# Orientação a objetos

## Protótipos de objetos

Protótipos são o mecanismo pelo qual objetos JavaScript herdam recursos uns dos outros. Agora vamos entender como as cadeias de protótipos funcionam e observamos como a propriedade prototype pode ser usada para adicionar métodos aos construtores existentes.

O JavaScript é frequentemente descrito como uma linguagem baseada em protótipos — para fornecer herança, os objetos podem ter um objeto de protótipo, que atua como um objeto de modelo do qual herda métodos e propriedades.

O objeto de protótipo de um objeto também pode ter um objeto de protótipo, do qual herda métodos e propriedades, e assim por diante. Isso geralmente é chamado de cadeia de protótipos e explica por que objetos diferentes têm propriedades e métodos definidos em outros objetos disponíveis para eles.

Bem, para ser exato, as propriedades e os métodos são definidos na propriedade prototype nas funções construtoras dos Objetos, não nas próprias instâncias do objeto.

Em JavaScript, é feito um link entre a instância do objeto e seu protótipo (sua propriedade **proto**, que é derivada da propriedade prototype no construtor), e as propriedades e os métodos são encontrados percorrendo a cadeia de protótipos.

## Noções básicas

Vamos construir Person() com a seguinte função construtora:

**Função construtora**

```
function Person(first, last, age, gender, interests) {
        this.name = {
          'first': first,
          'last' : last
        };
        this.age = age;
        this.gender = gender;
        this.interests = interests;
}
```

Nós criamos então uma instância de objeto como esta:

**Instância**

```
var person1 = new Person('Bob', 'Smith', 32, 'male', ['music', 'skiing']);
```

Se você digitar "person1." em seu console JavaScript, você deve ver o navegador tentar concluir automaticamente isso com os nomes de membros disponíveis neste objeto.

O que acontece se você chamar um método em person1, que é realmente definido em Object? Por exemplo: `person1.valueOf()`

Este método — `Object.valueOf()` é herdado por person1 porque seu construtor é `Person()`, e o protótipo de `Person()` é `Object()`. `valueOf()` retorna o valor do objeto em que é chamado — experimente e veja! Nesse caso, o que acontece é:

- O navegador verifica inicialmente se o objeto person1 tem um método valueOf() disponível nele, conforme definido em seu construtor, Person().
- Se não tem, então o navegador verifica se o objeto (Object()) de protótipo do construtor Person() tem um método valueOf() disponível, então ele é chamado, e tudo está bem!

## A propriedade prototype

Então, onde estão as propriedades e os métodos herdados definidos? Se você observar a página de referência do Object, verá, à esquerda, um grande número de propriedades e métodos — muito mais do que o número de membros herdados que vimos disponíveis no objeto person1. Alguns são herdados e outros não — por que isso acontece?

Como mencionado acima, os herdados são os definidos na propriedade prototype (você poderia chamá-lo de um subespaço de nomes) — ou seja, aqueles que começam com Object.prototype., e não os que começam com apenas Object. O valor da propriedade prototype é um objeto, que é basicamente um bucket para armazenar propriedades e métodos que queremos que sejam herdados por objetos mais abaixo na cadeia de protótipos.

Portanto, `Object.prototype.watch()`, `Object.prototype.valueOf()`, etc., estão disponíveis para qualquer tipo de objeto que herda de `Object.prototype`, incluindo novas instâncias de objeto criadas a partir do construtor `Person()`.

`Object.is()`, `Object.keys()`, e outros membros não definidos dentro do bloco prototype não são herdados por instâncias de objetos ou tipos de objetos que herdam de `Object.prototype`. Eles são métodos / propriedades disponíveis apenas no próprio construtor `Object()`.

1- Você pode conferir as propriedades de protótipo existentes para si mesmo — volte ao nosso exemplo anterior e tente inserir o seguinte no console JavaScript:

**Prototype**

```
Person.prototype
```

2- A saída não mostrará muito porque não definimos nada no protótipo do nosso construtor personalizado! Por padrão, o prototype de um construtor sempre começa vazio. Agora tente o seguinte:

**Prototype**

```
Object.prototype
```

Você verá outros exemplos de herança de cadeia de protótipos em todo o JavaScript — tente procurar os métodos e propriedades definidos no protótipo dos objetos globais String, Date, Number, e Array, por exemplo. Estes todos têm um número de membros definidos em seu protótipo, e é por isso que, por exemplo, quando você cria uma string, como esta:

**String**

```
var myString = 'This is my string.';
```

`myString` imediatamente tem vários métodos úteis disponíveis, como `split()`, `indexOf()`, `replace()`, etc.

## create()

O método Object.create() pode ser usado para criar uma nova instância de objeto.

- Por exemplo, tente isso no console JavaScript do seu exemplo anterior:

  `var person2 = Object.create(person1);`

- O que `create()` realmente faz é criar um novo objeto a partir de um objeto de protótipo especificado. Aqui, `person2` está sendo criado usando `person1` como um objeto de protótipo. Você pode verificar isso inserindo o seguinte no console:

  `person2.__proto__` Isso retornará o `person1`.

## A propriedade do construtor

Toda função de construtor possui uma propriedade prototype cujo valor é um objeto que contém uma propriedade constructor.

Esta propriedade construtora aponta para a função construtora original.

Portanto, a propriedade constructor também está disponível para os objetos person1 e person2.

- Por exemplo, tente estes comandos no console:

  `person1.constructor;` e `person2.constructor;`

  Estes devem retornar o construtor Person(), pois contém a definição original dessas instâncias.

A propriedade do constructor tem outros usos. Por exemplo, se você tiver uma instância de objeto e desejar retornar o nome do construtor do qual ela é uma instância, use o seguinte: `instanceName.constructor.name`.

## Herança em JavaScript

Agora vamos aprender como criar classes de objetos "child" (construtores) que herdam recursos de suas classes "parent".

### Herança Prototipada

Até agora vimos alguma herança em ação — vimos como funcionam as cadeias de protótipos e como os membros são herdados subindo em uma cadeia. Mas principalmente isso envolveu funções internas do navegador. Como criamos um objeto em JavaScript que herda de outro objeto?

Vamos explorar como fazer isso com um exemplo concreto.

Primeiro de tudo, faça uma cópia local do arquivo oojs-class-inheritance-start.html. Aqui dentro você encontrará o mesmo exemplo de construtor Person() que utilizamos antes. Os métodos são todos definidos no protótipo do construtor. Por exemplo:

**Example**

```
Person.prototype.greeting = function() {
  alert('Hi! I\'m ' + this.name.first + '.');
};
```

Digamos que quiséssemos criar uma classe Teacher, que herda todos os membros de Person, mas também inclui:

- Uma nova propriedade, subject — isso irá conter o assunto que o professor ensina.
- Um método greeting() atualizado, que soa um pouco mais formal do que o método padrão greeting() — mais adequado para um professor que se dirige a alguns alunos da escola.

## Definindo uma função construtora Teacher()

A primeira coisa que precisamos fazer é criar um construtor Teacher() — adicione o seguinte abaixo do código existente:

**Teacher**

```
function Teacher(first, last, age, gender, interests, subject) {
  Person.call(this, first, last, age, gender, interests);

  this.subject = subject;
}
```

Isto parece similar ao construtor `Person` de várias maneiras, mas há algo estranho aqui que nós não vimos antes — a função `call()`. Esta função basicamente permite chamar uma função definida em outro lugar, mas no contexto atual.

O primeiro parâmetro especifica o valor `this` que você deseja usar ao executar a função, e os outros parâmetros são aqueles que devem ser passados para a função quando ela é invocada.

Nós queremos que o construtor `Teacher()` pegue os mesmos parâmetros que o construtor `Person()` de onde ele está herdando, então especificamos todos eles como parâmetros na chamada `call()`.

A última linha dentro do construtor simplesmente define a nova propriedade subject que os professores terão, que pessoas genéricas não possuem.

## Definindo o protótipo e referência de construtor do Teacher()

Tudo está bem até agora, mas nós temos um problema. Nós definimos um novo construtor, e ele tem uma propriedade prototype, que por padrão apenas contém uma referência à própria função construtora.

O novo construtor também não herda todos os métodos. Para ver isso, compare as saídas de Person.prototype.greeting e Teacher.prototype.greeting. Precisamos obter `Teacher()` para herdar os métodos definidos no protótipo `Person()`. Então, como fazemos isso?

- Adicione a seguinte linha abaixo da sua adição anterior:

**Herança**

```
Teacher.prototype = Object.create(Person.prototype);

Object.defineProperty(Teacher.prototype, 'constructor', {
    value: Teacher,
    enumerable: false, // so that it does not appear in 'for in' loop
    writable: true });
```

- Aqui nosso amigo `create()` vem para o resgate novamente. Nesse caso, estamos usando para criar um novo objeto e torná-lo o valor de `Teacher.prototype`. O novo objeto tem `Person.prototype` como seu protótipo e, portanto, herdará, se e quando necessário, todos os métodos disponíveis no `Person.prototype`.

Agora que você digitou todo o código, tente criar uma instância de objeto do Teacher() colocando o seguinte na parte inferior do seu JavaScript (ou algo semelhante à sua escolha):

**Teste**

```
var teacher1 = new Teacher('Dave', 'Griffiths', 31, 'male', ['football',
'cookery'], 'mathematics');
```

A técnica que abordamos aqui não é a única maneira de criar classes herdadas em JavaScript, mas funciona bem e dá uma boa idéia sobre como implementar a herança em JavaScript.

Mais a frente também vamos em conferir alguns dos novos recursos ECMAScript que nos permitem fazer herança mais claramente em JavaScript.

## Classes com ES6

O ECMAScript 2015 introduz a sintaxe de classe em JavaScript como uma maneira de escrever classes reutilizáveis usando uma sintaxe mais fácil e mais limpa, que é mais semelhante a classes em C ++ ou Java.

Nesta seção, converteremos os exemplos Pessoa e Professor da herança protótipo para as classes, para mostrar como é feito.

Vejamos uma versão reescrita do exemplo Person, estilo de classe:

**Classe**

```
class Person {
  constructor(first, last, age, gender, interests) {
    this.name = {
      first,
      last
    };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }

  greeting() {
    console.log(`Hi! I'm ${this.name.first}`);
  };

  farewell() {
    console.log(`${this.name.first} has left the building. Bye for
now!`);
  };
}
```

A declaração **class** indica que estamos criando uma nova classe. Dentro deste bloco, definimos todos os recursos da classe:

- O método `constructor()` define a função construtora que representa nossa classe Person.
- `greeting()` e `farewell()` são métodos de classe. Quaisquer métodos que você deseja associar à classe são definidos dentro dela, após o construtor. Neste exemplo, usamos template literals em vez de concatenação de string para facilitar a leitura do código.

Agora podemos instanciar instâncias de objeto usando o operador new:

**New**

```
let han = new Person('Han', 'Solo', 25, 'male', ['Smuggling']);
han.greeting();
// Hi! I'm Han

let leia = new Person('Leia', 'Organa', 19, 'female', ['Government']);
leia.farewell();
// Leia has left the building. Bye for now
```

## Herança com sintaxe de classe

Acima nós criamos uma classe para representar uma pessoa. Eles têm uma série de atributos que são comuns a todas as pessoas; Nesta seção, criaremos nossa classe especializada `Teacher` , tornando-a herdada de `Person` usando a sintaxe de classe moderna. Isso é chamado de criação de uma subclasse ou subclasse.

Para criar uma subclasse, usamos a palavra-chave `extends` para informar ao JavaScript a classe na qual queremos basear nossa classe.

**extends**

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    this.name = {
      first,
      last
    };

  this.age = age;
  this.gender = gender;
  this.interests = interests;
  // subject and grade are specific to Teacher
  this.subject = subject;
  this.grade = grade;
  }
}
```

Podemos tornar o código mais legível definindo o operador `super` como o primeiro item dentro do `constructor()` . Isso chamará o construtor da classe pai e herdará os membros que especificarmos como parâmetros de `super()` , desde que sejam definidos lá:

**super**

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    super(first, last, age, gender, interests);

    // subject and grade are specific to Teacher
    this.subject = subject;
    this.grade = grade;
  }
}
```

Quando instanciamos instâncias de objeto `title` , podemos agora chamar métodos e propriedades definidos em `Teacher` e `Person` , como seria de esperar:

**Instace**

```
let snape = new Teacher('Severus', 'Snape', 58, 'male', ['Potions'], 'Dark arts', 5);
snape.greeting(); // Hi! I'm Severus.
snape.farewell(); // Severus has left the building. Bye for now.
snape.age // 58
snape.subject; // Dark arts
```

## Getters e Setters

Pode haver momentos em que queremos alterar os valores de um atributo nas classes que criamos ou não sabemos qual será o valor final de um atributo. Usando o exemplo `Teacher`, podemos não saber o assunto que o professor ensinará antes de criá-lo, ou o assunto pode mudar entre os termos.

Podemos lidar com essas situações com getters e setters.

Vamos melhorar a classe Professor com getters e setters. A aula começa da mesma forma que foi a última vez que olhamos para ela.

Os getters e setters trabalham em pares. Um getter retorna o valor atual da variável e seu setter correspondente altera o valor da variável para o que ela define.

**get/set**

```
class Teacher extends Person {
  constructor(first, last, age, gender, interests, subject, grade) {
    super(first, last, age, gender, interests);
    // subject and grade are specific to Teacher
    this._subject = subject;
    this.grade = grade;
  }

  get subject() {
    return this._subject;
  }

  set subject(newSubject) {
    this._subject = newSubject;
  }
}
```

Em nossa classe acima, temos um getter e setter para a propriedade `subject`. Usamos _ para criar um valor separado no qual armazenar nossa propriedade de nome. Sem usar essa convenção, obteríamos erros toda vez que chamássemos get ou set. Neste ponto:

- Para mostrar o valor atual da propriedade `_subject` do objeto `snape` , podemos usar o método getter `snape.subject` .
- Para atribuir um novo valor à propriedade `_subject` , podemos usar o método setter `snape.subject="new value"` .

O exemplo abaixo mostra os dois recursos em ação:

**example**

```
// Check the default value
console.log(snape.subject) // Returns "Dark arts"

// Change the value
snape.subject="Balloon animals" // Sets _subject to "Balloon animals"

// Check it again and see if it matches the new value
console.log(snape.subject) // Returns "Balloon animals"
```

As classes são basicamente uma sintaxe compacta para configurar cadeias de protótipos. Observe que não precisamos de classes para criar objetos. Também podemos fazer isso por meio de objetos literais . É por isso que o padrão singleton não é necessário em JavaScript e as classes são menos usadas do que em muitas outras linguagens que as possuem.

**Classe**

```
class Person {
  #name;

  constructor(first, last, age, gender, interests) {
    this.#name = {
      first,
      last
    };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }

  greeting() {
    console.log(`Hi! I'm ${this.#name.first}`);
  };

  farewell() {
    console.log(`${this.#name.first} has left the building. Bye for
now!`);
  };

  static extractNames(persons) {
    return persons.map(person => person.#name);
  }
```

```
25
26  }
```

**Info**

Array.map

- `#name` é um campo privado de instância : Esses campos são armazenados em instâncias. Eles são acessados de forma semelhante às propriedades, mas seus nomes são separados – eles sempre começam com símbolos de hash ( #). E eles são invisíveis para o mundo fora da classe.
- Antes de podermos inicializar .#name no construtor , precisamos declará-lo mencionando-o no corpo da classe.
- `.extractNames()` é um método estático . “Estático” significa que pertence à classe, não às instâncias: `Person.extractNames(han,leia)` .

**Exercício**

- Crie os getters e setters da classe Person
- Crie uma classe "Usuario" que deve receber dois parâmetros no método construtor, e-mail e senha. Ambos privados.
  - Crie os getters e settes da classe "Usuario"
  - Crie uma classe com nome "Admin", essa classe deve extender uma a classe "Usuario".
  - A classe "Admin" por sua vez não recebe parâmetros próprios apenas e-mail e senha da classe pai (super). Deve ser criada uma propriedade interna admin que recebe true em seu construtor.
  - Agora com suas classes criadas, adicione um método (função) na classe Usuario chamado isAdmin que retorna se o usuário é administrador ou não baseado na propriedade admin ser true ou não.

**Info**

Fonte: MDN web docs