FORMAÇÃO ACELERADA EM PROGRAMAÇÃO

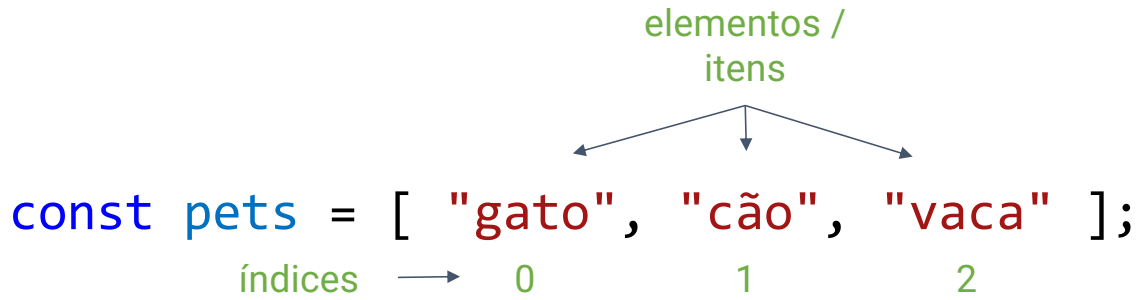
Módulo Javascript O.O.
Semana 3 e 4

Raoni Kulesza

INSTITUIÇÃO EXECUTORA
SOFTEX RECIFE
MCTI FUTURO
COORDENADORA
Softex
APOIO
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO
GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO

Arrays em Javascript

É uma coleção ordenada de itens.



Arrays em Javascript

Características dos Arrays:

1. Pode conter valores de tipos mistos
2. O tamanho do array é dinâmico

```
let variosTipos = [ 1, 2.5, “gato”];
variosTipos.push(“macaco”);
console.log(variosTipos); // Saída: [1, 2.5, “gato”, “macaco”]
```

Arrays em Javascript - Acessando elementos

As matrizes são indexadas com base em zero. Isso significa que o primeiro elemento do array começa no índice zero.

Por exemplo:

```
let pets = [“gato”, “cão”];
console.log(pets[0]); // Saída: gato
```

Arrays em Javascript - Outros métodos

Método length:

retorna o número de elementos de um array.

Por exemplo:

```
let num = [1, 2, 3, 4];
console.log(num.length); // Saída: 4
```

Arrays em Javascript - Outros métodos

Método push:

Ele adiciona elementos ao final do array.

Por exemplo:

```
let num = [1, 2, 3];
console.log(num.push(4)); // Saída: [1, 2, 3, 4]
```

Arrays em Javascript - Outros métodos

Método pop:

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

Por exemplo:

```
let num = [1, 2, 3, 4];
let numeroRemovido = num.pop();
console.log(num) // Saída: [1, 2, 3]
console.log(numeroRemovido) // Saída: 4
```

Arrays em Javascript - Outros métodos

Método shift:

Ele remove o primeiro elemento e retorna o elemento removido de um array.

Por exemplo:

```
let num = [1, 2, 3, 4];
let numeroRemovido = num.shift();
console.log(num) // Saída: [2, 3, 4]
console.log(numeroRemovido) // Saída: 1
```

Arrays em Javascript - Outros métodos

Método unshift:

Ele adiciona elementos no início do array.

Por exemplo:

```
let num = [1, 2, 3, 4];
console.log(num.unshift(0)) // Saída: [0, 1, 2, 3, 4]
```

Arrays em Javascript - Outros métodos

Método sort:

Ele classifica os itens de um array.

Por exemplo:

```
let num = [0, 2, 4, 1];
console.log(num.sort((a, b) => a - b)) // Saída: [0, 1, 2, 4]
console.log(num.sort((a, b) => b - a)) // Saída: [4, 2, 1, 0]
```

Arrays em Javascript - Outros métodos

Método reverse:

Ele inverte a ordem dos itens de um array.

Por exemplo:

```
let num = [3, 2, 4, 1];
console.log(num.reverse()) // Saída: [1, 4, 2, 3]
let num = [1, 2, 3, 4];
console.log(num.reverse()) // Saída: [4, 3, 2, 1]
```

Principais métodos de Arrays

- **push():** Adiciona um ou mais elementos ao final do array.
- **pop():** Remove e retorna o último elemento do array.
- **shift():** Remove e retorna o primeiro elemento do array.
- **unshift():** Adiciona um ou mais elementos no início do array.
- **concat():** Combina dois ou mais arrays, criando um novo array resultante.
- **join():** Converte todos os elementos do array em uma única string, separados por um delimitador.
- **slice():** Retorna uma cópia de parte do array, definida por índices de início e fim.

Principais métodos de Arrays

- **splice():** Altera o conteúdo do array, adicionando, removendo ou substituindo elementos.
- **indexOf():** Retorna o índice da primeira ocorrência de um elemento no array.
- **lastIndexOf():** Retorna o índice da última ocorrência de um elemento no array.
- **forEach():** Executa uma função para cada elemento do array.
- **map():** Cria um novo array com os resultados da aplicação de uma função a cada elemento.
- **filter():** Cria um novo array contendo todos os elementos que passam por um teste.
- **reduce():** Aplica uma função para reduzir o array a um único valor, acumulando os resultados.

Principais métodos de Arrays

- **sort((a, b) => a - b):** Ordena os elementos do array com números(modificando o array original).
- **sort():** Ordena os elementos do array com strings(modificando o array original).
- **reverse():** Inverte a ordem dos elementos no array.
- **length:** Propriedade que retorna o número de elementos no array.
- **includes():** Verifica se um elemento específico está presente no array.
- **some():** Verifica se pelo menos um elemento do array atende a uma condição.
- **every():** Verifica se todos os elementos do array atendem a uma condição.

Enums - Legibilidade e Intenção Clara

Enums fornecem nomes descritivos para valores, o que torna o código mais legível e fácil de entender. Isso ajuda a comunicar a intenção do código de maneira mais clara.

```
enum DiaDaSemana {
  Segunda,
  Terca,
  Quarta,
  Quinta,
  Sexta,
  Sabado,
  Domingo,
}
const diaHoje: DiaDaSemana = DiaDaSemana.Sexta;
```

Enums - Prevenção de Erros

O uso de enums reduz a probabilidade de erros de digitação, uma vez que os valores permitidos são limitados às constantes definidas na enumeração.

```
enum TipoDeUsuario {
  Administrador,
  UsuarioComum,
  Convidado,
}
function verificarPermissao(usuario: TipoDeUsuario): void {
  if (usuario === TipoDeUsuario.Administrador) {
    // Realizar ações administrativas
  } else {
    // Acesso restrito para outros tipos de usuário
  }
}
```

Enums - Facilidade na Manutenção

Se você precisar adicionar, remover ou alterar valores, a enumeração centraliza essas alterações em um único local, facilitando a manutenção.

```
enum EstadoPedido {
  Pendente,
  Processando,
  Concluido,
  Cancelado,
}
```

Enums - Autocompletar e Sugestões

A utilização de enums permite que a IDE forneça autocompletar e sugestões contextuais enquanto você escreve o código.

Enums - Refatoração Simplificada

Se você precisar alterar os valores representados pelos enums, poderá fazer isso centralmente, sem precisar vasculhar todo o código em busca de referências.

Enums - Em Resumo

No geral, as enumerações são úteis quando você tem um conjunto fixo de valores relacionados e quer atribuir significado semântico a esses valores. No entanto, é importante usá-las com moderação e considerar se uma abordagem diferente, como constantes ou mapeamentos, poderia ser mais adequada em determinadas situações.

Javascript O.O / ES5

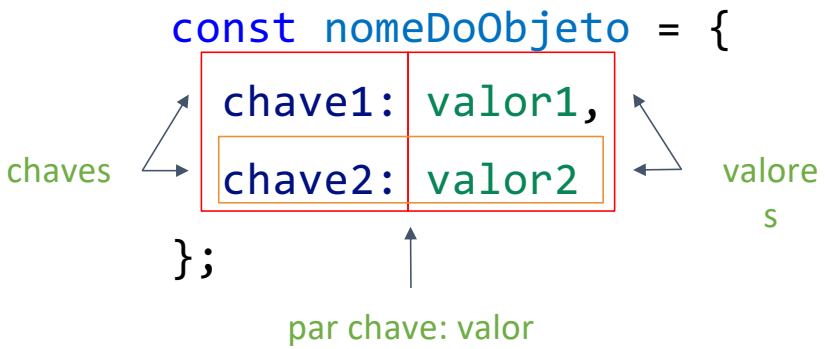
JavaScript e Orientação a Objetos

Objeto JavaScript é um tipo de dados não primitivo que permite armazenar várias coleções de dados.

Se vocês estiverem familiarizados com outras linguagens de programação, os objetos JavaScript são um pouco diferentes. Você não precisa criar classes para criar objetos.

Declaração de objeto JavaScript

Sintaxe:



Declaração de objeto JavaScript

O exemplo abaixo é um objeto chamado **pessoa** que armazena valores como string e number

```
const pessoa = {
  nome: "João",
  idade: 20
};
console.log(typeof aluno); // object
```

Declaração de objeto JavaScript

Também podemos definir um objeto em uma única linha.

```
const pessoa = { nome: 'João', idade: 20 };
```

No exemplo acima, **nome** e **idade** são chaves e **João** e **20** são valores, respectivamente.

Declaração de objeto JavaScript

Também podemos definir um objeto em uma única linha.

```
const pessoa = { nome: 'João', idade: 20 };
```

No exemplo acima, **nome** e **idade** são chaves e **João** e **20** são valores, respectivamente.

Existem mais maneiras de declarar um objeto em Javascript:

- usando objeto literal(como vimos aqui)
- usando instância de objeto diretamente(new Object())
- usando a função construtora(function Pessoa())

Propriedades do objeto JavaScript

Em JavaScript, os pares "chave: valor" são chamados de propriedades.

Por exemplo:

```
const pessoa = {
  nome: "João",
  idade: 20
```

Onde, nome: “João” e idade: 20 são propriedades.

Acessando Propriedades do Objeto

1. **Usando notação de ponto**

```
const pessoa = {
  nome: "João",
  turma: 20,
};
// acessando propriedade
console.log(pessoa.nome);
```

Acessando Propriedades do Objeto

1. **Usando notação de colchetes**

```
const pessoa = {
  nome: "João",
  turma: 20,
};
// acessando propriedade
console.log(pessoa["nome"]);
```

Objetos aninhados JavaScript

Um objeto também pode conter outro objeto.

Por exemplo:

```
const aluno = {
  nome: "João",
  idade: 20,
  notas: {
    ciência: 70,
    matemática: 75,
  },
};
console.log(aluno.notas); // {ciências: 70, matemática: 75}
```

Métodos de objetos JavaScript

Um objeto também pode conter uma função.

Por exemplo:

```
const pessoa = {
  nome: "Joana",
  idade: 30,
  //usando função como valor
  saudacao: function() { console.log("olá") }
}
pessoa.saudacao(); // olá
```

chave → `saudacao:` `function() { console.log("olá") }` ← valor

Portanto, basicamente, o método JavaScript é uma propriedade de objeto que possui um valor de função.

Acessando Métodos de Objeto

Notação de ponto:

```
const pessoa = {
  nome: "Joana",
  idade: 30,
  saudacao: function() { console.log("olá") }
}
console.log(pessoa.nome); // acessando a propriedade
pessoa.saudacao(); // acessando método
```

Se tentar acessar o método apenas com "`pessoa.saudacao`", irá obter uma definição de função.

```
[Function: saudacao]
```

Métodos integrados de JavaScript

Em JavaScript, existem muitos métodos integrados.

O método **parseInt()** é usado para converter o valor da string numérica em um valor inteiro.

```
let numero = "23.32";
let resultado = parseInt(numero);
console.log(resultado); // 23
```

Busque mais sobre métodos integrados na documentação do Javascript.

Métodos integrados de JavaScript

Também podemos adicionar um método em um objeto.

```
let aluno = {}; // criando um objeto
aluno.nome = "Paulo"; // adicionando uma propriedade

aluno.saudacao = function () { // adicionando um método
  console.log("Olá");
};
aluno.saudacao(); // acessando um método
```

Palavra-chave this

Para acessar uma propriedade de um objeto de dentro de um método do mesmo objeto, precisamos usar a palavra-chave this.

```
const pessoa = {
  nome: "João",
  idade: 20,
  // acessando a propriedade name usando this.name
  saudacao: function () {
    console.log("O nome é " + this.nome);
  },
};
pessoa.saudacao();
```

Palavra-chave this

No entanto, a função dentro de um objeto pode acessar sua variável de maneira semelhante a uma função normal.

```
const pessoa = {
  nome: "João",
  idade: 20,
  saudacao: function () {
    let sobrenome = "Silva";
    console.log("O nome é " + this.nome + " " + sobrenome);
  },
};
pessoa.saudacao();
```

Função construtora JavaScript

Em JavaScript, uma função construtora é usada para criar objetos.

função construtora

```
function Pessoa() {
    this.nome = "João",
    this.idade = 20
}
//cria um objeto
const pessoa = new Pessoa();
```

palavra-chave

Criando vários objetos com a função construtora

```
function Pessoa() {
  this.nome = "João",
  this.idade = 20,
  this.saudacao = function () {
    console.log("Olá");
  }
}
const pessoa1 = new Pessoa();
const pessoa2 = new Pessoa();

console.log(pessoa1.nome);
console.log(pessoa2.nome);
```

criando objetos

acessando propriedades

Palavra-chave this na função construtora

Quando a palavra-chave **this** é usada em uma função construtora, refere-se ao objeto quando o objeto é criado. Portanto, quando um objeto acessa as propriedades, ele pode acessar diretamente a propriedade como pessoa1.nome

```
function Pessoa() {
    this.nome = "João",
}
// criando objeto
const pessoa1 = new Pessoa();
// acessando propriedade nome
console.log(pessoa1.nome);
```

Parâmetros de uma função construtora

parâmetros recebidos

```
function Pessoa (nome_pessoa, idade_pessoa, genero_pessoa) {
    this.nome = nome_pessoa,
    this.idade = idade_pessoa,
    this.genero = genero_pessoa,
    this.saudacao = function () {
        return ("Oi " + this.nome);
    }
}
const pessoa1 = new Pessoa("João", 23, "masculino");
const pessoa2 = new Pessoa("Paula", 25, "feminino");

console.log(pessoa1.nome); // "João"
console.log(pessoa2.nome); // "Paula"
```

atribuindo valores de parâmetro ao objeto de chamada

argumentos passados

cada objeto possui diferentes propriedades

função construtora versus literal de objeto

Objeto Literal geralmente é usado para criar um único objeto, já a função construtora é útil se você deseja criar vários objetos.

```
const pessoa = {
  nome: "João",
};
```

```
function Pessoa() {
    this.nome = "João",
}

const pessoa1 = new Pessoa();
const pessoa2 = new Pessoa();
```

função construtora versus literal de objeto

Cada objeto criado a partir da função construtora é único. Você pode ter as mesmas propriedades da função construtora ou adicionar uma nova propriedade a um objeto específico.

```
function Pessoa() {
    this.nome = "João",
}

const pessoa1 = new Pessoa();
const pessoa2 = new Pessoa();

pessoa1.idade = 23;
```

Agora, a propriedade **idade** é exclusiva do objeto **pessoa1** e não está disponível para o objeto **pessoa2**.

função construtora versus literal de objeto

No entanto, se um objeto for criado como literal e se uma variável for definida com esse valor de objeto, quaisquer alterações no valor da variável irá alterar o objeto original.

```
let pessoa = {
  nome: "Paula"
}

console.log(pessoa.nome); // Paula
let aluno = pessoa;

aluno.nome = "João";

console.log(pessoa.nome); // João
```

altera a propriedade do objeto

altera a propriedade do **objeto original**

Adicionando propriedades e métodos em um objeto

```
function Pessoa() {
  this.nome = "João",
  this.idade = 23
}
let pessoa1 = new Pessoa();
let pessoa2 = new Pessoa();

pessoa1.genero = "masculino";

pessoa1.saudacao = function () {
  console.log("Olá");
};

pessoa1.saudacao(); // Olá
pessoa2.saudacao();
```

adicionando propriedade ao objeto **pessoa1**

adicionando método ao objeto **pessoa1**

TypeError - **pessoa2** não possui **método saudacao()**

Construtores integrados em JavaScript

JavaScript também possui construtores integrados. Alguns deles são:

```
let a = new Object(); // Um   novo objeto "Object"
let b = new String(); // Um   novo objeto "String"
let c = new Number(); // Um   novo objeto "Número"
let d = new Boolean(); // Um   novo objeto "Lógico"
```

Construtores integrados em JavaScript

Exemplos:

```
const nome = new String("João");
console.log(nome); // "João"

const numero = new Number(57);
console.log(numero); // 57

const contar = new Boolean(true);
console.log(contar); // true
```

Construtores integrados em JavaScript

Recomenda-se usar tipos de dados primitivos e criá-los de forma normal, como:

```
const nome = "João"; const numero = 57; e const contar = true;
```

Você não deve declarar strings, números e valores booleanos como objetos porque eles tornam o programa **mais lento**.

Getters e setters em JavaScript

Em JavaScript, existem dois tipos de propriedades de objeto:

- Propriedades de dados
- Propriedades do acessador

Métodos getters

Métodos getters são usados para acessar as propriedades de um objeto

```
const aluno = {
  nome: "Pedro",
  get getNome() {
    return this.nome;
  },
};
console.log(aluno.nome);

console.log(aluno.getNome);

console.log(aluno.getNome());
```

propriedade de dados

propriedade do acessador(getter)

acessando propriedade de dados

acessando métodos getter

TypeError - tentando acessar como um método

Métodos setters

Métodos setters são usados para alterar os valores de um objeto

```
const aluno = {
  nome: "Pedro",
  set setNome(novo_nome) {
    this.nome = novo_nome;
  },
};
aluno.setNome = "Maria";

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

propriedade de dados

propriedade do acessador(setter)

acessando propriedade de dados

alterar (set) propriedade do objeto usando um setter

Javascript O.O / ES6

Criando classe JavaScript

A classe JavaScript é semelhante à função construtora.

```
// função construtora
function Pessoa() {
  this.nome = "João",
  this.idade = 20
}
//cria um objeto
const pessoa = new Pessoa();
```

* palavra-chave

```
//criando uma classe
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }
}
//cria um objeto
const pessoa = new Pessoa("João", 23);
```

* A palavra-chave **class** é usada para criar uma classe

Criando classe JavaScript

```
class Pessoa {
  constructor(nome, idade) {
    this.nome = nome;
    this.idade = idade;
  }
}
const pessoa1 = new Pessoa("João", 20);
const pessoa2 = new Pessoa("José", 51);

console.log(pessoa1.nome); // João
console.log(pessoa2.nome); // José
```

Aqui **pessoa1** e **pessoa2** são objetos da classe **Pessoa**

Métodos de classe Javascript

Ao usar a função construtora, você define métodos como:

```
function Pessoa(nome) {
  this.nome = nome;

  this.saudacao = function () {
    return "Olá" + " " + this.nome;
  };
}
```

atribuindo valores de parâmetro ao objeto chamador

definindo método

Métodos de classe Javascript

É fácil definir métodos em uma classe JavaScript. Você simplesmente fornece o nome do método seguido de ( ).

```
class Pessoa {
  constructor(nome) {
    this.nome = nome;
  }
  saudacao() {                          definindo método
    console.log(`Olá ${this.nome}`);
  }
}
let pessoa1 = new Pessoa("João");
console.log(pessoa1.nome);              acessando a propriedade
pessoa1.saudacao();                     acessando método
```

Métodos Getters e Setters

Os métodos getters obtêm o valor de um objeto e os métodos setters definem o valor de um objeto.

Você usa a palavra-chave **get** para métodos **getters** e **set** para métodos **setters**.

Métodos Getters e Setters

Exemplo:

```
class Pessoa {
  constructor(nome) {
    this.nome = nome;
  }
  get nomePessoa() {
    return this.nome;
  }
  set nomePessoa(x) {
    this.nome = x;
  }
}
let pessoa1 = new Pessoa("José");
console.log(pessoa1.nome);
pessoa1.nomePessoa = "Maria";
console.log(pessoa1.nome); // Maria
```

alterando o valor da propriedade nome

Hoisting (“içamento”, ou “elevação”)

Permite que você execute funções antes das suas declarações. Saiba mais...

```
console.log(soma(2, 5));
function soma(a, b) {
  return a + b;
}
```

Hoisting

```
function soma(a, b) {
  return a + b;
}
console.log(soma(2, 5));
```

Hoisting (“içamento”, ou “elevação”)

Entretanto, ao contrário das funções e outras declarações JavaScript, o Hoisting não funciona em outros tipos de código.

Uma classe deve ser definida antes de usá-la.

```
const p = new Pessoa(); ← ReferenceError

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

'use strict'

O modo Estrito é um modo em Javascript que captura erros que falham silenciosamente e acionam **erros de execução**. Todo o código dentro da classe fica automaticamente no modo estrito.

```
class Pessoa {
  constructor() {
    nomePessoa = "João";
    this.nome = nomePessoa;
  }
}
let p = new Pessoa();
```

Variável **nomePessoa** não foi declarada.

ReferenceError: Can't find variable: nomePessoa

Herança de classe JavaScript

A herança permite que uma classe herde propriedades e métodos de outra classe e também permite que mais propriedades e/ou métodos sejam adicionados, o que é bastante útil que permite a reutilização de código. Para usar herança de classe, você usa a palavra-chave **extends.**

Herança de classe JavaScript

Exemplo:

```
class Pessoa {
  constructor(nome) {
    this.nome = nome;
  }
  saudacao() {
    console.log(`Olá ${this.nome}`);
  }
}
class Aluno extends Pessoa {}

let aluno1 = new Aluno("José");
aluno1.saudacao();
```

classe pai

herdando da classe pai

Palavra-chave super()

A palavra-chave **super** usada dentro de uma classe filha denota sua classe pai.

```
class Pessoa {
  constructor(nome) {
    this.nome = nome;
  }
  saudacao() {
    console.log(`Olá ${this.nome}`);
  }
}
```

```
class Aluno extends Pessoa {
  constructor(nome) {
    console.log("Criando turma de aluno");
    super(nome);
  }
}

let aluno1 = new Aluno("José");
aluno1.saudacao();
```

chama o construtor da superclasse e passa o parâmetro **nome**.

Substituindo método ou propriedade

Se uma classe filha tiver o mesmo método ou nome de propriedade da classe pai, ela usará o método e a propriedade da classe filha. Este conceito é chamado de **overriding**.

```
class Pessoa {
  constructor(nome) {
    this.nome = nome;
    this.ocupacao = "Desempregado";
  }
  saudacao() {
    console.log(`Olá ${this.nome}`);
  }
}
```

```
class Aluno extends Pessoa {
  constructor(nome) {
    super(nome);
    this.ocupacao = "Estudante";
  }
  saudacao() {
    console.log(`Olá ${this.nome}`);
    console.log(`Ocupação: ${this.ocupacao}`);
  }
}
```

Subreescrevendo a propriedade **ocupacao**.

Principais métodos de Objetos

- **Object.keys():** Retorna um array contendo as chaves (propriedades) enumeráveis de um objeto.
- **Object.values():** Retorna um array contendo os valores das propriedades enumeráveis de um objeto.
- **Object.entries():** Retorna um array contendo pares chave-valor (como arrays) das propriedades enumeráveis de um objeto.
- **Object.assign():** Copia as propriedades de um ou mais objetos para um objeto de destino.
- **Object.hasOwnProperty():** Verifica se um objeto possui uma propriedade própria (não herdada).

Principais métodos de Objetos

- **Object.is():** Compara se dois valores são iguais, usando o algoritmo SameValueZero.
- **Object.freeze():** Congela um objeto, tornando suas propriedades não modificáveis.
- **Object.seal():** Sela um objeto, impedindo a adição ou remoção de propriedades, mas permitindo a modificação de propriedades existentes.
- **Object.getPrototypeOf():** Retorna o protótipo de um objeto.
- **Object.setPrototypeOf():** Define o protótipo de um objeto.
- **Object.create():** Cria um novo objeto com o protótipo especificado.

Principais métodos de Objetos

- **Object.values():** Retorna um array com os valores das propriedades de um objeto.
- **Object.entries():** Retorna um array com pares chave-valor (como arrays) das propriedades de um objeto.
- **Object.fromEntries():** Cria um objeto a partir de um array de pares chave-valor.
- **Object.getOwnPropertyNames():** Retorna um array com todas as propriedades de um objeto, enumeráveis ou não.
- **Object.getOwnPropertyDescriptors():** Retorna um objeto contendo descritores de propriedades de um objeto.

FORMAÇÃO ACELERADA EM PROGRAMAÇÃO

INSTITUIÇÃO EXECUTORA

SOFTEX RECIFE

COORDENADORA

MCTI FUTURO

Softex

APOIO

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO

GOVERNO FEDERAL BRASIL UNIÃO E RECONSTRUÇÃO