria texto para documentos html.

exemplo:

```
let heading_text = document.createTextNode("Big Node");
```

- **appendChild:** adiciona um elemento a outro.

exemplo:

```
heading.appendChild(heading_text);
// adiciona h1 no body
document.body.appendChild(heading);
```

**Obs: Nós = elementos**

## 2.7 Try ... Catch

Esse tipo de declaração tem por meta **marcar um bloco de declarações para testar (try) determinada ação e especificar uma resposta caso uma exceção seja lançada.**

Exemplo:

```
// caso haja erro o bloco try não será executado
try {
 console.log(soma(1, 2));
 console.log(soma('1', 2));
 // passa para o bloco catch em caso de erro
} catch (erro) {
 console.log(erro);
}
```

o “erro” dentro de catch serve para capturar o valor especificado pela declaração.

Basicamente o try...catch pode ser escrito de 3 formas, são elas: **try...catch**, **try...finally** e **try...catch...finally**. Logo após, é importante ressaltar que catch é normalmente usado para o tratamento de alguns erros que podem vir a ocorrer

durante a execução de try, e finally será um bloco executado após todas as instruções de try. Também, **podemos aninhar diversas declarações try sem que haja algum erro de código colocando uma dentro da outra, nesse caso catch pertencera a try mais exterior.**

Podemos criar uma declaração try com dois tipos de catch, a primeira seria aquela que não possui nenhum tipo de condicional (catch incondicional) e a segunda é aquela que possui um condicional para que seja executada (catch condicional), caso sejamos deparados com ambas as declarações o js primeiro irá executar aquelas que apresentam condicionais e ao final executar as que não possuem condicionais.

### Exemplo:

```
try{
 try{
 throw new Error("ops");
 }catch(e){
 console.error("inner", ex.message);
 }finally{
 console.log("finally");
 }
}catch(e){
 console.error("outer", ex.message);
}
```

### INFORMAÇÕES EXTRAS:

**throw**: usado como identificador de exceção que foram lançadas, onde após ser lançado para toda a instrução passando o comando para o primeiro bloco catch.

**formato:** throw "Erro2", throw "42", throw "true"

**Error**: usado quando a ocorrência de erros de tempo de execução

- **EvalError**: erro que ocorre na função global
- **InternalError**: erro interno na engine do JavaScript
- **RangeError**: erro quando um valor ou parâmetro numérico está fora de seus limites válidos
- **ReferenceError**: erro de referência inválida
- **SyntaxError**: erro de código
- **TypeError**: erro de variável ou parâmetro do tipo inválido

## 2.8 setInterval e setTimeout

**setInterval()** serve para repetir chamadas de funções ou executar trechos de códigos com um tempo de espera fixo entre cada chamada, como também definir um atraso para funções que são executadas repetidamente como animações.

Formato:

- **let intervalID = scope.setInterval(func, delay[param1, param2]);**
  - func é uma função que será executada a cada delay em milissegundos.
- **let intervalID = scope.setInterval(code, delay)**
  - string que será executada da mesma forma que a função, mas não é uma sintaxe recomendada em virtude de segurança de código.

**Normalmente o retorno de setInterval e setTimeout é um intervalID em formato numérico, este valor pode ser passado para clearInterval quando se deseja cancelar o timeout.**

todas essas expressões podem ser colocadas em conjunto para resolver determinada situação, porém não deve ser usada pois pode gerar conflitos na manutenção do código.

**Exemplos:**

```
let intervalID = setInterval(function(){
 console.log('bom dia');
}, 500);
```

Um dos problemas que setInterval() e no setTimeout() pode está relacionado ao this, pois a função que se encontra dentro do setInterval não é a mesma que é referenciada pelo this.

**o this presente dentro de setInterval é setado como objeto window ou global.**

**O método global setTimeout() define um cronômetro que executa uma função ou trecho de código assim que o cronômetro expira.**



- esse método é considerado uma função assíncrona, ou seja, ela não irá pausar outras funções que possam vir depois dela.

#### Exemplo:

- `setTimeout(code, delay, ...paramsN);`

#### Exemplo:

```
setTimeout(function(){
 console.log("diogo dias")
}, 5000)
```

```
setTimeout(() => {
 console.log('primeira contagem');
}, 5000);

setTimeout(() => {
 console.log('segunda contagem');
}, 8000);

setTimeout(() => {
 console.log('terceira contagem');
}, 1000);

setTimeout(() => {
 console.log('quarta contagem');
}, 3000);
```

funcionalidade assíncrona.

## 3. Funções Avançadas

### 3.0 Função de Callback

Este tipo de função é normalmente utilizado como argumento de uma função mais externa que será executada com o intuito de completar uma rotina ou ação.

- os callbacks são usados normalmente para a execução do código após uma operação assíncrona, eles então recebem o nome de asynchronous callbacks.

#### Exemplo:

```

<script>

 function greeting(name){

 alert('Olá' + " " + name)

 }

 function processUserInput(callback){

 let name = prompt("Por favor insira o seu nome: ");
 callback(name);

 }

 processUserInput(greeting);

</script>

```

### 3.1 Função imediata

**As immediately invoked function expression é uma função que como o nome já diz será executado imediatamente assim que é definida.**

- podemos dizer que essa função possui um design pattern contendo as seguintes partes: a **primeira seria** uma função anônima cujo escopo léxico é encapsulado entre parênteses prevenindo acesso externos às variáveis declarada na IIFE e a **segunda parte** está relacionada a criação de expressão ().

A função se torna uma expressão que é imediatamente executada, também todas as variáveis presentes dentro do seu escopo não podem ser acessadas por fora.

**Exemplo:**

```

let nome = (function(){
 let nome = "Diogo";
})();

```

## 3.2 Função fábrica (Factory functions)

A factory function é reconhecida por encapsular o desenvolvimento de objetos a partir de um de seus métodos, assim é possível que seu funcionamento se dê como uma fábrica de objetos propriamente ditos.

as funções retornam objetos sem a utilização da palavra "new", bem como oferece uma melhor performance em caso de instancição de grande número de objetos. Desse modo, elas têm a mesma função que as “funções construtoras” (constructor functions), retornando somente objetos.

Com este tipo de função podemos isolar métodos e propriedades de maneira que ambos fiquem privados. Na sequência, caso busque deixar que métodos e propriedades sejam modificadas basta utilizar o padrão ice factory por meio do Object.freeze().

Exemplo:

```
function test(text, color, background){
 function el(){
 const paragraph = document.getElementsByTagName('p')[0];
 paragraph.textContent += text;
 paragraph.style.color = color;
 paragraph.style.backgroundColor = background;

 return paragraph;
 }

 return {
 el,
 text
 }
}
```

Exemplo:

```
function createNewParagraph(text, color, background){

 function element(){

 const paragraph = document.querySelector("p");
 paragraph.textContent += text;
 paragraph.style.color = color;
 paragraph.style.backgroundColor = background;

 return paragraph;

 }

 return Object.freeze({

 element,
 text

 });

}

let test = createNewParagraph("text new", "#000", "#fff");
console.log(test);
```

### 3.3 Função Construtora

Esses tipos de funções servem como molde para criação de objetos e para instanciar esse tipo função é importante o uso do operador **new**, assim como a utilização do **this** que irá referenciar o objeto criado a partir delas.

**Exemplo:**

```
function Pessoa(nome, sobrenome, idade){

 console.log(this);
 this.nome = nome;
 this.sobrenome = sobrenome;
 this.idade = idade;

}

let pessoa = new Pessoa('Diogo', 'Mello', 20);
console.log(pessoa);
```

**obs: Na maior parte dos casos os construtores não necessitam de return por tudo ser passado ao this.**

caso seja usado return em um construtor, ele irá retornar no lugar de this.

Exemplo:

```
function Pessoa1(aluno){
 this.aluno = aluno;
 return { nome: "alexa" }
}
```

retorna “alexa”.

### 3.4 Função Recursiva

**Este tipo de função está muito caracterizada por chamar a si mesmo até encontrar uma instrução de parada**, técnica assim chamada de recursão.

- para este tipo de função é sempre importante que haja um ponto de parada para o seu laço de repetição, caso contrário ela irá ser chamada infinitamente.

**A recursividade de uma função sempre é usada com o intuito de dividir grandes problemas em problemas ainda menores.**

- elas são muito utilizadas em estruturas de dados, árvores binárias, grafos e algoritmos de pesquisa binária.

É possível notar que o nome da função faz referência ao objeto de função real e se em algum lugar do código o nome da função for definido como null, a função recursiva irá parar de funcionar.

```
formato: function recurse(){
 ...
 recurse();
 ...
}
```

Exemplo:

```
function countDow(fromNumber) {
 console.log(fromNumber);

 let nextNumber = fromNumber - 1;

 if(nextNumber > 0) {
 countDow(nextNumber);
 }
}

countDow(3);
```

```
function countDow(fromNumber) {
 setTimeout(() => console.log(fromNumber);

 let nextNumber = fromNumber + fromNumber;

 if(nextNumber < 50) {
 countDow(nextNumber);
 }
}, 2000)
}
```

### 3.5 Função Geradora

A função geradora retorna um objeto Generator. Além disso, **geradores são funções cuja execução pode ser interrompida e posteriormente reconduzida**, logo seus **contextos de associação de variáveis ficarão salvos entre cada recondução**.

É **possível combinar promises com geradores** em js, desenvolvendo assim uma **ferramenta poderosa para programação assíncrona**, mitigando ou até eliminando problemas como callbacks. (funções async são fundamentadas nisso)

**Ao executar a função é retornado um objeto iterator que poderá chamar seu método next() fazendo com que o conteúdo da função do gerador seja executada até a primeira expressão yield**, que especifica o valor a ser devolvido do iterator. Além disso, podemos usar yield\* para delegar para outra função geradora determinada ação.

- next() retorna um objeto com uma propriedade value contendo o valor retornado e a propriedade boolean: done indicando se o gerador produziu

seu último valor; também, se colocarmos argumentos em next() fará com que substitua a expressão yield dentro do código.

**Return dentro de um gerador quando executado irá finalizar a operação do gerador**, onde a propriedade done do objeto será true e o valor a ser retornado será usado como propriedade value do objeto retornado pelo gerador.

- qualquer erro que seja lançado dentro do gerador finaliza qualquer atividade a não ser que seja tratado dentro do corpo do gerador.
- futuras chamadas next não iriam executar nenhum código do gerador.

### Exemplo:

```
function* idMaker(){
 let index = 0;
 while(true){
 yield index++;
 }
}

let gen = idMaker();

console.log(gen.next());
console.log(gen.next().value);
```

```
function* gerador(i){
 yield i + 1;
 yield i + 2;
 yield i + 3;
}

function* gerador2(i){
 yield i;
 yield* gerador(i);
 yield i + 10;
}

let gen2 = gerador2(5)

console.log(gen2.next());
console.log(gen2.next());
console.log(gen2.next());
console.log(gen2.next());
console.log(gen2.next());
```

```
function* gerador(){
 console.log(yield);
}

let gen = gerador();

gen.next('diogo');
gen.next('diogo');
```

## 4. Arrays Avançados

### 4.0 Filter

O **Filter** é um método que cria uma cópia superficial de uma parte de uma determinada matriz, filtrada apenas para os elementos da matriz fornecida que passam no teste implementado pela função fornecida.

exemplo:

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction'];

const result = words.filter(word => word.length > 6);

console.log(result);
```

Ao percorrer os elementos o resultado obtido são todos os que possuem tamanho maior que 6, ["exuberant", "destruction"].

**Formato:**

**\*\*\*ARROW FUNCTION\*\*\***

`filter((element) => { ... })`

`filter((element, index) => { ... })`

`filter((element, index, array) => { ... })`

**\*\*\*CALLBACK FUNCTION\*\*\***

`filter(callbackfn)`

`filter(callbackfn, thisArg)`

**\*\*\*INLINE CALLBACK FUNCTION\*\*\***

`filter(function(element){ ... })`

`filter(function(element, index){ ... })`

`filter(function(element, index, array){ ... })`

`filter(function(element, index, array){ ... }, thisArg)`



Os `callbacksFn` são funções que serão executadas para cada elemento dentro da matriz, onde seu retorno deve ser `true` para que seus elementos sejam mantidos.

- `element`, `index` e `array` que também fazem parte do método `filter` representam todos processos que podem ser feitos com cada propriedade do array.
- temos `thisArg` que é um valor a ser usado como `this` ao executar uma função de callback.

Conhecido como método de cópia, o `filter` não altera a matriz em si, e sim cria uma cópia dos seus valores que passaram pelo teste de filtragem.

- O comprimento da matriz será salvo antes mesmo da filtragem da seus valores, fazendo com que a função de callback não passe pelos valores novamente após iniciada a filtragem.

**Exemplo: filtragem de números primos.**

```
const array = [-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13];

function nPrimos(valor){
 for(let i = 2; valor > i; i++){
 if(valor % i === 0){
 return false;
 }
 }

 return valor > 1;
}

console.log(array.filter(nPrimos)); // [2, 3, 5, 7, 11, 13]
```

## 4.1 Map

O método `map()` invoca uma função de callback passada por argumento para cada elemento do array e devolve um novo array como resultado.

- o seu funcionamento se dá por meio de chamada de função que irá ser executada de forma ordenada para todos os valores que estão presentes dentro array e que **não possuem valor `undefined` ou foram removidos ou modificados durante a chamada.**

**formato:**

`arr.map(callback[, thisArgs])` - `callback` (função que retorna um novo array - argumentos: valor, índice e array) e `thisArg` - opcional.

O valor `this`, ao ser passado como parâmetro de `map`, ele será repassado para a função de `callback` assim que a mesma for invocada. Caso contrário, um valor `undefined` será repassado no lugar do `this`.

Exemplo:

```
const array = ['Diogo', 'Carol', 'Tatiana', 'Cristian', 'Geovana',
 'João Lucas', 'Silmara', 'Dulcila', 'Claudio', 'Valério'];

const numArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

let newArray = numArray.map(Math.sqrt);
newArray = array.map((value, indice, array) =>{
 | return value + ' ' + (indice + 1);
 |})

// console.log(array);
console.log(newArray);
```

usando `map` genericamente.

```
let map = Array.prototype.map;

let a = map.call('hello world', function(x) {return x.charCodeAt(0)});

console.log(a);
```

## 4.2 Reduce

O método `reduce()` executa uma função `reduce` para cada elemento do array, resultando num único valor de retorno.

- Esse tipo de função pode receber quatro parâmetros como `acumulador(acc)` que possui o valor de retorno, `valor atual(cur)`, `index atual(idx)` e `array original(src)`.
- seu `acumulador` terá seu valor repassado para cada iteração subsequente pelo array, fazendo com que resulte em um valor único final.

formato:

**array.reduce(callback(acumulador, valorAtual, index, array), valor inicial).**

- **callback** - função executada em cada valor no array (exceto no primeiro, se nenhum valor inicial for passado), recebe 4 elementos;
- **acumulador** - valor inicial, este valor inicia com o valor inicial e será retornado na última interação;
- **valorAtual** - índice do elemento atual que está sendo processado no array. Começa do índice 0 se um valor inicial for fornecido, caso contrário, começa do 1;
- **valorInicial** - valor a ser usado como primeiro argumento da primeira chamada da função callback. **Se nenhum valorInicial é fornecido, o primeiro elemento do array será usado como valor inicial do acumulador e o valorAtual não será lido.**

**O funcionamento do método ocorre pela chamada do callback**, onde na primeira chamada o acumulador pode ser igual ao valorInicial se for fornecido e o valorAtual será igual ao primeiro valor do array.

- caso não seja fornecido um valorInicial o acumulador irá obter o primeiro valor do array e o valor atual irá ficar com o segundo valor do array.

**Nota: Em casos onde o array se encontre vazio, uma exceção do tipo TypeError será lançada, agora se possuímos apenas um valor dentro do array e o valor inicial não for fornecido ou mesmo se for a função de callback não será chamada.**

**Exemplo:**

```
const array = [1, 2, 4, 8, 16];

const soma = array.reduce((acumulador, valor, indice, array) =>{
 | return (acumulador += valor);
 |});

console.log(soma);
```

```
const array = ['Diogo', ' ama muito', ' a Geovana', ' e a familia dele'];

const soma = array.reduce((acumulador, valor, indice, array) =>{
 | return (acumulador += valor);
}, 'obs: ');

console.log(soma);
```

## 5. Objetos prototypes avançados

### 5.0 this

**palavra-chave this** - determina a forma como uma função pode ser chamada e é estabelecida segundo o escopo de execução no qual está inserido.

Nota: Se this for encontrado fora do escopo de uma função ele irá indicar o objeto global, seja no modo estrito ou não, agora se estiver dentro de qualquer função irá referenciar a mesma.

#### Exemplo:

```
const valor = this;

console.log(valor);

function pessoa(){
 | const valor = this;
 | return valor;
}

console.log(pessoa());|
```

o primeiro this escrito fora da função está fazendo referência ao objeto global, enquanto o this que se encontra dentro da função faz referência a mesma.

Nota: usar “use strict” dentro da função faz com que o this não referencie o objeto global.

```
function pessoa(){
 "use strict";
 const valor = this;
 return valor;
}

console.log(pessoa());
```

retorna undefined por não ser um método ou propriedade de um objeto.

Dentro de objetos uma função é considerada um método que ao se chamar this irá fazer referência ao objeto onde se encontra.

Exemplo:

```
const obj = {
 nome: 'Diogo',
 sobrenome: 'Dias',
 f: function(){
 return this.nome;
 }
}

console.log(obj.f());
```

Obs: this sempre irá referenciar o objeto mais imediato, ou seja, supondo que se chama this dentro de uma função que se encontra em um objeto “o” que está dentro de um objeto “b”, a referência irá para b que está mais imediato.

Se o método está na cadeia de protótipo de um objeto, this refere-se ao objeto que é proprietário do método.

Exemplo:

```
let a = {f: function(){
 return this.b + this.c;
}};

let d = Object.create(a);

d.b = 1;
d.c = 4;

console.log(d.f());
```

o objeto “d” herda todos os métodos do objeto “a”.

Se houver uma função construtora seu this irá estar vinculado ao novo objeto sendo criado, ou seja, o padrão de retorno de this é o objeto referenciado ou algum outro objeto.

**Exemplo:**

```
function Pessoa (){
 this.p = 'Diogo';
}

let pe = new Pessoa();

console.log(pe.p);
```

próprio objeto.

```
function Pessoa (){
 this.p = 'Diogo Mello';
 return {p: 'Diogo Dias'}
}

let pe = new Pessoa();

console.log(pe.p);
```

retorna outro objeto (“Diogo Dias”) .

O valor de uma função pode ser vinculado a um determinado objeto na chamada utilizando os métodos **call ou apply** que são herdados do prototype de funções.

**Exemplo:**

```
function add (c, d){
 return this.a + this.b + c + d;
}

let o = {a: 1, b: 3};

add.call(o, 5, 7);

add.apply(o, [10, 20]);
```

caso os valores passados na função não fossem objetos, eles seriam transformados em objetos por meio da operação **ToObject**.

O método **Bind** serve para criar uma nova função que terá o mesmo escopo e corpo de sua origem, mas que onde o **this** ocorrer na função original estará ligado permanentemente no primeiro argumento de **bind** na nova função.

## 5.1 Protótipos de objetos

Protótipos de objetos podem herdar outros protótipos dentro deles, este tipo de comportamento gera uma cadeia de protótipos - javascript pode ser considerado como uma linguagem baseada em protótipos.

Propriedades e métodos são criados e definidos pela propriedade **prototype** nas funções construtoras dos objetos. Isso ocorre por meio de um link entre a instância do objeto e seu protótipo (**propriedade \_proto\_ derivada de prototype**).

**Obs: métodos e propriedades não são copiados de um objeto para outro na cadeia de protótipo.**

**Prototype** - bucket para armazenar propriedades e métodos que queremos que sejam herdados por objetos mais abaixo na cadeia de prototype.

### Exemplo:

```
let f = function () {
 this.a = 1;
 this.b = 2;
}

let o = new f();

f.prototype.b = 3;
f.prototype.c = 4;

console.log(o.a);
|

let f = function () {
 this.a = 1;
 this.b = 2;
}

let o = new f();

f.prototype.b = 3;
f.prototype.c = 4;

console.log(o.c);
```

Sombreamento de propriedade: quando ocorre a leitura da propriedade própria de b, mas não ocorre do protótipo.

### Pesquisa prototype:

- f > f.prototype > objeto.prototype > null
- f > f.prototype > array.prototype > objeto.prototype > null
- f > f.prototype > function.prototype > objeto.prototype > null

**herança de função**: this aponta para objeto que herda as propriedades.

**OBS: Grande impacto negativo em pesquisas de protótipos que se encontram no alto da cadeia, apresentando um crítico desempenho.**

"hasOwnProperty" - usado para verificar se um objeto tem uma propriedade definida em si.

formato: **obj.hasOwnProperty(propriedade)**

**Object.create** - cria objeto a partir de um objeto de protótipo especificado.



### Exemplo:

```
let person1 = function () {
 this.nome = "Diogo";
 this.sobrenome = "Mello";
}

let person2 = Object.create(person1);

person1.prototype.idade = 21;
person1.prototype.email = "diogoeng19@gmail.com";

console.log(person2.__proto__);
```

```
> person2.__proto__
< f () {
 this.nome = "Diogo";
 this.sobrenome = "Mello";
}
```

**obs: Se usar this dentro da modificação de um prototype, o mesmo irá referenciar o escopo global.**

Uma forma de melhorar os códigos seria **definir propriedades dentro do construtor e métodos no protótipo.**

**Object.defineProperty()** - define nova propriedade diretamente em um objeto ou modifica uma existente.

### formato:

**Object.defineProperty(obj, prop, descriptor)**

- **obj** - objeto que será definida propriedade.
- **prop** - propriedade que será definida ou modificada.
- **descriptor** - descritor para propriedade.

**Valores inseridos por meio da propriedade Object.defineProperty() são imutáveis**, ou seja, não poderão ser mudados da mesma forma que propriedades colocadas por atribuição.

### Descritores de propriedades presentes nos objetos:

- dado: contém valor, pode ser gravável ou não
- assessor: definida com par de funções getter-setter

### Chaves presentes nos descritores:

- configurable: true se pode ser alterado ou deletado. Padrão false
- enumerable: true se propriedade aparece em enumeração das propriedades no objeto. Padrão false
- value: valor associado a propriedade. Padrão undefined
- writable: true se valor poder ser modificado por atribuição. Padrão false
- get: função chamada sem argumentos com this referenciando o objeto. Valor retornado será como valor da propriedade. Serve como getter ou undefined
- set: chamada como argumento com this configurando o objeto onde se encontra a propriedade. Padrão undefined. Serve com setter ou undefined.

**obs: se não possuir todas as chaves será de dados. Caso contrário uma exceção será lançada.**

### Exemplo:

```
function Produto(nome, preco, estoque){
 this.nome = nome;
 this.preco = preco;

 Object.defineProperty(this, 'estoque', {
 // mostrando o estoque
 enumerable: true,
 // valor do estoque
 value: estoque,
 // fazendo com que o valor de estoque não possa ser alterado
 writable: false,
 // informando se a variavel pode ser alterada
 configurable: false
 });
}
```

**Nota: pode ser lançado um `TypeError` caso tente alterar uma propriedade não alterável, mas quando falamos de `writable` o processo já é um pouco diferente.**

**Obs: em métodos getter e setter, this aponta para o objeto no qual é usado para acessar ou modificar a propriedade.**

**Object.defineProperty()** - define uma nova propriedade ou modifica uma existente no objeto.

**formato:** **Object.defineProperty(obj, props).**

- **obj**: cria ou modifica as propriedades.
- **props**: propriedades enumeráveis constitui descritor para propriedades definidas ou modificadas.
  - **descritores** - configurable, enumerable, value, writable, get e set.

**DefineProperties** - define todas as propriedades correspondentes para as propriedades próprias enumeráveis de props sobre o objeto obj.

**Exemplo:**

```
function Pessoa(nome, sobrenome, idade){
 Object.defineProperty(this, {
 nome: {
 // mostrando o estoque
 enumerable: true,
 // valor do estoque
 value: nome,
 // fazendo com que o valor de estoque não possa ser alterado
 writable: false,
 // informando se a variavel pode ser alterada
 configurable: false
 },
```

```
 sobrenome: {
 // mostrando o estoque
 enumerable: true,
 // valor do estoque
 value: sobrenome,
 // fazendo com que o valor de estoque não possa ser alterado
 writable: false,
 // informando se a variavel pode ser alterada
 configurable: false
 },
```

**Métodos:**

- **Object.assign**: usado para copiar os valores de todas as propriedades próprias enumeráveis de um ou mais objetos de origem para um objeto destino.

- 

**Exemplo:**

```
const produto = {nome: 'Produto', preco: 1.80};
// copiando os valores de produto
const caneca = Object.assign({}, produto);
```

- **Object.setPrototypeOf()**: configura o prototype de um objeto específico para outro objeto ou null
  - formato: **Object.setPrototypeOf(obj, prototype)** - obj (prototype a ser definido) e prototype (novo prototype)

**exemplo:**

```
Object.setPrototypeOf(objB, objA);
Object.setPrototypeOf(objC, objB);
console.log(objB.chaveA);
console.log(objC.chaveB);
```

## 5.2 Herança

**Exemplo:**

```
function Professor(first, last, age, gender, interests, subject){
 Person.call(this, first, last, age, gender, interests);
}
```

**Call** - permite chamar uma função definida em outro lugar

- **Primeiro parâmetro** - valor this que deseja usar ao executar a função
- **Demais parâmetros** - devem ser passados para função quando invocada

**Obs: todos os parâmetros presentes dentro de call serão herdados de person()**

```
function Professor(first, last, age, gender, interests, subject){
 Person.call(this, first, last, age, gender, interests);

 this.subject = subject;
}
```

por fim, somente **"subject"** é criado dentro de professor.

**Exemplo: Se valores de propriedades do construtor não forem parâmetros o método de herança pode ser mais simples.**

```
function Brick(){
 this.width = 10;
 this.height = 20;
}

function BlueGlassBrick(){
 Brick.call(this);

 this.opacity = 0.5;
 this.color = 'blue';
}
```

BlueGlassBrick necessitou apenas do this para herdar as propriedades de Brick.

Para **Professor** herdar os **prototypes de person** é necessário utilizar **Object.create()** para criar um novo objeto e torná-lo o valor de **Professor.prototype**.

**Exemplo: propriedade construtora de Professor prototype é igual a **Person()**.**

```
Professor.prototype = Object.create(Person.prototype);
```

**Exemplo: Método de correção para apresentar somente Professor no lugar de constructor.**

```
Object.defineProperty(Professor.prototype, 'constructor', {
 value: Professor,
 enumerable: false,
 writable: true
});
```

Obs: Caso um membro existente em `Person()` seja recriado em `Professor()`, o mesmo ao ser chamado irá dar maior prioridade ao novo membro criado. No caso de `Professor`, `greeting` foi recriado para atender as necessidades do professor.

**Exemplo: Formato mais recente integrado de classes em JS.**

```
class Person{
 constructor(first, last, age, gender, interests){
 this.name = {
 first,
 last
 };

 this.age = age;
 this.gender = gender;
 this.interests = interests;
 }

 greeting(){
 console.log(`Hi! I'm ${this.name.first}`);
 };

 farewell(){
 console.log(`${this.name.first} has left the building. Bye for now!`)
 };
}
```

**recursos de classe:**

- **constructor** - define função construtora que representa class `Person`.
- **greeting()** e **farewell()** - métodos dentro da classe.

**Exemplo: instanciando a nova classe.**

```
const aluno = new Person();

const aluno = new Person('Diogo', 'Dias', 21, 'male', ['academia']);

aluno.greeting();
aluno.farewell();
```

### Exemplo:Criação de subClasse.

```
class Teacher extends Person{
 constructor(first, last, age, gender, interests, subject, grade){
 this.name = {
 first,
 last
 };

 this.age = age;
 this.gender = gender;
 this.interests = interests;

 this.subject = subject;
 this.grade = grade;
 }
}
```

Extends é usado para informar qual classe irá servir como base da subclass.

**super()** - chama o construtor da classe pai e herda os membros que especificamos como parâmetros em super().

### Exemplo:

```
class Teacher extends Person{
 constructor(first, last, age, gender, interests, subject, grade){
 super(first, last, age, gender, interests);
 this.subject = subject;
 this.grade = grade;
 }
}
```

**getter() e setter()** - o primeiro retorna o valor atual da variável e o segundo altera o valor da variável para o que ela define.

### Exemplo:

```
class Teacher extends Person{
 constructor(first, last, age, gender, interests, subject, grade){
 super(first, last, age, gender, interests);
 this._subject = subject;
 this.grade = grade;
 }

 get subject(){
 return this._subject;
 }

 set subject(newSubject){
 this._subject = newSubject;
 }
}
```

### Exemplo: chamadas de professor.

```
const professor = new Teacher('Severus', 'Snape', 58, 'male', ['Potions'], 'Dark arts', 5)
professor.greeting();
professor.farewall();
console.log(professor.age);
console.log(professor.subject);

professor.subject = "Balloom animals";

console.log(professor.subject);
```

**Delegação** - compartilhamento de funcionalidades entre objetos

## 5.3 Polimorfismo

Classes derivadas de uma classe base que apresentam uma mesma codificação só que de maneira diferente para cada uma das classes.

Seleção de funcionalidades utilizadas de forma dinâmica por um programa no decorrer de sua execução.

Um breve exemplo seria uma classe **Professor{}** herdando os métodos e propriedades de uma classe **Person{}**.



- Exemplo:

### Modelo atual

```
class Person{
 constructor(first, last, age, gender, interests){
 this.name = {
 first,
 last
 };

 this.age = age;
 this.gender = gender;
 this.interests = interests;
 }

 greeting(){
 console.log(`Hi! I'm ${this.name.first}`);
 };

 farewell(){
 console.log(`${this.name.first} has left the building. Bye for now!`)
 };
}
```

### classe PAI

```
class Teacher extends Person{
 constructor(first, last, age, gender, interests, subject, grade){
 super(first, last, age, gender, interests);
 this._subject = subject;
 this.grade = grade;
 }

 get subject(){
 return this._subject;
 }

 set subject(newSubject){
 this._subject = newSubject;
 }
}
```

- classe

### FILHA

### Modelo mais usado (antigo)

```
function Person(argumentos){
 this.argumento = argumento
}
```

```
Person.prototype.exemploPrototypeCreate = function(){ ... }
```

```
function Professor(argumentos){
 Person.call(this, argumentosHerdados);

 this.argumentos = argumentos;
} // herdando métodos e propriedades
```

```
Professor.prototype = Object.create(Person.prototype); // herdando o
prototype
```

## 5.4 Objeto Map

Contém pares chaves-valor. Ex:

```
const map1 = new Map();

map1.set('chav1', 'valor1'); // inserindo chaves e valores
map1.set('chav2', 'valor2');
map1.set('chav3', 'valor3');

console.log(map1.get('chav1')); // acessando a chave
```

formato de inserção: **set(chave, valor)**.

Itera sobre os elementos na ordem de inserção. **for...of** retorna um array chave-valor para cada iteração.

### Pontos positivos de utilização:

- Não apresenta conflitos com chaves padrões

- as chaves podem assumir qualquer valor (function, object ou tipo primitivo)
- chaves ordenadas de forma simples
- número determinado facilmente pela propriedade size
- performa melhor em adições e remoções frequentes

### Propriedades estáticas:

- get Map - função do construtor usada para criar a partir dos objetos

### Propriedades da instância:

- Map.prototype.size - retorna pares chave-valor do objeto map

### Métodos da instância:

- Map.prototype.clear() - remove todos os pares chave/valor do objeto

```
console.log(`valores apagados: ${map1.clear()}`);
```

- Map.prototype.delete(key) - retorna true se elemento existe e foi removido. Remove elemento específico do objeto.

```
console.log(`valores apagados: ${map1.delete('chav1')}`);
```

- Map.prototype.get(key) - retorna valor associado a chave, se não existe retorna undefined.

```
console.log(map1.get('chav1')); // acessando a chave
```

- Map.prototype.has(key) - retorna uma asserção se o valor tenha sido associado a chave ou não.

```
console.log(map1.has('chav1')); - resposta pode ser true ou false
```

- Map.prototype.set() - adiciona ou atualiza um elemento com a chave especificada e um valor do objeto

```
const map1 = new Map();

map1.set('nome', 'Diogo');

console.log(map1.get('nome'));
```

### Métodos iterativos:

- Map.prototype[@@iterator]() - retorna objeto iterator que contém um array[chave, valor] para cada elemento.

```
map1.set('nome', 'Diogo');
map1.set('sobrenome', 'Mello');
map1.set('idade', 21);
map1.set('hobbys', ['treinar', 'jogar']);

const iterator = map1[Symbol.iterator]();

for(const item of iterator){
 console.log(item);
}
```

Uma outra forma seria utilizar Map.prototype.entries()

```
const map1 = new Map();

map1.set('nome', 'Diogo');
map1.set('sobrenome', 'Mello');
map1.set('idade', 21);
map1.set('hobbys', ['treinar', 'jogar']);

const iterator = map1.entries();

console.log(iterator);
```

- Map.prototype.keys() - retorna um objeto que contém as chaves de cada elemento

```
const iterator = map1.keys();

console.log(iterator);
```

- Map.prototype.values() - retorna um objeto que contém os valores de cada elemento

```
const iterator = map1.values();

console.log(iterator);
```

## 6. Classes POO

São simplificações da linguagem para as heranças baseadas nos protótipos.

- criação de objetos mais simples.

- não introduz novo modelo de herança.

### tipos de classe:

- **expressão** → podem possuir nomes ou não.

#### Exemplo: sintaxe de uma classe expressa.

```
let quadrado = class {
 constructor (lado){
 this.lado = lado;
 }
}
```

- **declaração.**

#### Exemplo: sintaxe de uma classe declarada.

```
class Retangulo {
 constructor(altura, largura){
 this.altura = altura;
 this.largura = largura;
 }
}
```

Sempre usar a palavra chave class seguido pelo nome da classe.

**Nota: diferente das funções que tem suas declarações hoisted, as classes primeiro devem ser criadas para só então acessá-las.**

Os corpos das declarações e expressões de classes são executados em modo estrito.

- **strict mode** → variante restrita do javascript.
- **benefícios**: elimina possíveis erros js lançando exceções, otimização de código.
- Se aplica a scripts inteiros ou funções individuais, por exemplo: **eval**, **function**, **atributos de evento**, **strings passadas para setTimeout**.
- para invocar o modo coloque use strict antes de qualquer declaração.

#### Exemplo: para função.

```
function pessoa(){
 "use strict";
 console.log('teste');
}
```

```
function pessoa(){
 "use strict";
 function dados(){
 console.log('testando');
 }
 console.log('teste');
 dados();
}
```

A classe deve iniciar com um objeto que é **criado a partir de um construtor (tipo especial de método)**.

- só pode haver um construtor para cada classe.
- para herdar de pai o construtor pode usar a palavra **super**.

**Exemplo:**

```
class Retangulo {
 constructor(altura, largura){
 this.altura = altura;
 this.largura = largura;
 }
}
```

```
class Retangulo {
 constructor(altura, largura){
 this.altura = altura;
 this.largura = largura;
 }

 get area() {
 return this.calculaArea();
 }

 calculaArea() {
 return this.altura * this.largura;
 }
}
```

**Static** → **define métodos estáticos que podem ser chamados sem instancição da sua classe** e não pode ser chamado quando a classe é instanciada.

**exemplo: usando método estático.**

```
class Retangulo {
 constructor(altura, largura){
 this.altura = altura;
 this.largura = largura;
 }

 get area() {
 return this.calculaArea();
 }

 static calculaArea() {
 return this.altura * this.largura;
 }
}
```

```
console.log(Retangulo.calculaArea(retangulo));
```

Chamando o método aqui.

**OBS: se método estático ou protótipo for chamado sem this configurado, o valor de this será undefined.**

Exemplo:

```
class Retangulo {
 area() {
 return this;
 }

 static calculaArea() {
 return this;
 }
}
const retangulo = new Retangulo(5, 10);
const ret = retangulo.area;
const cal = Retangulo.calculaArea;
```

This não configurado e valores sendo passados a variáveis que serão undefined ao serem chamadas.

**Propriedades de instâncias** nada mais são que os valores colocados dentro do construtor e sempre devem ser criados dentro das classes.

- **propriedades estáticas e prototipadas devem ser definidas fora da classe.**

**Extends** - usada em declaração ou expressão de classe para criar uma classe filha de outra.

### Exemplo:

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

 falar() {
 console.log(this.nome + ' emite um barulho');
 }
}

class Cachorro extends Animal {
 falar () {
 console.log(this.nome + ' latidos');
 }
}
```

A classe filha está utilizando o “nome” que foi definido dentro da classe pai.

**OBS:** se subclasse possuir constructor, é importante chamar super() antes de usar this.

### Exemplo: outro modo de usar extends com funções.

```
function Animal (nome) {
 this.nome = nome;
}

Animal.prototype.falar = function() {
 console.log(this.nome + ' faça barulho');
}

class Cachorro extends Animal {
 falar () {
 console.log(this.nome + ' latidos');
 }
}

let cachorro = new Cachorro('Mat');
cachorro.falar();
```

Função animal com método falar criado em seu prototype.

### Exemplo: herdando de objetos.



```
let Animal = {
 falar() {
 console.log(this.nome + ' faça barulho');
 }
}

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

Object.setPrototypeOf(Cachorro.prototype, Animal);
```

A herança ocorre por meio da utilização de `setPrototypeOf()`.

`setPrototypeOf()` - configura o proto de um objeto específico para outro.

**Species** - permite sobrescrita do construtor padrão.

**exemplo: Retornando objeto Array.**

```
class MinhaArray extends Array {
 static get [Symbol.species]() {return Array;}
}

let a = new MinhaArray(1, 2, 3);
let mapped = a.map(x => x * x);

console.log(mapped instanceof MinhaArray);
console.log(mapped instanceof Array);
```

`mapped` está instanciado no objeto `Array`.

**Palavra-chave super** - usada para chamar funções que pertencem ao pai do objeto.

**Exemplo:**

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

 falar() {
 console.log(this.nome + ' faça barulho');
 }
}

class Leao extends Gato {
 falar() {
 super.falar();
 console.log(this.nome + " roars");
 }
}
```

Super está chamando a função falar() da classe pai Gato que acaba a ser executada antes da função falar da classe Leao.

**mix-ins (Subclasses)** - templates para classes.

- cria comportamento similar a uma herança múltipla.
- **função que instancia uma classe base** e retorna subclasse estendida da base.

**Exemplo:**

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

 andar() {
 return this.nome + ' andou um passo';
 }
}

const HumanoFalante = Base => class extends Base {
 falar() {
 return this.nome + ' dis: hola';
 }
}
```

Criando função que instancia classe Base.

```
const HumanoFalante = Base => class extends Base {
 falar() {
 return this.nome + ' dis: hola';
 }
}

const HumanoFalanteMixado = Base => class extends Base {}

const HumanoFinal = HumanoFalanteMixado(HumanoFalante(Humano));
```

Retornando à subclasse.

## 6.0 Getter / Setter

### 6.0.0 Getter

**sintaxe get** - associa propriedade de objeto a uma função que será chamada quando tal propriedade é chamada.

**sintaxe:** { get prop() { ... } }.

**{ get [expression] () { ... } }.**

- prop - nome da propriedade.
- expression - expressão para nome computado de uma propriedade.

**Exemplo: sintaxe de expressão.**

```
const obj = {
 get latest () {
 if (log.length == 0) return undefined;

 return log[log.length - 1];
 },

 get [nomePerson]() {
 return nomePerson;
 }
}
```

**OBS: uma propriedade que está associada a um getter não pode possuir um valor.**

**Exemplo:**

```
class Pessoa {
 get Fala(){
 console.log('meu nome');
 }
}

const pessoa = new Pessoa();

pessoa.Fala;
```

**características da sintaxe:**

- identificador pode ser número ou string.
- sem parâmetros.
- não pode haver mais de um getter para a mesma propriedade.
- Nenhuma propriedade pode ter o mesmo nome do getter.

**Nota: operador delete pode ser usado para remover um getter.**

**Exemplo:**

```
let log = ['test'];

const obj = {
 get latest () {
 if (log.length == 0) return undefined;

 return log[log.length - 1];
 }
}

console.log(obj.latest);
```

removendo com delete.

```
delete obj.latest;
```

**Exemplo: adicionando getter a um objeto a qualquer momento.**

```
let obj2 = { number: 1 };

Object.defineProperty(obj2, "Propriedade", { get: function () { return this.number
+ 1 } });

console.log(obj2.Propriedade);
```

**Técnica de otimização smart** (self-overwriting / lazy getters)- valor de getter fica cacheado para que acessos subsequentes retornem o valor em cache.

- **situações aplicadas:** propriedades que usam muita RAM / CPU, valor não necessário no momento e reutilização de valor.

### 6.0.1 Setter

Setter liga propriedade de função para ser **chamada quando existe tentativa de alteração ou definição de valor da propriedade.**

**Exemplo:**

```
class Pessoa{
 constructor(nome, sobrenome, idade, ano){
 this.nome = nome;
 this.sobrenome = sobrenome;
 this.idade = idade;
 this.ano = ano
 }
}
```

criado os parâmetros da classe.

```
set AnoNascimento(value){
 let anoAtual = value;
 this.ano = anoAtual - this.idade;
}

get AnoNascimento(){
 return this.ano;
}
```

Set recebe o ano atual e modifica o valor do this.ano. Get está retornando o novo valor do ano de nascimento.

sintaxe:

- `{ set prop(valeu) { ... } }.`
- `{ set [ expression ](value) { ... } }.`

**Prop** - propriedade ligada a função e **value** - variável que será atribuída a prop.

**Nota:** setters normalmente é usado com getters para criar uma pseudo-propriedade.

**OBS:** setters não pode ser usado para propriedade que possui valor real.

características da sintaxe:

- identificador pode ser número ou string.
- só pode possuir um parâmetro.
- não deve possuir a mesma nomenclatura.

**Nota:** operador delete e defineProperty também pode ser usado da mesma forma que em Getters.

## 7. JS Assíncrono

### 7.0 Promises

Representa a conclusão ou falha de uma operação assíncrona e seu valor resultante.

Objeto retornado que podemos adicionar callbacks em vez de passar callbacks para funções.

Ex:

```
function sucesso(sucesso){
 console.log('deu certo')
}

function falha(falha){
 console.log('deu errado')
}

const promise = faça_alguma_coisa();
promise.then(sucesso, falha);
```

outro modo,

```
function sucesso(sucesso){
 console.log('deu certo')
}

function falha(falha){
 console.log('deu errado')
}

const promise = faça_alguma_coisa();
promise.then(sucesso, falha);
faça_alguma_coisa().then(sucesso, falha);
```

Esse tipo de convenção é chamada de função assíncrona.

### **vantagens:**

- callbacks chamados depois da conclusão da execução atual do loop de eventos.
- callbacks adicionais com **.then** serão chamados mesmo depois de sucesso ou falha. Isso pode ocorrer várias vezes independente da ordem de inserção.

- representa a completude de outro passo na cadeia.

Uma segunda promise representa a conclusão de uma primeira promise e de seus callbacks passados que também podem ser funções assíncronas que retornam uma promise.

Outra de criar uma promise que rode um possível erro seria com a utilização catch.

Ex:

```
doSomething().then(result => doSomethingElse(result)).catch(failureCallback);
```

```
function doSomething(){
 return new Promise((resolve, reject) => {
 console.log('inicial')
 resolve();
 })
}

doSomething().then(() => {
 console.log('do this');
 throw new Error('something failed');
}).catch(() => {
 console.log('do that');
});
```

Encadeamento após uma falha:

```
function doSomething(){
 return new Promise((resolve, reject) => {
 console.log('inicial')
 resolve();
 })
}

doSomething().then(() => {
 console.log('do this');
 throw new Error('something failed');
}).catch(() => {
 console.log('do that');
}).then(() => {
 console.log('estou tomando uma nova ação aqui');
});
```

uma nova ação é retomada após o catch.

### Estados de uma promise:

- pending - estado inicial.
- fulfilled - operação concluída.
- rejected - operação rejeitada.

**OBS: uma promise pode ser considerada resolvida em seus estados fulfilled e rejected.**

Métodos como **then()**, **catch()** e **finally()** são usados para ação adicional com uma promise que se torna liquidada.

### método:

- **then()** - aceita dois argumentos, a primeira seria um callback de sucesso e o segundo um callback de falha.
  - cada then retorna um objeto novo.
  - cadeia pode omitir cada rejeição até o catch() final.
- **catch()** - nada mais é um then() que não pode retornar um sucesso.

**OBS: É IMPORTANTE LANÇAR UM ERRO DE UM ALGUM TIPO PARA MANTER O ESTADO DE ERRO NA CADEIA.**



**A condição de término** determina o estado estabelecido para a próxima promise na cadeia.

**Thenable** - implemente then com dois retornos de chamadas, sucesso e rejeição.

formato:

```
const athenable = {
 then(sucesso, rejeicao){
 onCumprido({
 then(sucesso, rejeicao){
 onCumprido(42);
 }
 });
 },
};

Promise.resolve(athenable);
```

**Objeto de conf. incumbente** - garante que navegador saiba qual usar determinado pedaço de código de usuário.

- informação específica do contexto de código do user que determina chamada de função.

### Construtor:

- **Promise** - usado para encapsular funções que ainda não suportam promises.
  - Cria objetos que serão resolvidos quando funções forem invocadas.
  - Formato - **new Promise(executor);**
    - executor: recebe duas funções de retorno, **resolveFunc(value)** e **rejectFunc(reason)**. Nota: Value pode ser um objeto de promessa e Reason é normalmente um error instância.

**OBS: ambas as funções recebem apenas um parâmetro de qualquer tipo.**

Exemplo:

```
function doSomething(){
 return new Promise((resolve, reject) => {
 console.log('inicial')
 resolve();
 })
}
```

### Composição das promises:

- **promise.resolve(value)** - retorna objeto promise que é resolvido com o valor de retorno passado.
  - value - pode ser uma promise ou thenable

```
Promise.resolve("sucesso").then(function() {
 console.log("value");
}).then(function(){
 console.log('resolvendo aqui também');
});
```

outra forma,

```
const numeros = Promise.resolve([1, 2, 3, 4]);

numeros.then((v) => {
 let a = v[0] + v[3];
 console.log(a);
});
```

- **promise.reject(motivo)** - retorna objeto promise que é rejeitado por um dado motivo.
  - É importante que o motivo seja uma **instanceOf Error**.

```
Promise.reject(new Error("falha")).then((motivo) => {
 console.log("teste reject 1", motivo);
}).catch((motivo) =>{
 console.log("teste reject 2",motivo);
});
```

outra forma,

```
Promise.reject('Falha').then((motivo) => {
 console.log("teste reject 3", motivo);
}).catch((motivo) =>{
 console.log("teste reject 4",motivo);
});
```

- **promise.all(iterable)** - retorna a promise que resolve todas as promises no argumento iterável.
  - retornos: **promise resolvida** - se iterável vazio, **promise resolvida assincronamente** - se iterável não contar promises e **promise pendente** - resolvida/rejeitada assincronamente quando promises no iterável forem resolvidas ou se alguma for rejeitada. **valores se**

encontraram na ordem em que foram passados independente da ordem que forem concluídos.

- usado para agregar resultados a várias promises.
- **rejeição**: Se uma promise passada for rejeitada, `promise.all` assincronamente é rejeitada com o valor da promise rejeitada, independente se outras promises forem resolvidas.

Exemplo:

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
 setTimeout(resolve, 10000, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((value) => {
 console.log(value);
});
```

3 promises são criadas `promise1`, `promise2` e `promise3`. `Promise.all` tem o objetivo de resolver todas as promises que estão dentro do argumento iterável (array ou string) que no caso é um array.

**OBS: valores que não são promises são ignorados dentro do iterável, mas considerados dentro do array.**

- **assincronia**:

Exemplo:

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
 setTimeout(resolve, 10000, 'foo');
});

const promise = [promise1, promise2, promise3];

const promiseAll = Promise.all(promise);

console.log(promiseAll);

setTimeout(() => {
 console.log(promiseAll)
}, 10000);
```

a resposta em `promiseAll` fica pendente enquanto todas as promises não são resolvidas, por esse motivo no primeiro console nada será mostrado além de

pendências. Ao chegar no console dentro de setTimeout que esperou 10.000 ms, os valores das promises retornadas serão mostrados.

- **resolvendo erros:**

Exemplo:

```
const promise = [promise1.catch(e => {console.log(e)}), promise2, promise3];

const promiseAll = Promise.all(promise);

console.log(promiseAll);

setTimeout(() => {
 console.log(promiseAll)
}, 10000);
```

**Catch é usado para solucionar um possível erro que venha a ocorrer em alguma promise.** O erro é retornado imediatamente e posteriormente as promises resolvidas serão retornadas dentro do iterável.

- **Promise.race(iterable)** - retorna promise que resolve/rejeita assim que promise no iterable é resolvida/rejeitada, com valor do iterable (obs: array).

Exemplo:

```
const promise1 = Promise.reject(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
 setTimeout(resolve, 10000, 'foo');
});

const promise = [promise1.catch(e => {console.log(e)}), promise2, promise3];

const promiseRace = Promise.race(promise);

console.log(promiseAll);

setTimeout(() => {
 console.log(promiseRace)
}, 10000);
```

## 7.1 Async/Await

**Async function** declara função assíncrona em que a palavra `await` palavra-chave é permitida no corpo da função.

- comportamento assíncrono baseado em promessas mais limpo.
- espera uma promise.
- permite uso de blocos `try/catch`

**formato: `async function expressão`**

`async function (params) { ... } (1)` **expressão**

`async function name(params) { ... } (2)` **declaração**

**`async function expressão` podem omitir o nome para criar funções anônimas.**

- executada assim que definida
- permite imitação de `await`

Exemplo: `async`

```
function resolveAfterSeconds() {
 return new Promise(resolve => {
 setTimeout(() => {
 resolve('resolved');
 }, 4000);
 })
}
```

Cria uma promessa que será lançada em 4s.

```
async function asyncCall() {
 console.log('calling');
 const result = await resolveAfterSeconds();
 console.log(result);
}

asyncCall();
```

**Função assíncrona que vai esperar a promessa.**

**Expressões `Await`** - suspende execução até que promessa retornada seja cumprida ou rejeitada.

- faz com que funções de retorno de promessa se comportem como assíncronas.
- valor resolvido - tratado como valor de retorno da expressão `await`.

**Nota: `await` pode ser usado sozinho com módulos JS.**

**Async/await tem como objetivo** simplificar a sintaxe necessária para consumir APIs baseadas em promessas.

O código é executado de maneira síncrona até que haja uma expressão de espera, mesmo sendo uma função assíncrona.

exemplo: funções assíncronas semelhantes

```
async function foo(){
 await 1;
 ...
}

function fooI(){
 return Promise.resolve(1).then(() => undefined);
}
```

Código após cada await é considerado como existente em um .then retorno de chamada, tendo o valor de retorno como elo final da cadeia.

**sequência de código:** (antes await) código síncrono em f1 → (await) controle vai para função de chamada f2 → (quando resolvida) volta para f1 → resposta em await.

## 8. JS Tooling

### 8.0 Import

Usado para importar vínculos que são exportados por outros módulos.

**OBS: DECLARAÇÃO IMPORT NÃO PODE SER USADA EM SCRIPTS EMBUTIDOS, A MENOS QUE TENHAM TYPE="MODULE".**

- função dinâmica **import()** não requer type="module".

**Nomodule** - garante compatibilidade com versões anteriores. Aplicar na tag script.

**tipos de importações:**

- **dinâmica** - situações para carregamento de módulo condicional ou sob demanda.
- **estática** - serve para carregar dependências iniciais.

### tipos de sintaxe:

1. `import defaultExport from "module-name";`
2. `import * as name from "module-name";`
3. `import { export } from "module-name";`
4. `import { export as name } from "module-name";`
5. `import { export1, export2 } from "module-name";`
6. `import defaultExport, * as name from "module-name";`
7. `import "module-name";`
8. `let promise = import("module-name");`

**defaultExport** - nome de referência para exportação padrão do módulo.

**name** - nome para se referir ao objeto do módulo. Namespace que se refere as exportações.

**"\*"** - importa todas as exportações.

**Acessando exportações do módulo** - utilizar como namespace o nome do módulo e depois a sua exportação.

### Exemplo: acessando conteúdo de um módulo.

```
import { documentos } from "../modulo1";
console.log(documentos.name);
```

### Exemplo: importando única exportação de módulo.

```
import { documentos } from "../modulo1";
```

### Exemplo: renomeando uma importação.

```
import { documentos as doc } from './modulo1';
console.log(doc.name);
```

Isso também é possível para várias importações ao mesmo tempo, basta separar cada importação por virgula.

**Exemplo: importando módulo sem nenhum valor.**

```
import './modulo1';
```

Isso executa código global do módulo.

**Nota: import pode ser uma função para importar dinamicamente um módulo.**

**Exemplo:**

```
import ('./modulo1').then((module) => {
 // ...
})
```

## 8.1 Export

**Utilizado para exportar ligações em tempo real** para funções, objetos ou valores primitivos de um módulo que será importado por outro programa.

- ligações que são exportadas ainda podem ser modificadas localmente.
- **somente lidas pelo módulo que importou.**
- **obs: não pode ser usado em script embutido.**

**Tipos de exportações:**

- **explícita (named exports)** - pode haver mais de uma por módulo.
  - É importante usar o mesmo nome de objeto correspondente.

**exemplo:**

```
export { documentos };

export class Pessoa {
 constructor (nome, sobrenome){
 this.nome = nome;
 this.sobrenome = sobrenome;
 }
}
```



- **padrão (default exports)** - só pode haver uma por módulo.
  - pode usar qualquer nome para importação.

**exemplo:**

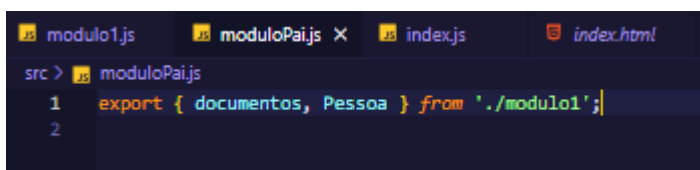
```
export default function soma(x, y){
 return x + y;
}
```

**sintaxe:**

- **recursos individuais**
  - `export (let/const/var) name1, n2, n...`
  - `export (let/const/var) name = ...`
  - `export function name() { ... }`
  - `export class name { ... }`
- **lista de exportações**
  - `export {n1 ... nN}`
- **Renomeando exports**
  - `export { variable as n1 ... nN }`
- **atribuições desestruturadas**
  - `export (let/const/var) { name1, name2: bar } = object;`
- **padrão**
  - `export default expression/function/class`
- **agregando módulos**
  - `export * from ...`

**Agregando** - significa que um único módulo irá concentrar várias exportações de vários módulos.

**Exemplo:**



```
src > moduloPai.js
1 export { documentos, Pessoa } from './modulo1';
2
```

Módulo pai recebendo módulos do filho e exportando.

```
import { documentos } from "../moduloPai";

console.log(documentos.name);
```

Módulo pai sendo importado dentro do código.

**Nota:** módulos podem ser inseridos no script html através do elemento **<script>** do tipo="module".

- módulos JS devem rodar em servidores HTTP.