

Executar os processos de codificação, manutenção e documentação de aplicativos computacionais para desktop

SENAC PE

Thiago Nogueira

Instrutor de Educação Profissional

JavaScript – Declarações Condicionais

Declarações Condicionais

Declaração If

Na programação, podemos realizar uma tarefa baseada em uma condição usando uma instrução **if**:

```
if (true) {  
  console.log('Essa mensagem será impressa!');  
}  
// Imprime: Essa mensagem será impressa
```

- A declaração é composta por:
 - A palavra-chave **if** seguida por um conjunto de parênteses **()** que é seguido por um bloco de código, indicado por um conjunto de chaves **{}**.
 - Dentro dos parênteses **()**, é fornecida uma condição avaliada como **true** ou **false**.
 - Se a condição for avaliada como true, o código entre chaves {} será executado.
 - Se a condição for avaliada como false, o bloco não será executado.

Declarações Condicionais

If...Else

- Se quisermos adicionar algum comportamento padrão a instrução **if**, podemos adicionar uma instrução **else** para executar um bloco de código quando a condição for avaliada como **false**.

```
if (false) {  
  console.log('Esse bloco de código não será executado.');} else {  
  console.log('Mas esse bloco será!');}  
// Imprime: Mas esse bloco será!
```

- No exemplo acima, a declaração **else**:
 - Usa a palavra-chave **else** após o bloco de código de uma declaração **if**.
 - Possui um bloco de código envolvido por um conjunto de chaves **{}**.
 - O código dentro do bloco da instrução **else** será executado quando a condição **if** da instrução for avaliada como **false**.

Declarações Condicionais

else if

- Podemos adicionar mais condições ao nosso programa com uma declaração **else if**. A declaração **if else** permite mais de dois resultados possíveis.

```
let semaforo = 'amarelo';

if (semaforo === 'vermelho') {
  console.log('Pare!');
} else if (semaforo === 'amarelo') {
  console.log('Devegar. ');
} else if (semaforo === 'verde') {
  console.log('Prosseguir!');
} else {
  console.log('Cuidado, desconhecido!');
}
```

Declarações Condicionais

switch case

- É uma construção condicional que permite que um bloco de código seja executado com base na avaliação de uma expressão.

```
let item = 'papaia';

switch (item) {
  case 'tomate':
    console.log('Tomates custam R$0.49');
    break;
  case 'limão':
    console.log('Limões custam R$1.49');
    break;
  case 'papaia':
    console.log('Papias custam R$1.29');
    break;
  default:
    console.log('Item inválido');
    break;
}
```

JavaScript – Operadores

Operadores comparação

- Menor que: <
- Maior que: >
- Menor que ou igual a: <=
- Maior que ou igual a: >=
- É igual a: ==
- Não é igual a: !=
- OBS: == e === são diferentes. Uma **comparação estrita** (===) só é verdade se os operandos são do mesmo tipo e possuem o mesmo valor. A comparação mais usada é a **abstrata** (==), que converte os operandos para o mesmo tipo antes de fazer a comparação.

Operadores lógicos

Trabalhar com condicionais significa que usaremos booleanos **true** ou **false** valores. Em JavaScript, existem operadores que trabalham com valores booleanos conhecidos como *operadores lógicos*.

- O operador and (**&&**)
- O operador or (**||**)
- O operador not (**!**)

Operadores lógicos são frequentemente usados em instruções condicionais para adicionar outra camada de lógica ao nosso código.

Operadores short-circuit

```
const value = 0;  
const result = value && 'Truthy Value';  
console.log(result);
```

value é 0, que é um valor falso. Como o primeiro operando é falso, a expressão entra em curto-circuito, e o resultado é 0

```
const name = '';  
const displayName = name || 'Guest';  
console.log(displayName);
```

Name é uma string vazia, o que é falso. Portanto, a expressão entra em curto-circuito e 'Guest' é atribuída a displayName.

```
const value = 'Hello';  
const result = value && 'Truthy Value';  
console.log(result);
```

value é uma string não vazia, que é verdadeira. Portanto, o segundo operando 'Truthy Value' é retornado, pois é o último operando verdadeiro.

```
const name = 'Alice';  
const displayName = name || 'Guest';  
console.log(displayName);
```

name é uma string não vazia, que é verdadeira. Portanto, o primeiro operando 'Alice' é retornado, pois é o primeiro operando verdadeiro encontrado.

Operadores ternário

Com o espírito de usar sintaxe abreviada, podemos usar um *operador ternário* para simplificar uma instrução **if...else**.

Instrução if...else

```
let eNoite = true;

if (eNoite) {
  console.log('Ligue as luzes!');
} else {
  console.log('Desligue as luzes!');
}
```

ternário equivalente

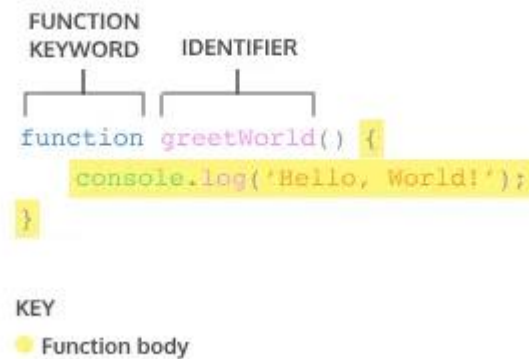
```
eNoite ? console.log('Ligue as luzes!') : console.log('Desligue as luzes!');
```

JavaScript - Funções

Funções

Declaração

- Em JavaScript, existem muitas maneiras de criar uma função. Uma maneira de criar uma função é usando uma *declaração*. Assim como uma declaração de variável vincula um valor a um nome de variável, uma declaração de função vincula uma função a um nome ou *identificador*.



```
function greetWorld() {  
    console.log('Hello, World!');  
}
```

FUNCTION
KEYWORD

IDENTIFIER

KEY
Function body

- Observe que, por hoisting, pode-se chamar a função `greetWorld()` antes dela ser declarada.

Funções

Chamando

- Uma declaração de função vincula uma função a um identificador. **No entanto, uma declaração de função não solicita a execução do código dentro do corpo da função,** apenas declara a existência da função.
- O código dentro do corpo de uma função **é executado, somente quando a função é chamada.**
- Para chamar uma função em seu código, digite o nome da função seguido de parênteses. Esta *chamada de função* executa o corpo da função ou todas as instruções entre chaves na declaração da função.

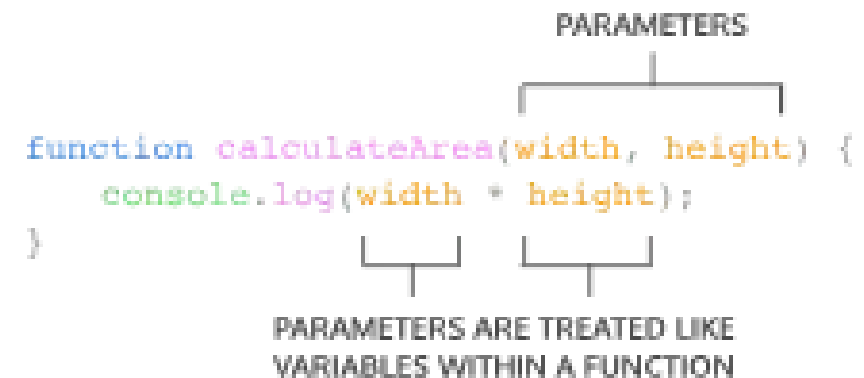
```
① function getGreeting() {  
  ③   console.log("Hello, World!");  
  }  
  ② getGreeting();  
  ④ // Code after function call
```



Funções

Parâmetros e argumentos

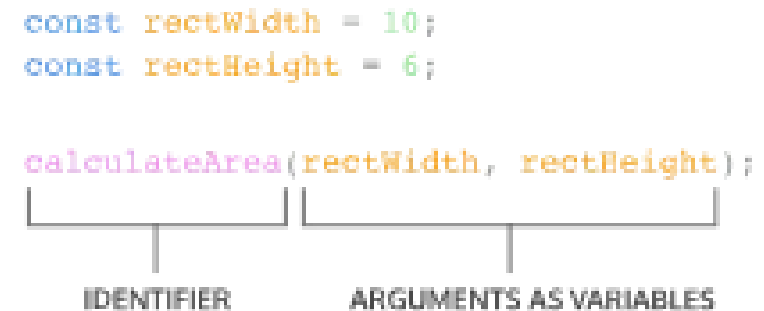
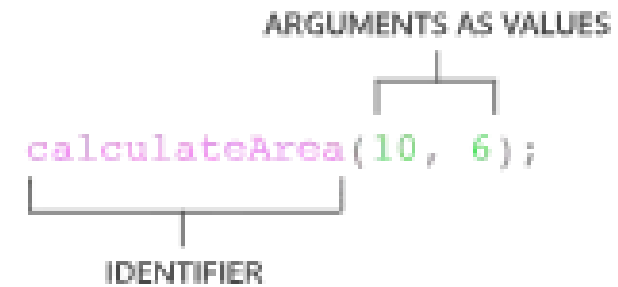
- Algumas funções podem receber entradas e usá-las para executar uma tarefa.
- Ao declarar uma função, podemos especificar seus *parâmetros*.
- Os parâmetros permitem que as funções aceitem entradas e executem uma tarefa usando as entradas.
- Usamos parâmetros como espaços reservados para informações que serão passadas para a função quando ela for chamada.



Funções

Parâmetros e argumentos

- Ao chamar uma função que possui parâmetros, especificamos os valores entre parênteses que seguem o nome da função.
- Os valores que são passados para a função quando ela é chamada são chamados *de argumentos*.
- As variáveis **rectWidth** e **rectHeight** são inicializadas com os valores de altura e largura de um retângulo antes de serem usadas na chamada de função.



Funções

return

- Quando uma função é chamada, o computador percorre o código da função e avalia o resultado. Por padrão, o valor resultante é **undefined**.
- Para retornar informações da chamada de função, usamos uma instrução **return**. Para criar uma instrução de retorno, usamos a palavra-chave **return** seguida do valor que desejamos retornar.

```
function rectangleArea(width, height) {  
  let area = width * height;  
}  
console.log(rectangleArea(5, 7)) // Imprime undefined
```

```
function calculateArea(width, height) {  
  const area = width * height;  
  return area;  
}
```

Diagram illustrating the components of the `return` statement in the example above:

RETURN	RETURN
KEYWORD	VALUE

Funções

Auxiliares

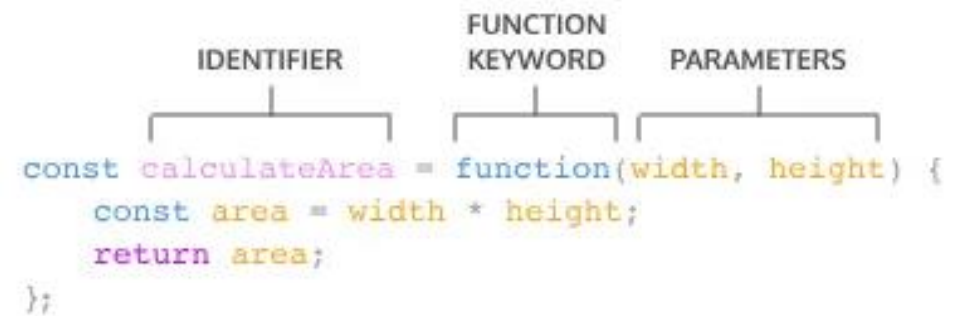
- Também podemos usar o valor de retorno de uma função dentro de outra função.
- Essas funções chamadas dentro de outra função são frequentemente chamadas de *funções auxiliares*.
- Como **cada função executa uma tarefa específica**, nosso código fica mais fácil de ler e depurar, se necessário.

```
function multiplicaPorNoveQuintos(number) {  
    return number * (9/5);  
};  
  
function getFahrenheit(celsius) {  
    return multiplicaPorNoveQuintos(celsius) + 32;  
};  
  
getFahrenheit(15); // Retorna 59
```

Funções

Expressão de Função

- Outra maneira de definir uma função é usar uma *expressão de função*. Para definir uma função dentro de uma expressão, podemos usar a palavra-chave **function**.
- Em uma expressão de função, o nome da função geralmente é omitido. Uma função sem nome é chamada de *função anônima*.
- Uma expressão de função geralmente é armazenada em uma variável para fazer referência a ela.



```
const calculateArea = function(width, height) {  
  const area = width * height;  
  return area;  
}
```

Funções

Arrow function

- ES6 introduziu *a sintaxe de função de seta*, uma maneira mais curta de escrever funções usando a notação especial “seta gorda” `() =>`
- As funções de seta eliminam a necessidade de digitar a palavra-chave `function` sempre que você precisar criar uma função.
- Em vez disso, você primeiro inclui os parâmetros dentro de `()` e depois adiciona uma seta `=>` que aponta para o corpo da função cercado `{ }` assim:

```
const rectangleArea = (width, height) => {  
  let area = width * height;  
  return area;  
};
```

Funções

Funções concisas

- JavaScript também fornece várias maneiras de refatorar a sintaxe da função de seta. A forma mais condensada da função é conhecida como *corpo conciso*.
- Funções que usam apenas um único parâmetro não precisam que esse parâmetro esteja entre parênteses. No entanto, se uma função tiver zero ou vários parâmetros, serão necessários parênteses.
- Um corpo de função composto por um bloco de linha única não precisa de chaves. Sem as chaves, tudo o que essa linha avaliar será retornado automaticamente. O conteúdo do bloco deve seguir imediatamente a seta => e a palavra **return** pode ser removida. Isso é conhecido como *retorno implícito*.

ZERO PARAMETERS

```
const functionName = () => {};
```

ONE PARAMETER

```
const functionName = paramOne => {};
```

TWO OR MORE PARAMETERS

```
const functionName = (paramOne, paramTwo) => {};
```

SINGLE-LINE BLOCK

```
const sumNumbers = number => number + number;
```

MULTI-LINE BLOCK

```
const sumNumbers = number => {  
  const sum = number + number;  
  return sum; } — RETURN STATEMENT  
};
```

JSDoc

JSDoc

Resumo

- Para documentar Javascript, é bom o uso do padrão descrito em JSDoc, cuja página com documentação explicativa é <https://jsdoc.app/>.
- Como instalar:
<https://github.com/jsdoc/jsdoc>
- Como utilizar:
<https://dev.to/cristuker/o-que-e-jsdoc-fdc>

```
/**
 * Esta é uma função de exemplo de uso de JSDoc
 *
 * @example
 *   exemplo(3, 5); // 8
 *
 * @param {Number} obrigatorio   Parametro obrigatório
 * @param {Number} [opcional]    Parametro opcional. Note os '[' ]'
 * @returns {Number}
 */
function exemplo (obrigatorio, opcional) {
  var resultado = 0;
  resultado = obrigatorio + (opcional || 0);
  return resultado;
}
```

Exercícios

Exercícios

Instruções


- Responder os exercícios propostos da aula 08
- Cada questão é resolvida em um arquivo .js separado seguindo o nome modelo:
 - **senac_UC13_aula08_qx.js** onde o x é substituído pelo número da questão
- Documentar o que cada questão pede e como foi resolvida em um arquivo README.md
 - Utilize o padrão JSDoc para comentar cada questão
- Enviar os exercícios para um repositório no GitHub
- Anexar link do repo na atividade do Teams


*“Ensinar é impregnar
de sentido o que
fazemos a cada
instante”*


Paulo Freire

Obrigado!

Thiago Nogueira

 [linkedin.com/tdn](https://www.linkedin.com/tdn)

 thiago.nogueira@pe.senac.br

 (81) 9 9627-0419