

Prototype 2

Construtores Nativos

Existem construtores de Objetos, Funções, Números, Strings e outros tipos de dados são criados utilizando construtores. Esses construtores possuem um protótipo com propriedades e métodos, que poderão ser acessadas pelo tipo de dado. E diversos outros dados como o `nodeList`, o `element` e por aí.

```
const pais = 'Brasil';
const cidade = new String('São Carlos');

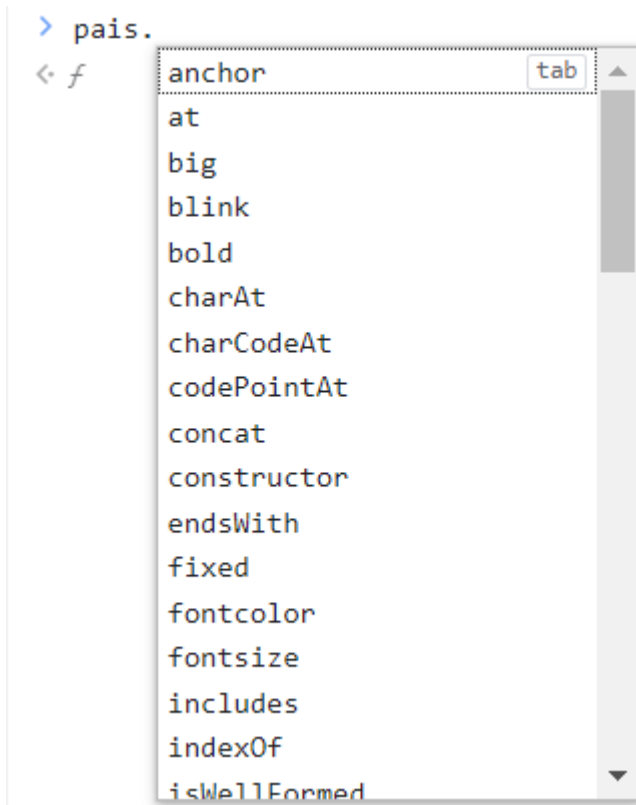
pais.charAt(0); // B
cidade.charAt(0); // R

String.prototype;
```

Esses construtores são funções e toda função sempre possui um protótipo com ela e nessas funções especiais nativas geralmente existem propriedades e métodos que eu vou poder usar baseado nesse tipo de dado.

```
const pais = 'Brasil';
```

```
> pais
< 'Brasil'
```



Quando eu coloco pais. ele me retorna um monte de propriedades e métodos eles vem desse construtor:

```
const cidade = new String('São Carlos');
```

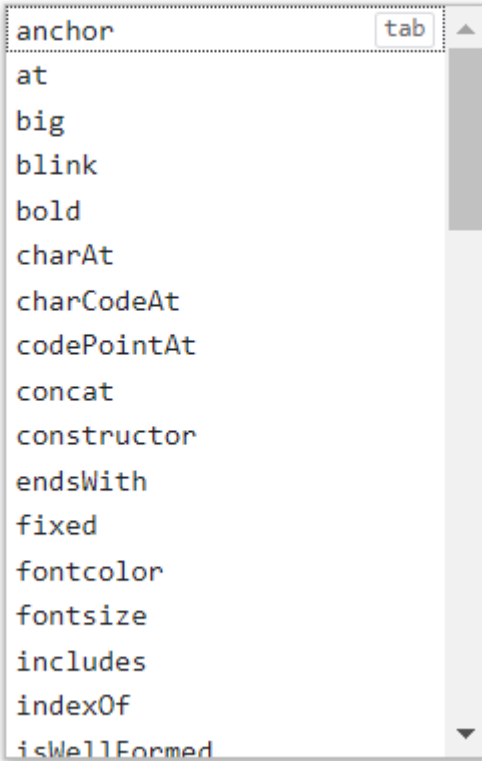
```
> cidade
< ▼ String {'São Carlos'} ⓘ
  0: "S"
  1: "ã"
  2: "o"
  3: " "
  4: "C"
  5: "a"
  6: "r"
  7: "l"
  8: "o"
  9: "s"
  length: 10
  ► [[Prototype]]: String
  [[PrimitiveValue]]: "São Carlos"
```

A variável cidade me retorna um objeto só que é um objeto que tem um construtor que é o `String` assim como:

```
> leonardo
< ▶ Person {name: 'Leonardo', age: 19, skill: f, walk: f}
```

Apesar do país ser retornado direto ele também tem seus métodos e propriedades:

```
> pais.
< f
```



Por um breve segundo ele vai ser envolvido no objeto String e vai ter acesso aos métodos e propriedades dele.

```
> cidade
< ▼ String {'São Carlos'} ⓘ
  0: "S"
  1: "ã"
  2: "o"
  3: " "
  4: "C"
  5: "a"
  6: "r"
  7: "l"
  8: "o"
  9: "s"
  length: 10
  ▼ [[Prototype]]: String
    ► anchor: f anchor()
    ► at: f at()
    ► big: f big()
    ► blink: f blink()
    ► bold: f bold()
    ► charAt: f charAt()
    ► charCodeAt: f charCodeAt()
    ► codePointAt: f codePointAt()
    ► concat: f concat()
    ► constructor: f String()
    ► endsWith: f endsWith()
    ► fixed: f fixed()
    ► fontcolor: f fontcolor()
    ► fontsize: f fontsize()
    ► includes: f includes()
    ► indexOf: f indexOf()
    ► isWellFormed: f isWellFormed()
    ► italics: f italics()
    ► lastIndexOf: f lastIndexOf()
    length: 0
    ► link: f link()
```

Dentro do proto de cidade tem todas aquelas funções pra modificar a string como alterar para *uppercase*.

```
String.prototype;
```

Eu posso passar o nome da função construtora com o prototype:

> String.prototype

◀ ▼ String {'', constructor: f, anchor: f, at: f, big: f, ...} ⓘ

- ▶ anchor: f anchor()
- ▶ at: f at()
- ▶ big: f big()
- ▶ blink: f blink()
- ▶ bold: f bold()
- ▶ charAt: f charAt()
- ▶ charCodeAt: f charCodeAt()
- ▶ codePointAt: f codePointAt()
- ▶ concat: f concat()
- ▶ constructor: f String()
- ▶ endsWith: f endsWith()
- ▶ fixed: f fixed()
- ▶ fontcolor: f fontcolor()
- ▶ fontsize: f fontsize()
- ▶ includes: f includes()
- ▶ indexOf: f indexOf()
- ▶ isWellFormed: f isWellFormed()
- ▶ italics: f italics()
- ▶ lastIndexOf: f lastIndexOf()
- length: 0
- ▶ link: f link()
- ▶ localeCompare: f localeCompare()
- ▶ match: f match()
- ▶ matchAll: f matchAll()
- ▶ normalize: f normalize()
- ▶ padEnd: f padEnd()
- ▶ padStart: f padStart()
- ▶ repeat: f repeat()
- ▶ replace: f replace()
- ▶ replaceAll: f replaceAll()

É possível acessar a função do protótipo

É comum, principalmente em códigos mais antigos, o uso direto de funções do protótipo do construtor Array.

Eu posso acessar:

```
> String.prototype.toUpperCase()  
◀ ''
```

Não faz tanto sentido porque a String não tenho nenhum valor, mas eu acessei a função e ela funcionou, ela só não tem nenhum valor dentro dela.

Mas é muito comum você vê códigos utilizando função direto de [Array](#).

```
const list = document.querySelectorAll('li');  
  
// Transforma em uma array  
const listArray = Array.prototype.slice.call(list);
```

[Array](#) é um construtor de arrays sempre que você cria um array o prototype do array vai pra esse [Array](#).

```
const listAnimais = ['Cat', 'Dog', 'Horse'];
```

```
> listAnimais  
◀ ▼ (3) ['Cat', 'Dog', 'Horse'] ⓘ  
  0: "Cat"  
  1: "Dog"  
  2: "Horse"  
  length: 3  
  ► [[Prototype]]: Array(0)
```

A variável listAnimais retorna um array normal. Esse array possui um protótipo de um Array que tem vários métodos e propriedades.

```
< ▼ (3) ['Cat', 'Dog', 'Horse'] ⓘ  
  0: "Cat"  
  1: "Dog"  
  2: "Horse"  
  length: 3  
  ▼ [[Prototype]]: Array(0)  
    ▶ at: f at()  
    ▶ concat: f concat()  
    ▶ constructor: f Array()  
    ▶ copyWithin: f copyWithin()  
    ▶ entries: f entries()  
    ▶ every: f every()  
    ▶ fill: f fill()  
    ▶ filter: f filter()  
    ▶ find: f find()  
    ▶ findIndex: f findIndex()  
    ▶ findLast: f findLast()  
    ▶ findLastIndex: f findLastIndex()  
    ▶ flat: f flat()  
    ▶ flatMap: f flatMap()  
    ▶ forEach: f forEach()  
    ▶ includes: f includes()  
    ▶ indexOf: f indexOf()  
    ▶ join: f join()  
    ▶ keys: f keys()  
    ▶ lastIndexOf: f lastIndexOf()  
    length: 0  
    ▶ map: f map()  
    ▶ pop: f pop()  
    ▶ push: f push()  
    ▶ reduce: f reduce()  
    ▶ reduceRight: f reduceRight()  
    ▶ reverse: f reverse()  
    ▶ shift: f shift()
```

Mesma coisa está no protótipo dela:

```
> Array.prototype
< ▼ [constructor: f, at: f, concat: f, copyWithin: f, fill: f, ...] ⓘ
  ▶ at: f at()
  ▶ concat: f concat()
  ▶ constructor: f Array()
  ▶ copyWithin: f copyWithin()
  ▶ entries: f entries()
  ▶ every: f every()
  ▶ fill: f fill()
  ▶ filter: f filter()
  ▶ find: f find()
  ▶ findIndex: f findIndex()
  ▶ findLast: f findLast()
  ▶ findLastIndex: f findLastIndex()
  ▶ flat: f flat()
  ▶ flatMap: f flatMap()
  ▶ forEach: f forEach()
  ▶ includes: f includes()
  ▶ indexOf: f indexOf()
  ▶ join: f join()
  ▶ keys: f keys()
  ▶ lastIndexOf: f lastIndexOf()
  ▶ length: 0
  ▶ map: f map()
  ▶ pop: f pop()
  ▶ push: f push()
  ▶ reduce: f reduce()
  ▶ reduceRight: f reduceRight()
  ▶ reverse: f reverse()
  ▶ shift: f shift()
  ▶ slice: f slice()
  ▶ some: f some()
```

```
> list
```

```
< ▶ NodeList(3) [li.ativo, li, li.ativo]
```

Isso é uma nodelist, ou seja, o construtor que construiu essa lista é um nodelist e não um array.

```
> list
< ▼ NodeList(3) [li.ativo, li, li.ativo] ⓘ
  ▶ 0: li.ativo
  ▶ 1: li
  ▶ 2: li.ativo
  length: 3
  ▼ [[Prototype]]: NodeList
    ▶ entries: f entries()
    ▶ forEach: f forEach()
    ▶ item: f item()
    ▶ keys: f keys()
    length: (...)
    ▶ values: f values()
    ▶ constructor: f NodeList()
    ▶ Symbol(Symbol.iterator): f values()
    Symbol(Symbol.toStringTag): "NodeList"
    ▶ get length: f length()
    ▶ [[Prototype]]: Object
```

Os métodos do protótipo não tem alguns métodos que o array tem direto e talvez você queira utilizar. Então você transformar a variável list da seguinte forma:

```
> Array.prototype.slice.call(list)
```

```
< ▼ (3) [li.ativo, li, li.ativo] ⓘ  
  ▶ 0: li.ativo  
  ▶ 1: li  
  ▶ 2: li.ativo  
    length: 3  
  ▼ [[Prototype]]: Array(0)  
    ▶ at: f at()  
    ▶ concat: f concat()  
    ▶ constructor: f Array()  
    ▶ copyWithin: f copyWithin()  
    ▶ entries: f entries()  
    ▶ every: f every()  
    ▶ fill: f fill()  
    ▶ filter: f filter()  
    ▶ find: f find()  
    ▶ findIndex: f findIndex()  
    ▶ findLast: f findLast()  
    ▶ findLastIndex: f findLastIndex()  
    ▶ flat: f flat()  
    ▶ flatMap: f flatMap()  
    ▶ forEach: f forEach()  
    ▶ includes: f includes()  
    ▶ indexOf: f indexOf()  
    ▶ join: f join()  
    ▶ keys: f keys()  
    ▶ lastIndexOf: f lastIndexOf()  
      length: 0  
    ▶ map: f map()  
    ▶ pop: f pop()  
    ▶ push: f push()  
    ▶ reduce: f reduce()
```

Obs: call é um método de funções.

Então agora ele me retorna um array elas parecem iguais, mas são diferentes porque o protótipo delas são diferentes. Por isso, é muito comum você pegar uma lista e sempre transformar em um array porque aí você vai ter acesso a diferentes métodos pra interagir com o array de forma diferente.

MÉTODO DO OBJETO VS PROTÓTIPO

Existe também uma diferença de métodos que são diretos do objeto e métodos que são do protótipo. Nós vimos anteriormente como transformar algo que parece como um array em um array, só que existe também um método direto do Array o **from**.

```
Array.from(list);
```

Ele vai transformar numa lista direto é melhor do que a solução anterior, mas antes se usava daquele jeito porque não existia o from.

```
const listArray2 = Array.from(list);
```

```
> listArray2  
◀ ▶ (3) [li.ativo, li, li.ativo]
```

Vai ser a mesma coisa, porém muito mais rápido. Só que o método from está linkado direto a função array diferente do slice que está ligado ao protótipo. A diferença é que um array normal foi criado com o construtor Array, mas ela não tem acesso ao método from direto do Array.

```
> [3,3,2].from()
```

```
✖ ▶ Uncaught TypeError: [3,3,2].from is not a function  
at <anonymous>:1:9
```

[VM664:1](#)

Mas se eu for no slice ele tem:

```
> [3,3,2].slice()  
◀ ▶ (3) [3, 3, 2]
```

Existe também:

```
Object.create();
```

Esses você não vai ter acesso a objetos que foram criados com o construtor Object.

Nós temos um método do construtor Objeto chamado `getOwnPropertyNames` que é pra pegar quais são as propriedades que o item que eu passar tem:

```
> Object.getOwnPropertyNames(Array)
< ▼ (6) ['length', 'name', 'prototype', 'isArray', 'from', 'of'] ⓘ
  0: "length"
  1: "name"
  2: "prototype"
  3: "isArray"
  4: "from"
  5: "of"
  length: 6
  ► [[Prototype]]: Array(0)
```

Diferente do que está no protótipo dela:

```
> Object.getOwnPropertyNames(Array.prototype)
< (40) ['length', 'constructor', 'at', 'concat', 'copyWithin', 'fill', 'find', 'findIndex', 'findLast', 'findLastIndex', 'lastIndexOf', 'pop', 'push', 'reverse', 'shift', 'unshift', 'slice', 'sort', 'splice', 'includes', 'indexOf', 'join', 'keys', 'entries', 'values', 'forEach', 'filter', 'flat', 'flatMap', 'map', 'every', 'some', 'reduce', 'reduceRight', 'toLocaleString', 'toString', 'toReversed', 'toSorted', 'toSpliced', 'with']
```

São 40 métodos que ele tem totalmente diferentes e ele não tem o from.

```
> Object.getOwnPropertyNames(leonardo)
< ▼ (4) ['name', 'age', 'skill', 'walk'] ⓘ
  0: "name"
  1: "age"
  2: "skill"
  3: "walk"
  length: 4
  ► [[Prototype]]: Array(0)
```

Mas leonardo não tem acesso ao método country e religion que eu crie do lado de fora, já o walk eu criei tanto fora quanto dentro mas se eu o tiro de dentro da função:

```
> Object.getOwnPropertyNames(leonardo)
< ► (3) ['name', 'age', 'skill']
```

Agora ele só tem esses três porque no método está dizendo para pegar as próprias propriedades ele não fala do protótipo. Agora se eu falo com o proto:

```
> Object.getPrototypeOfNames(leonardo.__proto__)
< ▼ (4) ['constructor', 'walk', 'religion', 'country'] ⓘ
  0: "constructor"
  1: "walk"
  2: "religion"
  3: "country"
  length: 4
  ▶ [[Prototype]]: Array(0)
```

Lembrando que eu não devo acessar assim e sim desse jeito:

```
> Object.getPrototypeOfNames(Person.prototype)
< ▶ (4) ['constructor', 'walk', 'religion', 'country']
```

Então você consegue pegar quais são as propriedades com esse método de Object. Uma forma de você saber o nome do construtor é passando o tipo de dado que você está passando:

dado.constructor.name

Veremos mais adiante!

APENAS OS MÉTODOS DO PROTÓTIPO SÃO HERDADOS

```
[1,2,3].slice(); // existe
[1,2,3].from(); // não existe
```

ENTENDA O QUE ESTÁ SENDO RETORNADO

O que veremos nessa parte está relacionado também ao protótipo, mas é programação em JavaScript em geral. Sempre entenda o que está sendo retornado do valor que você chama porque é a partir do valor que está sendo retornado que você vai ter acesso aos métodos e propriedades do protótipo dele e cada tipo de dado tem um método e propriedade diferente.

```
const Car = {
```

```
    marca: 'Honda',  
    preco: 2000,  
    walk() {  
        return true;  
    }  
}
```

> Car

< ▶ {marca: 'Honda', preco: 2000, walk: f}

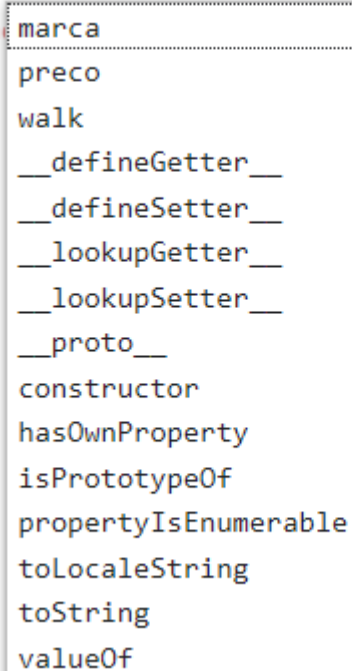
> typeof Car

< 'object'

Então Car possui acesso as propriedades e métodos do objeto:

> Car.

< 'Honda'



- marca
- preco
- walk
- __defineGetter__
- __defineSetter__
- __lookupGetter__
- __lookupSetter__
- __proto__
- constructor
- hasOwnProperty
- isPrototypeOf
- propertyIsEnumerable
- toLocaleString
- toString
- valueOf

Tudo isso está dentro de Object.prototype:

```
> Object.prototype
< {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGe
  tter__: f, ...} ⓘ
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: (...)
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
```

```
> Car.marca
```

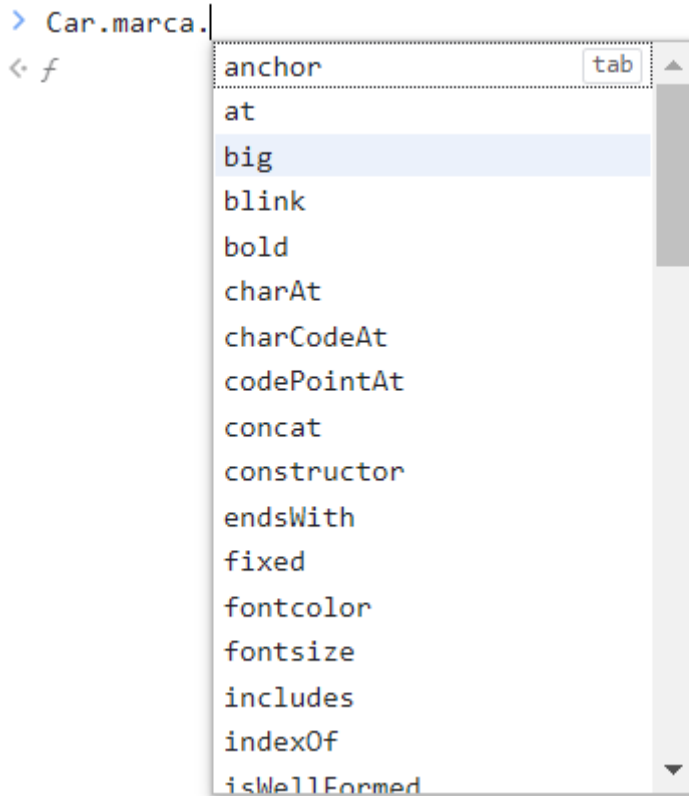
```
< 'Honda'
```

Isso não é um objeto nesse caso eu não me importo com o que vem atrás isso não se importa com o que foi feito pra chegar a isso o que importa aqui é o que retorna que é uma string. Então agora eu tenho acesso a todas as propriedades e métodos do construtor String:

> String.prototype

◀ ▼ String {'', constructor: f, anchor: f, at: f, big: f, ...} ⓘ

- ▶ anchor: f anchor()
- ▶ at: f at()
- ▶ big: f big()
- ▶ blink: f blink()
- ▶ bold: f bold()
- ▶ charAt: f charAt()
- ▶ charCodeAt: f charCodeAt()
- ▶ codePointAt: f codePointAt()
- ▶ concat: f concat()
- ▶ constructor: f String()
- ▶ endsWith: f endsWith()
- ▶ fixed: f fixed()
- ▶ fontcolor: f fontcolor()
- ▶ fontsize: f fontsize()
- ▶ includes: f includes()
- ▶ indexOf: f indexOf()
- ▶ isWellFormed: f isWellFormed()
- ▶ italics: f italics()
- ▶ lastIndexOf: f lastIndexOf()
- length: 0
- ▶ link: f Link()
- ▶ localeCompare: f localeCompare()



E não tem mais acesso a aquela parte que eu criei anteriormente do Car porque aqui retorna uma string.

```
> Car.preco  
< 2000
```

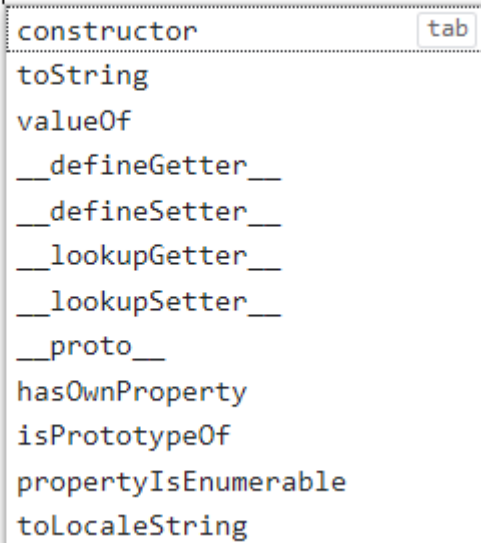
Agora preco é um número então eu tenho acesso a tudo que número tem:

```
> Number.prototype  
< ▼ Number {0, constructor: f, toExponential: f, toFixed: f, toPrecision: f, ...} ⓘ  
  ▶ constructor: f Number()  
  ▶ toExponential: f toExponential()  
  ▶ toFixed: f toFixed()  
  ▶ toLocaleString: f toLocaleString()  
  ▶ toPrecision: f toPrecision()  
  ▶ toString: f toString()  
  ▶ valueOf: f valueOf()  
  ▶ [[Prototype]]: Object  
    [[PrimitiveValue]]: 0
```

```
> Car.walk()
< true
```

walk() não é uma função quando eu faço assim eu estou executando uma função, uma função quando executada ela retorna um valor porque o que importa é o valor é o que está retornando que neste caso é true um valor booleano.

```
> Car.walk().|
< f
```



Ele me dá partes de valores booleanos e não de função.

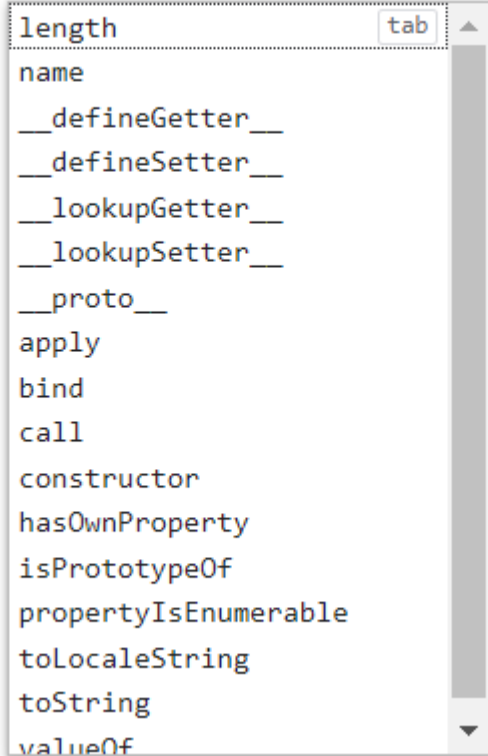
```
> Boolean.prototype
< ▼ Boolean {false, constructor: f, toString: f, valueOf: f} ⓘ
  ► constructor: f Boolean()
  ► toString: f toString()
  ► valueOf: f valueOf()
  ► [[Prototype]]: Object
  ► [[PrimitiveValue]]: false
```

Se a gente vê o protótipo de Boolean é um protótipo bem pequeno, porque Boolean true ou false não tem muita coisa pra fazer com ele, mas se você está herdando isso. Diferente se eu chamar:

```
> Car.walk
< f walk() {
    return true;
}
```

Assim, eu não executo a função eu apenas retorno a função.

```
> Car.walk.
< 0
```



`Car.walk()` tem acesso aos métodos e propriedades de funções do Constructor Function. Essas são as propriedades das funções que veremos adiante.

```
> Car.walk()
< true
```

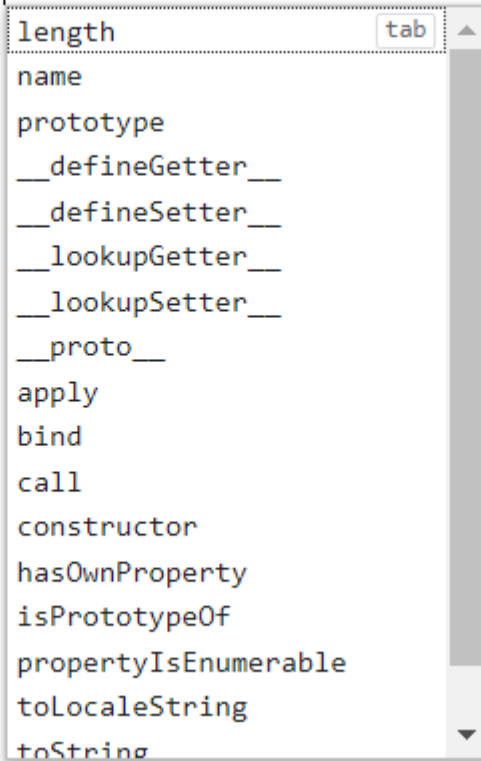
Aqui eu estou falando com o boolean. Se fosse uma string eu teria acesso a métodos de string então isso é importante é o que está retornando sempre.

Pra saber qual o tipo que está retornando uma das formas é com o constructor, todos eles tem praticamente só null e undefined que não tem:

```
> Car.walk.constructor  
↵ f Function() { [native code] }
```

Quando eu retorno o constructor ele me diz que é uma função e esse constructor possui também propriedades e métodos dentro dele:

```
> Car.walk.constructor.  
↵ 1
```



- length
- name
- prototype
- __defineGetter__
- __defineSetter__
- __lookupGetter__
- __lookupSetter__
- __proto__
- apply
- bind
- call
- constructor
- hasOwnProperty
- isPrototypeOf
- propertyIsEnumerable
- toLocaleString
- toString

Então walk() é uma função;

Agora walk() quando eu ativo é true porque ele retorna um valor boolean:

```
> Car.walk()  
↳ true
```

```
> Car.constructor.name  
↳ 'Object'
```

```
> Car.preco.constructor.name  
↳ 'Number'  
> Car.walk.constructor.name  
↳ 'Function'
```

```
> Car.preco.constructor.name  
↳ 'Number'
```

Esse valor é uma string, então se eu colocar:

```
> Car.preco.constructor.name.constructor.name  
↳ 'String'
```

Porque isso vai ser sempre uma string e se eu fizer:

```
> Car.preco.constructor.name.constructor.name.constructor.name  
↳ 'String'
```

Ele vai sempre retornar String por que ele está sempre passando String então vai ficar infinito agora.

```
Car.marca.charAt // Function  
Car.marca.charAt(0) // String
```

charAt(0) retorna o primeiro caractere de marca o H e vai retornar uma String.