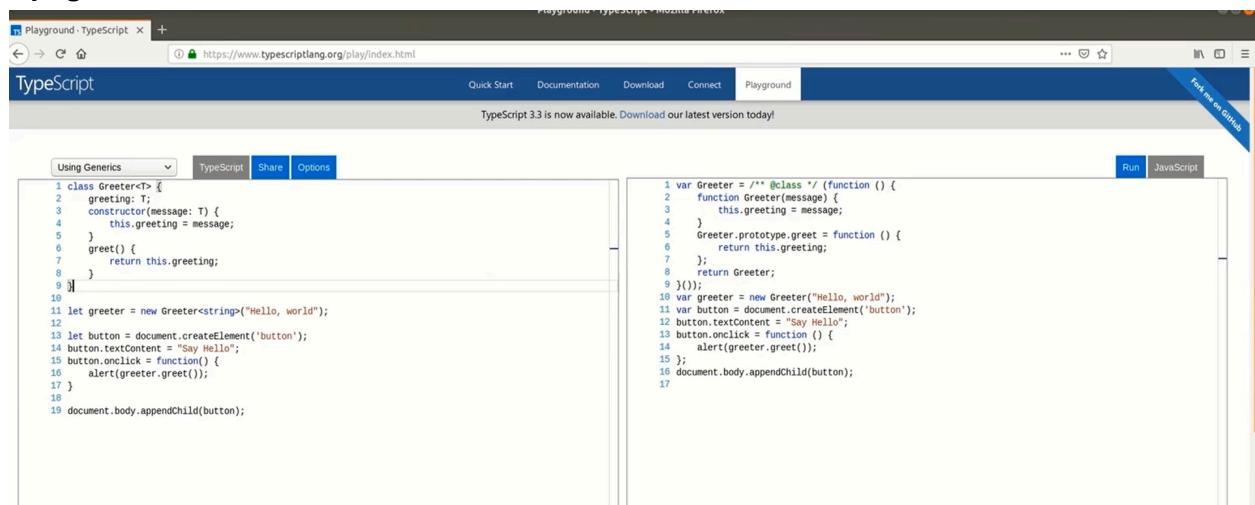


Estudo Javascript

```
conceitos > tipagem-fraca > 1-tipagem-fraca.js
1 var meuNumero = 20;
2 var meuTexto = 'Exemplo';
3
4 console.log(meuNumero + meuTexto);
```

Javascript é uma linguagem de tipagem fraca, por exemplo no javascript pode usar +, com * e string, no python, uma linguagem de tipagem forte, já não pode

Tipagem Dinâmica:



The screenshot shows the TypeScript playground interface. On the left, there is a code editor window titled "Using Generics" containing the following TypeScript code:

```
1 class Greeter<T> {
2   greeting: T;
3   constructor(message: T) {
4     this.greeting = message;
5   }
6   greet() {
7     return this.greeting;
8   }
9 }
```

On the right, there is another code editor window titled "Run" (JavaScript) containing the following JavaScript code:

```
1 var Greeter = /** @class */ (function () {
2   function Greeter(message) {
3     this.greeting = message;
4   }
5   Greeter.prototype.greet = function () {
6     return this.greeting;
7   };
8   return Greeter;
9 }());
10 var greeter = new Greeter("Hello, world");
11 var button = document.createElement('button');
12 button.textContent = "Say Hello";
13 button.onclick = function () {
14   alert(greeter.greet());
15 };
16 document.body.appendChild(button);
17
18
19
```

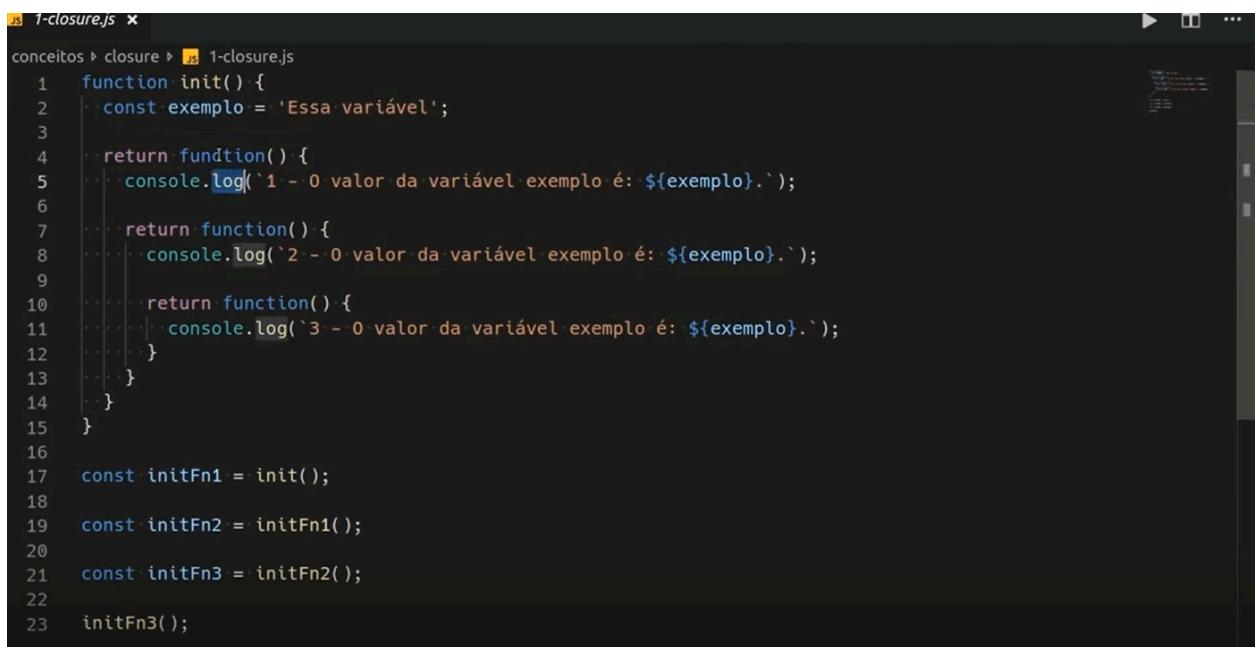
A principal diferença é que o TypeScript é um superconjunto do JavaScript que adiciona tipagem estática opcional, enquanto o JavaScript é tipado dinamicamente. A tipagem estática do TypeScript permite a detecção de erros em tempo de compilação, tornando o código mais robusto e fácil de manter, especialmente em projetos grandes. O código TypeScript é posteriormente transcompilado para JavaScript para ser executado no navegador.

```

1  function getName() {
2    return 'Guilherme Cabrini da Silva';
3  }
4
5  function logFn(fn) {
6    console.log(fn());
7  }
8
9  const logFnResult = logFn;
10
11 logFnResult(getName);
12

```

Função Para retornar nome



```

js 1-closure.js x
conceitos > closure > 1-closure.js
1  function init() {
2    const exemplo = 'Essa variável';
3
4    return function() {
5      console.log(`1 - O valor da variável exemplo é: ${exemplo}`);
6
7      return function() {
8        console.log(`2 - O valor da variável exemplo é: ${exemplo}`);
9
10     return function() {
11       console.log(`3 - O valor da variável exemplo é: ${exemplo}`);
12     }
13   }
14 }
15
16 const initFn1 = init();
17
18 const initFn2 = initFn1();
19
20 const initFn3 = initFn2();
21
22 initFn3();

```

Exemplo closure

Currying

Técnica de transformar função de vários parâmetros, em apenas uma de um parâmetro

Para cada parâmetro, se cria uma nova função

Hoisting

Significa levantar, as funções são levadas ao escopo, pode ser dividido em 2 partes, variáveis e escopo(função)



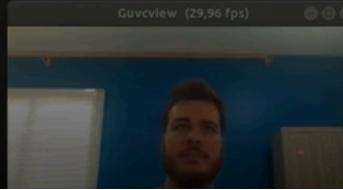
```
1 hoisting-variaveis.js x
conceitos > hoisting > 1-hoisting-variaveis.js
1 function fn() {
2   console.log(text);
3   I
4   var text = 'Exemplo';
5
6   console.log(text);
7 }
8
9 fn();
10
11 /**
12  * function fn() {
13  *   var text;
14  *   console.log(text);
15  *   text = 'Exemplo';
16  *   console.log(text);
17  * }
18 */
19
20 */

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Code
[Running] node "/home/cabrini/Desktop/exemplos/conceitos/currying/tempCodeRunnerFile.is"
4
5
6
7
[Done] exited with code=0 in 0.06 seconds
```

The screenshot shows a code editor window for a file named '1-hoisting-variaveis.js'. The code demonstrates function hoisting. It contains two definitions of the 'fn' function: one at the top and another inside a block comment. Both definitions log the value of the variable 'text' to the console. The output window shows the logs '4', '5', '6', and '7' from the first definition, followed by '[Done] exited with code=0 in 0.06 seconds'. To the right of the code editor, there is a video call interface showing a man with a beard speaking.

A Diferença entre hoisting de função e variável, a função é içada como um todo

Imutabilidade: Conceito de linguagem funcional presente no javascript, em linguagens funcionais, as variáveis nunca vão mudar, um objeto nunca é atualizado, deve ser copiado e alterado apenas o que for necessário



```
1-imutabilidade.js ●
conceitos > imutabilidade > 1-imutabilidade.js
1  const user = {
2    name: 'Guilherme',
3    lastName: 'Cabrini da Silva'
4  };
5
6  function getUserWithFullName(user) {
7    return {
8      ...user,
9      fullName: `${user.name} ${user.lastName}`
10   }
11 }
12
13 const userWithFullName = getUserWithFullName(user);
14
15 console.log(userWithFullName, user);
16
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Code

```
{ name: 'Guilherme',
  lastName: 'Cabrini da Silva',
  fullName: 'Guilherme Cabrini da Silva' } { name: 'Guilherme', lastName: 'Cabrini da Silva' }

[Done] exited with code=0 in 0.081 seconds

[Running] node "/home/cabrini/Desktop/exemplos/conceitos/imutabilidade/tempCodeRunnerFile.js"
/home/cabrini/Desktop/exemplos/conceitos/imutabilidade/tempCodeRunnerFile.js:3
};

SyntaxError: Unexpected token }
    at createScript (vm.js:80:10)
    at Object.runInThisContext (vm.js:139:10)
    at Module._compile (module.js:617:28)
```

Neste exemplo, um novo objeto com fullname foi criado, mas sua referência não foi alterada

Tipos e variáveis

Var

Let

Const

No javascript, sempre dar o nome a variável e seu valor

```
1 // escopo global
2
3 {
4     // escopo bloco
5 }
6
7 function test() {
8     // escopo de função
9 }
```

I

O Var não respeita o escopo de bloco, let e const respeitam

```
var test = 'example';

(() => {
    console.log(`Valor dentro da função ${test}`);

    if (true) {
        var test = 'example';
        console.log(`Valor dentro do if ${test}`);
    }

    console.log(`Valor após a execução do if ${test}`);
})();
```

O Escopo é global, a variável foi criada dentro dele

O var não aceita escopo de bloco, na imagem abaixo, a variável foi declarada no global e seu valor foi definido no escopo de bloco

```
ar test = 'example';

() => {
  var test;
  console.log(`Valor dentro da função "${test}"`);

  if (true) {
    test = 'example';
    console.log(`Valor dentro do if "${test}"`);
  }

  console.log(`Valor após a execução do if "${test}"`);
}();
```

```
((() => {
  const test = 'valor função';
  console.log(`Valor dentro da função "${test}"`);

  if (true) {
    const test = 'valor if';
    console.log(`Valor dentro do if "${test}"`);
  }
  I
  console.log(`Valor após a execução do if "${test}"`);
}))();
```

Exemplo do let

O Const é para criar constantes, se criar strings, ele não permite trocar os valores, mas se for objeto, ele permite alterar o valor

```
const user = {  
  name: 'Guilherme'  
};  
  
// Mas se for um objeto, podemos trocar suas propriedades  
user.name = 'Outro nome';  
  
// Não podemos fazer o objeto "apontar" para outro lugar  
user = {  
  name: 'Guilherme'  
};  
  
const persons = ['Guilherme', 'Pedro', 'Jennifer'];  
  
// Podemos adicionar novos itens  
persons.push('João');  
  
// Podemos remover algum item  
persons.shift();
```

```
// ...name: 'Guilherme'  
// ...};  
  
const persons = ['Guilherme', 'Pedro', 'Jennifer'];  
  
// Podemos adicionar novos itens  
persons.push('João');  
// ['Guilherme', 'Pedro', 'Jennifer', 'João']  
  
// Podemos remover algum item  
persons.shift();  
// ['Pedro', 'Jennifer', 'João']  
  
// Podemos alterar diretamente  
persons[1] = 'James';  
// ['Pedro', 'James', 'João']  
  
console.log('\nArray após as alterações: ', persons);
```

String: Length(Fala quantidade de caracteres na string)

Split: quebra a string

Replace: Troca determinada parte do texto

Slice: Retorna a “fatia de um valor”

```

4 // Retorna um array quebrando a string por um delimitador
5 const splittedText = 'Texto'.split('x');
6 console.log('\nArray com as posições separadas pelo delimitador:', splittedText);
7
8 // Busca por um valor e substitui por outro
9 const replacedText = 'Texto'.replace('Text', 'txeT');
10 console.log('\nSubstituição de valor:', replacedText);
11
12 // Retorna a "fatiada" de um valor
13 const lastChar = 'Texto'.slice(-1);
14 console.log('\nÚltima letra de uma string:', lastChar);
15
16 const allWithoutLastChar = 'Texto'.slice(0, -1);
17 console.log('\nValor da string da primeira letra menos a última:', allWithoutLastChar);
18
19 const secondToEnd = 'Texto'.slice(1);
20 console.log('\nValor da string da segunda letra até a última:', secondToEnd);
21
22 // Retorna N caracteres a partir de uma posição
23 const twoCharsBeforeFirstPos = 'Texto'.substr(0, 2);
24 console.log('\nAs duas primeiras letras são:', twoCharsBeforeFirstPos);
25
26

```

```

const myNumber = 12.4032;

// Transformar número para string
const numberAsString = myNumber.toString();
console.log('Número transformado em string:', typeof numberAsString);

// Retorna número com um número de casas decimais
const fixedTwoDecimals = myNumber.toFixed(2);
console.log('\nNúmero com duas casas decimais:', fixedTwoDecimals);

// Transforma uma string em float
console.log('\nString parseada para float:', parseFloat('13.22'));

// Transforma uma string em int
console.log('\nString parseada para int:', parseInt('13.20'));

```

Typeof: Retorna o tipo da variável

toFixed(Retorna a quantidade de casas decimais)

parseInt(Não usa decimais)

parseFloat(usa decimais)

Undefined(Variável indefinida)

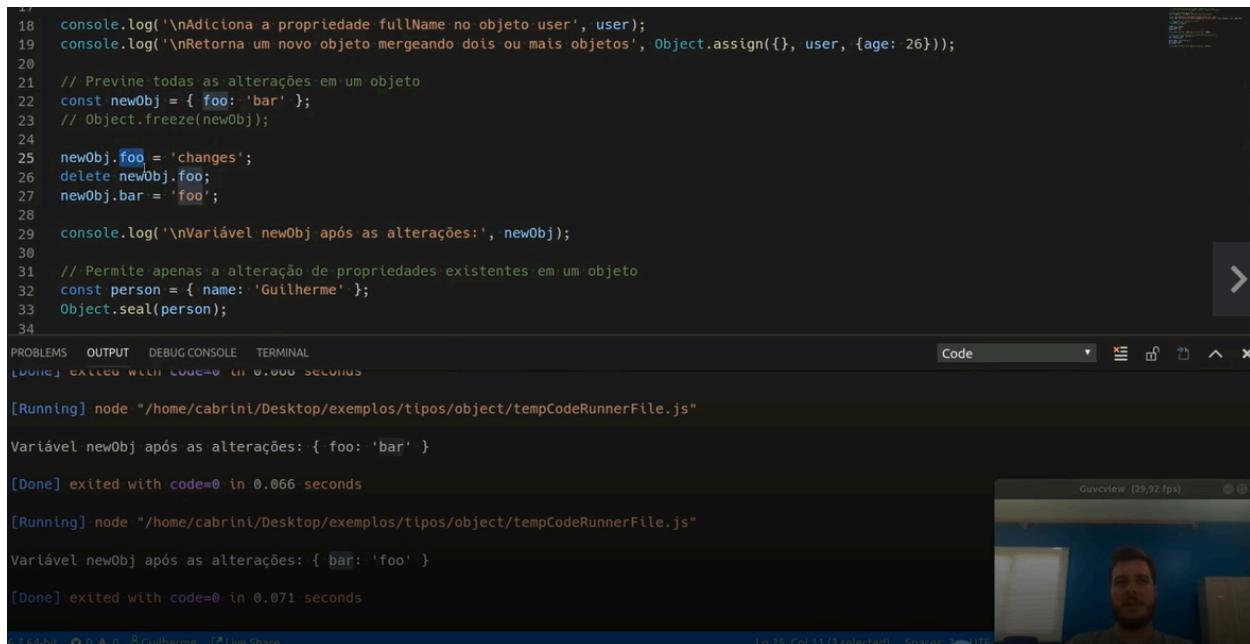
Boolean(Dois valores, geralmente true ou false)

```

5 // Recupera as chaves do objeto
6 console.log('Propriedades do objeto user:', Object.keys(user));
7
8 // Recupera os valores das chaves do objeto
9 console.log('\nValores das propriedades do objeto user:', Object.values(user));
10
11 // Retorna um array de arrays contendo [ nome_prop, valor_prop ]
12 console.log('\nLista de propriedades e valores:', Object.entries(user));
13
14 // Mergear propriedades de objetos
15 Object.assign(user, {fullName: 'Guilherme Cabrini da Silva'});
16
17 console.log('\nAdiciona a propriedade fullName no objeto user', user);
18
19 console.log('\nRetorna um novo objeto mergeando dois ou mais objetos', Object.assign({}, user, {age: 26}));

```

Objetc.assign(recebe um objeto por parametro e aceita n outros objetos)



```

18 console.log('\nAdiciona a propriedade fullName no objeto user', user);
19 console.log('\nRetorna um novo objeto mergeando dois ou mais objetos', Object.assign({}, user, {age: 26}));
20
21 // Previne todas as alterações em um objeto
22 const newObj = { foo: 'bar' };
23 // Object.freeze(newObj);
24
25 newObj.foo = 'changes';
26 delete newObj.foo;
27 newObj.bar = 'foo';
28
29 console.log('\nVariável newObj após as alterações:', newObj);
30
31 // Permite apenas a alteração de propriedades existentes em um objeto
32 const person = { name: 'Guilherme' };
33 Object.seal(person);
34

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[Done] exited with code=0 in 0.000 seconds

[Running] node "/home/cabrini/Desktop/exemplos/tipos/object/tempCodeRunnerFile.js"

Variável newObj após as alterações: { foo: 'bar' }

[Done] exited with code=0 in 0.066 seconds

[Running] node "/home/cabrini/Desktop/exemplos/tipos/object/tempCodeRunnerFile.js"

Variável newObj após as alterações: { bar: 'foo' }

[Done] exited with code=0 in 0.071 seconds

Guvview (29.92 fps)

A Propriedade freeze permite alteração e remoção da propriedade, mas após ser colocado, não permite criar remover ou alterar propriedades
Seal, não permite criar ou remover, mas permite alterar

```

const symbol1 = Symbol();
const symbol2 = Symbol();

// Symbols são únicos
console.log('symbol1 é igual a symbol2:', symbol1 === symbol2); I

// Prevenir conflito entre nomes de propriedades
const nameSymbol1 = Symbol('name');
const nameSymbol2 = Symbol('name');

const user = {
  [nameSymbol1]: 'Guilherme',
  [nameSymbol2]: 'Outro nome',
  lastName: 'Cabrini da Silva'
}

console.log(user);

// Symbols criam propriedades que não são enumeráveis
for (const key in user) {
  if (user.hasOwnProperty(key)) {
    console.log(`\nValor da chave ${key}: ${user[key]}`);
  }
}

pos ✕ functions ✕ 1-function.js
1  function fn() {
2    | return 'Code here';
3  }
4
5  const arrowFn = () => 'Code here';
6  const arrowFn2 = () => {
7    // Mais de uma expressão
8    | return 'Code here';           I
9  }
10
11 // Funções também são objetos
12 fn.prop = 'Posso criar propriedades';
13
14 console.log(fn());
15 console.log(fn.prop);
16
17 // Receber parâmetros
18 const logValue = value => console.log(value);
19 const logFnResult = fnParam => console.log(fnParam());
20
21 logFnResult(fn);

```

Nesse código, nenhuma função recebe parâmetro, functions e arrays também são objetos, função são objetos que podem ser chamados, executados

```

// Receber parâmetros
const logValue = value => console.log(value);
const logFnResult = fnParam => console.log(fnParam());

logFnResult(fn);

// Receber e retornar funções
const controlFnExec => fnParam => allowed => {
  if (allowed) {
    fnParam();
  }
}

const handleFnExecution = controlFnExec(fn);
|_
handleFnExecution(true); // Executará a função fn
handleFnExecution(); // Não executará a função fn

// controlFnExec como função
function controlFnExec(fnParam) {
  return function(allowed) {
    if (allowed) {

```

fnParam: Função de parâmetro

Diferença function e arrowfunction, arrow é mais enxuto que function, quando se cria ela, na imagem abaixo, sempre a arrow apontará para o this.name, se fosse function, ela sempre iria apontar para o contexto na qual ela foi executada

```
1  (( )=> {
2    this.name = 'arrow function';
3    const getNameArrowFn = ()=> this.name;
4
5    function getName(){
6      return this.name;
7    }
8
9    const user = {
10      name: 'Nome no objeto de execução',
11      getNameArrowFn,
12      getName
13    }
14
15    console.log(user.getNameArrowFn());
16    console.log(user.getName());
17  })();
18
```

O Arrowfn o this não vai mudar, ele vai apontar para onde sempre esteve, na function, ela iria referenciar o const user e retornar a propriedade name: 'nome no objeto de execução';

O Atributo lenght permite que retorne a quantidade de itens

```
  gender: gender.WOMAN
};

// Retornar a quantidade de itens de um array
console.log('Items:', persons.length);

// Verificar se é array
console.log('A variável persons é um array:', Array.isArray(persons));

// Iterar os itens do array
persons.forEach(person => {
  console.log(`Nome: ${person.name}`);
});

// Filtrar array
const mens = persons.filter(person => person.gender === gender.MAN);
console.log(`\nNova lista apenas com homens:`, mens);

// Retornar um novo
const personsWithCourse = persons.map(person => {
  person.course = 'Introdução ao Javascript';
});
```

Em JavaScript, um Array é uma estrutura de dados que funciona como uma lista ordenada de valores, permitindo armazenar múltiplos itens (números, strings, objetos, etc.) em uma única variável, acessando cada um por um índice numérico (começando do 0). Arrays são fundamentais para agrupar e manipular coleções de dados,

possuindo métodos nativos como `push` para adicionar e `pop` para remover itens, facilitando a manipulação de listas.

```
// Verificar se é array
console.log('A variável persons é um array:', Array.isArray(persons));

// Iterar os itens do array
persons.forEach((person, index, arr) => {
  console.log(`Nome: ${person.name}`);
});

// Filtrar array
const mens = persons.filter(person => person.gender === gender.MAN);
console.log('\nNova lista apenas com homens:', mens);

// Retornar um novo
const personsWithCourse = persons.map(person => {
  person.course = 'Introdução ao Javascript';
  return person;
});

console.log('\nPessoas com a adição do course:', personsWithCourse);
```

Filter é uma condicional, na função //filter array retorna apenas os homens

O Map permite que retorne um novo array

A Função reduce permite que transforme o array em outro tipo

```
// Filtrar array
const mens = persons.filter(person => person.gender === gender.MAN);
console.log('\nNova lista apenas com homens:', mens);

// Retornar um novo
const personsWithCourse = persons.map(person => {
  person.course = 'Introdução ao Javascript';
  return person;
});

console.log('\nPessoas com a adição do course:', personsWithCourse);

// Transformar um array em outro tipo
const totalAge = persons.reduce((age, person) => {
  age += person.age;
  return age;
}, 0);

console.log('\nSoma de idade das pessoas', totalAge);
```

No código, o reduce possibilita retornar a soma da idade das pessoas, na função a idade de cada pessoa é somada para determinar a soma da idade de todas as pessoas, que é um array, a idade é transforma em number, a soma da idade de todo mundo

```
48
49 // Transformar um array em outro tipo
50 const totalAge = persons.reduce((age, person) => {
51   age += person.age;
52   return age;
53 }, 0);
54
55 console.log(`\nSoma de idade das pessoas`, totalAge);
56
57 // Juntando operações
58 const totalEvenAges = persons
59   .filter(person => person.age % 2 === 0)
60   .reduce((age, person) => {
61     age += person.age;
62     return age;
63   }, 0);
64
65 console.log(`\nSoma de idades das pessoas que possuem idade par`, totalEvenAges);
```

A função soma a idade das pessoas com idades pares

OPERADORES JAVASCRIPT



Operadores

- ✓ Aritméticos
- ✓ Atribuição
- ✓ Comparação
- ✓ Condicional
- ✓ Lógicos
- ✓ Spread

Operador binário recebe dois operandos, como por exemplo $1+2$

Operador unário só possui um operando, que pode vir depois ou antes do operador

```
// Operador binário  
operando1 operador operando2  
  
1 + 2  
  
// Operador unário  
operando1 operador  
operador operando1  
  
x++  
++x
```

Se o operador tiver na frente, o a já vai receber o valor incrementado, se o operador tiver depois, ele vai receber o valor atual

```
// Incremento (++) e Decremento (--)  
++x  
x++  
  
const a = ++2; // 3  
const b = 2++; // 2  
  
--x  
x--  
  
// Negação (-) e Adição (+)  
-(3 // retorna -3  
+true // retorna 1  
+false // retorna 0  
-true // retorna -1  
I  
// Operador de exponenciação (**)  
2 ** 3 // retorna 8  
10 ** -1 // retorna 0.1  
  
// Operador de agrupamento ()  
2 * (3 + 2)
```

```
1 // Atribuição de igualdade
2 x = y
3
4 // Atribuição de adição→
5 x = x + y // ou
6 x += y I
7
8
9 // Atribuição de subtração→
10 x = x - y // ou
11 x -= y
12
13
14 // Atribuição de multiplicação→
15 x = x * y // ou
16 x *= y
17
18 // Atribuição de divisão→
19 x = x / y // ou
20 x /= y
21
22 // Atribuição de resto
23 x = x % y // ou
24 x %= y
```

```

// Igual (==)
// Retorna verdadeiro caso os operandos sejam iguais.
3 == var1
3 == '3'

// Não igual (!=)
// Retorna verdadeiro caso os operandos não sejam iguais.
var1 != "3"

// Estrictamente igual (===)
// Retorna verdadeiro caso os operandos sejam iguais e do mesmo tipo. Veja também Object.is e igualdade
3 === var1

// Estrictamente não igual (!==)
// Retorna verdadeiro caso os operandos não sejam iguais e/ou não sejam do mesmo tipo.
var1 !== "3"
3 !== '3'

// Maior que (>)
// Retorna verdadeiro caso o operando da esquerda seja maior que o da direita.
var2 > var1
"12" > '2'

// Maior que ou igual (>=)
// Retorna verdadeiro caso o operando da esquerda seja maior ou igual ao da direita.
var2 >= var1
var1 >= 3

// Menor que (<)
// Retorna verdadeiro caso o operando da esquerda seja menor que o da direita.
var1 < var2
"12" < "2"

// Menor que ou igual (<=)

```



No exemplo a seguir, se a condição for verdadeira retorna o primeiro, se for falsa retorna o 2

```

Operadores > condicional > 1-condicional.js
1 // Ternário
2 condicao ? valor1 : valor2
3
4 true ? 'foo' : 'bar' ..... // Retorna 'foo'
5 false ? 'foo' : 'bar' ..... // Retorna 'bar'

```

```

// AND lógico (&&)
exp1 && exp2

var a1 = true && true; // t && t retorna true
var a2 = true && false; // t && f retorna false
var a3 = false && true; // f && t retorna false
var a4 = false && (3 == 4); // f && f retorna false
var a5 = "Gato" && "Cão"; // t && t retorna Cão
var a6 = false && "Gato"; // f && t retorna false
var a7 = "Gato" && false; // t && f retorna false

// OU lógico (||)
exp1 || exp2

var o1 = true || true; // t || t retorna true
var o2 = false || true; // f || t retorna true
var o3 = true || false; // t || f retorna true
var o4 = false || (3 == 4); // f || f retorna false
var o5 = "Gato" || "Cão"; // t || t retorna Gato
var o6 = false || "Gato"; // f || t retorna Gato
var o7 = "Gato" || false; // t || f retorna Gato

// NOT lógico (!)
!exp1

var n1 = !true; // !t retorna false
var n2 = !false; // !f retorna true
var n3 = !"Gato"; // !t retorna false

```

Caso a 1 expressao for falsa, o a1, retornará para o a3, e o a6 não vai receber string, e sim um booleano(And)

No ou, se o o1 for falso, ele retornará o o2

Not lógico, ele nega o valor, o n3 retorna falso

```

// True
" "
1

// False
" "
0

```

!! - esse operador converte para booleano

O Spread intera e passa para o parametro

Ele é utilizado para concatenar arrays

```
var partes = ['ombro', 'joelhos'];
var musica = ['cabeca', ...partes, 'e', 'pés'];

var musica = ['cabeca', 'ombro', 'joelhos', 'e', 'pés'];
```

```
function fn(x, y, z){ }
var args = [0, 1, 2];
fn(...args);
```

```
// Arrays
var arvores = new Array("pau-brasil", "loureiro", "cedro", "carvalho", "sicômoro");
0 in arvores; // retorna true
3 in arvores; // retorna true
6 in arvores; // retorna false
"cedro" in arvores; // retorna false (você deve especificar o número do índice,
                    // não o valor naquele índice)
"length" in arvores; // retorna true (length é uma propriedade de Array)

// Objetos predefinidos
"PI" in Math; // retorna true
var minhaString = new String("coral");
"length" in minhaString; // retorna true

// Objetos personalizados
var meucarro = {marca: "Honda", modelo: "Accord", ano: 1998};
"marca" in meucarro; // retorna true
"modelo" in meucarro; // retorna true

// instanceof
objeto instanceof tipoObjeto

var dia = new Date(2018, 12, 17);

if(dia instanceof Date) {
    // code here
}
```

Estruturas condicionais, if, else e else if

If- se a condição for verdadeira, você vai conseguir executar o código

```
CONDICIONAIS > IF > JS > i-if.js
1  /**
2
3  if (condition) {
4    // code
5  }
6
7 */
8
9 const array = [0, 1, 2, 3, 4, 5];
10
11 array.forEach(item => {
12   if (item % 2 === 0) {
13     console.log(`O número ${item} é par.`);
14   } else {
15     console.log(`O número ${item} é ímpar.`);
16   }
17 });

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Co
[Running] node "/home/cabrini/Desktop/exemplos/condicionais/if/tempCodeRunnerFile.
0 número 0 é par.
0 número 1 é ímpar.
0 número 2 é par.
0 número 3 é ímpar.
0 número 4 é par.
0 número 5 é ímpar.
```

Else, é se for errado

Else if- se a condição for valida, encadeia os if

```

const array = [2, 3, 4, 5, 6, 8, 10, 15];

console.log('\nelse if');
array.forEach(item => {
  if (item % 2 === 0) {
    console.log(`O número ${item} é divisível por 2.`);
  } else if(item % 3 === 0) {
    console.log(`O número ${item} é divisível por 3.`);
  } else if(item % 5 === 0) {
    console.log(`O número ${item} é divisível por 5.`);
  }
});

console.log('\nif');
array.forEach(item => {
  if (item % 2 === 0) {
    console.log(`O número ${item} é divisível por 2.`);
  }
});

```

Se o usuário decidir que quer que caia em mais de uma condição, ele não pode usar o `else if`

```

23 });
24
25 console.log('\nif');
26 array.forEach(item => {
27   if (item % 2 === 0) {
28     console.log(`O número ${item} é divisível por 2.`);
29   }
30   if(item % 3 === 0) [
31     console.log(`O número ${item} é divisível por 3.`);
32   ]
33   if(item % 5 === 0) {
34     console.log(`O número ${item} é divisível por 5.`);
35   }
36 });

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Code

```

if
0 número 2 é divisível por 2.
0 número 3 é divisível por 3.
0 número 4 é divisível por 2.
0 número 5 é divisível por 5.
0 número 6 é divisível por 2.
0 número 6 é divisível por 3.
0 número 8 é divisível por 2.
0 número 10 é divisível por 2.
0 número 10 é divisível por 5.
0 número 15 é divisível por 3.
0 número 15 é divisível por 5.

```



```
const fruit = 'banana';

switch(fruit) {
    case 'banana':
        console.log('R$ 3,00 / kg');
        break;
    case 'mamão':
    case 'pera':
        console.log('R$ 2,00 / kg');
        break;
    default:
        console.log('Produto não existe no estoque.');
        break;
}
```

O Switch case, define um valor, basta apenas ir incrementando os case, ele executa o default caso não coloque um break

Estruturas de repetição:

O Continue permite que , por exemplo em uma lista de 3 indices, permite que você leia do 1 e pular para o 3

O Break permite “matar” um laço

No código abaixo, o for

```
for ([expressaoInicial]; [condicao]; [incremento])
    declaracao
/*
const array = ['one', 'two', 'three'];

for (let index = 0; index < array.length; index++) {
    const element = array[index];
    console.log(`Element #${index}: ${element}.`);
}
```

O for em JavaScript permite repetir um bloco de código várias vezes de forma controlada, sendo ideal para iterar sobre arrays, executar tarefas repetitivas (como contagens ou manipulações) e criar listas dinamicamente, oferecendo controle total sobre a inicialização, condição e o incremento da iteração, facilitando o processamento eficiente de dados e a automação de tarefas.

Já o while, sempre executa uma condição que for verdadeira, no código, se não incrementar o n, ele ficará em um loop infinito

```
2
3     while (condicao)
4         declaracao
5
6     */
7
8     var n = 0;
9     var x = 0;
10    while (n < 3) {
11        n++;
12        x += n; // 1, 3, 6
13    }
14
15    console.log(x);
```

O while em JavaScript permite repetir um bloco de código enquanto uma condição for verdadeira (true), sendo ideal quando o número de repetições não é conhecido previamente, testando a condição antes de cada execução do bloco, o que pode resultar em zero execuções se a condição for falsa de início, e prevenindo loops infinitos através de alguma lógica dentro do loop que torne a condição falsa eventualmente.

O Do while, executa primeiro e depois verifica se a condição é verdadeira, diferente do while que só executa se for verdadeira

```
1  /**
2
3  do
4  |· declaracao
5  while (condicao);
6
7  */
8  let i = 0;
9
10 do {
11   i += 1;
12   console.log(i);
13 } while (i < 5);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[Running] node "/home/cabrini/Desktop/exemplos/repeticao/do..

```
1
2
3
4
5
```

For in, faz o `console.log` para cada elemento do `let`

O `for of` só faz o `console.log` para os números

```
repeticao / for...in arr / for...of arr
1  let arr = [3, 5, 7];
2  arr.foo = "hello";
3
4  for (let i in arr) {
5    console.log(i); // logs "0", "1", "2", "foo"
6  }
7
8  for (let i of arr) {
9    console.log(i); // logs "3", "5", "7"
10 }
```

`for...in` itera sobre as chaves/índices de propriedades enumeráveis de um objeto (incluindo herdadas), enquanto `for...of` itera sobre os valores dos elementos em

estruturas iteráveis (como Arrays, Strings, Maps, Sets), sendo mais moderno e recomendado para coleções de dados, retornando diretamente o valor. A escolha depende: `for...in` para propriedades de objetos, `for...of` para valores de arrays/coleções.

`for...in` (Chaves/Índices)

- **O que itera: Propriedades enumeráveis de um objeto (chaves/índices).**
- **Exemplo: Em um array ['a' , 'b'], ele retorna 0, 1 (os índices), não 'a', 'b' (os valores).**
- **Uso: Ideal para objetos simples, mas pode ser confuso em arrays por retornar índices, não valores.**

javascript

```
const pessoa = { nome: 'Ana', idade: 30 };
for (const chave in pessoa) {
  console.log(chave); // Saída: 'nome', 'idade' (as chaves)
}
```

`for...of` (Valores)

- **O que itera: Valores dos elementos em objetos iteráveis (Arrays, Strings, Map, Set).**
- **Exemplo: Em um array ['a' , 'b'], ele retorna 'a', 'b' diretamente.**
- **Uso: Preferível para arrays e outras coleções, pois é mais direto e legível para acessar os dados.**

javascript

```
const cores = ['vermelho', 'verde', 'azul'];
for (const cor of cores) {
  console.log(cor); // Saída: 'vermelho', 'verde', 'azul' (os valores)
```

```
}
```

Resumo da Diferença

- `for...in`: Para propriedades de objetos (chaves/índices).
- `for...of`: Para valores de coleções iteráveis (arrays, strings, etc.).

```
1 // break
2 console.log('Exemplo da utilização de break');
3
4 var index = 0;
5
6 while(true) {
7     index++;
8
9     if (index > 2) {
10         break;
11     }
12
13     console.log(index);
14 }
```

Nesse código, o `break` cancela o laço mesmo que a condição seja verdadeira

```
6
7 // continue
8 console.log('\nExemplo da utilização de continue');
9 const array = [1, 2, 3, 4, 5, 6];
0
1 for (let index = 0; index < array.length; index++) {
2     const element = array[index];
3
4     if (element % 2 === 0) {
5         continue;
6     }
7
8     console.log(element);
9 }
```

O `continue` pula uma interação

Em JavaScript, `break` interrompe completamente um loop (for, while, etc.), saindo dele e continuando o código após o loop; enquanto `continue` pula apenas a iteração atual,

ignorando o restante do código naquela volta e indo direto para a próxima iteração do loop, sem sair dele. `break` é para parar tudo; `continue` é para pular uma etapa.

ORIENTAÇÃO A OBJETOS COM JAVASCRIPT



DIGITAL
INNOVATION
ONE

Orientação a objetos

✓ Herança

- baseada em protótipos
- prototype
- `__proto__`
- `constructor`

String é uma função construtora

```
● ● ●  
  
'use strict';  
  
const myText = String('Hello prototype!');  
  
console.log(myText.__proto__.split);  
// f split() { [native code] }
```

Nesse código, podemos ver a função construtora string



DIGITAL
INNOVATION
ONE

Orientação a objetos

Object.prototype = null;

f Object () {} ←

Function.prototype.constructor

f Function ←

f Animal.constructor

[A guide to prototype-based class inheritance in JavaScript](#)



DIGITAL
INNOVATION
ONE

Orientação a objetos

```
new Foo(...);
```

```
/**  
O que ocorre?
```

- 1 - Um novo objeto é criado, herdando Foo.prototype
- 2 - A função construtora Foo é chamada com os argumentos especificados e com o 'this' vinculado ao novo objeto criado.
- 3 - Caso a função construtora tenha uma retorno explícito, será respeitado o seu 'return'. Senão, será retornado o objeto criado no passo 1.

```
*/
```

Guavview (29,98 fps)

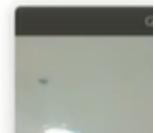


```
'use strict';

function Animal() {
    this.qtdePatas = 4;
}

const cachorro = new Animal();

console.log(cachorro.__proto__ === Animal.prototype);
// true
```



Cachorro aponta para animal, o proto dela aponta para o proto de function
Cachorro é uma instancia de animal, não de function



DIGITAL
INNOVATION
ONE

Orientação a objetos



```
'use strict';

function Animal(qtdePatas) {
    this.qtdePatas = qtdePatas;
}

function Cachorro(morde) {
    Animal.call(this, 4);

    this.morde = morde;
}

const pug = new Cachorro(false);

console.log(pug);
// Cachorro {qtdePatas: 4, morde: false}
```



The screenshot shows a browser's developer tools console with the following JavaScript code history:

```
> function Cachorro() {}
<- undefined
> Cachorro.prototype.latir = function() {}
<- f () {}
> const c = new Cachorro()
<- undefined
> c.__proto__
<- ▶ {latir: f, constructor: f}
> Cachorro.prototype.test = function() {}
<- f () {}
> c.__proto__
<- ▶ {latir: f, test: f, constructor: f}
> |
```

O **prototype** em JavaScript é um conceito fundamental que permite a herança de propriedades e métodos entre objetos, sendo a base do modelo de programação orientada a objetos (POO) da linguagem.

Diferentemente de linguagens baseadas em classes (como Java), onde a herança é definida por uma estrutura hierárquica rígida, o JavaScript usa um modelo flexível chamado herança prototípica.

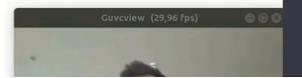


DIGITAL
INNOVATION
ONE

Orientação a objetos

✓ Classes

- ES6
- simplificação da herança de protótipos
- palavra chave **class**



DIGITAL
INNOVATION
ONE

Orientação a objetos

```
'use strict';

function Animal(qtdePatas) {
    this.qtdePatas = qtdePatas;
}

function Cachorro(morde) {
    Animal.call(this, 4);

    this.morde = morde;
}

const pug = new Cachorro(false);

console.log(pug);
// Cachorro {qtdePatas: 4, morde: false}
```



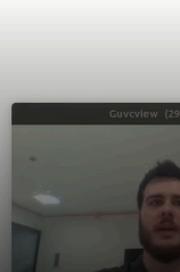
```
'use strict';

class Animal {
    constructor(qtdePatas) {
        this.qtdePatas = 4;
    }
}

class Cachorro extends Animal {
    constructor(morde) {
        super(4);
        this.morde = morde;
    }
}

const pug = new Cachorro(false);

console.log(pug);
// Cachorro {qtdePatas: 4, morde: 4}
```



O Class é uma simplificação do prototype

```

1 class Person {
2     constructor(name) {
3         this.name = name;
4     }
5 }
6
7 class PessoaF extends Person {
8     constructor(name, cpf) {
9         super(name);
10        this.cpf = cpf;
11    }
12 }

----- (redacted code)
----- && superClass !== null) { throw new TypeError("Super expression must either be
----- null or a function"); } subClass.prototype = Object.create(superClass &&
----- superClass.prototype, { constructor: { value: subClass, writable: true,
----- configurable: true } }); if (superClass) _setPrototypeOf(subClass, superClass);
----- }

12
13 function _setPrototypeOf(o, p) { _setPrototypeOf = Object.setPrototypeOf || |
----- function _setPrototypeOf(o, p) { o.__proto__ = p; return o; }; return
----- _setPrototypeOf(o, p); }
14
15 function _instanceof(left, right) { if (right != null && typeof Symbol !==
----- "undefined" && right[Symbol.hasInstance]) { return right[Symbol.hasInstance]
----- (left); } else { return left instanceof right; } }
16
17 function _classCallCheck(instance, Constructor) { if (!_instanceof(instance,
----- Constructor)) { throw new TypeError("Cannot call a class as a function"); } }
18
19 var Person = function Person(name) {
20     _classCallCheck(this, Person);
21
22     this.name = name;
23 };
24
25 var PessoaF =
26 /*#__PURE__*/
27 function (_Person) {
28     _inherits(PessoaF, _Person);
29
30     function PessoaF(name, cpf) {
31         var _this;
32
33         _classCallCheck(this, PessoaF);
34
35         _this = _possibleConstructorReturn(
----- this, name));
36
37         _this.cpf = cpf;
38         return _this;
39     }
40
41     return PessoaF;
42 }(Person);

```



Uma classe em JavaScript é uma estrutura que define um objeto com propriedades e métodos. Ela serve como um molde para criar objetos com características semelhantes.



Orientação a objetos

```

'use strict';

class Person {
    #name = '';

    constructor(initialName) {
        this.#name = initialName;
    }

    setName(name) {
        this.#name = name;
    }

    getName() {
        return this.#name;
    }
}

----- (redacted code)

----- && superClass !== null) { throw new TypeError("Super expression must either be
----- null or a function"); } subClass.prototype = Object.create(superClass &&
----- superClass.prototype, { constructor: { value: subClass, writable: true,
----- configurable: true } }); if (superClass) _setPrototypeOf(subClass, superClass);
----- }

12
13 function _setPrototypeOf(o, p) { _setPrototypeOf = Object.setPrototypeOf || |
----- function _setPrototypeOf(o, p) { o.__proto__ = p; return o; }; return
----- _setPrototypeOf(o, p); }
14
15 function _instanceof(left, right) { if (right != null && typeof Symbol !==
----- "undefined" && right[Symbol.hasInstance]) { return right[Symbol.hasInstance]
----- (left); } else { return left instanceof right; } }
16
17 function _classCallCheck(instance, Constructor) { if (!_instanceof(instance,
----- Constructor)) { throw new TypeError("Cannot call a class as a function"); } }
18
19 var Person = function Person(name) {
20     _classCallCheck(this, Person);
21
22     this.name = name;
23 };
24
25 var PessoaF =
26 /*#__PURE__*/
27 function (_Person) {
28     _inherits(PessoaF, _Person);
29
30     function PessoaF(name, cpf) {
31         var _this;
32
33         _classCallCheck(this, PessoaF);
34
35         _this = _possibleConstructorReturn(
----- this, name));
36
37         _this.cpf = cpf;
38         return _this;
39     }
40
41     return PessoaF;
42 }(Person);

```



Design patterns

✓ Formato de um pattern

- Nome
- Exemplo
- Contexto
- Problema
- Solução



Design patterns

✓ Padrões de criação

Os padrões de criação são aqueles que abstraem e/ou adiam o processo de criação dos objetos. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados.



Design patterns

✓ Padrões estruturais

Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores.

Design patterns

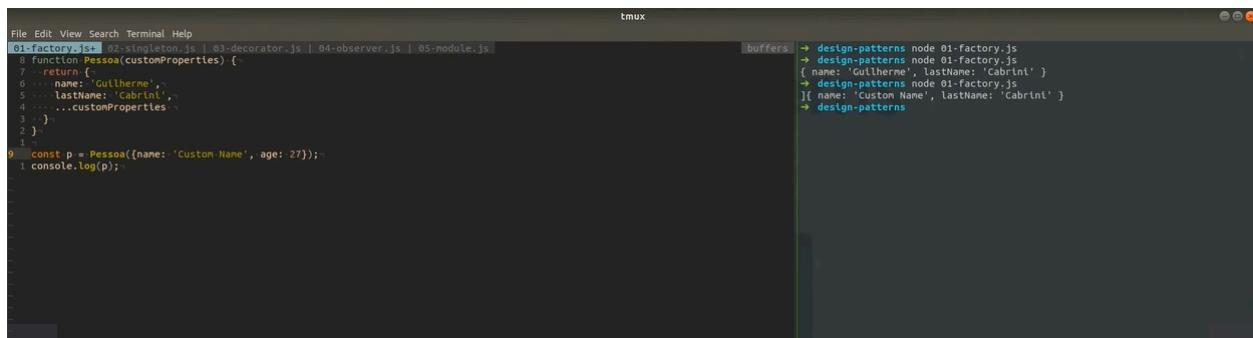
✓ Padrões comportamentais

Os padrões de comportamento se concentram nos algoritmos e atribuições de responsabilidades entre os objetos. Eles não descrevem apenas padrões de objetos ou de classes, mas também os padrões de comunicação entre os objetos.

Design patterns

✓ Factory

Todas as funções que retornam um objeto, sem a necessidade de chamá-las com o **new**, são consideradas funções Factory(fábrica).



The screenshot shows a tmux session with multiple panes. The left pane displays a code editor with a file named '01-factory.js' containing the following code:

```
File Edit View Search Terminal Help
01-factory.js | 02-singleton.js | 03-decorator.js | 04-observer.js | 05-module.js
0 function Pessoa(customProperties) {
1   return {
2     ...name: 'Guilherme',
3     ...lastName: 'Cabral',
4     ...customProperties
5   }
6 }
7 const p = Pessoa({name: 'Custom Name', age: 27});
8 console.log(p);
```

The right pane, titled 'buffers', shows the output of running the script:

```
design-patterns node 01-factory.js
design-patterns node 01-factory.js
{ name: 'Guilherme', lastName: 'Cabral' }
design-patterns node 01-factory.js
{ name: 'Custom Name', lastName: 'Cabral' }
design-patterns
```

Design patterns

✓ Singleton

O objetivo desse pattern é criar uma única instância de uma função construtora e retorná-la toda vez em que for necessário utilizá-la.

jQuery:

<https://jquery.com/>

Guvcview (29.92 fps)

Design patterns

```
function MyApp() {
    if (!MyApp.instance) {
        MyApp.instance = this;
    }

    return MyApp.instance;
}
```

Guvcview



Caso chame a função myapp, caso a função instance não é definida, ele instancia, mas o return sempre é chamado



Design patterns

✓ Decorator

Uma função decorator recebe uma outra função como parâmetro e estende o seu comportamento sem modificá-la explicitamente.



Proposta:

<https://github.com/tc39/proposal-decorators>

TypeScript:

<https://www.typescriptlang.org/docs/handbook/decorators>



```
13 let loggedIn = false;
12
11 function callIfAuthenticated(fn) {
10   return !!loggedIn && fn();
9 }
8
7 function sum(a, b) {
6   return a + b;
5 }
4
3 console.log(callIfAuthenticated(() => sum(2, 3)));
2 loggedIn = true;
1 console.log(callIfAuthenticated(() => sum(2, -3)));
$ loggedIn = false;
1 console.log(callIfAuthenticated(() => sum(2, -3)));
```



Design patterns

✓ Observer

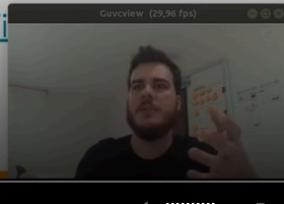
É um pattern muito popular em aplicações javascript. A instância (subscriber) mantém uma coleção de objetos (observers) e notifica todos eles quando ocorrem mudanças no estado.

Vue:

<https://github.com/vuejs/vue/blob/dev/src/core/observer/index.js>

RxJs:

<https://rxjs-dev.firebaseio.com/guide/observable>



Design patterns

✓ Module

É um pattern que possibilita organizarmos melhor o nosso código, sem a necessidade de expor variáveis globais.

```
~ Press ? for help
.. (up a dir)
/Aulas/design-patterns/
01-factory.js
02-singleton.js
03-decorator.js
04-observer.js
05-module.js
06-module-import.js

3 const {get fname, set fname} = require('./05-module.js');
4
5 console.log(fname());
6 console.log(set fname('Guilherme'));
7 console.log(fname());

→ design-patterns node 01-factory.js
name: 'Guilherme', lastName: 'Cabrinha' }
→ design-patterns node 01-factory.js
[ { name: 'Custom Name', lastName: 'Cabrin' } ]
→ design-patterns node 01-factory.js
{ name: 'Custom Name', lastName: 'Cabrin', age: 27 }
→ design-patterns node 02-singleton.js
( name: 'Guilherme' )
→ design-patterns node 03-decorator.js
false
→ design-patterns node 03-decorator.js
false
5
false
→ design-patterns node 04-observer.js
Subscribe 1: notified 1
Subscribe 2: notified 1
Subscribe 3: notified 1
→ design-patterns node 04-observer.js
Subscribe 1: notified 1
Subscribe 2: notified 1
Subscribe 3: notified 1
Subscribe 1: notified 2
Subscribe 3: notified 2
→ design-patterns node 06-module-import.js
default
undefined
Guilherme
→ design-patterns
```

Manipulação de Array

Criar um array

```
● ● ●  
const arr = [1, 2, 3];  
const arr2 = new Array(1, 2, 3);
```

Método mais simples, criar variável, dá um nome para ela e a separa por colchetes, na 2 linha instancia um novo array

Criar um array

- ✓ Array.of

Cria uma nova instância de array a partir do número de parâmetros informados

The screenshot shows a browser's developer tools with the 'Console' tab selected. The console output is as follows:

```
⚠ The key "target-densitydpi" is not supported.  
> const persons = Array.of('John', 'Cris', 'Jenny')  
< undefined  
> persons  
< ▶ (3) ["John", "Cris", "Jenny"]  
> |
```

Nesse código, o arrayof recebeu todos os parametros e os armazenou em persons



Criar um array

- ✓ Array

Cria uma nova instância de array de acordo com os parâmetros informados



Criar um array

- ✓ Array.from

Cria uma nova instância de array a partir de um parâmetro array-like ou iterable object

Inserir e remover elementos

- ✓ push

Adicionar um ou mais elementos no final do array e retorna o tamanho do novo array

```
const arr = ['banana', 'melancia', 'abacate'];
const arrLength = arr.push('acerola');

console.log(arrLength)
// 4

console.log(arr);
// ['banana', 'melancia', 'abacate', 'acerola'];
```

Se der um `console.log`, vai dar 4, o tamanho do array

Inserir e remover elementos

- ✓ pop

Remove o último elemento de um array e retorna o elemento removido



Inserir e remover elementos

- ✓ unshift

Adicionar um ou mais elementos no início do array e retorna o tamanho do novo array



Inserir e remover elementos

- ✓ shift

Remove o primeiro elemento de um array e retorna o elemento removido



Quase o mesmo comportamento do pop

```
● ● ●  
  
const arr = ['banana', 'melancia', 'abacate'];  
const removedItem = arr.shift();  
  
console.log(removedItem)  
// 'banana'  
  
console.log(arr);  
// ['melancia', 'abacate'];
```



Inserir e remover elementos

- ✓ concat

Concatena um ou mais arrays retornando um novo array

```
● ● ●

const arr = [1, 2, 3];
const arr2 = [4, 5, 6];

const novoArr = arr.concat(arr2);

console.log(arr);
// (3) [1, 2, 3]

console.log(arr2);
// (3) [4, 5, 6]

console.log(novoArr);
// (6) [1, 2, 3, 4, 5, 6]
```

Inserir e remover elementos

✓ slice

Retorna um novo array “fatiando” o array de acordo com ínicio e fim



```
const arr = [1, 2, 3, 4, 5];

arr.slice(0, 2);
// [1, 2]

arr.slice(2);
// [3, 4, 5]

arr.slice(-1);
// [5]

arr.slice(-3);
// [3, 4, 5]
```

Array sempre começa com o índice 0, com o slice, ele não pega o índice final, ou seja, ele só pega a posição 0 e 1, não é necessário passar 2 parâmetros



Inserir e remover elementos

- ✓ splice

Altera um array adicionando novos elementos enquanto remove elementos antigos



```
● ● ●  
  
const arr = [1, 2, 3, 4, 5];  
  
arr.splice(2)  
// [3, 4, 5]  
  
console.log(arr);  
// [1, 2]  
  
arr.splice(0, 0, 'first');  
// []  
  
console.log(arr);  
// ["first", 1, 2]
```

Se der um arr.splice, a partir da posição 2, ele vai remover os itens, ou seja, ele não é imutável, se der um console.log, ele só vai ler 1 e 2

```
> frutas
< ▶ (2) ["melancia", "banana"]
> frutas.splice(1, 0, "acerola")
< ▶ []
> frutas
< ▶ (3) ["melancia", "acerola", "banana"]
> frutas.splice(2, 1, "laranja", "amora")
< ▶ ["banana"]
> frutas
< ▶ (4) ["melancia", "acerola", "laranja", "amora"]
> |
```

forEach

Iteração de cada item dentro de um array



```
const arr = [1, 2, 3, 4, 5];

arr.forEach((value, index) => {
  console.log(` ${index}: ${value}`);
});
```

Iterar elementos

- ✓ map

Retorna um novo array, de mesmo tamanho, iterando cada item de um array

Iterar elementos

- ✓ flat

Retorna um novo array com todos os elementos de um sub-array concatenados de forma recursiva de acordo com a profundidade especificada(depth)

```
> frutas.map((fruta, index) => `${index} - ${fruta}`)
< ▶ (4) ["0 - melancia", "1 - acerola", "2 - laranja", "3 - amora"]
> frutas
< ▶ (4) ["melancia", "acerola", "laranja", "amora"]
>
```

Iterar elementos

- ✓ flat

Retorna um novo array com todos os elementos de um sub-array concatenados de forma recursiva de acordo com a profundidade especificada(depth)



```
const arr = [1, 2, [3, 4]];
```

```
arr.flat();
// [1, 2, 3, 4]
```

```
[✖] [✖] top ▾ [✖] Filter Default levels ▾
> frutas.map((fruta, index) => `${index} - ${fruta}`)
< ▶ (4) ["0 - melancia", "1 - acerola", "2 - laranja", "3 - amora"]
> frutas
< ▶ (4) ["melancia", "acerola", "laranja", "amora"]
> const idades = [20, 34, [35, 60, [70, 40]]];
< undefined
> idades
< ▶ (3) [20, 34, Array(3)]
> [20, 34, 35, 60, 70, 40]
< ▶ (6) [20, 34, 35, 60, 70, 40]
> idades.flat(2)
< ▶ (6) [20, 34, 35, 60, 70, 40]
> |
```

Cursor icon at the bottom left.

Iterar elementos

- ✓ flatMap

Retorna um novo array assim como a função map e executa um flat de profundidade 1

```
const arr = [1, 2, 3, 4];

arr.flatMap(value => [value * 2]);
// [2, 4, 6, 8]

arr.flatMap(value => [[value * 2]])
// [[2], [4], [6], [8]]
```

- ✓ keys

Retorna um **Array Iterator** que contém as chaves para cada elemento do array

```
const arr = [1, 2, 3, 4];

const arrIterator = arr.keys();

arrIterator.next();
// {value: 0, done: false}

arrIterator.next();
// {value: 1, done: false}

arrIterator.next();
// {value: 2, done: false}

arrIterator.next();
// {value: 3, done: true}
```



Iterar elementos

✓ values

Retorna um **Array Iterator** que contém os valores para cada elemento do array

```
const arr = [1, 2, 3, 4];
           ^
const arrIterator = arr.values();

arrIterator.next();
// {value: 1, done: false}

arrIterator.next();
// {value: 2, done: false}

arrIterator.next();
// {value: 3, done: false}

arrIterator.next();
// {value: 4, done: true}
```



DIGITAL
INNOVATION
ONE

Iterar elementos

- ✓ entries

Retorna um **Array Iterator** que contém os pares chave/valor para cada elemento do array



```
● ● ●
```

```
const arr = [1, 2, 3, 4];

const arrIterator = arr.values();

arrIterator.next();
// {value: [0, 1], done: false}

arrIterator.next();
// {value: [1, 2], done: false}

arrIterator.next();
// {value: [2, 3], done: false}

arrIterator.next();
// {value: [3, 4], done: true}
```

```
▶ | top ▶ | ⚡ | Filter

> frutas
< ▶ (4) ["melancia", "acerola", "laranja", "amora"]
> const frutasIterator = frutas.entries()
< undefined
> frutasIterator
< ▶ Array Iterator {}
> frutasIterator.next()
< ▶ {value: Array(2), done: false}
  done: false
  ▶ value: (2) [0, "melancia"]
  ▶ __proto__: Object
> frutasIterator.next()
< ▶ {value: Array(2), done: false}
  done: false
  ▶ value: Array(2)
    0: 1
    1: "acerola"
    length: 2
    ▶ __proto__: Array(0)
    ▶ proto__: Object
> frutasIterator.next()
< ▶ {value: Array(2), done: false}
  done: false
  ▶ value: (2) [2, "laranja"]
  ▶ __proto__: Object
> frutasIterator.next()
< ▶ {value: Array(2), done: false}
  done: false
  ▶ value: (2) [3, "amora"]
  ▶ __proto__: Object
> frutasIterator.next()
< ▶ {value: undefined, done: true}
  done: true
  value: undefined
  ▶ __proto__: Object
>
```



Buscar elementos

✓ **find**

Retorna o primeiro item de um array que satisfaz a condição

```
const arr = [1, 2, 3, 4];

const firstGreaterThanOrEqualToTwo = arr.find(value => value > 2);

console.log(firstGreaterThanOrEqualToTwo);
// 3
```



Buscar elementos

✓ findIndex

Retorna o índice do primeiro item de um array que satisfaz a condição

Ele, ao em vez de retornar o valor, igual o find, retorna o index

Buscar elementos

- ✓ filter

Retorna um novo array com todos os elementos que satisfazem a condição



```
const arr = [1, 2, 3, 4];

const allValuesGreaterThanOrEqualToTwo = arr.filter(value => value > 2);

console.log(allValuesGreaterThanOrEqualToTwo);
// [3, 4]
```

Buscar elementos

- ✓ indexOf

Retorna o primeiro índice em que um elemento pode ser encontrado no array



```
const arr = [1, 3, 3, 4, 3];  
  
const firstIndexofItem = arr.indexof(3);  
  
console.log(firstIndexofItem);  
// 1
```



Buscar elementos

- ✓ `lastIndexOf`

Retorna o último índice em que um elemento pode ser encontrado no array



DIGITAL
INNOVATION
ONE

Buscar elementos

- ✓ `includes`

Retorna um booleano verificando se determinado elemento existe no array



```
● ● ●  
const arr = [1, 3, 3, 4, 3];  
  
const hasItemOne = arr.includes(1);  
// true  
  
const hasItemTwo = arr.includes(2);  
// false
```



Buscar elementos

✓ `some`

Retorna um booleano verificando se pelo menos um item de um array satisfaz a condição

```
● ● ●  
const arr = [1, 3, 3, 4, 3];  
  
const hasSomeEvenNumber = arr.some(value => value % 2 === 0);  
// true
```

Guyview (29.86 s)

```
> students
< ▷ (3) [ { }, { }, { } ] ⌂
  ▷ 0: { name: "John", grade: 7 }
  ▷ 1: { name: "Jenny", grade: 5 }
  ▷ 2: { name: "Peter", grade: 4 }
  length: 3
  ▷ __proto__: Array(0)
> students.some(student => student.grade >= 7);
< true
> students.find(student => student.grade >= 7);
< ▷ { name: "John", grade: 7 }
> students.findIndex(student => student.grade >= 7);
< 0
> |
```



DIGITAL
INNOVATION
ONE

Buscar elementos

✓ `every`

Retorna um booleano verificando se todos os itens de um array
satisfazem a condição

```
const arr = [1, 3, 3, 4, 3];
const allEvenNumbers = arr.every(value => value % 2 === 0);
// false
```

```
> students
< ▷ (3) [ {}, {}, {} ] ⓘ
  ▷ 0: {name: "John", grade: 7}
  ▷ 1: {name: "Jenny", grade: 5}
  ▷ 2: {name: "Peter", grade: 4}
  length: 3
  ▷ __proto__: Array(0)
> students.some(student => student.grade >= 7);
< true
> students.find(student => student.grade >= 7);
< ▷ {name: "John", grade: 7}
> students.findIndex(student => student.grade >= 7);
< 0
> students.some(student => student.grade >= 7);
< true
> students.every(student => student.grade >= 7);
< false
> students.find(student => student.grade < 7);
< ▷ {name: "Jenny", grade: 5}
> ↶
```

✓ sort

Ordena os elementos de um array de acordo com a condição

✓ reverse

Inverte os elementos de um array



```
const arr = [1, 2, 3, 4, 5];
```

```
arr.reverse();
// [5, 4, 3, 2, 1]
```

✓ join

Junta todos os elementos de um array, separados por um delimitador e retorna uma string

✓ reduce

Retorna um novo tipo de dado iterando cada posição de um array



```
const arr = [1, 2, 3, 4, 5];  
  
arr.reduce((total, value) => total += value, 0);  
// 15
```

```
Default levels ▾
```

```
> students
< ▶ (3) [{-}, {-}, {-}] ⓘ
  ► 0: {name: "John", grade: 7}
  ► 1: {name: "Jenny", grade: 5}
  ► 2: {name: "Peter", grade: 4}
  length: 3
  ► __proto__: Array(0)
> students.some(student => student.grade >= 7);
< true
> students.find(student => student.grade >= 7);
< ► {name: "John", grade: 7}
> students.findIndex(student => student.grade >= 7);
< 0
> students.some(student => student.grade >= 7);
< true
> students.every(student => student.grade >= 7);
< false
> students.find(student => student.grade < 7);
< ► {name: "Jenny", grade: 5}
> students.sort((current, next) => current.grade - next.grade);
< ▶ (3) [{-}, {-}, {-}] ⓘ
  ► 0: {name: "Peter", grade: 4}
  ► 1: {name: "Jenny", grade: 5}
  ► 2: {name: "John", grade: 7}
  length: 3
  ► __proto__: Array(0)
> students.sort((current, next) => next.grade - current.grade);
< ▶ (3) [{-}, {-}, {-}] ⓘ
  ► 0: {name: "John", grade: 7}
  ► 1: {name: "Jenny", grade: 5}
  ► 2: {name: "Peter", grade: 4}
  length: 3
  ► __proto__: Array(0)
⚠ ▶ The key "target-densitydpi" is not supported.
> students
< ▶ (3) [{-}, {-}, {-}]
> students.reduce((totalGrades, student) => totalGrades += student.grade, 0)
< 16
> students.reduce((totalGrades, student) => totalGrades += student.grade, 0) / students.length
< 5.333333333333333
> |
> |
> students.reduce((studentsNames, student) => studentsNames += student.name + ' ', '')
< "John Jenny Peter "
```