

INSTITUTO FEDERAL DE MATO GROSSO DO SUL  
*CAMPUS NOVA ANDRADINA*

NOTAS DE AULA

LINGUAGEM E TÉCNICAS DE  
PROGRAMAÇÃO

Prof. Me. Luiz F. Picolo

NOVA ANDRADINA - MS

Atualizado em 14 de março de 2022

## 2 Orientação a Objetos com JavaScript

*Dedique-se aos estudos para que eles o façam  
melhor para a sociedade e para si mesmo.*

**Alvaro Granha Loregian**

Esse capítulo tem como objetivo aplicar o conhecimento adquirido anteriormente no Capítulo ?? utilizando a linguagem de programação JavaScript.

JavaScript, segundo Trasviña 2021, tem fortes capacidades de programação orientada a objetos, apesar de ocorrerem algumas discussões devido às diferenças da orientação a objetos no JavaScript em comparação com outras linguagens. Assim, no Capítulo ?? foi realizada uma introdução à programação orientada a objetos, e neste, será demonstrado os conceitos de programação orientada a objetos no JavaScript.

### 2.1 Classes

Classes em JavaScript<sup>1</sup> são introduzidas no ECMAScript<sup>2</sup> 2015 e são simplificações da linguagem para as heranças baseadas nos protótipos. A sintaxe para classes não introduz um novo modelo de herança de orientação a objetos em JavaScript. Classes em JavaScript provêm uma maneira mais simples e clara de criar objetos e lidar com herança.

O diagrama ilustra a transição da sintaxe de objetos literais para a sintaxe de classes. À esquerda, um objeto literal `Usuario` é definido com propriedades `nome` e `profissao`, e é acessado via `console.log(Usuario.nome)`. À direita, a mesma lógica é implementada usando a classe `Usuario` com um construtor, onde o objeto é criado com `new Usuario()` e acessado via `console.log(usuario.nome)`. Uma seta indica a transformação entre as duas abordagens.

```
let Usuario = {  
  nome: 'Luiz Picolo',  
  profissao: 'Professor'  
}  
  
console.log(Usuario.nome)
```

```
class Usuario {  
  constructor(){  
    this.nome = 'Luiz Picolo';  
    this.profissao =  
    'Professor'  
  }  
}  
  
usuario = new Usuario();  
console.log(usuario.nome)
```

**Figura 1** – Sintaxes de classes em JavaScript

Fonte: O Autor

Para definir uma classe é usando uma declaração de classe. Para declarar uma classe, deve-se usar a palavra reservada **class** seguida pelo nome da classe desejada.

<sup>1</sup> Para mais detalhes acesse: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Classes>>

<sup>2</sup> Para mais detalhes acesse: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>>

```
1  class Retangulo {  
2  
3  }
```

Outras forma de escrita de uma classe, também chamadas Expressões de Classes (*class expression*), podem ser usadas para fazer sua definição. Contudo, utilizaremos apenas a notação mais nova e comumente encontrada em outras linguagens, ou seja, a notação *class*.

```
1  // sem nome  
2  let Retangulo = class {  
3  
4  };  
5  
6  // nomeada  
7  let Retangulo = class Retangulo {  
8  
9  };
```

E para que possamos criar uma instância (objeto) da classe, utilizamos o seguinte trecho de código abaixo.

```
1  let retangulo = new Retangulo();
```

## 2.2 Construtor e Atributos

Como dito no Capítulo ?? as características que descrevem um objeto são chamadas na orientação a objeto de atributos. Assim, toda classe pode conter atributos que lhe dão as características necessárias para a criação do objeto.

Para que possamos criar nossos atributos podemos fazer de duas formas, uma usando funções diretas ou por meio do Construtor. Um construtor é um tipo especial de “método” para criar e iniciar um objeto criado pela classe. Só pode existir um método construtor dentro da classe. Um erro de sintaxe **SyntaxError** será lançado se a classe possui mais do que uma ocorrência do método construtor.

Para iniciarmos os atributos com o construtor usamos a palavra reservada **constructor** seguida dos valores dos atributos desejados.

```
1  class Retangulo {  
2      constructor(altura, largura) {  
3          this._altura = altura;  
4          this._largura = largura;  
5      }  
6  }
```

## 2.3 Métodos

Métodos são ações que o objeto pode realizar e são responsáveis pela troca de mensagem entre os objetos **emissores** e o **receptores**. As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos como possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, de um método deve ser suficiente conhecer apenas sua especificação, sem necessidade de saber detalhes de como a funcionalidade que ele executa é implementada<sup>3</sup>.



**Figura 2** – Troca de mensagens entre objetos.

Fonte: O Autor

Assim, para definir um método, devemos atribuir uma função a uma classe que, depois disso, poderá ser chamado como método do objeto usando o mesmo nome ao qual foi atribuída a função.

```
1  class Retangulo {  
2      constructor(altura, largura) {
```

<sup>3</sup> Veja mais em: <<https://www.devmedia.com.br/orientacao-a-objetos-parte-ii/7161>>

```
3     this._altura = altura;
4     this._largura = largura;
5 }
6
7 calcularArea() {
8     return this._altura * this._largura;
9 }
10 }
```

Para invocarmos o método, basta instanciar-mos a classe e criar assim um novo objeto.

```
1 const objeto = new Retangulo(10, 10);
2 console.log(objeto.calculaArea());
```

Contudo, existe aqui um erro semântico. Quando queremos saber a área de uma figura geométrica, deveríamos “perguntar” a está figura sua área e não solicitar o calculo. Logo, podemos resolver isso, criando um novo método que retorna apenas a área e que utiliza o método **calcularArea**.

```
1 class Retangulo {
2     constructor(altura, largura) {
3         this._altura = altura;
4         this._largura = largura;
5     }
6
7     area(){
8         return this.calcularArea();
9     }
10
11     calcularArea() {
12         return this._altura * this._largura;
13     }
14 }
15
16 const objeto = new Retangulo(10, 10);
17 console.log(objeto.area());
```

### 2.3.1 Método Get

Os métodos criados acima funcionam corretamente. Porém, podemos melhorar um pouco mais a semântica do objeto. Para isso usaremos a sintaxe **Get**.

Às vezes é desejável que se permita acesso a uma propriedade que retorna um valor computado dinamicamente, ou você pode querer refletir o status de uma variável interna sem requerer o uso de chamadas de método explícitas. Em Javascript, isso pode ser feito com o uso de um **getter**. Não é possível simultaneamente ter um **getter** associado a uma propriedade e a mesma possuir um valor, embora seja possível usar um **getter** e um **setter** em conjunto para criar algo como uma pseudo-propriedade.

```
1  class Retangulo {
2      constructor(altura, largura) {
3          this._altura = altura;
4          this._largura = largura;
5      }
6
7      get area(){
8          return this.calcularArea();
9      }
10
11     calcularArea() {
12         return this._altura * this._largura;
13     }
14 }
15
16 const objeto = new Retangulo(10, 10);
17 console.log(objeto.area);
```

### 2.3.2 Método Set

Já o **setter** pode ser usado para executar uma função sempre que se tenta mudar uma propriedade específica. **Setters** são geralmente usados em conjunto com **getters**, para criar um tipo de pseudo-propriedade. No entanto é impossível ter-se um **setter** para uma propriedade que contenha um valor real.

```
1  class Retangulo {
2
```

```
3      set altura(altura){
4          this._altura = altura;
5      }
6
7      set largura(largura){
8          this._largura = largura;
9      }
10
11     get area(){
12         return this.calcularArea();
13     }
14
15     calcularArea() {
16         return this._altura * this._largura;
17     }
18 }
19
20 const objeto = new Retangulo();
21 objeto.largura = 10;
22 objeto.altura = 10;
23 console.log(objeto.area);
24
25 objeto.largura = 100;
26 objeto.altura = 100;
27 console.log(objeto.area);
```

O setter se torna muito útil nesse caso pois não precisaremos criar um novo objeto, apenas alteramos seus atributos (seu estado) para obter novos valores.

### 2.3.3 Método Static

A palavra chave **static** define um método estático para a classe. Métodos estáticos não são chamados nas instâncias da classe. Em vez disso, eles são chamados na própria classe. Geralmente, são funções utilitárias, como funções para criar ou clonar objetos. Chamadas a métodos estáticos são feitas diretamente na classe e não podem ser feitas em uma instância da classe. Métodos estáticos são comumente utilizados como funções utilitárias<sup>4</sup>.

<sup>4</sup> Veja mais em: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Classes/static>>

```
1  class Retangulo {
2
3      //Codigo anterior
4
5      //Metodo adicionado
6      static formaGeometrica(objeto){
7          if (objeto._altura == objeto._largura){
8              return `A forma geometrica é um Quadrado`
9          } else {
10             return `A forma geometrica é um Retangulo`
11          }
12      }
13  }
14
15  const objeto = new Retangulo();
16  objeto.largura = 10;
17  objeto.altura = 10;
18  console.log(Retangulo.formaGeometrica(objeto));
19  console.log(objeto.area);
```

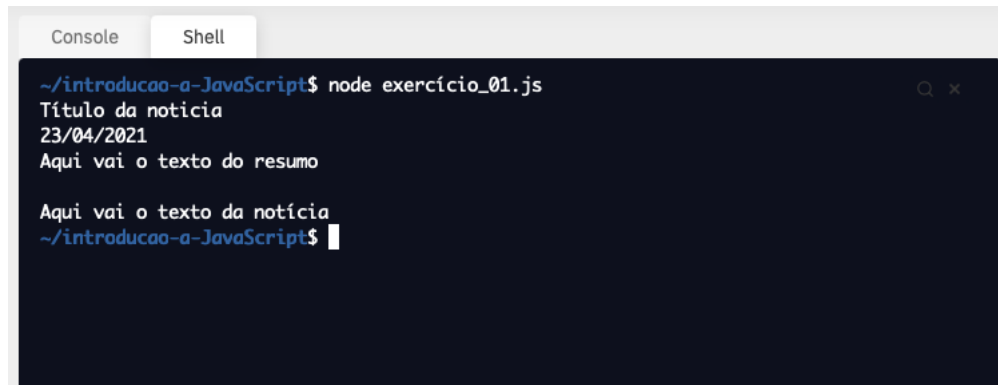
## 2.4 Exercícios de fixação

1. Uma empresa de notícias **IFNews** necessita desenvolver um portal de notícias. Após criada toda a parte documental, você, que é desenvolvedor/desenvolvedora na empresa, deverá iniciar a criação dos primeiros códigos. Sua primeira tarefa é a criação da classe **Noticia**. A classe deverá ser codificada seguindo, primordialmente, as definições abaixo:
  - a) **Nome da classe:** **Noticia**
  - b) **Atributos:** Título, Data da Publicação, Resumo e Texto (Outros atributos podem ser adicionados)
  - c) **Métodos:** Mostrar Noticia (Outros métodos podem ser adicionados)

Ao instanciar a classe e invocar o método **mostrar notícia** o resultado final deverá ser semelhante ao descrito abaixo (Lembre-se que o conteúdo no título, a data e o texto dependerá do **estado** adicionado a seu objeto):



### Resultado esperado:



```
~/introducao-a-JavaScript$ node exercicio_01.js
Título da noticia
23/04/2021
Aqui vai o texto do resumo

Aqui vai o texto da notícia
~/introducao-a-JavaScript$
```

Figura 3 – Resultado esperado

Fonte: O Autor

## 2.5 Exercícios extras

1. Crie uma classe para representar uma **pessoa**, com os atributos **nome**, **data de nascimento** e **altura**. Crie o método **constructor** para inicializar os atributos e também um método para imprimir todos dados de uma pessoa e outro para calcular a idade.
2. Crie uma classe para representar um **jogador de futebol**, com os atributos **nome**, **posição**, **data de nascimento**, **nacionalidade**, **altura** e **peso**. Crie o método para imprimir todos os dados do jogador. Crie um método para calcular a idade do jogador e outro método para mostrar quanto tempo falta para o jogador se aposentar. Para isso, considere que os jogadores da posição de defesa se aposentam em média aos 40 anos, os jogadores de meio-campo aos 38 e os atacantes aos 35.
3. Escreva uma classe cujos objetos representam alunos matriculados em uma disciplina. Cada objeto dessa classe deve guardar os seguintes dados do aluno: matrícula, nome, 2 notas de prova e 1 nota de trabalho. Escreva os seguintes métodos para esta classe:
  - **media**: calcula a média final do aluno (cada prova tem peso 2,5 e o trabalho tem peso 2)
  - **final**: calcula quanto o aluno precisa para a prova final (retorna zero se ele não for para a final)
4. Crie uma classe Agenda que pode armazenar 10 pessoas e que seja capaz de realizar as seguintes operações:

```
armazena(nome, idade, altura);
remove(nome);
```

```
busca(nome); // informa em que posição da agenda está a pessoa
imprimeDadosAgenda(); // imprime os dados de todas as pessoas da agenda
mostrar(index); // imprime os dados da pessoa que está na posição “i” da agenda.
```

## 2.6 Herança

A definição de herança, já abordada no Capítulo anterior, nós diz que: a herança é uma maneira de reutilizar código a medida que podemos aproveitar os atributos e métodos de classes já existentes para gerar novas classes mais específicas que aproveitarão os recursos da classe hierarquicamente superior [Ruiz 2008].

Para realizar a reutilização já citada, no ES6 foi adicionada a palavra reservada *extends*. Uma classe pode herdar atributos e métodos de outra classe por meio da extensão da classe desejada.

```
1  class Retangulo {
2
3      // Codigo anterior
4      ...
5  }
6
7  class Quadrado extends Retangulo{
8      constructor(lado) {
9          super(lado, lado);
10     }
11 }
12
13 let objeto = new Quadrado(100)
14 console.log(objeto.area)
```

Observem a palavra **super**. A palavra reservada **super** é usada para acessar o objeto pai de um objeto, em outros casos, é usada para acessar a classe pai de uma classe. Neste caso, precisamos observar alguns pontos:

- No construtor, o **super** deve ser chamado antes de qualquer acesso à **this**;
- Caso a classe que está realizando *extends* não defina o constructor ele será definido e chamado implicitamente passando todos os parâmetros.

```
1 // Errado
2 class Quadrado extends Retangulo{
3     constructor(lado) {
4         this._lado = lado;
5         super(lado, lado);
6     }
7 }
8
9 // Correto
10 class Quadrado extends Retangulo{
11     constructor(lado) {
12         super(lado, lado);
13         this._lado = lado;
14     }
15 }
```

## 2.7 Super classe e Subclasse

Arelado com o conceito de herança possuímos as definições de **Super classe** e **Subclasses**. O uso da palavra **extends** indica que a classe deveria ser baseado em um objeto padrão. Isso é chamado de Super Classe (*superclass*) e a classe derivada é a Subclasse (*subclass*) [Haverbeke 2018]

```
1 class Retangulo {
2     ...
3 }
4
5 class Quadrado extends Retangulo{
6     ...
7 }
```

Ocasionalmente, é útil saber se um objeto foi derivado de uma classe. Para isso, o JavaScript fornece um operador binário chamado **instanceof** [Haverbeke 2018].

```
1 console.log(new Quadrado(2) instanceof Retangulo) // True
```

Mas e **Retangulo** é instância de alguma classe? Neste caso sim, todos os objetos em JavaScript herdam a classe pai nomeado de **Object**

```
1 console.log(new Quadrado(2) instanceof Object) \\ True
2 console.log(new Retangulo(2, 2) instanceof Object) \\ True
```

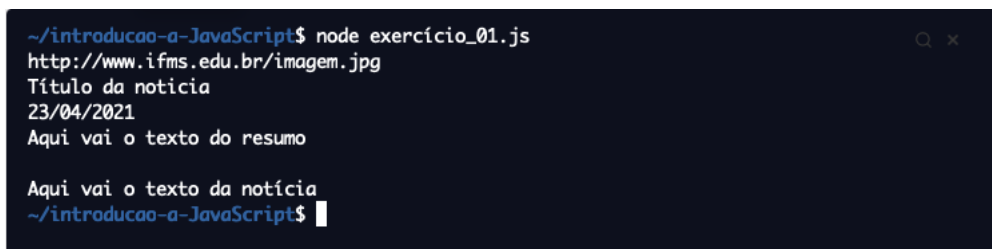
## 2.8 Exercícios de fixação

1. A empresa de notícias **IFNews** contratou você para desenvolver o portal de notícias, o qual teve seu início no exercício anterior. Sua primeira tarefa foi a criação da classe **Noticia** que possuía as seguintes definições:
  - a) **Nome da classe:** **Noticia**
  - b) **Atributos:** Título, Data da Publicação, Resumo e Texto (Outros atributos podem ser adicionados)
  - c) **Métodos:** Mostrar Noticia (Outros métodos podem ser adicionados)

Após desenvolver a tarefa, a empresa criou uma nova solicitação. O portal deve possuir **notícias em destaque**, as quais devem conter uma imagem seguida de um título e um resumo. Como você já criou uma notícia que contem parte do que o cliente necessita, a única tarefa será criar uma **subclasse** que herdará as características e ações da **Super Classe** e adicionará a característica desejada.

1. **Nome da classe:** **NoticiaDestaque**
2. **Atributos:** ImagemDestaque (apenas o caminho da imagem)
3. **Métodos:** Mostrar Destaque (Outros métodos podem ser adicionados)

Resultado esperado:



```
~/introducao-a-JavaScript$ node exercicio_01.js
http://www.ifms.edu.br/imagem.jpg
Título da noticia
23/04/2021
Aqui vai o texto do resumo

Aqui vai o texto da notícia
~/introducao-a-JavaScript$
```

Figura 4 – Resultado esperado

Fonte: O Autor

## 2.9 Tratamento de erros

Um requisito de software que deveríamos priorizar, em qualquer projeto, é como vamos realizar o tratamento de erros e exceções da nossa aplicação. **Saber como gerenciar erros é parte importante da programação.**

Existem dois tipos básicos de erros na programação:

- Erros de sintaxe;
- Erros de tempo de execução;

### 2.9.1 Erros de sintaxe

Os erros de sintaxe, também conhecidos como erros de análise, no tempo de interpretação no *JavaScript* (Linguagem interpretada / Linguagem compilada). Esse erro ocorre porque ao digitarmos algum código errado, ou seja, no momento em que esquecemos parênteses ou chaves, etc.

```
1  class Retangulo {  
2      constructor(altura, largura){  
3          this._altura = altura;  
4          this._largura = largura;  
5      }  
6      // Falta fechar as chaves
```

Ao executar o código acima, o retorno será: **SyntaxError: Unexpected end of input**

Os erros de Sintaxe são erros mais simples de serem resolvidos. Existem algumas maneiras de evitar esses tipos de erros, as quais são:

- Verificar o seu código e procurá-los (não é eficaz e pode levar muito tempo)
- Outra opção é usar alguma ferramenta ou plugin IDE. (eslint , jshint e jslint)

Essas ferramentas são muito populares e os(as) desenvolvedores(as), não gostam de tarefas repetitivas.

### 2.9.2 Erros de tempo de execução

Erros de execução, também chamados de **exceções**, ocorrem durante a execução (após a compilação/interpretação). Este tipo de erro não impede a compilação ou interpretação de um código. Contudo, o código gerado não funcionará.

```
1  class Retangulo {  
2      constructor(altura, largura){  
3          this._altura = altura;  
4          this._largura = largura;  
5      }  
6  }  
7  
8  let retangulo = new Retangulos(); // Nome no plural
```

Ao executar o código acima, o retorno será: *ReferenceError: Retangulos is not defined*

Por esse ser um erro mais complexo de ser tratado algumas linguagens implementam alguns mecanismos interessantes para manipular o retorno dos erros, sendo que, as exceções são utilizadas como uma forma de *feedback*, para que o(a) desenvolvedor(ar) possa saber o que fazer quando cair numa situação de erro.

Assim, para capturar estes erros e aplicar algum tratamento, podemos usar a sintaxe **try catch** a qual será o assunto a seguir.

### 2.9.3 Sintaxe do Try Catch Finally

A sintaxe do **try catch** é composta por dois blocos principais:

- O *try* é o local na qual o código é executado
- O *catch*, e o local que você recebe, via parâmetro na função, um **objeto do tipo Error**.

```
1  try {  
2      // seu codigo aqui  
3  } catch (error) {  
4      // tratamento de erro aqui  
5  }
```

### 2.9.4 Sintaxe do Try / Catch / finally

Nós podemos também usar a cláusula **finally**. Este bloco será executado sempre independente se houver ou não falha. Depois que o *try* ou *catch* serem executados, este bloco será acionado.

Isto pode ser útil por exemplo, para fechar um arquivo que foi aberto para leitura, registrar algum log ou fechar alguma conexão. Veja a sintaxe de exemplo.

```
1  try {  
2    // seu código aqui  
3  } catch (error) {  
4    // tratamento de erro aqui  
5  } finally {  
6    // executa sempre  
7  }
```

## 2.10 Exercícios de fixação

1. A classe Retangulo não possui tratamento de erros. Caso o usuário aplique uma altura e/ou uma largura igual ou menor que **Zero** ele não receberá um erro para que possa corrigir o problema. Assim, utilizando o conceito apresentado anteriormente, aplique o **Try** e o **Catch** no código abaixo para que, caso o usuário atribua um valor menor ou igual a zero, o código retorne um erro apontando como solucioná-lo.

```
1  class Retangulo {  
2    constructor(altura, largura){  
3      this._altura = altura;  
4      this._largura = largura;  
5    }  
6  
7    get area(){  
8      return this.calculaArea(this._altura, this._largura)  
9    }  
10  
11    calculaArea(altura, largura){  
12      return altura * largura
```

```
13     }  
14 }
```

### 2.10.1 Objeto Error

Sempre que uma exceção é lançada dentro do bloco `try`, o *JavaScript* cria um **objeto** do tipo **Error** e envia como argumento para o `catch`. Por padrão, este objeto é composto de duas propriedades principais:

- **name**: Representa o tipo do erro. Por exemplo, um erro de sintaxe - `SyntaxError`.
- **message**: Mensagem em texto, contendo mais detalhes do erro.

Em alguns ambientes onde você realiza o tratamento de erros em *JavaScript*, também terá a propriedade **stack**. Nela você tem a sequência de chamadas que levaram ao erro. Este tipo de informação é bastante útil para depuração.

### 2.10.2 Tipos nativos de erros

O objeto **Error** possui alguns tipos nativos que são invocados dependendo do contexto do erro. O JavaScript possui os seguintes tipos de erros nativos que podem ser lançados:

**ReferenceError** Lançado quando uma referência a uma variável ou função inexistente ou inválida é detectada.

**TypeError** Lançado quando um operador ou argumento passado para a função é de um tipo diferente do esperado.

**SyntaxError** Lançado quando ocorre algum erro de sintaxe ao interpretar o código, por exemplo ao realizar o parse de um JSON.

**URIError** Lançado quando ocorre algum erro no tratamento de URI, por exemplo, enviando parâmetros inválidos no `decodeURI()` ou `encodeURI()`.

**RangeError** Lançado quando um valor não está no conjunto ou intervalo de valores permitidos. Por exemplo, um valor em string num array número.

### 2.10.3 Customizar erros

Os erros nativos do JavaScript são muito úteis pelo fato de não termos ideia do problema que pode ocorrer. Contudo, se possuírmos regras de negócio específicas, podemos criar nossos próprios tipos de erros e lançar quando for preciso.



Para tanto, devemos estender a classe `Error` e criar uma modificação em seu construtor.

```
1  class ErroCustomizado extends Error {
2      constructor(message) {
3          super(message)
4          this.name = 'ErroCustomizado'
5      }
6  }
7
8  class Exemplo {
9      metodo(){
10         try {
11
12         } catch(e){
13
14         }
15     }
16 }
```

#### 2.10.4 Operador Throw

No exemplo anterior nos utilizamos o operador **throw** para lançar o erro. Quando há a necessidade de lançar uma exceção, usamos o operador **throw** precedida do valor, e este pode ser:

- string
- número
- objeto literal
- função ou até mesmo uma classe

```
1  throw "Descricao do erro"
2
3  throw 404
4
5  throw {
6      name: "Erro",
```

```
7     message: "Descricao do erro"
8   }
9
10   throw new UserTypeError("Tipo de usuario invalido")
```

Assim, para que possamos aplicar o erro customizado faríamos da seguinte forma:

```
1  class ErroCustomizado extends Error {
2    constructor(message) {
3      super(message)
4      this.name = 'ErroCustomizado'
5    }
6  }
7
8  class Exemplo {
9    metodo(){
10      try {
11        ...
12      } catch(e){
13        throw new ErroCustomizado("Mensagem de erro")
14      }
15    }
16  }
```

## 2.11 Exercícios de fixação

1. Com base na classe Retangulo do exercício anterior, crie e aplique o conceito de erro customizado.
2. Com base nas classe Notícias e NoticiaDestaque implementadas em exercícios anteriores, crie e aplique o conceito de erro customizado.

## Referências

HAVERBEKE, M. *Eloquent JavaScript: a modern introduction to programming*. [S.l.]: No Starch Press, 2018. Citado na página 11.

RUIZ, E. E. S. *IBm1030 Programação Orientada a Objetos*. 2008. Disponível em: <http://dcm.ffclrp.usp.br/~evandro/ibm1030/constru/heranca.html>. Citado na página 10.

TRASVIÑA, F. *Introdução ao JavaScript Orientado a Objeto*. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/conflicting/Learn/JavaScript/Objects>. Nenhuma citação no texto.