JS-12: JavaScript Moderno

Aula 1: Variáveis, Constantes, Tipos, Operadores, Estruturas Condicionais e de Repetição

Variáveis, Constantes e Tipos

Até a versão 5 do ECMAScript tínhamos apenas uma maneira de declarar variáveis em JavaScript: utilizando o comando **var**.

A partir da versão 6 da linguagem, temos agora a possibilidade de criar constantes com o comando **const** e também variáveis com o comando **let**.

Vejamos suas diferenças:

- **var**: variáveis criadas com este comando só respeitam 2 escopos: *escopo de função* e *escopo global*. Deste modo, uma variável criada com var dentro de uma função só existirá dentro dessa função, isto é, não estará disponível em nenhum outro lugar na sua aplicação. Caso seja declarada fora de uma função, ela será considerada *global*.

- **let**: variáveis criadas com este comando respeitará qualquer escopo onde ela foi declarada: *if, switch, while, for, funções, escopos vazios ({ }), etc.* Sendo assim, ao contrário do var, variáveis com let só serão consideradas globais se forem declaradas fora de QUALQUER escopo, isto é, se for diretamente dentro do escopo global da aplicação. Logo, uma vez que tudo que precisa ser processado dentro de um determinado escopo é finalizado, o garbage collector já pode eliminar essas referências da memória.

- **const**: constantes são valores que não podem ser REATRIBUÍDOS. Isto significa que, uma vez que um determinado valor é atribuído a uma constante, ele continuará com este valor ao longo do programa. Quando trabalhamos com valores primitivos (como *strings* e *números*), uma constante passa a impressão de não poder ser alterada (e nestes casos, de fato, ela não pode ser alterada). Porém, quando trabalhamos com tipos complexos como *Object* e *Array* fica claro que elas podem ser alteradas desde que esses valores não sejam substituídos de uma vez através do operador de atribuição (=). Ex:

```
const y = 10 // Number
y = 15 // TypeError: novo valor não pode ser reatribuído a uma constante

const carro = { titulo: “Ford K”, preco: 32552.56 } // Object
carro.titulo = “Volkswagen Gol” // Okay: alteração de uma propriedade
carro.quilometragem = 20000 // Okay: criação de uma nova propriedade
```

```
carro = { } // TypeError: não podemos reatribuir um novo valor à constante inteira

const nomes = [“Paula”, “Marcia”, “Pedro”] // Array “(object)”
nomes.push(“Jordan”) // Okay: adicionar um valor novo à lista existente
nomes.shift() // Okay: remoção de um valor da lista
nomes = [] // TypeError: não podemos reatribuir um novo valor à constante toda
```

Tipos de dados

O JavaScript (ECMAScript) possui os seguintes tipos disponíveis de modo geral:

- Tipos Primitivos:
  - **String**: textos - aspas duplas (“”), aspas simples (‘’) ou crase (``)
  - **Number**: numeros. Subtipos: float (decimal), integer (inteiro), NaN (not a number), Infinity e -Infinity (infinito positivo e negativo)
  - **null**: nulo (vazio)
  - **undefined**: indefinido (valor padrão das variáveis quando declaradas sem um valor inicial definido)
  - **Boolean**: true / false
  - **Symbol**: identificador único (usado em casos especiais com WeakMap, Map, Set e WeakSet)
- Tipos Complexos:
  - **Object**: objeto - coleção de informações estruturadas em propriedades/valores (chave-valor) e métodos.
  - **Array**: lista, vetor, matriz - lista de dados que podem ser acessados usando um índice numérico. O índice que representa o primeiro item de um Array é sempre ZERO (0)
- Valores considerados como FALSE no JavaScript (FALSY VALUES):
  - false
  - null
  - undefined
  - 0 (Zero)
  - NaN
  - “” (string vazia)

Operadores: Matemáticos, de Comparação, Lógicos e Atribuição

Vejamos os operadores disponíveis na linguagem JavaScript.

- Matemáticos:
    - **+** (adição)
    - **-** (subtração)
    - ***** (multiplicação)
    - **/** (divisão)
    - ****** (exponenciação. Ex: 5**3 = 5 elevado ao cubo)
    - **%** (módulo = resto de uma divisão)
    - **++** incremento de 1 em 1 (a++ faz com que o valor de a aumente 1)
    - **--** decremento de 1 em 1
    - **+=** incremento de outro valor (a += 5 faz com que o valor de a incremente de 5 em 5)
    - **-=** decremento de outro valor
    - ***=** multiplicação/atribuição (a *= 3 pega o que está em **a**, multiplica por 3 e atribui de volta na variável **a**)
    - **/=** divisão/atribuição (a /= 2 pega o que está em **a**, divide por 2 e atribui novamente na variável **a**)

- Comparação e Atribuição:
    - **==** igualdade (a == b testa se o que está em **a** é IGUAL ao que está em **b**)
    - **===** igualdade restritiva (a === b testa se o que está em **a** é IGUAL ao que está em **b** e se são do mesmo tipo)
    - **!=** diferença (a != b testa se **a** é DIFERENTE de **b**)
    - **!==** diferença restritiva (a !== b testa se o conteúdo e o tipo de dado da variável **a** é DIFERENTE de **b**)

- **>** maior que
- **>=** maior ou igual a
- **<** menor que
- **<=** menor ou igual a
- **=** atribuição (a = b coloca o valor que está em **b** em **a**)

- Lógicos:
  - **&&** (E) = todas as condições unidas por esse operador devem ser verdadeiras
    Ex:
    animal1 == “gato” && animal2 == “rato”
    Se animal1 for realmente “gato” E animal2 for realmente “rato” a expressão retornará TRUE. Se uma ou ambas não for verdade, retornará FALSE.
  - **||** (OU) = pelo menos uma das condições unidas por esse operador deve ser verdadeira
    Ex:
    animal1 == “gato” || animal2 == “rato”
    Se pelo menos uma das duas comparações for verdadeira, a expressão retornará TRUE. Se ambas forem falsas, a expressão retornará FALSE.
  - **!** (NOT) = verifica se a condição que vem em seguida é FALSE.
    Ex:
    !isAposentado
    Testa se o valor da variável isAposentado é FALSE. Se for, a expressão retorna TRUE (porque isAposentado tem valor definido como FALSE). Se a variável NÃO FOR FALSA aí a expressão retorna FALSE.

Estruturas Condicionais: if, switch e operador ternário

Dependendo de certas condições devemos tomar ações específicas. Nesses casos utilizamos as estruturas condicionais para checar tais condições e decidir o que fazer. No JavaScript temos 3 tipos de operações condicionais: if, switch e o operador ternário.

Vejamos cada um deles funciona:

Estrutura IF

Sintaxe:

```
if (condicao/condicoes) {
        // ação 1
}
else if (condicao/condicoes) {
        // ação 2
}
else {
        // ação padrão: se nenhuma condição foi satisfeita
        // realizar esta ação
}
```

- Estruturas **if** que se checam um mesmo "assunto" onde cada condição anula a anterior devem ser agrupados com uso de **else if**
- O comando **else** só anula o **if** anterior. Se há apenas 1 **else** no seu grupo de **if's** ele anulará apenas o **if** imediatamente anterior a ele.

Estrutura SWITCH

Sintaxe:

```
switch(condicao) {
        case valor1:
        [case valor2:
         case valor3…]
                // ação 1
        break

        case valor4:
        [case valor5:
         case valor6…]
                // ação 2
        break

        default:
                // ação padrão: se nenhum caso foi satisfeito, essa ação será executada
        break
}
```

- Cada grupo de casos de um switch deve ter obrigatoriamente como comando final o cláusula **break**

OPERADOR TERNÁRIO

Há casos em que devemos checar uma determinada condição e realizar uma operação simples dependendo do caso (o mais comum é atribuir um valor específico a uma variável dependendo do caso). Nesses cenários, o operador ternário é o indicado a ser usado.

Sintaxe:

(condicao**) ?** valor1 **:** valor2

Ex:

let msg = **(**idade >= 18**) ?** “é MAIOR de idade” **:** “é MENOR de idade”

Se o valor de idade for maior ou igual a 18, a primeira mensagem será atribuída à variável msg, caso contrário, a segunda mensagem será atribuída.

Estruturas de Repetição: while, do...while, for

Loopings são úteis quando queremos repetir uma operação várias vezes até que uma condição específica seja satisfeita. Na linguagem JavaScript temos 6 tipos de loopings, sendo 4 deles especializados. Inicialmente veremos 3 destes loopings: **while**, **do...while** e o **for**.

Estrutura WHILE

Sintaxe:

```
while (condicao) {
    // ação que deve ser repetida
}
```

- **Enquanto** a condição for verdadeira, o código disponível dentro das chaves ({ }) será executado
- O looping **WHILE**, primeiro verifica se a condição é verdadeira para depois executar o código. Se de cara a condição for FALSE, o looping nem sequer chega a ser executado.

Estrutura DO...WHILE

Sintaxe:

```
do {
    // ação a ser repetida
}
while (condicao)
```

- Diferentemente do **WHILE**, o **DO...WHILE** primeiro realiza a operação e depois verifica se a condição é verdadeira. Se for, continua realizando a operação, se não, para. Isso significa que no **DO...WHILE** a ação pretendida será realizada *PELO MENOS 1 VEZ SEMPRE*, pois só depois a condição será avaliada para saber se o looping deve

continuar ou não. No **WHILE** a condição é checada primeiro, logo, há a possibilidade da ação do looping *NUNCA* ser executada se de cara ela já for falsa.

Estrutura FOR

Sintaxe:

```
for (contador; condicao; incremento/decremento) {
    // ação a ser realizada
}
```

- Embora possamos usar o looping FOR para realizar outros tipos de operação de repetição, ele é um looping especificamente criado para quando queremos realizar contagens, ou seja, quando o laço de repetição envolve uma espécie de contador. Por isso, o primeiro parâmetro do laço FOR é justamente a definição de uma variável que irá servir de controle para as operações de contagem e o último é onde definimos o incremento/decremento dessa variável.
  Ex:

```
for (let indice = 1; indice <= 10; indice++) {
    // ação a ser repetida 10x
}
```

  *O exemplo anterior usa a variável **indice** para realizar o controle do looping. Ela irá de 1 até 10 (**indice<=10**) de 1 em 1 (**indice++**). Desse modo, definimos todas as condições necessárias para o nosso looping ser executado em apenas 1 linha. O **FOR** é um looping especializado, isto é, ele foi feito para facilitar operações de contagem (ou seja, quando temos um número específico de vezes que devemos executar determinada operação. O **WHILE** é um looping mais flexível e de propósito mais aberto podendo estar envolvido em contagens ou não).*