o
poder
do
generator

olá!

sou o Fillipe Hott

- dev javascript
- grad. jogos digitais
- pós em desenvolvimento de sistemas web
- apaixonado por cinema e música

mas o
que é
um generator?

para explicar o que é um **generator…**

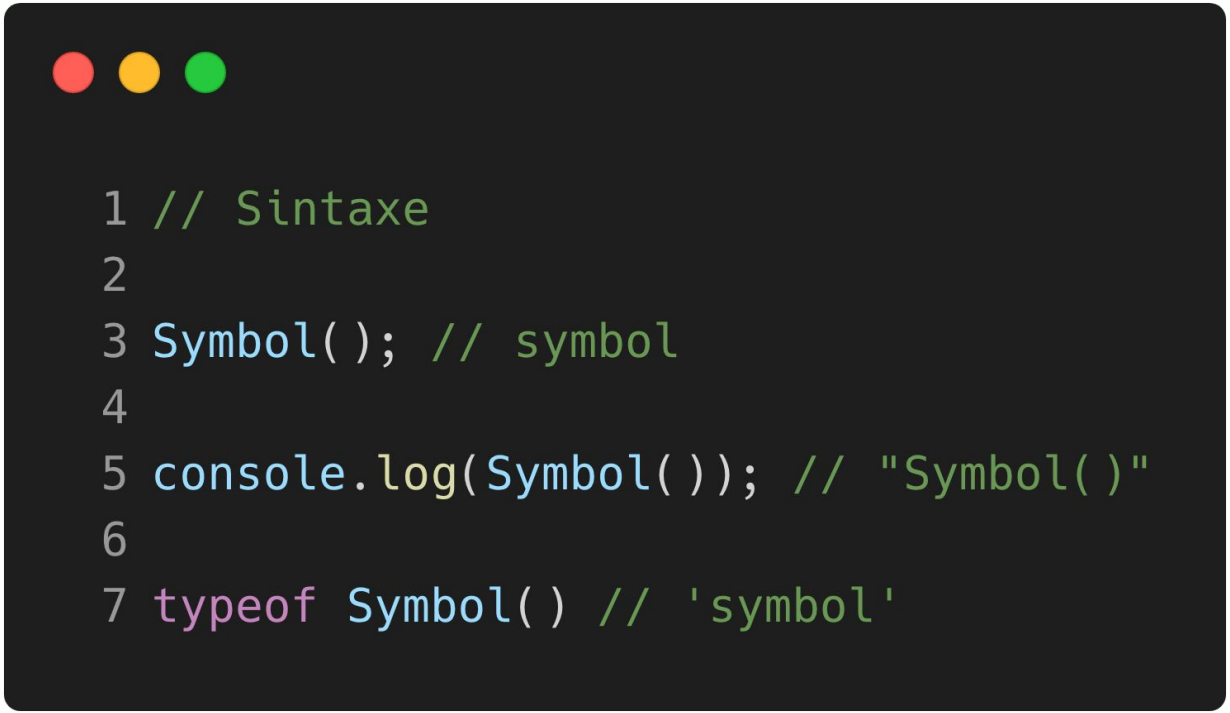
para explicar o que é um **generator…**

eu preciso, primeiro,

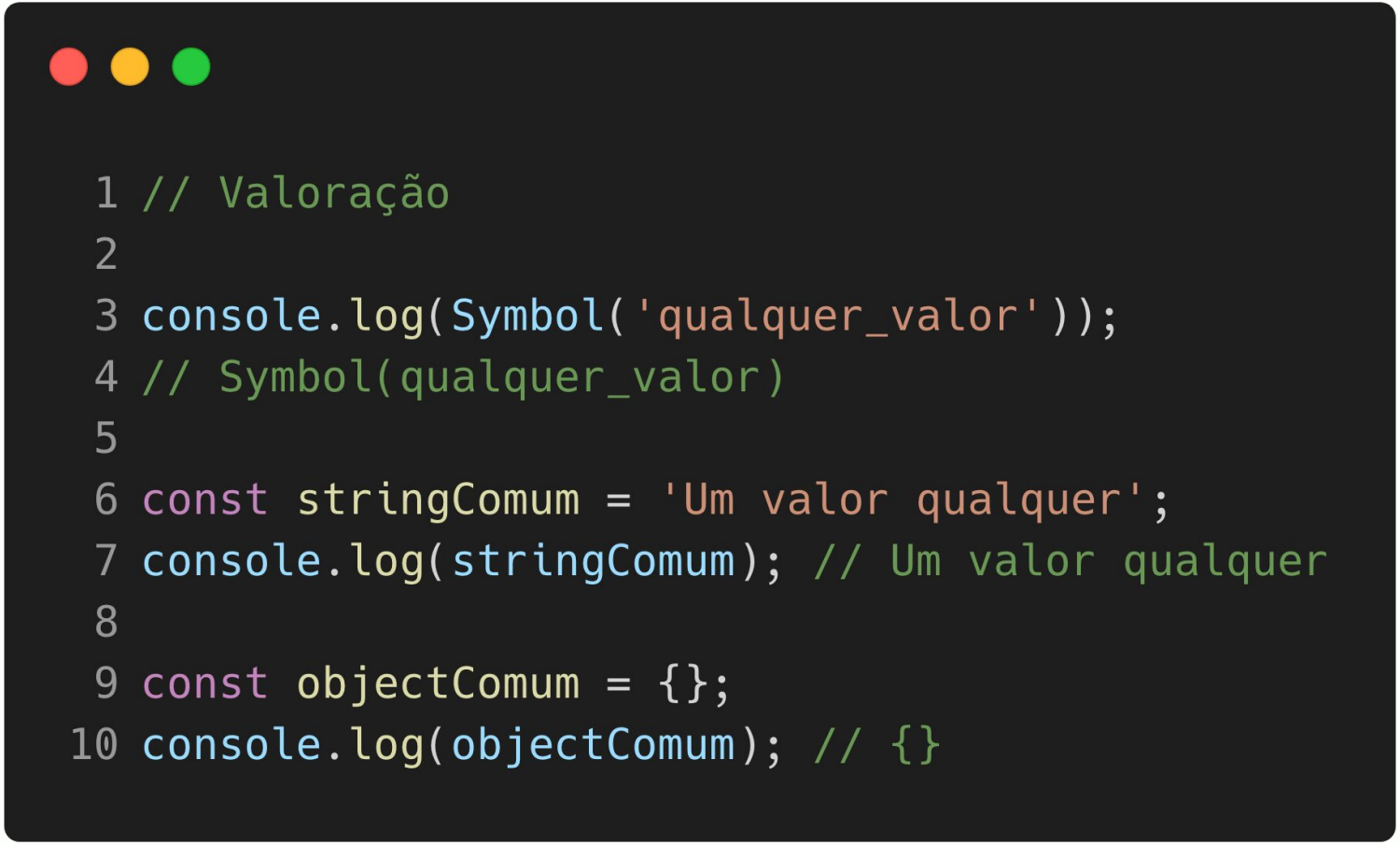
explicar o que é um **Symbol** :)

Symbol

_ chegou na versão ECMAScript 2015 / ES6
_ é um tipo primitivo, igualmente number, string e boolean.
_ possui uma função para ser criado: Symbol().
_ quando é declarado, possui um valor **único**.

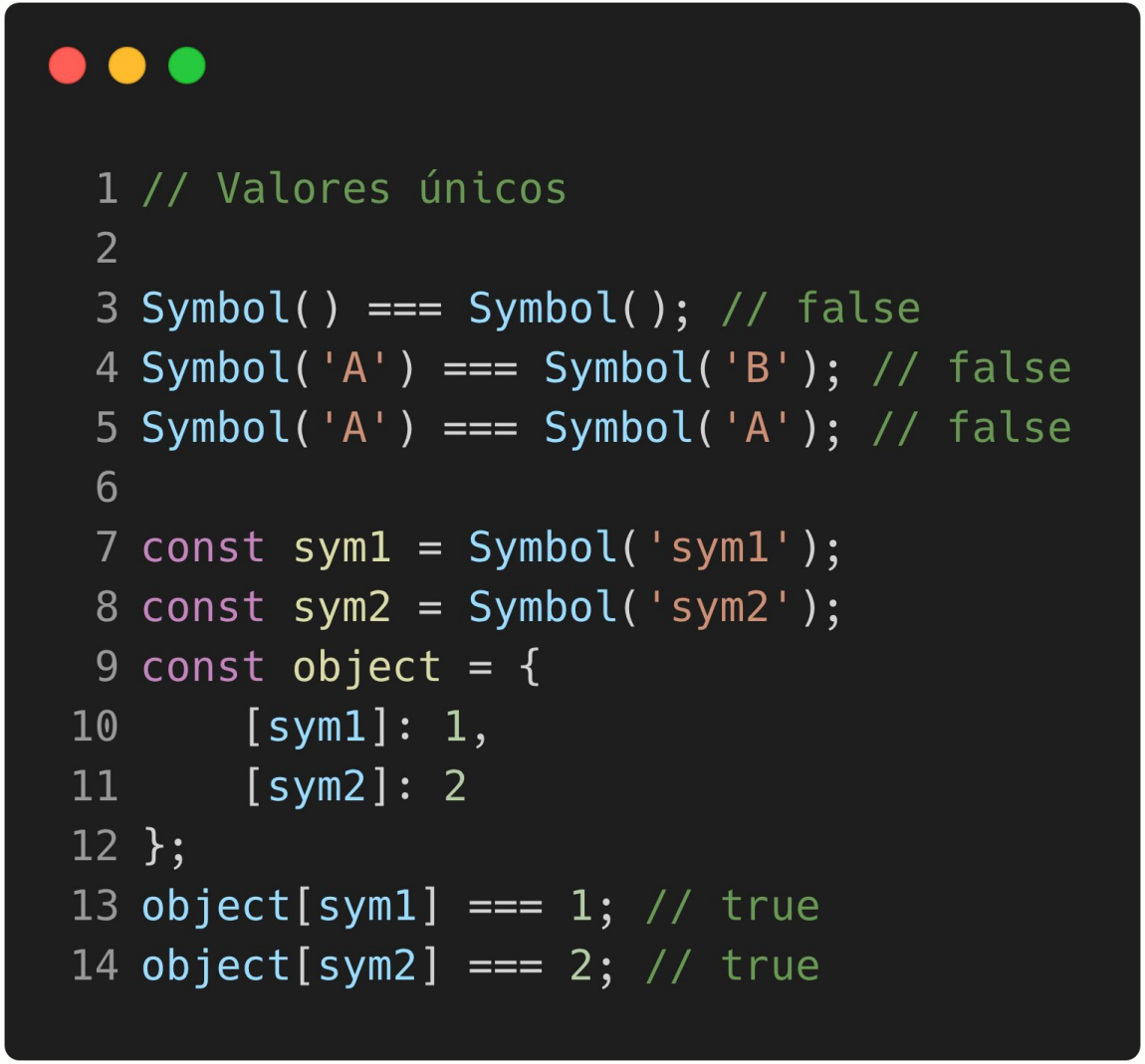


```
// Sintaxe

Symbol(); // symbol

console.log(Symbol()); // "Symbol()"

typeof Symbol() // 'symbol'
```



```
// Valoração

console.log(Symbol('qualquer_valor'));
// Symbol(qualquer_valor)

const stringComum = 'Um valor qualquer';
console.log(stringComum); // Um valor qualquer

const objectComum = {};
console.log(objectComum); // {}
```

```
// Chaveamento de objetos

const chaveSymbol = Symbol('chave_1');
const meuObj = {};

meuObj['nome'] = 'Maria';
meuObj[chaveSymbol] = 'Algum valor';

console.log(meuObj);
// { nome: "Maria", Symbol(chave_1): "Algum valor" }
console.log(meuObj[chaveSymbol]); // Algum valor
```


// Valores únicos
Symbol() === Symbol(); // false
Symbol('A') === Symbol('B'); // false
Symbol('A') === Symbol('A'); // false
const sym1 = Symbol('sym1');
const sym2 = Symbol('sym2');
const object = {
[sym1]: 1,
[sym2]: 2
};
object[sym1] === 1; // true
object[sym2] === 2; // true

tudo certo até aqui?

agora que vocês sabem

o que é um **Symbol…**

agora que vocês sabem

o que é um **Symbol…**

eu vou mostrar que são

iterables e **iterators** :)

iterables & iterators

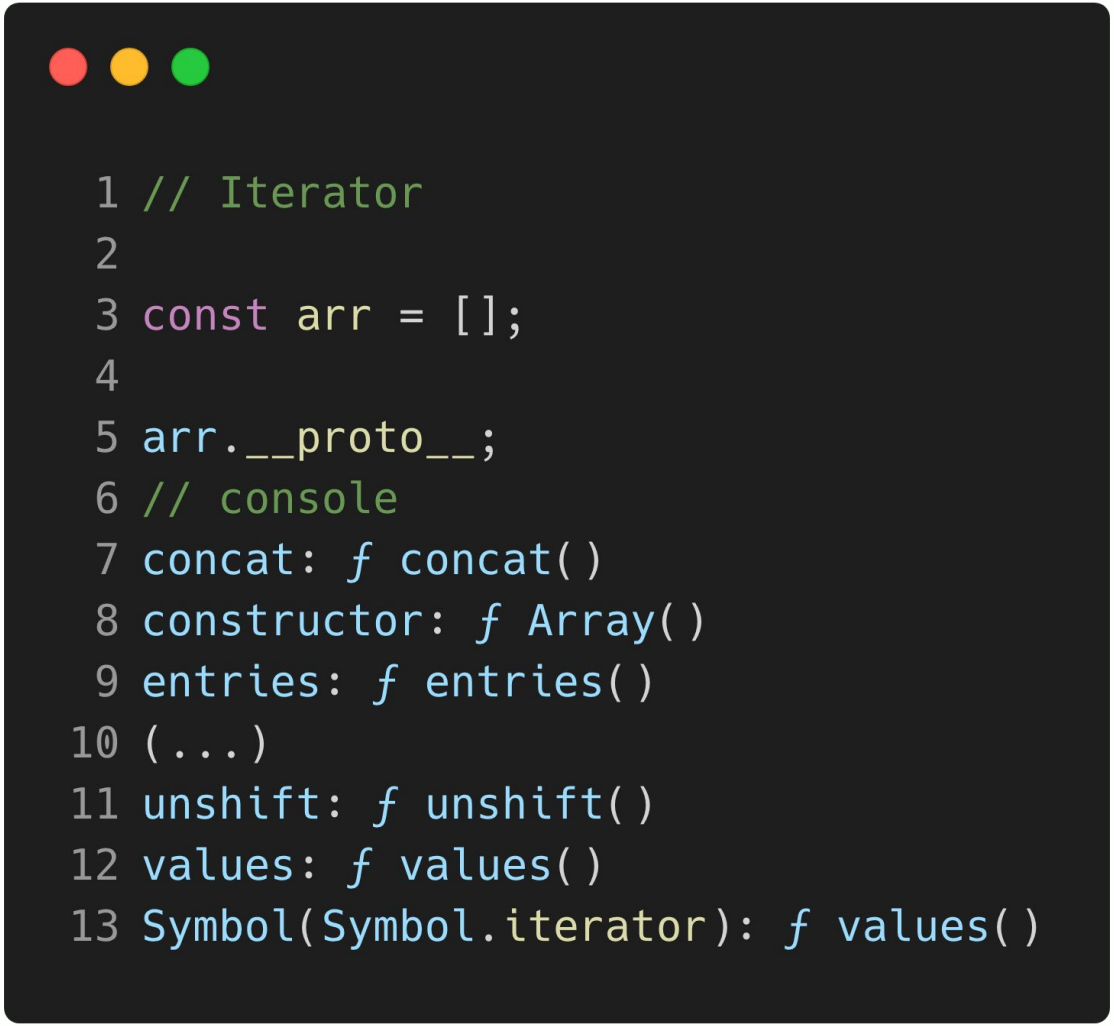
algumas adições ao ECMAScript 2015 / ES6 não são novos *built-ins* ou sintaxe, mas **protocolos**.

existem dois protocolos: o **protocolo iterável** e o **protocolo iterador**.

iterables

todo objeto que implementa uma chave Symbol.iterator.

_ arrays
_ strings
_ maps
_ sets
_ ...


// Iterator
const arr = [];
arr.__proto__;
// console
concat: ƒ concat()
constructor: ƒ Array()
entries: ƒ entries()
(...)
unshift: ƒ unshift()
values: ƒ values()
Symbol(Symbol.iterator): ƒ values()

iterators

objeto que sabe como acessar itens de uma coleção, um de cada vez, enquanto mantém o controle de sua posição atual dentro dessa sequência.

iterators

_ o Symbol tem um valor único;
_ o Symbol.Iterator sabe em qual ponto está o *loop*;
_ o retorno do Symbol.Iterator traz um objeto com o valor da iteração e se ela terminou;
_ esse objeto é obtido pela função next().

```
1 // string iterator
2
3 const minhaString = 'hi';
4 typeof minhaString[Symbol.iterator]; // "function"
5
6 const iterator = minhaString[Symbol.iterator]();
7
8 iterator.next(); // { value: "h", done: false }
9 iterator.next(); // { value: "i", done: false }
10 iterator.next(); // { value: undefined, done: true }
```

```
// Alterando um iterator

const minhaString = new String('oi');

minhaString[Symbol.iterator] = function() {
  return {
    _primeiro: true, // atributo adicionado
    next: function() {
      if (this._primeiro) {
        this._primeiro = false;
        return { value: 'tchau', done: false };
      } else {
        return { done: true };
      }
    }
  };
};

[...minhaString]; // ['tchau']
```

tudo certo
até aqui também?

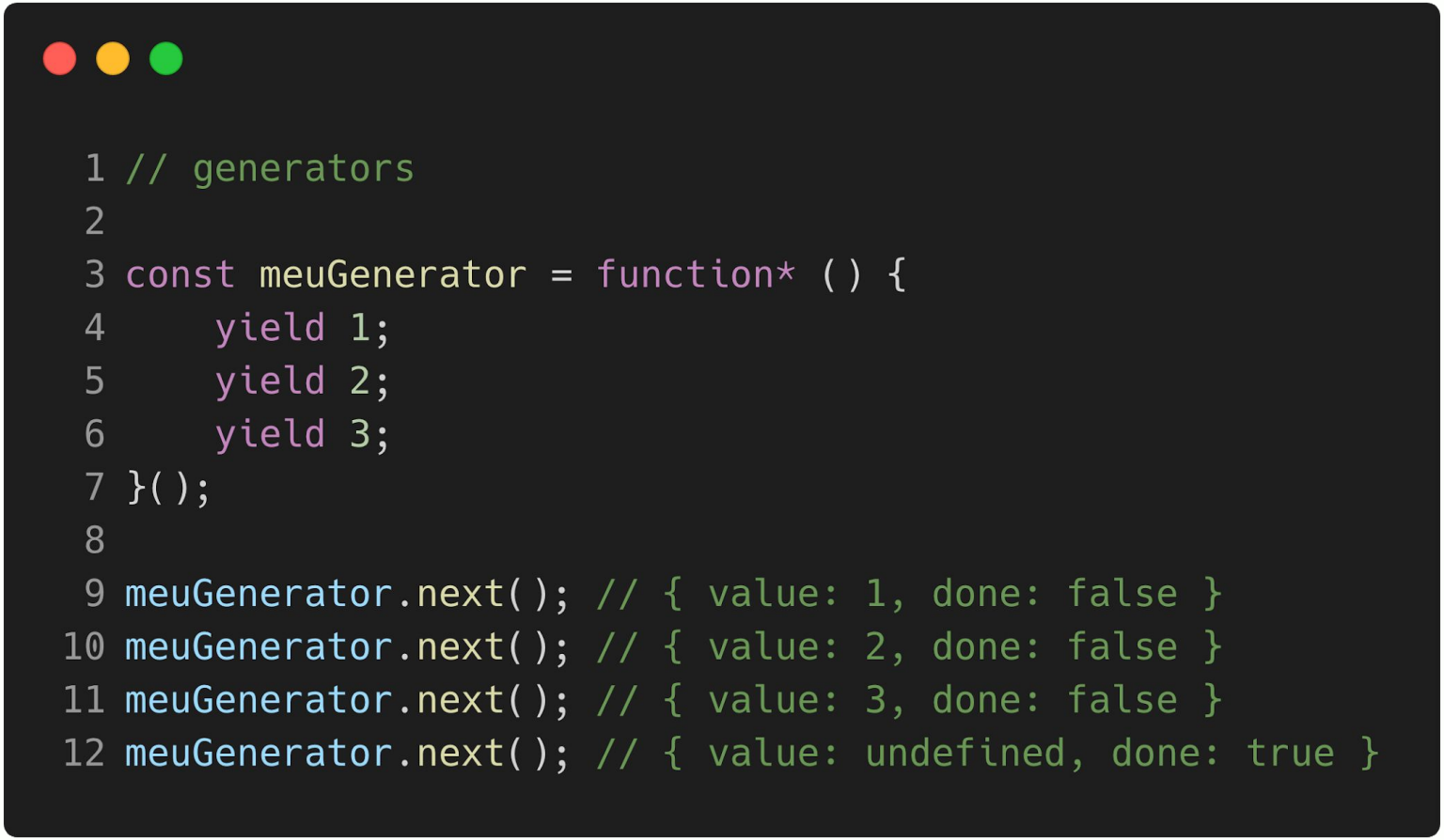
agora sim…

agora sim…

o **generator**!

generator

_ é declarado como uma função, mas com um * (asterísco);
_ possui chamadas yield(), que param o processamento;
_ possui o método next(), para seguir o processamento.



```
// generators

const meuGenerator = function* () {
    yield 1;
    yield 2;
    yield 3;
}();

meuGenerator.next(); // { value: 1, done: false }
meuGenerator.next(); // { value: 2, done: false }
meuGenerator.next(); // { value: 3, done: false }
meuGenerator.next(); // { value: undefined, done: true }
```

```
// generator

// certo!
function* myGenerator() {
  // ...
}

// certo também!
const myGenerator = function* () {
  // ...
}

// errado
const* myGenerator => () {
  // ...
}
```

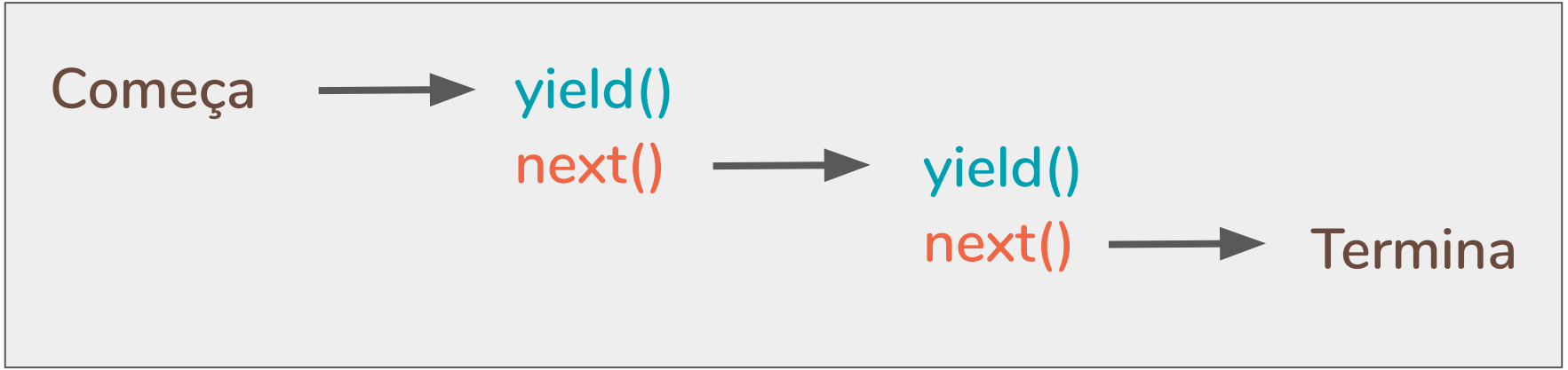
generator

_ é um iterable;
_ e também um iterator.

```
const meuGenerator = function* () {
    yield 1;
    yield 2;
    yield 3;
}();

typeof meuGenerator.next;
// "function", por que tem o método next(), logo é um iterator
typeof meuGenerator[Symbol.iterator];
// "function", então é um iterable
meuGenerator[Symbol.iterator]() === meuGenerator;
// true, por que o método que o torna um iterable
// retorna ele mesmo (um iterator)

[...meuGenerator]; // [1, 2, 3]
```

função comum

generator

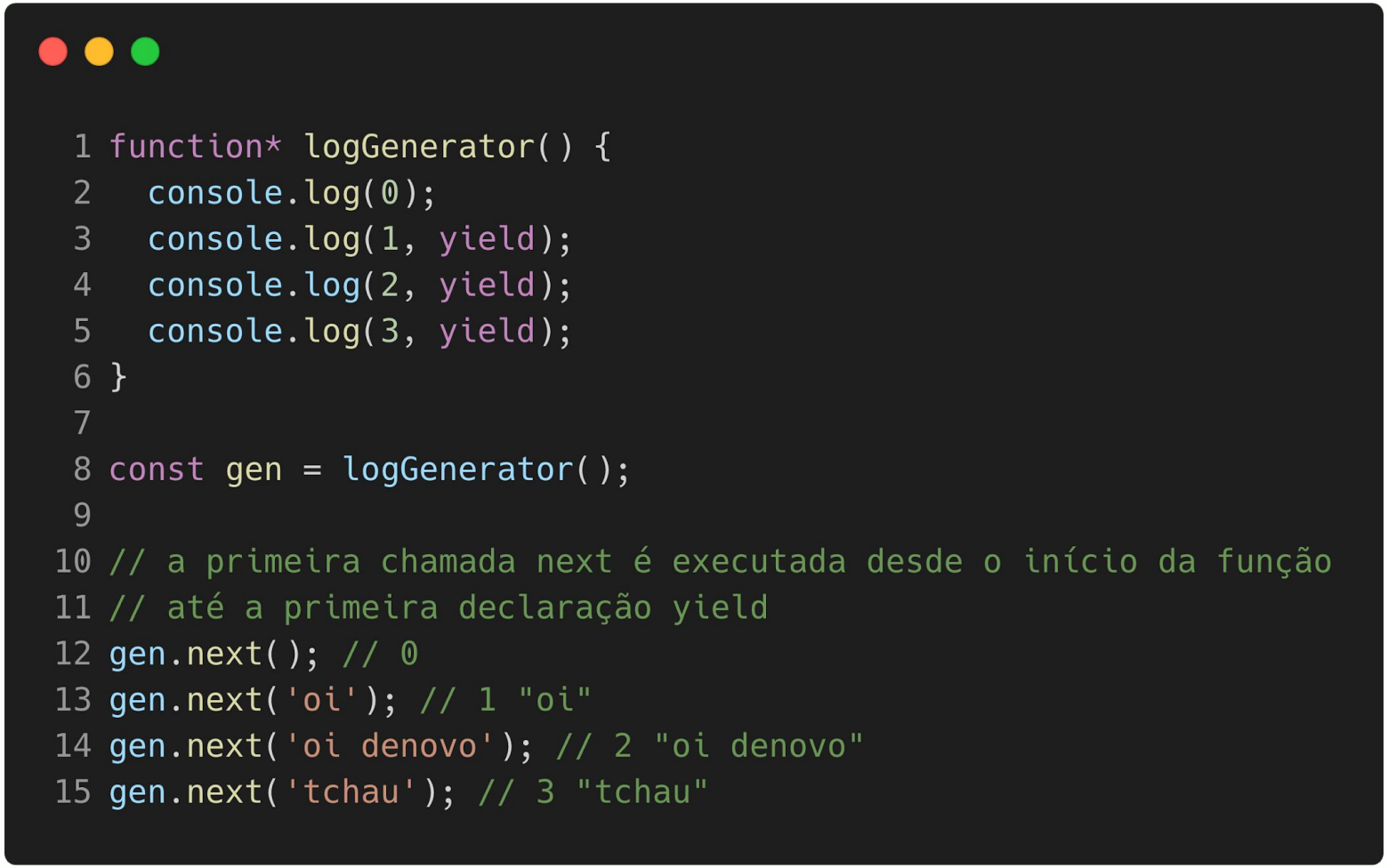


mas o **yield()** não funciona

somente "sozinho"

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

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

```js
console.log(gen.next().value); // 10
console.log(gen.next().value); // 11
console.log(gen.next().value); // 12
console.log(gen.next().value); // 13
console.log(gen.next().value); // 20
```



```
function* logGenerator() {
  console.log(0);
  console.log(1, yield);
  console.log(2, yield);
  console.log(3, yield);
}

const gen = logGenerator();

// a primeira chamada next é executada desde o início da função
// até a primeira declaração yield
gen.next(); // 0
gen.next('oi'); // 1 "oi"
gen.next('oi denovo'); // 2 "oi denovo"
gen.next('tchau'); // 3 "tchau"
```

ex. generators

- async/await
- redux-saga
- koa
- ...

referências

_ Paleta de cores
_ Carbon
_ Metaprogramming in ES6: Symbols and why they're awesome
_ Iteratores e geradores
_ Javascript ES6—Iterables and Iterators
_ Object.prototype
_ Iteration protocols
_ ES6 Generators estão mudando nosso modo de escrever JavaScript

dúvidas?
sugestões?
feedbacks?

made with ❤ by fhott

muito obrigado!

twitter.com/fhott
github.com/fhott
linkedin.com/in/fhott