Object Oriented Programming/ JAVASCRIPT

Criar um objeto JavaScript básico

Pense sobre as coisas que as pessoas veem todos os dias, como carros, lojas e pássaros. Tudo isso são *objetos*: coisas tangíveis com que pessoas podem observar e interagir.

Quais são algumas qualidades destes objetos? Um carro possui rodas. Lojas vendem itens. Pássaros possuem asas.

Estas qualidades, ou *propriedades*, definem o que faz um objeto. Note que objetos similares compartilham de propriedades iguais, mas podem ter diferentes valores para estas propriedades. Por exemplo, todos os carros possuem rodas, mas nem todos os carros possuem o mesmo número de rodas.

Objetos em JavaScript são utilizados como modelos de objetos do mundo real, dando a eles propriedades e comportamentos assim como se fossem análogos ao mundo real. Aqui está um exemplo utilizando estes conceitos para a criação de um objeto `duck`:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
```

O objeto `duck` possui dois pares de propriedades/valores: um `name` sendo `Aflac` e um `numLegs` sendo 2.

Exercicio = Cria um objeto `dog` com as propriedades `name` e `numLegs`, e definem eles como sendo do tipo string e numérico, respectivamente.

```
let dog = {
  name: "blair",
  numLegs: 4


};
```

Usar notação de ponto para acessar as propriedades de um objeto

O último desafio criou um objeto com várias propriedades. Agora você verá como acessar os valores dessas propriedades. Exemplo:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
console.log(duck.name);
```

A notação de ponto é utilizada no nome do objeto, `duck`, seguida pelo nome da propriedade, `name`, para acessar o valor de `Aflac`.

Exercicio = Exiba ambas as propriedades do objeto `dog` no seu console.

```
let dog = {
  name: "Spot",
  numLegs: 4
};
// Altere apenas o código abaixo desta linha
console.log(dog.name)
console.log(dog.numLegs)
```

Criar um método em um objeto

Objetos podem ter um tipo especial de propriedade, chamado de *método*.

Métodos e propriedades são funções. Isso adiciona diferentes comportamentos para um objeto. Aqui está o exemplo `duck` com um método:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " + duck.name +
"."}
};
```

```
duck.sayName();
```

O exemplo adiciona o método `sayName`, que é uma função responsável por retornar uma frase contendo o nome do `duck`. Note que o método acessou a propriedade `name` na declaração de retorno usando `duck.name`. O próximo desafio vai mostrar outra forma de fazer isso.

Exercicio = Usando o objeto `dog`, de a ele um método chamado `sayLegs`. O método deve retornar a frase `This dog has 4 legs. (Este cachorro possui 4 pernas.)`

```
let dog = {
  name: "Spot",
  numLegs: 4,
 sayLegs: function()  {return "This dog has " + dog.numLegs + " legs.";}
  };

dog.sayLegs();
```

Fazer código mais reutilizável com a palavra-chave this

O último desafio introduziu um método ao objeto `duck`. Ele utiliza a notação de ponto (`duck.name`) para acessar o valor da propriedade `name` dentro da declaração de retorno:

```
sayName: function() {return "The name of this duck is " + duck.name + ".";}
```

Enquanto isso é uma forma válida de acessar a propriedade do objeto, tem uma armadilha aqui. Se o nome da variável mudar, qualquer código referenciando o nome original seria necessário ser atualizado também. Em uma definição curta de objeto, isso não é um problema, mas se um objeto

possui muitas referencias para suas propriedades, há uma chance maior de erro.

Uma forma para evitar estes problemas é utilizar a palavra-chave `this`:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " +
this.name + ".";}
};
```

`this` é um tópico bem extenso e o exemplo acima é apenas uma forma de utilizá-lo. No contexto atual, `this` refere-se ao objeto que o método está associado à: `duck`. Se o nome do objeto é alterado para `mallard`, não é necessariamente para encontrar todas as referencias para `duck` no código. Isso torna o código reutilizável e legível.

Exercicio = Modifique o método `dog.sayLegs` para remover qualquer referencia para `dog`. Utilize o exemplo `duck` como guia.

```
let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs: function() {return "This dog has " + this.numLegs
+ " legs.";}
};


dog.sayLegs();
```

Definir uma função construtora

Construtores são funções que criam novos objetos. Eles definem propriedades e comportamentos que pertencerão ao novo objeto. Pense neles como uma planta para a criação de novos objetos.

Aqui está um exemplo de construtor:

```
function Bird() {
  this.name = "Albert";
  this.color = "blue";
  this.numLegs = 2;
}
```

O construtor define um objeto `Bird` com propriedades `name`, `color`, e `numLegs` definidos como Albert, blue e 2, respectivamente. Construtores seguem algumas convenções:

- Construtores são definidos com a primeira letra do nome maiúscula para distinguir eles de outras funções que não são `constructors`.
- Construtores usam a palavra-chave `this` para definir as propriedades do objeto que eles criarão. Dentro do construtor, `this` referencia para um novo objeto quer vai ser criado.
- Construtores definem propriedades e comportamentos em vez de retornar valores como outras funções podem fazer.

Exercicio = Crie um construtor, `Dog`, com as propriedades `name`, `color` e `numLegs` que são definidos como string, string e número, respectivamente.

```
function Dog() {
  this.name = "Blair";
  this.color = "white";
  this.numLegs = 4;
}
```

Usar um construtor para criar objetos

Aqui está o construtor de `Bird` do desafio anterior:

```
function Bird() {
  this.name = "Albert";
  this.color  = "blue";
  this.numLegs = 2;
}

let blueBird = new Bird();
```

Observação: `this` dentro do construtor sempre refere-se ao objeto sendo criado.

Note que o operador `new` é usado quando chamamos o construtor. Isso avisa ao JavaScript para criar uma nova instância de `Bird` chamado `blueBird`. Sem o operador `new`, `this` dentro do construtor não iria apontar para o objeto recentemente criado, dando resultados inesperados. Agora `blueBird` possui todas as propriedades definidas dentro do construtor `Bird`:

```
blueBird.name;
blueBird.color;
blueBird.numLegs;
```

Assim como qualquer outro objeto, suas propriedades podem ser acessadas e modificadas:

```
blueBird.name = 'Elvira';
blueBird.name;
```

Exercicio = Utilize o construtor de `Dog` da última lição para criar uma nova instância de `Dog`, atribuindo a instância para a variável `hound`.

```
function Dog() {
  this.name = "Rupert";
  this.color = "brown";
  this.numLegs = 4;
}
```

```
// Altere apenas o código abaixo desta linha

let hound = new Dog();
```

Estender construtores para receber argumentos

Os construtores `Bird` e `Dog` do último desafio funcionaram bem. No entanto, note que todos os `Birds` que são criados com o construtor `Bird` são automaticamente nomeados Albeart, são da cor azul e possuem duas pernas. E se você deseja pássaros com diferentes valores para seus nomes e cores? É possível alterar estas propriedades de cada pássaro manualmente, mas isso daria bastante trabalho:

```
let swan = new Bird();
swan.name = "Carlos";
swan.color = "white";
```

Suponha que você está escrevendo um programa para registrar centenas ou até milhares de diferentes pássaros em um aviário. Seria necessário muito tempo para criar todos estes pássaros, e então alterar as propriedades para os diferentes valores de cada um. Para criar diferentes objetos `Bird` de forma mais fácil, você pode projetar o construtor de Bird para aceitar parâmetros:

```
function Bird(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 2;
}
```

Em seguida, passe os valores como argumentos para definir cada pássaro único no construtor `Bird`: `let cardinal = new Bird("Bruce","red");` Isso dará uma nova instância de `Bird` com as propriedades `name` e `color` definidas como `Bruce` e `red`, respectivamente. A propriedade `numLegs` ainda está definida como 2. O `cardinal` possui três propriedades:

```
cardinal.name
cardinal.color
```

```
cardinal.numLegs
```

O construtor é mais flexível. Agora é possível definir as propriedades de cada `Bird` ao mesmo tempo em que são criados, o que é uma grande utilidade dos construtores JavaScript. Eles agrupam junto objetos baseados em características e comportamentos compartilhados e definem uma planta que automatiza a criação deles.

Exercicio = Crie outro construtor `Dog`. Desta vez, defina o construtor para receber os parâmetros `name` e `color`, e que tenham a propriedade `numLegs` fixada em 4. Em seguida, crie um novo `Dog` salvo na variável `terrier`. Passe duas strings como argumentos para as propriedades `name` e `color`.

```
function Dog(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 4;
}
let terrier = new Dog("George", "White");
```

Importante !!!!!!

Verificar o construtor de um objeto com instanceof

Toda vez que a função construtora cria um novo objeto, o objeto é definido como uma *instance* do seu construtor. JavaScript provê uma forma conveniente para verificar isso com o operador `instanceof`. `instanceof` permite que você compare um objeto a um construtor, retornando `true` ou `false` caso seja ou não um objeto criado pelo construtor, respectivamente. Exemplo:

```
let Bird = function(name, color) {
  this.name = name;
  this.color = color;
  this.numLegs = 2;
}
```

```
let crow = new Bird("Alexis", "black");

crow instanceof Bird;
```

Este método `instanceof` irá retornar `true`.

Se um objeto for criado sem usar um construtor, `instanceof` verificará que não é uma instância daquele construtor:

```
let canary = {
  name: "Mildred",
  color: "Yellow",
  numLegs: 2
};

canary instanceof Bird;
```

Este método `instanceof` irá retornar `false`.

Exercicio = Crie uma nova instância do construtor `House`, atribuindo à variável `myHouse` e passe o número de quartos. Então, utilize `instanceof` para verificar que é uma instância de `House`.

```
function House(numBedrooms) {
  this.numBedrooms = numBedrooms;
}


// Altere apenas o código abaixo desta linha
let myHouse = new House(4);
myHouse instanceof House;
```

Entender propriedades próprias

No próximo exemplo, o construtor de `Bird` define duas propriedades: `name` e `numLegs`:

```
function Bird(name) {
  this.name = name;
  this.numLegs = 2;
}

let duck = new Bird("Donald");
let canary = new Bird("Tweety");
```

`name` e `numLegs` são chamados *own properties*, pois são definidos diretamente na instância do objeto. Isso significa que cada `duck` e `canary` possuem suas próprias cópias separadas destas propriedades. Na verdade, toda instância de `Bird` terá sua própria cópia dessas propriedades. O código a seguir adiciona todas as propriedades próprias (own properties) de `duck` para o array `ownProps`:

```
let ownProps = [];

for (let property in duck) {
  if(duck.hasOwnProperty(property)) {
    ownProps.push(property);
  }
}

console.log(ownProps);
```

O console vai exibir o valor `["name", "numLegs"]`.

Exercicio = Adicione as propriedades próprias do `canary` para o array `ownProps`.

```
function Bird(name) {

  this.name = name;

  this.numLegs = 2;
```

```
}



let canary = new Bird("Tweety");
let ownProps = [];
// Altere apenas o código abaixo desta linha
for (let property in canary) {
  if (canary.hasOwnProperty(property)){
    ownProps.push(property);
  }
}
console.log(ownProps);
```

Usar propriedades de protótipos para reduzir código duplicado

Já que `numLegs` provavelmente terá o mesmo valor para todas as instâncias de `Bird`, você tem a variável `numLegs` duplicada dentro de cada instância de `Bird`.

Isso pode não ser um problema quando há apenas duas instâncias, mas imagine se há milhões de instâncias. Neste cenário teríamos muitas variáveis duplicadas.

Uma maneira melhor é usar o `prototype` de `Bird`. Propriedades dentro de `prototype` são compartilhados entre todas as instâncias de `Bird`. Aqui está como adicionar `numLegs` para o `prototype` de `Bird`:

```
Bird.prototype.numLegs = 2;
```

Agora todas as instâncias de `Bird` possuem a propriedade `numLegs`.

```
console.log(duck.numLegs);
console.log(canary.numLegs);
```

Já que todas as instâncias automaticamente possuem as propriedades no `prototype`, pense no `prototype` como uma "receita" para criar objetos. Note que o `prototype` para `duck` e `canary` faz parte do construtor de `Bird` como

`Bird.prototype`. Quase todos objetos em JavaScript possuem a propriedade `prototype` o qual é parte da função construtora que o criou.

Exercicio = Adicione a propriedade `numLegs` para o `prototype` de `Dog`

```
function Dog(name) {
  this.name = name;
}


Dog.prototype.numLegs = 4;


// Altere apenas o código acima desta linha
let beagle = new Dog("Snoopy");
```

Iterar sobre todas as propriedades

Até agora você já viu dois tipos de propriedades: as propriedades *own properties* e `prototype`. Propriedades próprias (ou Own properties) são definidas diretamente na própria instância do objeto. E as propriedades do protótipo são definidas em `prototype`.

```
function Bird(name) {
  this.name = name;  //own property
}

Bird.prototype.numLegs = 2; // prototype property

let duck = new Bird("Donald");
```

Aqui está como você adiciona own properties `duck` para o array `ownProps` e propriedades `prototype` para o array `prototypeProps`:

```
let ownProps = [];
let prototypeProps = [];
```

```
for (let property in duck) {
  if(duck.hasOwnProperty(property)) {
    ownProps.push(property);
  } else {
    prototypeProps.push(property);
  }
}

console.log(ownProps);
console.log(prototypeProps);
```

`console.log(ownProps)` deve exibir no console `["name"]`, e `console.log(prototypeProps)` exibirá no console `["numLegs"]`.

Adicione todas as propriedades próprias de `beagle` para o array `ownProps`. Adicione todas as propriedades `prototype` de `Dog` para o array `prototypeProps`.

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");
```

```
let ownProps = [];
let prototypeProps = [];



// Altere apenas o código abaixo desta linha
for (let property in beagle){
  if (beagle.hasOwnProperty(property)){
    ownProps.push(property);}
  else {
    prototypeProps.push(property);
  }
}
 Dog.prototype.numLegs;
```

Entender a propriedade construtora

Tem uma propriedade especial do `constructor` localizada nas instâncias dos objetos `duck` e `beagle` que foram criados nos desafios anteriores:

```
let duck = new Bird();
let beagle = new Dog();

console.log(duck.constructor === Bird);
console.log(beagle.constructor === Dog);
```

Ambas as chamadas a `console.log` vão exibir `true` no console.

Note que a propriedade `constructor` é uma referência a função construtora que criou a instância. A vantagem da propriedade `constructor` é que se torna possível verificar essa propriedade para descobrir qual o tipo do objeto. Aqui está um exemplo de como isso poderia ser utilizado:

```
function joinBirdFraternity(candidate) {
  if (candidate.constructor === Bird) {
    return true;
```

```
  } else {
    return false;
  }
}
```

Observação: já que a propriedade `constructor` pode ser sobrescrita (o que será abordado nos próximos dois desafios), geralmente é melhor utilizar o método `instanceof` para verificar o tipo de um objeto.

Escreva a função `joinDogFraternity` que recebe o parâmetro `candidate` e, utilizando a propriedade `constructor`, retorne `true` se o candidato é um `Dog`, e caso não seja, retorne `false`.

```
function Dog(name) {
  this.name = name;
}


// Altere apenas o código abaixo desta linha
function joinDogFraternity(candidate) {
if (candidate.constructor === Dog){
  return true;
} else  {
  return false;
}
}
```

Mudar o protótipo para um novo objeto

Até o momento, você tem adicionado propriedades para cada `prototype` individualmente:

```
Bird.prototype.numLegs = 2;
```

Isto se torna entediante após mais do que algumas propriedades.

```
Bird.prototype.eat = function() {
  console.log("nom nom nom");
}

Bird.prototype.describe = function() {
  console.log("My name is " + this.name);
}
```

Uma forma mais eficiente é definir o `prototype` para um novo objeto que já possui as propriedades. Dessa maneira, as propriedades são adicionadas todas de uma vez:

```
Bird.prototype = {
  numLegs: 2,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Adiciona a propriedade `numLegs` e os dois métodos `eat()` e `describe()` para o `prototype` de `Dog` definindo o `prototype` para um novo objeto.

```
function Dog(name) {
  this.name = name;
}
```

```
Dog.prototype = {
  // Altere apenas o código abaixo desta linha
 numLegs: 2,
```

```
 eat: function () {
   console.log("food");
 },
 describe: function (){
   cosole.log("my name is mary");
 }
};
```

Lembrar de definir a propriedade construtora quando alterar o protótipo

Tem um efeito colateral crucial de definir manualmente o protótipo de um novo objeto. Isso apaga a propriedade `constructor`! Essa propriedade pode ser utilizada para verificar qual função construtora criou a instância, mas já que a propriedade foi sobrescrita, agora retorna resultados falsos:

```
duck.constructor === Bird;
duck.constructor === Object;
duck instanceof Bird;
```

Em ordem, essas expressões seriam avaliadas a `false`, `true` e `true`.

Para corrigir isso, toda vez que o protótipo é manualmente definido para um novo objeto, lembre-se de definir a propriedade `constructor`:

```
Bird.prototype = {
  constructor: Bird,
  numLegs: 2,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Define a propriedade `constructor` no `prototype` de `Dog`.

```
function Dog(name) {
  this.name = name;
}


// Altere apenas o código abaixo desta linha
Dog.prototype = {
constructor : Dog,
  numLegs: 4,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Entender de onde vem o protótipo de um objeto

Assim como uma pessoa herda o gene de seus parentes, um objeto herda seu `prototype` diretamente da função construtora que o criou. Por exemplo, aqui o construtor de `Bird` cria um objeto `duck`:

```
function Bird(name) {
  this.name = name;
}

let duck = new Bird("Donald");
```

`duck` herda seu `prototype` da função construtora de `Bird`. Você pode mostrar a relação com o método `isPrototypeOf`:

```
Bird.prototype.isPrototypeOf(duck);
```

Isso retornaria `true`.

Utilize `isPrototypeOf` para verificar o `prototype` de `beagle`.

```
function Dog(name) {
  this.name = name;
}

let beagle = new Dog("Snoopy");

// Altere apenas o código abaixo desta linha
Dog.prototype.isPrototypeOf(beagle);
```

Entender a cadeia de protótipos

Todos os objetos em JavaScript (com algumas exceções) possuem um `prototype`. Além de que, um `prototype` de um objeto ser um próprio objeto.

```
function Bird(name) {
  this.name = name;
}

typeof Bird.prototype;
```

Devido ao fato de um `prototype` ser um objeto, um `prototype` pode ter seu próprio `prototype`! Neste caso, o `prototype` de `Bird.prototype` é um `Object.prototype`:

```
Object.prototype.isPrototypeOf(Bird.prototype);
```

Como isso é útil? Você pode ser lembrar que o método `hasOwnProperty` do desafio anterior:

```
let duck = new Bird("Donald");
duck.hasOwnProperty("name");
```

O método `hasOwnProperty` é definido em `Object.prototype`, o qual pode ser acessado por `Bird.prototype`, o qual pode ser acessado por `duck`. Este é um exemplo de cadeia de `prototype`. Nesta cadeia de `prototype`, `Bird` é um `supertipo` para `duck`, enquanto `duck` é o `subtipo`. `Object` é um `supertipo` para ambos `Bird` e `duck`. `Object` é um `supertipo` para todos os objetos em JavaScript. Desta forma, qualquer objeto pode utilizar o método `hasOwnProperty`.

Modifique o código para mostrar corretamente a cadeia de protótipo.

```
function Dog(name) {
  this.name = name;
}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle);  // retorna verdadeiro

// Corrija o código abaixo para que ele seja avaliado como verdadeiro
Object.prototype.isPrototypeOf(Dog.prototype);
```

Utilizar herança para não se repetir

Tem um princípio da programação chamado *Don't Repeat Yourself (DRY) - Não se Repita*. O motivo pelo qual código repetido é um problema se deve ao fato de qualquer alteração exige correção de código em vários locais. Geralmente isso significa mais trabalho para os programadores e mais espaço para erros.

Note que, no exemplo abaixo, o método `describe` é compartilhado por `Bird` e `Dog`:

```
Bird.prototype = {
  constructor: Bird,
  describe: function() {
    console.log("My name is " + this.name);
  }
};

Dog.prototype = {
  constructor: Dog,
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

O método `describe` é repetido em dois locais. O código pode ser editado para seguir o princípio DRY (Não se Repita) ao criar um `supertype` (ou parente) chamado `Animal`:

```
function Animal() { };

Animal.prototype = {
  constructor: Animal,
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Desde que `Animal` inclui o método `describe`, você pode remover ele de `Bird` e `Dog`:

```
Bird.prototype = {
  constructor: Bird
```

```
};

Dog.prototype = {
  constructor: Dog
};
```

O método `eat` é repetido em `Cat` e `Bear`. Edite o código no espírito do princípio DRY ao mover o método `eat` do `supertype` de `Animal`.

```
function Cat(name) {
  this.name = name;
}


Cat.prototype = {
  constructor: Cat,

};


function Bear(name) {
  this.name = name;
}


Bear.prototype = {
  constructor: Bear,

};
```

```
function Animal() { }


Animal.prototype = {
  constructor: Animal,
 eat: function (){
   console.log("nom");
 }
};
```

Herdar comportamentos de um supertipo

No desafio anterior, você criou um `supertype` chamado `Animal` que define os comportamentos compartilhados por todos os animais:

```
function Animal() { }
Animal.prototype.eat = function() {
  console.log("nom nom nom");
};
```

Este e o próximo desafio vai abordar como reutilizar métodos de `Animal` dentro de `Bird` e `Dog` sem ter de definir os métodos novamente. Ele utiliza uma técnica chamada herança. Este desafio cobrirá o primeiro passo: fazer uma instância do `supertype` (ou parente). Você já sabe uma forma de criar instâncias de `Animal` utilizando o operador `new`:

```
let animal = new Animal();
```

Há algumas desvantagens quando utilizamos essa sintaxe para herança, que são muito complexas para o escopo deste desafio. Em vez disso, aqui está uma abordagem alternativa sem essas desvantagens:

```
let animal = Object.create(Animal.prototype);
```

`Object.create(obj)` cria um novo objeto, e define `obj` como o novo `prototype` do objeto. Lembre-se que o `prototype` é como uma "receita" para criar um objeto. Ao definir o `prototype` de `animal` para ser um `prototype` de `Animal`, você está efetivamente dando a instância `animal` a mesma "receita" de qualquer outra instância de `Animal`.

```
animal.eat();
animal instanceof Animal;
```

O método `instanceof` aqui vai retornar `true`.

Utilize `Object.create` para fazer duas instâncias de `Animal` nomeados `duck` e `beagle`.

```
function Animal() { }


Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};


// Altere apenas o código abaixo desta linha


let duck = Object.create(Animal.prototype);
let beagle = Object.create(Animal.prototype);
```

Definir o protótipo da classe filha para que seja uma instância do pai

No desafio anterior, você viu o primeiro passo para herdar comportamento do supertipo (ou parente) `Animal`: fazendo uma instância de `Animal`.

Este desafio cobre o próximo passo: definir o `prototype` do subtipo (ou filho) - neste caso, `Bird` - para ser uma instância de `Animal`.

```
Bird.prototype = Object.create(Animal.prototype);
```

Lembre-se de que o `prototype` é como uma receita para criar um objeto. De certa forma, a receita para `Bird` agora inclui todos os "ingredientes" chave para `Animal`.

```
let duck = new Bird("Donald");
duck.eat();
```

`duck` herda todas as propriedades de `Animal`, incluindo o método `eat`.

Modifique o código para que as instâncias de `Dog` herdem de `Animal`.

```
function Animal() { }


Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};


function Dog() { }
```

```
// Altere apenas o código abaixo desta linha


Dog.prototype = Object.create(Animal.prototype);
```

Redefinir uma propriedade herdada do construtor

Quando um objeto herda seu `protótipo` de outro objeto, ele também herda a propriedade construtora do supertipo.

Exemplo:

```
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
let duck = new Bird();
duck.constructor
```

Mas `duck` e todas as instâncias de `Bird` devem mostrar que eles foram construídos por `Bird` e não `Animal`. Para fazer isso, você pode manualmente definir a propriedade construtora de `Bird` para o objeto `Bird`:

```
Bird.prototype.constructor = Bird;
duck.constructor
```

Corrija o código para que `duck.constructor` e `beagle.constructor` retornem seus respectivos construtores.

```
function Animal() { }

function Bird() { }

function Dog() { }
```

```
Bird.prototype = Object.create(Animal.prototype);
Dog.prototype = Object.create(Animal.prototype);


// Altere apenas o código abaixo desta linha


Bird.prototype.constructor = Bird;
Dog.prototype.constructor = Dog;



let duck = new Bird();
let beagle = new Dog();
```

Adicionar métodos após a herança

Uma função construtora, ou simplesmente construtor, que herda seu objeto de `prototype` de uma função construtora de supertipo, além dos métodos herdados, ainda poderá ter seus próprios métodos.

Por exemplo, `Bird` é um construtor que herda seu `prototype` de `Animal`:

```
function Animal() { }
Animal.prototype.eat = function() {
  console.log("nom nom nom");
};
function Bird() { }
Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Bird;
```

Como adicional do que é herdado da classe `Animal`, você deseja adicionar o comportamento que é único de objetos `Bird`. Aqui, `Bird` definirá a função `fly()`. As funções são adicionadas ao `Bird's prototype` (protótipo do pássaro) da mesma forma que qualquer função construtora:

```
Bird.prototype.fly = function() {
  console.log("I'm flying!");
};
```

Agora as instâncias de `Bird` terão ambos os métodos, `eat()` e `fly()`:

```
let duck = new Bird();
duck.eat();
duck.fly();
```

`duck.eat()` exibe no console a string `nom nom nom`, e `duck.fly()` mostra a string `I'm flying!`.

Adiciona todos os códigos necessários para que o objeto `Dog` herde de `Animal` e o `prototype` de construtor de `Dog` está definido para `Dog`. Então adiciona o método `bark()` para o objeto `Dog` para que um `beagle` possa `eat()` e `bark()`. O método `bark()` deveria imprimir no console a string: `Woof!`.

```
function Animal() { }

Animal.prototype.eat = function() { console.log("nom nom nom"); };


function Dog() { }


// Altere apenas o código abaixo desta linha

Dog.prototype = Object.create(Animal.prototype);
```

```
Dog.prototype.constructor = Dog;
Dog.prototype.bark = function(){
  console.log ("Woof!");
};



// Altere apenas o código acima desta linha


let beagle = new Dog();
```

Sobrescrever métodos herdados

Nas lições passadas, você aprendeu que um objeto pode herdar seus comportamentos (métodos) de outro objeto ao referenciar o `prototype` do objeto:

```
ChildObject.prototype =
Object.create(ParentObject.prototype);
```

Em seguida, o `ChildObject` recebeu seu próprio método ao encadear eles neste `prototype`:

```
ChildObject.prototype.methodName = function() {...};
```

É possível sobrescrever um método herdado. É feito da mesma maneira - ao adicionar o método a `ChildObject.prototype` utilizando o mesmo nome do método que aquele a ser sobrescrito. Aqui está um exemplo de `Bird` sobrescrevendo o método `eat()` herdado de `Animal`:

```
function Animal() { }
Animal.prototype.eat = function() {
```

```
  return "nom nom nom";
};
function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.eat = function() {
  return "peck peck peck";
};
```

Se você tem uma instância `let duck = new Bird();` e você chamar `duck.eat()`, é assim que o JavaScript procura pelo método na cadeia de `prototype` de `duck`:

1. `duck` => o método `eat()` está definido aqui? Não.
2. `Bird` => o método `eat()` está definido aqui? => Sim. Execute isso e pare de procurar.
3. `Animal` => `eat()` também é definido, mas o JavaScript parou de procurar antes de chegar a este level.
4. Objeto => JavaScript parou de procurar antes de chegar a este nível.

Sobrescreva o método `fly()` para `Penguin` para que retorne a string `Alas, this is a flightless bird.` (Infelizmente, este pássaro não voa.)

```
function Bird() { }


Bird.prototype.fly = function() { return "I am flying!"; };


function Penguin() { }
Penguin.prototype = Object.create(Bird.prototype);
Penguin.prototype.constructor = Penguin;
```

```
// Altere apenas o código abaixo desta linha


Penguin.prototype.fly = function () {
  return "Alas, this is a flightless bird.";
};



// Altere apenas o código acima desta linha


let penguin = new Penguin();
console.log(penguin.fly());
```

Usar um mixin para adicionar comportamentos comuns entre objetos não relacionados

Como você já viu, comportamento é compartilhado através de herança. Porém, existem casos em que a herança não é a melhor solução. Herança não funciona muito bem para objetos não-relacionados como `Bird` e `Airplane`. Ambos podem voar, mas um `Bird` não é um tipo de `Airplane` e vice-versa.

Para objetos não relacionados, é melhor usar *mixins*. Um mixin permite outros objetos para utilizar uma coleção de funções.

```
let flyMixin = function(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  }
```

```
};
```

O `flyMixin` recebe qualquer objeto e da a ele o método `fly`.

```
let bird = {
  name: "Donald",
  numLegs: 2
};

let plane = {
  model: "777",
  numPassengers: 524
};

flyMixin(bird);
flyMixin(plane);
```

Aqui `bird` e `plane` são passados para `flyMixin`, o que em seguida atribui a função `fly` para cada objeto. Agora `bird` e `plane` podem ambos voar:

```
bird.fly();
plane.fly();
```

O console irá mostrar a string `Flying, woosh!` duas vezes, uma para cada chamada a `.fly()`.

Note como o mixin permite que o mesmo método `fly` seja reutilizado por objetos não-relacionados `bird` e `plane`.

Crie um mixin chamado `glideMixin` que define o método chamado `glide`. Em seguida, use `glideMixin` para dar ambos `bird` e `boat` a habilidade de deslizar (glide).

```
let bird = {
  name: "Donald",
  numLegs: 2
```

```
};


let boat = {
  name: "Warrior",
  type: "race-boat"

};


// Altere apenas o código abaixo desta linha
let glideMixin = function (obj) {
  obj.glide = function() {
 console.log("Gliding!")
  };


}
glideMixin(bird);
 glideMixin(boat);
```

Usar closure para evitar que propriedades de um objeto sejam modificadas externamente

No desafio anterior, `bird` possuía uma propriedade pública `name`. É considerado publico porque ele pode ser acessado e modificado fora da definição de `bird`.

```
bird.name = "Duffy";
```

Portanto, qualquer parte do seu código pode facilmente alterar o nome do `bird` para qualquer valor. Pense sobre coisas como senhas e contas de banco sendo facilmente modificáveis em qualquer parte do seu código. Isso poderia causar inúmeros problemas.

A forma mais simples para tornar essa propriedade pública em privada, seria criando uma variável dentro da função constructor. Isso alteraria o escopo daquela variável para ser apenas o escopo da função construtora ao invés de globalmente disponível. Dessa maneira, a variável pode ser acessada e modificada apenas pelos métodos dentro da função construtora.

```
function Bird() {
  let hatchedEgg = 10;

  this.getHatchedEggCount = function() {
    return hatchedEgg;
  };
}
let ducky = new Bird();
ducky.getHatchedEggCount();
```

Aqui `getHatchedEggCount` é um método privilegiado, porque ele possui acesso à variável privada `hatchedEgg`. Isso é possível porque `hatchedEgg` é declarado no mesmo contexto que `getHatchedEggCount`. Em JavaScript, a função sempre possui acesso ao contexto na qual foi criada. Isso é chamado de `closure`.

Modifique como `weight` é declarado na função `Bird` para que seja uma variável privada. Em seguida, crie o método `getWeight` que retorna o valor de `weight` 15.

```
function Bird() {
  let weight = 15;


  this.getWeight = function() {
```

```
    return weight;
  };
}
```

Entender a expressão de função invocada imediatamente (IIFE)

Um padrão comum em JavaScript é executar a função assim que ela é declarada:

```
(function () {
  console.log("Chirp, chirp!");
})();
```

Essa é uma expressão de função anônima que executa logo após ser declarada, e exibe imediatamente no console `Chirp, chirp!`.

Note que a função não possui nome e não é armazenada em uma variável. Os dois parênteses () ao final da expressão da função faz com que ela seja imediatamente executada ou invocada. Este padrão é conhecido como *immediately invoked function expression (expressão de função invocada imediatamente)* ou *IIFE*.

Rescreva a função `makeNest` e remova a chamada a ela para que no lugar seja uma expressão de função imediatamente invocada (IIFE) anônima.

```
(function() {
  console.log("A cozy nest is ready");
})();
```

Usar uma IIFE para criar um módulo

Uma expressão de função imediatamente invocada (IIFE) é frequentemente utilizada para agrupar funcionalidades relacionadas para um único objeto ou *módulo*. Por exemplo, um desafio anterior definiu dois mixins:

```
function glideMixin(obj) {
  obj.glide = function() {
    console.log("Gliding on the water");
  };
}
function flyMixin(obj) {
  obj.fly = function() {
    console.log("Flying, wooosh!");
  };
}
```

Podemos agrupar esses mixins em um módulo como o seguinte:

```
let motionModule = (function () {
  return {
    glideMixin: function(obj) {
      obj.glide = function() {
        console.log("Gliding on the water");
      };
    },
    flyMixin: function(obj) {
      obj.fly = function() {
        console.log("Flying, wooosh!");
      };
    }
  }
})();
```

Note que você possui uma expressão de função imediatamente invocada (IIFE) que retorna um objeto `motionModule`. Esse objeto retornado contém todos os comportamentos de mixin como propriedades do objeto. A vantagem do padrão módulo é que todos os comportamentos de movimento podem ser embalados em um único objeto que pode em seguida ser usado por outras partes do seu código. Aqui está um exemplo utilizando isso:

```
motionModule.glideMixin(duck);
duck.glide();
```

Crie um módulo chamado `funModule` para embrulhar os dois mixins `isCuteMixin` e `singMixin`. `funModule` deve retornar um objeto.

```
let funModule = (function() {
  return {
    isCuteMixin: function(obj) {
      obj.isCute = function() {
        return true;
      };
    },
    singMixin: function(obj) {
      obj.sing = function() {
        console.log("Singing to an awesome tune");
      };
    }
  };
})();
```