**Java-Script**

O Javascript é uma das linguagens mais importantes de nosso tempo porque: é uma linguagem que, além de ser a **padrão** nos navegadores atuais, vem **expandindo** para outras áreas como servidores, banco de dados e até plataformas de Internet of Things.

O Javascript cresceu bastante e hoje em dia pode ser encontrado nas mais diversas plataformas de desenvolvimento. Dominá-la é essencial se você deseja desenvolver sites para a Web, já que ela é a linguagem que funciona em todos os navegadores e responsável por dar interatividade e dinamismo as páginas web.

Sabemos que o navegador entende apenas HTML e que se colocarmos scripts diretamente no corpo da mensagem eles serão interpretados como texto. Como resolvemos isso?

Através da tag <script></script>. Tudo que estiver dentro dela será interpretado como script e não texto.

Correto, a tag script possibilita que tudo que estiver dentro dela seja interpretado com um script de Javascript.

Ex: <script>

alert("Olá mundo");

</script>

A função alert é uma das opções para se exibir mensagens com JavaScript, porém, ela não é a única. Outra função útil para isto é o console.log, considerado "mais agradável" para a exibição de uma mensagem por não envolver um pop-up. A mensagem será exibida no **console do navegador**.

<script>

console.log("Oi Mundo");

</script>

**Uso de variáveis**: Variáveis. Você usa variáveis como nomes simbólicos para os valores em sua aplicação. O nome das variáveis, chamados de identificadores, obedecem determinadas regras. Sendo assim eu posso atribuir uma variável com um nome X e dizer um valor para ela , ou melhor dizer o quanto ela recebe, observe o exemplo abaixo:

**<script>**

**var ano = 2050;**

**document.write("Marcos tem " + (ano - 2002) + " anos");**

**document.write("<br>");**

**document.write("Paulo Vitor tem " + (ano - 2006) + " anos");**

**document.write("<br>");**

**document.write("Fabio tem " + (ano - 1977) + " anos");**

**</script>0**

No exemplo acima podemos ver que eu defini um valor a variável e deixei um calculo de ano (nome da variável) – o ano que a pessoa nasceu para calcular sua idade, e toda vez que eu mudo o valor da variável o resultado muda pra todas elas. Posso ainda chegar no meio do texto entre Paulo e Fabio e dizer que “ano” recebe outro valor assim fabio ou qualquer outro abaixo terá outro valor. OBS ( observe que apenas ponho o “ano” = 2017 eu não ultilizo var porque eu só uso o var quando estou declarando a variável pela primeira vez)

document.write("Paulo Vitor tem " + (ano - 2006) + " anos");

document.write("<br>"); ano = 2017;

document.write("Fabio tem " + (ano - 1977) + " anos");

OBS ( observe que apenas ponho o “ano” = 2017 eu não ultilizo var porque eu só uso o var quando estou declarando a variável pela primeira ve

**Outro exemplo de utilização de variavel**

**var idadeMarcos = 20;**

**var idadePaulo = 16;**

**var idadeFabio = 45;**

**var idadeAna = 50;**

**var media = (idadeMarcos + idadePaulo + idadeFabio + idadeAna) /3;**

**document.write('A media de todas as idades é ' + Math.round (media));**

**Prefixo usado para abreviar números muito extensos:**

Math.round()

**Função**

A vantagem de uma função é tornar legível a intenção do que queremos fazer. Se quisermos pular uma linha, por exemplo, criaremos o pulaLinha(), sempre utilizando um verbo e indicando uma ação.

Uma função representa um conjunto de instruções, e é possível ter uma, duas, cem instruções - **não há limites**. Sendo assim, será desnecessário reescrevê-las em diversos pontos no código, basta utilizarmos a função criada.

OBS: Tudo que está inserido em uma função só existe dentro desta, a menos que seja utilizado um **return** para capturar o seu resultado.

<script>

function pulaLinha () {

document.write("<br>");

}

var ano = 2022;

document.write("Marcos tem " + (ano - 2002) + " anos");

pulaLinha();

document.write("Paulo Vitor tem " + (ano - 2006) + " anos");

pulaLinha();

Ultilizamos // (para bloquear uma linha na programação aquela linha permanecerá apenas para fins didáticos)

Parâmetro é o nome que se da para a variável que nós declaramos na definição de uma função. Fazendo isso eu criei uma função na qual uso ela de parâmetro para uma tag especifica, no caso em vez de eu escrever document.write e oque eu quero exibir em seguida , apenas escrevo o nome da minha função que por tras dos panos ela vai aplicar a tag dentro dela no lugar, assim como posso colocar uma função dentro de outra função que seja de parâmetro assim não preciso escrever mais <b> para pular linha , a função faz com que o depois de excutar a primeira função juntamente do texto ela em seguida pula linha automaticamente. (Para que a função seja executada precisamos chamá-la! Logo após a declaração da função, usamos seu nome )

function pulaLinha () {

document.write("<br><br>");

}

function exibir (frase) {

document.write (frase);

pulaLinha ();

}

var ano = 2022;

exibir("Marcos tem " + (ano - 2002) + " anos");

exibir("Paulo Vitor tem " + (ano - 2006) + " anos");

exibir("Fabio tem " + (ano - 1977) + " anos");

exibir("Ana Paula tem " + (ano - 1972) + " anos");

Outro exemplo ultilizando função :

function calculaImc(altura, peso) {

var imc = peso / (altura \* altura);

mostra("O imc calculado é " + imc);

}

FUNCTION RETURN: A declaração ”RETURN” finaliza a execução de uma função e especifica os valores que devem ser retonados para onde a função foi chamada. No exemplo abaixo eu primeiramente crio uma função, dou um nome, ponho ela uma função de parâmetro , dentro dela crio uma variável imc = atrubio o calculo dentro dela , e uso o return imc para retorna o valor daquele calculo pra mim, já fora da função crio uma variável com a altura e peso de cada individuo e puxo essa função nesses valores para realizar o calculo, depois usando a função respectiva de document.write que criei anteriormente com o nome exibir chamo o return para exibir aquele calculo que atribuir na função.

function calculaImc(altura,peso) {

var imc = peso / (altura \* altura);

return imc;}

var imcMarcos = calculaImc(1.84, 84);

var imcPauloVitor = calculaImc(1.89, 75);

exibir(imcMarcos);

exibir(imcPauloVitor);

SEGUE EXEMPLOS:

function calculaImc(altura,peso) {

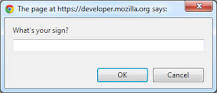
            return imc = peso / (altura \* altura);}

        var imcMarcos = calculaImc(1.84, 84);

        var imcPauloVitor = calculaImc(1.89, 75);

        exibir("A soma dos imc's é : " + Math.round (imcMarcos + imcPauloVitor));

**O que é prompt no JavaScript?**



**prompt**() O Window. **prompt**() exibe uma caixa de diálogo com uma mensagem opcional solicitando ao usuário a entrada de algum texto.

Usando a função prompt em java-script posso interagir com o usuário fazendo com que ele responda perguntas dais quais minha pagina está fazendo , e assim atribuindo um calculo a ela que foi ordenado na função.

function calculaImc(altura,peso) {

            return imc = peso / (altura \* altura);}

        var nome = prompt ("Informe seu nome");

        var alturaInformada = prompt ( nome + ", Informe sua altura");

        var pesoInformado = prompt ( nome + ", Informe seu peso");

        var imc = calculaImc (alturaInformada , pesoInformado);

        exibir(nome + ", o seu IMC é " + Math.round (imc));

Exemplo 2 :

<script>

function pulaLinha() {

document.write("<br>");

}

function mostra(frase) {

document.write(frase);

pulaLinha();

}

var numero1 = prompt("Digite o primeiro número");

var numero2 = prompt("Digite o segundo número");

mostra("O valor de " + numero1 + " vezes " + numero2 + " é: " + (numero1 \* numero2));

**Função parseInt( ):** A função parseInt converte seu primeiro argumento para uma string, analisa, e retorna um inteiro ou NaN . Se não NaN , o valor retornado será a representação decimal inteira do primeiro argumento obtido como um número na base especificada , ou seja converte uma string para um numero inteiro.

function pulaLinha () {

            document.write("<br><br>");

    }

    function exibir (frase) {

            document.write (frase);

            pulaLinha ();

    }

var vitorias = parseInt( prompt ("Entre com o seu numero de vitórias"));

var empates = parseInt(prompt("Entre com o seu numero de empates"));

var pontos = vitorias \* 3 + empates;

exibir("Os pontos do seu time são " + pontos);

Importante lembrar que se quiséssemos ler um nome, por exemplo, não seria necessário utilizar a função parseInt(), porque neste caso queremos que isto seja considerado como texto. Sempre que formos ler uma entrada, e a intenção for trabalhar com números, temos que fazer o parseInt() para que o valor digitado, que por padrão é lido como texto, seja interpretado como um número. retorno da função prompt() é sempre string.

**O que é a estrutura if?**

A condicional if é uma estrutura condicional que executa a afirmação, dentro do bloco, se determinada condição for verdadeira. Se for falsa, executa as afirmações dentro de else. Abaixo um exemplo de sua estruturação

 function calculaImc(altura,peso) {

            return imc = peso / (altura \* altura);}

        var nome = prompt ("Informe seu nome");

        var alturaInformada = prompt ( nome + ", Informe sua altura");

        var pesoInformado = prompt ( nome + ", Informe seu peso");

        var imc = calculaImc (alturaInformada , pesoInformado);

        exibir(nome + ", o seu IMC é " + (imc) .toFixed (1));

        if(imc > 25.0 && imc < 29.9) {

            exibir ("Seu IMC está acima do normal (Sobrepeso).");

        }

        if(imc < 18.5) {

            exibir ("Seu IMC está abaixo do normal (Magreza).");

        }

        if(imc > 18.5 && imc < 24.9  ) {

            exibir ("Parabéns ! Seu IMC está normal.");

        }

        if(imc > 30.0 && imc < 39.9  ) {

            exibir ("Seu IMC está muito acima do normal (Obsidade).");

        }

        if(imc > 40.0) {

            exibir ("Alerta ! Seu IMC está extremo , procure um médico (Obsidade GRAVE).");

        }

Estrutura de repetição While

O laço while, um dos mais simples em Javascript, testa uma determinada condição no início de cada volta do laço e continua fazendo isso enquanto (while) a expressão condicional for avaliada como true. Ou seja a condição vai continuar enquanto (while) a ação requisitada não for terminada.

var limiteAno = parseInt (prompt ("Entre com a ano limite (ANO)"));

var limiteIdade = parseInt (prompt ("Entre com a idade limite (IDADE)"));

var ano = 2002;

var idade = 0;

while (ano <= limiteAno , idade <= limiteIdade) {

exibir("No ano " + ano + " eu terei " + idade );

ano = ano + 1;

idade = idade + 1;

}

OBS: Veja que essa variável só existe por causa de uma finalidade: é ela que quebrará o lanço de repetição do while. A cada repetição do exibir, precisamos incrementá-la para saber que já mostramos, uma, duas, três vezes e por ai vai. Se não tivermos a instrução ano = ano + 1 ( ou idade = idade + 1 ) nunca a condição passada para o while dará false e cairemos em uma repetição infinita, o famoso loop infinito.

**A função isNaN():** determina se o valor é NaN ou não. Tenha cuidado, o cast em isNaN tem regras para serem observadas. Você pode ficar interessado no Number. isNaN() que foi definido no ECMAScript 6 ou você pode usar typeof para determinar se o valor é Not-A-Number, NaN

Estrutura de repetição FOR :

O que é o for no JavaScript?

A instrução for cria um loop que consiste em três expressões opcionais, dentro de parênteses e separadas por ponto e vírgula, seguidas por uma declaração ou uma sequência de declarações executadas em sequência. E praticamente igual a while mas basicamente entra dentro de parêntesis ao em vez de fora.

for(var multiplicador = 1; multiplicador <= 10; multiplicador++) {

mostra(7 \* multiplicador);

}

mostra("FIM");

</script>

A instrução for é dividida em três partes que são separadas por ponto e vírgula. A primeira é destinada a variável de incremento, a segundo à condição de repetição e a última ao incremento da variável de incremento.

Aliás, quando queremos incrementar de um em um, podemos usar o **pós incremento**:

OBS: existe um atalho que e usado de forma pratica em javaScript que e usando ++ no luga de usar { nome da variável = nome da variável + 1 } que quando ultilizamos {nome da variável++} que quer dizer a mesma coisa que +1 da anterior.

Tanto o while() quanto o for() são utilizados em estruturas criadas para repetir instruções, ou seja, estruturas de loops. A escolha entre um e outro é questão de gosto e legibilidade. Com o tempo você saberá quando um é melhor do que o outro, o importante, neste momento, é conhecer as duas estruturas

Processo de acumular valores :

var totalFamiliares = parseInt(prompt("Informe o total de familiares"));

var numero = 1;

totalIdades = 0;

var quantidadeFamiliar = 0;

while (numero <= totalFamiliares) {

quantidadeFamiliar++;

var idade = parseInt(prompt("Informe a idade do " + quantidadeFamiliar + " familiar " ));

totalIdades = totalIdades + idade;

numero++;

}

var mediaDasIdades = totalIdades / totalFamiliares ;

mostra ("A média das idades dos familiares é " + mediaDasIdades .toFixed (1));

mostra ("FIM")

No exemplo acima podemos ver que totalIdades faz com que a idade de cada familiar fique armazenada e some com a próxima dada pelo usuário até por fim dividir todas elas por 3, seguindo o exemplo ficaria 0 + 20 (primeira idade ) depois no seguinte loop 20 a idade fica acumulada ou seja ficaria 20 + 15 (segunda idade).

**Interrompendo uma estrutura de repetição:** para interromper uma estrutura de repetição basta utilizar a função **break** como mostrado abaixo, nesse caso a ideia era fazer com que quando o usuário acertasse o numero pensado o loop acabasse sem dar chance para outra tentativa.

var numeroPensado = Math.round(Math.random() \* 10);

var tentativas = 1;

while(tentativas <= 3) {

var chute = parseInt(prompt("Digite seu chute!"));

if(chute == numeroPensado) {

mostra("Você ACERTOU, o número pensado era " + numeroPensado);

**break**;

} else {

mostra("Você ERROU!");

}

tentativas++;

}

**Repetições alinhadas:**

for(var linha = 1; linha <= 5; linha++) {

for(var coluna = 1 ; coluna <= 1; coluna++) {

document.write("Marcos");

}

pulaLinha ()

Função parseFloat( ) : Aprendemos a utilizar parseInt() para converter um texto em número. Certo? Contudo, ele converte um texto para um número inteiro e nem sempre queremos abdicar dos números decimais. Como podemos então para converter em “números reais”?

Vejamos um exemplo:

var numero = parseInt("12.13");

O valor de numero será 12. Para que as casas decimais sejam mantidas, usamos o método parseFloat():

var numero = parseFloat("12.13");

OBS: Não é necessário usar o parseFloat() quando lemos os dados de peso e altura no cálculo do IMC, são como operações de divisão e multiplicação o JavaScript já realiza a conversão implícita para nós. Contudo, é uma boa prática usar parseInt() ou parseFloat() se queremos ler números inteiros ou decimais fornecido pela função prompt. Nem sempre a conversão implícita vai dar certo, como é o caso do número de vitórias e empates.

FUNÇÃO ARRAY:

Arrays em JavaScript são usados para armazenar vários valores em uma única variável.

**var segredos = [5,10,2,7];**

OBS: usar .length em um array retorna a quantidade de elementos de um array , Quando queremos acessar um elemento específico do array, precisamos passar seu índice, ou seja, sua posição na lista. Contudo, precisamos estar muito atentos porque os índices do array começam de 0 até o tamanho do array menos um.

Todo array é declarado com colchetes, o famoso []. Contudo se tivermos uma declaração desta forma var coisas = []; temos um array, ou seja, uma lista vazia sem qualquer elemento.

Podemos, no momento da declaração de um array, adicionar elementos. Esses elementos podem ser de qualquer tipo conhecido do JavaScript:

var coisas = ["Gisele", 12, true];

Temos como primeiro item do Array uma String, o segundo um número e o último um boleano.

Função **push : faz com que eu possa empurra coisas para dentro do array , assim posso editar a quantidades de itens do array individualmente.**

var segredos = [];

segredos.push(Math.round(Math.random() \* 10));

segredos.push(Math.round(Math.random() \* 10));

segredos.push(Math.round(Math.random() \* 10));

segredos.push(Math.round(Math.random() \* 10));

segredos.push(Math.round(Math.random() \* 10));

**”APIs – Blibioteca javaScript”**

Com <canvas> conseguimos criar a tela de desenho e dizer qual será sua altura e largura.

<canvas></canvas>

Exemplo de pintar um retângulo na tela, primeiro defino a largura e altura do canvas (isso tudo no mundo HTML), depois dentro do javaScript chamo aquela tag do mundo HTML para o mundo javaScript usando **document.querySelector** juntamente da tag ( ) isso nos dará a área para escrever , para criarmos um pincel definimos uma variável com nome pincel que receber de parâmetro uma função, no mundo da programação podemos ter gráficos em 2D, 3D, 4D, enfim, portanto, precisamos informar à tela qual o tipo de pincel, ou seja, o contexto no qual escreveremos na tela. Para isso, utilizaremos o **getContext()** e passaremos o **2d**como parâmetro, depois definimos a cor do nosso quadro pois agora ele está em branco não conseguimos exerga-lo no mundo HTML , ultilizaremos então a função de cor para o pincel **pincel.fillStyle** que recebe a cor que queremos, depois definimos a largura e altura do nosso canvas , reparem que essa largura e altura nos já definimos antes , porem aquela largura e altura era a máxima que sera permitido no nosso quadro, sendo assim agora podemos mexer nesse tamanho como quisermos e claro não passa do limite que definimos,usando então **pincel.fillRect** podemos definir esses parâmetros , Como sabemos, na matemática trabalhamos com a ideia de **planos cartesianos**, onde temos um eixo x, na horizontal, e um eixo y, na vertical.

De acordo com as dimensões que especificamos, começaremos nosso preenchimento no canto superior esquerdo, isso significa que iniciaremos no ponto 0 tanto no eixo x quanto y, depois e so definir normalmente a largura e altura.

<canvas width="600" height="400"></canvas>

<script>

    var tela = document.querySelector("canvas");

    var pincel = tela.getContext ("2d");

    pincel.fillStyle = "lightgrey";

    pincel.fillRect (0 , 0 , 600, 400);

</script>

Para poder modelar o pincel para fazer uma arte para x posição para y e etc fazendo um triangulo por exemplo usamos as funções abaixo , primeiro com **pincel.beginPath ()** para começar o caminho com o pincel, depois ultilizo a função **pincel.moveTo** para mover o pincel para posição que eu quero que ele comece a pintar, la ele vai marcar um ponto que vou utilizar para desenhar de uma linha para outra ultilizando pincel.lineTo , depois que faltar a ultima linha para fechar o desenho ultilizo **pincel.fill();**

 pincel.fillStyle = "yellow";

    pincel.beginPath ();

    pincel.moveTo (300,1);

    pincel.lineTo (1,200);

    pincel.lineTo (300,400);

    pincel.lineTo (600,200);

    pincel.fill();

Para desenhar um cirulo usa a função **pincel.arc**

pincel.fillStyle = 'blue';

    pincel.beginPath();

    pincel.arc(300, 200, 150, 0, 2 \* 3.14);

    pincel.fill();

.para definir uma borda para um desenho ultiliza a propriedade **pincel.strokeStyle = "black" pincel.strokeRect (0,0,50,50);**

**Uma função mais genérica**

Ultilizando uma função mais genérica me permite fazer com que minha função possua uma varias opções de pre definidas como parâmetros e não apenas outras funções , no exemplo abaixo definir um elemento x e y como parâmetro me referindo a o calculo em área do desenho e logo em seguida a cor que irei escolher para o desenho.

function desenhaQuadrado (x, y , cor) {

    var tela = document.querySelector('canvas');

    var pincel = tela.getContext('2d');

        pincel.fillStyle = cor;

        pincel.fillRect (x, y , 50, 50);

        pincel.strokeStyle = "black";

        pincel.strokeRect (x,y,50,50);

}

desenhaQuadrado (0,0, "blue");

desenhaQuadrado (0,50, "red");

desenhaQuadrado (0,100, "yellow");

Automatizar tarefas com código mais genérico.

Podemos criar uma função genérica que receba os dados necessários por meio de parâmetros, e deixar que um loop chame a mesma função várias vezes, com um parâmetro diferente a cada vez.

Repetição de código.

Em vez de escrever uma linha para cada forma que queremos criar, passando manualmente as informações de cada uma, podemos escrever uma função para, por exemplo, criar um quadrado colorido de qualquer tamanho e cor.

Deixar o código organizado e separado.

Utilizamos funções para deixar cada parte do código bem separada, assim sabemos exatamente o que está acontecendo em cada uma. Além do mais, é prática comum utilizarmos nomes bem específicos para cada função, assim quem ler o código não terá problemas em saber o que elas fazem.

Loop de repetição (while) ultilizando desenhos :

Ultilizando o código com x na horizontal.

var x = 0;

while(x < 600) {

desenhaQuadrado(x, 0, 'green');

x = x + 50;

Ultilizando o código com y na vertical.

var y = 0;

while ( y < 400) {

desenhaQuadrado (0,y, "blue");

y = y + 50;

}

Ultilizando for no lugar de while.

for(var x = 0; x < 600; x = x + 50) {

desenhaQuadrado(x, 0, 'green');

Escrevendo texto no desenho :

function desenhaTexto (texto , x , y) {

        var telaPincel = document.querySelector('canvas');

        var pincelTexto = telaPincel.getContext("2d");

        pincelTexto.font='20px Georgia';

        pincelTexto.fillStyle='black';

        pincelTexto.fillText(texto , x , y);

        }

Como vimos, o JavaScript nos permite trabalhar com **eventos**. No caso, utilizaremos a propriedade onclick, tudo que atribuirmos a ela - se for uma função - esta será chamada pelo clique. Assim, conectaremos o exibeAlerta ao onClick:

<canvas width="600" height="400"></canvas>

<script>

    var tela = document.querySelector('canvas');

    var pincel = tela.getContext('2d');

    pincel.fillStyle = "grey";

    pincel.fillRect (0,0,600,400);

function jogo (evento) {

    alert ("Cliquei");

    console.log (evento);

}

tela.onclick = jogo;

</script>

Lembrando que não podemos utilizar os parênteses neste último caso pois, assim executaríamos a função automaticamente, em vez de guardá-la para ser executada ao clique. Neste caso, quem chama a função exibeAlerta? O próprio navegador, ao clicarmos sobre o <canvas>. Além de exibirmos um alerta, é importante que nosso programa seja capaz, também, de identificar em qual posição a tela foi clicada.

Como podemos fazer isso?

Ao clicarmos na tela, como dito anteriormente, o próprio navegador é quem chama a função exibeAlerta. Toda vez que faz isso, ele passa um parâmetro especial para a função - até então, não havíamos feito isso em nosso código.

Nós criaremos este parâmetro, que chamaremos de evento, e utilizaremos o console.log(evento) para que ele seja exibido no navegador: Isso fará com que tenhamos acesso a este parâmetro passado pelo navegador, e assim poderemos descobrir a posição exata do clique. O nome do parâmetro que nós passamos poderia ser qualquer um, o importante é **haver um parâmetro**.

Se expandirmos o MouseEvent, clicando na seta antes de seu nome, há diversos detalhes sobre ele, inclusive a posição do mouse dentro do <canvas>.

Antes de aprendermos a obter estas informações, vamos recapitular:

* Criamos uma função chama exibeAlerta;
* Ela recebe como parâmetro um evento;
* Em seu bloco, ela exibe o alerta 'Cliquei' apenas, e faz um console.log(evento);

Se chamarmos o exibeAlerta, temos que passar um parâmetro, mas não temos como saber de antemão qual ponto da tela será clicado. Assim, quem chama essa função é, exclusivamente, o navegador, ele quem tem o parâmetro que trará para nós as coordenadas da posição do cursor no momento do clique.

Código criado para fazer com que o usuário possa interagir com a pagina fazendo com que toda vez que ele clicar no canvas será exibido o valor de x e y do determinado ponto que ele clicou, porém essa função necessita de uns ajustes pois se deixarmos apenas var x = eventpageX , var y = evento.pageY o valor de x e y irá aparecer mas o html irá considerar a bordar fora do canvas como valor assim para dizer que quero transformar essa parte de fora do canvas que sobrou como parte do canvas uso - tela.offsetLeft e - tela.offsetTop

// código anterior omitido

function exibeAlerta(evento) {

var x = evento.pageX - tela.offsetLeft;

var y = evento.pageY - tela.offsetTop;

alert("posição do clique : " + x + ", " + y);

}

// código posterior omitido

Com esse raciocínio em mente eu posso criar um comando para toda vez que eu clicar na tela faça um circulo

function jogo (evento) {

    var x = evento.pageX - tela.offsetLeft;

    var y = evento.pageY - tela.offsetTop;

    pincel.fillStyle = "blue";

    pincel.beginPath ();

    pincel.arc(x, y, 10, 0, 2 \* 3.14);

    pincel.fill();

    console.log (x + "," + y);

}

    tela.onclick = jogo;

Vamos permitir que o usuário altere a cor da bolinha que é desenhada na tela. As cores serão obrigatoriamente **blue**, **red** e **green**. Perceba que temos uma **lista** de cores e isso deve remetê-lo à aula de Array do primeiro módulo do curso.

Como essa escolha será feita? A cada clique do botão **DIREITO** do mouse, a cor padrão, que é blue, deverá se tornar red. Se o usuário clicar mais uma vez com o botão **DIREITO**, a cor escolhida deverá ser green, **nessa ordem**. Por fim, se clicarmos novamente, voltamos para a cor blue e repetimos a ordem de seleção de cores.

Obs: com o botão **ESQUERDO** faremos os cliques para as bolinhas aparecerem.

É claro que lhe darei uma dica, pois não ensinei a você a associar a execução de uma função ao clique do botão direito do mouse. O evento responsável por isso é o **oncontextmenu**. Para você começar bem, crie um programa com o código anterior e faça um teste desse evento conforme meu exemplo:

CODIGO ANTERIOR +

function mudaCor() {

alert('Funcionou!');

return false;

}

tela.oncontextmenu = mudaCor;

</script>

Nesse exemplo será exibido um alerta , mas para trocar a cor como queremos bastar utilizar array juntamente de if.

Repare na logica atribuída abaixo primeiro criei uma variável cores e dentro dela um array com cores que quero , depois um variável indiceCorAtual = 0 que faz com que eu possa selecionar cada cor desse array lembrando que no array se começa com 0 ,1 ,2 ,3 etc ... para 0 corresponde a primeira opção já 1 a segunda e assim por diante. Depois lá embaixo criei uma função mudaCor que dentro dela coloquei um if ( ) que dentro do parêntese diz que se meu indiceCorAtual for maior ou igual ao cores.length executa a função se ele for maior ou igual ele passara para instrução seguinte indiceCorAtual = 0; fazendo com que volte para a primeira cor blue no caso que e igual a 0 no array pois começa de 0 como dito antes, Não estranhe a última linha com a instrução return false. Ela é importante para que o menu contextual padrão do canvas não seja exibido, ou seja, queremos apenas trocar de cor com o clique do botão e não exibir um menu para o usuário.

 var cores = ["blue","red","green"];

    var indiceCorAtual = 0;

    function jogo (evento) {

    var x = evento.pageX - tela.offsetLeft;

    var y = evento.pageY - tela.offsetTop;

    pincel.fillStyle = cores[indiceCorAtual];

    pincel.beginPath ();

    pincel.arc(x, y, 10, 0, 2 \* 3.14);

    pincel.fill();

    console.log (x + "," + y);

}

    tela.onclick = jogo;

    function mudaCor() {

    indiceCorAtual ++;

    if (indiceCorAtual >= cores.length)

    indiceCorAtual = 0;

    return false;

}

tela.oncontextmenu = mudaCor;

</script>

EXEMPLOS MAIS COMPLEXOS DE ULTILIZAÇÃO DE ARRAY COM FUNÇÕES E VARIAVEIS FAZENDO USO DE BOTOES DO TECLADO !

var cores = ["blue","red","green"];

    var indiceCorAtual = 0;

    var raio = 10;

    var limiteRaio = 50;

    var raioDiminue = 5;

    alert ("BEM-VINDO AO JOGO DA BOLINHA !!!")

    function jogo (evento) {

    var x = evento.pageX - tela.offsetLeft;

    var y = evento.pageY - tela.offsetTop;

    pincel.fillStyle = cores[indiceCorAtual];

    pincel.beginPath ();

    pincel.arc(x, y, 10, 0, 2 \* 3.14);

    pincel.fill();

    console.log (x + "," + y);

    if (evento.shiftKey && evento.altKey) {

    alert('Só aperte uma tecla por vez, por favor!');

    }

    if (evento.shiftKey && raio <= limiteRaio){

        raio = raio + 10;

    }

    if  (evento.altKey && raio > raioDiminue) {

        raio = raio - 5 ;

    }

    pincel.fillStyle = cores[indiceCorAtual];

    pincel.beginPath ();

    pincel.arc(x, y, raio, 0, 2 \* 3.14);

    pincel.fill();

    console.log (x + "," + y);

    }

    tela.onclick = jogo;

    function mudaCor() {

    indiceCorAtual ++;

    if (indiceCorAtual >= cores.length)

    indiceCorAtual = 0;

    return false;

}

tela.oncontextmenu = mudaCor;

**DICAS DE FUNÇÕES :** O evento do JavaScript que permite capturar o movimento do mouse, e logo sua coordenada, é o **onmousemove**, contudo esse evento não é capaz de saber se o botão do mouse está clicando ou não. E agora?

Existem os eventos, onmousedown e onmouseup. O primeiro é disparado toda vez que o botão esquerdo do mouse é pressionado e o segundo quando ele é solto.

**OBS:** Como sabemos, na matemática, o valor de Pi é equivalente a 3.14. Como sabemos, no JS temos algumas funções que iniciam com Math. e nos dão funcionalidades básicas, justamente, temos o Math.PI, que é o que utilizaremos em nosso código daqui pra frente.

**ANIMAÇÕES**

Criamos animações ultilizando funções simples com parâmetros interligados umas com as outras , usando também a logica juntamente de uma função importante: setInterval , que faz com eu defina um intervalo de tempo para uma ação ocorrer , a nossa CPU faz este processo muito rapidamente, antes que possamos visualizar o processo, conseguimos ver apenas o resultado final.

A ideia é podermos esperar alguns segundos entre o surgimento das esferas. Sendo assim ultilizamos dessa função para isso.

<canvas width="600" height="400"></canvas>

<script>

    var tela = document.querySelector('canvas');

    var pincel = tela.getContext('2d');

    pincel.fillStyle = 'lightgray';

    pincel.fillRect(0, 0, 600, 400);

function desenhaCirculo (x, y, raio) {

    pincel.fillStyle = "blue";

    pincel.beginPath ();

    pincel.arc (x, y, raio, 0 , 2 \* Math.PI);

    pincel.fill ();

}

function limpaTela () {

    pincel.clearRect (0,0,600,400);

}

var x = 20

function atualizaTela () {

    limpaTela ();

    desenhaCirculo (x,20,10);

    x ++;

}

setInterval (atualizaTela, 10);

</script>

**Recapitulando:**

* O método setInterval() chama atualizaTela();
* Este, quando executado, faz o limpaTela(), desenhaCirculo() e incrementa o valor de x;
* O setInterval() aguarda 20 milisegundos, para repetir o processo, sempre incrementando o valor de x, o que faz com que nossa esfera "se mova" para a direita.

Esta lógica é fundamental quando pensamos em animações, ou mesmo em criar jogos.

Aprendemos que a função setInterval recebe dois parâmetros. O primeiro é o nome da função que desejamos que o JavaScript chame em um intervalo de tempo. O segundo parâmetro é esse intervalo de tempo em milissegundos.

Toda vez que atualizaTela for chamada, o valor de x deve ser atualizado para que na próxima chamada da função pelo setInterval a bolinha seja desenhada em uma nova posição.

**OBS**: A função setInterval requer dois parâmetros: o código ou função a ser processado e o tempo em milissegundos entre os eventos do temporizador.

**OBS:**problema de **escopo** de variável no mundo da programação. Uma variável pode ter um escopo de função, ou seja, só dentro da função ou ter um escopo chamado de escopo global. Quando declarada fora de uma função, a variável é acessível por qualquer outra função do nosso programa

Nesse exercício de reflexão, vamos detectar onde ele errou?

O problema no código de Herculano está nesta linha:

// MESMO CODIGO ACIMA //

**setInterval(atualizaTela(), 10);**

Lembre-se mais uma vez que a função setInterval ,que já vem pronta no JavaScript, precisa receber dois parâmetros para funcionar. O primeiro é a função que ela deve chamar e o segundo o intervalo de tempo no qual a função deve ser chamada.

Veja que não foi passada a função como parâmetro, mas o retorno dela. **Como** assim Flávio? Se você fizer atualizaTela() está chamando uma função, certo? Mas a função atualizaTela retorna alguma coisa? Ela possui algum return? Não! Sendo assim, para que fique mais claro ainda o motivo disso não funcionar, vamos dividir o código em duas etapas:

var retorno = atualizaTela();

setInterval(retorno, 10);COPIAR CÓDIGO

O código acima faz sentido? Não faz, porque atualizaTela não possui retorno, sendo assim, a variável retorno será undefined e será esse valor passado para o setInterval. Faz sentido setInterval chamar a cada 10 milissegundos algo que é undefined? Não mesmo!

Outro ponto é que queremos que o JavaScript, através do setInterval, chame a função e não faz sentido nós mesmos chamarmos atualizaTela().

Então, para corrigir, basta fazermos:

// veja que não há mais o ()

setInterval(atualizaTela, 10)

A pergunta que lhe faço é. Se não estamos usando o () estamos chamando a função? Não, não estamos. O que fizemos foi passar a função inteira como parâmetro para setInterval que internamente a chamará para nosso programa.

Quando queremos passar uma função como parâmetro para outra, precisamos omitir seus parênteses (), chamamos isso de **callback** em programação

Criando uma bolinha que se mexe com os comandos do teclado !

<canvas width="600" height="400"></canvas>

<script>

var tela = document.querySelector('canvas');

var pincel = tela.getContext('2d');

pincel.fillStyle = 'lightgray';

pincel.fillRect(0, 0, 600, 400);

var x = 20;

var y = 20;

// códigos do teclado

var esquerda = 37

var cima = 38

var direita = 39

var baixo = 40

var taxa = 10;

function desenhaCirculo(x, y, raio) {

pincel.fillStyle = 'blue';

pincel.beginPath();

pincel.arc(x, y, raio, 0, 2 \* Math.PI);

pincel.fill();

}

function limpaTela() {

pincel.clearRect(0, 0, 600, 400);

}

function atualizaTela() {

limpaTela();

desenhaCirculo(x, y, 10);

}

setInterval(atualizaTela, 20);

function leDoTeclado(evento) {

if(evento.keyCode == cima) {

y = y - taxa;

} else if (evento.keyCode == baixo) {

y = y + taxa;

} else if (evento.keyCode == esquerda) {

x = x - taxa;

} else if (evento.keyCode == direita) {

x = x + taxa;

}

}

document.onkeydown = leDoTeclado;

</script>

**Fazendo com que altere bandeiras a cada intervalo de tempo**

**var mostraBrasil = true**

function trocaBandeira(){

    if(mostraBrasil == true) {

        desenhaBandeiraBrasil();

        mostraBrasil=false

    } else {

        desenhaBandeiraAlemanha();

        mostraBrasil= true

    }

}

setInterval(trocaBandeira, 3000)

**String.length**

A propriedade length de um objeto [String](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/String) contém o comprimento da string. length é uma propriedade read-only (somente leitura) de instâncias de string.

Essa propriedade retorna o número de unidades de código presentes na string. [UTF-16](https://pt.wikipedia.org/wiki/UTF-16), a codificação utilizada pelo JavaScript, utiliza uma unidade de código de 16-bits para representar os caracteres mais comuns, mas precisa usar duas unidades para caracteres menos usados, então é possível que o valor retornado por length não seja exatamente o número de caracteres na string.

**Palavra.length** se fosse executada dessa maneira essa função retornaria o valor de 7 pois “palavra” possui 7 caracteres.

Para começarmos a manipular a página, se quiséssemos alterar o título "Meus pacientes" para "Meus clientes", como isso poderia ser feito com JavaScript? Precisaríamos ter acesso ao código do arquivo HTML. Tudo o que estiver contido na tag **<script>** será interpretado como JS e, o que está fora, como HTML. Teremos que levar as funcionalidades criadas em HTML para o mundo JS.

Primeiramente, vamos conhecer o **DOM** (Document Object Model), representação do HTML para o nosso JavaScript, acessível por uma palava do JavaScript chamada document.

No console do navegador, quando digitamos document e pressionamos "ENTER" em seguida, veremos **tudo** o que está na página HTML. O document será a ponte entre o JavaScript e o HTML, e tudo que for alterado nele será alterado na exibição para o usuário.

Vamos experimentar adicionar o document na tag <script>, lembrando que não iremos usá-lo entre "" (aspas), pois ele funcionará como uma varíavel, e as aspas são usadas quando trabalhamos com uma string, e não é uma palavra ou frase que queremos imprimir.

<head>

<meta charset="UTF-8">

<title>Aparecida Nutrição</title>

<link rel="icon" href="favicon.ico" type="image/x-icon">

<link rel="stylesheet" type="text/css" href="css/reset.css">

<link rel="stylesheet" type="text/css" href="css/index.css">

<script>

console.log("Oi Mundo");

console.log(document);

</script>

</head>

Se voltarmos ao navegador e atualizarmos a página, veremos que no JavaScript teremos acesso ao document.

Mas se não quisermos manipular o DOM inteiro, e sim apenas um pedaço, por exemplo, o texto dentro da tag <h1>, localizada acima do fechamento do <header>,

<header>

<div class="container">

<h1>Aparecida Nutrição</h1>

</div>

</header>

Vamos encontrar uma forma de **pesquisar** somente a tag <h1>. Para isto, usaremos o método querySelector(), passando como parâmetro o que queremos encontrar - neste caso, entre aspas, pois queremos o termo exato. No console, iremos digitar: document.querySelector("h1");

Após executarmos o método, ele retornará o conteúdo da tag: document.querySelector("h1");

<h1>Aparecida Nutrição</h1>

OBS: é uma boa prática colocar a tag <script> no fim do HTML, mais precisamente, como último elemento de <body> após o fechamento de <main>: pois o browser, ao carregar a página HTML, vai lendo linha por linha, de cima para baixo. Quando ele chega na tag <script>, ele tenta buscar um h1 na página, porém, isto não está carregado em sua memória. A tag <h1> está **abaixo** do código JavaScript e ainda não foi interpretado pelo navegador, logo, ele não poderá ser selecionado.

Agora que conseguimos selecionar o h1, o que é preciso fazer para alterarmos o texto? Primeiramente, em vez de imprimir, salvaremos a parte selecionada, no caso o h1, dentro da variável titulo. Para isso, usaremos a palavra var: <!-- ... -->

</section>

</main>

<script>

var titulo = document.querySelector("h1");

console.log(titulo);

</script>

</body>

Porém, nosso real objetivo é pegar o conteúdo textual Aparecida Nutrição.

Algumas tags, como h1, h2, p e span, possuem um **conteúdo de texto**. Nesses casos, o JavaScript possui uma propriedade que nos permite acessá-lo: textContent. Vamos testar e imprimir o conteúdo de texto da variável titulo, que representa o h1:

<!-- ... -->

<script>

var titulo = document.querySelector("h1");

console.log(titulo);

console.log(titulo.textContent);

</script>

De volta ao navegador, veremos uma diferença no que será impresso pelos dois console.log()s:

<h1>Aparecida Nutrição</h1>

Aparecida Nutrição

Somente o texto **Aparecida Nutrição** será impresso no segundo console.log(). Então conseguiremos acessar e exibir o conteúdo de texto da tag. E para alterá-lo, basta usar o textContent e passar um novo valor para o titulo, igualando-o a um novo texto:

<script>

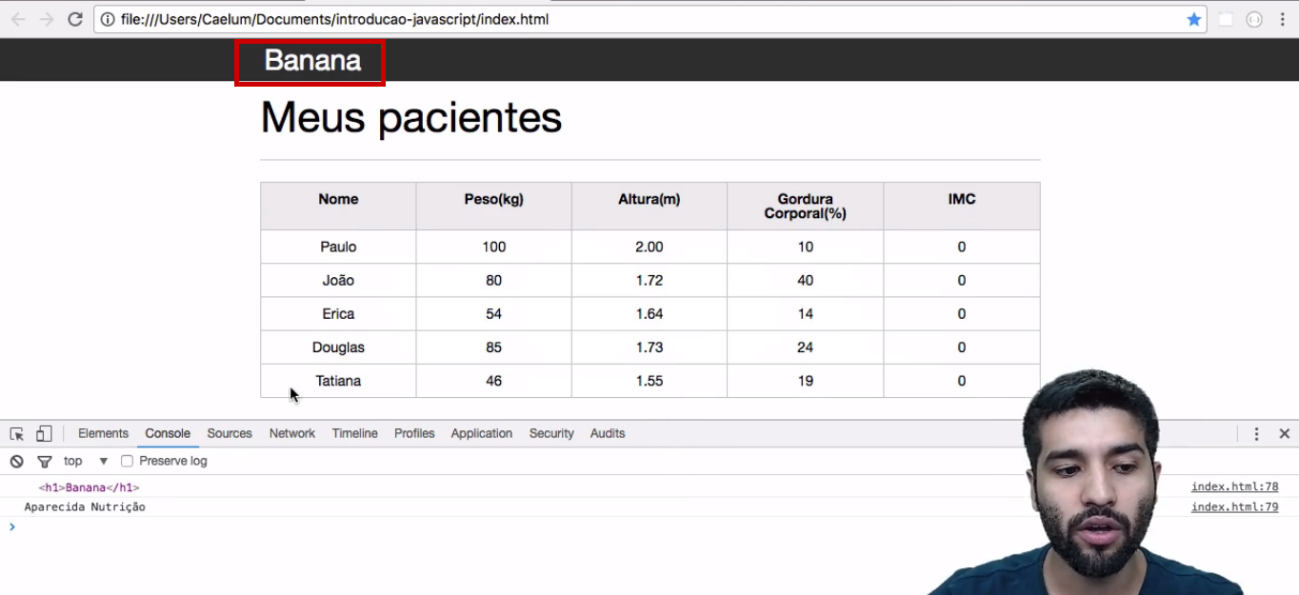
var titulo = document.querySelector("h1");

console.log(titulo);

console.log(titulo.textContent);

titulo.textContent = "Banana";

</script>



Observe que o título da página foi trocado. Se quisermos alterá-lo novamente, por exemplo, para "Aparecida Nutricionista", basta modificar a propriedade textContent.

<script>

var titulo = document.querySelector("h1");

titulo.textContent = "Aparecida Nutricionista";

</script>

Resumindo: aplica uma variável titulo = document.querrySelector (“a tag de mudança desejada”);

titulo.texteContent (que altera o conteúdo de texto dela) = “Mudança requisitada”; isso tudo claro dentro da tag script no final da pagina.

**Boas Praticas**

Além das tags HTML, o querySelector() nos permite buscar por um elemento por meio da **classe**, do **id**, ou seja, dos **seletores CSS**. Por isso e sempre recomendado se utilizar de seletores para fazer uma mudança em uma tag javaScript

<script>

var titulo = document.querySelector**(".titulo");**

titulo.textContent = "Aparecida Nutricionista";

</script>

Trata-se de uma boa prática separarmos o código JavaScript do HTML assim como se faz com o CSS , porém com javaScript e um pouquinho diferente.

Primeiramente crie um arquivo com um nome X escrito.js em sequencia , O próximo passo será mover o código da tag <script> para dentro desse arquivo:  para apontarmos para esse arquivo externo ultilizamos o atributo src (referente ao termo source). Como o arquivo principal.js está na pasta js, especificaremos o caminho completo no atributo

<script src="js/principal.js"></script>

</body>

**Recapitulando informações :** (Nesta aula, nós abordamos vários temas sobre a linguagem JavaScript, vimos que ela ganhou relevância por estar sendo usada de diferentes formas, no navegador, banco de dados, placas Arduíno. É importante que desenvolvedores tenham, pelo menos, um conhecimento básico de como funciona a linguagem.

Vimos algumas funções como console() e alert(), que nos permitem exibir mensagens ao usuário, e conhecemos a utilidade do console de desenvolvedor, presente nos navegadores. Podemos executar códigos JavaScript no console, com isso, conseguimos testar algumas coisas diretamente no navegador.

Vimos também como realizar a busca por algum elemento do HTML, no código JavaScript. Para isto, utilizamos a variável document, que contém todo o conteúdo HTML da página. Quando manipulamos o document, conseguimos manipular o que será exibido ao usuário. Essa manipulação foi realizada por meio da seleção de trechos da tela, feita com o querySelector(). Vimos que este método busca por nome de tags - o que pode trazer problemas -, assim como seletores de CSS.

Na aula, mostramos como criar uma variável no JavaScript e alterar um conteúdo de texto utilizando a propriedade textContent.)

**Console do desenvolvedor**

O console do desenvolvedor é uma ferramenta muito poderosa e versátil para o desenvolvedor Javascript. Com ele conseguimos rapidamente ser notificados de **possíveis erros em nosso código**, podemos **executar funções**, observar a **saída** de algumas funções e **executar pequenos testes** das funcionalidades do Javascript de modo rápido.

É uma boa prática sempre que estivermos testando nosso site ficar com o console aberto, para ficar monitorando possíveis erros em nosso código.

Como utilizar o console é algo bastante corriqueiro, podemos nos aproveitar dos atalhos **CTRL + SHIFT + J** (Win/Linux) ou **CMD + ALT + J** (Google Chrome no Mac. No Safari, o atalho é **CONTROL + ALT + C**, com o menu "Develop" ativado) para ter acesso ao console de modo rápido.

O **document** é uma variável muito importante do Javascript. Ela contém o DOM ou **D**ocument **O**bject **M**odel, que é como o navegador enxerga o HTML utilizado por ele para renderizar a página.

O navegador, ao ler o seu arquivo HTML, cria uma cópia em memória daquele HTML e a partir dessa cópia ele vai desenhando a sua página, colocando as tags e aplicando os estilos. Esta cópia é o que chamamos de **DOM** uma representação em memória do HTML do seu arquivo, que o navegador usa para desenhar a página, e a variável **document** é quem contêm o DOM.

Por isso, quando dizemos que com o Javascript nós estamos **manipulando** o DOM, estamos manipulando o que o navegador renderizou. Então ao trocar algum texto do DOM, o navegador imediatamente desenha novamente aquele texto, pois o DOM é o que o navegador usa para desenhar o seu site.

Outra caracteristica interessante, é que como **manipulamos o DOM** , quando trocamos um texto de um <h1> ou um estilo de um elemento, na verdade estamos alterando a representação **em memória** , o que faz com que o arquivo fonte que contêm seu HTML fique intacto!

**Buscando dados do paciente**

Iremos começar procurando X conteúdo dentro do nosso HTML como fizemos anteriormente mas dessa vez são dados de um paciente que inserimos no HTML , dentro do arquivo javaScript criamos uma variável com um nome coerente no caso paciente + = , usamos o document com query selector para dizer que queremos procurar no nosso html X tag = document.querySelector , juntamente com parênteses mais o nome da tag , id ou class que corresponde a X informação (“#primeiro-paciente”) , e ultilizo console.log +variável para exibir essa busca dentro do meu console de desenvolvedor.

var paciente = document.querySelector("#primeiro-paciente");

console.log(paciente)

no entanto, nosso interesse é no peso e na altura do Paulo, que estão dentro de tags <td>. Mais especificamente, estamos interessados no conteúdo de texto da <td>.

Para isso vamos utilizar novamente uma variável coerente **+ =** , aqui em vez de procurarmos em todo o **document** que não e uma pratica muito utilizada iremos procurar na **class paciente** que possui a informação que procuramos que está dentro da tag **td** , e complementamos com entre parênteses a class respectiva a tal informação

<tr class="paciente" id="primeiro-paciente">

<td class="info-nome">Paulo</td>

<td class="info-peso">100</td>

<td class="info-altura">2.00</td>

<td class="info-gordura">10</td>

<td class="info-imc">0</td>

</tr>

var tdPeso = paciente.querySelector(".info-peso");

console.log(Peso);

faço esse mesmo processo para encaminhar a altura do paciente

var tdAltura = paciente.querySelector(".info-altura");

var altura = tdAltura.textContent;

Agora com essas duas informações vou calcular o IMC ultilizando apenas java-Script, primeiro eu busco essas informações exatamente como antes depois ultilizo da var imc = mais o calculo em imc que e peso dividido por altura vezes altura. E puxo essa informação para aparecer na minha tabela usando a variável mais textContent = (imc);

var tdImc = paciente.querySelector(".info-imc")

var imc = peso / (altura \* altura);

tdImc.textContent = (imc) .toFixed (1);

posso melhorar mais ainda com a utilização do if eu posso ajudar o meu usuário para dizer que caso a altura ou o peso esteja errado tipo muito acima do normal aquela opção será invalida , sendo assim adicionamos que

if(peso <= 0 || peso >= 1000){

console.log("Peso Inválido!");

que se o peso for menor ou igual a zero assim também como se for igual ou maior que zero no console.log o peso aparecera invalido. Mas nem todo mundo presta atenção no console.log para poder estar vendo se essa informação está correta ou não para isso seria melhor dizer que se ocorrer o que está descrito a conta não será realizada e uma mensagem de erro aparecerá aonde fica o resultado informando o erro sendo assim habilite uma variavel

var pesoEhValido = true

informando que se o peso for valido e verdade ou seja correto, dentro do if do peso coloque pesoEhValido = false , para indicar que se for false tal texto usando text.content irá aparecer

if(peso <= 0 || peso >= 1000){

console.log("Peso Inválido!");

pesoEhValido = false;

tdImc.textContent = "Peso Inválido!"

faça o mesmo para altura agora para informar que o canculo só sera feito caso peso e altura sejam verdadeiros ou seja corretos ponha assim (o & siguinifica e)

if( alturaEhValida && pesoEhValido) {

var imc = peso / (altura \* altura);

tdImc.textContent = (imc) .toFixed (1);

}

O código responsável por calcular o IMC do paciente e as validações dos dados é, praticamente, todo o arquivo principal.js. Então, bastaria copiarmos esse código, modificar a classe primeiro-paciente e os nomes das variáveis, que já validaríamos outro paciente.

No entanto, a **repetição de código** não é uma boa prática de programação, o ideal é sempre reaproveitar o trabalho já pronto. No caso, poderíamos fazer um loop - conhecido na lógica de programação -, para conseguirmos iterar por cada linha da tabela, ou seja, por cada paciente. Desta forma, selecionaremos os dados de cada um deles e calcularemos os IMCs.

Então, se quisermos iterar por todos os pacientes, devemos selecioná-los - e não somente um deles, como estamos fazendo atualmente. Se analisarmos o código da tabela no index.html, veremos que **todos os pacientes** possuem a classe paciente:

Apesar de termos selecionado por classe, só nos foi retornado o primeiro paciente, a primeira linha com a classe paciente. Isso porque a função querySelector() nos retorna apenas **um elemento**, independentemente do que passarmos para a mesma. Se estivermos interessados em buscar vários elementos com a classe .paciente, devemos utilizar querySelectorAll():Agora que temos todos os pacientes da tabela, devemos iterar por eles, e para cada um deles executaremos o código responsável por calcular o IMC e validar o peso e a altura.

Há várias formas de fazermos isso, e uma delas, bastante conhecida por todo mundo que já viu algo de lógica de programação é o loop for. Ele receberá três argumentos: a declaração da variável inicial, até onde queremos que essa variável cresça, e o que queremos fazer no fim de cada iteração. Por exemplo:

var pacientes = document.querySelectorAll(".paciente");

for (var i = 0; i < 5; i++) { }

Mas não queremos percorrer um número fixo, e sim percorrer a lista de pacientes. Todo array possui uma propriedade chamada length, que nos informa o seu tamanho. Então vamos usá-la para percorrer até o tamanho da lista de pacientes:

Feito isso, temos acesso ao pacientes[i], e bastará movermos o código que calcula o IMC e faz a validação para dentro do for. Além disso, faremos um ajuste para não precisarmos alterar todo o código. Usaremos um pequeno "truque", criando a variável paciente, que será um atalho pacientes[i];

var paciente = pacientes[i];

SEGUE O CÓDIGO COMPLETO ABAIXO :

var titulo = document.querySelector(".titulo");

titulo.textContent = "Dr Marcos Maio - Nutrição";

var pacientes = document.querySelectorAll(".paciente");

for(var i = 0; i < pacientes.length ; i++){

        var paciente = pacientes[i];

        var tdPeso = paciente.querySelector(".info-peso");

        var peso = tdPeso.textContent;

        var tdAltura = paciente.querySelector(".info-altura");

        var altura = tdAltura.textContent;

        var tdImc = paciente.querySelector(".info-imc")

        var pesoEhValido = true

        var alturaEhValida = true

        if(peso <= 0 || peso >= 1000){

            console.log("Peso Inválido!");

            pesoEhValido = false;

            tdImc.textContent = "Peso Inválido!" }

        if(altura <= 0.50 || altura >= 2.50){

            console.log("Altura Inválido!");

            pesoEhValido = false;

            tdImc.textContent = "Altura Inválido!" }

        if( alturaEhValida && pesoEhValido) {

            var imc = peso / (altura \* altura);

            tdImc.textContent = (imc) .toFixed (1);}

}

Faremos ajustes para deixar os nossos erros mais identificáveis. Para alterarmos a cor do fundo, selecionaremos a linha (cada tag tr) - o que já fizemos anteriormente e será representada pela variável paciente. Nosso objetivo é modificar o **estilo** do elemento, portanto utilizaremos a propriedade style, com os estilos aplicados. Ou seja, vamos alterar o CSS aplicado. Se adicionarmos style.color, poderíamos alterar a cor da fonte na linha. paciente.style.color = "red";

Assim, por conseguirmos alterar a cor da fonte usando a propriedade color, poderemos modificar o fundo da linha ao adicionarmos a propriedade backgroundColor. Porém, se tentarmos escrever da mesma forma como é feito no CSS (background-color), nosso código não funcionará. Quando queremos modificar um estilo com duas palavras, o hífen (-) não é aceito pela linguagem JavaScript. Teremos que utilizar o padrão **Camel Case** - no qual as palavras compostas ou frases são iniciadas com maiúsculas e unidas sem espaços.

É importante mantermos essa separação dos "mundos": o HTML é utilizado para marcar os elementos, o JavaScript para adicionar dinamismo e manipular a página, enquanto o CSS é responsável por estilizar. Logo, o ideal não é modificar o estilo da linha no código JavaScript, então qual é o meio termo entre não modificar o estilo diretamente, mas conseguir alterá-lo, no JavaScript?

É uma boa prática, em vez de adicionarmos um estilo, incluirmos uma classe ao elemento.

Então no mundo css criei uma class

.paciente-invalido {

background-color: lightcoral;}

De volta ao JavaScript, assim como temos o style\*\* para acesso aos estilos e o textContent para acessarmos o conteúdo de texto de um elemento, teremos a classList para acessarmos as classes. E usando o método add, adicionaremos a classe desejada, no caso, paciente-invalido: Se o designer quiser modificar a cor de fundo, deverá modificá-la no CSS e não mais no JavaScript.

**Escutando eventos**

Atualmente, para adicionar um novo paciente, ela deve acessar o arquivo index.html e incluir uma nova <tr> na tabela, juntamente com as tags <td>, e modificar os dados da pessoa. Depois disso, uma nova linha surgirá na tabela.

Ou seja, o HTML é alterado manualmente, o que não é prático. Seria muito simples se em vez de adicionarmos o paciente diretamente no HTML, houvesse um formulário no site, o qual ela pudesse preencher com os dados do paciente. E, após clicar-se no botão "Adicionar", o paciente seria adicionado à tabela. É exatamente isso que iremos implementar.

Quando a Aparecida preencher esses campos e clicar no botão "Adicionar", uma nova linha na tabela deverá ser criada, com a exibição dos dados recém adicionados através do formulário. Mas para fazê-lo, primeiramente devemos entender algumas funcionalidades do JavaScript. A ação de "perceber o que o usuário está fazendo na página" é o que chamamos de **evento** do browser, que pode ser escutado com o JavaScript. Ações como clicar, duplo clique, scrollar, passar o mouse em cima, são exemplos dos tipos de interação que o usuário pode fazer com a página.

Para escutarmos um evento de clique, devemos especificar em qual elemento queremos fazê-lo. Se queremos identificar o momento do clique do botão, usaremos a variável titulo seguida de um "escutador de evento" que escutará as interações dos usuários, addEventListener(). Ela será responsável por adicionar o escutador de evento, como já diz o nome traduzido para o português. Em que tipo de evento estamos interessados? No caso, justamente no evento de click. Queremos que algo aconteça quando o usuário clicar no elemento. Indicaremos o que deve ser feito usando-se a função mostraMensagem(). Dentro dela, adicionaremos console.log("Olá, eu fui clicado!").

titulo.addEventListener("click", mostraMensagem);

function mostraMensagem(){

console.log("Olá eu fui clicado!");

}

Uma outra abordagem que poderíamos adotar seria usarmos uma **função anônima**. Em vez de declararmos mostraMensagem como um parâmetro addEventListener(), poderíamos chamar diretamente function():

titulo.addEventListener("click", function (){

console.log("Olha só posso chamar uma função anônima.")

});

Observe que a função ainda possui as chaves ({}), mas não recebe um nome. Funções anônimas são bastante utilizadas, principalmente no caso de eventos.

var botaoAdicionar = document.querySelector("#adicionar-paciente");

botaoAdicionar.addEventListener("click", function(){

console.log("Oi, cliquei no botão.");

});

No entanto, se tentarmos clicar no botão "Adicionar" do formulário, a mensagem não será exibida no console. Por que isso acontece?

Por padrão, sempre que clicamos em um botão contido em uma tag <form> do HTML, os seus dados serão enviados para outra página. Como não especificamos uma página para ser o alvo da tag <form>, a única ação realizada é a limpeza dos dados, e a página sendo recarregada em seguida. Ao fazermos isto, além do formulário, o console fica limpo também - por isso, não veremos a mensagem.

O evento de clique está sendo escutado corretamente, porém, como a página é recarregada rapidamente, não conseguiremos ver a mensagem impressa no console. Desta forma, não conseguiremos salvar os dados do paciente na tabela, nem exibir a mensagem.

O comportamento padrão do botão que se encontra no formulário é limpar os dados preenchidos nos campos, recarregar a página e enviar os dados. Nós queremos evitar esse comportamento padrão.

Para isso, devemos chamar uma função característica dos eventos, que faz com que o evento não faça seu comportamento padrão. Desta forma, ele não executará este comportamento e reagirá conforme o que pedimos no evento, no caso, a exibição de uma mensagem. Implementaremos a função event.preventDefault():

Essa função faz exatamente o que seu nome diz, prevenindo o comportamento padrão do evento. Mas para utilizá-la, receberemos como segundo parâmetro a função anônima, o event:

var botaoAdicionar = document.querySelector("#adicionar-paciente");

botaoAdicionar.addEventListener("click", function (event) {

    event.preventDefault ();

    console.log ("olha eu fui clicado")

})

Para adicionarmos o paciente na tabela, devemos capturar seus dados do formulário. Para isto, sabemos que devemos trazê-los do mundo HTML para o do JavaScript, selecionando-os.

var form = document.querySelector("#form-adiciona");

Para acessar o valor de um campo em JavaScript, utilizaremos a propriedade **value**:

   var nome = form.nome.value;

    var peso = form.peso.value;

    var altura = form.altura.value;

    var gordura = form.gordura.value;

Com isso, conseguimos trazer os dados do formulário do HTML para o JavaScript, e adicionar o paciente na tabela.

## **Criando um elemento HTML com JavaScript**

Com os dados em mãos, como vamos fazer para "criar" um paciente? Até agora vimos como trazer algo do mundo HTML para o mundo JavaScript, mas agora é o contrário: temos os dados no mundo JavaScript e queremos criar uma nova linha na tabela com eles, que é no mundo HTML. Ou seja, queremos **criar um elemento HTML** utilizando JavaScript.

Para isso, vamos utilizar a função **createElement**, que como o próprio nome indica, cria um elemento HTML. Poderemos olhar o HTML da tabela e ver que se queremos criar um novo paciente, devemos criar uma nova **tr**, que por sua vez possui várias **tds**, cada uma com o seu dado.

Logo, se queremos criar um paciente, vamos criar uma nova **tr**. Para criarmos um elemento, precisaremos informar onde queremos criar, isto é, em **document**: var pacienteTr = document.createElement("tr");

Agora que sabemos criar um elemento, podemos criar as **tds**:

var nomeTd = document.createElement("td");

    var pesoTd = document.createElement("td");

    var alturaTd = document.createElement("td");

    var gorduraTd = document.createElement("td");

    var imcTd = document.createElement("td");

Só que criamos seis elementos distintos, seria como se tivéssemos criado um HTML assim:

<tr></tr>

<td></td>

<td></td>

<td></td>

<td></td>

<td></td>

Sabemos que temos que colocar os dados dentro das tds, e depois pegá-las e colocá-las na tr. Vamos por passos, primeiro colocando os dados dentro das tds, que já sabemos como é feito.

Lembra quando alteramos o conteúdo de texto do título? Utilizamos a propriedade **textContent**, passando para ela o novo dado. Pois aqui é a mesma coisa, vamos alterar o conteúdo de texto das **tds**, passando os seus respectivos valores, capturados do formulário:

 nomeTd.textContent = nome;

    pesoTd.textContent = peso;

    alturaTd.textContent = altura;

    gorduraTd.textContent = gordura;

    var imcTd = document.createElement("td");

## Utilizando o JavaScript para adicionar um elemento dentro de outro

Agora devemos adicionar cada **td** dentro da **tr**. No JavaScript existe uma função chamada **appendChild**. Ela adiciona como filho o elemento passado a ela como parâmetro. Logo, vamos chamar essa função na **tr**, passando a **td** como parâmetro, uma por uma:

  pacienteTr.appendChild(nomeTd);

    pacienteTr.appendChild(pesoTd);

    pacienteTr.appendChild(alturaTd);

    pacienteTr.appendChild(gorduraTd);

    pacienteTr.appendChild(imcTd);

Agora já temos algo mais próximo do que é um paciente, e falta colocarmos ele dentro da tabela. Para isto, vamos selecionar a tabela, o **tbody** do HTML através do seu id, e utilizar a função **appendChild** para adicionar a **tr** como sua filha:

 var tabela = document.querySelector("#tabela-pacientes");

    tabela.appendChild(pacienteTr);

Agora, no momento em que preenchemos os dados do formulário e clicamos em **Adicionar**, o paciente é adicionado à tabela!

Quando estamos adicionando um paciente na tabela por meio do formulário, atualmente, a linha fica sem o resultado do cálculo do IMC.

Ou seja, precisaremos calcular o IMC também no form.js, não só no código que atualmente está preso no calcula-imc.js. Queremos **reaproveitar** o código, e isto será possível se chamarmos uma função que calcule o IMC em **form.js**. A função calculaImc() receberá peso e altura como parâmetros**: imcTd.textContent = calculaImc(peso, altura) OBS:(isso no form não no calcula imc) ;**Reaproveitando assim a lógica de cálculo do IMC. Vamos criá-la em seguida, e ela já retornará o IMC com duas casas decimais: **OBS:( isso no calcula imc , não no form).**

function calculaImc (peso,altura){

    var imc = 0;

    imc = peso / (altura \* altura);

    return imc.toFixed (1);}

**OBS**:Para cada evento existente no JavaScript, há a propriedade on + nomeDoEvent. No caso, temos onclick para o evento click, onmouseover para o evento mouseover e assim por diante. Nesse caso, a função que desejamos executar é atribuída direto na propriedade. No entanto, essa forma tem uma limitação, como estamos guardando a função em uma propriedade, se adicionarmos outra função, essa sobrescreverá a anterior. Sendo assim, a boa prática é trabalhar com addEventListener() mesmo que você só queira adicionar um única função para ser executada quando o evento ocorrer. Porque mais tarde, se outro desenvolvedor quiser adicionar mais uma função para também ser chamada quando aquele evento ocorrer, não há risco de substituir a função existente pela adicionada.

**Dicas:** Leopoldo se vangloriava do seu arsenal de funções criadas por ele para resolver problemas do dia a dia. Todas elas eram declaradas no arquivo minhas-funcoes.js.

Você enxerga desvantagens em declarar todas as funções em um arquivo? Há alguma vantagem? Pense a respeito e em seguida veja a opinião do instrutor.

VER OPINIÃO DO INSTRUTOR

### Opinião do instrutor

A vantagem de declararmos todas as funções em um único arquivo é favorecer sua importação em nossas páginas, pois se fossem vários arquivos precisaríamos importar um a um.

Uma desvantagem dessa abordagem é que fica complicado para mais de um desenvolvedor trabalhar com o mesmo arquivo ao mesmo tempo. Se o desenvolvedor A precisa alterar a função X e o desenvolvedor B a função Y o risco do trabalho ser perdido quando o arquivo for atualizado é grande. Mesmo com ferramentas de versionamento de código o desenvolvedor terá que resolver conflitos que são bastantes comuns nessa abordagem.

Outro ponto é o seguinte. Se das 100 funções declaradas no arquivo apenas um necessitar manutenção, caso o desenvolvedor cometa algum erro de sintaxe, isso comprometerá todas as funções, pois nada será mais carregado. Sendo assim, as chances de introduzir problemas quando for resolver um são bem maiores.

Quando temos arquivos separados, cada arquivo possui uma responsabilidade e uma ou mais funções que fazem sentido naquela responsabilidade. Por exemplo, podemos ter um arquivo chamado conversao.js e nele termos funções que convertem valores monetários com R$ para números, e números para o formato com R$. Além disso, se separamos outras funcionalidades por arquivo, quando alterarmos esses arquivos, se cometermos algum erro, apenas uma ou mais funções do arquivo serão comprometidas e o restante dos outros arquivos poderão ser carregados sem problema (a não ser que um arquivo dependa do outro).

Por fim, assim como na vida real geralmente guardamos peças de roupas por categorizações que julgamos satisfazerem nossa organização, separar arquivos por grupos comuns também nos ajuda a encontrar e dar manutenção em nosso código.

Continuando com as melhorias e refatorações do nosso código: e aplicando-se boas práticas, seria interessante quebrarmos o código do form.js em funções menores, considerando que atualmente temos uma função com várias responsabilidades, como capturar os valores do formulário, criar a tr e as tds do paciente, colocar os valores nas linhas, e por último colocar as tds na tr, e por fim a tr na tabela. Isto é, há quatro funcionalidades em uma mesma função, o que dificulta a manutenção do código.

Imagine outro desenvolvedor tendo que interpretar o bloco de código gigante, ele terá dificuldade para entender qual a sua utilidade, pois isto não está explícito. O ideal é quebrarmos o código em várias funções menores, o que, além de deixá-lo mais organizado, faz com que as responsabilidades sejam separadas, cada função com uma diferente

## **Função para capturar os dados do formulário**

A primeira coisa que podemos fazer é extrair a responsabilidade de capturar os dados do paciente do formulário para uma nova função que receberá o nome obtemPacienteDoFormulario(). Ela receberá o formulário por parâmetro e extrairá os dados dele:

Criaremos a função obtemPacienteDoFormulario(), para onde iremos mover as variáveis nome, peso, altura e gordura.

function obtemPacienteDoFormulario(form) {

var nome = form.nome.value;

var peso = form.peso.value;

var altura = form.altura.value;

var gordura = form.gordura.value;

}

Esse código está pegando todos os valores e extraindo para variáveis. O nome, peso, altura e gordura são características do paciente. Logo, eles pertencem ao mesmo paciente e poderiam ser representados pela mesma coisa. Quando falamos em **representar um paciente**, falamos de **objetos**. Nas linguagens de programação, objetos representam coisas do mundo real, ou mesmo da programação.

Ao criarmos um paciente, sabemos que ele deve ter um nome, peso, altura e gordura. Então, agruparemos todas as características em uma mesma variável criando um objeto em JavaScript usando **chaves** ({}):

Dentro das chaves, passamos as **propriedades** do objeto, que nada mais são que as suas características. Para criar uma propriedade, passamos o seu nome e o seu valor, mas não com um igual e sim com dois pontos. Por exemplo, a propriedade nome:

function obtemPacienteDoFormulario(form) {

var paciente = {

nome: form.nome.value,

peso: form.peso.value,

altura: form.altura.value,

gordura: form.gordura.value

}

return paciente;

}

Na parte de cima do arquivo, vamos declarar a variável paciente.

//Extraindo informacoes do paciente do form

var paciente = obtemPacienteDoFormulario(form);

**Obs:** quando declaramos um objeto no Javascript utilizamos o sinal de **:** para separar propriedades e seus valores

**RESUMINDO O CODIGO PARA FACILITAR ENTENDIMENTO**

Outra responsabilidade a ser extraída é a criação da tr e das tds do paciente.

Na parte de baixo do arquivo, criaremos a função montaTr, que receberá um paciente como parâmetro e, como o próprio nome indica, montará a tr com os dados:

function montaTr(paciente) {

var pacienteTr = document.createElement("tr");

return pacienteTr;

}

Em seguida, devemos preenchê-la com as tds do paciente. Se movermos o código referente aos dados do paciente para dentro da função, ela ficaria mais legível:

function montaTr(paciente) {

var pacienteTr = document.createElement("tr");

var nomeTd = document.createElement("td");

var pesoTd = document.createElement("td");

var alturaTd = document.createElement("td");

var gorduraTd = document.createElement("td");

var imcTd = document.createElement("td");

nomeTd.textContent = paciente.nome;

pesoTd.textContent = paciente.peso;

alturaTd.textContent = paciente.altura;

gorduraTd.textContent = paciente.gordura;

imcTd.textContent = paciente.imc;

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd);

pacienteTr.appendChild(alturaTd);

pacienteTr.appendChild(gorduraTd);

pacienteTr.appendChild(imcTd);

return pacienteTr;

}

As tags td serão criadas, e então preenchidas com paciente.nome, paciente.peso, paciente.altura, paciente.gordura e já não precisaremos calcular o IMC, pois o cálculo foi feito em paciente.imc.

Por fim, chamaremos a função montaTr quando o botão for clicado. A função ficará dentro da variável pacienteTr:

// Cria a tr e a td do paciente

var pacienteTr = montaTr(paciente);

## Adicionando classes aos elementos

Ao inspecionarmos o HTML da nossa tabela, veremos que os pacientes adicionados por meio do formulário não possuem algumas características dos pacientes nativos. A tr do paciente nativo possui a classe paciente, assim como as tds - cada uma com uma classe indicando a informação contida na td. Já os pacientes que adicionamos com o formulário não possuem classes - tanto na tr quanto nas tds. Ou seja, não estamos criando um paciente exatamente igual ao paciente nativo.

Vamos alterar o código da função montaTr para criar um paciente com as classes corretas. Já sabemos como adicionar uma classe a um elemento, a seguir, adicionaremos a classe paciente na tr. Para isso, usaremos o método add():

function montaTr(paciente){

var pacienteTr = document.createElement("tr");

pacienteTr.classList.add("paciente");

MAIS CODIGO ABAIXO

Se testarmos no navegador, veremos que a tr será criada com a classe. Porém, falta adicionarmos as classes nas tds, por exemplo, info-nome e info-peso:

var nomeTd = document.createElement("td");

nomeTd.classList.add("info-nome");

nomeTd.textContent = paciente.nome;

Temos que fazer esse código para todas as tds, criar o elemento, incluir a classe e o valor. Observe que por termos separado as funções, já sabemos onde fazer as alterações.

## **Função para criar e montar uma td**

Quando identificamos códigos repetidos, temos a opção de exportá-los para uma função, que será responsável por eles. A função montaTd criará a td e adicionará a classe juntamente com o dado. Como a classe e o dado variam de acordo com a td, iremos recebê-los por parâmetro na função:

function montaTd(dado, classe) {

var td = document.createElement("td");

td.classList.add("info-nome");

td.textContent = paciente.nome;

return td;

}

Porém, teríamos que ter diversas linhas de código fazendo a mesma tarefa. O código repetidamente criaria a td, adicionaria uma classe e depois o dado. Será que existe alguma forma de simplificarmos esse trabalho? Sim, podemos criar diretamente uma função que monta as tags td: montaTd().

function montaTd(dado){

var td = document.createElement("td");

td.textContent

}

No td criado, deve ser adicionado como o conteúdo de texto o dado, além de uma classe. Com as alterações, o trecho ficará da seguinte maneira:

function montaTd(dado,classe){

var td = document.createElement("td");

td.textContent = dado;

td.classList.add(classe);

return td;

}

Assim, deixaremos bem claro quais são as responsabilidades do nosso código, separando em uma função a criação da tag td. Para cada uma delas, chamaremos a função montaTd().

var nomeTd = montaTd(paciente.nome, "info-nome");

var pesoTd = montaTd(paciente.peso, "info-peso");

var alturaTd = montaTd(paciente.altura, "info-altura");

var gorduraTd = montaTd(paciente.gordura, "info-gordura");

var imcTd = montaTd(paciente.imc, "info-imc");

É possível adicionarmos o paciente pelo formulário e inspecionarmos o seu HTML. Assim, sua estrutura, incluindo as classes, está igual à de um paciente nativo.

## Limpando o formulário após adicionar o paciente

**OBS: É possível "enxugar" ainda mais o nosso código, adicionando o montaTd diretamente no appendChild().**

pacienteTr.appendChild(montaTd(paciente.nome, "info-nome"));

pacienteTr.appendChild(montaTd(paciente.peso, "info-peso"));

pacienteTr.appendChild(montaTd(paciente.altura, "info-altura"));

pacienteTr.appendChild(montaTd(paciente.gordura, "info-gordura"));

pacienteTr.appendChild(montaTd(paciente.imc, "info-imc"));

Após adicionarmos um paciente na tabela, os dados continuarão no formulário.

É recomendável limpá-los para não corrermos o risco de adicionarmos pacientes iguais. Poderemos limpar os campos do formulário chamando a função reset() depois de inserirmos o paciente na tabela.

var form = document.querySelector("#form-adiciona");

var paciente = obtemPacienteDoFormulario(form);

var pacienteTr = montaTr(paciente);

var tabela = document.querySelector("#tabela-pacientes");

tabela.appendChild(pacienteTr);

form.reset();

Nesta aula vimos boas práticas, aprendemos que não é bom trabalharmos com um código gigantesco em um único arquivo - o ideal é quebrá-lo, dividindo as responsabilidades em diferentes arquivos, simplificando a manutenção. Se temos um problema no formulário, por exemplo, saberemos que devemos trabalhar com o arquivo form.js - em vez de buscarmos o erro em um arquivo de 300 linhas. Além disso, vimos que também é boa prática separarmos as responsabilidades em funções, deixando o código mais legível. Cada linha tem uma função específica, independente ao restante do código, que está mais organizado e legível.

Falamos também sobre os objetos do JavaScript, que possuem características e representam coisas do mundo real. No nosso caso, o objeto representa um paciente, mas já havíamos trabalhado com outros, porém, sem denominá-los como tal. Para criarmos um objeto, usamos as chaves ({}) e as propriedades, separando com vírgula as diferentes características.

**Segue abaixo o código completo :**

var botaoAdicionar = document.querySelector("#adicionar-paciente");

botaoAdicionar.addEventListener("click", function(event) {

    event.preventDefault();

    var form = document.querySelector("#form-adiciona");

    //Extraindo informações do paciente do form

    var paciente = obtemPacienteDoFormulario(form);

    //Criar a tr a td do paciente

   var pacienteTr = montaTr(paciente);

    //Adicionando o paciente na tabela.

    var tabela = document.querySelector("#tabela-pacientes");

    tabela.appendChild(pacienteTr);

    form.reset ();

});

    function obtemPacienteDoFormulario (form) {

    var paciente = {

        nome:form.nome.value,

        peso:form.peso.value,

        altura:form.altura.value,

        gordura:form.gordura.value,

        imc: calculaImc(form.peso.value,form.altura.value)

    }

    return paciente

}

function montaTr (paciente) {

    var pacienteTr = document.createElement("tr");

    pacienteTr.classList.add("paciente");

    var nomeTd = montaTd(paciente.nome,"info-nome");

    var pesoTd = montaTd(paciente.peso,"info-peso");

    var alturaTd = montaTd(paciente.altura,"info-altura");

    var gorduraTd = montaTd(paciente.gordura,"info-gordura");

    var imcTd = montaTd(paciente.imc,"info-imc");

    pacienteTr.appendChild(montaTd(paciente.nome,"info-nome"));

    pacienteTr.appendChild(montaTd(paciente.peso,"info-peso"));

    pacienteTr.appendChild(montaTd(paciente.altura,"info-altura"));

    pacienteTr.appendChild(montaTd(paciente.gordura,"info-gordura"));

    pacienteTr.appendChild(montaTd(paciente.imc,"info-imc"));

    return pacienteTr;

}

function montaTd (dado,classe) {

    var td = document.createElement ("td");

    td.textContent = dado;

    td.classList.add(classe);

    return td;

}

Atualmente as validações implementadas são feitas apenas com pacientes que já estavam na tabela. O ideal é reaproveitar o código de validação, para também validar os dados do formulário antes de adicionarmos o paciente na tabela. Caso os dados sejam inválidos, exibiremos uma mensagem informando qual deles contém um valor inapropriado.

A reutilização do código se mostra útil, por termos o objeto paciente montado e pronto para ser utilizado no validaPaciente(paciente). Ela será responsável por testar as propriedades com peso e altura para identificar se os valores são válidos. Começaremos criando um if:

function validaPaciente(paciente){

if(paciente.altura < 3.0 && paciente.altura >= 0){

return true;

}

}

Criamos a condição de que se a altura for menor que 3 e maior ou igual a 0, o valor será válido. Porém, esta validação dos dados já foi realizada em calcula-imc.js:

if(peso <= 0 || peso >= 1000){

console.log("Peso inválido!");

pesoEhValido = false;

tdImc.textContent = "Peso inválido!";

paciente.classList.add("paciente-invalido");

}

if(altura <= 0 || altura >= 3.00){

console.log("Altura inválida!");

alturaEhValida = false;

tdImc.textContent = "Altura inválida!";

paciente.classList.add("paciente-invalido");

}

If(alturaEhValida && pesoEhValido){

var imc = calculaImc(peso,altura);

tdImc.textContent = imc;

Queremos reaproveitar esta lógica para facilitar inclusive a manutenção do código. Para não corrermos o risco de rescrevermos um código já existente, extrairemos a lógica de verificação dos dados para uma função específica, validaPeso(). Ela receberá peso como parâmetro e retornará true se o valor estiver entre 0 e 1000. Se não estiver, o retorno será false:

function validaPeso(peso){

if (peso >= 0 && peso <= 1000) {

return true;

} else {

return false;

}

}

Do mesmo jeito, vamos criar a função

 validaAltura():

function validaAltura(altura) {

if (altura >= 0 && altura <= 3.0) {

return true;

} else {

return false;

}

}

Em vez de inicializarmos as variáveis pesoEhValido e alturaEhValida com true, chamaremos as funções de validação.

var pesoEhValido = validaPeso(peso);

var alturaEhValida = validaAltura(altura);

Teremos que fazer ajustes no if também. No entanto, como estamos testando se o peso **não** é válido, para identificar se ele é negativo ou maior que uma tonelada, então vamos utilizar o **operador de negação** (!) no calcula-imc.js. O ! será responsável por inverter os valores da condição avaliada:

Esse operador de negação faz com que automaticamente aquela codição return false ;

if (!pesoEhValido) {

console.log("Peso inválido!");

pesoEhValido = false;

tdImc.textContent = "Peso inválido!";

paciente.classList.add("paciente-invalido");

}

Em seguida, repetiremos o processo para a função validaAltura(altura), que receberá a altura e fará um teste semelhante dentro do if:

if (!alturaEhValida) {

console.log("Altura inválida!");

alturaEhValida = false;

tdImc.textContent = "Altura inválida!";

paciente.classList.add("paciente-invalido");

}

Agora, com essas duas funções criadas, poderemos chamá-las na função validaPaciente(paciente), em form.js. Vamos testar se o peso e a altura são válidos utilizando as funções que acabamos de criar. Por ora, testaremos apenas com o peso:

function validaPaciente(paciente) {

if (validaPeso(paciente.peso)) {

return true;

} else {

return false;

}

}

O próximo passo será adicionar a função validaPaciente(paciente) no momento do clique do botão após a montagem do objeto paciente. Incluiremos uma condição, na qual vamos imprimir uma mensagem de aviso no console se o valor do validaPaciente() for false.

var botaoAdicionar = document.querySelector("#adicionar-paciente");

botaoAdicionar.addEventListener("click", function(event) {

event.preventDefault();

var form = document.querySelector("#form-adiciona");

var paciente = obtemPacienteDoFormulario(form);

if (!validaPaciente(paciente)) {

console.log("Paciente inválido");

}

var pacienteTr = montaTr(paciente);

Além disso, caso o paciente tenha algum valor inválido, queremos que ele não seja adicionado na tabela. Então devemos sair da função caso a condição do if seja válida. Ou seja, se o código entrar no if, exibiremos a mensagem de paciente inválido no console e **retornaremos** a função, adicionando-se o return vazio:

var form = document.querySelector("#form-adiciona");

var paciente = obtemPacienteDoFormulario(form);

var pacienteTr = montaTr(paciente);

if (!validaPaciente(paciente)) {

console.log("Paciente inválido");

return;

}

# Exibindo mensagens de erro

Além de impedirmos que um paciente com dados inválidos seja adicionado na tabela, queremos mostrar uma mensagem para o usuário informando qual problema tivemos na validação.

Em vez de retornarmos verdadeiro ou falso (true ou false), poderemos **retornar uma mensagem de erro** caso a validação seja false e uma String vazia se a validação for true.

function validaPaciente(paciente) {

if (validaPeso(paciente.peso)) {

return "";

} else {

return "Peso é inválido";

}

}

No momento de chamarmos a função validaPaciente(paciente), extrairemos o retorno da função para a variável erro. Se o tamanho da String for maior que 0, significa que ocorreu algum erro.

var pacienteTr = montaTr(paciente);

var erro = validaPaciente(paciente);

if(erro.length > 0){

return;

}

Exibiremos a mensagem de erro em cima do form. Para isto, faremos ajustes no index.html, adicionando uma tag <span> com o id mensagem-erro.

<section class="container">

<h2 id="titulo-form">Adicionar novo paciente</h2>

<span id="mensagem-erro"></span>

De volta ao form.js, se ocorrer algum erro na validação, selecionaremos o span que possui o id #mensagem-erro, e depois alteraremos o conteúdo de texto.

var erro = validaPaciente(paciente);

if (erro.length > 0) {

var mensagemErro = document.querySelector("#mensagem-erro");

mensagemErro.textContent = erro;

return;

}

**Para validarmos a altura**

function validaPaciente(paciente) {

if (validaPeso(paciente.peso)) {

return "";

} else {

return "Peso é inválido";

}

if (validaAltura(paciente.altura)) {

return "";

} else {

return "Altura é inválida";

}

}

A função só retornará uma coisa de cada vez, e não duas strings concatenadas. Ou ela retorna que o peso é inválido, ou que a altura é inválida. Então, em vez de retornarmos uma única string, retornaremos um **array de strings**.

Após declararmos o array de erros com [], vamos fazer alterações na condição. Se o valor do peso tiver algum problema, pegaremos o array de erros e colocaremos algo dentro, usando o método push(). Faremos o mesmo para a altura. Ao final da função, retornamos o array:

No entanto, usaremos o operador de negação ! para deixar o código mais legível. Apenas no caso do peso **não** ser válido, o erro.push() será adicionado. O mesmo será feito para a altura, e assim poderemos remover o else.

function validaPaciente(paciente) {

var erros = [];

if (!validaPeso(paciente.peso)) {

erros.push("Peso é inválido");

}

if (!validaAltura(paciente.altura)) {

erros.push("Altura é inválida");

}

return erros;

}

Um detalhe é que, por estarmos trabalhando com o if simples, poderemos remover as chaves ({}) e colocar o código em uma única linha. Ele continuará sendo compreensível para o JavaScript.

function validaPaciente(paciente) {

var erros = [];

if (!validaPeso(paciente.peso)) erros.push("Peso é inválido");

if (!validaAltura(paciente.altura)) erros.push("Altura é inválida");

return erros;

}

Porém, o código estava adaptado para imprimir uma mensagem única. Nós gostaríamos que ele imprimisse uma lista de **erros**. Vamos ajustar o nome da variável em form.js:

var erros = validaPaciente(paciente);

if (erros.length > 0){

var mensagemErro = document.querySelector("#mensagem-erro");

mensagemErro.textContent = erros;

return;

Precisaremos adicionar uma string. Mas, por exemplo, se tivéssemos 10 erros, teríamos que adicionar a mesma quantidade de tags <span>s? Seria mais interessante termos uma **lista** no HTML e, para cada erro, adicionaríamos um item nessa lista. No index.html, vamos trocar o **<span>** por uma tag **<ul>** e, dentro, adicionar várias <li>s:

Se o nosso objetivo é adicionar diversas mensagens de erros, uma para cada erro que surgir para o array, teremos que acessar a função do form.js e não apenas alterar o conteúdo da <ul>. Será necessário criar uma <li> para cada erro e, depois, adicioná-la na <ul>.

Como esta lógica pode ficar complicada, iremos extrair a parte de exibição das mensagens de erro para a função exibeMensagensDeErro(). Ela será responsável por pegar o array de erros e, para cada item, criaremos uma <li> a ser adicionada na <ul> de mensagens-erro. Observe que faremos uma pequena alteração no id, em index.html:

<section class="container">

<h2 id="titulo-form">Adicionar novo paciente</h2>

<ul id="mensagens-erro">

</ul>

No arquivo form.js, se queremos exibir a mensagem de erro, antes teremos que criar a função exibeMensagensDeErro(erros), recebendo o array de mensagens de erros. Vamos adicioná-la logo acima da função obtemPacienteDoFormulario():

Para cada item do array, selecionaremos a ul, que será guardada em uma variável:

function exibeMensagensDeErro(erros){

var ul = document.querySelector("#mensagens-erro");

}

E para cada item do array, adicionaremos a tag <li>. Poderemos fazê-lo de diferentes formas, como usando o for tradicional:

function exibeMensagensDeErro(erros){

var ul = document.querySelector("#mensagens-erro");

for(var i = 0; i < erros.lenght ; i++){

var erro = erros[i];

}

}

Temos que falar que o texto da <li> será uma mensagem de erro, que está dentro do array de erros. Para termos acesso ao erro, recebemos o item de iteração dentro da function() e o textContent será o erro. Nós colocaremos li dentro ul, usando a função appendChild().

function exibeMensagensDeErro(erros){

var ul = document.querySelector("#mensagens-erro");

erros.forEach(function(erro){

var li = document.createElement("li");

li.textContent = erro;

ul.appendChild(li);

});

}

A função que exibe a mensagem de erro é:

var erros = validaPacientes(paciente);

console.log(erros);

if(erros.length > 0){

exibeMensagensDeErro(erros);

return;

}

Nosso código está todo organizado, isolamos as responsabilidades em diferentes funções, tornando simples a ação de adicionar uma nova validação. Atualmente, se não preenchermos o campo "Nome", o paciente será adicionado na tabela do mesmo jeito

Como validamos o peso e a altura, vale a pena validarmos o nome. Vamos acessar a função validaPaciente() e adicionar um novo if. Para verificar se um campo está em branco, podemos analisar o seu tamanho (length), se ele for igual 0, significa que o campo não foi preenchido. Por exemplo, o nome do paciente:

function validaPaciente(paciente) {

var erros = [];

if (paciente.nome.length == 0){

erros.push("O nome não pode ser em branco");

}

if (paciente.gordura.length == 0){

erros.push("A gordura não pode ser em branco");

}

if (paciente.peso.length == 0){

erros.push("O peso não pode ser em branco");

}

if (paciente.altura.length == 0){

erros.push("A altura não pode ser em branco");

}

if (!validaPeso(paciente.peso)){

erros.push("Peso é inválido");

}

if (!validaAltura(paciente.altura)){

erros.push("Altura é inválida");

}

return erros;

}

## Limpando as mensagens de erro

Ao tentarmos adicionar um paciente com algum dado inválido, a mensagem é impressa. Depois da correção e dos dados serem enviados, o paciente será adicionado à tabela, mas a mensagem continuará na página, em acúmulo.

Nós queremos limpar a lista de mensagens (ul), antes que as mensagens de erro sejam exibidas. Para esvaziar a ul, removeremos **todo o conteúdo HTML**. Para isto, utilizaremos a propriedade **innerHTML**, que nos permite controlar o HTML interno de um elemento. Passaremos uma string vazia para a propriedade:

function exibeMensagensDeErro(erros) {

var ul = document.querySelector("#mensagens-erro");

ul.innerHTML = "";

erros.forEach(function(erro) {

var li = document.createElement("li");

li.textContent = erro;

ul.appendChild(li);

});

}

Deste modo, sempre que as mensagens de erro forem exibidas, as anteriores serão apagadas, de forma a validarmos o formulário.

Faremos um pequeno ajuste para os casos em que adicionamos um paciente na tabela. Vamos limpar a ul no form.js. Na variável mensagensErro, usaremos document.querySelector(). Após a adição, vamos limpar as mensagens:

var tabela = document.querySelector("#tabela-pacientes");

tabela.appendChild(pacienteTr);

form.reset();

var mensagensErro = document.querySelector("#mensagens-erro");

mensagensErro.innerHTML = "";

});

Agora, quando adicionamos um paciente com sucesso, as mensagens de erro são limpas da tela.

Vimos nessa aula como validar um formulário, mostramos também a importância de organizarmos nosso código à medida em que ele vai crescendo. Demonstramos ser útil saber trabalhar com arrays, como fizemos no caso de exibirmos mais de uma mensagem de erro. É importante conhecermos as estruturas básicas do JavaScript - incluindo strings e arrays - e saber quando utilizar cada uma delas. Mostramos como fazer a validação com uma tag <ul> em vez de <span>. Nós estamos agrupando todo o conhecimento adquirido e seguiremos melhorando nossa aplicação!

DICAS: Iterando com o forEach , observe o código abaixo

var nomes = ["Douglas", "Flávio", "Nico", "Rômulo", "Leonardo"];

Em JavaScript, todo array possui a função **forEach**. Passamos para ela uma função por parâmetro, e nessa função fazemos o que quisermos **para cada item do array**. O item do array é recebido por parâmetro na função interna.

nomes.forEach(function(nome) {

console.log(nome + " é instrutor da Alura");

})

DICAS: A propriedade innerHTML:

Com a propriedade **innerHTML**, podemos editar obter o conteúdo HTML (HTML interno) de um elemento.

Para editar o HTML interno, como o **innerHTML** é uma **propriedade**, utilizamos um sinal de igual (**=**). Fazemos:

ObjetoDeUmElementoHTML.innerHTML = "Novo conteúdo"

E para obter o HTML interno, fazemos:

ObjetoDeUmElementoHTML.innerHTML

O seu retorno será todo o conteúdo HTML, tanto tags, atributos, classes, etc, no formato de uma String.!

**Removendo pacientes**

Agora que já estamos validando totalmente os nossos pacientes, podemos nos concentrar em adicionar novas funcionalidades ao site do Dr Marcos Maio.

Ela pode adicionar um paciente com algum dado errado, sem querer, faltando alguma letra no nome, ou algum número no peso, por exemplo. O paciente passará pela validação, mas com os dados incorretos. E ela não conseguirá editá-lo nem removê-lo, para adicioná-lo novamente.

Então vamos implementar a opção de remoção de pacientes para a Aparecida. Ela pode ser feita através de um ícone, ou para ser mais simples, um duplo clique! Ao clicarmos duas vezes em um paciente, o mesmo é removido.

Se queremos colocar um evento de duplo clique nas linhas, devemos escutar esse evento para cada linha da tabela, então vamos fazê-lo no JavaScript.

Como estamos criando uma funcionalidade nova, em nada relacionado com o formulário ou com o cálculo de IMC, vamos criar um novo arquivo na pasta **js**, o **remover-paciente.js**. É nesse arquivo que ficará toda a lógica de remoção.

Primeiramente, devemos selecionar as linhas. Todas elas possuem a classe **paciente**, portanto iremos selecioná-las:

Agora, para cada linha e paciente, faremos algo

E para cada paciente, adicionaremos um escutador de eventos para escutar o evento de duplo clique, o **dblclick**:

var pacientes = document.querySelectorAll(".paciente");

pacientes.forEach(function(paciente) {

paciente.addEventListener("dblclick", function() {

});

});

Agora falta sabermos como removeremos um elemento do DOM com JavaScript.

## Removendo um elemento do HTML com JavaScript

A função do JavaScript responsável por remover um elemento do HTML é a **remove()**. Então temos que chamá-la no paciente que foi clicado.

Quem será clicado? O dono do evento, certo? Este é que sofrerá a ação de duplo clique e executará a função. Para acessar o dono do evento, em que o evento está atrelado, utilizaremos uma palavra reservada do JavaScript chamada **this**:

var pacientes = document.querySelectorAll(".paciente");

pacientes.forEach(function(paciente) {

paciente.addEventListener("dblclick", function() {

this.remove();

});

});

O **this** é uma palavra de contexto ligada com quem acionou o evento, a quem o evento está atrelado. Como o evento está atrelado ao paciente, o **this** fará referência a ele.

Ao testarmos na página, e darmos um duplo clique em um paciente nativo, o mesmo é removido! Então conseguimos implementar a remoção de um paciente de um jeito bem fácil.

Mas nem tudo é perfeito, e essa implementação tem um pequeno defeito: ao adicionarmos um paciente pelo formulário, não conseguimos removê-lo. Isso porque o novo paciente não estava na página no momento em que o **remover-paciente.js** foi carregado (lembrando que o navegador abre a página e vai lendo o seu HTML, carregando os nossos scripts logo depois).

Então, o script seleciona somente os pacientes que já estão na página. Ao adicionar um ou mais pacientes, eles não estão escutando o evento! No momento em que um paciente for adicionado, deveríamos colocá-lo para ouvir o evento e executar o mesmo código, mas aí estaríamos duplicando código, que já sabemos não ser uma boa prática.

Dicas: O nome do evento correto é dblclick ! Além dos eventos que já vimos aqui no curso, existem muitos outros tipos, que variam desde aonde o seu usuário interage com a página até quando ele altera o tamanho do navegador!

# Delegando eventos

Para resolver o problema da remoção de pacientes, vamos nos aproveitar de uma característica do JavaScript chamada **Event Bubbling**, ou "borbulhamento" de eventos. Quando escutamos um evento no JavaScript, ele na verdade não acontece só no dono do evento (no nosso caso, na linha do paciente), ele acontece também no pai do paciente, no pai do pai do paciente, e assim vai subindo.

Na nossa estrutura, ao darmos um duplo clique na <tr> do paciente, o pai (<tbody>) também escuta o evento, assim com a tag <table>, até chegar no <body>.

## Descobrindo o elemento clicado

Para não termos que ficar escutando o evento a cada linha da tabela, e para as linhas adicionadas por meio do formulário, iremos delegar a responsabilidade de escutar os eventos para o pai de todos, no caso, a tag <table>. Faremos isso com addEventListenter() no remover-paciente.js:

Dentro da função, perguntamos ao pai qual filho recebeu o clique, pois é ele que será removido. Desta vez não podemos utilizar o this, já que o dono do evento é a tabela, logo, ela acabará sendo removida. Para descobrirmos qual filho foi clicado, utilizaremos o event como parâmetro na função: Ele irá nos dizer quem foi clicado, quem foi o alvo, pela propriedade target. Enquanto o this se refere ao dono do evento, o event.target será quem sofreu propriamente o evento. O próximo passo será remover o elemento após o duplo clique:

var tabela = document.querySelector("table");

tabela.addEventListener("dblclick", function(event) {

event.target.remove();

});

Mas se testarmos, veremos que isso **quase** funciona. Ao clicarmos em algum campo da tabela, apenas o <td> clicado será removido.

No entanto, queremos remover a linha completa, ou seja, a tag <tr>, pai do <td>.Para selecionarmos o pai de um elemento, trabalharemos com a propriedade parentNode. A seguir selecionaremos quem foi clicado e removeremos o seu pai, uma tag <tr>:

var tabela = document.querySelector("table");

tabela.addEventListener("dblclick", function(event) {

event.target.parentNode.remove();

});

Poderemos até separar em variáveis para deixar mais explícito:

var tabela = document.querySelector("table");

tabela.addEventListener("dblclick", function(event) {

var alvoEvento = event.target;

var paiDoAlvo = alvoEvento.parentNode; // TR = paciente = remover

paiDoAlvo.remove();

});

Assim, mesmo que adicionarmos novos pacientes, no momento em que ele receber um duplo clique, o evento irá subir até chegar à tabela. Esta por sua vez, estará escutando. Desta forma, o paciente será removido.

Quando clicamos em qualquer filho, o evento consegue chegar até o pai (table). Essa estratégia é muito boa por economizarmos código, deixando-o mais sucinto. E é possível simplificarmos ainda mais o código:

DICAS: Todo evento disparado em JavaScript possui um contexto que é acessível através da função executada quando o evento for disparado.Na função, acessamos o contexto através do objeto implícito this. Ele é uma referência para o elemento do DOM que esta recebendo o evento, logo, sua natureza é dinâmica, ou seja, se clicarmos no primeiro elemento da lista o this será o primeiro elemento, se clicarmos no último ele será o último. É a natureza dinâmica do this que nos permite utilizar a mesma função em diferentes contextos.

**Animando a remoção do paciente**

o removermos um paciente, a linha some instantaneamente, abruptamente. Para melhorarmos a experiência do usuário, seria interessante que a linha fosse sumindo gradualmente, até desaparecer totalmente da tabela.

Sabemos que, se queremos incluir um efeito visual, devemos fazer as alterações no arquivo CSS. Em index.css, adicionaremos a classe fadeOut, que definirá a opacidade do elemento com 0, fazendo-o sumir:

.fadeOut {

opacity: 0;

transition: 0.5s;

}

Comentaremos a linha que remove o elemento com a função remove em remover-paciente.js, e depois adicionaremos a classe fadeOut ao elemento:

var tabela = document.querySelector("table");

tabela.addEventListener("dblclick", function(event) {

event.target.parentNode.classList.add("fadeOut");

//event.target.parentNode.remove();

});

No entanto isso e apenas uma parte visual , ou seja o intem apenas desaparecera por causa da opacidade , porém continua na tabela.

Como o computador é muito rápido, o browser processa as linhas do script instantaneamente. A classe é adicionada e logo em seguida o elemento é removido, por isso não conseguimos ver a transição acontecendo. Porém, devemos remover a linha da tabela só após o término da transição. Devemos pedir para o JavaScript aguardar meio segundo (tempo que colocamos para a transição ocorrer) para remover a linha.

Quando queremos aguardar um tempo, devemos usar a função setTimeout. Será passada como parâmetro uma função anônima com quanto tempo deve ser aguardado:

var tabela = document.querySelector("#tabela-pacientes");

tabela.addEventListener("dblclick", function(event) {

event.target.parentNode.classList.add("fadeOut");

setTimeout(function() {

event.target.parentNode.remove();

}, 500);

});

Com a função acima iremos assegurar a execução da função passada por parâmetro para ela, com o tempo de 500 milissegundos, equivalente a meio segundo.

**Filtrando os pacientes da tabela**

Seria interessante se disponibilizássemos um filtro, com o qual a busca fosse realizada usando-se algum critério. Por exemplo, se quiséssemos acessar os dados de um determinado paciente, bastaria digitarmos seu nome em um campo de pesquisa e, então, apenas os dados do mesmo seriam listados. A tendência de uma tabela de dados é que a lista aumente cada vez mais, e caso se queira utilizar algum tipo de filtro, queremos que ela possa fazer uma busca mais específica.

Começaremos implementando um campo de busca na tabela, adicionando uma tag <input> no index.html. Criaremos uma label com o texto Filtre:. Dentro da tag, teremos também placeholder, com o texto Digite o nome do paciente.

## Acessando o conteúdo digitado no campo de filtragem

Resta agora implementarmos o código JavaScript que fará essa filtragem.

Nosso objetivo é que conforme o nome for digitado no campo, os pacientes relacionados à busca sejam listados. Por exemplo, ao digitarmos **Doug**, o paciente **Douglas** deverá ser exibido. O conteúdo de texto que está sendo digitado no campo de busca deverá ser pego.

Só que quando queremos filtrar? Conforme o usuário vai digitando, a cada letra digitada. Ou seja, queremos detectar o evento de **digitar**, conhecido como **input de dados**. Toda vez que alguém inserir dados no campo, esse evento será disparado.

Queremos que ao começarmos a digitar no filtro, seja iniciado o processo de filtragem, e para isto, adicionaremos um escutador de evento. Com o addEventListener() escutaremos um evento de input. Passaremos como segundo parâmetro uma função com a ação a ser executada quando alguém clicar no campo. A cada letra que inserimos, a função é chamada, o value do campo é filtrado na tabela. Poderemos utilizar campoFiltro.value dentro da função, no entanto, campoFiltro é o **dono** do evento. Utilizaremos a palavra de contexto this, referente ao próprio dono do evento.

var campoFiltro = document.querySelector("#filtrar-tabela");

campoFiltro.addEventListener("input", function(){

console.log(this.value);

});

Conforme o nome é digitado, podemos ir comparando com os de todos os pacientes da tabela com o querySelectAll(). Exibiremos aqueles que forem iguais, e os que forem diferentes serão escondidos. Vamos, então, selecionar todos os pacientes:

var campoFiltro = document.querySelector("#filtrar-tabela");

campoFiltro.addEventListener("input", function() {

console.log(this.value);

var pacientes = document.querySelectorAll(".paciente");

});

No entanto, queremos fazer a comparação com o nome dos pacientes, não com a tr. Precisaremos iterar sobre os pacientes, para então acessarmos o nome de cada um

Agora, selecionaremos o paciente, e a partir dele a td com a classe info-nome. Daí, extrairemos o nome do paciente

var campoFiltro = document.querySelector("#filtrar-tabela");

campoFiltro.addEventListener("input", function() {

var pacientes = document.querySelectorAll(".paciente");

for (var i = 0; i < pacientes.length; i++) {

var paciente = pacientes[i];

var tdNome = paciente.querySelector(".info-nome");

var nome = tdNome.textContent;

}

});

A partir de então temos acesso ao nome de todos os pacientes da tabela, e também ao conteúdo de texto do campo de filtragem, só precisaremos esconder todos os pacientes que são diferentes do conteúdo de texto, e mostrar os que são iguais.

# Implementando a lógica de filtragem

Nosso próximo objetivo será esconder todos elementos diferentes do valor do campo de texto (this.value) e mostrar os que forem iguais para remover o paciente, portanto adicionaremos uma classe e, em index.css, incluiremos a classe .invisivel:

.invisivel{

display: none;

}

Faremos o mesmo para esconder os pacientes, se (if) o nome for diferente do conteúdo de texto, adicionaremos uma classe. Caso não seja, isto é, se o nome for igual ao que foi digitado, removemos a classe. Faremos isso no filtra.js:

if (nome != this.value) {

paciente.classList.add("invisivel");

} else {

paciente.classList.remove("invisivel");

}

}

});

Se testarmos no browser digitando **Douglas**, somente este paciente será exibido.

Porém, ao apagarmos o que foi digitado, a tabela com todos os pacientes não voltam a ser exibidos. Isso porque todos os pacientes estão com a classe invisivel, mas só deveríamos colocá-la quando houvesse algum conteúdo digitado no campo...

Criaremos um if para que a classe invisivel seja adicionada apenas quando houver algo digitado. Veremos se há algo digitado ou não por meio do seu length - se ele for 0, significa que o campo está em branco, e se for maior que 0, significará que há algo digitado:

Com algo digitado, queremos que o for seja executado:

No caso de não haver nada digitado, percorreremos todos os pacientes e removeremos a classe com um segundo for:

var campoFiltro = document.querySelector("#filtrar-tabela");

campoFiltro.addEventListener("input", function(){

console.log(this.value);

var pacientes = document.querySelectorAll(".paciente");

if (this.value.length > 0){

for (var i = 0; i < pacientes.length; i++){

var paciente = pacientes[i];

var tdNome = paciente.querySelector(".info-nome");

var nome = tdNome.textContent;

if (nome != this.value){

paciente.classList.add("invisivel");

} else {

paciente.classList.remove("invisivel");

}

}

} else {

for (var i = 0; i < pacientes.length; i++) {

var paciente = pacientes[i];

paciente.classList.remove("invisivel");

}

}

});

Em seguida, realizaremos um novo experimento com o filtro da tabela. Ao digitarmos uma letra, a lista será filtrada. Mas ao apagarmos a letra do campo de busca, a listagem será liberada.

**Filtrando com expressão regular**

 Para tornar a nossa filtragem melhor, seria interessante que, ao digitarmos uma letra no campo de busca, todos os nomes com essa letra fossem listados. Por exemplo, ao digitarmos a letra "P", todos os pacientes que começassem com essa letra seriam exibidos. Como "Pedro", "Paulo" e "Pablo".

Conforme formos digitando as letras, a filtragem seria atualizada e a busca ficaria mais interativa.

Para que isso aconteça, é preciso compararmos letra por letra pelos termos buscados com os nomes cadastrados na lista. Seria trabalhoso fazer isso manualmente, mas o JavaScript, além de outras linguagens de programação, já possuem uma solução para a busca de texto: **expressões regulares**.

As **expressões regulares** são um tipo especial de texto, que nos auxilia a buscarmos por strings, facilitando quando o termo for maior. Pode ser uma busca simples, como no nosso caso, em que queremos identificar quais nomes contêm determinadas letras; ou casos complexos, se queremos pesquisar se o parágrafo contém a palavra "nome", por exemplo, é como quando os editores de texto buscam por uma palavra usando o comando "CTRL + F".

## Criando expressões regulares no JavaScript

É bastante simples criar expressões regulares. Criaremos uma variável, que no caso chamaremos expressao, e em seguida colocaremos uma expressão regular dentro dela. Vamos gerar um objeto especial do JS, adicionando new e o nome RegExp():

var expressao = new RegExp();

Nós poderemos passar dois parâmetros para o objeto, sendo o primeiro o texto que queremos buscar, no caso, o termo digitado no campo de busca (this.value), e o segundo parâmetro será referente às características dos termos que devem ser buscados. É importante que a busca não seja case sensitive, que é a diferenciação entre letras maiúsculas e minúsculas. Devem ser buscadas tanto letras maiúsculas como minúsculas, e passaremos a letra "i" como segundo parâmetro, para indicarmos a opção **case insensitive**:

Porém, como utilizamos a expressão regular para buscar um texto específico na tabela? Em vez de compararmos com o nome inteiro do paciente (como estávamos fazendo), vamos pedir para a expressão regular verificar se um pedaço do nome do paciente possui as letras digitadas no campo de busca. Para isso, a expressão regular tem o método test(), com a qual passaremos o que queremos testar:

Esse teste irá retornar **verdadeiro** caso o nome contenha a expressão, ou **falso** caso não contenha. Como estamos testando se o nome não contém a expressão (por isso adicionaremos a classe invisivel), utilizaremos novamente o **operador de negação** (!). Logo, se o teste falhar, adicionaremos a classe; se não, ela será removida:

RESTO DO CÓDIGO ACIMA//

var expressao = new RegExp(this.value, "i");

if (this.value.length > 0) {

for (var i = 0; i < pacientes.length; i++) {

var paciente = pacientes[i];

var tdNome = paciente.querySelector(".info-nome");

var nome = tdNome.textContent;

var expressao = new RegExp(this.value, "i");

if (!expressao.test(nome)) {

paciente.classList.add("invisivel");

} else {

paciente.classList.remove("invisivel");

}

}

} else {

for (var i = 0; i < pacientes.length; i++) {

var paciente = pacientes[i];

paciente.classList.remove("invisivel");

}

}

});

Dicas sobre como criar um **Regex** : O primeiro parâmetro que devemos passar para o construtor é o padrão (o texto da expressão regular, o que deve ser buscado) e o segundo parâmetro são uma ou mais flags (representando como queremos que a expressão regular busque). Por exemplo, podemos definir que não queremos que haja distinção entre letras maiúsculas e minúsculas, através da flag **i**.

**Filtrando sem regex?**

Aprendemos como filtrar, comparar letra a letra, utilizando expressões regulares, representado pelo fragmento de código abaixo:

var expressao = new RegExp(this.value,"i");

if (!expressao.test(nome)) {

paciente.classList.add("invisivel");

} else{

paciente.classList.remove("invisivel

Mas há um modo de fazer essa comparação sem a necessidade de utilizar expressões regulares! Podemos utilizar a função **substring**, que recebe dois parâmetros, fazendo com que ela devolva parte da string, com o tamanho definido nos parâmetros. O primeiro parâmetro é o início, começando do 0 (que representa o primeiro caractere). O segundo parâmetro define o fim (exclusivo, mostramos até o penúltimo caractere). Por exemplo:

var string = "Alura";

var resultado = string.substring(1, 4);

Extraímos uma string que começa no segundo caractere (número 1) e termina no quarto caractere (número 3, índice anterior ao número 4). O valor da variável **resultado** é: lur

Agora vamos utilizar esse coceito em nosso código do filtrar tabela:

Como o primeiro parâmetro é o inicio, e queremos comparar desde o início da string **nome**, vamos utilizar como início o valor **0**, ou seja, sempre a partir do primeiro caractere. Mas qual é o fim? O fim é justamente o tamanho do valor digitado:

nome.substr(0, this.value.length);

Podemos guardar essa string em uma variável, e compará-la com o que está sendo digitado. Caso seja falso, adicionamos a classe **invisivel**, se não for, removemos-a:

var comparavel = nome.substr(0, this.value.length);

if (!(this.value == comparavel)) {

paciente.classList.add("invisivel");

} else{

paciente.classList.remove("invisivel");

}

Mas e a distinção entre letras maiúsculas e minúsculas? Nesse caso não temos distinção entre letras maiúsculas e minúsculas, mas para contornar isso, antes de compará-las, podemos colocar as duas strings em letras minúsculas, para efetuar a comparação entre elas em seguida:

var comparavel = nome.substr(0, this.value.length);

var comparavelMinusculo = comparavel.toLowerCase();

var valorDigitadoMinusculo = this.value.toLowerCase();

if (!(valorDigitadoMinusculo == comparavelMinusculo)) {

paciente.classList.add("invisivel");

} else{

paciente.classList.remove("invisivel");

}

Esta é uma alternativa de implementar a mesma funcionalidade sem expressão regular, porém temos que escrever mais e nos preocupar com mais detalhes! Fica ai esta opção para você guardar nos seus conhecimentos.

**Introdução ao AJAX**

a última feature que vamos desenvolver é a capacidade de integração do site a um sistema externo, que também contém pacientes e está em outro computador. O Dr Marcos Maio quer trazer esses pacientes para que eles fiquem cadastrados no mesmo sistema.

O sistema externo de pacientes pode ser acessado no navegador pelo endereço [api-pacientes.herokuapp.com/pacientes](https://api-pacientes.herokuapp.com/pacientes). Nele, há uma lista de pacientes que devem ser integrados ao sistema do Marcos .

Nós queremos que esses pacientes sejam importados com o clique de um botão para a tabela do site de modo que, quando o usuário clicar, os dados dos pacientes serão pesquisados no sistema externo e depois adicionados ao site do Marcos.

A primeira coisa que devemos fazer é adicionar o botão na página index.html. É nele que clicaremos para buscar os pacientes externos. Adicionaremos o botão abaixo do fechamento da tag <table>:

Precisaremos criar uma nova feature, e para isto vamos gerar um novo arquivo JavaScript na pasta js, que receberá o nome de buscar-pacientes.js. Importaremos este novo arquivo à página index.html:

Se queremos procurar os pacientes ao clicarmos no botão de "Buscar Paciente", devemos selecioná-lo e atrelá-lo ao evento de click. Começaremos adicionando a variável botaoAdicionar, no arquivo buscar-pacientes.js:

var botaoAdicionar = document.querySelector("#buscar-pacientes");

botaoAdicionar.addEventListener("click", function(){

console.log("Buscando pacientes");

});

Queremos replicar o passo que fizemos com o navegador, em que sabemos como acessar um endereço que pode ser aberto em uma nova aba, digitá-lo e pressionar na tecla "Enter". Ou seja, o navegador é o responsável pela requisição, indo até a URL e nos mostrando os dados. Mas como faremos isso dentro da nossa página?

Veremos como fazer a requisição com JavaScript de modo que ela vá até a URL, busque e retorne os dados, sem perder os que já estão no site da Aparecida Nutricionista. Desta forma, não dependeremos do navegador, e a nossa página vai recarregar.

No código JavaScript, devemos acessar o endereço [api-pacientes.herokuapp.com/pacientes](https://api-pacientes.herokuapp.com/pacientes), buscar e trazer os seus dados e colocá-los na tabela. Esse endereço é uma API, uma interface de programação que disponibiliza os dados para o usuário.

Não sabemos de onde vêm os dados, como e de que maneira eles foram gerados, pois eles se encontram disponibilizados e prontos para uso, sendo esta uma característica de uma API.

Ao analisarmos os dados, eles possuem uma estrutura que já conhecemos: cada paciente fica entre chaves, dentro das quais há itens formados por um par contendo chave e valor.

Além disso, todos os dados estão armazenados entre colchetes ([]), característica de um array do JavaScript. Como os dados estão disponibilizados em uma estrutura que conhecemos, será bem fácil trazê-los para dentro do código JavaScript, mas teremos que encontrar uma forma de acessá-los. Ele está com uma notação bastante parecida com o JavaScript, para realizarmos a requisição sem o navegador, somente com a linguagem.

Para fazermos essa requisição, temos um objeto bastante conhecido no JS, o XMLHttpRequest.

O XMLHttpRequest é um objeto do JS responsável por fazer requisições HTTP. O trecho XML do nome indica que ele era utilizado anteriormente para realizar o transporte de dados do tipo XML, no entanto, atualmente ele consegue trafegar outros tipos de dados, como textos.

Para que ele realize as requisições, devemos ensiná-lo e configurá-lo do jeito que queremos. Por exemplo, informaremos que uma requisição será feita para o seguinte endereço: https://api-pacientes.herokuapp.com/pacientes, com alguns de seus métodos.

O primeiro será open(), com o qual especificaremos o tipo de requisição a ser feita, no caso, GET. Também indicaremos para onde queremos fazê-la:

var botaoAdicionar = document.querySelector("#buscar-pacientes");

botaoAdicionar.addEventListener("click", function(){

console.log("Buscando pacientes...");

var xhr = new XMLHttpRequest();

xhr.open("GET", "https://api-pacientes.herokuapp.com/pacientes");

});

Essa ação será equivalente a chegarmos no navegador no momento em que ainda não enviamos a requisição, apenas verificando se o endereço está correto, se existe e está fazendo as configurações da requisição. Para que ela seja realizada, precisaremos chamar o método send():

xhr.send();

Podemos testar o nosso botão somente com esse código! Porém, ao clicarmos no botão, nada acontecerá. Por quê? Com o código atual, é como se o JavaScript estivesse abrindo uma nova aba no navegador, em que digitamos o endereço e clicamos em "Enter". Ficou faltando a parte final, de exibição dos dados para o usuário.

## Obtendo e exibindo a resposta da requisição

Para os dados serem exibidos, após o envio da requisição, devemos escutar um evento específico que é acionado quando a requisição termina e a sua resposta é carregada. Ao escutarmos o evento, carregaremos a resposta da requisição - que no caso, serão nossos dados. Esse evento é o load, característico do XMLHttpRequest:

E para acessarmos os dados da resposta, usaremos a propriedade responseText do XMLHttpRequest. Para testarmos, podemos guardá-la em uma variável, e depois imprimi-la no console do navegador:

var botaoAdicionar = document.querySelector("#buscar-pacientes");

botaoAdicionar.addEventListener("click", function() {

var xhr = new XMLHttpRequest();

xhr.open("GET", "https://api-pacientes.herokuapp.com/pacientes");

xhr.addEventListener("load", function() {

console.log(xhr.responseText);

});

xhr.send();

});

Agora, o xhr.responseText será exibido no console:

DICAS: O objeto XMLHttpRequest é quem é responsável por fazer requisições HTTP assíncronas com Javascript. Apesar de ter o XML no nome hoje em dia este objeto pode trafegar diversos outros tipos de dados além do XML, este nome só se manteve devido a um legado histórico.

E para instanciar um novo Objeto XMLHttpRequest devemos utilizar a sintaxe com a palavrinha **new** :

var xhr = new XMLHttpRequest();

Dado que temos um XMLHttpRequest, precisamos configurar nossa requisição para dizer para qual servidor queremos enviá-la e também qual método HTTP devemos usar.

Fazemos isto através do método .open() , passando o método e a url :

**Capturando erros de requisição**

Estamos conseguindo trazer os pacientes e adicioná-los à tabela com um simples clique no botão. Mas sabemos que, na web, nem todas as requisições dão certo.

Muitas vezes, ao acessarmos um site, podemos receber um erro 404. Por exemplo, ao acessarmos a plataforma da Alura por meio de uma URL inexistente, seremos notificados sobre o tal erro. E este não é o único erro possível, também existem os erros 201, 402, 503, entre outros, que podem ocorrer ao fazermos uma requisição AJAX na web.

O ideal seria que em casos em que fizéssemos uma requisição com o JavaScript, fosse possível detectar algum erro e avisar isso ao usuário.

Criaremos um if, com o qual testaremos o xhr.status e, neste caso, carregaremos os dados da página. Em caso de erro, cairemos no else e exibiremos o erro no console. Mostraremos tanto o xhr.status como xhr.responseText.

No browser, a requisição continuará funcionando. Vamos testar o que aconteceria caso utilizássemos uma URL inexistente, como https://api-pacientes.herokuapp.com/paci1111entes.

xhr.addEventListener("load", function() {

if (xhr.status == 200) {

var resposta = xhr.responseText;

var pacientes = JSON.parse(resposta);

pacientes.forEach(function(paciente) {

adicionaPacienteNaTabela(paciente);

});

} else {

console.log(xhr.status);

console.log(xhr.responseText);

}

});

xhr.send();

Com isso, o erro e a mensagem de resposta serão exibidos no console como gostaríamos, mas seria interessante deixar mais claro ao usuário que um erro ocorreu.

Ele mostra o status 404 e nos informa que a URL não foi encontrada. Ele imprimiu Cannot GET /paciente1111entes, ou seja, ele não conseguiu pegar a referida URL. Esta é uma forma para que as ações do código só sejam executadas quando as requisições funcionarem, caso contrário, avisaremos no console, ou alertaremos o usuário de algum outro modo, como por exemplo, exibir uma mensagem de erro.

Para isto, adicionaremos um span abaixo da tag <table> no arquivo index.html:

Incluímos no span a classe invisivel, e por padrão ele não aparecerá na página. E em buscar-pacientes.js criaremos a variável erro-ajax, com a qual selecionaremos o span. Em caso de erro, removeremos a classe invisivel, deixando a mensagem visível para o usuário mesmo para aqueles que estiverem com o console fechado:

Se alterarmos a URL para uma que não exista, a mensagem "Erro ao buscar os pacientes" é apresentada ao usuário.

No caso da requisição ser bem sucedida e a requisição ocorrer normalmente, nós ocultaremos a mensagem novamente, adicionando a classe invisivel e movendo a variável erroAjax para cima do if:

xhr.addEventListener("load", function() {

var erroAjax = document.querySelector("#erro-ajax");

if (xhr.status == 200) {

erroAjax.classList.add("invisivel");

var resposta = xhr.responseText;

var pacientes = JSON.parse(resposta);

pacientes.forEach(function(paciente) {

adicionaPacienteNaTabela(paciente);

});

} else {

erroAjax.classList.remove("invisivel");

}

});

xhr.send();

});

Ao testarmos, verificaremos que tudo ocorre conforme esperado. No caso de erro, visualizaremos a mensagem e, se ajustarmos o erro da URL, deixaremos de ver a mensagem de erro.

DICAS: O XMLHttpRequest é objeto responsável por fazer requisições HTTP com o Javascript.

Olhando as afirmativas, sabemos sim que ele deve ser configurado anteriormente com a função .open(), que ele pode trafegar diversos tipos de dados e não somente o XML ( ele tem esse nome graças a um vestígio histórico), e a requisição só é enviada mesmo após chamarmos o método .send().