**01 Quebrando o código em arquivos diferentes**

## Transcrição

Nossa aplicação JavaScript começou a crescer bastante, nós conseguimos calcular o IMC dos pacientes da tabela, além de ser possível adicionarmos um novo cadastro após preenchermos um formulário. Ficou faltando calcular o IMC dos novos pacientes cadastrados!

À medida em que formos adicionando mais funcionalidades, o arquivo JS cresceu - mesmo que com apenas duas. Com isso, vai se tornando mais difícil sua manutenção. O nosso código está funcional, porém não muito organizado. O código está agrupado em um grande arquivo com muitas responsabilidades e isso dificulta a adição de novas funcionalidades. Imagine se formos calcular o IMC do paciente recém adicionado...

Teremos que aumentar ainda mais o código; uma boa prática para quem está começando a desenvolver é sempre organizá-lo, dividindo o código por funcionalidades. Uma maneira de fazê-lo é criar novos arquivos JS e separar o trecho responsável pelo cálculo IMC do código relacionado ao <form>.

O próximo passo será criar o arquivo form.js na pasta js. Ele será responsável por cuidar de todos os arquivos que tenham a ver com o formulário. Moveremos para lá todo o trecho responsável por adicionar um paciente na tabela. A seguir, moveremos o código:

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

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

event.preventDefault();

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

var nome = form.nome.value;

var peso = form.peso.value;

var altura = form.altura.value;

var gordura = form.gordura.value;

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 = nome;

pesoTd.textContent = peso;

alturaTd.textContent = altura;

gorduraTd.textContent = gordura;

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd);

pacienteTr.appendChild(alturaTd);

pacienteTr.appendChild(gorduraTd);

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

tabela.appendChild(pacienteTr);

});

Agora os arquivos ficaram menores e simplificamos a manutenção do código.

Para deixar os arquivos mais semânticos e organizados, vamos alterar o nome do arquivo principal.js, e deixar claro qual a sua responsabilidade. No caso, iremos renomeá-lo como calcula-imc.js.

Em seguida, no index.html, vamos alterar o src da tag <script>, abaixo do fechamento da tag <section>:

<script src="js/calcula-imc.js" ></script>

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

Separamos os arquivos, deixando cada uma com uma funcionalidade, e o código ficará melhor organizado. Mais adiante, continuaremos fazendo outras melhorias na aplicação.

# 02 Agulha no palheiro!

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.

### 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 mais complicado 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.

# 03 Criando funções

## Transcrição

Outra boa prática que poderemos implementar no código é separar os blocos de código em blocos menores, em **funções**. Mostramos como um bloco grande de código é responsável por fazer muitas coisas.

No arquivo calcula-imc.js, temos um for responsável por pegar os dados dos pacientes, fazer a validação do peso e da altura e, depois, calcular o IMC e inserir os dados na tabela. Ou seja, temos três funcionalidades dentro do mesmo bloco.

Essas funcionalidades, como o cálculo do IMC, são necessárias em outros pontos do nosso código no **form.js**. 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:

nomeTd.textContent = nome;

pesoTd.textContent = peso;

alturaTd.textContent = altura;

gorduraTd.textContent = gordura;

imcTd.textContent = calculaImc(peso, altura);

A função também poderá ser chamada em calcula-imc.js, reaproveitando assim a lógica de cálculo do IMC. Vamos criá-la em seguida, e ela já retornará o IMC com duas casas decimais:

function calculaImc(peso, altura) {

var imc = 0;

imc = peso / (altura \* altura);

return imc;

}

Depois, vamos chamá-la também dentro da validação do pesoe altura que antecede a função calculaImc():

if ( alturaEhValida && pesoEhValido) {

var imc = calculaImc(peso, altura);

tdImc.textContent = imc.toFixed(2);

}

function calculaImc(peso, altura){

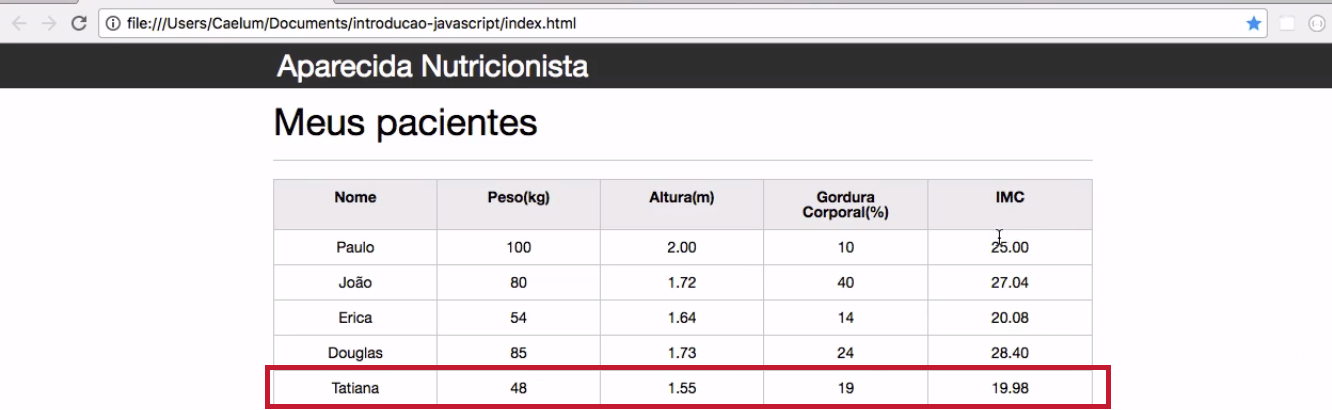
var imc = 0;

imc = peso / (altura \* altura);

return imc;

}

No browser, veremos que o cálculo será realizado e o resultado será exibido na tabela.



Mas faremos um pequeno ajuste no código: em vez de passarmos toFixed() em if, iremos movê-lo para o retorno da função calculaImc(). Com as alterações, o nosso código fica assim:

if ( alturaEhValida && pesoEhValido) {

var imc = calculaImc(peso, altura);

tdImc.textContent = imc;

}

}

function calculaImc(peso, altura){

var imc = 0;

imc = peso / (altura \* altura);

return imc.toFixed(2);

}

A vantagem de termos uma função separada é que conseguimos executar o calculaImc() no arquivo form.js:

nomeTd.textContent = nome;

pesoTd.textContent = peso;

alturaTd.textContent = altura;

gorduraTd.textContent = gordura;

imcTd.textContent = calculaImc(peso,altura);

Pode-se afirmar que o conteúdo de texto do imcTd será o retorno da função calculaImc(). Neste caso, também teremos peso, altura, e queremos calcular o IMC. Em seguida, a variável pacienteTr também receberá o imcTd quando formos adicioná-la na tr.

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd);

pacienteTr.appendChild(alturaTd);

pacienteTr.appendChild(gorduraTd);

pacienteTr.appendChild(imcTd);

Agora, ao adicionarmos um paciente na tabela pelo formulário, uma nova linha será inserida, com o valor do IMC calculado e exibido na tabela. Para exemplificar, vamos experimentar cadastrar os dados do paciente Alvaro.

Com isto, conseguiremos extrair a responsabilidade de se calcular o IMC, que estava presa no calcula-imc.js, fazendo o reuso desse código. Logo, é uma boa prática extrairmos um código que queremos reusar em outros locais, para funções externas.



Nós conseguimos reutilizar o código para se calcular o IMC e, após movermos a função, separamos a responsabilidade de se fazer este cálculo. Uma boa prática de programação é extrairmos ações que queremos reutilizar - como o cálculo do IMC - para funções externas. Fica mais essa dica de melhorias do código!

# 04 Criando um objeto paciente

## Transcrição

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. Vamos comentar o que cada trecho faz:

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

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

event.preventDefault();

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

//Extraindo informacoes do paciente do form

var nome = form.nome.value;

var peso = form.peso.value;

var altura = form.altura.value;

var gordura = form.gordura.value;

Mais abaixo teremos o trecho responsável por criar as tags tr e td:

//cria a tr e a td do 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 = nome;

pesoTd.textContent = peso;

alturaTd.textContent = altura;

gorduraTd.textContent = gordura;

imcTd.textContent = calculaImc(peso,altura);

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd);

pacienteTr.appendChild(alturaTd);

pacienteTr.appendChild(gorduraTd);

pacienteTr.appendChild(imcTd);

Em seguida, temos o trecho responsável por adicionar o paciente.

//adicionando o paciente na tabela.

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

tabela.appendChild(pacienteTr);

Temos pelo menos três funções que trabalharão com tarefas menores.

## 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:

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

// Extraindo informacoes do paciente do form

obtemPacienteDoFormulario(form)

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** ({}):

function obtemPacienteDoFormulario(form) {

var paciente = {

}

var nome = form.nome.value;

var peso = form.peso.value;

var altura = form.altura.value;

var gordura = form.gordura.value;

}

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;

}

Assim, atribuímos às propriedades os valores extraídos do formulário e, no fim, a função retornará o objeto paciente.

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

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

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

event.preventDefault();

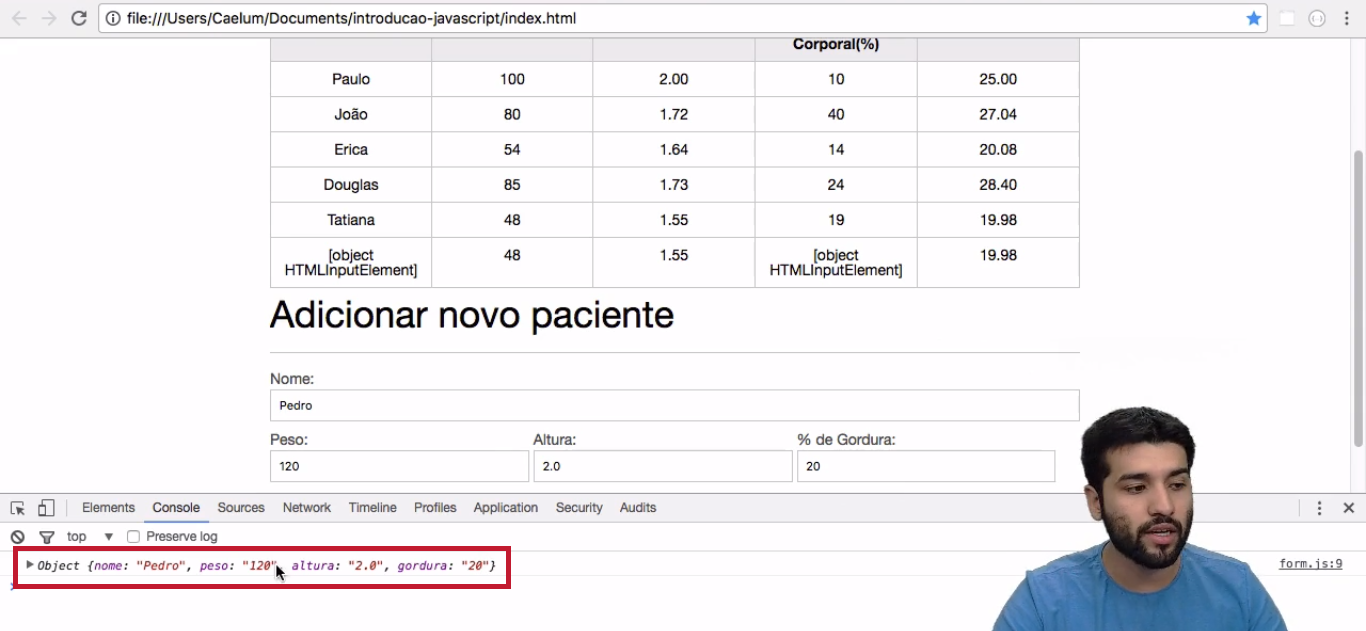
var form = document.querySelector("#form-adiciona");

//Extraindo informacoes do paciente do form

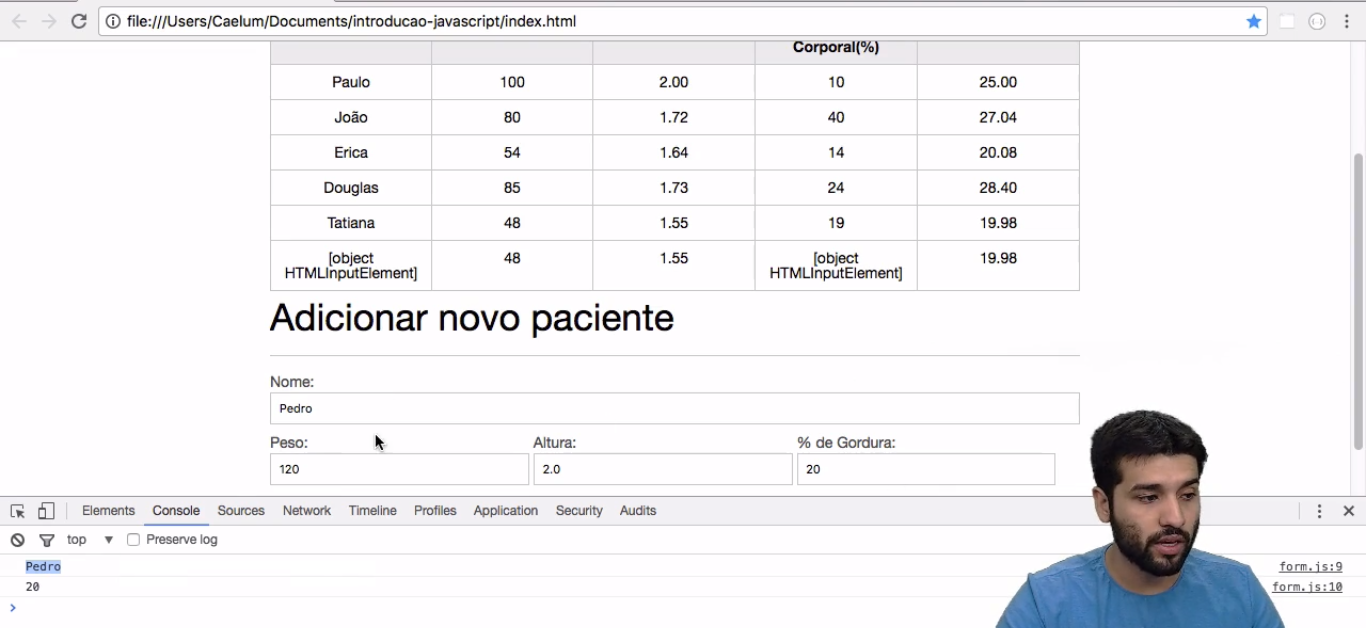
var paciente = obtemPacienteDoFormulario(form);

console.log(paciente);

Em seguida, testaremos preencher o formulário no browser, após clicarmos no botão "Adicionar", veremos os dados quebrados na ultima linha. No entanto, o objeto será impresso no console.



O objeto paciente representa as propriedades do paciente. Se imprimirmos paciente.nome, paciente.gordura, poderemos acessar cada um dos dados individualmente.



Esta maneira de representar uma variável que contém várias características já foi utilizada anteriormente. Quando selecionamos um elemento com o querySelector(), ele também nos devolverá um objeto, como no trecho abaixo do calcula-imc.js:

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

titulo.textContent = "Aparecida Nutricionista";

No exemplo, o elemento possui a propriedade textContent e value. Esse trecho do HTML é representado por um objeto. No caso da função obtemPacienteDoFormulario() do form.js, nós criamos as características do paciente.

Vamos continuar com o código adicionando outra característica: o imc. Seu valor será a função criada anteriormente calculaImc() e passaremos o peso e a gordura do formulário como parâmetros:

function obtemPacienteDoFormulario(form) {

var paciente = {

nome: form.nome.value,

peso: form.peso.value,

altura: form.altura.value,

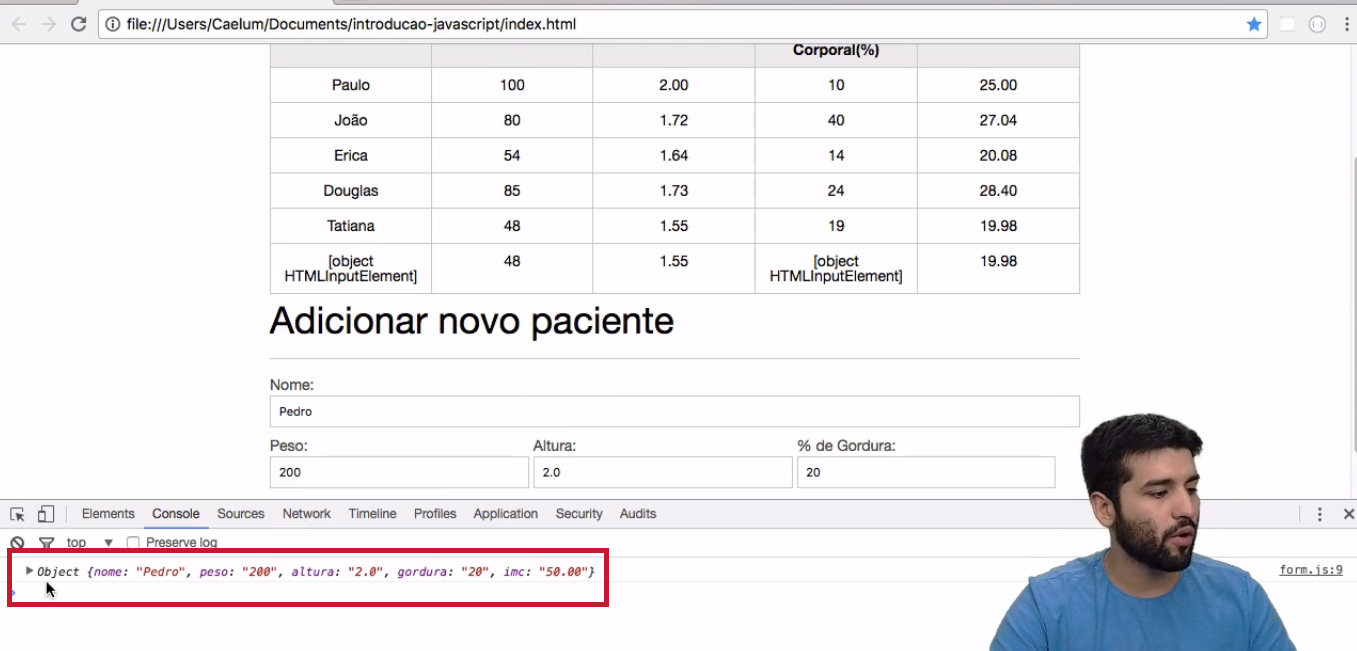
gordura: form.gordura.value,

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

}

}

Por fim, a função retornará o objeto que contém todos os dados do paciente, incluindo o imc:



Conseguimos representar o paciente adicionado por meio de um objeto e este se encontra disponível no começo do arquivo. Enxugamos as responsabilidades, e podemos continuar com esse processo de melhorias com o restante da função!

# 05 Objeto inválido

**O que está errado com o objeto Javascript abaixo ?**

**var carro = {**

**cor = "azul",**

**modelo = "fusca",**

**marca = "Volkswagen",**

**kms = 10000,**

**combustivel = "gasolina"**

**}**

Top of Form

* Alternativa correta



Não utilizamos o sinal de **=** entre a propriedade e o valor dela em um objeto Javascript, e sim o sinal de **:**. O correto seria:

var carro = {

cor : "azul",

modelo : "fusca",

marca : "Volkswagen",

kms: 10000,

combustivel : "gasolina"

}

Bottom of Form

 Correto, quando declaramos um objeto no Javascript utilizamos o sinal de **:** para separar propriedades e seus valores.

 Alternativa correta



A o valor da propriedade kms ficou sem aspas, o que dá um erro no Javascript. O correto seria:

var carro = {

cor = "azul",

modelo = "fusca",

marca = "Volkswagen",

kms = "10000",

combustivel = "gasolina"

}

  Alternativa correta



Faltou colocar aspas no nome das propriedades, apenas os valores estão com aspas. O correto seria:

var carro = {

"cor" = "azul",

"modelo" = "fusca",

"marca" = "Volkswagen",

"kms" = 10000,

"combustivel" = "gasolina"

}

Top of Form

Bottom of Form

Utilizamos objetos no Javascript como na maioria das linguagens de programação orientadas , aonde os objetos podemos compará-los com objetos da vida real. Um objeto é uma entidade independente, com propriedades e tipos. Compare-o com uma xícara, por exemplo. Uma xícara é um objeto, com propriedades. Uma xícara tem uma cor, uma forma, peso, um material de composição, etc. Da mesma forma, objetos em JavaScript podem ter propriedades, que definem suas características.

Para declaramos um objeto, utilizamos a sintaxe com {}, exemplo:

var xicara = {};

Só que de nada nos serve um objeto vazio, então podemos dar características a este objeto através de suas propriedades:

var xicara = {

cor: "azul",

peso: 125,

tipo: "chá"

};

As propriedades de um objeto são separadas por um **:** do seu valor e utilizamos uma vírgula ao final de cada propriedade para separá-la da próxima. Podemos acessar as propriedades de um objeto Javascript como abaixo:

xicara.cor // azul

xicara.peso // 125

xicara.tipo // chá

xicara.modelo // undefined, este objeto não possui a propriedade modelo

Ou seja, nosso exemplo estava errado por que estava utilizando o sinal de **=** em vez do de **:** para separar as propriedades de seus valores.

# 06 Melhorando ainda mais o código

## Transcrição

Outra responsabilidade a ser extraída é a criação da tr e das tds do paciente. Atualmente o trecho do código do form.js está assim:

//cria a tr e a td do 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 = nome;

pesoTd.textContent = peso;

alturaTd.textContent = altura;

gorduraTd.textContent = gordura;

imcTd.textContent = calculaImc(peso,altura);

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd);

pacienteTr.appendChild(alturaTd);

pacienteTr.appendChild(gorduraTd);

pacienteTr.appendChild(imcTd);

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:

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

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

event.preventDefault();

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

// Extraindo informacoes do paciente do form

var paciente = obtemPacienteDoFormulario(form);

// Cria a tr e a td do paciente

var pacienteTr = montaTr(paciente);

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

tabela.appendChild(pacienteTr);

});

Ao preenchermos os dados do formulário, veremos que eles continuarão sendo adicionados à tabela com o código mais legível.

## 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");

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;

}

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:

function montaTr(paciente){

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

pacienteTr.classList.add("paciente");

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;

}

Agora basta chamar essa função em montaTr:

function montaTd(dado, classe) {

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

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

td.textContent = paciente.nome;

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

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

nomeTd.textContent = paciente.nome;

//...

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;

}

Depois, chamaremos a função montaTd() na variável pesoTd.

function montaTr(paciente) {

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

pacienteTr.classList.add("paciente");

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

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

nomeTd.textContent = paciente.nome;

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

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd)

pacienteTr.appendChild(alturaTd)

pacienteTr.appendChild(gorduraTd)

pacienteTr.appendChild(imcTd)

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().

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-peso");

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

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

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd)

pacienteTr.appendChild(alturaTd)

pacienteTr.appendChild(gorduraTd)

pacienteTr.appendChild(imcTd)

return pacienteTr;

}

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

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

function montaTr(paciente){

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

pacienteTr.classList.add("paciente");

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

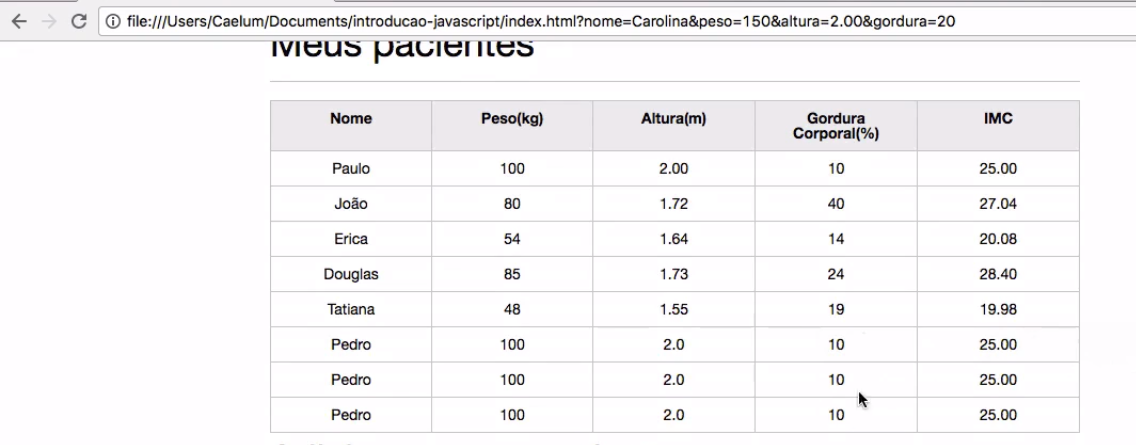
pacienteTr.appendChild(montaTd(paciente.peso, "info-peso"));

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

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

event.preventDefault();

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();

});

Desta forma, quando um paciente é adicionado, os dados no formulário serão apagados.

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.

Esta aula foi sobre boas práticas e refatoração, para que você possa se tornar um excelente desenvolvedor. Continuaremos a seguir!

# 07 Sobre boas práticas de código

**Pensando em boas práticas de código, qual das afirmativas é falsa quando falamos em quebrar uma função grande em diversas funções menores ?**

Top of Form

* Alternativa correta



Evitamos o problema das múltiplas responsabilidades em única grande função, afinal uma função só deve ter uma única responsabilidade.

Bottom of Form

  Alternativa correta



Melhora a manutenção do código, já que é mais fácil você cuidar de vários pequenos blocos simples do que um grande bloco complexo.

  Alternativa correta



O código fica mais fácil de testar, pois com diversas pequenas funções é muito mais fácil testá-las individualmente em busca de um erro.

  Alternativa correta



A legibilidade do código fica prejudicada visto que temos ler diversas funções em vez de apenas uma.

Top of Form

* Esta é a falsa, quebrar o código em pequenas partes **aumenta** a legibilidade.

Bottom of Form

Sabemos que quebrar uma grande função complexa é uma boa prática por causa de diversos fatores, mas podemos citar como os principais deles:

* Dar manutenção ao código fica muito mais fácil, visto que agora podemos examinar vários pequenos blocos , que são muito mais fáceis de compreender do que um grande bloco de texto
* Ao quebrar uma grande função, também estamos deixando ela com menos responsabilidades, com a meta de atingir o ideal de que cada função tenha apenas uma única responsabilidade.
* O código também fica muito mais fácil de testar, pois se temos diversas funções pequenas conseguimos ir testando uma a uma em busca de erros ou bugs do código.
* E por último, a legibilidade do código aumenta muito, pois dando nomes semânticos a cada uma das funções menores conseguimos deixar bem claro o que aquela parte do código deve fazer e facilita o entendimento do todo como um geral.

# 08 Quebrando o problema em partes menores

Temos o seguinte código:

<button class="botao">Calcula</button>

<input class="numero">

<input class="tabuada">

<span class="resultado"></span>

<script>

var botao = document.querySelector('.botao');

var numero = document.querySelector('.numero');

var tabuada = document.querySelector('.tabuada');

var resultado = document.querySelector('.resultado');

botao.addEventListener('click', function() {

resultado.textContent = numero.value \* tabuada.value;

});

</script>

Apesar de um código funcional ele pode ser melhor organizado para torná-lo mais legível e favorecer sua manutenção. Isso pode ser feito de diversas formas com auxílio de funções. Pense em como você organizaria diferente este código e compare o que pensou com a opinião do instrutor.

### Opinião do instrutor

Ao invés de escrevermos toda vez document.querySelector podemos criar uma função que funciona como atalho para ele:

<button class="botao">Calcula</button><input class="numero"><input class="tabuada"><span class="resultado"></span>

<script>

function buscaElemento(seletor) {

return document.querySelector(seletor);

}

var botao = buscaElemento('.botao');

var numero = buscaElemento('.numero');

var tabuada = buscaElemento('.tabuada');

var resultado = buscaElemento('.resultado');

botao.addEventListener('click', function() {

resultado.textContent = numero.value \* tabuada.value;

});

</script>

Para quem não conhece JavaScript, lendo o nome da função já faz ideia do que ela faz, no caso, ela busca um elemento.

Podemos deixar mais claro ainda o cálculo da tabuada isolando a lógica em uma função:

<button class="botao">Calcula</button><input class="numero"><input class="tabuada"><span class="resultado"></span>

<script>

function buscaElemento(seletor) {

return document.querySelector(seletor);

}

function aplicaTabuada(numero, tabuada) {

return numero \* tabuada;

}

var botao = buscaElemento('.botao');

var numero = buscaElemento('.numero');

var tabuada = buscaElemento('.tabuada');

var resultado = buscaElemento('.resultado');

botao.addEventListener('click', function() {

resultado.textContent = aplicaTabuada(numero.value, tabuada.value);

});

</script>

Muitas vezes pode ser que criar funções aqui ou ali possa estar aumentando as linhas de código que resolvem nosso problema. No entanto, quando esse código extra visa melhorar a sua legibilidade e manutenção, ele é muito bem-vindo!

# 09 Separando responsabilidades

Ubiraci criou o seguinte programa que calcula o total de idades dos convidados em uma lista:

<ul>

<li class="convidado">

Nome <span class="nome">Douglas</span>,

idade <span class="idade">23</span>

</li>

<li class="convidado">

Nome <span class="nome">Daniel</span>,

idade <span class="idade">42</span>

</li>

<li class="convidado">

Nome <span class="nome">Marcos</span>,

idade <span class="idade">27</span>

</li>

<li class="convidado">

Nome <span class="nome">Flávio</span>,

idade <span class="idade">18</span>

</li>

Total das idades: <span class="total"></span>

</ul>

<script>

var itens = document.querySelectorAll('.convidado');

var totalDasIdades = 0;

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

var idade = itens[i].querySelector('.idade').textContent;

totalDasIdades+=parseInt(idade);

}

document.querySelector('.total').textContent = totalDasIdades;

</script>

Apesar de funcionar, podemos separar as responsabilidades do código. Por exemplo, é possível extrair o trecho referente a uma lista de convidados da página para em seguida ter aquele que recebe a lista e realiza o total para nós. Pense em uma maneira de organizar melhor o código. Não há certo nem errado aqui. Em seguida, veja a opinião do instrutor.

### Opinião do instrutor

Uma solução possível é:

<ul>

<li class="convidado">

Nome <span class="nome">Douglas</span>,

idade <span class="idade">23</span>

</li>

<li class="convidado">

Nome <span class="nome">Daniel</span>,

idade <span class="idade">42</span>

</li>

<li class="convidado">

Nome <span class="nome">Marcos</span>,

idade <span class="idade">27</span>

</li>

<li class="convidado">

Nome <span class="nome">Flávio</span>,

idade <span class="idade">18</span>

</li>

Total das idades: <span class="total"></span>

</ul>

<script>

/\* esta função isola a responsabilidade de converter cada elemento do DOM em um convidado.

Esse convidado é um objeto JavaScript com as propriedade nome e idade.

Se alguém em nosso código quiser ler facilmente a lista de convidados,

basta chamar esse método que retornará uma lista de objetos já mastigada para se trabalhar.

\*/

function criaListaDeConvidados() {

var itens = document.querySelectorAll('.convidado');

var convidados = [];

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

var convidado = {

nome: itens[i].querySelector('.nome').textContent,

idade: parseInt(itens[i].querySelector('.idade').textContent)

};

convidados.push(convidado);

}

return convidados;

}

/\* essa função tem como responsabilidade extrair o total da lista de convidados retornando-o para quem chamá-la.

Isso é interessante, porque quem receber o resultado pode querer exibir na tela com um `alert`,

`console.log` ou até mesmo atualizando essa informação em algum elemento da página.\*/

function calculaTotalDasIdades(convidados) {

var total = 0;

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

total+=convidados[i].idade;

}

return total;

}

/\* essa função tem como responsabilidade receber um total qualquer e exibí-lo no HTML \*/

function exibeTotalDasIdades(total) {

document.querySelector('.total').textContent = total;

}

/\* usando nossas funções. Veja que uma pessoa fora do universo da programação

está mais inclinada a entender o que essas instruções fazem devido aos nomes autoexplicativos. \*/

var convidados = criaListaDeConvidados();

var totalDasIdades = calculaTotalDasIdades(convidados);

exibeTotalDasIdades(totalDasIdades);

</script>

# 10 Mãos na massa: organizando o código em vários arquivos

## Começando deste ponto ?

Começando deste ponto? Você pode fazer o [download](https://s3.amazonaws.com/caelum-online-public/introducao-javascript/stages/introducao-javascript-capitulo-5.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Neste capítulo vamos organizar nosso código para que ele fique mais fácil de manter e para que sigamos as boas práticas.

1- Nosso arquivo principal.js estava começando a ficar muito grande, logo vamos separá-lo em arquivo Javascript individuais, aonde cada arquivo ficará responsável por uma funcionalidade do sistema. Primeiro, crie um novo arquivo chamado form.js, e importe-o no fim no seu HTML:

// Restante do código HTML

</section>

</main>

<!-- Importação dos Javascripts AQUI -->

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

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

</body>

</html>

2- Com o arquivo criado e importado, mova tudo que estiver relacionado com nosso formulário de adicionar paciente para lá. Basicamente tudo dentro do event listener do click do botão:

//form.js

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

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

// Todo o código da função.

});

3- Aproveite e renomeie o seu arquivo principal.js para um nome mais semântico, como calcula-imc.js , que diz melhor o que aquele código faz.

4- Agora que já fizemos esta separação em arquivos, vamos começar a melhorar mais ainda o código. É a hora de exportarmos certas partes do código para funções, assim fica mais fácil reutilizá-las. Um bom exemplo é o código responsável por calcular um imc, que deve ser usado tanto na função cálcula IMC quanto quando criamos um novo paciente através do formulário. Crie a função calculaImc em seu arquivo calcula-imc.js como abaixo:

//calcula-imc.js

// restante do código

function calculaImc(peso, altura){

var imc = 0;

imc = peso / (altura \* altura);

return imc.toFixed(2);

}

5- Agora substitua o antigo cálculo na mão que fazíamos dentro do if por uma chamada a nossa recém criada função calculaImc, passando os parâmetros de acordo:

//calcula-imc.js

// Restante do código

if (pesoEhValido && alturaEhValida) {

var imc = calculaImc(peso,altura);

imcTd.textContent = imc;

}

// Restante do código

6- Vamos também chamar a função calculaImc no nosso form.js, para que o IMC do paciente também seja calculado quando ele for inserido na tabela. Adicione junto dos outros <td's> para que o contéudo do tdImc seja o retorno da função calculaImc

//form.js

nomeTd.textContent = nome;

pesoTd.textContent = peso;

alturaTd.textContent = altura;

gorduraTd.textContent = gordura;

//Adicionar aqui:

imcTd.textContent = calculaImc(peso,altura);

7- E não vamos esquecer de colocar o tdImc também dentro do <tr> paciente:

//form.js

pacienteTr.appendChild(nomeTd);

pacienteTr.appendChild(pesoTd);

pacienteTr.appendChild(alturaTd);

pacienteTr.appendChild(gorduraTd);

//Adicionar aqui:

pacienteTr.appendChild(imcTd);

Agora quando o paciente for adicionado na tabela, seu IMC também será calculado e inserido automaticamente!

### Opinião do instrutor

Graças as boas práticas de reuso de código e de uma melhor organização não precisamos escrever o código responsável de calcular o IMC duas vezes, e ainda separamos nosso código Javascript em arquivos diferentes, para que fique mais fácil dar manutenção ao mesmo.

# 11 Mãos na massa: extraindo mais códigos

Dando continuidade as boas práticas de organização de código, uma outra que devemos atacar é que nossa função anônima do form.js está com muitas responsabilidades, ela sozinha está fazendo muitas coisas. Ela obtém um paciente do formulário, cria a <tr> paciente, cria diversos <td>, coloca um dentro do outro e depois ainda adiciona-os na tabela!

São muitas funcionalidades para uma única função, vamos quebrá-la em funções menores para melhorar a legibilidade de nosso código:

1- O primeiro passo é extrair a responsabilidade de obter os dados do formulário para uma nova função. Crie a função obtemPacienteDoFormulario, que irá cuidar disto. Esta função deve receber o formulário e retornar todos os dados do paciente, e para isto vamos salvar estes dados dentro de um objeto do Javascript :

// form.js

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;

}

2- Agora vamos chamar está função no local aonde criávamos várias variáveis com peso, altura,etc..:

// form.js

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

event.preventDefault();

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

// Remova a criação das variaveis individuais e deixe apenas o objeto paciente

var paciente = obtemPacienteDoFormulario(form);

// Restante do código

...

});

3- A próxima parte que iremos atacar é a criação da <tr> paciente. Vamos criar uma função para criar a <tr> e uma para as <td>. Vamos começar pela função montaTd, que deve receber o dado que vai ser colocado dentro da td e a classe, e nos retornar o objeto montado:

function montaTd(dado, classe) {

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

td.classList.add(classe);

td.textContent = dado;

return td;

}

4- Agora aproveitando a função montaTd, vamos criar a montaTr, que vai receber um objeto paciente, criar cada uma das td, e colocar dentro da tr :

function montaTr(paciente) {

//Cria TR

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

pacienteTr.classList.add("paciente");

//Cria as TD's e a adiciona dentro da TR

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"));

// retorna a TR

return pacienteTr;

}

5- Agora vamos fazer as substituições no event listener:

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

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

event.preventDefault();

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

var paciente = obtemPacienteDoFormulario(form);

var pacienteTr = montaTr(paciente);

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

tabela.appendChild(pacienteTr);

});

6- O código ficou bem mais limpo e organizado! Por último , vamos fazer que o form seja limpo após adicionar um paciente. Utilize a função .reset() como último comando:

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

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

// restante do código

tabela.appendChild(pacienteTr);

form.reset();

});

Veja que agora temos um código muito mais organizado e fácil de ler!

### Opinião do instrutor

Neste capítulo vimos

* Dividir o código, separando por funcionalidades do sistema
* Os objetos em Javascript
* A quebrar funções grandes em funções menores ,com cada uma com sua responsabilidade
* A função form.reset() para limpar o formulário