Seção 06: Javascript Objetos e prototypes (avançado)

* Aula 70 Revisando Obetos

const pessoa = {

    nome: 'Luiz',

    sobrenome: 'Otávio'

};

console.log(pessoa['nome']);

console.log(pessoa.sobrenome);

forma de chamar

const pessoa = {

    nome: 'Luiz',

    sobrenome: 'Otávio',

    idade: 30

};

// formas de chamar um atributo

console.log(pessoa['nome']);

console.log(pessoa.sobrenome);

// chave dinamica

const chave = 'idade';

console.log(pessoa[chave]);

console.log('#############################################################');

// new ArraY(); ou []

const pessoa1 = new Object();

pessoa1.nome = 'Luiz';

pessoa1.sobrenome = 'Otávio';

pessoa1.falarNome = function(){

    return `${this.nome} está falando seu nome.`;

};

pessoa1.idade = 30;

pessoa1.getDataNascimento = function() {

    const dataAtual = new Date();

    return dataAtual.getFullYear() - this.idade;

};

const pessoa2 = {

    nome: 'Luiz',

    sobrenome: 'Otávio'

};

// delete pessoa2.nome;

// console.log(pessoa1);

// console.log(pessoa2);

// pessoa1.falarNome()

// console.log(pessoa1.getDataNascimento());

for (let chave in pessoa1) {

    console.log(chave, pessoa1[chave]);

}

console.log('#############################################################');

// factory functions / Constructor functions // Classes

// Factory function

function criaPessoa(nome, sobrenome) {

    return {

        nome,

        sobrenome,

        nomeCompleto() {

            return `${this.nome} ${this.sobrenome}`

        }

    };

};

const p1 = criaPessoa('Luiz', 'Otávio');

console.log(p1.nomeCompleto());

// Constructor functions

function Pessoa(nome, sobrenome) {

    this.nome = nome,

    this.sobrenome = sobrenome;

    Object.freeze(this);  // Nenhum objeto poderar ser alterado seu valor ou criado

}

// {} <- this -> retornando esse this

const pe1 = new Pessoa('Maria', 'Joaquina');

Object.freeze(pe1);   // para travar o objeto

pe1.nome = 'Monica';  // iria aterar o objeto caso não estivesse travado

const pe2 = new Pessoa('João', 'Pedro');

console.log(pe1);

console.log(pe2);

* Aula 71 Object.defineProperty() e Object.defineProperties()

// defineProperty - defineProperties

function Produto(nome, preco, estoque) {

    this.nome = nome;

    this.preco = preco;

    Object.defineProperty(this, 'estoque', {

        enumerable: true,     // mostra a chave

        value: estoque,       // valores que vai receber

        writable: false,      // controla se o valor pode ou não ser alterado

        configurable: false   // se pode reconfigurar a propriedade (ex: delete p1.estoque)

    });

}

const p1 = new Produto('Camiseta', 20, 3);

console.log(p1);

console.log(Object.keys(p1)); [ 'nome', 'preco', 'estoque' ]

para mostrar as chaves do seu objeto

Se a opção de enumerable for false, essa chave não aparecerá.

 Object.defineProperty(this, 'estoque', {

        enumerable: false,     // mostra a chave

    });

console.log(Object.keys(p1)); [ 'nome', 'preco' ]

essa foi a defineProperty, que serve para pegar so um valor.

Agora vamos ver a defineProperties para pegar mais de uma propriedade

function Produto2(nome, preco, estoque) {

    Object.defineProperties(this, {

        nome: {

            enumerable: true,

            value: nome,

            writable: true,

            configurable: true

        },

        preco: {

            enumerable: false,

            value: preco,

            writable: true,

            configurable: true

        },

        estoque: {

            enumerable: true,

            value: estoque,

            writable: true,

            configurable: true

        }

    });

}

const p2 = new Produto2('Camiseta', 20, 3);

console.log(p2);

você vai interando com cada chave do seu objeto particularmente, um a um

* Aula 72 Getters e Setters

// defineProperty -> Getters e Setters

// Get - Obter valor

// Set - setar esse valor

function Produto(nome, preco, estoque) {

    this.nome = nome;

    this.preco = preco;

    let estoquePrivado = estoque

    Object.defineProperty(this, 'estoque', {

        enumerable: true,

        configurable: false,

        get: function() {    // Pegar e mostrar o valor

            return estoquePrivado;

        },

        set: function(valor) {

            if(typeof valor !== 'number') {

                console.log('Bad value');

                return

            }

            estoquePrivado = valor;

        }

    });

}

const p1 = new Produto('Camisa', 20, 3);

//console.log(p1);

p1.estoque = 500;

console.log(p1.estoque);

agora com function factory

// Function Factory get e set

function criaProduto(nome) {

    return {

        get nome(){

            return nome;

        },

        set nome(valor){

            nome = valor;

        }

    };

}

const p2 = criaProduto('camiseta');

console.log(p2.nome);

* Aula 73 Metodos uteis para objetos

/\*

Object.values               // Mostra somente seus valores

Object.entries              // Mostra as chaves e os valores

Object.getOwnPropertyDescriptor(o, 'prop') // Mostras suas propriedades

Object.assign(des, any)     // Copiar um objeto

...(spread)                 // Espalha os valores do objeto

// Já vimos

Object.keys (retorna as chaves)

Object.freeze (congela o objeto)

Object.defineProperties (define várias propriedade)

Object.defineProperty (define uma propriedade)

\*/

const produto = {nome: 'Produto', preco: 1.8};

const caneca = {...produto, material: 'porcelana'};

console.log(produto);

console.log(caneca);

modelo que copia uma array.

const produto = {nome: 'Produto', preco: 1.8};

//const caneca = {...produto};

const caneca = Object.assign({}, produto, {material: 'porcelana'});

caneca.nome = 'Outro nome';

caneca.preco = 2.5;

outra forma de copiar os valores de um objeto e criando novas chaves

console.log(Object.getOwnPropertyDescriptor(produto, 'nome'));

{

  value: 'Produto',

  writable: true,

  enumerable: true,

  configurable: true

}

Mostra as propriedades da chaves.

console.log(Object.entries(produto));

[ [ 'nome', 'Produto' ], [ 'preco', 1.8 ] ]

mostra as chaves e os valores

// Desestruturando um objeto

for(let valor of Object.entries(produto)){

    console.log(valor[0], valor[1]);

}

nome Produto

preco 1.8

Para desestruturar um objeto

* Aula 74 Prototypes

// Construtora -> molde (classe)

function Pessoa(nome, sobrenome) {

    this.nome = nome;

    this.sobrenome = sobrenome;

    this.nomeCompleto = () => `${this.nome} ${this.sobrenome}`;

}

// instância

const pessoa1 = new Pessoa('Luiz', 'O.') // <- Pessoa = Função construtora

const pessoa2 = new Pessoa('Maria', 'A.') // <- Pessoa = Função construtora

console.log(pessoa1);

console.log(pessoa2);

Pessoa {

  nome: 'Luiz',

  sobrenome: 'O.',

  nomeCompleto: [Function (anonymous)]

}

Pessoa {

  nome: 'Maria',

  sobrenome: 'A.',

  nomeCompleto: [Function (anonymous)]

}

Para cada pessoa que for criada, ele ira criar nome, sobrenome e nomeCompleto.

Vimos que nomeCompleto se repete em toda vez que for chamado, para resolver isso, utilizamos o prototype.

Pessoa.prototype.estouAqui = 'Estou dentro do prototype';

E criado dentro do \_\_proto\_\_, o método estouAqui. Agora todos os objetos que vierem de Pessoa, teram esse método de estouAqui.

Pessoa.prototype === pessoa1.\_\_proto\_\_

function Pessoa(nome, sobrenome) {

    this.nome = nome;

    this.sobrenome = sobrenome;

    this.nomeCompleto = () => 'ORIGINAL ' +this.nome +' ' +this.sobrenome;

}

Pessoa.prototype.nomeCompleto = function() {

    return this.nome + ' ' + this.sobrenome;

};

const pessoa1 = new Pessoa('Luiz', 'O.')

console.log(pessoa1.nomeCompleto());

Aqui estou dizendo que Pessoa terá um método em seu prototype chamado de nomeCompleto, caso o objeto que foi criado tenha em seus atributos o mesmo nome que o de seu pai o proto, o que valerá será o do filho, do objeto que foi criado.

Então primeiro ele ira procurar no objeto que foi criado, caso não encontre ira procurar no pai o \_\_photo\_\_.

function Pessoa(nome, sobrenome) {

    this.nome = nome;

    this.sobrenome = sobrenome;

}

Pessoa.prototype.nomeCompleto = function() {

    return this.nome + ' ' + this.sobrenome;

};

const pessoa1 = new Pessoa('Luiz', 'O.')

const pessoa2 = new Pessoa('Maria', 'A.')

console.log(pessoa1);

console.log(pessoa2);

console.log(pessoa1.nomeCompleto());

console.log(pessoa2.nomeCompleto());

Pessoa { nome: 'Luiz', sobrenome: 'O.' }

Pessoa { nome: 'Maria', sobrenome: 'A.' }

Luiz O.

Maria A.

Dessa maneira foi resolvida a questão de repetição, deixando os valores que são variável como atributos, e os valores que se repetem foram jogado em um so lugar no prototype principal.

* Aula 75 Manipulando prototypes

// new Object -> Object.prototype

const objA = {

    chaveA: 'A'

    // \_\_proto\_\_: Object.prototype

};

const objB = {

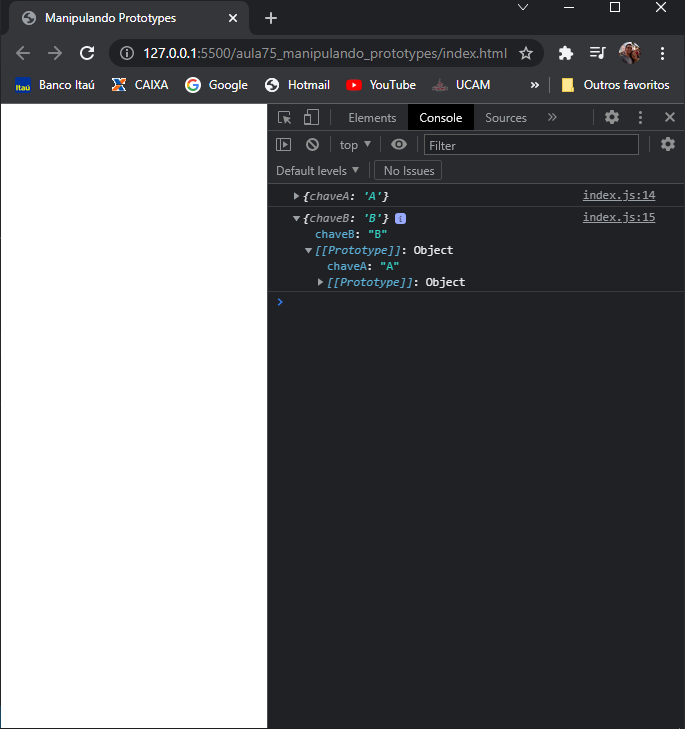
    chaveA: 'B'

    // \_\_proto\_\_: objA

};

Object.setPrototypeOf(objB, objA);

Código para que o prototype do objB seja lincando ao do objA, feito isso, objB terá seus atributos normais e dentro de seu \_\_proto\_\_, terá os atributos de objA



Com isso você vai relacionando seus prototypes com diversos outros protos

function Produto(nome, preco) {

    this.nome = nome;

    this.preco = preco;

}

Produto.prototype.desconto = function(percentual) {

    this.preco = this.preco - (this.preco \* (percentual / 100));

};

Produto.prototype.aumento = function(percentual) {

    this.preco = this.preco + (this.preco \* (percentual / 100));

};

const p1 = new Produto('camiseta', 50);

// p1.desconto(100);

p1.aumento(100)

console.log(p1); Produto { nome: 'camiseta', preco: 100 }

modelo de código, que recebe o método que estão em Produtos.

const p1 = new Produto('camiseta', 50);

// Literal

const p2 = {

    nome: 'Caneca',

    preco: 15

};

Object.setPrototypeOf(p2, Produto.prototype);

Setando o p2, que e um objeto literal a ter um o prototype de Produtos

const p3 = Object.create(Produto.prototype);

aqui estamos criando um novo objeto e setando o seu prototype para ser o de Produto

const p3 = Object.create(Produto.prototype, {

    preco: {

        writable:true,

        configurable: true,

        enumerable: true,

        value: 99

    },

    tamanho: {

        writable:true,

        configurable: true,

        enumerable: true,

        value: 42

    }

});

Aqui estamos manipulando o prototype direto quando ele foi criado.

* Aula 76 Herança

// Produto -> aumento, desconto

// Camiseta = Cor, caneca = material

function Produto(nome, preco) {

    this.nome = nome;

    this.preco = preco;

}

Produto.prototype.aumento = function(quantia) {

    this.preco += quantia

};

Produto.prototype.desconto = function(quantia) {

    this.preco -= quantia

};

Criando o objeto produto, e seus métodos direto no prototype.

function Camiseta(nome, preco, cor) {

    Produto.call(this, nome, preco);

}

Aqui foi criado uma função de camiseta que ira chamar os atributos de Produtos.

const camiseta = new Camiseta('Regata', 7.5, 'Preta');

console.log(camiseta);

foi criado uma camiseta com os argumentos.

function Camiseta(nome, preco, cor) {

    Produto.call(this, nome, preco);

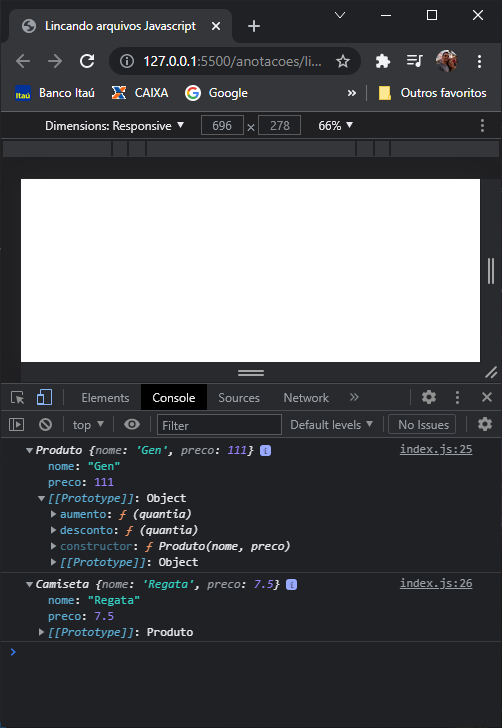
}

Camiseta.prototype = Object.create(Produto.prototype);

Aqui esta complementando o uso do prototype, esta criando um objeto prototype dentro de Camiseta para fazer a ligação.

Sendo assim o prototype de Camieta será lincado ao prototype de Produto.

Preste atenção, que usando esse método sera necessário linkar o método constructos para esse prototype, dizendo qual será seu método construtor

veja o molde de Camisa não tem um constructor, para resover isso terá que ser chamado

function Camiseta(nome, preco, cor) {

    Produto.call(this, nome, preco);

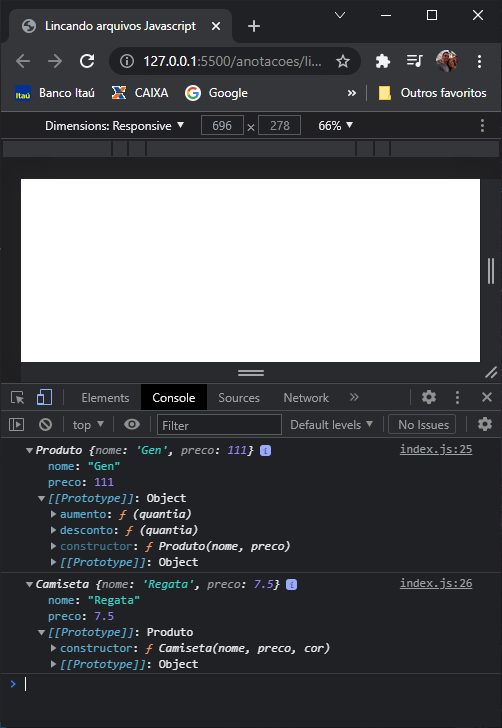
}

Camiseta.prototype = Object.create(Produto.prototype);

Camiseta.prototype.constructor = Camiseta;

Ficando assim.

Aqui estamos criando o molde de Camisa, que irá receber o prototype de Produto e o seu constructor sera o de Camisa.



function Camiseta(nome, preco, cor) {

    Produto.call(this, nome, preco);

    this.cor = cor;

}

Aqui foi adicionado o atributo cor em Camiseta, agora além dela herdar tudo de Produto, somente ela terá esse atributo de cor.

Camiseta.prototype.aumento = function(percentual) {

    this.preco = this.preco + (percentual \* (percentual / 100));

};

Aqui estamos alterando a função de aumento, direto no protopyte, somente o protopype de Camiseta sofrerá alteração.

function Caneca(nome, preco, material, estoque) {

    Produto.call(this, nome, preco);

    this.material = material;

    Object.defineProperty(this, 'estoque', {

        enumerable: true,

        configurable: false,

        get: function() {

            return estoque;

        },

        set: function(valor) {

            if(typeof valor !== 'number') return;

            estoque = valor;

        }

    })

}

Caneca.prototype = Object.create(Produto.prototype);

Caneca.prototype.constructor = Caneca;

Agora criamos um novo objeto chamado de Caneca, que herda as propriedade nome e preco de Produto, tem em si próprio o atributo material e tem um atributo chamado estoque que recebe as definições de propriedade de GET e SET, além de enumerable e configureble.

No set ele recebe uma função que so e acessada se o valor for um valor numérico.

* Aula 77 exercicio validando cpf

function ValidaCPF(cpfEnviado) {

  Object.defineProperty(this, 'cpfLimpo', {

    enumerable: true,

    get: function() {

      return cpfEnviado.replace(/\D+/g, '');

    }

  });

}

ValidaCPF.prototype.valida = function() {

  if(typeof this.cpfLimpo === 'undefined') return false;

  if(this.cpfLimpo.length !== 11) return false;

  if(this.isSequencia()) return false;

  const cpfParcial = this.cpfLimpo.slice(0, -2);

  const digito1 = this.criaDigito(cpfParcial);

  const digito2 = this.criaDigito(cpfParcial + digito1);

  const novoCpf = cpfParcial + digito1 + digito2;

  return novoCpf === this.cpfLimpo;

};

ValidaCPF.prototype.criaDigito = function(cpfParcial) {

  const cpfArray = Array.from(cpfParcial);

  let regressivo = cpfArray.length + 1;

  const total = cpfArray.reduce((ac, val) => {

    ac += (regressivo \* Number(val));

    regressivo--;

    return ac;

  }, 0);

  const digito = 11 - (total % 11);

  return digito > 9 ? '0' : String(digito);

};

ValidaCPF.prototype.isSequencia = function() {

  const sequencia = this.cpfLimpo[0].repeat(this.cpfLimpo.length);

  return sequencia === this.cpfLimpo;

};

const cpf = new ValidaCPF('070.987.720-03');

console.log(cpf);

if(cpf.valida()) {

  console.log('Cpf válido');

} else {

  console.log('Cpf inválido');

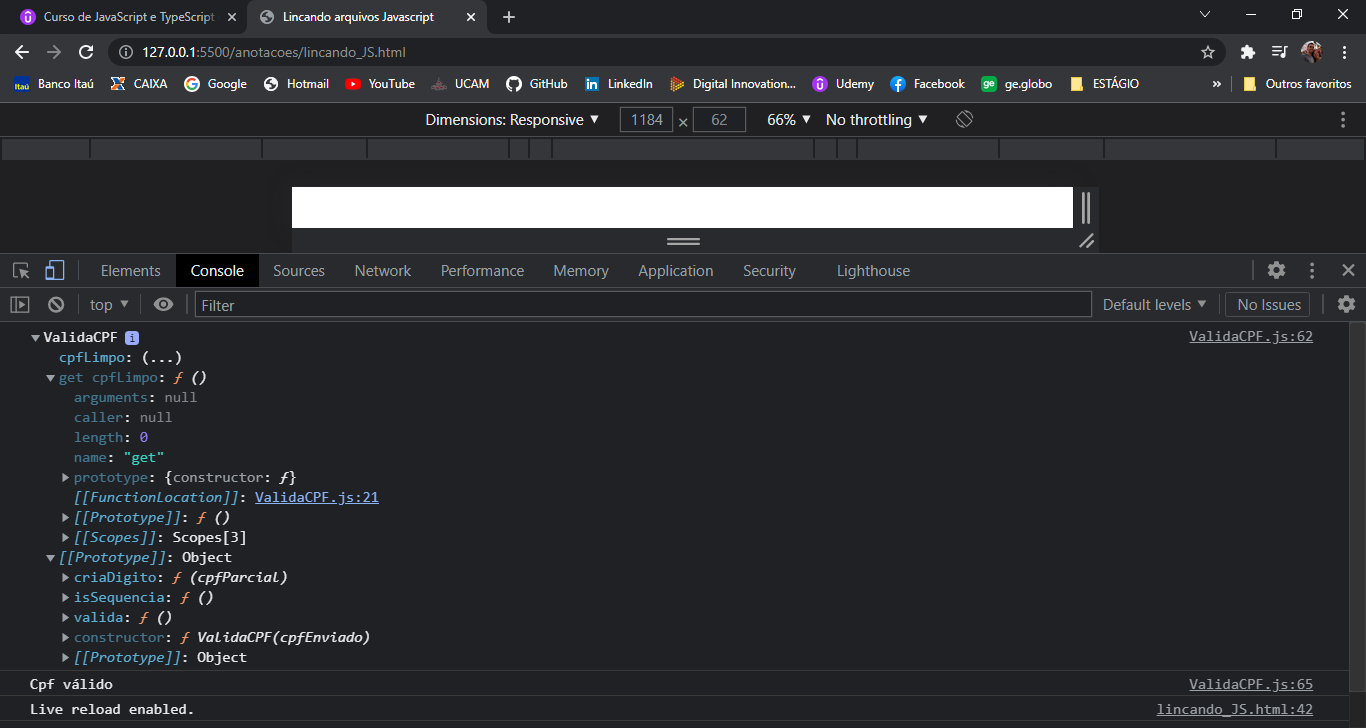
}

Aqui ele cria um molde de ValidaCpf que recebe um valor.

Esse molde e instanciado em cpf, esse cpf recebe um atributo cpfLimpo.

E no molde principa temos os prototypes, valida, criaDigito e isSequencia.

Dentro do da função de valida, que ele executa a funão criaDigito e isSequencia



* Aula 78 Polimorfismo

// Superclass

function Conta(agencia, conta, saldo) {

    this.agencia = agencia;

    this.conta = conta;

    this.saldo = saldo;

}

Conta.prototype.sacar = function(valor){

    if(valor > this.saldo) {

        console.log(`Saldo insuficiente: Seu saldo é R$ ${this.saldo}`);

        return;

    }

    this.saldo -= valor;

    this.verSaldo();

};

Conta.prototype.depositar = function(valor) {

    this.saldo += valor;

    this.verSaldo();

};

Conta.prototype.verSaldo = function() {

    console.log(

        `Ag/c:. ${this.agencia}/${this.conta} |` +

        ` Saldo: R$ ${this.saldo.toFixed(2)}`);

};

function CC(agencia, conta, saldo, limite) {

    Conta.call(this, agencia, conta, saldo);

    this.limite = limite;

}

CC.prototype = Object.create(Conta.prototype);

CC.prototype.constructor = CC;

CC.prototype.sacar = function(valor){

    if(valor > (this.saldo + this.limite)) {

        console.log(`Saldo insuficiente: Seu saldo é R$ ${this.saldo}`);

        return;

    }

    this.saldo -= valor;

    this.verSaldo();

};

function CP(agencia, conta, saldo) {

    Conta.call(this, agencia, conta, saldo);

}

CP.prototype = Object.create(Conta.prototype);

CP.prototype.constructor = CP;

const cc = new CC(11, 22, 0, 100);

cc.depositar(10);

cc.sacar(110);

cc.sacar(1)

console.log('-----------------------------------------');

const cp = new CP(12, 33, 0);

cp.depositar(10);

cp.sacar(10);

cp.sacar(1)

aqui foi criado uma superclass um função construtora Conta, que recebe 3 atributos, agencia, conta e saldo.

Em sequida e inserida em seu prototype, os métodos sacar, depositar e verSaldo.

Em sequida criamos outro método construtor, o CC (conta corrente).

Nele recebemos os atributos de seu pai Conta, e criamos um atributo somente para ele chamado de limite.

Criamos o prototype Conta dentro de CC, e definimos o seu constructor sendo o seu CC.

Logo depois nos modificamos o método sacar de seu pai para,

Valor - (this.saldo – this.limite).

Isso que e o polimorfismo, alteramos seu método, somente o construtor CC terá esse novo método de sacar.

E a segunda função construtora e a CP (conta poupança), que recebe os atributos de seu pai Conta, depois definimos o prototype de Conta dentro de CP e criamos o seu método construtor, nesse caso não alteramos o método sacar, ficando assim igual ao do seu pai Conta.

* Aula 79 Factory functions + prototypes

function criaPessoa(nome, sobrenome) {

    const pessoaPrototypes = {

        falar() {

            console.log(`${this.nome} está falando`);

        },

        comer() {

            console.log(`${this.nome} está comendo`);

        },

        beber() {

            console.log(`${this.nome} está bebendo`);

        },

    }

    return Object.create(pessoaPrototypes, {

        nome: {value: nome},

        sobrenome: {value: sobrenome}

    });

};

const p1 = criaPessoa('Luiz', 'Otávio');

const p2 = criaPessoa('Maria', 'Joaquina');

console.log(p1);

console.log(p2);

foi criado agora funções construtoras direto como funções.

Foi criado o modelo criaPessoa, que seu prototype está dentro de uma constante que está sendo acoplada, que mais a frente chamada com o uso do object.create.

Nesse object create e passado os atributos que irão receber os argumentos,

No caso o nome:{ value: nome}.

Agora vamos ver desacoplando os métodos, fazendo compossição

const falar = {

    falar() {

        console.log(`${this.nome} está falando`);

    },

};

const comer = {

    comer() {

        console.log(`${this.nome} está comendo`);

    },

};

const beber = {

    beber() {

        console.log(`${this.nome} está bebendo`);

    },

};

const pessoaPrototypes = Object.assign({}, falar, comer, beber);

function criaPessoa(nome, sobrenome) {

    return Object.create(pessoaPrototypes, {

        nome: {value: nome},

        sobrenome: {value: sobrenome}

    });

};

const p1 = criaPessoa('Luiz', 'Otávio');

const p2 = criaPessoa('Maria', 'Joaquina');

console.log(p1);

console.log(p2);

* Aula 80 objeto Map() estrutura de dados

const pessoas = [

    {id: 3, nome: 'Luiz'},

    {id: 2, nome: 'Maria'},

    {id: 1, nome: 'Helena'}

];

// const novasPessoas = {};

// for(const pessoa of pessoas) {

//     const { id } = pessoa;

//     novasPessoas[id] = {...pessoa};

// }

const novasPessoas = new Map();

for (const pessoa of pessoas) {

    const {id} = pessoa;

    novasPessoas.set(id, {...pessoa});

}

for (const pessoas of novasPessoas) {

    console.log(pessoas);

}

console.log(novasPessoas);

console.log(novasPessoas.get(2));