

# FrontEnd

# FrontEnd - JavaScript

## Orientação a Objetos

A orientação a objetos em JavaScript é um paradigma de programação que permite modelar o mundo real em termos de objetos.

Os objetos podem conter dados e códigos relacionados, que representam informações sobre o que você está tentando modelar e a funcionalidade ou o comportamento que você deseja ter.

**Objetos:** Em JavaScript, um objeto é uma coleção de propriedades que têm um nome (chave) e um valor associado. Por exemplo, um objeto **carro** pode ter propriedades como **cor**, **marca** e **modelo**, onde cada uma tem um valor correspondente.

**Classes e Protótipos:** As classes são modelos para criar objetos. Elas definem as propriedades e métodos que os objetos podem ter.

**Métodos:** Os métodos são funções definidas dentro de uma classe que permitem que os objetos executem ações. Eles são usados para manipular os dados de um objeto e podem receber parâmetros para realizar tarefas específicas.

**Propriedades:** As propriedades são variáveis que armazenam dados dentro de um objeto. Elas representam as características de um objeto e podem ser de vários tipos, incluindo números, strings, booleanos e até mesmo outros objetos.

## Orientação a Objetos

**Encapsulamento:** O encapsulamento é o conceito de agrupar propriedades e métodos dentro de um objeto e controlar o acesso a eles. Isso ajuda a organizar o código e proteger os dados de serem modificados de maneira inesperada.

**Herança:** A herança é um mecanismo que permite que um objeto herde propriedades e métodos de outro objeto. Isso promove a reutilização de código e a criação de hierarquias de objetos.

**Polimorfismo:** O polimorfismo permite que objetos diferentes respondam ao mesmo método de maneira diferente. Isso é alcançado por meio da substituição de métodos em classes derivadas.

# FrontEnd - JavaScript

## Orientação a Objetos

Na programação front-end, lidamos frequentemente com interfaces de usuário complexas e interações dinâmicas. A orientação a objetos nos ajuda a organizar nosso código de forma mais modular e reutilizável. Por exemplo, podemos criar objetos representando diferentes elementos da interface do usuário e definir métodos para manipulá-los de forma independente.

Além disso, muitas estruturas e bibliotecas JavaScript amplamente utilizadas, como React e Angular, são baseadas em conceitos de orientação a objetos. Portanto, entender esses conceitos é fundamental para se tornar um desenvolvedor front-end competente.

# FrontEnd - JavaScript

## Orientação a Objetos

**Objeto** : representação de algo do mundo real.

```
<script language="JavaScript">
console.log('-> Classe e Objeto')
class Carro {
  constructor() {
    this.marca = "Toyota";
    this.modelo = "Corolla";
    this.ano = 2022;
  }

  ligar() {
    alert("Motor ligado. Bora dar uma volta!");
  }
}
let carro = new Carro();
alert(carro.marca);
alert(carro.modelo);
alert(carro.ano);
carro.ligar();
</script>
```

# FrontEnd - JavaScript

## Orientação a Objetos

**Classes:** são o molde de como as informações devem ser estruturadas. Para que eu consiga manipular corretamente uma classe eu conto com um construtor, e a cada vez que eu uso esse molde, devo usar o “**New**” para fazer a chamada da classe e a passagem de parâmetros .

```
class Pessoa {  
  constructor(nome, idade){  
    this.nome = nome  
    this.idade = idade  
  }  
  apresentar(){  
    console.log(`Olá meu nome é ${this.nome} e tenho ${this.idade}`)  
  }  
}  
  
let pessoa1 = new Pessoa("Fernanda", 15)  
let pessoa2 = new Pessoa("Maria", 20)  
pessoa1.apresentar()  
pessoa2.apresentar()
```

-> Classe e Objeto

Olá meu nome é Fernanda e tenho 15

Olá meu nome é Maria e tenho 20

```
class Pessoa {  
  constructor(nome, idade) {  
    this._nome = nome;  
    this._idade = idade;  
  }  
  
  get nome() {  
    return this._nome;  
  }  
  
  set nome(novoNome) {  
    this._nome = novoNome;  
  }  
  
  get idade() {  
    return this._idade;  
  }  
  
  set idade(novaIdade) {  
    this._idade = novaIdade;  
  }  
}  
  
let pessoa = new Pessoa("João", 30);  
console.log(pessoa.nome); // Saída: "João"  
pessoa.nome = "Pedro";  
console.log(pessoa.nome); // Saída: "Pedro"
```

## Orientação a Objetos

**Encapsulamento:** uma forma controlada de tratar os dados.

Com o encapsulamento, os dados de um objeto são protegidos de acesso não autorizado e modificações inesperadas. Isso é alcançado usando técnicas como métodos de acesso (getters) e métodos de modificação (setters) para controlar como os dados são lidos e escritos. Por exemplo, podemos validar os dados antes de atribuí-los a uma propriedade, garantindo que apenas valores válidos sejam aceitos.



# FrontEnd - JavaScript Orientação a Objetos

```
class Produto {
  constructor(nome, preco) {
    this._nome = nome;
    this._preco = preco;
  }

  // Getter para obter o nome do produto
  get nome() {
    return this._nome;
  }

  // Setter para definir o nome do produto
  set nome(novoNome) {
    this._nome = novoNome;
  }

  // Getter para obter o preço do produto
  get preco() {
    return this._preco;
  }

  // Setter para definir o preço do produto, incluindo uma validação básica
  set preco(novoPreco) {
    if (novoPreco > 0) {
      this._preco = novoPreco;
    } else {
      console.error("Preço inválido.");
    }
  }
}
```

```
// Criando uma instância de Produto
let produto = new Produto("Camiseta", 25.99);

// Acessando os getters
console.log(produto.nome); // Saída: "Camiseta"
console.log(produto.preco); // Saída: 25.99

// Modificando os valores usando os setters
produto.nome = "Calça";
produto.preco = 39.99;

// Acessando novamente os getters após a modificação
console.log(produto.nome); // Saída: "Calça"
console.log(produto.preco); // Saída: 39.99

// Tentando atribuir um valor inválido para o preço
produto.preco = -10; // Saída no console: "Preço inválido."
console.log(produto.preco); // Saída: 39.99 (não foi alterado devido à validação)
```

**Herança:** consigo obter todas as características de uma classe mãe

```
class Veiculo {
  constructor(marca, modelo) {
    this.marca = marca;
    this.modelo = modelo;
  }

  acelerar() {
    console.log(`${this.marca} ${this.modelo} está acelerando.`);
  }

  frear() {
    console.log(`${this.marca} ${this.modelo} está freando.`);
  }
}

// Subclasse Carro, herda de Veiculo
class Carro extends Veiculo {
  constructor(marca, modelo, numPortas) {
    super(marca, modelo);
    this.numPortas = numPortas;
  }
}
```

```
  abrirPortas() {
    console.log(`Abrindo ${this.numPortas} portas do ${this.marca} ${this.modelo}.`);
  }
}

// Subclasse Bicicleta, herda de Veiculo
class Bicicleta extends Veiculo {
  pedalar() {
    console.log(`Pedalando a ${this.marca} ${this.modelo}.`);
  }
}

// Criando instâncias de Carro e Bicicleta
let carro = new Carro("Toyota", "Corolla", 4);
let bicicleta = new Bicicleta("Caloi", "Elite");

// Usando métodos das classes
carro.acelerar(); // Saída: "Toyota Corolla está acelerando."
carro.abrirPortas(); // Saída: "Abrindo 4 portas do Toyota Corolla."

bicicleta.acelerar(); // Saída: "Caloi Elite está acelerando."
bicicleta.pedalar(); // Saída: "Pedalando a Caloi Elite."
```

# FrontEnd - JavaScript

## Polimorfismo : O

polimorfismo é um conceito fundamental na programação orientada a objetos que se refere à capacidade de objetos de diferentes classes responderem ao mesmo método de maneira diferente. Em outras palavras, o polimorfismo permite que um método em uma classe base seja implementado de forma diferente em cada classe derivada, de acordo com as necessidades específicas de cada classe.

```
class Forma {  
  desenhar() {  
    console.log("Desenhando uma forma genérica.");  
  }  
}  
  
class Circulo extends Forma {  
  desenhar() {  
    console.log("Desenhando um círculo.");  
  }  
}  
  
class Quadrado extends Forma {  
  desenhar() {  
    console.log("Desenhando um quadrado.");  
  }  
}  
  
let forma1 = new Circulo();  
let forma2 = new Quadrado();  
  
forma1.desenhar(); // Saída: "Desenhando um círculo."  
forma2.desenhar(); // Saída: "Desenhando um quadrado."
```

```
console.log('-> Array de objetos')
```

```
let hqs = [  
  {  
    titulo: 'V de Vingança',  
    volume: 'Único',  
    autor: 'Alan Moore',  
    publicacao: 1982,  
    preco: 100.00  
  },  
  {  
    titulo: 'Akira',  
    volume: '1',  
    autor: 'Katsuhiro Otomo',  
    publicacao: 1980,  
    preco: 90.00  
  },  
  {  
    titulo: 'Maus',  
    volume: 'Único',  
    autor: 'Art Spiegelman',  
    publicacao: 1980,  
    preco: 85.00  
  },  
  {  
    titulo: 'Cavaleiro das Trevas',  
    volume: 'Único',  
    autor: 'Frank Miller',  
    publicacao: 1986,  
    preco: 120.00  
  },  
  {  
    titulo: 'Watchmen',  
    volume: 'Único',  
    autor: 'Alan Moore',  
    publicacao: 1986,  
    preco: 95.00  
  },  
  {  
    titulo: 'Absolute Sandman',  
    volume: '5',  
    autor: 'Neil Gaiman',  
    publicacao: 1989,  
    preco: 250.00  
  },  
]
```

# FrontEnd - JavaScript

## Array de Objetos

```
// Lê e apresenta todos os autores  
const autores = hqs.map( (hqs) => hqs.autor )  
  
console.log('Todos os autores')  
console.log(autores)  
console.log('\n')  
  
// Apresenta somente os títulos do autor Alan Moore  
const autor = hqs.map( (hqs) => hqs.autor == 'Alan Moore' ? hqs.titulo : '' )  
  
console.log('Títulos do autor Alan Moore')  
console.log(autor)  
console.log('\n')  
  
// Filtra somente os preços abaixo de R$ 100,00  
const precos = hqs.filter( (hqs) => hqs.preco < 100.00 ).map( (hqs) => hqs.preco )  
  
console.log('Preços abaixo de R$ 100,00')  
console.log(precos)  
console.log('\n')  
  
// Calcula a somatória dos preços  
const total = hqs.reduce( (total, hqs) => total + hqs.preco, 0 )  
  
console.log('Total preços')  
console.log(total)
```

# FrontEnd - JavaScript

## VIACEP

Consulte CEPs de todo o Brasil

Webservice gratuito de alto desempenho para consulta de Código de Endereçamento Postal (CEP) do Brasil.

### Acessando o webservice de CEP

Para acessar o webservice, um CEP no formato de **(8)** dígitos deve ser fornecido, exemplo: "01001000". Após o CEP, deve ser fornecido o tipo de retorno desejado, que deve ser "json" ou "xml".

Exemplo de consulta de CEP:  
[viacep.com.br/ws/01001000/json/](http://viacep.com.br/ws/01001000/json/)

### Validação do CEP

Quando consultado um CEP de formato inválido, exemplo: "950100100" (9 dígitos), "95010A10" (alfanumérico), "95010 10" (espaço), o código de retorno da consulta será um **400** (Bad Request). Antes de acessar o webservice, valide o formato do CEP e certifique-se que o mesmo possua **(8)** dígitos. Exemplo de como validar o formato do CEP em javascript está disponível nos exemplos abaixo.

Quando consultado um CEP de formato válido, porém inexistente, por exemplo: "99999999", o retorno conterá um valor de "erro" igual a "true". Isso significa que o CEP consultado não foi encontrado na base de dados. Veja como tratar este "erro" em javascript nos exemplos abaixo.

### Formatos de Retorno

Veja exemplos de acesso ao webservice e os diferentes tipos de retorno:

#### JSON

URL: [viacep.com.br/ws/01001000/json/](http://viacep.com.br/ws/01001000/json/)

```
{
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
  "ibge": "3550308",
  "gia": "1004",
  "ddd": "11",
  "siafi": "7107"
}
```

#### JSONP

URL: [viacep.com.br/ws/01001000/json/?callback=callback\\_name](http://viacep.com.br/ws/01001000/json/?callback=callback_name)

```
callback_name({
  "cep": "01001-000",
  "logradouro": "Praça da Sé",
  "complemento": "lado ímpar",
  "bairro": "Sé",
  "localidade": "São Paulo",
  "uf": "SP",
  "ibge": "3550308",
  "gia": "1004",
  "ddd": "11",
  "siafi": "7107"
})
```

<http://viacep.com.br>

É um webserver que permite a consultas de CEPs de forma “Rápida” e gratuita.

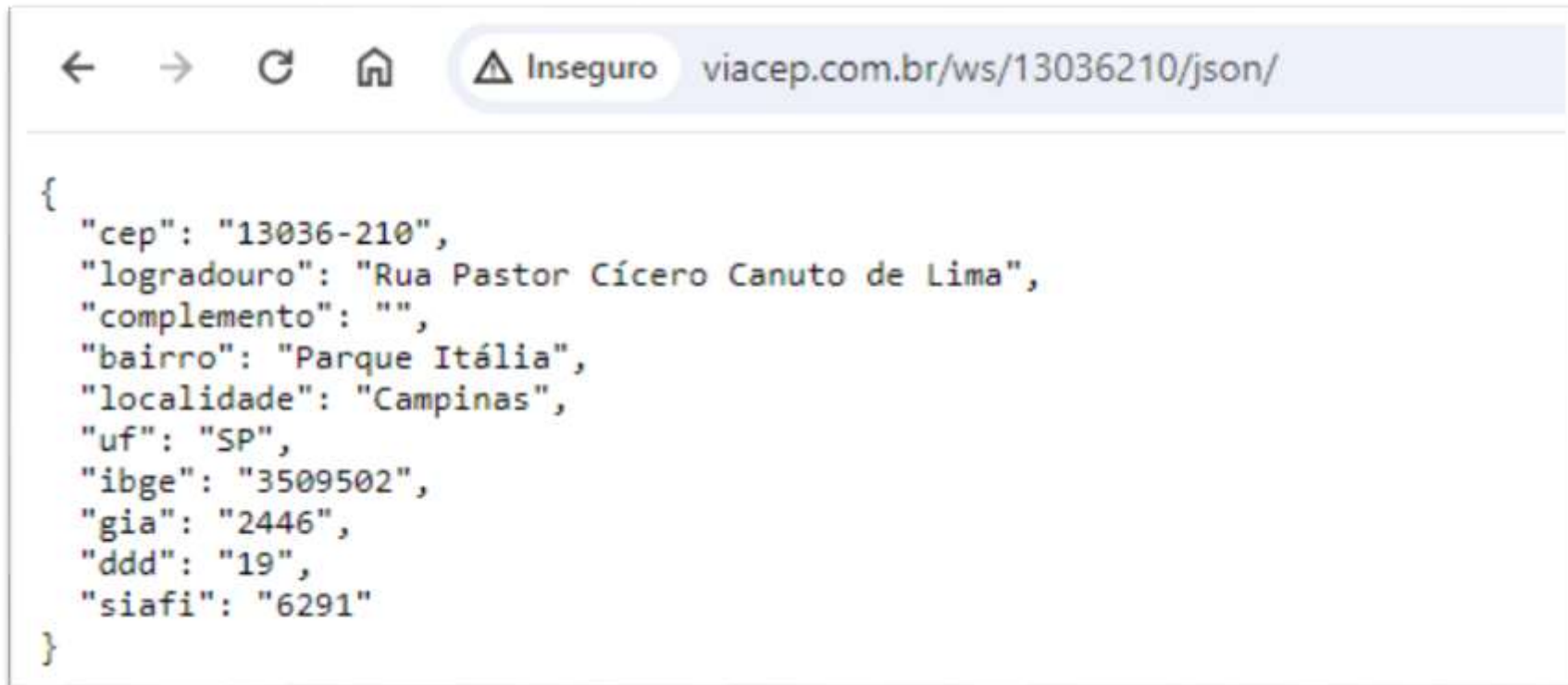
Quando começamos a fazer uma comunicação via API é necessário que ambas as pontas saibam quais são as informações necessárias, para que o resultado seja o esperado.

# FrontEnd - JavaScript

<http://viacep.com.br>

O Via CEP necessita de um cep existente para que retorne o endereço correspondente.

Colocando o endereço <http://viacep.com.br/ws/13036210/json/>, temos o resultado



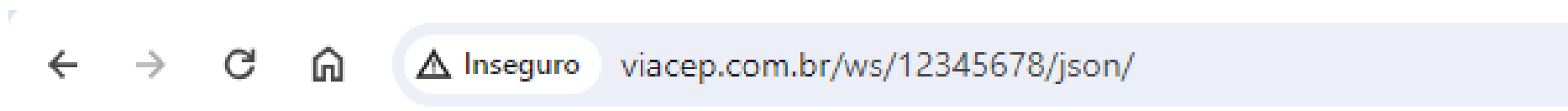
A screenshot of a web browser window. The address bar shows the URL `viacep.com.br/ws/13036210/json/` with a warning icon and the text "Inseguro". The main content area displays a JSON object representing address data for the CEP 13036-210.

```
{
  "cep": "13036-210",
  "logradouro": "Rua Pastor Cícero Canuto de Lima",
  "complemento": "",
  "bairro": "Parque Itália",
  "localidade": "Campinas",
  "uf": "SP",
  "ibge": "3509502",
  "gia": "2446",
  "ddd": "19",
  "siafi": "6291"
}
```

# FrontEnd - JavaScript

<http://viacep.com.br>

Agora se inserirmos um CEP inexistente



```
{  
  "erro": "true"  
}
```

E se inserirmos um CEP com mais dígitos do que esperado



## Http 400

Verifique a URL

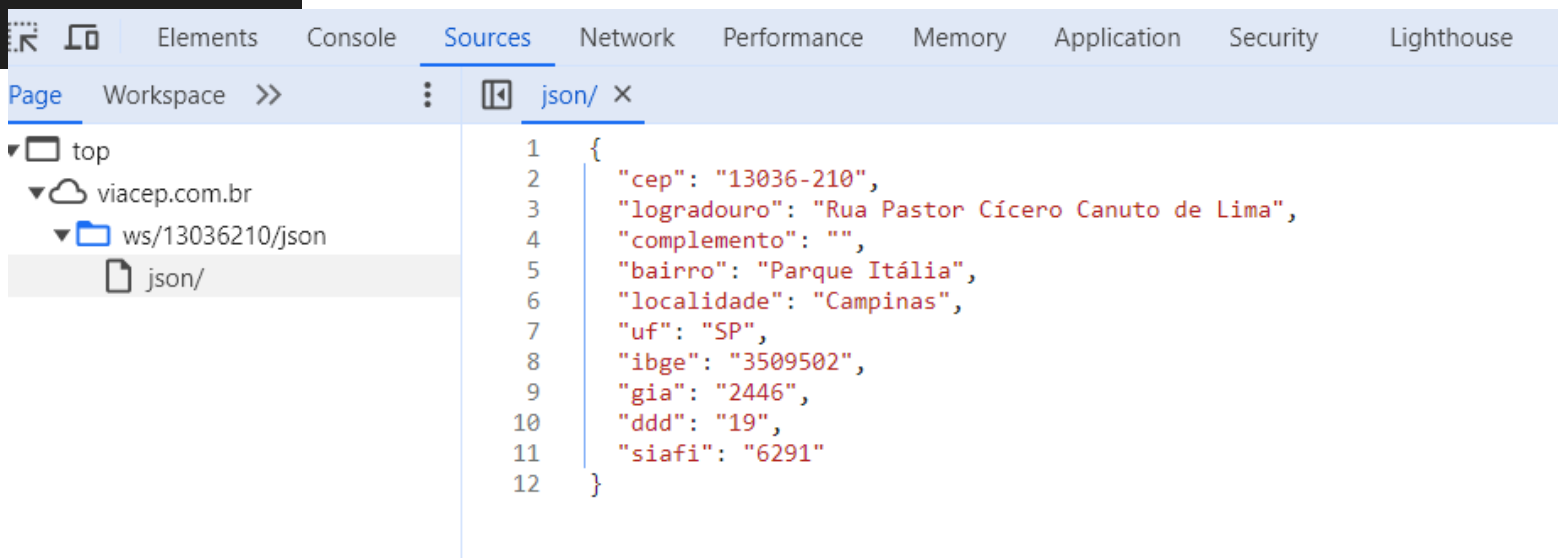
{Bad Request}

# FrontEnd - JavaScript

<http://viacep.com.br>

Mas se colocar todos os comandos corretos, os dados correspondentes ao cep são retornados.

```
const funcaoCepFetch = () => {  
  fetch('http://viacep.com.br/ws/13036210/json/')  
    .then( retorno => console.log(retorno) )  
    .catch( motivo => console.log(motivo) )  
}  
funcaoCepFetch()
```





# FrontEnd - JavaScript

<http://viacep.com.br>

Para pegar os dados do endereço e o tornar manipulável no JS

```
const funcaoCepFetch = (cep) => {  
  fetch(`http://viacep.com.br/ws/${cep}/json/`)  
    .then( retornoFetch => retornoFetch )  
    .then( retornoJSON => retornoJSON.json() )  
    .then( endereco => { console.log(`Rua: ${endereco.logradouro}`)  
                        console.log(`Bairro: ${endereco.bairro}`)  
                        console.log(`Cidade: ${endereco.localidade}`) } )  
    .catch( erro => console.log(`Ops! Ocorreu um erro na consulta (${erro})`) )  
}  
funcaoCepFetch(13036210)
```

```
Rua: Rua Pastor Cícero Canuto de Lima  
Bairro: Parque Itália  
Cidade: Campinas
```

```
>
```

# FrontEnd - JavaScript

**Usando a programação orientada a objetos:**

Criar uma classe ContaBancaria com propriedades “Saldo”, e “tipo” (corrente ou poupança).

Adicionar os métodos “Sacar” e “Depositar”.

Criar Subclasses “Conta corrente” e “Conta Poupança” que herdam de conta bancária.

Adicione um método “renderJuros()” a classe conta poupança que calcula e adiciona os juros ao saldo.

# FrontEnd - JavaScript

## **Usando a programação orientada a objetos:**

Crie um sistema que acompanhe os empréstimos e devoluções de um livro.

Para cada livro temos:

Nome

Autor

Identificador

Categoria

Quantidade de livros do mesmo título

Crie métodos que permita fazer o empréstimo e a devolução dos livros.

# FrontEnd - JavaScript

**Usando a programação orientada a objetos:**

**Sistema de Autenticação de Usuários:** Desenvolva um sistema de autenticação de usuários com classes para usuários, autenticação e métodos para registrar, fazer login e fazer logout.