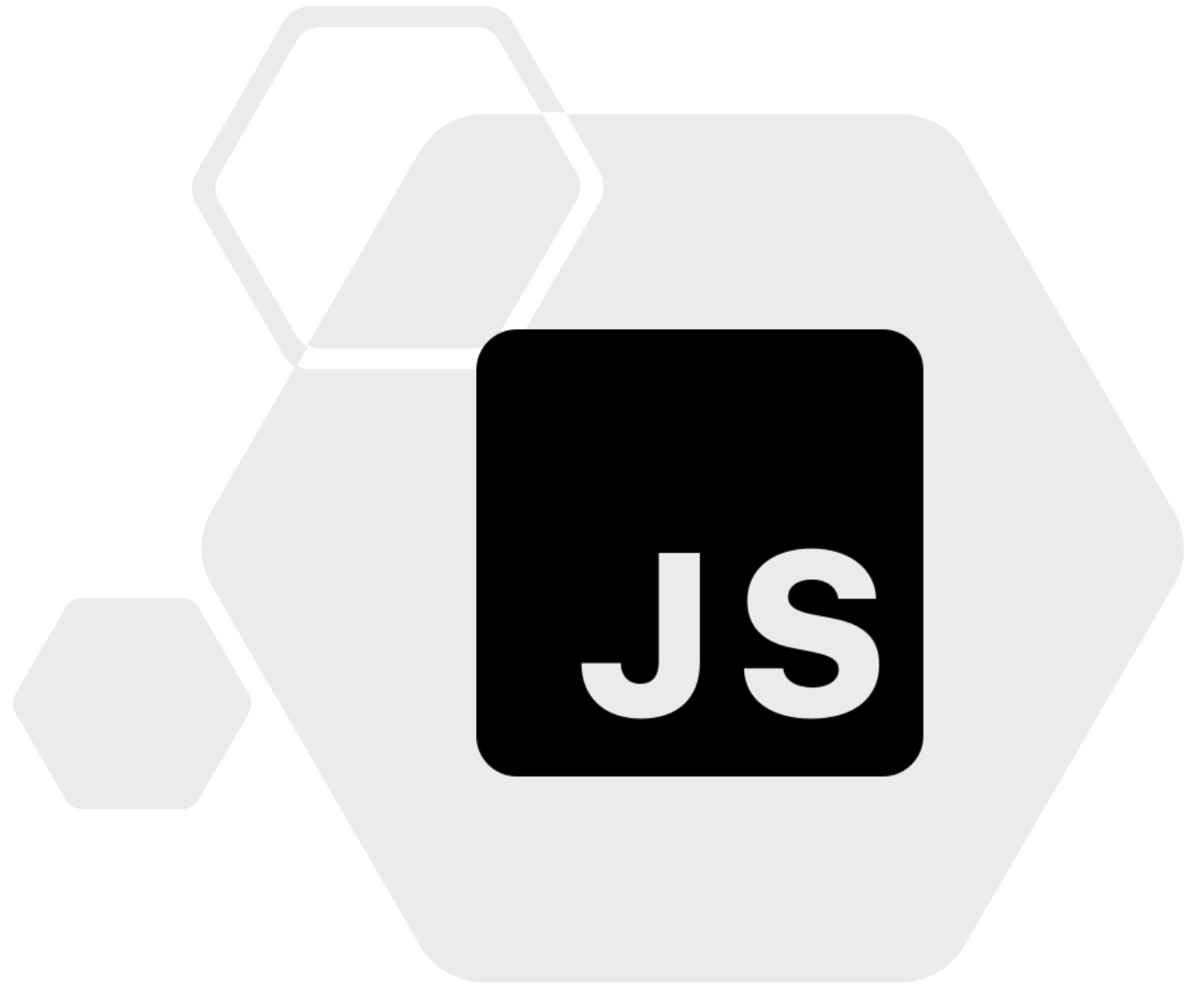
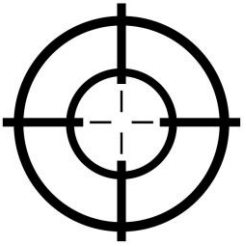


Objetos



O que são objetos?

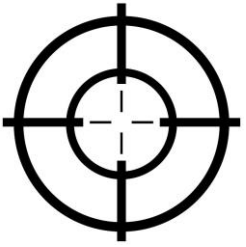


Em JavaScript, os objetos são reis! Se você entende de objetos, entende JavaScript.

Um objeto é uma coleção de propriedades, e uma propriedade é uma associação entre um nome (ou chave) e um valor.

Importante! Todos os valores (tipos) em JavaScript, exceto primitivos, são objetos.

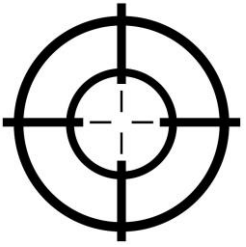
O que são primitivos?



Importante! Um valor primitivo é um valor que não possui propriedades ou métodos. JavaScript define 5 tipos de tipos de dados primitivos: string, number, boolean, null, undefined.


Geralmente, as propriedades de um objeto são uma coleção de valores primitivos.

Como criar objetos?




Para criar um objeto pode-se usar a forma literal.

```
const empresa = {  
  nome: "Banco Nacional",  
  cnpj: "12548796542395"  
}
```


A diagram consisting of a vertical line and a horizontal line forming an L-shape. The horizontal line points to the closing curly brace of the object literal in the code above.

Propriedades podem ser adicionadas depois que o objeto foi criado!

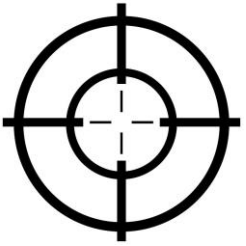
```
const empresa = {}  
personagem.nome = "Banco Nacional"  
personagem.cnpj = "12548796542395"
```

A diagram consisting of a vertical line and a horizontal line forming an L-shape. The horizontal line points to the assignment of the 'nome' property in the code above.

empresa.nome = "Banco Nacional"
empresa.cnpj = "12548796542395"

A green curved arrow pointing from the underlined property names in the code above to the resulting object properties in the text below.

Como criar objetos?



Usar o método construtor, fornecido pelo JavaScript, é outra forma de criar objetos.

```
const empresa = new Object()  
personagem.nome = "Banco Nacional"  
personagem.cnpj = "12548796542395"
```

Cabe ao desenvolvedor escolher a forma que lhe é mais conveniente ou mais adequada, para o momento.

Referência



Uma vez que objetos não são primitivos, eles são endereçados por referência, não por valor.

Uma referência é uma variável especial, que armazena o endereço do objeto na memória, como um “controle remoto”.

```
const empresa = new Object()
empresa.nome = "Banco Nacional"
empresa.cnpj = "12548569325478"

const empresa2 = empresa
empresa2['nome'] = "Banco Internacional"

console.log(empresa.nome)
```

O que será impresso pelo
console.log()?

Sobre referência...



Toda vez que um desenvolvedor atribui uma referência a outra não ocorre cópia dos seus valores. Ocorre a cópia do endereço onde o objeto está, na memória.

Entende-se o objeto como uma “capsula”. Suas propriedades possuem valores e diz-se que elas definem o “estado” do objeto.

Sobre referência...



Os parâmetros, em uma chamada de função, são os argumentos da função.

```
let valorPrimitivo = 10

function aumentarMaisDez(valor) {
  valor = valor + 10
}

aumentarMaisDez(valorPrimitivo)

console.log('Este é o valor da variável: ' + valorPrimitivo)
```

Diagram illustrating the flow of data in the provided JavaScript code. A bracket on the left groups the variable declaration and the function call. An arrow points from the `valorPrimitivo` argument in the function call to the `valor` parameter in the function definition. Another arrow points from the `valor` parameter to its assignment statement inside the function, showing that the function operates on a local copy of the value.

Os argumentos são passados por valor: a função só conhece os valores, não a localização dos argumentos, quando estes são primitivos!

Sobre referência...



Atenção! Agora o argumento é um objeto!

```
[let valorReferencia = {valor: 10}

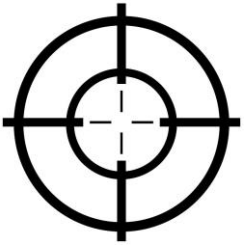
function aumentarMaisDez(referencia) {
  |   referencia.valor = referencia.valor + 10
  |
}

aumentarMaisDez(valorReferencia)

console.log('Este é o valor da variável: ' + valorReferencia.valor)]
```

Se uma função altera a propriedade de um objeto, ela altera o valor original da propriedade!

Método call()



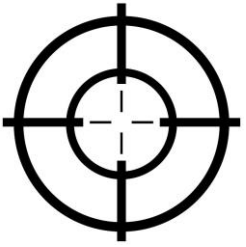
O método call() é um método predefinido em JavaScript.

Pode ser usado para invocar (chamar) um método com um objeto proprietário como argumento (parâmetro). Também, com call() um objeto pode usar um método que pertence a outro objeto.

```
const detalhador = {  
  detalhar: function () {  
    return '\n Nome Fantazia: ' + this.nome + " Razão Social: " + this.razaoSocial  
  }  
}  
  
let empresa = {nome: 'Mercado Online', razaoSocial: 'ABC LTDA'}  
  
console.log('Quais os detalhes da empresa: ' + detalhador.detalhar.call(empresa))
```

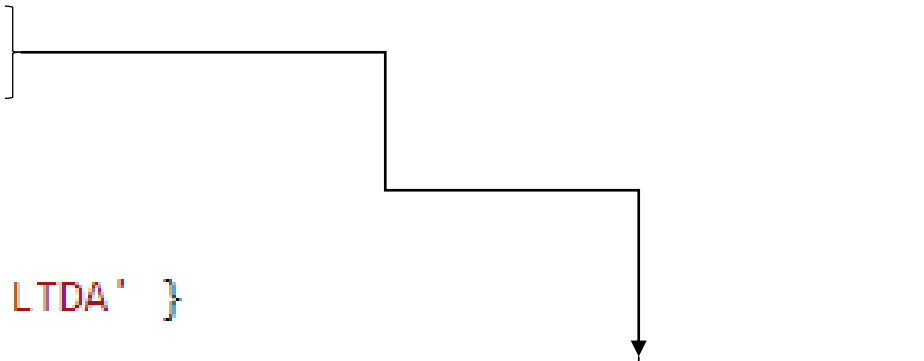
A diagram with arrows illustrating the call method. One arrow points from the 'detalhador' object to the 'detalhar' function. Another arrow points from the 'empresa' object to the 'call' method. A third arrow points from the 'call' method to the 'empresa' object, indicating it is passed as an argument.

Método call()

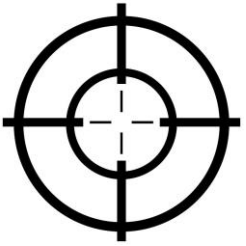


Além do objeto também pode-se adicionar mais argumentos ao método call().

```
const detalhador = {  
  detalhar: function (cidade, estado) {  
    return '\n Nome Fantazia: ' + this.nome + " Razão Social: " + this.razaoSocial  
      + '\n---\n'  
      + 'cidade: ' + cidade + '\nestado: ' + estado  
  }  
}  
  
let empresa = { nome: 'Mercado Online', razaoSocial: 'ABC LTDA' }  
  
console.log('Quais os detalhes da empresa: ' + detalhador.detalhar.call(empresa, 'São José', 'SP'))
```

A diagram illustrating the execution of the `call` method. A bracket on the right side of the `detalhar` function definition in the first code block is connected by a horizontal line to a vertical line. This vertical line then turns left and connects to a horizontal line that spans over the arguments `'São José', 'SP'` in the `call` method invocation in the last code block. An arrow points from the end of this horizontal line down to the `call` method, indicating the flow of the function call.

Método apply()



O método `apply()` é semelhante ao método `call()`. A diferença é que o método `call()` aceita argumentos separadamente. O método `apply()` aceita argumentos como um Array.

```
const detalhador = {  
  detalhar: function (cidade, estado) {  
    return '\n Nome Fantazia: ' + this.nome + " Razão Social: " + this.razaoSocial  
      + '\n---\n'  
      + 'cidade: ' + cidade + '\nestado: ' + estado  
  }  
}  
  
let empresa = { nome: 'Mercado Online', razaoSocial: 'ABC LTDA' }  
  
console.log('Quais os detalhes da empresa: ' + detalhador.detalhar.apply(empresa, ['São José', 'SP']))
```

A diagram illustrating the execution of the `apply` method. A line originates from the closing curly brace of the `detalhar` function in the `detalhador` object. This line extends to the right and then turns downwards, ending with an arrowhead pointing to the `apply` property of the `detalhador` object in the `console.log` statement. A bracket is placed above the arguments `['São José', 'SP']` in the `apply` call, indicating they are passed as an array.

Adicionando propriedades



Adicionar uma nova propriedade a um objeto existente é fácil.

```
const empresa = {  
  nome: 'Mercado Online'  
}
```

```
{ empresa.razaoSocial = 'ABC LTDA'
```

```
console.log('Qual o nome da empresa: ' + empresa.razaoSocial)
```

A propriedade passa a existir no objeto.

Propriedades podem ser outros objetos?



Sim! Isso já era esperado, uma vez que pode-se colocar uma função como propriedade e uma função também é um objeto.

```
const empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA',  
  telefone: {  
    ddd: '00',  
    numero: '999999999'  
  }  
}
```

A propriedade telefone também é um objeto.



```
console.log("Telefone da empresa: " + empresa.telefone.ddd + " " + empresa.telefone.numero)
```

Sobre propriedades...



Dado que um objeto é formado por uma coleção de propriedades, é possível percorrer suas propriedades como se fosse um Array.

```
const empresa = new Object()
empresa.nome = "Banco Nacional"
empresa.cnpj = "12548569325478"

for(let propriedade in empresa){
  console.log(propriedade)
}
```

E se for necessário imprimir o valor da propriedade, como você faria?

Outra forma de obter-se os valores das propriedades...



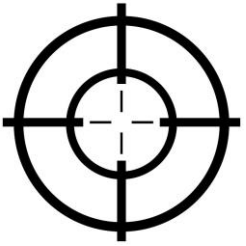
Qualquer objeto pode ser convertido em um Array, com o valor das suas propriedades, usando `Object.values()`.

```
let empresa = { nome: 'Mercado Online', razaoSocial: 'ABC LTDA' }
```

```
let dados = Object.values(empresa)
```

```
console.log(dados)
```


Excluindo propriedades



Assim como é possível adicionar uma propriedade, também é possível remover uma propriedade do objeto.

```
const empresa = new Object()  
empresa.nome = "Banco Nacional"  
empresa.cnpj = "12548569325478"  
[empresa.endereco = "Av. Brasil, nº 541, Rio de Janeiro"]
```

```
delete empresa.endereco  
console.log(empresa.endereco)
```

A propriedade passa a não mais existir no objeto!

Adicionando métodos



Assim como propriedades, também pode-se adicionar novos métodos ao objeto.

```
let empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA'  
}
```

Importante! A exclusão também é possível, assim como nas propriedades.

```
empresa.detalhe = function () {  
  return this.nome + '\n' + this.razaoSocial  
};
```

```
console.log('Qual o nome da empresa: \n' + empresa.detalhe())
```

Métodos de acesso (getters e setters)



O ECMAScript 5 (ES5 2009) introduziu o conceito de métodos de acesso, os Getter e Setters. Getters e setters permitem definir maneiras, diferenciadas, de manipulação das propriedades de um objeto.

```
let empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA',  
  get pegarNome() {  
    return this.nome  
  }  
}
```

Este é um exemplo de como usar o Get!

```
console.log('Qual o nome da empresa: ' + empresa.pegarNome())
```

Métodos de acesso (getters e setters)



Não é somente acesso, também inserção!

```
let empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA',  
  set colocarNome(novoNome){  
    this.nome = novoNome  
  }  
}
```

Este é um exemplo de como usar o Set!

Importante! O objeto pode ter Get e Set para todas as suas propriedades.

```
empresa.colocarNome = "Mercado Online Americano"
```

```
console.log('Qual o nome da empresa: ' + empresa.nome)
```

Usar função ou métodos de acesso?

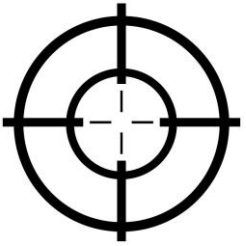


E agora? O que é melhor usar, funções ou métodos de acesso? São parecidos correto?

A resposta: é melhor usar métodos de acesso, por dois motivos principais. O primeiro é que a sintaxe para chamar os métodos é mais simples, como se fossem propriedades.

O segundo é o melhor controle da qualidade e manipulação dos dados.

Qualidade dos dados...



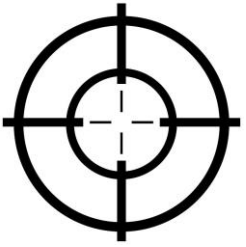
Importante! Usar métodos ou funções facilitam a manipulação e manutenção da qualidade dos dados!

```
let empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA',  
  get pegarNome() {  
    return this.nome.toUpperCase()  
  }  
}
```

```
console.log('Qual o nome da empresa: ' + empresa.pegarNome)
```

Antes de entregar o valor para impressão no console.log(), foi possível fazer um pré-processamento, sem comprometer o dado original.

Qualidade dos dados...



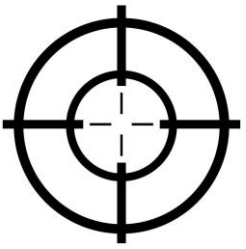
```
const empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA',  
  get descricao() {  
    return 'Nome: ' + this.nome + "\nRazão Social: " + this.razaoSocial  
  }  
}
```

```
empresa.descricao = 'Tentando modificar o método get!'
```

```
console.log("Descrição:\n" + empresa.descricao)
```

Não é possível alterar os métodos de acesso, como se fossem propriedades comuns. Isto garante confiabilidade na manipulação de dados.

Outra forma de adicionar métodos Get ou Set...



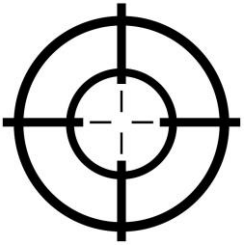
```
let empresa = {  
  nome: 'Mercado Online',  
  razaoSocial: 'ABC LTDA',  
  set colocarNome(novoNome) {  
    this.nome = novoNome  
  }  
}
```

O método `Object.defineProperty()` pode ser usado para adicionar Getters e Setters.

```
Object.defineProperty(empresa, 'pegarNome',  
  {  
    get: function () { return this.nome }  
  }  
)
```

```
console.log('Qual o nome da empresa: ' + empresa.pegarNome)
```


Função construtora



Às vezes, precisa-se de um “plano”, uma “estratégia”, um “modelo ou molde”, para criar muitos objetos do mesmo “tipo”.

A maneira de criar um “tipo de objeto” é usar uma função construtora de objetos.

```
function Empresa(nome, razaoSocial) {  
    this.nome = nome  
    this.razaoSocial = razaoSocial  
}
```

Objetos do mesmo tipo são criados chamando a função construtora com a palavra-chave "new".

```
let empresa = new Empresa('Mercado Online', 'ABC LTDA')
```

Função construtora



Atenção! A função construtora não retorna valor!

Em JavaScript, a palavra-chave “this” representa o objeto que “possui” o código. Na função construtora, “this” não tem um valor. É um substituto para o novo objeto. O valor deste se tornará o novo objeto quando ele for criado.

Adicionar novas propriedades ou métodos ao objeto criado não altera a função construtora! As novas propriedades ou métodos passam a existir apenas no objeto que as recebeu!

Função construtora

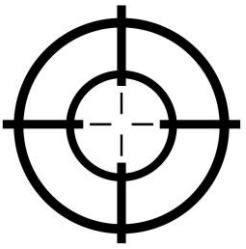


Uma boa prática de programação é colocar o nome de funções construtoras com letra maiúscula. Esta prática é comum entre os desenvolvedores que usam linguagens orientadas à objetos.

Além da letra maiúscula, geralmente, o nome da função construtora é escolhido como algo que representa os objetos de maneira geral, uma classificação ou classe.

Outra boa prática é colocar o nome dos parâmetros da função iguais aos nomes das propriedades, dado que é possível diferenciá-los com o “this”.

Adicionando métodos a função construtora

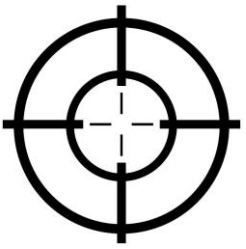


```
function Empresa(nome, razaoSocial) {  
  this.nome = nome  
  this.razaoSocial = razaoSocial,  
  this.detalhe = function () {  
    return this.nome + '\n' + this.razaoSocial  
  }  
}
```

O método passa a existir em cada objeto que for criado.

```
let empresa = new Empresa('Mercado Online', 'ABC LTDA')  
  
console.log("Detalhes da empresa: \n" + empresa.detalhe())
```

Propriedades com valor pré-fixado



Se necessário pode-se pré-fixar o valor de um propriedade, na função construtora.

```
function Produto() {  
  this.lote = '1ADF2019MAR03'  
}
```

O valor ficará padrão (default) para todos os objetos, mas pode ser modificado depois.

```
const produto = new Produto()  
const produto2 = new Produto()
```

```
console.log("Lote do produto: " + produto.lote + "\nLote do produto 2: " + produto2.lote)
```

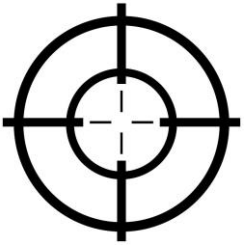
Protótipo e herança



Todos os objetos herdam propriedades e métodos de um protótipo, um objeto “superior” ou “pai”. Por exemplo, objetos Array herdam de Array.prototype. A própria biblioteca da linguagem fornece protótipos para seus elementos.

O Object.prototype está no topo da cadeia de herança de protótipos, ou seja, é o "pai"/"superior" de todos.

Protótipo e herança



Às vezes, deseja-se adicionar, posteriormente, novas propriedades (ou métodos) a todos os objetos existentes de um determinado tipo. Para isso, pode-se usar o protótipo relacionado ao objeto.

```
function Empresa(nome, razaoSocial) {  
  this.nome = nome  
  this.razaoSocial = razaoSocial  
}  
  
const empresa = new Empresa('Mercado Online', 'ABC LTDA')  
  
Empresa.prototype.descricao = function () {  
  return 'Nome: ' + this.nome + '\n' + 'Razão Social: ' + this.razaoSocial  
}  
  
console.log("Descrição:\n" + empresa.descricao())
```

A diagram consisting of two lines. One line starts from the right side of the text 'Para isso, pode-se usar o protótipo relacionado ao objeto.' and extends horizontally to the right. Another line starts from the right side of the text 'Empresa.prototype.descricao = function () {' and extends horizontally to the left. These two lines meet at a point, with a small arrowhead pointing towards the 'descricao' property assignment, indicating that the object 'empresa' inherits the 'descricao' method from the 'Empresa' prototype.

Protótipo e herança



Importante! As propriedades ou métodos, quando adicionados pelo protótipo do objeto, passam a existir para todos os objetos daquele tipo, inclusive para aqueles que já foram criados.

Importante! Adicionar propriedades ou métodos utilizando o protótipo do objeto altera a função construtora dinamicamente, ou seja, adiciona coisas nela sem alterar a declaração da função explicitamente.

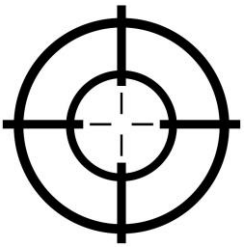
Protótipo, por “debaixo dos panos”...



O protótipo faz algumas coisas, implicitamente, para poder criar o objeto pela função construtora.

```
function Empresa(nome, razaoSocial) {  
  // A palavra-chave new adiciona, implicitamente a linha:  
  // var this = Object.create(Empresa.prototype)  
  
  this.nome = nome;  
  this.razaoSocial = razaoSocial;  
  
  // Por fim, a função retorna implicitamente este valor:  
  // return this  
}
```

Adicionando métodos de acesso na função construtora



O protótipo pode ser usado para adicionar métodos de acesso, diretamente na função construtora!

```
function Empresa(nome, razaoSocial) {  
  this.nome = nome  
  this.razaoSocial = razaoSocial  
  Object.defineProperty(this, 'descricao', {  
    get: function () {  
      return 'Nome: ' + this.nome + "\nRazão Social: " + this.razaoSocial  
    }  
  })  
}
```

```
let empresa = new Empresa('Mercado Online', 'ABC LTDA')
```

```
console.log("Descrição:\n" + empresa.descricao)
```

- Modifique apenas seus próprios protótipos. Nunca modifique os protótipos de objetos padrões da linguagem. Você pode acabar prejudicando seu próprio trabalho.



JavaScript

