Unidade IV: Funções soulcode.com

**1 Funções**

As funções na programação desempenham um papel fundamental para a es truturação e organização do nosso código. Sem elas teríamos de repetir quan tas vezes fossem necessárias um conjunto de linhas de código para obter al gum resultado importante.

Seu uso é fundamental em qualquer linguagem, e há até mesmo aque las linguagens que seu desenvolvimento se pauta apenas na construção de funções.

Neste material, o objetivo é o aspecto prático e de consulta, portanto, vamos entender a utilidade de uma função e como definí-la.

**1.1 Para que serve uma função?**

Observe a seguinte situação:

Listagem 1: Repetição excessiva de código

1 console.log("Hello, World!");

2 console.log("!!!!!!!");

3 ...

4 console.log("Hello, World!");

5 console.log("!!!!!!!");

6 ...

7 console.log("Hello, World!");

8 console.log("!!!!!!!");

9 ...

10 console.log("Hello, World!");

11 console.log("!!!!!!!");

Temos duas linhas de código que são repetidas em excesso ao longo do ar quivo, imagine que estas linhas executam algum código complexo ou algum cálculo específico, seria muito imprático repetir o mesmo código sempre que quiser utilizar esta funcionalidade.

É daí que vem a ideia de função, podemos criá-la uma vez e utilizar quantas vezes forem necessárias, além de podermos modificar seu comportamento de diversas formas.

**1.2 Sintaxe de uma função**

Para declarar uma função, devemos nos atentar que se assemelha a declara ção de variáveis, seguindo as mesmas regras descrita na Unidade I.

Listagem 2: Sintaxe de uma função

1 function <nome da funcao> (<parametros?>) {

2 ... corpo da funcao

3 }

Vemos no código acima, como estruturar uma função, iremos precisar de um nome específico para ela, e ela também pode receber parâmetros (veremos adiante).

1

Unidade IV: Funções soulcode.com

O escopo da função é delimitado pelas chaves que se iniciam na linha 1 e encerram na linha 3. Ou seja, quaisquer variáveis definidas dentro da função, **não existirão fora dela**.

O exemplo descrito na Listagem 1, seria refatorado no seguinte esquema de funções:

Listagem 3: Refatoração em funções

1 function boasVindas() {

2 console.log("Hello, World!");

3 console.log("!!!!!!!");

4 }

5 ...

6 boasVindas();

7 ...

8 boasVindas();

9 ...

10 boasVindas();

Observe a declaração da função, após o nome, somos obrigados a definir parênteses mesmo vazios (sem parâmetros).

Nas linhas 6, 8 e 10 temos a chamada da função, ou invocação. Observe o uso dos parênteses após o nome da função, eles quem irão indicar que a função está sendo invocada na respectiva linha. Neste caso, os parênteses estão vazios na chamada pois a função não precisa de **parâmetros**.

**1.3 Sobre escopo de função**

É importante reforçar como o escopo de funções irá funcionar. Será similar ao escopos de blocos de controle como if/else e loops.

Listagem 4: Escopo de funções

1 function exemplo01() {

2 // escopo de funcao

3 let a = 5;

4 let b = 10;

5 console.log(`A soma eh ${a + b}`);

6 }

As variáveis **a** e **b** existem apenas dentro da função declarada. Porém, pode mos usar variáveis de fora da função desde que elas estejam em um escopo mais externo:

Listagem 5: Escopo externo

1 let num1 = 3;

2 let num2 = 5;

3

4 function exemplo02() {

5 console.log(`A soma eh ${num1 + num2}`);

6 }

Nesse caso, as variáveis **num1** e **num2** serão globais, pois estão definidas no escopo global do arquivo, e qualquer função pode acessar seus valores e re alizar alguma operação, como mostrada no exemplo.

2

Unidade IV: Funções soulcode.com

**2 Parâmetros**

**2.1 Noção de parâmetros**

Os parâmetros são informações que podemos passar para as funções, e estas informações podem ser usadas para algum processo.

Observe o seguinte exemplo com base na Listagem 5.

Listagem 6: Noção de parâmetros

1 exemplo02(); // 'A soma eh 8'

2 num1 = 10;

3 num2 = 15;

4 exemplo02(); // 'A soma eh 25'

Na invocação da função na linha 1, o resultado será pautada em como as variáveis foram definidas primeiramente.

Porém, caso queiramos uma soma diferente, somos forçados a alterar as variáveis antes para então executar a função. Isso se torna imprático, pois toda vez que necessitarmos uma soma diferente, é necessário modificar estas mesmas variáveis, e existe uma forma melhor de passar valores para uma função.

**2.2 Definindo parâmetros**

Ao invés de dependermos de variáveis globais, podemos escrever uma função mais independente com o uso de parâmetros.

Listagem 7: Definição de parâmetros

1 // Parametros = entrada = o que a funcao precisa para executar 2 // a e b sao PARAMETROS

3 function soma(a, b) {

4 console.log(`A soma de ${a} + ${b} = ${a + b}`);

5 }

Resumidamente, os parâmetros são informações dinâmicas passadas para uma função. Na linha 3, definimos quantos parâmetros uma função tem e definimos um nome para eles, similar a uma variável.

Listagem 8: Passagem de argumentos

1 soma(20, 75);

2 soma(10, 5);

3 soma(9, 80);

4 soma(1, 2);

É importante fazer uma distinção entre dois conceitos comumente ignorados na programação: **parâmetro** e **argumento**.

• **Parâmetro**: É uma variável definida na declaração de uma função. Ela atua como um espaço reservado para receber um valor quando a função é chamada. Os parâmetros são especificados entre os parênteses na

3

Unidade IV: Funções soulcode.com

declaração da função. Eles servem para permitir que a função aceite dados externos para serem processados dentro do seu bloco de código.

Listagem 9: Parâmetros

1 function soma(a, b) {

2 return a + b;

3 }

• **Argumento**: É o valor real fornecido para um parâmetro quando uma função é chamada. Ao chamar uma função, você passa argumentos que correspondem aos parâmetros da função. Esses argumentos podem ser literais, variáveis, expressões ou até mesmo outras funções.

Listagem 10: Parâmetros

1 // Passando literais como argumento

2 const resultado = soma(2, 3);

3

4 // Passando variaveis como argumento

5 const num1 = 5;

6 const num2 = 7;

7 const resultado2 = soma(num1, num2);

8

9 // Passando expressoes como argumento

10 // nesse caso as somas sao resolvidas primeiro

11 // antes da funcao ser chamada

12 const resultado3 = soma(1+1, 2+5);

**2.3 Mais sobre parâmetros**

Podemos definir quantos parâmetros forem necessários para uma função. Veja o exemplo a seguir:

Listagem 11: Mais parâmetros

1 function calculadora(a, b, c) {

2 console.log(`A soma eh ${a + b + c}`);

3 console.log(`A subtracao eh ${a - b - c}`);

4 console.log(`A multiplicacao eh ${a \* b \* c}`);

5 console.log(`A divisao eh ${a / b / c}`);

6 }

7 ...

8 calculadora(10, 20, 30);

9 calculadora(2, 5, 9);

10 calculadora(3, 2, 1);

**3 Retorno**

**3.1 Noção de retorno**

Em muitas situações as funções geram um resultado que pode ser usado de diversas formas. Neste exemplo, vamos usar o cálculo da média de 3 notas.

4

Unidade IV: Funções soulcode.com

Para isso temos a função a seguir:

Listagem 12: Cálculo da média

1 function media(n1, n2, n3) {

2 let resultado = (n1 + n2 + n3) / 3;

3 console.log(`A media eh ${resultado}`);

4 }

**A grande questão é**: Como podemos externalizar o resultado do cálculo? Podemos usar a ideia de variáveis globais para armazenar o valor calcu lado pela função, porém vamos cair no mesmo problema de depender de variáveis globais, só que dessa vez o problema será bem mais perceptível.

Listagem 13: Salvando resultado do cálculo

1 let resultado;

2

3 function media(n1, n2, n3) {

4 resultado = (n1 + n2 + n3) / 3;

5 }

6

7 media(7.5, 10, 9);

8

9 console.log(`A media eh ${resultado}`);

Agora, imagine que temos várias médias de diferentes disciplinas. Seria muito problemático lidar com a variável resultado e o código seria muito frágil e suscetível a erros.

Precisamos de um recurso melhor, pois cada chamada da função nos dá um resultado diferente, principalmente se tivermos diversas notas para cal cular. E essa ideia se estende para qualquer situação envolvendo funções e o seu "produto"final.

**3.2 Definindo retorno de função**

O retorno de uma função representa o encerramento da mesma, podemos junto a instrução do retorno indicar um valor que será externalizado. Ou seja, o valor poderá ser "capturado"por algo fora da função.

Listagem 14: Uso do return

1 function media(n1, n2, n3) {

2 let resultado = (n1 + n2 + n3) / 3;

3 console.log(`A media eh ${resultado}`);

4

5 return resultado; // indica que o valor da variavel sera "externalizado" 6 console.log("Oiee"); // dead-code = codigo que nunca vai ser executado 7 }

8

9 media(7.5, 6.5, 10); // chamada da funcao

No exemplo acima, temos na linha 5 a expressão de retorno, ela indica que a função media não tem mais motivos para continuar a executar, e deve voltar para o ponto de onde foi invocada (linha 9) para seguir a diante na execução.

5

Unidade IV: Funções soulcode.com

Após a palavra reservada **return**, podemos colocar valores literais, variá veis ou expressões. Tudo vai depender do objetivo da função que estamos construindo.

Listagem 15: Possibilidades do return

1 // o valor da variavel sera externalizado

2 return resultado;

3 // a expressao sera resolvida e o valor sera externalizada

4 return (a + b) / 2;

5 // a expressao sera resolvida (booleano) e sera externalizada 6 return a > 5;

7 // a expressao sera concatenada e sera externalizada

8 return "Hello, " + "World!";

**Observação**: O código abaixo da instrução return, nunca será executado, en tão é preciso prestar atenção em como o código é formado, para evitar o chamado **dead-code**.

**3.3 Utilizando o valor de retorno**

Nos exemplos acima, podemos dizer que a função retorna para onde foi in vocada, porém o valor presente no return é desperdiçado. Para usar o valor precisamos salvar ele em uma variável ou passar ele adiante para outras fun ções.

Listagem 16: Valor do retorno

1 ...

2 let medMatematica = media(7.5, 6.5, 10);

3 let medCiencias = media(9, 6.5, 8.5);

4 let medGramatica = media(6, 6.5, 7.5);

5

6 let mediaSemestre = media(medMatematica, medCiencias, medGramatica);

Com o uso do return, podemos ter vários calculos diversos a partir da mesma função. Essa ideia será muito útil para qualquer situação na programação. A forma de capturar os retornos vai depender muito de como você deseja que seu código fique estruturado. Por exemplo, o código abaixo funciona da mesma maneira, mas talvez não seja tão legível.

Listagem 17: Outra forma de utilizar os valores de retorno

1 let mediaSemestre = media(media(7.5, 6.5, 10), media(9, 6.5, 8.5), media(6, 6.5, 7.5));

Nesse formato, cada chamada para media, é resolvida antes da chamada mais externa seja realizada. O código Javascript será irá resolver as expressões internas antes de chamar a função. Nesse caso da Listagem 17, perdemos a capacidade de mostrar os valores de média de cada disciplina (caso fosse necessário).

6

Unidade IV: Funções soulcode.com

**4 Fundamentos das funções**

Todo programa, ou função numa perspectiva mais aproximada possui os se guintes componentes ou etapas básicas:

• **Entrada**: captação de dados que entram na função (parâmetros); • **Processamento**: transformação da entrada (lógica);

• **Saída**: transferência do produto final ao destino (retorno);

**4.1 Undefined implícito nas funções**

É importante notar que nas funções no Javascript não somos obrigados a pra ticar o **return**. Nesses casos, o valor retornado será implicitamente **undefined**. Da mesma forma na entrada, o Javascript é dinâmico nesse aspecto e po demos passar menos argumentos, porém, os parâmetros que não forem pre enchidos por valor serão **undefined** por padrão.

Listagem 18: Detalhes sobre funções

1 // retorno undefined

2 function imc(peso, altura) {

3 console.log(`Seu peso: ${peso}`);

4 console.log(`Sua altura: ${altura}`);

5 }

6

7 // peso = 90, altura = undefined

8 imc(90);

A execução acima não produzirá erros, porém é necessário estar atento a quantos argumentos cada função deve receber.

**4.2 Desvio de fluxos**

Uma função pode apresentar sim mais de uma expressão **return**, porém é necessário estar atento a questão dos fluxos que um código pode ter. Mui tas funções vão percorrer caminhos diversos para gerar uma saída, por isso podemos ter vários **return** dentro do corpo de uma função.

Listagem 19: Desvio de fluxos

1 function estaAprovado(n1, n2, n3) {

2 let med = media(n1, n2, n3);

3

4 if (med >= 7) {

5 console.log("Aprovado!");

6 return true;

7 } else {

8 console.log("Reprovado!");

9 return false;

10 }

11 }

7

Unidade IV: Funções soulcode.com

12 ...

13 let statusAluno = estaAprovado(6.5, 3.0, 8.6);

14 console.log(statusAluno);

No código acima, apenas um dos return será executado de fato, pois há a presença de uma estrutura if/else.

**5 Arrow functions**

As arrow functions caracterizam uma forma mais **sucinta** de representar fun ções no Javascript. Elas são utilizadas em contextos onde o código se torna mais simplificado como nas famosas **callbacks** de eventos.

Listagem 20: Sintaxe de uma arrow

1 let ola = () => {

2 console.log("Hello!");

3 };

4

5 ola();

Todos os conceitos de função abordados anteriormente se aplicam aqui. A grande diferença é a forma de definir, pois nesse caso guardamos a função dentro de uma variável, ou seja, a variável está servindo de container para uma função, que é chamada da maneira convencional na linha 5.

**5.1 Retorno imediato**

O retorno dentro de uma arrow funciona da mesma forma das functions con vencionais. Porém, é importante ressaltar as características da arrow func tion:

Listagem 21: Retorno imediato em arrow

1 const soma = (a, b) => {

2 console.log(`A soma eh ${a + b}`);

3 return a + b;

4 };

5

6 let resultadoSoma = soma(1, 2);

Existem situações em que uma função apenas realiza um cálculo entre os parâmetros e não necessita de mais de uma linha. Nesses casos, podemos omitir as chaves e palavra reservada return tornando a arrow mais sucinta ainda.

Listagem 22: Retorno em arrow

1 const soma = (a, b) => a + b;

2

3 let resultadoSoma = soma(1, 2);

8

Unidade IV: Funções soulcode.com

**5.2 Diferenças avançadas a respeito das arrow functions**

• **Valor do this fixo**: O valor do this em uma Arrow Function é determinado pelo contexto de onde a função é definida, e não pelo contexto de onde ela é chamada. Isso difere das funções tradicionais, onde o valor do this pode variar dependendo de como a função é invocada.

• **Não possuem binding de arguments**: As Arrow Functions não possuem a propriedade arguments, que é presente nas funções tradicionais e for nece acesso aos argumentos passados para a função.

O valor de **this**, será melhor abordado nas próximas unidades. Porém pode mos entender melhor a diferença a respeito do **binding** de arguments:

Listagem 23: Arguments

1 function soma() {

2 // Funcao tradicional

3 console.log(arguments);

4 }

5

6 const somaArrow = () => {

7 // Arrow Function

8 console.log(arguments);

9 }

10

11 soma(2, 3); // Saida: [2, 3]

12 somaArrow(2, 3); // Erro: ReferenceError: arguments is not defined

O **arguments** representa uma lista dos argumentos passados para a função de maneira implícita. Observe que a função soma não define parâmetro algum, porém podemos ainda sim passar argumentos.

9