**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){

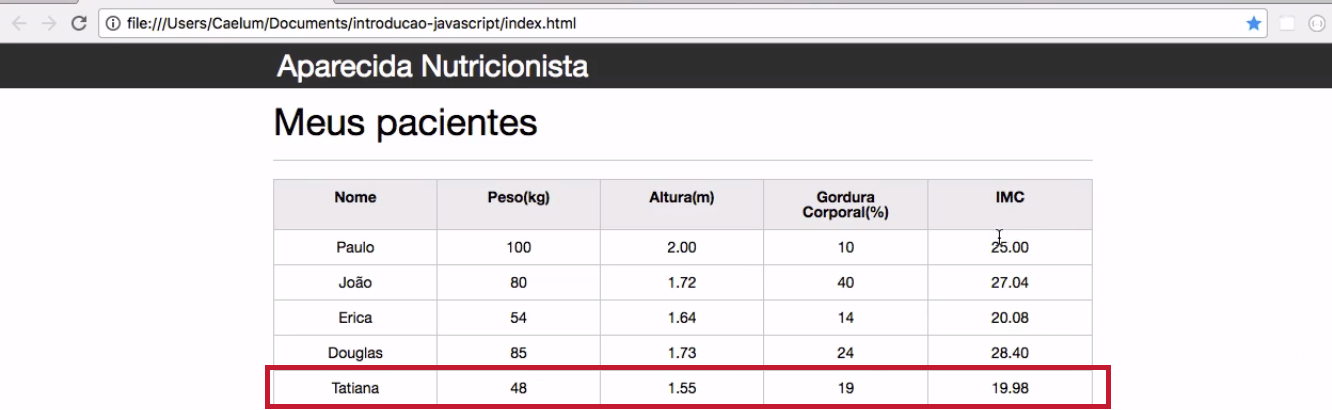
var imc = 0;

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();

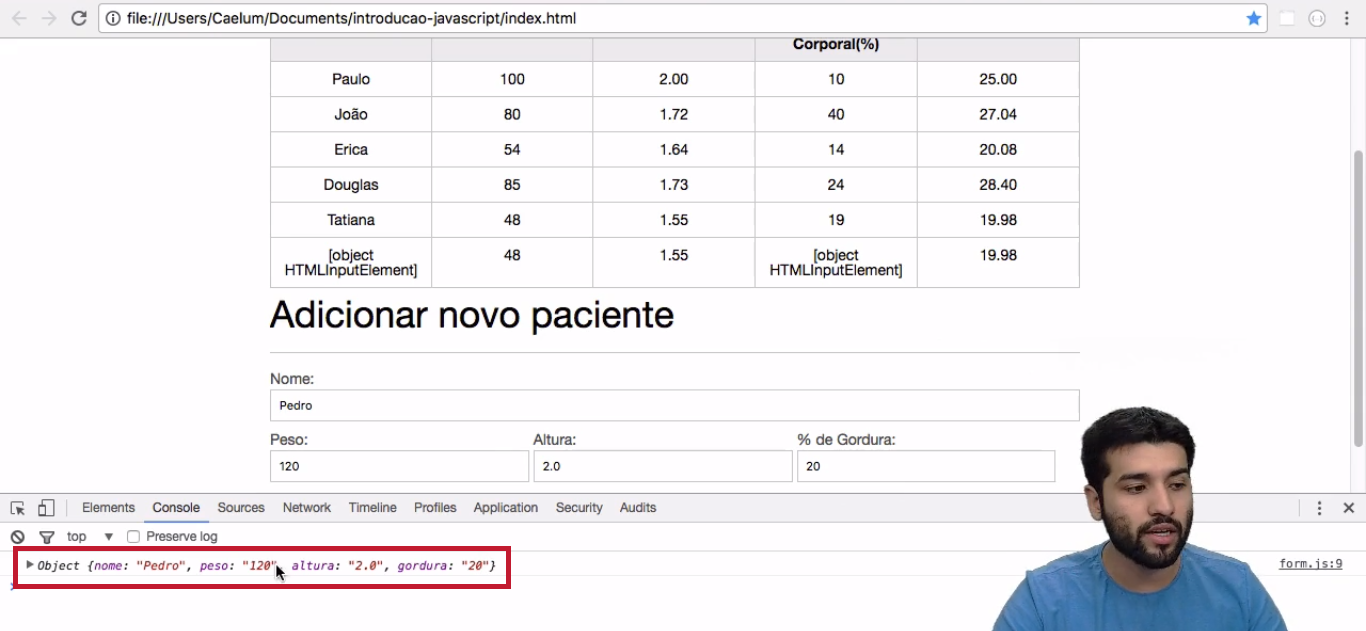
var form = document.querySelector("#form-adiciona");

//Extraindo informacoes do paciente do form

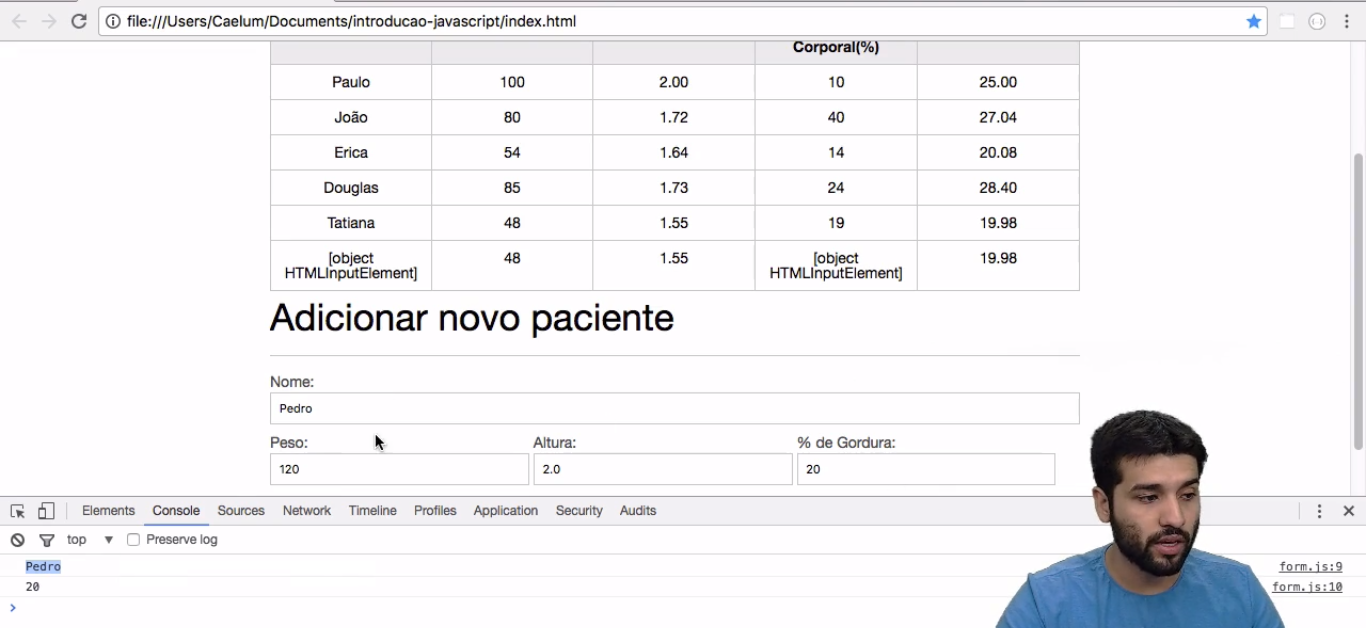
var paciente = obtemPacienteDoFormulario(form);

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,

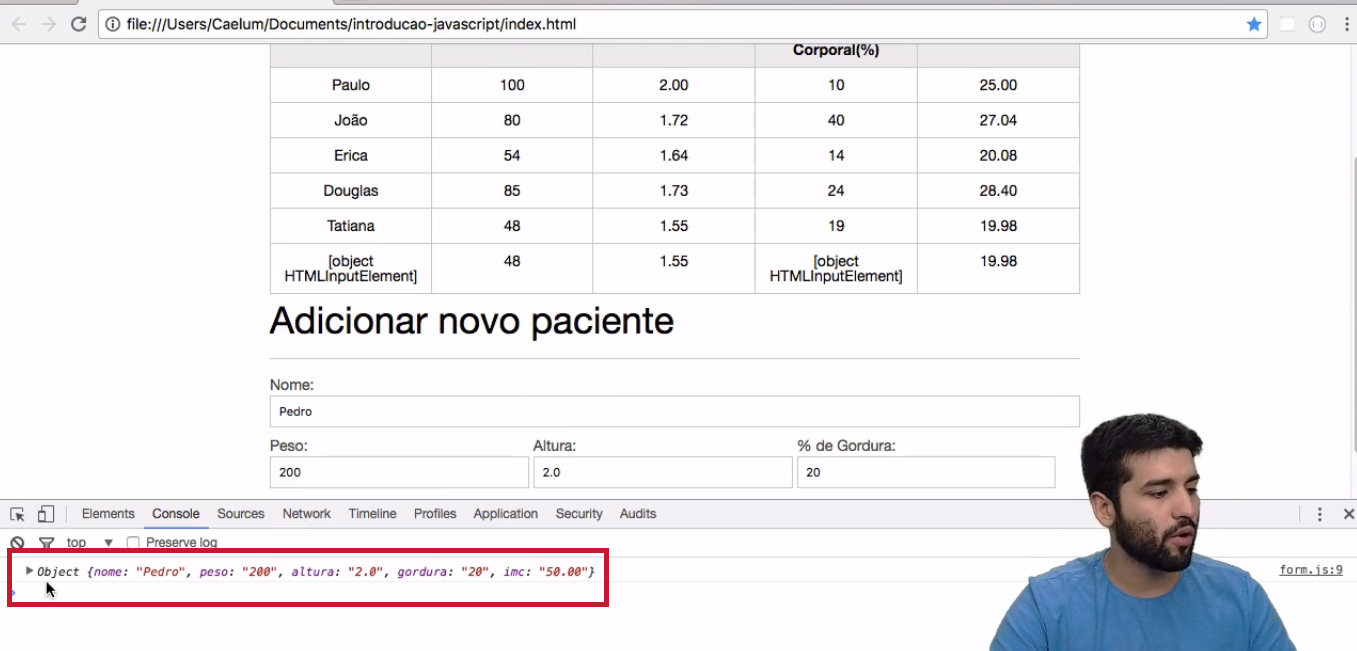
gordura: form.gordura.value,

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.