Prototype - Parte 1

É como se fosse o núcleo do JavaScript tudo aquilo que vimos sobre acessando os métodos de outros objetos, métodos que a gente nunca criou isso só existe por causa dos protótipos.

O Protótipo sempre está ligado a função ele é uma propriedade das funções então você criou uma função nova ela vai ter essa propriedade *prototype* dela então você acessa o nome da função.prototype. Essa propriedade é um objeto que retorna dela.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
const leonardo = new Pessoa('Leonardo', 19);

console.log(Person.prototype); // retorna o objeto
console.log(leonardo.prototype); // undefined
```

Aqui eu criei uma função construtora, mas qualquer função tem o prototype.

No primeiro console.log ele retorna o objeto do protótipo se eu colocar o meu nome direto com o prototype vai dar undefined porque leonardo é um objeto e Person é a função o prototype está só na função sempre.

```
function test() {
  return 1 + 1;
}
```

```
> test.prototype
< ▼ {constructor: ƒ} i
    ▶ constructor: ƒ test()
    ▶ [[Prototype]]: Object
```

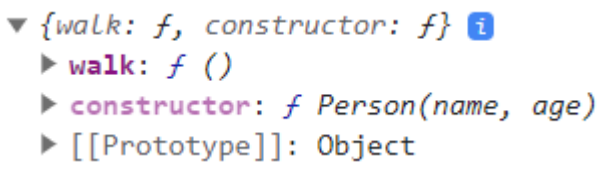
```
> typeof test.prototype
< 'object'
```

Mas nesse momento esse objeto só tem uma propriedade a *constructor* que diz quem construiu isso e o proto que está vazio por enquanto.

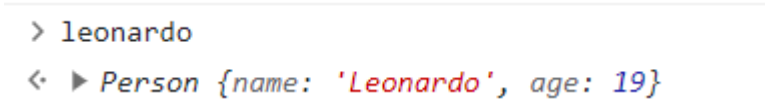
Funcao.Prototype

Eu posso pegar agora o nome da função e adicionar métodos ou propriedades a esse prototype específico:

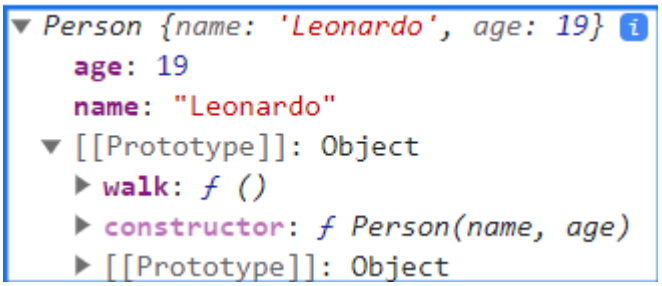
```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

Person.prototype.walk = function() {
  return 'The person walked under the sidewalk';
}

const leonardo = new Person('Leonardo', 19);

console.log(Person.prototype);
console.log(leonardo.prototype);
```



Além do constructor agora ele passa a ter o método walk.



Se eu procurar o método na variável leonardo nessa pessoa que eu criei o método ele não vai ter o método walk, mas se eu for em __proto__ sim:



Então eu posso acessar:

```
> leonardo.walk()
'The person walked under the sidewalk'
```

E eu posso passar as propriedades que eu criei na minha função construtora:

```
Person.prototype.walk = function() {
  return `${this.name} walked under the sidewalk`;
}
```

Então ele consegue acessar a propriedade name através do prototype e como se trata de uma função eu posso acessá-la diretamente:

```
> Person.prototype.walk()
'undefined walked under the sidewalk'
```

Como eu não criei o objeto o name não existe essa função vai ser como *undefined*. Lembrando que eu estou acessando direto da função Person então a variável leonardo não vai ter influência nenhuma.

E aí eu posso ir adicionando quantos eu quiser:

```
Person.prototype.religion = function() {
  return `${this.name} is Catholic`;
}

Person.prototype.country = function() {
  return `${this.name} lives in Brazil`;
}
```

E o interessante quando criamos uma função Construtora assim é que o protótipo é único se eu tivesse colocado uma função dentro da minha função construtora:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.skill = function() {
    return `${this.name} is good at communication`;
  }
}
```

```
> leonardo
< ▼ Person {name: 'Leonardo', age: 19, skill: ƒ}
      age: 19
      name: "Leonardo"
    ▶ skill: ƒ ()
    ▶ [[Prototype]]: Object
```

Agora a função skill está dentro direto de leonardo ele ainda tem os protótipos.

Se eu fazer:

```
> leonardo.skill()
< 'Leonardo is good at communication'
```

Vai funcionar da mesma forma;

A diferença é que toda vez que eu criar um objeto novo essa função também vai ser criada nova então vai estar instanciando essa função toda vez que eu criar um objeto novo. Nesse caso que eu coloco no protótipo direto não, isso só vai criar uma vez essa função que o objeto novo criado vai ter acesso a ela.

Então eu posso ter a mesma função:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.skill = function() {
    return `${this.name} is good at communication`;
  }
  this.walk = function() {
    return `I walked over the object`;
  }
}
```

```
> leonardo.walk()
< 'I walked over the object'
```

E existe a mesma função no protótipo o que acontece é ele primeiro procura dentro do objeto se existe o método se não existe ele vai no protótipo e não tendo em nenhum dos dois ocorre um erro.

Então, é possível adicionar novas propriedades e métodos ao objeto prototype.

Acesso ao Prototype

O objeto que eu criei com o Construtor Person possui acesso a métodos e propriedades que estão dentro daquele objeto prototype da função.

Proto

Então, o objeto criado utilizando o construtor, possui acesso aos métodos e propriedades do protótipo deste construtor. Lembrando, prototype é uma propriedade de funções apenas.

```
const leonardo = new Person('Leonardo', 19);

leonardo.name;
leonardo.age;
leonardo.walk();
leonardo.religion();
```

Qualquer coisa que a gente você for passar que for um objeto vai sempre aparecer o __proto__ ele é o protótipo só que ele é o protótipo criado no momento.

```
> leonardo.__proto__
< ▶ {walk: ƒ, religion: ƒ, country: ƒ, constructor: ƒ}
```

Ele vai me dar o objeto que está dentro dele.

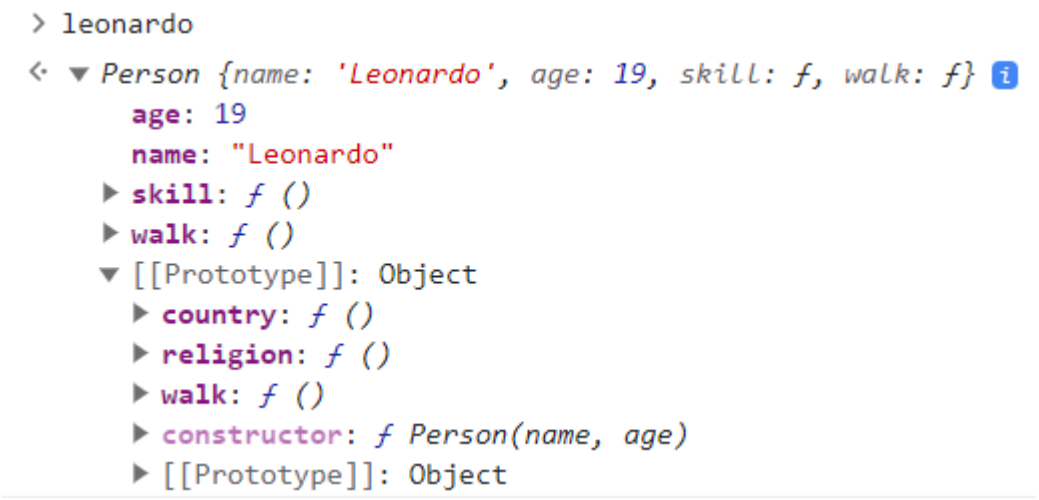
Eu poderia falar:

```
> leonardo.__proto__.walk()
< 'undefined walked under the sidewalk'
```

Agora temos um problema. Em leonardo acessamos o proto e o walk() e o resultado foi undefined porque ele acessou direto a função sem antes passar pela construção da função então ele não pegou o que é *this.name*.

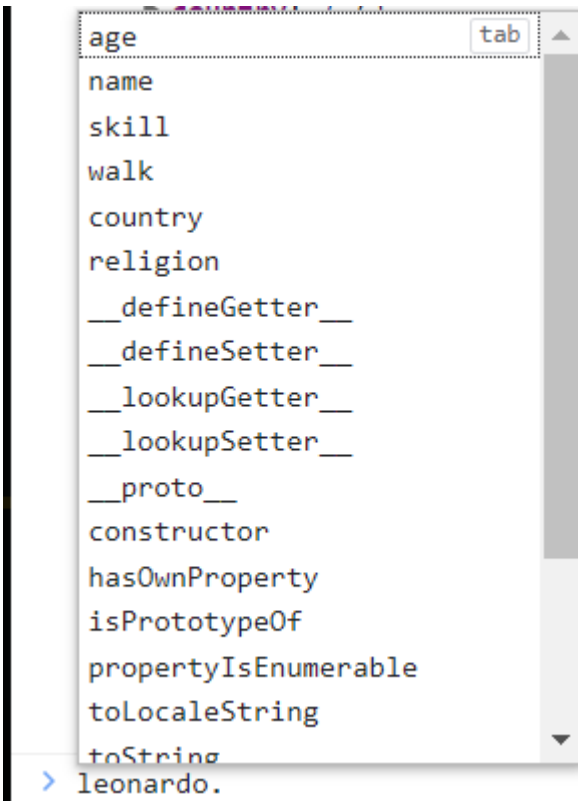
Essa forma que acessamos o proto nós nunca vamos utilizar no nosso código isso é algo interno que o próprio Chrome está gerando pra nós a partir do prototype daquela função que a gente tinha. Então isso vai ser trabalho dele saber que se eu tentar acessar algum método e ele não existir na função construtora ele vai tentar procurar no protótipo e ele vai encadeando isso é uma coisa muito boa do protótipo

ele encadea o protótipo se você entra em um proto você vai ter outro proto dentro dele:



O primeiro pertence a função construtora Person e o próximo é do constructor Object. Então ele consegue acessar também isso do objeto:

Se eu colocar:



Aparece diversos métodos e propriedades que eu nunca defini no meu código da minha máquina.

Exemplo:

```
> leonardo.hasOwnProperty
ƒ
```

Ele possui isso porque está na cadeia de protótipo dele o que essa função faz não importa agora pra gente, mas ele tem acesso a isso porque está no protótipo dele.

É papel da engine fazer essa busca, não devemos falar com __proto__ diretamente. Se você uma pessoa fazendo isso a pessoa provavelmente não sabe o que tá fazendo ou ela sabe e é um código antigo e aí ela precisou ter acesso a isso.

```
// Acessam o mesmo método
// mas __proto__ não terá
// acesso ao this.nome
leonardo.walk();
leonardo.__proto__.walk();
```

Herança de Protótipo

O objeto possui acesso aos métodos e propriedades do protótipo do construtor responsável por criar este objeto. O objeto abaixo possui acesso a métodos que nunca definimos, mas são herdados do protótipo de Object.
O constructor que cria o leonardo é aquela função Object, mas o constructor que cria a própria função é um objeto também então por isso que eu tenho acesso a:

```
Object.prototype;
leonardo.toString();
leonardo.isPrototypeOf();
leonardo.valueOf();
```

São métodos do protótipo do constructor Object.

```
> Object
< ƒ Object() { [native code] }
```

Assim a gente tem acesso ao constructor assim como Person. Aqui, ele retorna uma função também mas ele não dá o código que está dentro de {} no caso de Person sim, mas aqui não. Esse Constructor Object está sendo criado com a linguagem nativa da engine deles então ele não nem mostra o código que está dentro porque é irrelevante, não importa para nós. O que importa são os métodos que esse constructor tem no protótipo dele:

```
> Object.prototype
< {constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __looku
  pGetter__: ƒ, …}
    constructor: ƒ Object()
    hasOwnProperty: ƒ hasOwnProperty()
    isPrototypeOf: ƒ isPrototypeOf()
    propertyIsEnumerable: ƒ propertyIsEnumerable()
    toLocaleString: ƒ toLocaleString()
    toString: ƒ toString()
    valueOf: ƒ valueOf()
    __defineGetter__: ƒ __defineGetter__()
    __defineSetter__: ƒ __defineSetter__()
    __lookupGetter__: ƒ __lookupGetter__()
    __lookupSetter__: ƒ __lookupSetter__()
    __proto__: (...)
    get __proto__: ƒ __proto__()
    set __proto__: ƒ __proto__()
```

São aquelas mesmas funções que a gente acessa com a variável leonardo.