Disciplina: Programação Cliente-servidor

Aula 2: JavaScript – Parte 1

Apresentação

As páginas HTML foram se tornando cada vez mais complexas. Ao invés das páginas simplesmente informativas do início da Web, hoje temos páginas extremamente interativas e conceitos como RIA (Rich Internet Application). Neste sentido, acaba se tornando necessário um ferramental que traga maior flexibilidade às páginas em termos funcionais, e não apenas visuais.

Veremos nesta aula que o uso da linguagem JavaScript irá nos permitir ampliar as funcionalidades básicas de nossas páginas Web, já que é uma linguagem tão bem aceita que passou a ser utilizada em outras áreas além do controle do browser, como na programação de dispositivos móveis e servidores Web.

Objetivos

- Identificar as características gerais e histórico do JavaScript;
- Explicar a sintaxe básica do JavaScript;
- Usar o JavaScript para a criação de funções e bibliotecas.

Origens e características

A linguagem **JavaScript** nem sempre teve este nome. Desenvolvida originalmente pela Netscape, ela se chamava **Mocha**, tendo o nome modificado posteriormente para LiveScript, quando ocorreu o lançamento da mesma junto ao navegador Netscape 2.0 versão beta, em setembro de **1995**.

Em dezembro de 1995, em um anúncio conjunto com a Sun Microsystems, é lançado o **Netscape 2.0B3**, com suporte à tecnologia de Applets, sendo o nome da linguagem modificado para JavaScript, o que causa muita confusão até hoje no que tange à sua relação com a linguagem Java.

Atenção

É importante que tenhamos em mente que JavaScript não é Java, e a similaridade entre as duas linguagens ocorre apenas pelo fato de ambas utilizarem sintaxes baseadas na linguagem C.

Utilizamos o **JavaScript para controle de elementos da página HTML e viabilização da interatividade da mesma**, caracterizando-se originalmente como uma tecnologia cliente, podendo ser embebida na própria página, ou organizada no formato de biblioteca, como arquivo externo.

Recentemente, o JavaScript passou a ser utilizado também do lado servidor, através de tecnologias como o node.js.

Outro ambiente que recebeu a possibilidade de desenvolvimento com JavaScript foi o dos dispositivos móveis, com uso de ferramentas como Ionic.

Sintaxe

Inicialmente devemos compreender a declaração de variáveis e operações aritméticas da linguagem.

Uma variável pode ser declarada com o uso de var, ou simplesmente com a inicialização da mesma. Como o JavaScript não é fortemente tipado, a variável assume o tipo do valor associado a ela.

Os tipos com os quais a linguagem trabalha são:

- Inteiro;
- Ponto flutuante;
- Booleano;
- Texto;
- Objeto; e
- Vetor (como objeto da classe Array).

Exemplo

No exemplo a seguir, são declaradas e inicializadas as variáveis **a** e **b**, sendo impressas na página algumas operações sobre as mesmas através do comando **document.writeln**.
É interessante observar que este comando escreve sobre a página HTML, permitindo a inclusão de tags, como o uso de **
** para quebra de linha.

```
<html>
   <body>
     <script>
       var a = 10, b = 5;
       // Algumas operações básicas...
       document.writeln(
"Soma: "+(a+b));
       document.writeln(
"Subtração: "+(a-b));
       document.writeln(
"Multiplicação: "+(a*b));
       document.writeln
("Divisão: "+(a/b));
       document.writeln(
"Teste de igualdade: "+(a==b));
      <</script>
    <</body>
</html>
```

Soma: 15

Subtração: 5

multiplicação: 50

Divisão: 2

Teste de igualdade: false

Dica

Lembre-se sempre de comentar seu código para que você e outros programadores possam revisá-lo com maior facilidade em adaptações posteriores.
O JavaScript aceita dois tipos de comentários:

- Comentário de linha, com uso de //;
- Comentário longo, iniciado com /* e finalizado com */.

Operadores aritméticos utilizados no JavaScript

Operador	Operação
+	Soma (concatenação para texto)
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão inteira
++	Incremento de 1
--	Decremento de 1

Os operadores relacionais permitem a comparação entre valores, tendo como resultado um valor verdadeiro (true) ou falso (false), o que pode ser armazenado em uma variável booleana.

Podemos observá-los no quadro:

Operador	Operação
==	Compara a igualdade entre os termos
!=	Compara a diferença entre os termos
>	Compara se o primeiro é maior que o segundo
<	Compara se o primeiro é menor que o segundo
>=	Compara se o valor é maior ou igual
<=	Compara se o valor é maior ou igual

Nós também podemos combinar valores booleanos com o uso de operadores lógicos expressos pela tabela-verdade a seguir:

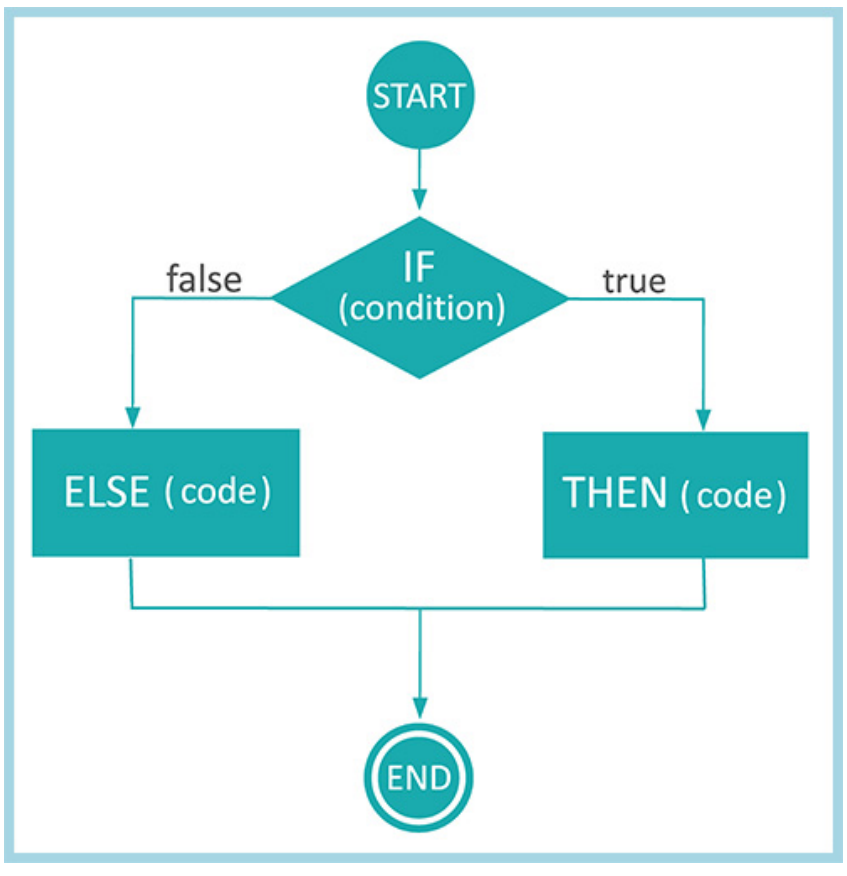
A (Booleano 1)	B (Booleano 2)	A && B - AND	A || B - OR	!A - NOT
false	false	false	false	true
false	false	true	true	true
true	false	true	true	false
true	true	true	true	false

Estruturas de decisão

Estando definidas as variáveis e métodos de saída, o nosso próximo passo será a compreensão das estruturas de decisão existentes na sintaxe do JavaScript.

O fluxo de execução sequencial indica que as diversas instruções serão executadas na ordem em que são apresentadas no código, mas existem formas de redirecionar o fluxo de execução para partes específicas.

É comum encontrarmos situações onde efetuamos determinadas ações apenas perante determinadas condições. Para isso, em termos de algoritmos, contamos com as estruturas de decisão.



Exemplo

Um exemplo simples pode ser observado a seguir:
SE fizer sol **ENTÃO** irei à praia **SENÃO** irei ao cinema.

Note que a condição de "fazer sol ou não" pode ser interpretada como um valor booleano, sendo a decisão tomada a partir desta condição.

Outro tipo de decisão é a que tomamos perante um conjunto de opções, algo como:

SELECIONE o dia da semana:

- **CASO SEJA** segunda-feira **FAÇA**: passar no escritório;
- **CASO SEJA** terça-feira **OU** quarta-feira **FAÇA**: visitar os clientes;
- **CASO SEJA** quinta-feira **FAÇA**: fechar relatórios;
- **SENÃO**: ficar em casa.

Neste exemplo, as opções de segunda-feira até quinta-feira apresentam ações específicas, enquanto os demais dias, que não foram explicitados, executam a ação padrão a partir do comando **SENÃO**.

Em ambos os casos, é fácil de observar os desvios no fluxo de execução, pois diversos trechos serão executados apenas sob condições específicas.

Estrutura if..else

Na sintaxe do JavaScript, a instrução **SE..ENTÃO** é expressa como **if(<<condição>>)**, podendo se referir a um único comando, ou um bloco de comandos delimitado pelo uso de chaves.

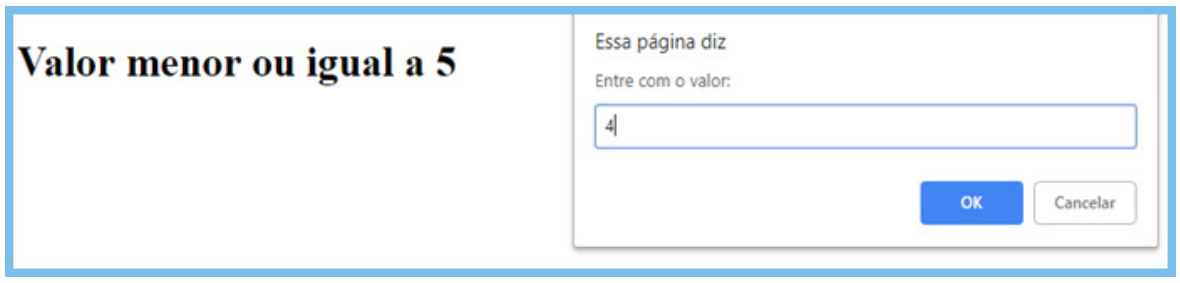
O uso da instrução **SENÃO** (**else**) é opcional, e também pode ser aplicado a um ou mais comandos, segundo as mesmas regras de escrita do **if**.

Exemplo

Podemos ver, no exemplo abaixo, a aplicação da estrutura if..else.

Aqui, a variável x recebe o valor digitado pelo usuário com o uso da função prompt, mas com o valor, que era originalmente texto, convertido para número com o uso de eval.

```
<!DOCTYPE html>
<html>
   <body>
      <script>
         var x = eval(prompt("Entre com o valor:",""));
         if(x > 5)
             document.writeln("<h1>Valor maior que 5</h1>");
         else
             document.writeln("<h1>Valor menor ou igual a 5</h1>");
      </script>
   </body>
</html>
```



Estrutura switch..case

Para o comando **SELECIONE** devemos utilizar **switch(<<variável>>)** no JavaScript.

Este comando irá desviar o fluxo de execução para os comandos case, de acordo com o valor da variável, sendo que o comando **default** será executado caso o valor não esteja entre aqueles que foram previstos.

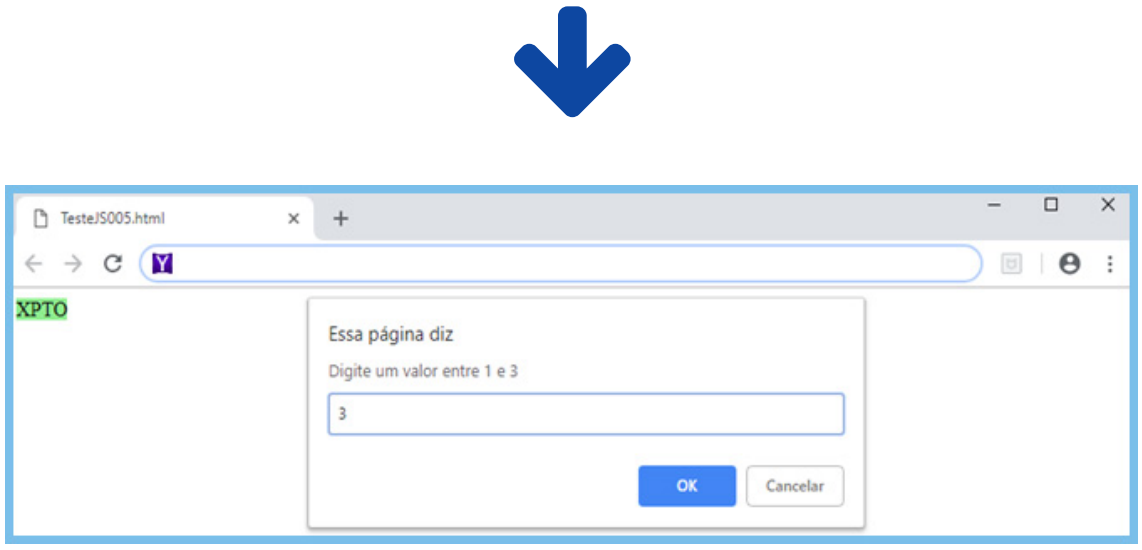
Devemos observar que cada seção case deve ser terminada com o comando **break**.

Exemplo

O exemplo a seguir ilustra a utilização da estrutura **switch..case**.

Aqui, a página solicitará um valor entre 1 e 3, e, de acordo com o valor utilizado, será definida uma cor de fundo diferente para a palavra “XPTO”, sendo assumido fundo preto para opções que não forem previstas.

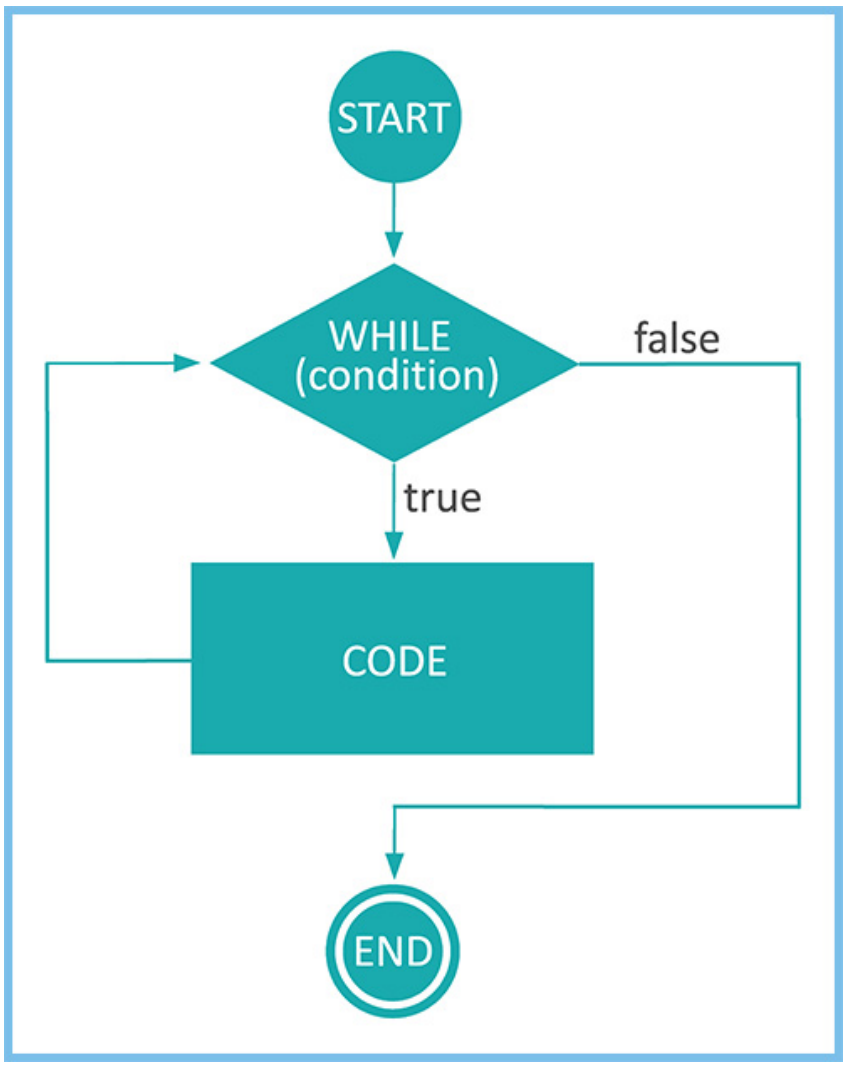
```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8"/>
   </head>
   <body>
      <script>
         var x1 = eval(prompt("Digite um valor entre 1 e 3",""));
         var cor;
         switch(x1){
            case 1:
            cor = "yellow";
            break;
            case 2:
            cor = "lightblue";
            break;
            case 3:
            cor = "lightgreen";
            break;
            default:
            cor = "black";
            }
            document.writeln("<span style='background-color:"+cor+"'>XPTO<span>");
      </script>
   </body>
</html>
```



Estruturas de repetição

Outra forma e redirecionar o fluxo de execução é através do uso de estruturas de repetição.

Elas servem para repetir a execução de um comando ou bloco de comandos enquanto determinada condição for verdadeira.



A primeira estrutura de repetição que iremos analisar é denominada **PARA..FAÇA**. Ela permite repetir um determinado bloco de comandos para cada valor assumido por uma determinada variável dentro de uma faixa pré-especificada.

Podemos observar um exemplo simples, a seguir:

```
PARA X DE 1 A 10 FAÇA:
INÍCIO
ESCREVA ( X )
FIM
```

Neste exemplo, a variável x assumirá os valores de 1 até 10, sendo impresso o valor da mesma a cada rodada através do comando escreva. Em termos práticos serão impressos os valores de 1 a 10.

O bloco de execução é especificado pelas palavras “INÍCIO” e “FIM”, e a presença das mesmas não seria necessária, neste caso, por envolver apenas uma instrução.

Outra estrutura de repetição de grande utilização é denominada **ENQUANTO..FAÇA**, e ela permite repetir um bloco de comandos enquanto determinada condição for verdadeira, como no exemplo a seguir.

```
X = 1
ENQUANTO X < 11 FAÇA
INÍCIO
ESCREVA ( X )
X = X + 1
FIM
```

Teremos aqui o mesmo efeito do exemplo de PARA..FAÇA, ou seja, serão impressos os números de 1 a 10, no entanto, a forma de construir é diferente. A variável X é inicializada com o valor 1, e a estrutura ENQUANTO..FAÇA repetirá a execução do bloco enquanto esta variável tiver valor menor do que 11.

Note a necessidade de incrementar o valor de X no bloco de execução, caso contrário o loop nunca acabaria.

Caso o valor de X tivesse sido inicializado com qualquer valor maior do que 10, não ocorreria a execução do bloco nenhuma vez, o que diferencia a estrutura **ENQUANTO..FAÇA**, onde o teste é feito na entrada, da estrutura **FAÇA..ENQUANTO**, onde o teste é feito na saída.

Reescrevendo o exemplo anterior teríamos:

```
X = 1
FAÇA
INÍCIO
ESCREVA ( X )
X = X + 1
FIM
ENQUANTO X < 11
```

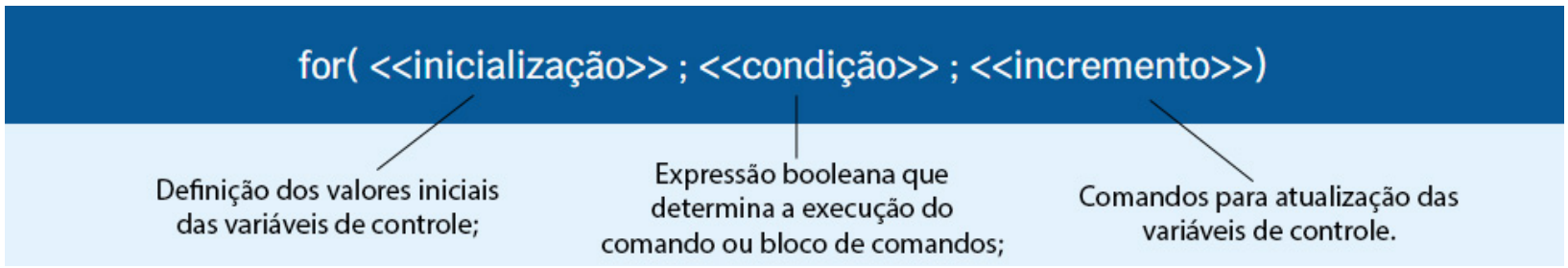
Com este novo formato, se X fosse inicializado com o valor 12, por exemplo, este valor seria impresso, pois o teste é feito apenas na saída da estrutura de controle.

Estruturas de controle do tipo FAÇA..ENQUANTO são interessantes quando queremos obrigar a execução do bloco pelo menos uma vez, sendo bastante utilizadas na criação de menus para interatividade, como:

```
FAÇA
  INÍCIO
    ESCREVA ( “DESEJA CADASTRAR NOVO CLIENTE OU
SAIR?” )
    LEIA ( OPCAO )
    SE OPCAO != “SAIR” ENTÃO
      INÍCIO
      // COMANDOS DIVERSOS
    FIM
  FIM
ENQUANTO OPCAO != “SAIR”
```

Estrutura for

A implementação de estruturas do tipo **PARA..FAÇA** utiliza a seguinte sintaxe:



Por exemplo, um loop de 1 a 5 seria escrito como **for(i=1 ; i<=5 ; i++)**.

No código seguinte podemos observar um loop para imprimir os números de 1 a 10, classificando-os como pares ou ímpares.

```
<!DOCTYPE html>
<html>
   <body>
      <ul>
         <script>
            for(i=1 ; i<=10 ; i++){
            tipo = ( i%2==0 ) ? "par" : "impar";
            document.writeln("<li>O numero "+i+" e "+tipo+"</li>");
         }
         </script>
      </ul>
   </body>
</html>
```

- O numero 1 e impar
- O numero 2 e par
- O numero 3 e impar
- O numero 4 e par
- O numero 5 e impar
- O numero 6 e par
- O numero 7 e impar
- O numero 8 e par
- O numero 9 e impar
- O numero 10 e par

Neste código podemos observar, também, a utilização de um operador de decisão para escolher entre “par” ou “ímpar”.

Um operador de decisão segue a sintaxe abaixo:

variável = (<< condição >>) ? valor para true : valor para false

Baseado nisto, podemos dizer que, se a condição (i%2==0) for verdadeira, tipo recebe “par”, senão recebe “impar”.

Isso equivale à seguinte estrutura de decisão:

```
if ( i%2==0 )
    tipo = "par";
else
    tipo = "impar";
```

Podemos utilizar o operador de decisão para a simplificação da escrita nas situações em que uma determinada variável recebe dois valores alternativos perante uma determinada condição.

Estruturas while e do..while

ENQUANTO..FAÇA

A implementação de estruturas do tipo **ENQUANTO..FAÇA** utiliza a seguinte sintaxe:

```
while( <<condição>> ) {
    // Bloco de Comandos
}
```

Condição – Expressão booleana que determina a execução do comando ou bloco de comandos.

FAÇA..ENQUANTO

Com relação às estruturas do tipo **FAÇA..ENQUANTO**, a sintaxe utilizada seria:

```
do {
    // Bloco de Comandos
} while( <<condição>> );
```

Condição – Expressão booleana que determina a continuidade da execução do bloco de comandos.

Exemplo

Clique em Exemplo <galeria/aula2/anexo/exemplo.pdf> e observe a utilização do comando while.

Funções e bibliotecas

Em muitas situações, precisamos efetuar uma determinada sequência de comandos repetidas vezes em diferentes contextos. Toda vez que replicamos código estamos aumentando o tamanho do mesmo e dificultando a manutenção.

Por exemplo, se um erro é detectado nesta sequência de comandos, ocorrerá a necessidade de procurar todas as diversas ocorrências da mesma e alterar cada uma delas.

Uma solução para isso é o uso de uma função, que é a denominação de um processo englobando uma sequência de comandos, e que recebe um nome e a possibilidade de parâmetros de entrada e retorno de valor.

Já utilizamos algumas funções nativas do JavaScript em outros exemplos, como **prompt** e **eval**, e agora utilizaremos mais uma.

A função **confirm** tem por objetivo efetuar uma pergunta, retornando **true** para o caso do clique em **OK** e **false** para **Cancela**.

Exemplo

```
<script>
   function meuProcesso(){
      a = eval(prompt("Entre com o valor:",""));
      b = a * 5;
      document.writeln("<br/>"+a+" * 5 = "+b);
   }

   meuProcesso();
   z = 3;
   while(z>0) {
      meuProcesso();
      z--;
   }
   if(confirm("Mais uma vez?")) {
      meuProcesso();
   }
</script>
```

Neste exemplo, um mesmo procedimento é repetido diversas vezes, apenas alterando a nomenclatura das variáveis.

Se a multiplicação fosse alterada de 5 para 4, ocorreria um problema de manutenção bastante relevante, mesmo para um código pequeno como esse.

A solução é transformar o processo repetitivo em uma função, como no código abaixo:

```
<script>
   function meuProcesso(){
   a = eval(prompt("Entre com o valor:",""));
   b = a * 5;
   document.writeln("<br/>"+a+" * 5 = "+b);
   }
   meuProcesso();
   z = 3;
   while(z>0) {
      meuProcesso();
      z--;
   }
   if(confirm("Mais uma vez?")) {
      meuProcesso();
   }
</script>
```

Note como o código ficou mais legível com a definição da função. Além disso, as tarefas de manutenção são muito facilitadas, pois a alteração do corpo da função irá impactar de forma automática em todas as suas utilizações no decorrer do código.

O formato geral de uma função, no JavaScript, seria:

```
function NOME (PARÂMETRO1, PARÂMETRO2 ... PARÂMETRON) {
      [COMANDOS]
      return VALOR;
}
```

Tanto os parâmetros quanto o valor de retorno são opcionais na definição de uma função.

Poderíamos exemplificar uma função para o cálculo da hipotenusa de um triângulo retângulo a partir do fornecimento dos valores de seus catetos, baseada no Teorema de Pitágoras.

```
function pitagoras(cateto1, cateto2){
return Math.sqrt(Math.pow(cateto1,2) + Math.pow(cateto2,2));
}
```

Neste exemplo, são utilizados **Math.sqrt** para cálculo da raiz quadrada e **Math.pow** para elevar um valor a uma potência, no caso ao quadrado.

Como as funções visam a reutilização, é muito comum organizá-las em grupos denominados bibliotecas, normalmente guardadas em arquivos texto com a extensão "**js**".

Diversas bibliotecas JavaScript encontram-se disponíveis e podemos também criar as nossas.

Para o próximo exemplo, utilizaremos dois arquivos, sendo um denominado "**cores.js**" e outro "**testeCores.html**".

Incialmente, vamos observar o código de "**cores.js**":

```
var letras = new Array("0","1","2","3","4","5","6","7","8","9",
"A","B","C","D","E","F");
function getHexa(inteiro){
      resto = inteiro % 16;
      quociente = (inteiro - resto) / 16;
      return letras[quociente] + letras[resto];
}
function getCor(r,g,b){
      return "#" + getHexa(r) + getHexa(g) + getHexa(b);
}
```

Esta biblioteca visa efetuar a conversão dos números inteiros entre 0 e 255 para hexadecimal, e a formação de códigos de cores no padrão RGB a partir de 3 inteiros na faixa citada.

Incialmente, com o uso de **Array**, é definido um vetor de elementos texto, onde cada posição representa um número de 0 a 15 no formato hexadecimal, como **cores[11]** equivalendo a "**B**".

Em seguida, definimos a **função getHexa**, que recebe um valor inteiro como parâmetro, e efetua a conversão para hexadecimal.

Esta conversão é efetuada com a divisão inteira por 16, sendo o quociente utilizado para alta ordem do hexadecimal e o resto para baixa ordem.

Por exemplo, se dividirmos **222** por **16**, teremos quociente **13** e resto **14**, o que retornará o valor "**DE**" em hexadecimal.

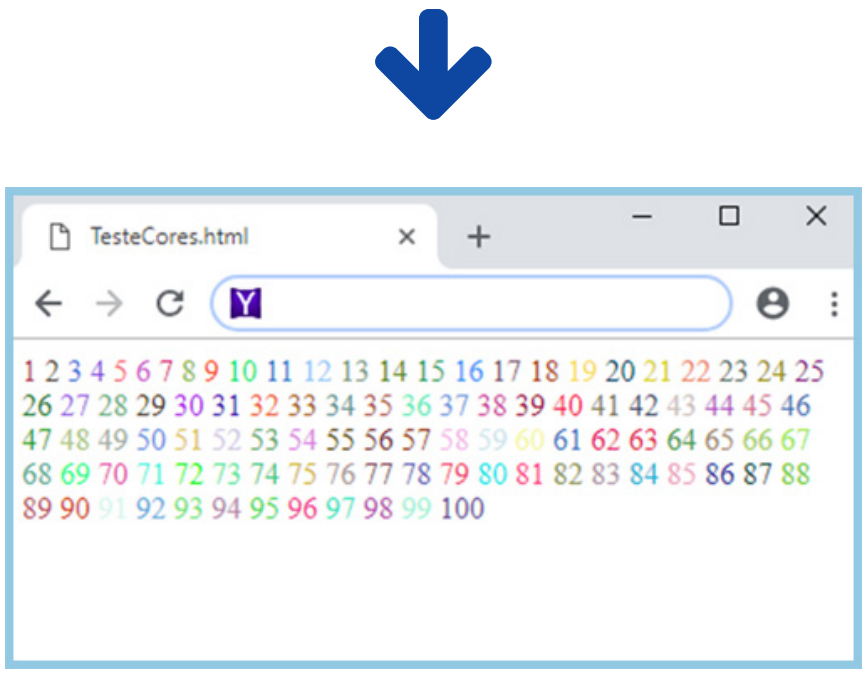
A última função se chama **getCor**, e recebe três parâmetros denominados r, g e b, os quais correspondem aos componentes vermelho, verde e azul da composição da cor.

No corpo desta função podemos observar a concatenação do símbolo "#" com as chamadas de **getHexa** para os três parâmetros sucessivamente.

Se efetuarmos uma chamada **getCor(222,220,224)** teremos como retorno o valor "**#DEDCE0**".

Agora que construímos nossa biblioteca para tratamento de cores, podemos utilizá-la em uma página HTML, a qual recebeu o nome de "testeCores.html", cujo código e saída podem ser observados a seguir:

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8"/>
      <script src="cores.js"></script>
   </head>
   <body>
      <script>
         for(i=1;i<=100;i++){
         red = Math.trunc(Math.random()*255);
         green = Math.trunc(Math.random()*255);
         blue = Math.trunc(Math.random()*255);
         document.writeln("<span style='color:"+getCor(red,green,blue)+"'<");
         document.writeln(i + "</span>");
}
      </script>
   </body>
</html>
```



No código da página, podemos observar um loop de 1 a 100, com uso de for, e internamente o uso de **Math.trunc**, que faz a aproximação inteira, e **Math.random**, para obter um número aleatório entre 0 e 1, segundo uma distribuição uniforme.

Ao multiplicar o resultado de **Math.random** por 255, obtemos um número aleatório entre 0 e 255, porém no formato de ponto flutuante, o que é resolvido com a transformação em inteiro com **Math.trunc**, viabilizando a chamada de **getCor**.

A cada rodada do loop, é escrita uma tag ****, tendo como conteúdo o valor de **i** corrente, e sendo formatada a cor da fonte com uso do atributo **style**.

Como os números para red, green e blue são randômicos, a cada vez que atualizamos a página, os números serão expostos com cores diferentes.

Para as operações matemáticas fazemos grande uso de Math.

Método ou Atributo	Descrição
sqrt(x)	Cálculo da raiz quadrada de x
trunc(x)	Trunca o valor x, retornando apenas a parte inteira
pow(x,y)	Retorna o resultado de x elevado a y
random( )	Retorna um valor aleatório entre 0 e 1
PI	Constante com o valor de PI
max(x1,x2,...,xn)	Retorna o maior valor da sequência
min(x1,x2,...,xn)	Retorna o menor valor da sequência
sin(x)	Calcula o seno de um ângulo x em radianos
cos(x)	Calcula o cosseno de um ângulo x em radianos
abs(x)	Retorna o valor absoluto de x

Saiba mais

Os outros métodos e atributos de Math podem ser observados em w3schools.com <https://www.w3schools.com/js/js_math.asp> .

Recursividade

Quando falamos sobre recursividade, estamos nos referindo a funções que chamam a si mesmas, repetidamente, para a solução de determinado problema.

Talvez o exemplo mais simples de recursividade seja o cálculo do fatorial de um número.

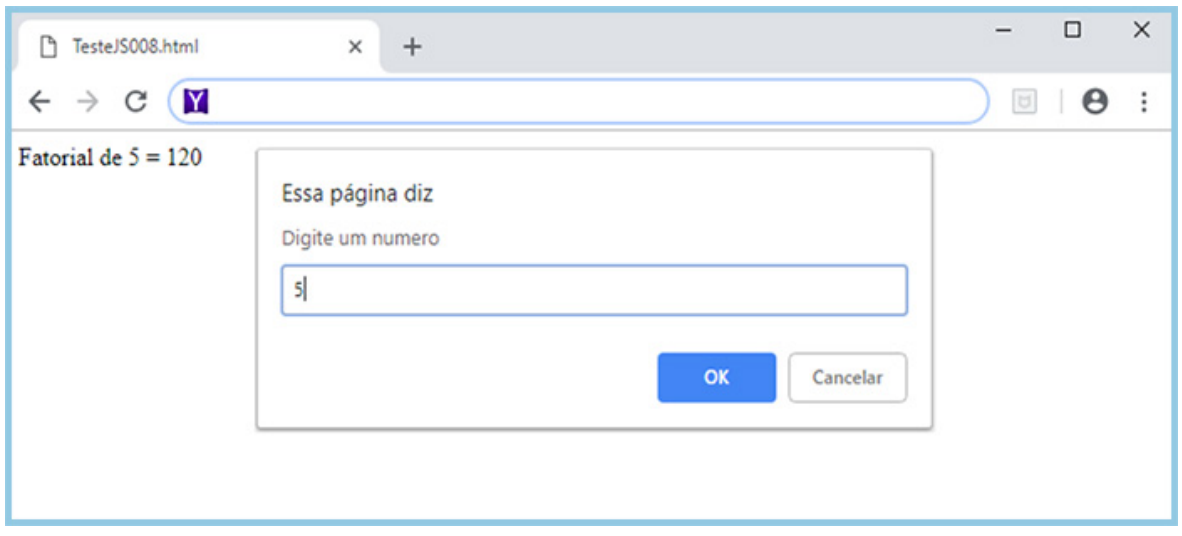
Por exemplo, $4! = 4 * 3 * 2 * 1$, $3! = 3 * 2 * 1$, $2! = 2 * 1$ e $1! = 1$.

Podemos observar que 4! = 4 * 3! ou que 3! = 3 * 2! e, a partir disso, definir uma regra de recursividade.

Toda função recursiva chama a si mesma, e tem uma regra de parada para que não ocorra uma execução indefinida. Neste caso, a regra seria “quando chegar a 1 retorne 1”.

A seguir, podemos observar um exemplo de implementação da função fatorial de forma recursiva.

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8"/>
      <script src="cores.js"></script>
   </head>
   <body>
      <script>
         function fatorial(n){
            if(n>1)
            return n * fatorial(n-1);
         else
            return 1;
         }
         x = eval(prompt("Digite um numero",""));
         document.writeln("Fatorial de "+x+" = "+fatorial(x));
      </script>
   </body>
</html>
```



Atividade

1 - Marque Verdadeiro ou Falso para as opções sobre as características gerais da linguagem JavaScript:

a) A sintaxe JavaScript diferencia maiúsculas e minúsculas.

b) JavaScript é fortemente tipado.

c) Foi desenvolvida originalmente pela NetScape com o nome de Mocha.

d) Hoje é possível desenvolver para dispositivos móveis com JavaScript.

e) JavaScript foi criado como uma versão script da linguagem Java.

2 - Você começou a desenvolver a página de um novo cliente e ele solicitou que uma das páginas confirmasse uma sequência de caracteres digitados pelo usuário de acordo com uma imagem da tela antes de apresentar o conteúdo, técnica conhecida como CAPTCHA. Considerando que o usuário terá que digitar ao menos uma vez, e repetir até acertar, qual estrutura de controle de fluxo deverá ser utilizada?

a) if..else
b) while
c) do..while
d) switch..case
e) for

---

3 - Implemente uma função em JavaScript para efetuar o cálculo do Imposto sobre a Renda em um determinado país, considerando o desconto por faixa, de acordo com a tabela seguinte.

Faixa	Alíquota
R$0,00 a R$1.500,00	Isento
R$1.500,01 a R$2.500,00	10%
R$2.501,00 a R$4.000,00	20%
R$4.000,01 ou acima	30%

Gabarito comentado

Referências

CASSATI, J. P. **Programação Cliente em Sistemas Web**. Rio de Janeiro: Estácio, 2016.

DEITEL, P; DEITEL, H. Ajax, **Rich Internet Applications e Desenvolvimento Web para Programadores**. São Paulo: Pearson Education, 2009.

PLOTZE, R. **Tecnologias Web**. Rio de Janeiro: Estácio, 2016.

SANTOS, F. **Tecnologias para Internet II**. 1. ed. Rio de Janeiro: Estácio, 2017.

Próxima aula

- Elementos de interatividade com a página;
- JavaScript para a validação de formulários;
- A sintaxe JavaScript para Orientação a Objetos.

Explore mais

Visite as páginas sugeridas a seguir e saiba mais sobre o conteúdo estudado nesta aula:

- Básico de JavaScript <https://developer.mozilla.org/pt-BR/docs/Aprender/JavaScript>
- Uso de JavaScript com DOM <https://www.w3schools.com/js/js_htmldom.asp>
- Tutorial de JavaScript <https://www.w3schools.com/js/>
- Document Object Model <https://en.wikipedia.org/wiki/Document_Object_Model>