Javascript

Programa de atração de formação de talentos
Bradesco / Visionaire

Aula 03 – Funções

Mark Joselli

Mark.Joselli@pucpr.br

Um dos **conceitos centrais**
no JavaScript é o de funções

Em Javascript funções são um **tipo de dado**, tal como números, objetos, etc...

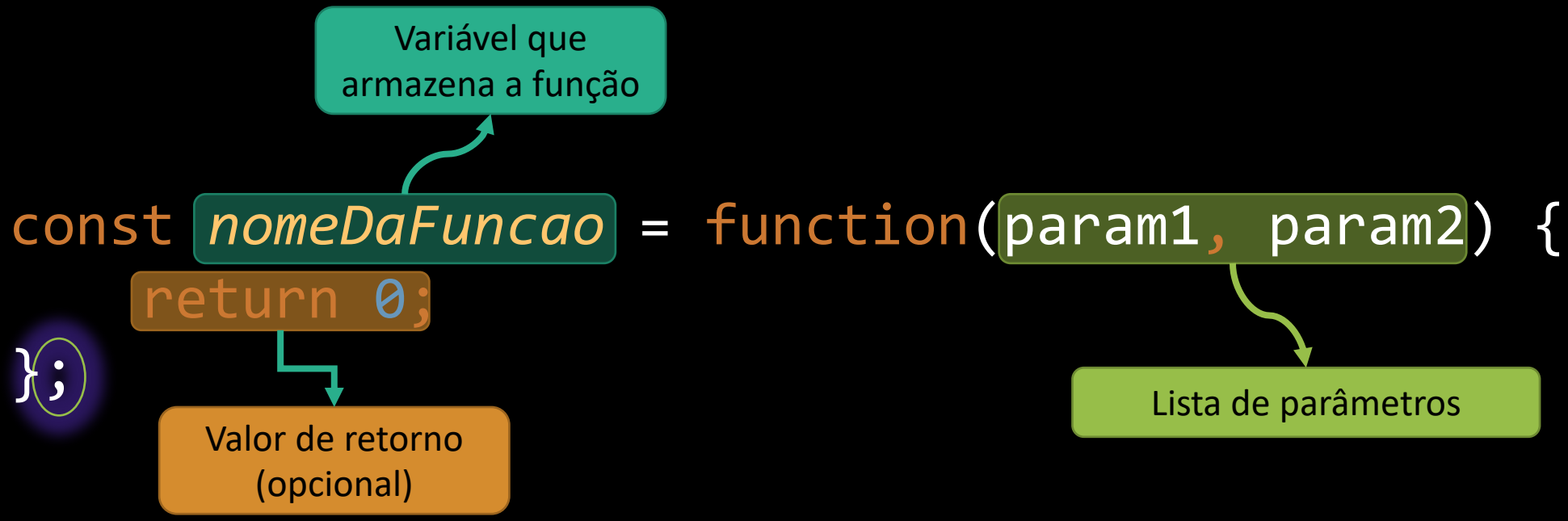
Definindo uma função

A sintaxe básica de definição de uma função é a seguinte:

```
const nomeDaFuncao = function(param1, param2) {
    return 0;
};
```

Definindo uma função

A sintaxe básica de definição de uma função é a seguinte:



Definindo uma função

- Sintaxe alternativa:
  - Também define uma variável chamada `nomeDaFuncao`
  - Não se utiliza ponto-e-vírgula
  - Será visível em qualquer ponto do código, mesmo antes da sua declaração ocorrer (hoisting)

```
function nomeDaFuncao(param1, param2) {
    return 0;
}
```

Sintaxe de flecha (arrow function)

- Ao invés da palavra **function** usa-se a =>

```
const quadrado = (x) => {
    return x*x;
};
```

  - Caso só haja um parâmetro, pode-se omitir os parênteses
  - Casó só tenha uma expressão, pode-se omitir a palavra chave **return** juntamente com as chaves

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

Escopo

Variáveis criadas com `var` utilizam a função como escopo. Elas também existem desde o início da função.

```
const minhaFuncao = function(param1, param2) {
    console.log(exemplo); //Undefined aqui, mas existe!
    if (param1 > param2) {
        var exemplo = "Criada aqui";
    }

    console.log(exemplo); //Existe aqui
    return 0;
}
```

OBSOLETO

Escopo

- `let` e `const` consideram o bloco onde foram criadas. E só existem do ponto onde foram declaradas em diante.
- Uma variável pode ser declarada em um bloco mais interno, ocultando a externa.

```
const minhaFuncao = function(param1, param2) {
    const exemplo = "Olá!";

    if (param1 > param2) {
        let exemplo = "Mais interno";
        let exemplo2 = "Criada aqui!";
    }

    console.log(exemplo2); //Erro, não existe aqui
}
```

Chamando uma função

```
const valor = quadrado(4);
```

- Caso uma função seja chamada com mais parâmetros do que os declarados, os parâmetros extras serão ignorados
- Caso uma função seja chamada com menos, os demais receberão valores `undefined`.

```
const valor = quadrado(4,5); //Também funciona!
```

Valores padrão

- Valores padrão são utilizados caso o parâmetro não seja fornecido
- Os parâmetros com valor padrão devem estar ao fim da lista de parâmetros

```
function elevar(base, expoente = 2) {
    let result = 1;
    for (let i = 0; i < expoente; i++) {
        result *= expoente;
    }
    return result;
}
```

Demais valores (varargs)

- Também é possível adicionar ao final um *array* para os demais valores fornecidos na função

```
function imprimir(rotulo, ...valores) {
    for (let valor of valores) {
        console.log(`${rotulo}: ${valor}`);
    }
}
```

- O que permitiria o uso da seguinte forma:

```
imprimir("Aluno", "Pedro", "Daniele", "Marcelo");
```

Espalhamento

- O conteúdo de um array também pode ser “espalhado” como parâmetro de uma função

```
let numeros = [2,5];
const resultado = elevar(...numeros);
```

- É possível usar esse operador até mesmo em outros arrays:

```
const valores = ["Católica", "do"];
const pucpr = ["Universidade", ...valores, "Paraná"];
console.log(pucpr);
```

Recursividade

- Uma função pode chamar ela mesma
- Isso permite resolver muitos problemas de maneira interessante

```
function fatorial(n) {
    if (n < 1) return 1;
    return n * fatorial(n-1);
}
```

- Funções recursivas geralmente possuem uma **condição de parada** e uma **expressão que se repete**.

Funções internas

- Você pode declarar uma função dentro de outra

```
function listarProdutos(desconto, ...produtos) {
    function imprimir(produto) {
        let precoFinal = produto.preco - desconto;
        console.log(`Item: ${produto.nome} Valor: ${precoFinal}`);
    }
    console.log("PRODUTOS");
    for (const produto of produtos) {
        imprimir(produto);
    }
}
```

Funções internas

- Observe que a regra de escopo também vale!

```
function listarProdutos(desconto, ...produtos) {
    function imprimir(produto) {
        let precoFinal = produto.preco - desconto;
        console.log(`Item: ${produto.nome} Valor: ${precoFinal}`);
    }
    console.log("PRODUTOS");
    for (const produto of produtos) {
        imprimir(produto);
    }
}
```

Funções internas

- Observe que a regra de escopo também vale!

```
function listarProdutos(desconto, ...produtos) {
    function imprimir(produto) {
        let precoFinal = produto.preco - desconto;
        console.log(`Item: ${produto.nome} Valor: ${precoFinal}`);
    }
    console.log("PRODUTOS");
    for (const produto of produtos) {
        imprimir(produto);
    }
}
```

Ordem das funções

- Uma função é chamada de **primeira ordem** se não receber ou retornar outras funções em seus parâmetros

- Já uma que faz isso é chamado de função de **segunda ordem** (high order function ou HOF)

- Funções de segunda ordem que alteram o comportamento de outras funções são chamadas de **decoradores**.

Considere a função

```
function filtrarMaiores(elemento, valores) {
    const ret = [];
    for (const valor of valores) {
        if (valor > elemento) {
            ret.push(valor);
        }
    }
    return ret;
}

const maiores = filtrarMaiores(5, [2,4,6,8,10,12]);
```

Considere a função

```
function filtrarMenores(elemento, valores) {
    const ret = [];
    for (const valor of valores) {
        if (valor <= elemento) {
            ret.push(valor);
        }
    }
    return ret;
}

const menores = filtrarMenores(5, [2,4,6,8,10,12]);
```

Considere a função

```
function filtrarMaiores(elemento, valores) {
    const ret = [];
    for (const valor of valores) {
        if (valor > elemento) {
            ret.push(valor);
        }
    }
    return ret;
}
```

O que muda é a condição

```
function filtrarMenores(elemento, valores) {
    let ret = [];
    for (const valor of valores) {
        if (valor <= elemento) {
            ret.push(valor);
        }
    }
    return ret;
}
```

Criando uma HOF

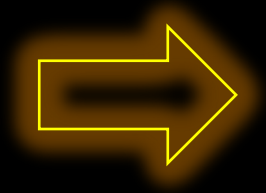
```
function filtrar(condicao, valores) {
    const ret = [];
    for (const valor of valores) {
        if (condicao(valor)) {
            ret.push(valor);
        }
    }
    return ret;
}

const maiores = filtrar(x => x > 5, [2,4,6,8,10,12]);
const menores = filtrar(x => x <= 5, [2,4,6,8,10,12]);
```

Closures

Quando uma função é retornada, ela guardará em sua memória o valor de todas as variáveis do escopo externo:

```
function criarContador() {
    let valor = 0;
    return function() {
        valor = valor + 1;
        return valor;
    }
}
```



```
const contador1 = criarContador();
const contador2 = criarContador();
console.log(contador1()); // 1
console.log(contador1()); // 2
console.log(contador2()); // 1
console.log(contador1()); // 3
console.log(contador2()); // 2
```

Atividades

1. Crie uma função chamada `mais`, que aceite 2 parâmetros e os some. Utilize as 3 sintaxes possíveis.
2. Crie uma função chamada `menos`. Caso seja passado apenas 1 parâmetro, retorne o valor negativo. Caso sejam passados 2, retorne a subtração dos dois. Exemplo:

```
console.log(menos(10)); //Imprime -10
console.log(menos(5,2)); //Imprime 3
```

3. Crie uma função `eCrescente` que teste se a lista informada é ou não crescente. A sequencia não será considerada crescente se houver um número menor que seu antecessor imediato

Atividades

4. Crie a função `maior`, que encontre o maior entre todos os valores passados em seus argumentos.

```
console.log(maior(1,10,-1,5)); //Imprime 10
console.log(maior(1,-100,5)); //Imprime 5
```

5. Escreva sua própria versão da função `join`. Esta função recebe uma lista e um separador (por padrão ",") e gera o texto dos objetos em seu interior separados por esse separador. Não se esqueça que o separador não ocorre após o último objeto da lista

6. Crie uma função que receba uma lista de objetos e um campo, e retorne uma lista com todos os valores desse campo sem repetição

Atividades

7. Escreva a versão não recursiva e a recursiva de uma função para calcular o n-ésimo termo da sequencia de fibonnaci:

$$\begin{cases} 1, se\ n = 0 \\ 1, se\ n = 1 \\ fib(n-1) + fib(n-2), se\ n > 1 \end{cases}$$

Ex.: Fib(6) = 13, pois:
1,1,2,3,5,8,13,21...

Atividades

8. Crie uma função `mapear` que aceite um array e uma função de mapeamento. Essa função recebe um elemento do array, realiza sobre ele qualquer cálculo, retornando outro.

- Exemplo:

```
const dobro = mapear([1,2,3,4], x => x * 2);
console.log(dobro); //[2,4,6,8]
```

Atividades

9. Crie a função Collatz que aceita como parâmetro o elemento inicial da sequencia de Collatz e retorna uma função. A cada chamada dessa função, retorne o próximo elemento da sequencia.

```
const seq = collatz(5);
console.log(seq()); //16
console.log(seq()); //8
console.log(seq()); //4
console.log(seq()); //2
console.log(seq()); //1
```

Se o número n for par, o próximo é n / 2
Se for ímpar é 3n+1
A sequencia termina em 1

Atividades

10. Crie a função `verbose` que recebe uma função como parâmetro e retorna outra, que imprime no console toda chamada que for feita na função original com seu resultado. Exemplo:

```
const soma = (a, b) => a + b;
const sum = verbose(soma);
sum(5,2); //Imprime soma(5,2) = 10
```

Dicas:

- A função `join` de uma lista pode ser usada para transforma-la em uma string separada por vírgula: `valores.join(",");`
- Além disso, variáveis de função possuem a propriedade `name` que imprimem o seu nome no momento da declaração.

Atividades

- Crie a função `fixar` que aceita uma função *f* e valores de parâmetros.
- Ela retorna outra função que chama *f* com esses parâmetros passados por primeiro como se estivessem “fixos”.

- Exemplos:

```
function log(modulo, nivel, texto) {
    console.log(`${nivel}: ${texto} (${modulo})`)
}
let logAula = fixar(log, `aula.js`, 'INFO');
logAula("Exemplo");  //Imprime INFO: Exemplo (aula.js)
logAula("PUCPR");  //Imprime INFO: PUCPR (aula.js)

let soma10 = fixar(soma, 10);
console.log(soma10(50)); //imprime 60
```

Dúvidas?

- Materiais:
  - Livro Eloquent Javascript:
    - 3ª edição (inglês): https://eloquentjavascript.net/
    - 2ª edição (português): https://github.com/braziljs/eloquente-javascript
  - The Modern Javascript Tutorial: https://javascript.info/