Conceitos Básicos de JavaScript

JavaScript é uma linguagem de programação client-side. Ela é utilizada para controlar o HTML e o CSS para manipular comportamentos na página. JavaScript não tem nada a ver com Java. Java é uma linguagem server-side ou backend, como PHP, Ruby, Python e tantas outras. A única coisa parecida entre eles é o nome

1. Inserindo códigos na própria página (*inline*):

Cria-se uma *tag* <script>, informando que o valor do atributo 'type' é 'text/javascript', então, coloca-se o código JavaScript dentro dessa *tag*.

Exemplo:

```
1.  <script type="text/javascript">
2.   alert('Olá mundo!');
3.  </script>
```

2. Relacionando um arquivo externo na página

Essa forma é bem parecida com a inserção de códigos JavaScript *inline*, a maior diferença é que não coloca-se o código JavaScript dentro da *tag*, visto que esse código estará em um arquivo externo. Assim, simplesmente é preenchido o atributo 'src' da *tag* **<script>** com o caminho para o arquivo em questão.

Essa forma também permite carregar arquivos JavaScript sem ter que baixá-los para o seu projeto. Isso é bastante utilizado como uma forma de fazer com que arquivos que são usados por muitos projetos, como por exemplo a jQuery, fiquem armazenados em cache, sendo então carregados de forma mais rápida.

Exemplo 1 - adicionando um JavaScript do nosso projeto:

Imagine que o projeto está com a seguinte estrutura de diretórios:

projeto/

arquivo.html (página que irá adicionar o arquivo JavaScript)

js/

meu-arquivo.js

Assim, se queremos que a página 'arquivo.html' carregue o arquivo 'js/meu-arquivo.js', utilizamos a seguinte marcação:

```
1.  <script type="text/javascript"
    src="js/meu-arquivo.js"></script>
```

Qual é o melhor local para colocar a *tag* <script>?

No geral, o melhor local para ser inserida uma *tag* **<script>** é antes do fechamento da *tag* **<body>**. Isso se dá devido ao fato de que o browser, ao encontrar uma *tag* **<script>**, precisa executar o que foi especificado ou

dentro da *tag* ou pelo atributo 'src', bloqueando assim a renderização do restante da página.

Assim, o código JavaScript é executado assim que é interpretado, logo, se existem elementos abaixo do código em questão que são manipulados por esse código (por exemplo, quer-se remover todos os links de uma determinada página e os links encontram-se abaixo da *tag* **<script>**), é necessário adicionar eventos indicando que a página já foi carregada completamente, caso contrário, é bem possível que o código não funcione corretamente.

3. Criando variáveis

Para criar uma variável utiliza-se **var** (opcional) e, para determinar o seu valor, o operador de atribuição (**=**). Para facilitar a compreensão do código, deve-se sempre escolher um nome que identifique o tipo de dado a ser armazenado.

Exemplo:

```
1. <script type="text/javascript">
2.  var nome = "Gabriel Mendonça";
3.  var idade = 25;
4. </script>
```

```
1.  <script type="text/javascript">
```

```
    var nome = prompt('Digite seu nome: ');
    alert(nome + ', seja bem vindo!');
</script>
```

4. Operadores

4.1 Aritméticos

Para as operações matemáticas básicas são utilizados os seguintes, adição(+), subtração(-), multiplicação(*) e divisão(/).

```
//Adição
2+2 //4
2.3+4 //6.3
1.5+1.5 //3

//Subtração
2-2 //0
8-5 //-8
3.2-1 //2.2

//Multiplicação
2*3 //6
1.5*2 //3

//Divisão
1/2 //0.5
1.5/2 //0.75
2/0 //Infinity
```

```
19.
```

Você notou que podemos ter resultados com casas decimais e que é retornada a constante Infinity em qualquer número dividido por zero. Além dos desses operadores básicos temos outros operadores bem úteis:

Resto (%)

Retorna o resto inteiro da divisão.

```
1. 5%4 //1
2. 4%5 //4
```

Incremento (++)

Adiciona um a variável. Se utilizado antes(++x) adiciona um e retorna o valor, caso o operador seja utilizado depois da variável(x++) retorna o valor e adiciona um.

```
1. var x = 1;
2. ++x //2
3. x++ //2
```

Decremento (--)

O comportamento desse operador é parecido com o de incremento(acho que você já entendeu). Ele subtrai um da variável. Se utilizado antes(--x) subtrai um e retorna o valor, caso o operador seja utilizado depois da variável(x--) retorna o valor e subtrai um.

```
1. var x = 2;
2. --x //1
3. x-- //1
```

4.2 Relacionais

De comparação

Igual (==)
Retorna verdadeiro se os valores comparados forem iguais.

```
1. 1=='1' //true
```

Não igual (!=)

Retorna verdadeiro se os valores comparados não forem iguais. Esse operador também pode ser chamado de diferente de.

```
1. 4!=1 //true
```

Igual estrito (===)

Esse operador é mais severo, só retorna verdadeiro se o valor e o tipo comparados forem iguais.

```
1. 3==='3' //false
2. 3===3 //true
```

Não igual estrito (!==)

Não se engane, esse operador vai retornar verdadeiro se os valores e ou os tipos forem diferentes

```
1. 3!=='3' //true
2. 3!==3 //false
3. 3!==4 //true
```

Maior que (>)

Compara se o operador da esquerda é maior que o da direita. Caso seja retorna verdadeiro

```
1. 1>2 //false
2. 5>3 //true
3. 4>'1' //true
```

Maior ou igual que (>=)

Compara se o operador da esquerda é maior ou igual ao da direita. Caso seja retorna verdadeiro

```
1. 1>=2 //false
2. 5>=3 //true
3. 4>='1' //true
4. 2>=2 // true
```

Menor que (<)

Compara se o operador da esquerda é menor que o da direita. Caso seja retorna verdadeiro

```
1. 1<2 //true
2. 5<3 //false
3. 4<'1' //false
```

Menor ou igual que (<=)

Compara se o operador da esquerda é menor ou igual ao da direita. Caso seja retorna verdadeiro

```
1. 1<=2 //true
2. 5<=3 //false
3. 4<='1' //false
4. 2<=2 // true
```

4.3 Lógicos

Operadores lógicos são utilizados normalmente com expressões que retornam verdadeiro ou falso, entretanto caso seja utilizado com valores não booleanos o retorno será não booleano.

Operadores Lógicos: and(e); or(ou); not(não).

5. Funções

Criando e usando funções personalizadas Inicialmente, pense em funções como caixinhas mágicas: você joga coisas dentro, algo acontece e você recebe de volta uma transformação do que foi colocado. Por exemplo: imagine uma caixinha em que você coloca dois números: 3 e 5 e a caixa te devolve 8.

Os números 3 e 5 que colocamos dentro da caixa iremos chamar de **argumentos** e o 8 que recebemos, iremos chamar de **retorno**.

Nem sempre uma função terá argumentos e nem sempre terá um retorno, mas certamente ela irá realizar alguma coisa de nosso interesse, seja mudar a cor de algum elemento, abrir uma janela popup, ou qualquer uma das outras coisas que a linguagem JavaScript pode fazer.

As linguagens de programação já possuem algumas funções pré-definidas para nos ajudar. Da linguagem JavaScript podemos citar: **open()**(usada para abrir janelas popup), **parseInt()** (usada para converter um número ou string em inteiro), **encodeURI()** (codifica uma cadeia de caracteres em um URI válido).

Resolvendo problemas

Sempre que estivermos programando em qualquer linguagem, temos que ter em mente uma coisa muito importante: *"o melhor jeito de resolver um problema grande é dividi-lo em problemas menores"*, cada um desses "problemas menores" serão resolvidos por funções pequenas, assim juntando as várias pequenas funções, teremos resolvido "o todo".

Apesar de ser possível escrever todo o código que resolve o problema grande numa única função gigante, não fazemos isso. Pois isso tornaria muito complexo nosso código, dificultaria uma futura manutenção e impossibilitaria o reaproveitamento de pequenas rotinas. Por isso preferimos dividir e depois criar uma função grande que utilize nossas outras funções pequenas, do que escrever tudo num só lugar.

Declarando

A declaração de uma função é quando construímos a caixa mágica.

```
1. function exemplo () {
2.  //corpo
3. }
```

exemplo é o nome da função e o código entre as chaves {}, é o corpo da nossa função. A palavra **function** é a forma da linguagem JavaScript indicar que estamos **declarando** uma função (criando a caixa mágica).

Invocando

Após construirmos a caixa, ela por si só não faz absolutamente nada até a chamarmos. A invocação consiste em colocar o nome da função seguido pelos parênteses. Isso faz com que o código dentro do corpo da nossa função seja executado.

```
1. exemplo();
```

Experimente salvar um arquivo .html e abrir com o seu navegador, contendo o seguinte código:

```
1. <script>
2. function bar() {
3.  alert('Hello World');
4. }
5. </script>
```

Após abrir o arquivo com qualquer navegador, você irá notar que não irá acontecer absolutamente nada. Sim, nada. Pois ainda não **invocamos** a

função. Apenas a **declaramos**. Para invocar, o arquivo ficaria com o seguinte conteúdo:

```
<script>
// declarando
function bar() {
  alert('Hello World');
}

// invocando
bar();
</script>
```

Exemplo de função com argumentos e retorno

Lembram da caixa mágica que recebia 2 números e devolvia a soma deles ? Essa função ficaria assim:

```
function somar(x, y) {
  return x + y;
}
```

E para invocar essa função **somar()** podemos passar quaisquer dois números:

```
somar(3, 5);
somar(1, 2);
somar(10, 32);
```

Só que não veremos nada, pois não direcionamos o **return** para lugar nenhum.

```
<p id="resultado"></p>
<p id="resultado2"></p>
<script>
alert(somar(3, 5));
document.getElementById('resultado').innerHTML =
somar(1, 2);
document.getElementById('resultado2').innerHTML =
somar(10, 32);
</script>
```

6. Declarações (Statements)

Entendendo Controles de Fluxo e Controles de Repetição

Controles de Fluxo: São comandos da linguagem que permitem desviar o fluxo do programa, dependendo de um teste.

IF

A sintaxe do if é a seguinte:

```
if (<teste>) {
  <código a ser executado caso o teste seja verdadeiro>
}
```

Podemos, por exemplo, executar um trecho do código unicamente se uma variavel nossa for maior do que dez.

```
1. var x = 11;
2. if (x > 10) {
3.  console.log('x é maior do que dez, corram para as colinas!');
4. }
```

Note, que o console.log não apareceria caso o valor de x fosse 10, 9, 8...

```
1. var x = 9;
2. if (x > 10) {
3.  console.log('x é maior do que dez, corram para as colinas!');
4. }
5. console.log('Serei executado independente do if ser true ou false');
```

ELSE

o else não existe sem o if, pois ele não testa nada. Só é executado caso o teste do if retorne falso.

```
1. var x = 9;
2. if (x > 10) {
3.  console.log('x é maior do que dez, corram para as colinas!');
4. } else {
```

```
5.  console.log('Está tudo bem, podemos ficar tranquilos.');
6. }
```

SWITCH

O switch é uma estrutura para testes simples, muito usado quando temos que testar uma mesma condição diversas vezes, pois é mais legível do que uma cadeia de else if.

```
1. var tinta = 'azul';
2. switch (tinta) {
3.  case 'azul':
4.    console.log('Irei pintar o carro de azul');
5.    break;
6.  case 'amarela':
7.    console.log('Vou pintar a casa de amarelo');
8.    break;
9.  case 'verde':
10.      console.log('Vou pintar o chão da garagem de verde');
11.      break;
12.    default:
13.      console.log('Não vou pintar nada');
14.      break;
15.   }
```

Note que para cada uma das cores, eu farei uma coisa completamente diferente da outra. Caso a tinta seja verde, eu vou pintar o chão da garagem, mas se a tinha for amarela, irei pintar a casa.

Se fossemos reescrever esses testes com elseif, ficaria assim:

```
1. var tinta = 'azul';
2.
3. if (tinta === 'azul') {
4.   console.log('Irei pintar o carro de azul');
5. } else if(tinta === 'amarela') {
6.   console.log('Vou pintar a casa de amarelo');
7. } else if (tinta === 'verde') {
8.   console.log('Vou pintar o chão da garagem de verde');
9. } else {
10.     console.log('Não vou pintar nada');
11.    }
```

Laços de repetição (loops)

Se existe uma coisa que os computadores são muito bons é em executar algo várias vezes. Desde que saibamos o que queremos que o computador faça. Felizmente, para não precisamos repetir inúmeras vezes a invocação de uma função ou certo código, existe os loops (laços de repetição).

FOR

Formado por três partes: inicialização, condição e incremento. A sintaxe é:

```
1. for (var i = 0; i <= 10; i++) {
2.   //código a ser executado até a condição se tornar falsa
3. }
```

FOR IN

É utilizado quando não sabemos quantas vezes temos que interar sobre um array ou objeto.

```
1. var arr = [1,2,3];
2. for(var n in arr) {
3.  console.log(n);
4. }
```

FOREACH

Utilizamos o **foreach** quando queremos percorrer as propriedades de um objeto ou os itens de um array, sem precisamos nos preocupar em contar quantos são.

```
1. var arr = [1,2,3];
2. arr.forEach(function(each){
3.  console.log(each);
4. });
```

WHILE

Funciona basicamente igual ao for, e é possível sempre trocar o for() por um while(). Escolhemos um ou outro pela clareza do que estamos fazendo. Geralmente preferimos utilizar o loop for() para interar com contadores e loops while() até que alguma condição mude (de true para false, por exemplo).

```
1. var x = true;
2. while(x) {
3.  console.log('Programando o Futuro');
4.  x = false;
5. }
```

Nesse caso acima, o console.log será executado uma única vez, pois eu altero para false a variavel x, logo na primeira interação do laço. Mas eu poderia ter feito algo assim:

```
1. var i = 1,
2.     x = 2;
3.
4. while(x < 20) {
5.  x = x + (x * i);
6.
7.  console.log('O valor atual de x é: ' + x);
8.  i++;
9. }
```

DO WHILE

Segue o mesmo princípio do while, mas o corpo é sempre executado pelo menos uma vez, independente da condição, pois primeiro ele faz **do** e depois testa a condição.

```
1. do {
2.  console.log('Programando o Futuro');
3. } while(false);
```

Apesar da condição já começar como falsa, veremos a string "Programando o Futuro" uma vez no console do browser.

7. Arrays

Valores agrupados

Com o Array é possível armazenar um conjunto de quaisquer valores javascript, como números, caracteres ou textos ou uma mistura deles. Imagine o array como um gaveteiro onde você pode adicionar ou retirar gavetas e cada gaveta contém o objeto que quiser, vamos criar aqui um gaveteiro onde a primeira gaveta contém o valor 10 a segunda 20 e a terceira 30.

Vejamos:

```
1. var  gaveteiro = [10,20,30];
```

Simples não é? A diferença no Javascript é que a contagem de cada posição do array começa em zero, assim temos: gaveta 0, gaveta 1 e gaveta 2.

Acessando elementos do array

Com o nosso array criado podemos visualizar cada uma das posições individualmente colocando o indice dentro de colchetes:

```
1. console.log(gaveteiro[2]); //30
2. console.log(gaveteiro[1]); //20
3. console.log(gaveteiro[0]); //10
```

Também podemos alterar o valor de cada posição da seguinte forma:

```
1. var  gaveteiro = [10,20,30];
2. gaveteiro[2] = 99;
3. console.log(gaveteiro[2]);
```

Assim dizemos que gaveteiro na posição 2 recebeu o valor 99.

Adicionando elementos no array

Caso precise adicionar uma nova gaveta, podemos usar o método *push*:

```
1. var  gaveteiro = [10,20,30];
2. gaveteiro.push(100);
3. console.log(gaveteiro[3]); //100
```

O método *push* recebe **100** como parametro e adiciona na ultima posição do array.

Removendo elementos no array

Caso precise remover/recortar uma gaveta, podemos usar os seguintes métodos:

- Para remover a ultima gaveta, utilizamos o *pop*:

```
1. var  gaveteiro = [10,20,30];
```

```
2. console.log(gaveteiro[2]); //30
3. gaveteiro.pop();
4. console.log(gaveteiro[2]); //undefined
```

- Para remover a primeira gaveta, utilizamos o *shift*:

```
1. var  gaveteiro = [10,20,30];
2. console.log(gaveteiro[0]); //10
3. gaveteiro.shift();
4. console.log(gaveteiro[0]); //20
```

- Para retornar apenas algumas gavetas (recortar), utilizamos o *slice*:

```
1. var  gaveteiro = [10,20,30];
2. var novaGaveta = gaveteiro.slice(1,3);
3. console.log(novoGaveteiro); //[20, 30]
```

Quantidade de elementos do array

Depois de ter adicionado várias gavetas, pode surgir a necessidade de saber quantas já existem, para isso vamos acessar a propridade *length*:

```
1. var  gaveteiro = [1,2,3,10,20,30];
2. console.log(gaveteiro.length); //6
3. gaveteiro.push(100);
4. gaveteiro.push(200);
5. gaveteiro.push(300);
6. gaveteiro.push(400);
7. console.log(gaveteiro.length); //10
8. gaveteiro.push(200);
9. console.log(gaveteiro.length); //11
```

Fonte: https://tableless.github.io/iniciantes/manual/js/index.html