JavaScript: Fundamentos para Desenvolvimento Web Interativo - Turma 2025B

3.1 Definindo funções

Definir funções em JavaScript é uma tarefa fundamental para qualquer desenvolvedor, pois permite modularizar o código, tornando-o mais reutilizável e organizado. Existem várias maneiras de definir funções em JavaScript, cada uma com suas próprias características e usos. Veremos neste curso quais são as principais formas de definir funções.

Declaração de função

A declaração de função (Function Declaration) é o método mais comum para definir uma função. A função é definida com a palavra-chave **function**, seguida pelo:

- Nome da função.
- Lista de argumentos para a função, entre parênteses e separados por vírgulas.
- Declarações JavaScript que definem a função, entre chaves **{ }**.

Por exemplo, o código demonstrado no quadro a seguir define uma função simples chamada **square**:

```
function square(numero) {
  return numero * numero;
}
```

Neste exemplo, a função **square** recebe um argumento chamado **numero**. A função consiste em uma instrução que indica para retornar o argumento da função (isto é, numero) multiplicado por si mesmo. A declaração **return** especifica o valor retornado pela função.

OBS: A definição de uma função não a executa. Definir a função é simplesmente nomear a função e especificar o que fazer quando a função é chamada. Chamar a função executa realmente as ações especificadas com os parâmetros indicados. Por exemplo, se você definir a função square, você pode chamá-la conforme quadro abaixo:

```
// Declaração da função square
function square(numero) {
  return numero * numero;
}

// Chamando a função square
square(5);     // resultado = 25
```

Esta declaração chama a função square com o argumento 5. A função executa as instruções e retorna o valor 25.

Hoisting

A declaração de função tem "**hoisting**", o que significa que ela pode ser chamada antes de ser definida no código. Para entender melhor este conceito, veja um exemplo no quadro abaixo:

```
// Chamando a função square
const resultado = square(2);
console.log(resultado);

// Declaração da função square
function square(numero) {
  return numero * numero;
}
```

Expressão de função

Uma expressão de função em JavaScript (Function Expression) é uma maneira de definir uma função e atribuí-la a uma variável ou a uma propriedade de um objeto. A diferença fundamental entre uma expressão de função e uma declaração de função (Function Declaration) é o comportamento em relação ao "hoisting" e a flexibilidade de uso em contextos mais dinâmicos.

Sintaxe

Uma expressão de função é geralmente definida usando a palavra-chave **function**, seguida por um nome (que é opcional), uma lista de parâmetros entre parênteses e um bloco de código entre chaves. Quando uma expressão de função é atribuída a uma variável (ou constante), ela pode ser chamada pelo nome da variável (ou constante), conforme exemplo demonstrado no quadro abaixo:

```
const somar = function(a, b) {
    return a + b;
};

console.log(somar(2, 3)); // Retorna 5
```

Aqui, a função **somar** é definida como uma expressão de função e atribuída à constante **somar**. Ela pode ser chamada pelo nome da constante.

Nomeadas vs. anônimas

As expressões de função podem ser nomeadas ou anônimas. Uma expressão de função anônima não tem um nome explícito, enquanto uma função nomeada tem um identificador que pode ser usado para referência interna. Veja um exemplo no quadro abaixo:

```
// Expressão de função anônima
const multiplicar = function (x, y) {
   return x * y;
};
console.log(multiplicar(2, 4));  //resultado = 8

// Expressão de função nomeada
const fatorial = function fac(n) {
    return n < 2 ? 1 : n * fac(n - 1);
};

console.log(fatorial(3)); // resultado = 6
```

A principal diferença entre essas duas abordagens é que uma expressão de função nomeada pode referenciar a si mesma dentro do próprio corpo da função, o que pode ser útil para a recursão.

Hoisting

Ao contrário das declarações de função, as expressões de função não sofrem "hoisting". Isso significa que uma expressão de função deve ser definida antes de ser usada. Se você tentar usar uma expressão de função antes de sua definição, receberá um erro. Veja um exemplo no quadro abaixo:

```
console.log(somar(2, 3)); // Causa um erro de referência
```

```
const somar = function(a, b) {
    return a + b;
};
```

Funções de seta

As funções de seta, também conhecidas como "**arrow functions**", são uma forma mais concisa de declarar funções em JavaScript, introduzidas no ECMAScript 2015 (ES6). Elas trazem uma sintaxe simplificada e comportamentos específicos em relação ao contexto this, tornando-as úteis para certos casos de uso.

Sintaxe simplificada

A sintaxe das arrow functions é mais compacta do que a das funções tradicionais. Para criar uma arrow function, você usa a seta **=>** para separar a lista de parâmetros do corpo da função.

Veja abaixo alguns exemplos:

- **Sem parâmetros:** Uma função sem parâmetros requer parênteses vazios. Veja um exemplo no quadro abaixo:

```
const ola = () => console.log("Olá, mundo!");
```

- **Com um parâmetro:** Se houver apenas um parâmetro, você pode omitir os parênteses ao redor do parâmetro. Veja um exemplo no quadro abaixo:

```
const saudar = nome => console.log("Olá, " + nome + "!");
```

- **Com múltiplos parâmetros:** Quando há mais de um parâmetro, os parênteses são necessários. Veja um exemplo no quadro abaixo:

```
const somar = (a, b) => a + b;
```

Corpo da função

O corpo das arrow functions pode ser uma expressão ou um bloco de código. Se for uma expressão, o retorno é implícito, o que significa que não é necessário usar a palavra-chave return. Veja os exemplos abaixo:

- **Retorno implícito:** Para uma expressão simples, o resultado é retornado automaticamente. Veja um exemplo no quadro abaixo:

```
const quadrado = x => x * x;
```

- **Bloco de código:** Se você precisa de um corpo de função mais complexo, use chaves {}. Nesse caso, é necessário usar return para retornar valores explicitamente. Veja um exemplo no quadro abaixo:

```
const calcular = (a, b) => {
  const resultado = a + b;
  return resultado;
};
```

Contexto this

Escopo léxico é uma maneira de definir como variáveis e funções são acessadas com base na posição em que foram definidas no código-fonte. Em JavaScript, uma função tem acesso ao seu próprio escopo, ao escopo de qualquer função que a envolva e ao escopo global.

O **this** é uma palavra-chave que refere-se ao contexto no qual uma função está sendo executada. Nas funções tradicionais, o contexto this é dinâmico, significando que ele pode mudar dependendo de como a função é chamada. Isso pode gerar confusão, especialmente em callbacks ou métodos que são usados em contextos variados. As arrow functions, por outro lado, têm um comportamento diferente: elas não têm seu próprio **this**, mas usam o contexto léxico do local onde foram definidas.

Veja um exemplo no quadro abaixo:

```
function Pessoa() {
    this.idade = 0;
    setInterval(() => {
        this.idade++; // "this" refere-se ao contexto da função envolvente (Pessoa)
        console.log(this.idade);
    }, 1000);
}

const p = new Pessoa();
```

Neste exemplo, o **setInterval** usa uma arrow function para incrementar a idade. O contexto **this** refere-se à instância de **Pessoa** porque a arrow function herda o escopo léxico do ambiente em que foi definida. Se tivéssemos usado uma função tradicional, o comportamento seria diferente, exigindo técnicas como **bind**, **call**, ou **apply** para manter o contexto correto.

Arrow functions e closures

As arrow functions, como outras funções em JavaScript, podem criar closures. Um closure ocorre quando uma função "lembra" das variáveis e do contexto do escopo onde foi definida, mesmo quando é usada fora desse escopo.

Veja um exemplo no quadro abaixo:

```
function criarContador() {
  let cont = 0;
  return () => {
    cont++;
    return cont;
  };
}

const contador = criarContador();
console.log(contador()); // Retorna 1
console.log(contador()); // Retorna 2
```

Neste exemplo, a arrow function mantém o contexto léxico do escopo em que foi definida, lembrando-se da variável contador. Esse comportamento é o mesmo para funções tradicionais, mas as arrow functions facilitam o uso de closures por causa de seu contexto **this** fixo.

◄ 2.10 Teste seus conhecimentos

Seguir para...

3.2 Chamando funções ►