Imagine que eu vá fazer um sistema de um e-commerce a primeira questão é o que a gente vai vender:

Camiseta, calça e tênis

O que tudo isso tem em comum?

Há como eu abstrair tudo e depois especializar pra ficar mais simples o meu código então os subtraios numa coisa só e depois eu especializo o programa pra que eu reutilize o meu código.

Os 3 itens dentro de um e-commerce representam produto então aqui está a minha abstração: produto;

Então eu vou criar uma função que cria um produto:

```
function product(name, price) {
    this.name = name;
    this.price = price;
}
```

Agora eu já tenho uma abstração dos meus produtos.

A herança na hora que eu especializar o produto vai ser novidade.

Camiseta, calça e tênis as três podem ter duas coisas específicas a camiseta pode ter cor, a calça pode ter o tipo e o tênis pode ter tamanho. Além das especifiçações de cada uma delas eu também quero nos três uma possibilidade de aumento de preço e de desconto:

```
product.prototype.aumento = function(quantia) {
    this.price += quantia;
};
product.prototype.desconto = function(quantia) {
    this.price -= quantia;
};
```

Agora eu quero especializar a função produto eu não quero ter que reescrever aumento, desconto e eu não quero ter que reescrever a função produto principalmente então eu vou criar uma função construtora para cada um dos três itens:

```
function shirt(name, price, color) {

```

```
}
```

Agora eu preciso chamar o construtor da função calça, ou seja, dentro da função shirt eu tenho que fazer o código:

```
this.name = name;
this.price = price;
```

Seja executado!

A gente pode usar um método que tem dentro de todas as funções chamado de call:

```
function Shirt(name, price, color) {
    product.call();
}
```

Eu tenho que passar dentro de () quem que vai ser o this da minha função shirt que vai ser o objeto que eu criar usando essa função construtora então eu vou passar o this pra função produto:

```
function Shirt(name, price, color) {
    product.call(this);
}
```

E o restante eu vou passar os meus argumentos que eu preciso nessa função construtora que é nome e preço só:

```
function Shirt(name, price, color) {
    product.call(this, name, price);
}
```

Fazendo só isso daqui eu já linkei as duas funções a minha função shirt vai funcionar exatamente como a função produto, mas ela vai ter uma diferença.

```
const shirt = new Shirt('Regata', 10, 'Black');
console.log(shirt);
```

Se eu tentar fazer isso:

```
shirt.aumento(15);
```

Vai dar um erro dizendo que isso não é uma função dentro de Shirt. Eu não tenho os prototypes dentro da função Shirt mesmo eu tendo chamado a função porque cada

função construtora tem a sua propriedade prototype a gente não linkou das duas funções -> product e Shirt. Eu quero que a Shirt inicialmente tenha o mesmo prototype do product e a partir disso eu preciso poder modificar o prototype apenas de Shirt e não de product, ou seja, se eu incluir ou alterar alguma coisa dentro de Shirt ele não afeta o meu product:

```
Shirt.prototype = product.prototype; // Isso não funciona
```

Faça:

```
Shirt.prototype = Object.create(product.prototype);
```

Eu vou criar um objeto e vou setar o prototype desse objeto vazio para o prototype do product, como product. O prototype da Shirt vai ser o prototype product e agora o aumento vai funcionar:

```
function product(name, price) {
    this.name = name;
    this.price = price;
}
product.prototype.aumento = function(quantia) {
    this.price += quantia;
};
product.prototype.desconto = function(quantia) {
    this.price -= quantia;
};

function Shirt(name, price, color) {
    product.call(this, name, price);
}
Shirt.prototype = Object.create(product.prototype);

const shirt = new Shirt('Regata', 10, 'Black');
shirt.aumento(15);
console.log(shirt);
```

Resultado:

```
product { name: 'Regata', price: 25 }
```

Agora funcionou mas eu arrumei outro problema agora o que eu crio com Shirt está falando que o construtor é product e não é o construtor é Shirt.

Façamos:

```
const product = new Product('Any', 110);
```

Eu não tenho o construtor dentro da função Shirt porque eu criei um objeto vazio e eu usei o prototype de Product como o prototype do meu objeto vazio Shirt.

A gente resolve da seguinte forma:

```
Shirt.prototype.constructor = Shirt;
```

A gente linka de volta o construtor da função Shirt.

Resultado:

```
Product { name: 'Any', price: 110 }
Shirt { name: 'Regata', price: 10 }
```

Agora a gente tem o construtor real da camiseta.

Eu coloquei a cor porque eu posso extender a funcionalidade das classes que eu estou especializando, Shirt é uma especialização de Product ou seja Shirt herda tudo o que Product tem mas Shirt também pode ter suas coisas específicas e é daqui que vem o poder da herança eu posso criar uma coisa simples, reutilizar esse código:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}
Product.prototype.aumento = function(quantia) {
    this.price += quantia;
};
Product.prototype.desconto = function(quantia) {
    this.price -= quantia;
};
```

E sobrescrever o que não quero nos filhos da classe Product:

```
function Shirt(name, price, color) {
    Product.call(this, name, price);
    this.color = color;
```

```
}
```

Agora a minha Shirt tem uma propriedade a mais do que o produto original e eu poderia fazer isso com quantos atributos eu quisesse.

Eu poderia sobrescrever o prototype aumento:

```
Shirt.prototype.aumento = function(percentual) {
    this.price = this.price + (this.price * (percentual / 100));
}
```

Eu mudei o comportamento dele dentro do prototype da Shirt.

Código da aula:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}
Product.prototype.aumento = function(quantia) {
    this.price += quantia;
};
Product.prototype.desconto = function(quantia) {
    this.price -= quantia;
};

function Shirt(name, price, color) {
    Product.call(this, name, price);
    this.color = color;
}
// Shirt.prototype.constructor = Shirt;
Shirt.prototype = Object.create(Product.prototype);
Shirt.prototype.constructor = Shirt;

Shirt.prototype.aumento = function(percentual) {
    this.price = this.price + (this.price * (percentual / 100));
};

function Pants(name, price, material, stock) {
    Product.call(this, name, price);
    this.material = material;
    Object.defineProperty(this, 'stock', {
        enumerable: true,
```

```
        configurable: false,
        get: function() {
            return stock;
        },
        set: function(value) {
            if (typeof value !== 'number') return;
            stock = value;
        }
    });
}
Pants.prototype = Object.create(Product.prototype);
Pants.prototype.constructor = Pants;

function Tennis(name, price, size) {
    Product.call(this, name, price);
    this.size = size;
}
Tennis.prototype = Object.create(Product.prototype);
Tennis.prototype.constructor = Tennis;

const product = new Product('Any', 110);
const shirt = new Shirt('Regata', 10, 'Black');
const pants = new Pants('Jeans', '180', 'Plastic', 15);
const tennis = new Tennis('Nike', 200, 42);
pants.stock = 30; // Aqui estou usando Setter

console.log(pants.stock); // Aqui estou usando Getter
console.log(pants)
console.log(shirt);
console.log(tennis);
console.log(product);
```