



1. Apresentar o conceito de protótipos e cadeia de protótipos
2. Apresentar a estrutura de classes em Javascript

Etapa 1: Protótipos

Basicamente, eles são o esqueleto de todos os objetos então todos os objetos em JS vão herdar propriedades e métodos de um prototype.



Todos os objetos Javascript herdam propriedades e métodos de um prototype.
O objeto `Object.prototype` está no topo desta cadeia.

```
> const objeto = {}
< undefined

> objeto
< {}
  < __proto__:
    > 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__()
    > get __proto__: f __proto__()
    > set __proto__: f __proto__()
```

```
> array
< []
  < __proto__: Array(0)
    > 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()
    > 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
```

Por exemplo, sempre que a gente tem uma constante `const` em JS um objeto que é de um tipo não primitivo que é de um tipo complexo, composto ele vai ter a propriedade `__proto__` que vai ter uma série de métodos e propriedades. Então a gente consegue utilizar, por exemplo, `objeto.hasOwnProperty` pra vê se um objeto tem uma chave com aquele nome por conta do prototype porque isso está na classe pai dele mesmo que a gente não tenha escrito essa função. A gente pode converter um objeto para string porque a classe pai ou a classe `__proto__` dele, o protótipo dele

tem essa função. Então quando a gente faz a chamada de um objeto por exemplo na:

```
const objeto = {}
```

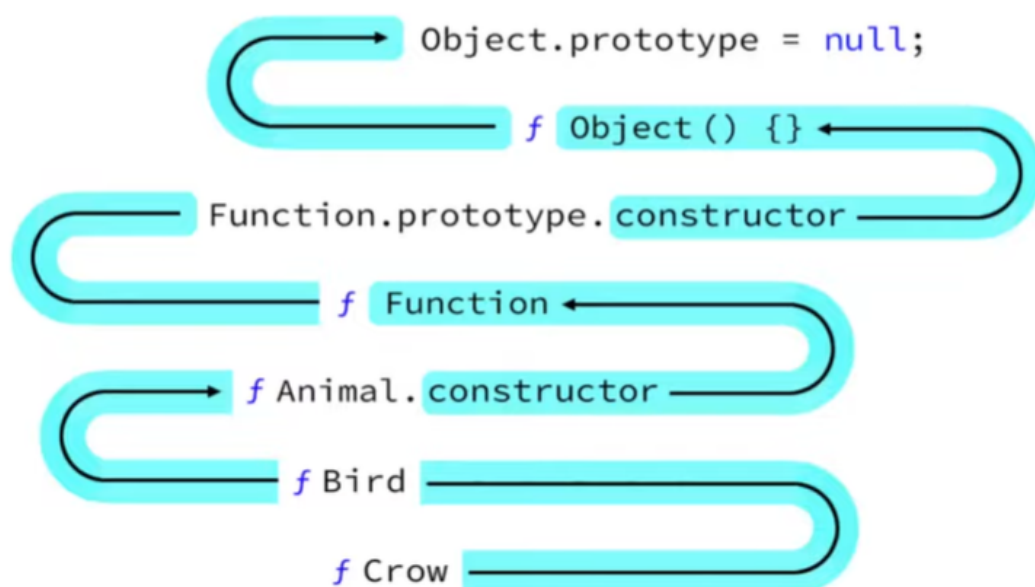
Se eu desse um `.ToString` ele tentaria procurar no meu objeto esse método, viria que não existe e então ele iria procurar no objeto ascendente a ele, no prototype dele que no caso é o `Object.prototype` se existe o método, então utilizaria ele. E ele faria essa procura pelo método até que o resultado fosse null, que acabasse a cadeia de protótipos.

No proto do Array, é o prototype do Objeto Array então ele tem todos os métodos dos Arrays. Todos os métodos do Array porque ele está herdando esse protótipo do Objeto Array que é o objeto que define arrays em JavaScript.

L
ATION

Protótipos

Cadeia de protótipos (prototype chain)

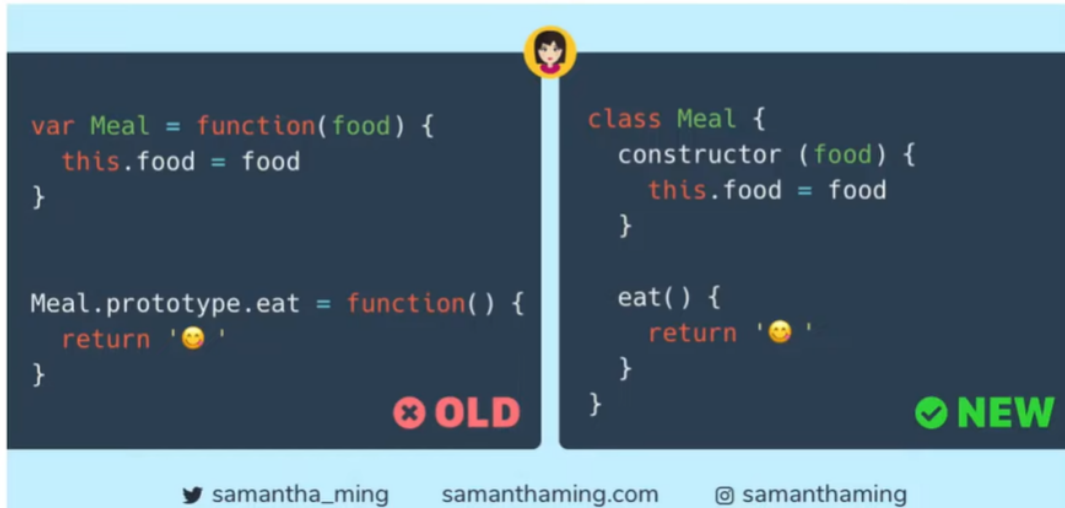


Uma imagem pra exemplificar a cadeia de protótipos onde basicamente um objeto que é um tipo de pássaro, ele herdaria de pássaro pra Animal que herdaria Function.prototype que herdaria do Object.prototype. Então a cadeia ela vai procurar as funções sempre que cada um dos objetos pai tem até chegar ao null.

Etapa 2: Classes

Classes

Syntatic sugar: uma sintaxe feita para facilitar a escrita



The image shows a comparison between two ways to write a JavaScript class-like structure. On the left, under the heading 'OLD' with a red 'X' icon, is the traditional prototype-based approach. It defines a function 'Meal' with a 'this.food' property and then adds an 'eat' method to 'Meal.prototype'. On the right, under the heading 'NEW' with a green checkmark icon, is the modern 'class' syntax. It uses the 'class' keyword, a 'constructor' method for initialization, and a regular 'eat' method. A small cartoon character is positioned at the top center between the two code blocks. At the bottom of the image, there are social media handles: a Twitter bird icon followed by 'samantha_ming', the website 'samanthaming.com', and an Instagram icon followed by '@samanthaming'.

```
var Meal = function(food) {  
  this.food = food  
}  
  
Meal.prototype.eat = function() {  
  return '😋'  
}
```

✗ OLD

```
class Meal {  
  constructor (food) {  
    this.food = food  
  }  
  
  eat() {  
    return '😋'  
  }  
}
```

✓ NEW

🐦 samantha_ming samanthaming.com @samanthaming

Classes no JavaScript não existem nativamente elas são um “açúcar sintático” que está em inglês que é basicamente uma sintaxe feita pra facilitar a escrita, mas o que acontece é que a gente usa sempre objetos e objetos que tem protótipos. Esse é o tipo de herança que existe em JavaScript nativamente, mas com o JS atualizado a partir do EcmaScript6 a gente consegue fazer essa sintaxe de classe mais parecida com outras linguagens que são feitas para acomodar o paradigma de orientação a objetos como o Java por exemplo, a gente pode escrever assim mas não é isso que está acontecendo por “baixo dos panos” o que está acontecendo são objetos e qualquer tipo de herança são feitos por protótipos.

Facilita bastante o entendimento quando a gente utiliza a sintaxe sugar de classes e é só uma explicação para sabermos que o que está acontecendo não é exatamente uma classe e sim um objeto sempre.

Classes



Javascript não possui classes nativamente. Todas as classes são objetos e a herança se dá por protótipos.



```

1 class Animal {
2   constructor(type = 'animal') {
3     this.type = type
4   }
5
6   get type() {
7     return this._type
8   }
9
10  set type(val) {
11    this._type = val.toUpperCase()
12  }
13
14  makeSound() {
15    console.log('Making animal sound')
16  }
17 }
18
19 let a = new Animal()
20 console.log(a.type) //ANIMAL
21

```

```

1 class Cat extends Animal {
2   constructor() {
3     super('cat')
4   }
5
6   makeSound() {
7     super.makeSound()
8     console.log('Meow!')
9   }
10 }
11
12 let b = new Cat()
13 console.log(b.type) //CAT
14

```

Aqui a gente tem um print de uma classe chamada Animal e da classe filha dele chamada Cat e tem a anatomia de uma classe em JS. Então a classe em JS sempre tem um constructor onde ele vai construir a classe e se você passar parâmetros ele vai atribuir esses parâmetros a certos valores dentro dessa classe, então o tipo do construtor é animal ele colocou um parâmetro default e se não for passado nada o tipo vai ser animal e se for passado alguma coisa vai ser esse outro tipo, esse é o construtor.

Tem também getter e setter pra gente ter acesso a propriedades de um objeto, a gente usa uma sintaxe pra poder setar esses objetos então o:

get type

Que é o tipo do animal ele vai retornar o tipo.

set type

Vai ser pra gente determinar qual tipo tem um outro valor.

Então esses são métodos que a gente pode utilizar é importante que a gente faça dessa forma porque aí a gente tem a questão do encapsulamento que as classes elas são realmente apenas dessa classe e não tem outras classes que manipulam ela então isso é muito importante. E a gente também tem as classes filhas então a classe filha de cat ela tem um método chamado *super* ele vai mandar pra cima, pra função pai os parâmetros que estão ali então quando eu falo *super('cat')* eu estou construindo o cat e no momento que ele for construído a gente vai utilizar a classe *Animal* onde o *type* = 'animal' que o default é 'animal', mas a gente está mandando

cat então o tipo vai ser cat. O super ele serve pra gente utilizar as propriedades do construtor que existem na classe pai.

E também tem o método que é o *makeSound*, a gente pode sempre sobrescrever métodos então a classe pai tem o makeSound mas se eu quero que a classe filha tenha um comportamento diferente dentro desse método eu posso sempre sobrescrever e vai funcionar.

Então basicamente essa é a anatomia de uma classe no JS, você tem os construtores, getters e setters, o método super() que é muito importante quando você quer fazer herança e os métodos específicos pra cada classe que no exemplo foi o makeSound.