Lógica de Programação com JavaScript

Template String e Funções

Template String

Aprendemos anteriormente que textos em JavaScript devem estar sempre entre aspas duplas (“”) ou aspas simples (‘’). Contudo, isso não é totalmente verdade... rs. O JavaScript introduziu recentemente uma terceira maneira de identificarmos dados do tipo texto na linguagem, que é com o uso da crase (``). Essa nova maneira é chamada de *Template String*. Mas o que muda com esse novo jeito? Vejamos abaixo:

- Textos entre aspas duplas ou simples devem estar sempre NUMA MESMA LINHA. Isto é, se usarmos este método, todo o texto que queremos exibir deverá estar na mesma linha de comando do JavaScript. Sendo assim, esse formato não suporta a criação no código de textos multilinhas.

  Ex:

```
/* a linha abaixo não causa erro porque o texto a ser exibido está na mesma linha do comando */
document.write(“Meu texto para ser exibido”);

/* a linha abaixo causa erro porque, apesar de estar entre as aspas, o texto que queremos exibir está em linhas diferentes e o JS não dá suporte a isso com o uso de aspas (duplas ou simples) */
document.write(“Meu texto para ser exibido lá na página
do meu site”); // vai causar erro no console
```

  Com Template Strings esse erro não ocorre pois ele dá suporte a textos multilinha dentro do JS.

  Ex:

```
/* usando Template String (ou seja a crase ``) podemos
tranquilamente inserir textos em múltiplas linhas dentro do nosso
código JS */
document.write(`Meu texto para
dentro do JS agora pode ter várias linhas
porque o Template String da suporte a isso :) `);
```

- Quando precisamos concatenar valores com textos usando aspas (duplas ou simples) sempre devemos fazer uso do operador + para unir os valores (principalmente quando precisamos unir um texto a um valor vindo de uma variável).

  Ex:

  ```
  var nome = prompt("Digite seu nome:");
  document.write("Seu nome é: " + nome);
  ```

  O código acima vai exibir um pop-up pedindo ao usuário informar seu nome e em seguida guarda o que foi digitado pelo usuário na variável nome. Em seguida, exibe na tela o texto **"Seu nome é: "** e o nome informado pelo usuário.

  Perceba que para unirmos as duas informações tivemos que usar o operador + para concatenar o texto **"Seu nome é:"** com o valor guardado na variável **nome**.

  Com o **Template Strings**, esse tipo de união – entre uma informação estática e outra dinâmica – é facilitado, principalmente quando precisamos exibir várias informações de uma vez. Veja:

  ```
  var nome = prompt("Digite seu nome:");
  var sobrenome = prompt("Digite seu sobrenome:");
  var email = prompt("Digite seu email:");
  var telefone = prompt("Digite seu telefone:");
  var cidade = prompt("Digite sua cidade:");
  ```

```
document.write(`
    Seu nome é: ${ nome } <br>
    Seu sobrenome é: ${ sobrenome } <br>
    Seu e-mail é: ${ email } <br>
    Seu telefone é: ${ telefone } <br>
    Sua cidade é: ${ cidade }
`);
```

Note que estamos unindo textos estáticos (ex: “Seu nome é:”, “Seu telefone é:”) com textos vindos das variáveis que criamos anteriormente sem usar o operador +. Isso é possível graças ao marcador **${ ... }** que indica pro JavaScript que o que estiver ali é dinâmico e não deve ser tratado como texto estático.

Por isso, se eu guardar na variável nome o valor Andréia quando o document.write for executado ele exibirá na primeira linha **“Seu nome é: Andréia”** e não **“Seu nome é: ${ nome }”** pois o JS sabe que o que está entre o **${ ... }** é um valor dinâmico, neste caso o valor de uma variável. Aí, ele verifica se a variável existe, pega o que está dentro dela e exibe junto com o restante do texto. E, do mesmo modo, ele faz com os outros valores.

Vamos combinar, né?! Isso é bem mais simples do que ficar escrevendo vários **document.write()** um embaixo do outro com vários **“+”** unindo o texto que a gente quer mais o valor da variável. Isso facilita bastante a nossa vida como programadores. :)

Funções

Na primeira aula, aprendemos que **variáveis** *são gavetas na memória do computador onde guardamos alguma informação*. Ex: um texto, um nome, uma idade, um ano, um preço, etc. Ou seja, quando criamos uma variável, nela podemos guardar **VALORES** que, uma vez guardados, podem ser utilizados em qualquer lugar no nosso código, certo?

Da mesma forma, podemos guardar lógicas inteiras (ou seja, receitas

inteiras) num lugar para usarmos quando for necessário. Quando precisamos guardar uma receita inteira (código) e não um valor simples, utilizamos *funções*.
Sendo assim, *funções servem para guardar ações que podem ser executadas quando precisamos*.

Retomando nosso exemplo de receitas apresentado na primeira aula, vamos imaginar que temos um programa chamado *"Confeitaria"*. Nossa confeitaria faz 3 tipos de bolo: *bolo de chocolate*, *bolo de laranja* e *bolo de fubá*.

Quem escreve aqui é um professor que é péssimo cozinheiro e não faz a menor ideia de como se prepara um bolo. Logo, eu – na minha infinita ignorância sobre a arte da cozinha – vou criar um passo a passo básico do que seria o preparo de um bolo de chocolate de modo geral:

Bolo de Chocolate
1. Misturar ovos, farinha, leite, fermento e o chocolate na batedeira;
2. Pegue uma forma e unte com manteiga e um pouco de farinha;
3. Coloque a mistura da batedeira na fôrma;
4. Leve ao forno por 30 minutos a 100 graus;

Pronto! Temos a primeira receita da nossa confeitaria. Bom, para fazer os demais bolos seria mais ou menos a mesma coisa, veja:

Bolo de Laranja
1. Misturar ovos, farinha, leite, fermento e a laranja na batedeira;
2. Pegue uma forma e unte com manteiga e um pouco de farinha;
3. Coloque a mistura da batedeira na fôrma;
4. Leve ao forno por 30 minutos a 100 graus;

Por último temos o bolo de fubá:

Bolo de Fubá

1. Misturar ovos, farinha, leite, fermento e o fubá na batedeira;
2. Pegue uma forma e unte com manteiga e um pouco de farinha;
3. Coloque a mistura da batedeira na fôrma;
4. Leve ao forno por 30 minutos a 100 graus;

Agora, vamos pensar: Nosso programa se chama “Confeitaria”. Se eu criasse um código pro nosso programa contendo essa sequência de passos, um embaixo do outro, toda vez que eu executasse o programa ele me faria 3 bolos: um de chocolate, um de laranja e outro de fubá.

Mas não é isso que nós queremos! Nós queremos que ele faça o bolo que nós escolhermos, quando nós pedirmos, certo?
Sendo assim, queremos que ele execute a receita só quando eu precisar dela, então eu preciso ARMAZENAR A RECEITA em algum lugar pra quando eu precisar dela ela seja executada.

Receitas são códigos inteiros, certo? Receitas não são valores simples como um texto ou um número. *Então **não** dá pra guardar numa **variável***. Logo, se eu preciso armazenar uma sequência inteira de instruções pra eu poder usar quando eu precisar eu preciso de uma **FUNÇÃO**, porque como vimos anteriormente *funções servem para guardar ações que podem ser executadas quando precisamos.*

Sendo assim, cada uma das receitas anteriores seria uma função dentro do programa “Confeitaria”. Veja:

```
...
<script>
    function boloDeCholocate() {
        /* passo a passo para fazer o bolo de chocolate */
    }
```

```
        function boloDeLaranja() {
            /*  passo a passo para fazer o bolo de Laranja */
        }
        function boloDeFuba() {
            /*  passo a passo para fazer o bolo de fubá */
        }
    </script>
    ...
```

Sendo assim, TUDO que é necessário para fazer um bolo de chocolate está armazenado na função **boloDeChocolate()**. Assim como, TUDO que é necessário para fazer o bolo de laranja e o bolo de fubá está armazenado nas suas respectivas funções.

Finalmente, sempre que eu precisar fazer um bolo de chocolate, basta no meu programa eu chamar a função **boloDeChocolate** desse jeito, oh:

```
    <script>
        ... aqui vai todo o código que cria as funções ...
        // chamamos a função boloDeChocolate
        boloDeChocolate();
    </script>
```

E se quiséssemos fazer um bolo de fubá? Bastaria chamar a função **boloDeFuba();** e pronto!

Reforçando, a *função serve para eu armazenar uma receita inteira (sequência de instruções, códigos, ações) para eu usar quando eu quiser/precisar*.
Se eu preciso de um bolo de fubá, chamamos *boloDeFuba();*.
Se eu preciso de um bolo de laranja, chamamos *boloDeLaranja();*.
Se eu preciso de um bolo de chocolate, chamamos *boloDeChocolate();*

Obs.: Quando criamos uma função ela não é executada, ela só armazena a

sequência de códigos que colocamos dentro dela. Ela só é executada quando a chamamos em nosso código. Para executar uma função no JS basta seguir a sintaxe abaixo: **nomeDaFuncao();**.

As funções que acabamos de criar são chamadas de *funções simples*.

Agora, vamos rever nossas receitas e pensar um pouco mais.
Se você prestar bem atenção verá que os passos para fazer os bolos são exatamente os mesmos, certo? A única coisa que muda é o **sabor** do bolo: um é de chocolate, um é de laranja e outro é de fubá.

Então quer dizer que tivemos que copiar e colar o mesmo código nas três funções e só alterar o sabor (pra não termos 3 funções que fazem o mesmo tipo de bolo), não é?

Isso não é muito legal!
Já que os passos são os mesmos, e se nós tivéssemos só uma função que fizesse bolos e pudéssemos dizer qual sabor nós queremos para ela? Isso é possível, e é aí que introduzimos o 2º tipo de função que são as *funções com parâmetros*.

Parâmetro *é um meio seguro de passar uma informação para dentro de uma função, que ela usará na execução da tarefa que ela foi programada para fazer*.

Nossa receita será assim:

> **Faça um Bolo:**
> 1. Misturar ovos, farinha, leite, fermento e o SABOR na batedeira;
> 2. Pegue uma forma e unte com manteiga e um pouco de farinha;
> 3. Coloque a mistura da batedeira na fôrma;
> 4. Leve ao forno por 30 minutos a 100 graus;

A receita agora continua com a tarefa de fazer um bolo, porém ela não TEM POR PADRÃO um sabor definido. Você terá que escolher o sabor e em seguida seguir os passos da receita.

Nesse caso, podemos chamar o SABOR de parâmetro, pois ele será definido quando a função for chamada, ou seja, quando a tarefa for executada.

Se fossemos escrever essa função no JS ela ficaria mais ou menos assim:

```
...
```

```
<script>
    function facaUmBolo( sabor ) {
        /* passo a passo para fazer o bolo. Ex: */
        document.write(`
            Fazendo um bolo de ${ sabor } <br>
            1. Misturar ovos, farinha, leite, fermento e
            ${ sabor } na batedeira <br>
            2. Pegue uma forma e unte com manteiga e um
            pouco de farinha; <br>
            3. Coloque a mistura da batedeira na fôrma; <br>
            4. Leve ao forno por 30 minutos a 100 graus; <br>
        `);
    }
</script>
...
```

No código acima criamos uma função chamada **facaUmBolo** que possui um **parâmetro** chamado **sabor** (*parâmetros sempre estarão dentro dos parênteses de uma função*).

Assim sendo, quando chamarmos a função **facaUmBolo** devemos informar qual o sabor do bolo que deverá ser feito.

Você se lembra como executamos uma função? **nomeDaFuncao();** certo?

Aqui é a mesma coisa só que, como nossa função agora tem um parâmetro, dentro dos parênteses devemos informar o VALOR DESSE PARÂMETRO. No caso, vou usar como exemplo o valor *"Chocolate"*. Então veja como vai ficar a chamada da nossa função:

```
<script>
    ... aqui vai todo o código que cria a função facaUmBolo...
    // chamamos a função facaUmBolo
    facaUmBolo("Chocolate");
</script>
```

Vamos ver o que está acontecendo aqui:

1. Primeiro, estamos chamando a função **facaUmBolo**;
2. Em seguida, dentro dos parênteses informamos o valor *"Chocolate"*;
3. Como a função **facaUmBolo** possui um parâmetro chamado SABOR, a palavra *"Chocolate"* é armazenada no parâmetro SABOR;
4. Sendo assim, quando a função inicia a execução do código que está armazenado nela (no nosso exemplo é um document.write), toda vez que ela consultar o parâmetro SABOR, qual valor ela vai encontrar? Isso mesmo, *"Chocolate"*!
5. Logo, o que será escrito na tela do navegador quando essa função for executada será:

   Fazendo um bolo de **Chocolate**
   1. Misturar ovos, farinha, leite, fermento e **Chocolate** na batedeira;
   2. Pegue uma forma e unte com manteiga e um pouco de farinha;
   3. Coloque a mistura da batedeira na fôrma;
   4. Leve ao forno por 30 minutos a 100 graus;

Reforçando: sempre que eu preciso armazenar uma receita inteira (ou seja, toda uma sequência de instruções) para usá-la quando eu precisar, eu uso **FUNÇÕES**.

Agora, se eu preciso não só criar uma função (guardar uma receita inteira), mas também preciso enviar informações para dentro dela toda vez que ela for executada, eu utilizo **PARÂMETROS**.

Como no nosso exemplo: toda vez que eu precisar de um bolo basta chamar a função **facaUmBolo**. Porém, eu preciso falar para a função qual será o sabor do meu bolo. Aí basta criar um parâmetro (no caso, criamos o parâmetro **sabor**) e quando eu chamar a função passar o **sabor** que eu quero, se for *"Morango"*, vai ser **facaUmBolo("Morango");**, blz?

Ainda falando de parâmetros, podemos passar mais de um parâmetro para uma função.

Olhando nossa receita de bolo, dependendo do caso, podemos ter quantidades diferentes de ovos, farinha, leite e fermento para fazer um bolo, não podemos?

Do mesmo modo que pudemos informar o sabor do bolo que a função fará, podemos passar mais informações (se quisermos) que podem ser importantes/necessárias para um melhor funcionamento da nossa função.

Veja o exemplo abaixo:

```
...
<script>
    function facaUmBolo( sabor, qtdOvos, coposLeite ) {
        /* passo a passo para fazer o bolo. Ex: */
        document.write(`
            Fazendo um bolo de ${ sabor } <br>
            1. Misturar ${ qtdOvos } ovos, farinha,
            ${ coposLeite } copos de leite, fermento e
            ${ sabor } na batedeira <br>
            2. Pegue uma forma e unte com manteiga e um
            pouco de farinha; <br>
            3. Coloque a mistura da batedeira na fôrma; <br>
            4. Leve ao forno por 30 minutos a 100 graus; <br>
        `);
    }
</script>
...
```

Nossa função agora é um pouco mais específica. Ela aceita 3 parâmetros, ao invés de 1 apenas como no exemplo anterior.

Vamos chamar essa função modificada agora, da maneira abaixo para ver o que acontece:

```
<script>
    ... aqui vai todo o código que cria a função facaUmBolo...
```

```
        // chamamos a função facaUmBolo
        facaUmBolo("Chocolate", 3, 5);
    </script>
```

O que está acontecendo é o seguinte agora:

1. Primeiro, estamos chamando a função **facaUmBolo**;
2. Em seguida, dentro dos parênteses informamos o valor *"Chocolate"*, depois o valor *3* e por último *5*;
3. Como a função **facaUmBolo** possui 3 parâmetros chamados SABOR, QTDOVOS e COPOSLEITE, a palavra *"Chocolate"* é armazenada no parâmetro SABOR, o valor 3 é armazenado no parâmetro QTDOVOS e o valor 5 é armazenado no parâmetro COPOSLEITE;
4. Sendo assim, quando a função inicia a execução do código que está armazenado nela (no nosso exemplo é um document.write), toda vez que ela consultar o parâmetro SABOR, qual valor ela vai encontrar? Isso mesmo, *"Chocolate"*! E com os outros parâmetros vai acontecer o mesmo: o QTDOVOS será 3 e o COPOSLEITE será 5;
5. Logo, o que será escrito na tela do navegador quando essa função for executada será:

   Fazendo um bolo de **Chocolate**
   1. Misturar **3** ovos, farinha, **5** copos de leite, fermento e **Chocolate** na batedeira;
   2. Pegue uma forma e unte com manteiga e um pouco de farinha;
   3. Coloque a mistura da batedeira na fôrma;
   4. Leve ao forno por 30 minutos a 100 graus;

Facinho, né?!

Agora, o que acontece se eu chamo uma função com parâmetros e não informo algum valor. Ex:

```
    <script>
        ... aqui vai todo o código que cria a função facaUmBolo...
        /* chamamos a função facaUmBolo sem informar o
        parâmetro coposLeite (que é o último) */
```

```
        facaUmBolo("Chocolate", 3);
    </script>
```

Aqui, estamos informando o **sabor** ("Chocolate") e o **qtdOvos** (3) porém não informamos o **coposLeite**. Quando a função consultar os dois primeiros parâmetros encontrará *"Chocolate"* e *3* respectivamente. Contudo, quando ela consultar **coposLeite** não encontrará nada. O valor padrão que o JS atribui para **variáveis** e **parâmetros** é *undefined*.
Logo, quando o texto for exibido na tela do usuário ficará assim:

> Fazendo um bolo de **Chocolate**
> 1. Misturar **3** ovos, farinha, ***undefined*** copos de leite, fermento e **Chocolate** na batedeira;
> 2. Pegue uma forma e unte com manteiga e um pouco de farinha;
> 3. Coloque a mistura da batedeira na fôrma;
> 4. Leve ao forno por 30 minutos a 100 graus;

Não é muito legal isso né?
Para corrigir isso, o JS nos permite criar **parâmetros com valores padrão**. Isso é, se o valor de algum parâmetro não for fornecido, ele usa o valor padrão.

Vejamos:

```
    ...
    <script>
    function facaUmBolo(sabor="Fubá", qtdOvos=3, coposLeite=2 ) {
        /* passo a passo para fazer o bolo. Ex: */
        document.write(`
            Fazendo um bolo de ${ sabor } <br>
            1. Misturar ${ qtdOvos } ovos, farinha,
            ${ coposLeite } copos de leite, fermento e
            ${ sabor } na batedeira <br>
            2. Pegue uma forma e unte com manteiga e um pouco
            de farinha; <br>
            3. Coloque a mistura da batedeira na fôrma; <br>
            4. Leve ao forno por 30 minutos a 100 graus; <br>
```

```
        `);
    }
    </script>
    ...
```

Note que quando criamos os parâmetros da nossa função colocamos um sinal de igual (=) e em seguida atribuímos um valor para cada um dos parâmetros:

```
sabor="Fubá", qtdOvos=3, coposLeite=2
```

Com isso estamos dizendo o seguinte para o JS: *se esta função for chamada e nenhum ou algum dos parâmetros não for informado, use o valor padrão que estamos informando*.

Veja:

```
<script>
    ... aqui vai todo o código que cria a função facaUmBolo...
    /* chamamos a função facaUmBolo sem informar o
    parâmetro coposLeite (que é o último) */
    facaUmBolo("Chocolate", 5);
</script>
```

O resultado dessa chamada será:

Fazendo um bolo de **Chocolate**
1. Misturar **5** ovos, farinha, ***2*** copos de leite, fermento e **Chocolate** na batedeira;
2. Pegue uma forma e unte com manteiga e um pouco de farinha;
3. Coloque a mistura da batedeira na fôrma;
4. Leve ao forno por 30 minutos a 100 graus;

Quando chamamos a função **facaUmBolo**, informamos só os parâmetros **sabor** e **qtdOvos**. Por padrão, **sabor** possui o valor *"Fubá"* guardado em si, enquanto o **qtdOvos** possui o valor *3*. Como, ao chamarmos a função fornecemos valores novos, os valores padrão foram substituídos pelos que informamos na chamada da função: **"Fubá"** foi trocado por **"Chocolate"**, e **3** foi trocado por **5**.

O único parâmetro que não foi informado foi o **coposLeite**. Como por padrão ele agora tem o número **2**, quando a função **facaUmBolo** consultar esse parâmetro não vai mais achar *undefined* como no exemplo anterior. Agora ela vai achar o valor **2**. Por isso, ela exibe *"2 copos de leite"* na tela.

Se chamássemos a função como no modelo abaixo:

```
<script>
    ... aqui vai todo o código que cria a função facaUmBolo...
    /* chamamos a função facaUmBolo informando todos os
    Parâmetros */
    facaUmBolo("Chocolate", 5, 12);
</script>
```

Agora o valor do parâmetro **coposLeite** será substituído, trocando o 2 (valor padrão) pelo informado (que é 12), e na tela ficará assim:

Fazendo um bolo de **Chocolate**
1. Misturar **5** ovos, farinha, **12** copos de leite, fermento e **Chocolate** na batedeira;
2. Pegue uma forma e unte com manteiga e um pouco de farinha;
3. Coloque a mistura da batedeira na fôrma;
4. Leve ao forno por 30 minutos a 100 graus;

E se chamássemos a função assim:

```
<script>
    ... aqui vai todo o código que cria a função facaUmBolo...
    /* chamamos a função facaUmBolo sem informar nada */
    facaUmBolo();
</script>
```

Se eu não estou informando nada e os parâmetros tem valores padrão, logo o que será exibido? **Os valores padrão!!!**

Fazendo um bolo de **Fubá**
1. Misturar **3** ovos, farinha, **2** copos de leite, fermento e **Fubá** na

batedeira;
2. Pegue uma forma e unte com manteiga e um pouco de farinha;
3. Coloque a mistura da batedeira na fôrma;
4. Leve ao forno por 30 minutos a 100 graus;

Ou seja, por padrão minha função faz sempre um Bolo de Fubá até que você mude o valores dos parâmetros que ela recebe no momento que você a chamar!

Simples assim 😉

Mas ainda não acabou!

Já vimos que podemos ter funções simples (que só executam uma mesma tarefa sempre e não recebe nenhuma informação) e funções com parâmetros (que executam uma mesma tarefa, porém elas podem receber informações externas através desses parâmetros). Certo?

Um exemplo de função simples que comentei na aula foi o console.clear(); lembram? Ela simplesmente limpa o console. Ou seja, toda vez que eu a chamo, ela faz a mesma coisa (limpa o console). Logo, ela é um exemplo bem legal de função simples: só executa uma coisa e não recebe nenhuma informação externa.

Agora, como exemplo de função com parâmetro nós temos por exemplo o alert(); que usamos bastante nas aulas. O alert(); também sempre executa a mesma coisa: *mostrar um pop-up pro usuário com uma mensagem*. Mas como é que ele sabe qual mensagem mostrar pro usuário? Ele não tem como saber se a gente não informar, certo? *E como é que a gente passa uma informação para uma função?* Isso mesmo, através de **parâmetros**. Por isso, a gente sempre chama o alert(); assim:

```
<script>
    alert("Esta é a mensagem que eu quero exibir!");
</script>
```

O texto que a gente passa entre os parênteses é armazenado num parâmetro chamado *message* que o alert(); usa para poder pegar o texto e mostrar no pop-up. Exatamente como nas nossas funções de exemplo.

Por fim, existe um terceiro tipo de função chamada *função com retorno*.

Até o momento criamos funções que acionam operações (com ou sem parâmetros) cuja execução *"morre"* ali mesmo. Ou seja, ela executa o que ela foi programada para fazer e PRONTO!

Não há problema algum nisso. Às vezes é exatamente isso que queremos. Mas há momentos em que queremos usar O RESULTADO DA OPERAÇÃO DE UMA FUNÇÃO para fazer alguma outra coisa. Isto é, quando uma função for executada eu quero que ela ME RETORNE O RESULTADO DO QUE ELA FEZ pois eu vou utilizar isso para alguma outra coisa.

Para isso existem as *funções com retorno*: **quando eu preciso usar o resultado de uma operação feita por uma função**.

Quer um exemplo? A função parseInt(); que usamos várias vezes já. O que ela faz? Ela recebe um número, converte ele para um inteiro e RETORNA o número que informamos agora como um INTEIRO não é?

Veja:

```
<script>
    var numeroConvertido = parseInt(7.2356);
</script>
```

A função **parseInt** é uma *função com parâmetro e **com retorno***. Ela recebe um número (no caso 7.2356), converte esse número num inteiro (7) e RETORNA ESSE NÚMERO que é armazenado na variável **numeroConvertido**.

Como exemplo, vamos pensar num código que recebe um número e o transforma num valor tipo "moeda" no formato: "R$ 00,00", certo?

Queremos armazenar essa lógica para que toda vez que nós precisarmos mostrar um valor como uma moeda em real brasileiro (R$), nós possamos usar essa lógica para converter os valores que informarmos.

Então, vamos lá!

Se eu preciso armazenar uma lógica, eu preciso criar uma **função**. A função que vou criar vai se chamar **formataMoeda** e a receita dela será a seguinte:

formataMoeda:
1. Recebe um VALOR;
2. Converte o VALOR num número com casas decimais (float);
3. Fixa as casas decimais de VALOR em 2 casas;
4. Substitui o ponto "." por vírgula "," pois a escrita de moeda no Brasil é assim;
5. Pega o texto "R$" e concatena com o VALOR informado;
6. Retorna o VALOR em moeda no formato "R$ 00,00"

Vamos ao código:

```
...
<script>
function formataMoeda(valor) {
    var moeda = parseFloat(valor).toFixed(2).replace(".", ",");
    return "R$ " + moeda;
}
</script>
```

1. Como indicado na receita, criamos uma função chamada **formataMoeda**;
2. Como ela precisa *receber um valor*, criamos um parâmetro nela chamado **valor**;
3. Convertemos o **valor** num número com casas decimais (float) usando a função *parseFloat();*
4. Em seguida dizemos que depois de convertido, o **valor** precisa ter só 2 casas decimais. Para fixar as casas usamos a função *toFixed(2);*
5. Depois, substituímos o ponto (".") pela vírgula (",") usando a função *replace();*

6. Continuando, depois que o **valor** passou por todo esse processo de conversões e substituições, armazenamos o resultado disso na variável **moeda**;
7. Logo após isso, concatenamos o texto *"R$ "* com o valor armazenado na variável **moeda**;
8. Por fim, usamos o comando **return** para que a função RETORNE O RESULTADO DAS OPERAÇÕES QUE ELA FEZ;

Vamos fazer um teste dessa função. Veja o código abaixo:

```
<script>

function formataMoeda(valor) {

        var moeda = parseFloat(valor).toFixed(2).replace(".", ",");

        return "R$ " + moeda;

}

// executamos a função formataMoeda

formataMoeda(17.5689);

</script>
```

Quando executamos esse código, aparentemente nada acontece já que nada é exibido na tela, nenhum pop-up aparece ou nada é mostrado no console. Todavia sim, a função está rodando e convertendo o valor **17.5689** para **"R$ 17,56"**, mas o que ela está fazendo com essa informação?

Ela está retornando a informação, mas a informação que ela retorna não ESTÁ SENDO USADA ou GUARDADA EM LUGAR NENHUM!

Veja o código abaixo agora:

```
<script>

function formataMoeda(valor) {

        var moeda = parseFloat(valor).toFixed(2).replace(".", ",");

        return "R$ " + moeda;

}
```

```
// executamos a função formataMoeda e guardamos o resultado
// na variável precoProduto

var precoProduto = formataMoeda(17.5689);
// Exibe na tela: “Preço do Produto é: R$ 17,56”
document.write(`Preço do Produto é: ${ precoProduto }`);

</script>
```

Uma vez retornado o resultado da função *geralmente* precisamos armazená-lo em algum lugar (ou seja, numa variável).

No caso, criamos uma variável chamada **precoProduto** que recebe o valor formatado pela função **formataMoeda();**

Com isso armazenado na variável **precoProduto**, usamos o **document.write()** para exibir o valor convertido o que resulta em:

```
Preço do Produto é: R$ 17,56
```

E assim, concluímos o conceito de **funções** em programação.

ADENDO

Funções (assim como outros códigos JavaScript) podem ser escritos no bloco <script></script> dentro do arquivo **.html**, como já aprendemos a fazer, ou podem estar num arquivo separado com extensão **.js**

O motivo para colocarmos código JavaScript num arquivo separado com extensão **.js** é quando queremos utilizar o mesmo código em vários lugares diferentes.

Vamos imaginar a seguinte situação: digamos que tenhamos um código chamado **lojaDoZe.html** e outro chamado **lojaDoJoao.html.** Os dois códigos exibem produtos. Se exibem produtos, exibem preços e se os dois trabalham com moeda no formato Real Brasileiro, os dois precisarão converter os preços em Real, certo?

Criamos uma função chamada formataMoeda() anteriormente que poderia servir para os dois programas, não é? Tanto a **lojaDoZe.html** quanto a **lojaDoJoao.html** precisariam dessa funcionalidade.

Como poderíamos deixar a função **formataMoeda();** disponível para as duas lojas?

Bom, o primeiro método seria copiar a função e colar num bloco <script></script> tanto no lojaDoZe.html quando no lojaDoJoao.html, certo?

Sim, funcionaria! Mas se precisássemos mudar algo em nossa função, teríamos que mudar em 2 lugares: na lojaDoZe.html e na lojaDoJoao.html.

E se 50 lojas como essas usassem nosso código? Primeiro, teríamos que copiar e colar o código 50 vezes e se algo mudasse, teríamos que mudar 50 vezes (nos 50 arquivos diferentes).

Isso DEFINITIVAMENTE não é legal!

E se nós deixássemos essa função num lugar que todas as 50 lojas pudessem acessar e usar a função quando precisassem?

Aí é que entra o arquivo **.js**!!!

Se criarmos um arquivo chamado **formataMoeda.js** e colocarmos nossa função lá, ela estará guardada num único lugar.

Aí, basta irmos no arquivo **lojaDoZe.html** e no arquivo **lojaDoJoao.html** e adicionarmos a seguinte linha (sempre antes do bloco <script> que possui a lógica da loja):

```
<script src="caminho_do_arquivo/formataMoeda.js"></script>

<script>

    ... aqui vai a lógica do arquivo lojaDoZe.html ...
```

Assim, disponibilizamos tanto para o Zé quanto para o João (ou para todo mundo que precisar) um arquivo contendo a funcionalidade que queremos compartilhar (no caso é a função **formataMoeda**).

E se precisarmos mudar algo na função **formataMoeda()** vamos alterar **apenas em 1 lugar** (o arquivo **formataMoeda.js**) e assim que o salvarmos, todos os programas **.html** (lojaDoZe, lojaDoJoao, lojaDaMaria, mercadaoDoRicardao, etc) que fizerem uso desse arquivo pegarão automaticamente essas mudanças, pois eles sempre consultam essa

função. E se ela mudou, todos eles recebem a versão atualizada da função que foi alterada.

Quando disponibilizamos funções para serem utilizadas por outras pessoas em arquivos como o **formataMoeda.js** em programação chamamos esses arquivos de "libs" (que é um apelido para "Library" que significa "Biblioteca").

Então, quando um programador disser que ele *"tem uma lib com umas funções que você pode usar"*, ele está dizendo que ele tem um ou mais arquivos (seja de JavaScript ou seja de outra linguagem qualquer) que possui um conjunto de funções úteis que você pode usar no seu programa.

Aqui, nosso arquivo **formataMoeda.js** só possui uma função. Mas nada impede que um arquivo .js tenha várias funções. Por exemplo, poderíamos ter dentro desse arquivo várias funções que formatam moedas: para real (como no exemplo), para dólar, para peso, etc.

Veja:

Arquivo: **formataMoeda.js**

```
function formataReal(valor) {
        var moeda = parseFloat(valor).toFixed(2).replace(".", ",");
        return "R$ " + moeda;
}
function formataDolar(valor) {
        var moeda = parseFloat(valor).toFixed(2);
        return "$ " + moeda;
}
... e outras aqui ...
```

Ou seja, quando eu importar o arquivo formataMoeda.js

```
<script src="formataMoeda.js"></script>
```

Eu estou disponibilizando no meu programa todas as funções que existem dentro dele, aí eu as utilizo QUANDO E COMO EU PRECISAR!!! 😊