Aula 12: Funções

Nesta aula explicaremos o que são e como usar funções nos seus programas em JavaScript. Você aprenderá como elas podem receber e retornar valores à estrutura que as acionou. Entenderá como funciona a visibilidade das variáveis dependendo do ponto em que são criadas. E conhecerá as formas de ativar as funções.

Objetivos:
Aprender:
- Funções sem parâmetros
- Funções com parâmetros
- Comando `return`
- Funções com retorno de valor
- Ativação de funções a partir de um hiperlink
- Escopo de variáveis (locais x globais)

Pré-requisitos:
Todas as aulas anteriores deste módulo.

1. Aproveitando Códigos no Programa

Já comentamos que um programa na linguagem JavaScript é uma coleção de comandos para manipular variáveis e constantes. Usando funções, você poderá ter partes de código definidos uma vez e executados ou invocados muitas vezes pelo programa. Além desta vantagem de economia de código, há outras proporcionadas pela modularidade, pela possibilidade de divisão de trabalho, pelo reaproveitamento de software que elas proporcionam aos seus programas. Sem contar que JavaScript, pela flexibilidade oferecida quanto ao tipo de dados, lhe dá a possibilidade de ter o mesmo código executado com diferentes dados.

1.1. Funções

Uma função é uma seqüência de comandos, realizando uma tarefa específica, a qual se atribui um nome. Em JavaScript, elas sempre são identificadas por um nome, podem ou não receber parâmetros e também podem ou não retornar um valor.

A linguagem apresenta diversas funções já predefinidas como as já vistas em aulas anteriores. Funções feitas pelo programador, depois de definidas, podem ser usadas exatamente da mesma maneira que as já disponíveis na linguagem.

Quando se inclui o **nome da função** no programa diz-se que estamos fazendo uma **chamada à função** (ou **invocação** à função). Quando o programa é **executado,** isso produz um **desvio** no seu curso para executar os comandos da função. Depois que toda a função é executada, o fluxo de execução **retorna** para a instrução seguinte ao ponto onde foi incluído o nome da função.

A função **`writeln`** é idêntica a **`write`** em todos os aspectos, exceto que ela inclui uma mudança de linha depois de escrever seus argumentos. Mas, como HTML ignora mudanças de linha, usualmente, essa característica só será conveniente com documentos ou trechos não HTML. Nestes casos, a única diferença entre ambas será que `writeln` deixará um espaço em branco do tamanho de um caracter entre os textos escritos.

A **definição de uma função** em JavaScript usa a palavra-chave `function` e segue a seguinte sintaxe mínima:

```
function nome()
{
    ... comandos ...
}
```

Uma vez que foi definida, a função pode ser **chamada** ou **invocada** pelo seu nome, seguido de parênteses.

```
    ...
nome();
    ...
```

As linhas seguintes são exemplos de definição e de utilização de duas funções:

```
function abreTabela()
{
    document.writeln("<TABLE border=2>");
}
function fechaTabela()
{
    document.writeln("</TABLE>");
}
```

O interior ou corpo da função pode ser composto por qualquer número de comandos, que devem ser sempre contidos pelas chaves. As chaves sempre fazem parte da função e diferentemente dos outros comandos (como `if`, `while`, `for` etc) que também as usam, eles são indispensáveis mesmo que ela se constitua de um único comando.

Onde você quiser usá-las, elas são chamada por:

```
abreTabela();
...
fechaTabela();
```

Uma função pode receber **parâmetros** (também chamados **argumentos**) que influem na sua execução. A "passagem de parâmetros" é feita dentro do ( ) que segue ao nome da função. Quando há mais de um parâmetro ou argumento eles aparecem separados por vírgulas. Por exemplo, a linha que segue define uma com 3 argumentos:

```
function xpto (p1, p2, p3)
{
    comandos;
  }
```

Cuja utilização seria:

```
x = 20;
xpto (100, "bobo", x);
```

A "passagem de parâmetros" é "por valor", isto é, a função recebe do programa o valor do dado (e não o dado como variável). A implicação disso é que o valor do parâmetro pode ser modificado à vontade sem que a variável original seja alterada. Veja esses outros exemplos:

```
function abreTabela (cor, borda)
{
    document.writeln ("<TABLE bgcolor=",
    cor," border=", borda, ">");
}
```

A passagem de parâmetros na invocação de uma função tem o efeito de uma atribuição de variáveis. Assim, quando a função acima for chamada pela linha:

```
abreTabela ("white", 2);
```

`cor` receberá o valor `"white"` e `borda` o valor `2`.

Como JavaScript é uma linguagem **não tipada,** ela não espera que o dado enviado seja de um tipo pré-definido, e também não faz qualquer verificação quanto ao tipo de dado ser ou não aquele esperado pela função. Se isso for importante, o próprio programador deve verificar o tipo de dado antes de usá-lo. Também não é verificado se o número de parâmetros enviado é o que a função espera. Na chamada à mesma função anterior se for usada a linha que segue:

```
abreTabela (2, 4,"oi");
```

`cor` receberá o valor `2`, `borda` o valor `4`, e o terceiro valor será ignorado, já que não há um terceiro parâmetro na definição da função.

Se forem passados menos parâmetros que o esperado, é atribuído o valor `undefined` aos valores que faltam. Isso, em alguns casos, pode causar mau funcionamento da função.

1.2. Comando `return`

A inclusão de um comando `return` em uma função, faz com que sua execução seja interrompida e que o programa volte ao ponto onde ela foi chamada. Pode-se dizer que o comando `return` está para uma função assim como o `break` está para um laço. Observe o exemplo:

```
function montaLista( )
{
  while (true)
  {
    num=window.prompt("Digite um"+
    "número:");
    if (parseInt(num) == 0) return;
    document.write("<LI>", num);
  }
}
```

O comando `return` só pode ser utilizado no corpo de uma função, ocorrendo um erro de sintaxe se for utilizado no programa principal.

Em algumas linguagens, como por exemplo Pascal, existe uma distinção entre funções que retornam e que não retornam valores (em Pascal denominadas *functions* e *procedures*). Em JavaScript não há esta distinção e os dois tipos de funções são declaradas da mesma maneira.

Quando uma função retorna um valor, sua chamada pode ser incluída no meio de uma expressão. O valor retornado após a execução será utilizado no cálculo desta expressão.

A forma de fazer uma função retornar um valor é utilizar o comando `return` seguido de um valor ou de uma expressão. Se for seguido de uma expressão, esta será avaliada antes de retornar da função para que o valor resultante possa ser retornado.

Assim definindo a função `quadrado` como abaixo:

```
function quadrado(x)
{   return x * x;   }
```

é possível seu uso de diversos modos no programa como:

```
if (quadrado(x) > 100) //Numa condição
    y=quadrado(x);     //Numa atribuição
```

Também é possível, ao invocar a função, usar na passagem de parâmetros. Neste caso, a expressão é avaliada e o resultado é que é usado como argumento na função. O valor dos parâmetros são apenas definidos enquanto a função estiver sendo executada:

```
x=2;
y=quadrado(x*2+5);        //y=81, x=2
y=quadrado(quadrado(x));
```

Se no entanto a função executar o comando `return`, sem nenhum valor ou expressão associado a ele, o valor associado à chamada da função é indefinido. O mesmo acontece se o retorno ocorrer quando ela chega ao fim da declaração de seu corpo. Por exemplo, as linhas abaixo fariam que fosse impresso "`2 undefined`"

```
var x=2;
function quadrado(x)
{
  x=x * x;
  return;
}
y=quadrado(x);//nao traz o resultado
document.writeln(x," ",y);
```

Como foi feito o `return` sem nenhum valor associado e, dentro da função, toda referência a x é feita em relação à variável local (o parâmetro), logo a função do código acima não tem efeito algum.

1.3. Ativação de Funções a partir de um Hiperlink

É possível associar uma função à seleção de um link, como no exemplo:

```
<script>
function clicou()
{
   window.alert("Ei você me cutucou!!!");
}
</script>
<A href="javascript:clicou()">Não clique!
</A>
```

2. Escopo de Variáveis

Quando uma variável é criada em uma função numa declaração precedida da palavra `var` (como na variável q da função quadrado do exemplo seguinte), ela só existe dentro desta função, enquanto esta estiver sendo executada (e é desconhecida pelo resto do seu programa). Diz-se que ela tem **escopo local** e todas as variáveis criadas desta maneira são denominadas **variáveis locais**.

A tabela 12.1 mostra a declaração da mesma função

`quadrado` de uma forma levemente diferente e sua utilização no cálculo da expressão $x = 2^2 + 3 * 5^2$:

Tabela 12.1 - Definindo e utilizando uma função

Declaração da função	Utilização da função
`function quadrado(x) { var q=x*x; return q; }`	`... x=Quadrado(2)+3*quadrado(5); //x = 4 + 3 * 25 = 79 q=Quadrado(9)-1; //q = 80 //este q não tem nada a ver //com o interno de quadrado`

Chama-se **Escopo** de uma variável a **região do programa** onde esta é conhecida.

Definindo mais precisamente variáveis locais: são aquelas que só existem dentro da função que as criou, e enquanto esta estiver sendo executada. Sua área de armazenamento é liberada quando a função for finalizada. Além disso, elas precisam ser declaradas com o uso da palavra-chave `var.`

O uso de diversas variáveis com o mesmo nome é legal, mas deve ser feito com cuidado, pois dependendo do escopo dela, você pode ou não estar se referindo à mesma variável. Vimos que você pode ou não declarar variáveis usando a palavra-chave `var`. Mas, na realidade, as duas opções não têm para o interpretador da linguagem JavaScript exatamente efeitos iguais.

Uma **variável global** tem **escopo global**, isto é , é conhecida em toda a parte do seu código. Ao contrário das declaradas dentro de uma função que só são definidas dentro do corpo da função (que têm **escopo local**). Os parâmetros de funções são também **variáveis locais** e conhecidos apenas no interior da função.

Dentro do corpo da função uma variável local tem **precedência** sobre uma variável global de mesmo nome. Se você declarou uma variável local ou um parâmetro da função com o mesmo nome de uma variável global, você estará "escondendo" a variável global para a função. Veja o exemplo que segue.

```
.......
<script language="Javascript">
document.writeln("entendendo escopo:");
var escopo="global";//declara global
document.writeln(escopo);
function vendoEscopo()
{
 var escopo="local";//local de mesmo nome
```

```
 document.writeln(escopo);
}
vendoEscopo();
document.writeln(escopo);//usa a global
</script>
........
```

O uso de "...." significa que: para que a página seja visualizada, outras linhas devem ser incluídas, ou seja ,indicam que o exemplo não está completo.

Mas se você não tivesse usado `var` dentro da função, esta não seria uma outra variável e, o que você estaria fazendo seria se referir a mesma variável global, de modo que ao alterá-la na função estaria alterando a variável global. Compare os dois casos:

```
<script language="Javascript">
document.writeln("entendendo escopo de variaveis:" );
escopo="global";
document.writeln(escopo);
function vendoEscopo()
{
  escopo="local";//agora altera a global
  document.writeln(escopo);
}
vendoEscopo();
document.writeln(escopo);
</script>
```

Resumindo, as funções "não sabem" para que você está usando as variáveis. Se você usar o mesmo nome para outra variável local de modo a "esconder" o nome da global no interior da função, deve usar a palavra-chave `var` para declarar a variável como local. Se não fizer isso estará usando uma variável global e poderá correr o risco de alterar indevidamente seu valor em outras partes do programa.

Outro ponto é que este conceito de escopo é **relativo**. Em JavaScript, as definições de funções podem ser "aninhadas" e, quando isso ocorre, cada um destes níveis de funções interiores tem seu próprio grupo de variáveis globais e locais, mas o sentido e a importância de usar a palavra-chave `var` é a mesma. Copie e rode o trecho abaixo, incluindo ou retirando a declaração de variáveis, que você compreenderá perfeitamente este conceito.

```
<script language="Javascript">
document.writeln("entendendo escopo:" );
escopo="global";
function vendoEscopo()
{
 var escopo="local";//depois rode sem var
  function interior()
  {
    escopo="maisInterno";//depois com var
    document.writeln("3"+escopo);
  }
```

```
  document.writeln("2"+escopo);
  interior();
  document.writeln("2"+escopo);
}
document.writeln("1"+escopo);
vendoEscopo();
document.writeln("1"+escopo);
</script>
```

Uma diferença importante entre C, C++ ou Java e JavaScript é que não existe nesta o conceito de escopo a nível de blocos. Mesmo variáveis que tenham sido criadas dentro de laços como nos laços `for` por exemplo, são sempre conhecidas e definidas em toda a função (isso não seria verdade nas outras linguagens). No exemplo abaixo, a variável `K,` embora definida no interior do `for,` é conhecida de toda a função. O mesmo acontece com a variável `I`.

(*) Tecnicamente falando, isso ocorre porque, diferente dos comandos, as funções são estruturas estáticas no programa.

Os comandos são avaliados em tempo de execução, mas as funções são definidas quando o código é analisado ou compilado antes de realmente rodar. Quando o analisador encontra uma função, ele a analisa e armazena os comandos do corpo da função sem a executar.

A atribuição do valor a uma variável é uma operação que usa o comando de atribuição. A definição de variáveis ocorre, portanto, em um tempo diferente da definição das funções.

```
<script language="Javascript">
document.writeln("escopo de funcoes:" );
var K="k";
var I='i';

function vendoEscopo()
{
 I=1;
 for (var K=0;K<5;++K)
 {
  document.writeln('loop:',I," ",K);
  I++;}
 document.writeln("saiu");
 document.writeln(I," ",K);
 if (K<=20) {var I=8;
 document.writeln("dentro ",I," ",K);}
}
document.writeln("antes ", I," ", K);
vendoEscopo();
document.writeln("depois ", I," ",K);
</script>
```

A regra é que todas as variáveis declaradas em uma função, não importa onde sejam declaradas, são conhecidas em toda a função. Isso pode até causar coisas aparentemente estranhas, como no exemplo anterior, onde a função acaba entendendo como local a variável, mesmo se o fluxo do programa não passar pelo ponto onde ela é definida (*). Faça, por exemplo, o fluxo do programa nunca entrar no `if` (troque 20 por 0 neste `if` por exemplo e veja o que acontece). Essa é certamente uma boa ilustração do porquê ser recomendado, como boa prática de programação, colocar todas as declarações de variáveis juntas no início das funções, deixando claro todas as variáveis que realmente são locais.

Rode depois o mesmo exemplo mais duas vezes tirando em cada caso uma das palavras `var`, ou seja, fazendo serem globais e não variáveis locais. Mas embora caracterizada como local em toda a função, a variável pode não ter sido definida, desde que seu valor não tenha sido inicializado. Se você comentar a linha `I=1`, ainda neste exemplo, veria no local correspondente à primeira impressão do *loop* ser escrito "`undefined`".

Finalmente, usando o exemplo acima, ainda é interessante observar que em JavaScript existe uma diferença entre a variável ser indefinida porque não foi inicializada ou por não ter sido declarada. A variável não ser declarada causa um erro quando o programa for executado (ou em tempo de execução - *runtime error*), porque você usou uma coisa que simplesmente não existe. Para ver esta diferença, comente a linha que declara uma das variáveis globais (por exemplo `//var K="k";`). Dependendo do navegador que você esteja usando pode ser que sua página fique simplesmente vazia.

3. Peculiaridades dos Navegadores

A forma como os navegadores consideram alguns detalhes costuma diferir em relação à visualização do código fonte e ao tratamento dos erros de sintaxe. Nas próximas duas seções 3.1 e 3.2 comentamos estas peculiaridades.

3.1. Visualização do Código Fonte

Os navegadores costumam ter uma opção de menu que permite visualizar o código fonte da página que está sendo exibida (**`view > Page Source`** no caso do Netscape 4.x e **`exibir > Código Fonte`** no caso do Explorer).

Este comportamento do Netscape em relação ao código fonte desapareceu após a versão 6 do navegador.

No caso de páginas que são modificadas por um `document.write`, a seleção desta opção do menu no Netscape (até a versão 4.x) vai mostrar a página final, após a execução do JavaScript. Neste caso, se o arquivo original `teste.html` for:

```
<HTML>
<BODY>
  Hello,
  <SCRIPT>
    document.write(" there.")
  </SCRIPT>
</BODY>
</HTML>
```

O Netscape exibira após a seleção da função **`view > Page Source`** o seguinte resultado:

```
<HTML>
<BODY>
  Hello,
   there.
</BODY>
</HTML>
```

Para ter acesso ao código original da página é necessário acrescentar **`view-source`**: antes da URL da página na **barra de endereços**. Se a página do exemplo tiver como URL `file:///C|/tmp/teste.html`, deve-se colocar na barra de endereços `view-source:file:///C|/tmp/teste.html`.

3.2. Erros de Sintaxe

Em qualquer linguagem de programação, por mais cuidadoso que seja o programador, é muito difícil não cometer erros de sintaxe. Quando ocorre numa **linguagem compilada**, isto não é muito problemático, pois, para poder executar qualquer pedaço do código, é necessário que o programa tenha passado pelo **compilador** e, conseqüentemente, não conterá mais nenhum erro de sintaxe.

Numa **linguagem interpretada** (como no caso do JavaScript), o **interpretador** da linguagem só vai descobrindo os erros à medida que executa o programa. Um erro existente num trecho que é executado após um desvio condicional pode, eventualmente, jamais ser descoberto se o valor da condição nunca "levar" o código errado a ser executado. Quando descobre um erro de sintaxe, o interpretador **não pode prosseguir** com a execução do programa.

A forma como os navegadores informam que há um erro no código JavaScript difere um pouco entre o Netscape e o Explorer. No Explorer, ao encontrar um erro de sintaxe, o navegador interrompe a execução do JavaScript e abre uma janela indicando o erro e em qual linha ocorreu. Dado o seguinte código fonte:

```
<HTML>
<BODY>
 Hello,
 <SCRIPT language=”javascript”>
   document.write(" there.” //assim mesmo
 </SCRIPT>
```

```
</BODY>
</HTML>
```

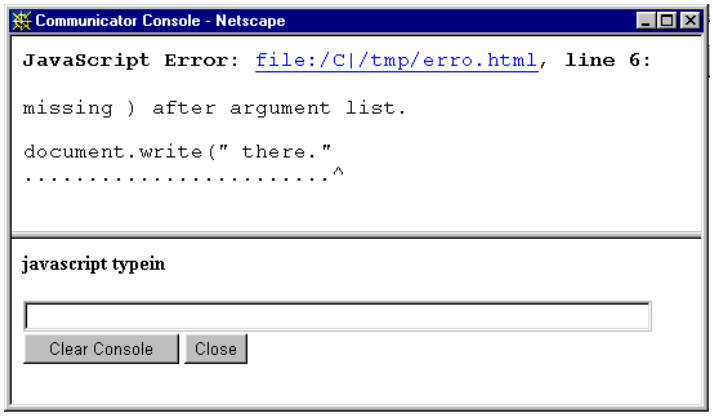
Se tentarmos exibi-lo no Explorer, a falta do caracter ") " no `write` fará com que o navegador abra a janela mostrada na figura 12.1. Com as informações desta janela, basta editar o arquivo, ir na linha e coluna indicada e corrigir o erro.

Figura 12.1 - Como o Explorer mostra erros de sintaxe

A forma de reportar os erros de sintaxe também mudou a partir da versão 6 do Netscape. Nesta versão, o console JavaScript é invocado a partir do menu: **tasks>tools> Javascript Console**

No caso do Netscape 4.x, o navegador não dá nenhum sinal que ocorreu o erro. Só é possível descobrir que alguma coisa está errada porque não vai acontecer o que esperávamos. Para fazer com que o navegador "mostre" o erro é necessário digitar `"JavaScript:"` (incluindo os pois pontos) na barra de endereços e teclar **<enter>**, o que causará a exibição da janela mostrada na figura 12.2. Com as informações desta janela deve-se ir na linha e coluna indicada e corrigir o erro.

Figura 12.2 - Como o Netscape mostra erros de sintaxe



Exercícios:

1. Transforme em função o cálculo de potências do exemplo atividade da primeira aula deste módulo (Aula 9). Depois chame a função através da seleção de um link. Use esta função para calcular potências de qualquer número fornecido pelo usuário.

2. Transforme agora o exercício 2 da aula passada, o que calculava fatoriais dos números de 1 a 10, também em função, mas de forma que o valor de retorno seja o fatorial do número fornecido.

3. Crie uma função de 3 variáveis que calcule potências do primeiro parâmetro, fatorial do segundo e escreva o terceiro como parte de um texto.

Resumo:

Nesta aula você aprendeu a criar funções com e sem parâmetros, a utilidade do comando `return` e como ele pode ser utilizado para criar funções que retornam valor, e como lidar com algumas diferenças entre os principais navegadores do mercado (Netscape e Explorer).

Auto-avaliação:

Você concluiu com facilidade os exercícios e entendeu bem funções e escopo de variáveis? Se algum ponto não ficou muito claro, releia-o. Depois observe atentamente cada passo das atividades desenvolvidas, executando-as logo a seguir. Tente entender bem essa lição antes da próxima aula! Nela você verá uma forma de agrupar dados para representar estruturas complexas: os objetos.