Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Rodrigo Costa

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Innominatum CONSULTORIA

JavaScript

Módulo 2

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Innominatum CONSULTORIA

Modularização e Reutilização de Código

Sessão 7

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Objetivo geral

Dotar os formandos das competências necessárias para estruturar lógica de negócio de forma modular, compreendendo as diferentes formas de definição de funções e as suas implicações no comportamento da linguagem. Os formandos deverão ser capazes de gerir o fluxo de dados através de parâmetros e contextos (Scope), aplicando sintaxes modernas de ES6 para otimizar a escrita de código e estabelecendo as bases para a programação funcional através de callbacks simples.

Objetivos específicos

Ao final da sessão, os formandos deverão ser capazes de:

- Definir e invocar funções em JavaScript, compreendendo a diferença entre function declaration e function expression;
- Utilizar arrow functions para escrever funções de forma mais concisa;
- Aplicar parâmetros em funções, incluindo parâmetros com valores por defeito;
- Utilizar callbacks simples para passar comportamento como argumento para uma função;
- Distinguir variáveis locais de variáveis globais.

Innominatum CONSULTORIA

Objetos em JS

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

O que é um objeto?

Um objeto em JavaScript é uma estrutura de dados fundamental, definida como uma coleção não ordenada de pares **chave-valor** (propriedades) que **representam entidades do mundo real ou conceitos abstratos**.

Coleção de dados

Um objeto tem como característica ser uma coleção de dados, que agrupa informações relacionadas (ex: nome, idade) em uma única variável. Possui uma estrutura de {chave: valor} onde as suas propriedades guardam os dados e os métodos são funções que executam ações relacionadas ao objeto.

Declarando um objeto

```
let carro = {
    marca: "Toyota",
    modelo: "Corolla",
    ano: 2023,
    cor: "Prata",
};
```

A nossa variável que antes armazenava apenas um valor, agora armazena uma coleção de dados, organizados por nomes (propriedades)

Objetos com condicionais

```
if (pessoa.estudante) {
    console.log(pessoa.nome + " é estudante.");
} else {
    console.log(pessoa.nome + " não é estudante.");
}
```

Podemos utilizar das chaves do nosso objeto para a criação de condicionais.

Objetos possuem propriedade de qualquer tipo de valor

```
// >> Propriedades podem ter QUALQUER tipo de valor <<

let produto = {
    nome: "Notebook",              // string
    preco: 899.99,                 // number
    emStock: true,                 // boolean
    categorias: ["Informática", "Portáteis"],  // array
    especificacoes: {              // outro objeto!
        ram: "16GB",
        processador: "Intel i7"
    }
};
```

Objetos possuem propriedade de qualquer tipo de valor

```
// >> Propriedades podem ter QUALQUER tipo de valor <<

let produto = {
    nome: "Notebook",              // string
    preco: 899.99,                 // number
    emStock: true,                 // boolean
    categorias: ["Informática", "Portáteis"],  // array
    especificacoes: {              // outro objeto!
        ram: "16GB",
        processador: "Intel i7"
    }
};
```

Objetos podem ser alterados

```
// ALTERAR propriedade existente
utilizador.email = "joao.novo@example.com";
console.log("Após alterar email:", utilizador);

// ADICIONAR nova propriedade
utilizador.idade = 30;
utilizador.ativo = true;
console.log("Após adicionar propriedades:", utilizador);

// REMOVER propriedade
delete utilizador.ativo;
console.log("Após remover 'ativo':", utilizador);
```

Podemos adicionar,

remover ou editar

informações de um objeto

Arrays podem receber objetos

```
// Lista de produtos (cada produto é um objeto)
let produtos = [
    { id: 1, nome: "Teclado", preco: 45.99, stock: 15 },
    { id: 2, nome: "Mouse", preco: 25.50, stock: 30 },
    { id: 3, nome: "Monitor", preco: 299.00, stock: 8 }
];
```

Umas das formas mais comum de receber uma coleção de dados de

API é via Array, Objetos ou ambos em uma mesma estrutura.

Manipulando um array de objetos

```
// Acessando propriedades do primeiro produto
console.log("Produto: ", produtos[0].nome, ": ", produtos[0].preco);

// Percorrendo o array de produtos
for (let i = 0; i < produtos.length; i++) {
    console.log("Produto ID:", produtos[i].id,
        "- Nome:", produtos[i].nome,
        "- Preço:", produtos[i].preco,
        "- Stock:", produtos[i].stock
    );
}
```

O que é uma Função?

O que é uma função?

Uma função em JavaScript é como uma máquina ou receita que executa uma tarefa específica. Ela recebe dados (entrada), processa esses dados e pode devolver um resultado (saída).

Por que precisamos de funções?

- Vantagens das funções:
  - Evita repetição de código;
  - Facilita manutenção e correção de erros;
  - Organiza melhor o código;
  - Permite reutilização;
  - Torna código mais legível e compreensível;

```
Exemplo - Sem vs Com Funções:
❌ SEM Função (repetitivo):
console.log("Olá, João! Bem-vindo!");
console.log("Olá, Maria! Bem-vindo!");
console.log("Olá, Pedro! Bem-vindo!");  // Repetitivo!
✅ COM Função (reutilizável):
function saudar(nome) {
  console.log("Olá, " + nome + "! Bem-vindo!");
}

saudar("João");
saudar("Maria");
saudar("Pedro");  // Reutilizável!
```

Função não executa sozinha

```
function dizerOla() {
  console.log("Olá!");
} // Declara a função

// Chama a função
dizerOla();
```

A definição de uma função, não garante que ela seja executada, para isso, a função precisa ser chamada (invocada).

Funções evitam repetição

```
// sem função
console.log("Olá!");
console.log("Olá!");
console.log("Olá!");
```

```
// com função
function dizerOla() {
  console.log("Olá!");
}

dizerOla();
dizerOla();
dizerOla();
```

Funções dão significado ao código

O nome da função explica o que acontece e não como acontece. Além disso, as funções ajudam a pensar de forma modular, onde cada função resolve um problema, e o programa é a soma dessas funções.

```
calcularTotal();
validarFormulario();
```

Anatomia da Função

Tipos de declaração de funções

Existem 4 formas de se declarar uma função em JavaScript (JS), que são elas:

- Declaração de função (Function Declaration);
- Expressão de Definição de Função (Function Definition Expression);
- Função de Seta (Arrow Function);
- Função Construtora (Function Construction).

Function Declaration

É a forma mais comum, onde a função é definida como uma sentença completa usando a palavra-chave function seguida de um nome obrigatório.

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

```
let resultado = somar(2, 3);
console.log(resultado);
```

O que é o Hoisting?

O içamento, também conhecido informalmente como hoisting, é uma característica do JavaScript em que o código se comporta como se todas as declarações de variáveis e de funções fossem "içadas" para o topo do seu escopo de execução. Esse comportamento ocorre tanto em nível de script global quanto dentro do corpo de funções onde as declarações foram feitas.

O hoisting com o var – Exemplo

```
console.log(linguagem);
// Resultado: undefined
var linguagem = "JavaScript";
console.log(linguagem);
// Resultado: "JavaScript"
```

Com o var, a declaração é içada, mas a atribuição não. É como se o JavaScript soubesse que a variável existe, mas ainda não sabe o que tem dentro dela. Nos bastidores é algo semelhante a sequência abaixo:

1. O JS move var linguagem; para a linha 1.
2. O console.log tenta ler e encontra algo vazio (undefined);
3. Só na linha 2 é que o valor "JavaScript" é atribuído.

Function expression

Define a função como parte de uma expressão maior, como uma atribuição de variável. Elas podem ser:

- Anônimas: Quando não possuem nome (ex: var f = function(x) { return x+1; });
- Nomeadas: Quando possuem um nome interno, útil para **recursividade.** Diferente das instruções de declaração, as expressões de função não são içadas e só podem ser chamadas após a linha em que foram atribuídas à variável.

```
const multiplicar = function (a, b) {
  return a * b;
};
```

Arrow functions – Função de seta

Podem ser utilizadas com um escopo e logo um return explícito, ou se, escopo onde o return é determinado pelo => podendo na mesma receber um ou mais parâmetros.

```
// Arrow Function (sintaxe mais curta com bloco definido)
const dividir = (x, y) => {
    return x / y;
};

console.log("20 / 4 =", dividir(20, 4));

// Arrow Function com retorno implícito (sem chaves)
const subtrair = (x, y) => x - y;

console.log("10 - 3 =", subtrair(10, 3));
```

Parâmetros e Reutilização de Código

Funções precisam de dados

Funções tornam-se mais úteis quando recebem valores e produzem resultados diferentes, esse valores que são passados para as funções, são chamados de **parâmetros da função**, os parâmetros tornam as funções reutilizáveis.

O que são parâmetros?

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

```
somar(2, 3); // 5
```

São uma lista de **identificadores (nomes)** colocados entre parênteses no momento da definição da função. Eles funcionam como **variáveis locais** dentro do corpo da função. Enquanto os parâmetros são os nomes definidos na criação da função, os **argumentos são os valores reais passados para esses parâmetros** no momento em que a função é chamada.

Parâmetros default

```
function dizerOla(nome = "Visitante") {
  console.log("Olá " + nome);
}
```

```
const multiplicar = (a, b = 1) => a * b;
```

As funções podem, também, possuir valor por defeito, atribuídos na declaração da função.

Funções também são valores

Em JS, uma função é um valor que poder ser:

- Guardado numa variável;
- Passada como argumento;
- Retornada como função.

Funções podem receber qualquer tipo de valor

Os parâmetros podem ser de qual tipo ou seja, eles podem receber:

- Números;
- Strings;
- Objetos;
- Arrays;
- Outras funções.

Callbacks

O que é um callback?

Um callback em JavaScript é uma função que é passada como argumento para outra função, para ser executada depois que algo acontecer. É literalmente uma função "chamada de volta" (called back) mais tarde.

O que é um callback?

Um callback:

- Passas uma função para outra função;
- A função guarda o teu callback;
- A função faz o trabalho dela/
- Quando termina, chama o teu callback;
- O teu código é executado.

Callback – Exemplo

```
// 1. Definimos a função que será o callback
function saudar(nome) {
  console.log("Olá, " + nome);
}

// 2. Criamos uma função que recebe outra função (callback) como argumento
function processarUsuario(callback) {
  const nome = "Formando";
  callback(nome); // Executamos a função recebida
}

// 3. Passamos 'saudar' como argumento (sem parênteses!)
processarUsuario(saudar);
```

Callback com arrow function

```
// 1. O setTimeout é a função principal (já vem no JS)
// 2. O primeiro argumento é o nosso CALLBACK (uma arrow function)
// 3. O segundo argumento é o tempo de espera (3000ms = 3 segundos)

setTimeout(() => {
  console.log("⏰ O tempo acabou! O teu bolo está pronto.");
}, 3000);

console.log("📖 Enquanto o bolo coze, vou ler um livro...");
```

Escopo (Variáveis Locais vs Globais)

O que é escopo?

O escopo (scope) define onde uma variável existe e onde ela pode ser utilizada, uma variável não pode ser acedida, quando ela está fora do escopo. Basicamente o escopo controla a visibilidade de uma determinada variável.

Por que o escopo é importante?

O escopo evita conflitos e efeitos colaterais, ao impedir que as variáveis interferiam uma nas outras, impedem, também, que os valores sejam alterados sem intenção e ainda facilita a manutenção do código.

Variáveis globais

```
let total = 100;

function mostrar() {
  console.log(total);
}
```

Variáveis globais podem ser alteradas de qualquer lugar, isso dificulta o debug e criam dependências invisíveis.

Variáveis locais

```
function calcular() {
  let subtotal = 50;
  console.log(subtotal);
}
```

Variáveis locais são privadas ao bloco onde foram criadas. Isto evita conflitos de nomes, facilita o debug e garante que a variável sirva apenas ao propósito específico da função ou bloco.

Escopo de bloco

Funções e estruturas condicionais possuem seus próprios blocos e portanto seus próprios escopos, que atuam como fronteiras naturais. Tudo que é declarado dentro deste escopo, só pertence a esse escopo, logo não existem fora dele, salvo quando declaramos como **var**.

```
if (true) {
  const mensagem = "Olá";
}
// mensagem não existe aqui fora

if (true) {
  var mensagem = "Olá";
}
// mensagem existe aqui fora
```

Hierarquia de escopos

O JS procura variáveis:

1. No escopo local;
2. No escopo externo;
3. Até ao global

Isso é conhecido como **cadeia de escopos**.

```
let y = 10;

function teste() {
  console.log(y);
}

teste();
// O JS encontra y no escopo global
```

Síntese

- Introdução às funções como blocos reutilizáveis de código;
- Definição da Anatomia da Função;
- Parâmetros e Reutilização;
- Callbacks (Introdução);
- Scope (Introdução).

Conclusão

Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Rodrigo Costa

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL