# Java Script – Anotações

## ATALHOS

Ctrl + k + c -> transforma o conteúdo selecionado em comentário

## NODE NO PROMPT DE COMANDO

node –v: verifica a versão dentro do Prompt de Comando

node: entra dentro do node para execução no Prompt de Comando

node .\nomedoprojeto

## CONSOLE LOG E OPERAÇÕES ARITMÉTICAS

console.log(“Meu Primeiro programa”); -> “printf” \*O ; na linguagem JS é facultativo

const idade = 19;

const idadeSomada = idadeSomada + 2;

console.log(idade);

comentários -> //comentário

\*JS é uma linguagem *case sensitive*, ou seja, diferencia maiúsculas de minúsculas

Idade = 29; e idade = 19; são variáveis diferentes e nesse caso estão sem tipagem de declaração, o que não é interessante porque dificulta em achar e diferenciar os estados ao debugar um código.

console.log(parseInt("2") + parseInt("2")); -> converte a variável string em int

O operador + para texto no console.log concatena os valores (junta); os demais operadores a própria linguagem subentende que se trata de uma operação aritmética e realiza a operação: console.log(“10” / “2”);

const nome = "Anna";

const sobrenome = "Luiza";

console.log(nome + " " + sobrenome);

console.log(nome, sobrenome);

console.log(`Meu nome é ${nome}`);

## VARIÁVEIS

const -> constante e não varia ao longo do programa, ou seja, não pode receber atribuições posteriores

let -> variável que pode ser alterada ao longo do programa

\_variavel / #variavel -> representa as variáveis privadas do código, ou seja, as que não podem ser acessadas fora de uma classe

## LISTAS

const listaDestinos = new Array(

    `Salvador`,

    `Belo Horizonte`,

    `Montes Claros`

);

listaDestinos.push(`Curitiba`); //adiciona um item na lista após ela ser delcarada

listaDestinos.splice(1,1); //retira um item na lista após ser declarada, nesse caso o 2 é a posição de início e

//o 1 é a quantidade a ser deletada

console.log(listaDestinos);

console.log(listaDestinos[0]); //mostra o item na posição 0

## CLASSE

É o molde do programa, ou seja, sempre que precisar de um certo conjunto de informações a classe é utilizada. Exemplo: classe Cliente que contém os atributos padrões que todos as variáveis cliente1, cliente2.... precisam ter.

class Cliente{

    nome;

    cpf;

    agencia;

    saldo;

}

const cliente1 = new Cliente();

cliente1.nome = "Anna";

cliente1.cpf = 12345678923;

cliente1.agencia = 3044;

cliente1.saldo = 0;

console.log(cliente1);

A classe possui então o conjunto de atributos.

Cliente1 é um objeto e a classe o molde.

Quando um arquivo for criado para representar uma classe, o arquivo deve ser nomeado começando com letra maiúscula.

## FUNÇÕES

São os métodos da programação orientada a objetos, assim nelas são adicionados comportamentos para definir/manipular as variáveis do código.

Um ***return*** dentro de uma função sem nada atribuído é utilizado para “finalizar” o uso do método quando ele for aplicado, ex.: dentro de um ***if*** que representa o que eu não quero que a função execute caso seja verdadeiro.

## MÓDULOS NO JAVASCRIPT

Terminal -> npm init -> nome do pacote (pasta do projeto)

Isso cria o package.json dentro da pasta do projeto

## SET E GET (assessores)

Atribuir algo a uma variável dentro da classe

set \_cliente(novoValor){

        if(novoValor instanceof Cliente){

            this.\_cliente = novoValor;

        }

    }

Instanceof -> se o novo valor for uma instância para a classe cliente

get cliente(){

        return this.\_cliente;

    }

“pega” o valor da instância cliente e retorna para ficar acessível no console.log diretamente

Encapsulamento: proteger atributos de novas classes

## CONSTRUCTOR

Os construtores são utilizados para inicializar atributos, organizando-os e deixando eles privados ali dentro. São usados também para simplificar o chamado do conteúdo das variáveis dentro do código.

constructor(agencia, cliente){

        this.agencia = agencia;

        this.\_cliente = cliente;

    }

const contaCorrenteAnna = new ContaCorrente(2098, cliente1); *//usando o constructor*

## EXTENDS E SUPER (HERANÇA)

A herança é utilizada para criar uma classe mãe/pai que contenha as principais informações que serão utilizadas em outras classes, sem necessidade de duplicar o código em todas elas. Assim, tratando de construtores, é preciso referenciar ao construtor pai para herdar as variáveis ali declaradas usando a palavra especial super(parâmetros do construtor);

O super também pode chamar funções e métodos.

import {Conta} from "./Conta.js";

export class ContaPoupanca extends Conta{

    constructor(saldoInicial, cliente, agencia){

        super(saldoInicial, cliente, agencia); *//super chama o construtor pai que está dentro de Conta*

    }

}

## SOBRESCRITA

Caso uma classe esteja herdando outra (classe mãe) é possível reescrever um dado método, por exemplo, na classe filha que seja igual ao método da classe mãe só que com especificações distintas. Assim, o que será executado realmente na classe filha será essa sobrescrita.

sacar(valor){

        let taxa = 1;

        const valorSacado = taxa \* valor;

        if(this.\_saldo >= valor){

            this.\_saldo -= valor;

            return valor;

        }

    } -> CLASSE MÃE

sacar(valor){

        let taxa = 1.1;

        const valorSacado = taxa \* valor;

        if(this.\_saldo >= valorSacado){

            this.\_saldo -= valorSacado;

            return valorSacado;

        }

    } -> CLASSE FILHA

## ADICIONAR ERROR AO CÓDIGO

Cria um erro que bloquei determinadas ações no código e impede que ele continue sendo executado.

*//Classe Abstrata*

export class Conta{

    constructor(saldoInicial, cliente, agencia){

        if(this.constructor == Conta){

            throw new Error ("Você não deveria instanciar uma classe do tipo Conta");

        }

        this.\_saldo = saldoInicial;

        this.\_cliente = cliente;

        this.\_agencia = agencia;

    }

## CLASSE ABSTRATA

É um tipo de classe que só pode ser herdada, ou seja, não pode ser instanciada no código. Ex. Classe Conta.

## MÉTODO ABSTRATO

É um método que é feito para não ser chamado diretamente e sim sobrescrito.

sacar(valor){ *//método abstrato*

    } -> CLASSE MÃE

import { Conta } from "./Conta.js";

export class ContaSalario extends Conta{

    constructor(cliente){

        super(0, cliente, 1020);

    }

}; -> CLASSE FILHA NÃO SOBRESCREVENDO

sacar(valor){ *//método abstrato*

        throw new Error ("O método Sacar é abstrato");

    } -> CLASSE MÃE ALERTANDO DO ERRO

import { Conta } from "./Conta.js";

export class ContaSalario extends Conta{

    constructor(cliente){

        super(0, cliente, 1020);

    }

    sacar(valor){

        let taxa = 1.01;

        return super.\_sacar(valor, taxa);

    }

}; -> CLASSE FILHA SOBRESCREVENDO

import {ContaCorrente} from "./Conta/ContaCorrente.js";

import {ContaPoupanca } from "./Conta/ContaPoupanca.js";

import { ContaSalario } from "./Conta/ContaSalario.js";

*//import {Conta} from "./Conta.js";*

const cliente1 = new Cliente("Anna", 12345678923);

const contaCorrenteAnna = new ContaCorrente(cliente1, 2098);

const contaPoupancaAnna = new ContaPoupanca(50, cliente1, 2098);

const contaSalarioAnna = new ContaSalario(cliente1);

*//const conta = new Conta(0, cliente1, 3456);*

contaCorrenteAnna.depositar(300);

contaCorrenteAnna.sacar(100);

contaPoupancaAnna.sacar(10);

contaSalarioAnna.depositar(300);

contaSalarioAnna.sacar(150);

*//console.log(conta);*

console.log(contaSalarioAnna);

console.log(contaCorrenteAnna);

console.log(contaPoupancaAnna);

## POLIMORFISMO

Diferentes classes/objetos sendo tratados da mesma forma utilizando de uma classe/objeto generalizado. O objeto autenticável representa funcionários e cliente.

“Tratar objetos diferentes de maneiras semelhantes”

export class SistemaAutenticacao{

    static login(autenticavel, senha){

        return autenticavel.senha == senha;

    }

}

## AUTENTICAÇÕES

static eAutenticavel(autenticavel){

        return "autenticar" in autenticavel && autenticavel.autenticar instanceof Function;

*//Verifica se a função autenticar existe dentro de autenticável (cliente...)*

*//e verifica se o método autenticar é uma função*

    }

*Instanceof* pode ser utilizado para verificar se um objeto é uma instância tanto de uma função quanto de uma classe.

## DUCK TYPE

“Tipo de pato” é quando o código faz uma verificação para ver se um dado objeto possui o comportamento desejado, não importa se o objeto é necessariamente uma função, ou método... importa se ele tem um comportamento igual.

Utilizado em linguagens fracamente tipadas.