**ARROW FUNCTION**

// Utilizando arrow function (na arrow function pode omitir o return, pois é implicito e por ser uma linha, podemos remover as chaves)

const maxNumero2 = (*x*, *y*) => *x* > *y* ? `${*x*} > ${*y*}` : `${*y*} > ${*x*}`;

Cria uma variável com que recebe os valores da função e com o símbolo => aponta para a função que ele trata

**TRATANDO E LANÇANDO ERROS**

Utilizamos o try catch para realizar o tratamento de erros em nosso código, podendo também levantar erros com o throw.

function soma (*x*, *y*) {

    if (typeof *x* !== 'number' || typeof *y* !== 'number') {

        throw **new** *TypeError*('x e y precisam ser números');

    }

    return *x* + *y*;

}

try {

    console.log(soma(1, 2));

    console.log(soma('1', 2));

} catch(error) {

    // console.log(error)

    console.log('Alguma coisa mais amigavel para o usuário')

}

Junto com o try e catch pode ser usado tbm o finally, que sempre será executada, independente de ter erro ou não:

try {

    // É executado quando não há erros

    console.log('Abri um arquivo')

    console.log('Manipulei um arquivo e gerou erro')

} catch (e) {

    // É executado quando há erros

    console.log('Tratando o erro')

} finally {

    // Sempre é executado

    console.log('FINALLY: Fechei o arquivo')

}

**REMOVENDO ESPAÇOS A MAIS:**

trim() -> remove os espaços em branco antes e depois da string

**FUNÇÕES**

Em JavaScript, existem algumas preculiaridades envolvendo os argumentos das funções.

Primeiro, o JS aceita que uma função que foi declarada sem receber nenhum parametro receba valores enviados em sua chamada, sem causar nenhum erro.

Todas os valores enviados são adicionados a um objeto de nome ‘arguments’, que pode ser utilizado dentro da função. Exemplo:

// IMPORTANTE: Em funções decladas com a palavra functions, os valores enviados na chamada da função são todos colocados no objeto 'arguments'

function funcao() {

    let total = 0;

    for (let argumento of *arguments*) {

        total += argumento;

    }

    console.log(total);

}

funcao(1, 2, 3, 4, 5, 6);

*\*IMPORTANTE: isso só acontece em funções que são declaradas com o function antes delas. Em arrow functions por exemplo não temos a mesma propriedade.*

Outra coisa é que quando é enviado menos valores do que parametros declarados na função, o JS também não retorna um erro, mas sim aplica o valor ‘undefined’ para os parametros que não receberam valores. Exemplo:

function funcao(*a*, *b*, *c*, *d*, *e*, *f*) {

    console.log(*a*, *b*, *c*, *d*, *e*, *f*);

}

funcao(1, 2, 3);

>>> RETORNO:

1 2 3 undefined undefined undefined

**Adicionando todos os valores restantes a uma variavel com rest:**

function conta(*operador*, *acumulador*, ...*numeros*) {

    for (let numero of *numeros*) {

        if (*operador* === '+') *acumulador* += numero;

        if (*operador* === '-') *acumulador* -= numero;

        if (*operador* === '/') *acumulador* /= numero;

        if (*operador* === '\*') *acumulador* \*= numero;

    }

    console.log(*acumulador*)

}

conta('/', 1, 20, 30, 40, 50);

**\*\*IMPORTANTE:**

shift()-> remove o primeiro ELEMENTO de uma array e retorna esse ELEMENTO

**OBJETOS**

**Função construtora:**

A função construtora é declarada com a primeira letra maiuscula (convenção) para que seja lembrado de utilizar o new ao criar um novo objeto com ela.

Essa função se assemelha muito ao conceito de classe, retornando um objeto quando chamada.

function Conta(*agencia*, *conta*, *saldo*) {

*this*.agencia = *agencia*;

*this*.conta = *conta*;

*this*.saldo = *saldo*;

}

Para declaração de métodos (funções internas) utilizamos a seguinte sintaxe:

Conta.prototype.sacar = function (*valor*) {

    if (*this*.saldo < *valor*) {

        console.log(`Saldo insufuciente. Saldo atual: R$${*this*.saldo.toFixed(2)}`)

        return;

    }

*this*.saldo -= *valor*;

*this*.verSaldo();

};

Veja que usamos o proto do objeto para adicionar o método dela.

**Herança com funções construtoras:**

Utilizando a mesma função construrora ‘Conta’ acima, podemos criar novas funções que herdam da “superclass” conta. Veja um exemplo:

function ContaCorrente(*agencia*, *conta*, *saldo*, *limite*) {

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

*this*.limite = *limite*;

}

ContaCorrente.prototype = *Object*.create(Conta.prototype);

ContaCorrente.prototype.constructor = ContaCorrente;

Para criar esse vínculo de herança, chamamos a função call do objeto que queremos herdar dentro da função e passamos os parametros necessários. Qualquer novo parametro é passado abaixo, normalmente.

Após isso, precisamos vincular o proto da nova função ‘ContaCorrente’ com o proto da função que será herdade ‘Conta’, que realizamos da seguinte maneira:

ContaCorrente.prototype = *Object*.create(Conta.prototype);

E por último corrigimos o constructor do proto da nova função para o nome da propriafunção que está sendo criada:

ContaCorrente.prototype.constructor = ContaCorrente;

**OBJETO MAP:**

const pessoas = [

    { id: 3, nome: 'Luiz' },

    { id: 2, nome: 'Maria' },

    { id: 1, nome: 'Helena' },

];

//

// const novasPessoas = {};

// for (const { id, nome } of pessoas) {

//     novasPessoas[id] = { id, nome };

// }

const novasPessoas = **new** *Map*();

for (const { id, nome } of pessoas) {

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

}

No exemplo acima, utilizando o metódo de inserção normal (comentado), alguns problemas acontecem:

1º -> A chave de cada objeto será convertidas para string;

2° -> A ordem dos objetos será redefinida (ordem crescente) pela sua chave

Para evitar esses problemas utilizamos o objeto Map (não comentado), que mantem a ordem original de inserção e também não altera o tipo de dado da chave.

Para buscar uma chave especifica do Map, utilizamos o ‘get’:

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

Ao iterar por um Map, ele retorna arrays com os valores das chaves e seus valores. Também podemos iterar somente pelas chaves e valores do Map, utilizando a função keys() e values():

for (const pessoa of novasPessoas.values()) {

    console.log(pessoa)

}

**POO – Classes e orientação a objeto**

As classes são muito semelhantes a função construtura, mas traz alguns beneficios e facilidades na sua declaração e utilização.

A primeira diferença é que ao declarar uma classe, precisamos utilizar o método constructor para passar seus parametros, conforme o exemplo abaixo:

class Pessoa {

    constructor(*nome*, *sobrenome*) {

*this*.nome = *nome*;

*this*.sobrenome = *sobrenome*;

    }

    falar() {

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

    }

}

const p1 = **new** Pessoa('Luiz', 'Miranda');

A segunda diferença, e uma das mais impactantes, é que não precisamos ficar adicionando os metodos no proto do objeto e ficar mexendo e alterando essa propriedade, já que ao criarmos um método dentro de uma classe, ele já é automaticamente adicinado ao proto da classe.

**Getter e Setter**

// criando uma propriedade privada

const \_velocidade = Symbol('velocidade');

class Carro {

    constructor(*nome*) {

*this*.nome = *nome*;

        // utilizando a propriedade privada

*this*[\_velocidade] = 0;

    }

    // criando um getter para a velocidade

    get velocidade() {

        return *this*[\_velocidade];

    }

    // criando um setter para a velocidade

    set velocidade(*valor*) {

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

        if (*valor* >= 100 || *valor* <= 0) return;

*this*[\_velocidade] = *valor*;

    }

    acelerar() {

        if (*this*[\_velocidade] >= 100) return;

*this*[\_velocidade]++;

    }

    freia() {

        if (*this*[\_velocidade] <= 0) return;

*this*[\_velocidade]--;

    }

}

Em alguns momentos, podemos querer que algumas propriedades de uma classe não possa ser alterada de fora dela ou mesmo criar uma verificação/validação dessa alteração. Para isso servem os getters e setters.

No código acima, temos a criação de uma propriedade privada utilizando o Symbol(), e passamos a utilizar essa propriedade entre colchetes ao invés de ‘ . ’ . Então this.velocidade se torna this[\_velocidade]

Após isso, criamos um getter, que exibirá o valor como ‘velocidade’ caso a propriedade seja chamada de fora da classe:

    // criando um getter para a velocidade

    get velocidade() {

        return *this*[\_velocidade];

    }

Depois do getter, podemos ou não criar um setter, que permite setar novos valores para a propriedade:

    // criando um setter para a velocidade

    set velocidade(*valor*) {

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

        if (*valor* >= 100 || *valor* <= 0) return;

*this*[\_velocidade] = *valor*;

    }

No setter é onde podemos criar validações para checar se o valor pode ou não ser aplicado à propriedade.

**HERANÇA EM CLASSES:**

A herança em classes é bem simples, se comparada a herança nas funções. Para entender, veja exemplo da seguinte classe:

class DispositivoEletronico {

    constructor(*nome*) {

*this*.nome = *nome*;

*this*.ligado = false;

    }

    ligar() {

        if (*this*.ligado) {

            console.log(`${*this*.nome} já está ligado`)

            return;

        }

*this*.ligado = true;

    }

    desligar() {

        if (!*this*.ligado) {

            console.log(`${*this*.nome} já está desligado`)

            return;

        }

*this*.ligado = false;

    }

}

A classe DispositivoEletronico possui dois metodos e recebe um valor. Tendo isso em mente, para criar outras classes e herdar de DispositivoEletronico é só utilizar a palavra extends na declaração, como no exemplo:

// utiliza ao extends para herdar de uma classe

class Smartphone extends DispositivoEletronico {

    constructor(*nome*, *cor*, *modelo*) {

        // utilizando o super() para enviar os parametros da classe pai

*super*(*nome*);

*this*.cor = *cor*;

*this*.modelo = *modelo*;

    }

}

Note que utilizamos o super( ) para enviar os parametros que são da classe pai, da qual a nova classe está herdando. Nesse caso, estamls enviando o nome para a classe DispositivoEletrinico.

Somente fazendo isso, a nova classe Smartphone já está herdando os parametros e os metodos de DispositivoEletrinico.

**ALGUMAS FUNCIONALIDADES JS:**

*campo*.insertAdjacentElement('afterend', div);

O insertAdjacentElement adiciona um elemento em algum local do elemento ‘campo. ‘afterend’ -> adiciona depois do final; ‘beforebegin’ -> antes do começo

const label = campo.previousElementSibling.innerText;

previousElementSibling pega o primeiro elemento irmão anterior ao elemento

**ASSINCRONISMO JAVASCRIPT:**

**Promise:**

A promise é uma funcionalidade que serve para deixar a execução de um código JS assíncrono. Ela faz com que a ordem de execução seja respeitada, esperando uma promise resolver para chamar a próxima. Veja um exemplo:

function esperaAi(*msg*, *tempo*) {

    return **new** *Promise*((*resolve*, *reject*) => {

        if (typeof *msg* !== 'string') reject(**new** *Error*('BAD VALUE'));

        setTimeout(() => {

            resolve(*msg*);

        }, *tempo*);

    });

}

esperaAi('Conexão com BD', rand(1, 3)).then(*msg* => {

    console.log(*msg*);

    return esperaAi('Buscando dados da BASE', rand(1, 3));

}).then(*msg* => {

    console.log(*msg*);

    return esperaAi(22, rand(1, 3));

}).then(*msg* => {

    console.log(*msg*);

}).then(() => {

    console.log('Exibe dados na tela');

})

    .catch(*e* => {

        console.log(*e*);

    });

No código, primeiramente envolvemos uma função de timeout, simulando um processo real que não tem tempo definido, com um construtor Promisse, esse que recebe os argumentos resolve e reject.

Após isso, ao instanciarmos a função esperaAi, adicionamos o then e o catch ao seu final.

then( ) (resolve) -> é quando o código dentro da promise é executado com sucesso

catch( ) (reject) -> executado quando o código dentro da promise gera algum erro

**Métodos uteis - Promise:**

**Promise.all** -> retorna todas as promises (ex, de um array) já resolvidas. Exemplo:

function esperaAi(*msg*, *tempo*) {

    return **new** *Promise*((*resolve*, *reject*) => {

        if (typeof *msg* !== 'string') {

            reject(**new** *Error*('BAD VALUE'));

            return;

        }

        setTimeout(() => {

            resolve(*msg*.toUpperCase() + '- Passei na promise');

            return;

        }, *tempo*);

    });

}

const promises = [

    'Primeiro valor',

    esperaAi('Promisse 1', 3000),

    esperaAi('Promisse 2', 500),

    esperaAi('Promisse 3', 1000),

    // esperaAi(100, 1000),

    'Outro valor',

];

*Promise*.all(promises).then(*valor* => {

    console.log(*valor*);

})

    .catch(*error* => {

        console.log(*error*);

    });

**Promise.race** -> retorna a primeira promisse que for resolvida. Ex:

const promises = [

    esperaAi('Promisse 1', 3000),

    esperaAi('Promisse 2', 500),

    esperaAi('Promisse 3', 1000),

];

*Promise*.race(promises).then(*valor* => {

    console.log(*valor*);

})

    .catch(*error* => {

        console.log(*error*);

    });

*\*IMPORTANTE: retorno nesse caso -> ‘PROMISSE 2- Passei na promise’, pois o promise 2 está para ser resolvido em 500 milisegundos.*

**Promise.resolve** -> retorna um promise já resolvido.

Um exemplo de uso seria na verificação de um navegador, onde é visto se uma página já possui ou não cache. Se ela possuir, é retornado na hora o promise resolvido com o valor da página. Caso não possua, retorna o promise a ser resolvido. Ex:

const promises = [

    esperaAi('Promisse 1', 3000),

    esperaAi('Promisse 2', 500),

    esperaAi('Promisse 3', 1000),

];

function baixaPagina() {

    const emCache = false;

    if (emCache) {

        return *Promise*.resolve('Página em cache');

    } else {

        return esperaAi('Baixei a página', 2000);

    }

}

baixaPagina()

    .then(*dadosPagina* => {

        console.log(*dadosPagina*);

    })

    .catch(*e* => console.log(*e*));

**Promise.reject** -> retorna um promise rejeitado (entra na excessão).

**Async e await:**

Com intuito de deixar o código mais limpo, podemos utilizar as funções async com o await no lugar de then e catch. A diferença entre os dois ficaria dessa forma:

// Promise com then e catch

esperaAi('Fase 1', rand())

    .then(*fase* => {

        console.log(*fase*);

        return esperaAi('Fase 2', rand());

    })

    .then(*fase* => {

        console.log(*fase*);

        return esperaAi('Fase 3', rand());

    })

    .then(*fase* => {

        console.log(*fase*);

        return *fase*;

    })

    .then(*fase* => console.log('Terminamos na fase: ' + *fase*))

    .catch(*e* => console.log(*e*));

// Com asyns e await

async function executa() {

    try {

        const fase1 = await esperaAi('Fase 1', rand());

        console.log(fase1);

        const fase2 = await esperaAi('Fase 2', rand());

        console.log(fase2);

        const fase3 = await esperaAi('Fase 3', rand());

        console.log(fase3);

        console.log('Terminamos na fase: ', fase3);

    } catch (e) {

        console.log(e);

    }

}

executa();

Como é possível observar, utilizando o async + await, temos um código mais limpo e intuitivo (maioria das vezes). O await simplesmente faz com que o código espere a promisse ser resolvida para poder continuar, lembrando que para utilizar o await dentro da função, preciamos utilizar o async.

**Request com XMLHttpRequest:**

O XMLHttpRequest nos permites realizar requests em varios tipos de dados (XML, HTTP). Nesse caso, estamos realizando o request de uma página html utilizando o método GET.

Para realizar a requisição, primeiro criamos uma constante (normalmente nomeada request ou ajax) que irá receber uma arrow function e será responsável pela request.

Dentro dessa função, criamos um objeto utilizando o molde XMLHttpRequest() e abrimos ele.