

Disciplina: Programação Cliente-servidor

Aula 3: JavaScript – Parte 2

Apresentação

A partir do momento em que conseguimos construir nossas páginas, estruturando com o HTML e formatando com CSS, devemos cuidar da interatividade da mesma. A linguagem JavaScript será o ferramental necessário para prover esta interatividade, atuando por meio do DOM e dos eventos da página.

Com o domínio dessas técnicas, podemos também efetuar a validação de nossos formulários, viabilizando diversas críticas acerca do formato e exigência dos dados envolvidos nos cadastros.

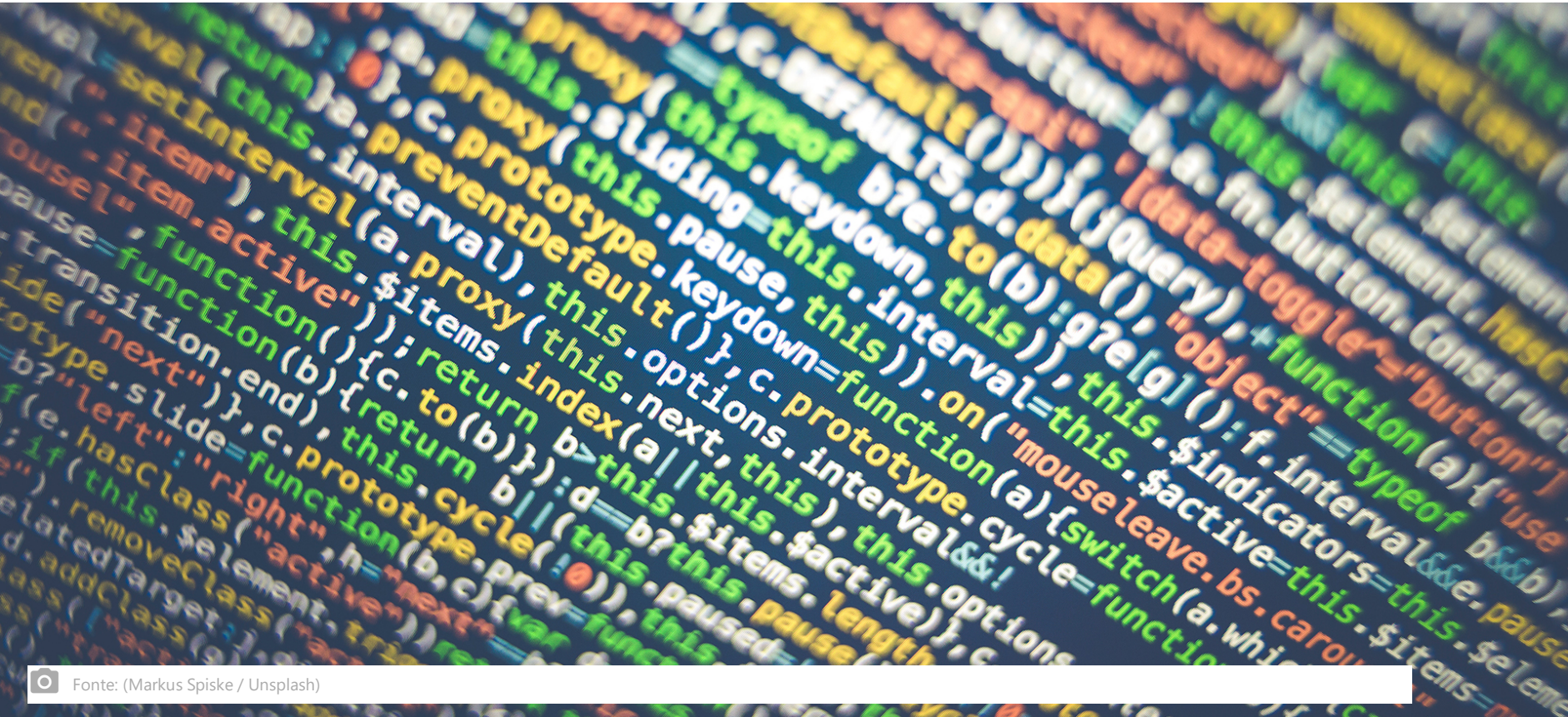
Por fim, assim como em outras plataformas, o uso da orientação a objetos irá trazer maior robustez ao código implementado.

Objetivos

- Explicar os elementos de interatividade com a página;
- Aplicar o JavaScript para a validação de formulários;
- Analisar a sintaxe JavaScript para orientação a objetos.

Interação com a página HTML

Precisamos ter em mente que o principal objetivo do JavaScript é expandir as funcionalidades básicas de uma página HTML. Para isso, deve ser capaz de interagir com seus componentes e com o utilizador da página.



Fonte: (Markus Spiske / Unsplash)

Neste contexto, temos dois caminhos que funcionam em sentidos inversos:

O uso de DOM permitirá ao JavaScript acessar os componentes da página;	Através de eventos , a página poderá acionar rotinas em JavaScript.
---	--

Antigamente, a forma de acesso aos componentes HTML variava muito de um navegador para outro, mas a [W3C¹](#) padronizou o acesso aos componentes através do uso de **document** e identificadores do **DOM**.

W3C¹

Modal: W3C – World Wide Web Consortium é uma comunidade internacional onde diversas organizações-membro, uma equipe em tempo integral e o público trabalham juntos para definir padrões para a Web.

Fonte: <https://www.w3.org/Consortium/> <<https://www.w3.org/Consortium/>>

Métodos de acesso DOM

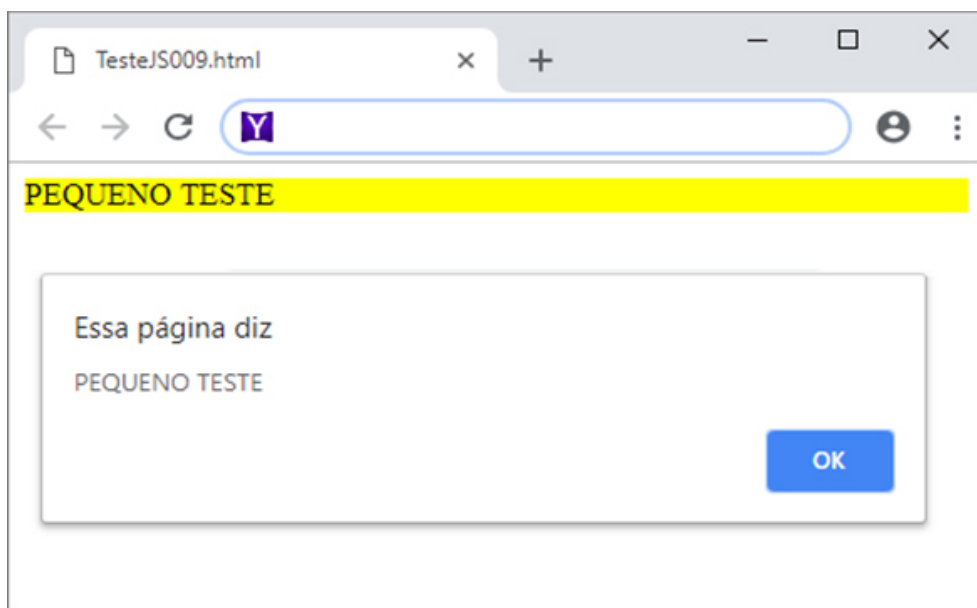
Através de **document**, acessamos os elementos DOM de diferentes formas, sendo a mais tradicional através do atributo **id** do elemento desejado, como podemos observar no exemplo:

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>
    <div id="Camada1">PEQUENO TESTE</div>
    <script>
      var obj = document.getElementById("Camada1");
      obj.style.backgroundColor = "yellow";
      alert(obj.firstChild.nodeValue);
    </script>
  </body>
</html>
```



Inicialmente, devemos considerar as formas de acesso ao **DOM** proporcionadas através de **document**. Neste exemplo, a variável **obj** recebe uma referência ao componente **div** da página cujo **Id** é "Camada1", observando as letras maiúsculas e minúsculas utilizadas, já que é **case-sensitive**.

Alguns dos métodos de **document** para acesso aos componentes da página podem ser observados no quadro:

Método	Retorno
getElementById	Retorna um objeto como referência a um componente unicamente identificado através de seu atributo id
getElementsByClassName	Retorna uma coleção de objetos referenciando todos os componentes que apresentem o atributo class desejado
getElementsByTagName	Retorna uma coleção de objetos referenciando todos os componentes do tipo desejado, como button, div ou h1
querySelector	Retorna um objeto com o primeiro elemento que utilize o seletor CSS
querySelectorAll	Retorna uma coleção de objetos com todos os elementos que utilizem o seletor CSS

Assim como foi utilizado **document.getElementById("Camada1")**, poderíamos utilizar o comando **document.querySelector("#Camada1")** no lugar, e iríamos obter o mesmo resultado final.

O objeto receptor assume as características do componente HTML referenciado, podendo ser acessados seus diversos atributos, como **style**, ou no caso de uma tag <input> atributos como **value** e **maxLength**.

Ao mesmo tempo em que podemos utilizar os atributos HTML para o tipo de tag associado ao objeto, podemos observar este objeto como um nó de árvore da estrutura DOM, o que permite a navegação entre os nós e tipificação.

Isto pode ser observado ao final do exemplo, onde é utilizado **obj.firstChild.nodeValue**, correspondendo ao conteúdo do primeiro nó filho de obj.

Sempre devemos lembrar de que, em termos de DOM, obj corresponde a um nó do tipo Element, referenciando uma tag <div>, e o conteúdo HTML interno desta tag é um outro nó da árvore, do tipo Text, obtido com firstChild.

Como este outro nó é do tipo texto, o atributo **nodeValue** retornará o texto contido na tag original.

Navegação DOM

Inicialmente, devemos lembrar que o DOM permite a manipulação de diferentes tipos de nós, como elementos, comentários e textos.

Para descobrir qual o tipo de nó, podemos utilizar a propriedade **nodeType**, que é numérica, enquanto o nome do elemento associado pode ser obtido com **nodeName**, e o conteúdo do mesmo com **nodeValue**.

A tabela a seguir mostra a relação entre estas três propriedades:

nodeType	Tipo de Nó	nodeName	nodeValue
1	Element	Nome do Elemento (tag)	Null
2	Attr	Nome do atributo	Valor do atributo
3	Text	#text	Conteúdo do nó
4	CDATASection	#cdata-section	Conteúdo do nó
5	EntityReference	Referência de Entidade	Null
6	Entity	Nome da Entidade	Null
7	ProcessingInstruction	Alvo da ação	Conteúdo do nó
8	Comment	#comment	Comentário na forma de texto
9	Document	#document	Null
10	DocumentType	Nome do DocType	Null
11	DocumentFragment	#document-fragment	Null
12	Notation	Nome da notação	Null

Fonte: Adaptado de [W3schools <https://www.w3schools.com/jsref/prop_node_nodetype.asp>](https://www.w3schools.com/jsref/prop_node_nodetype.asp)

Exemplo

Por exemplo, para uma tag do tipo **<div>**, teríamos **nodeType** com valor 1 (**Element**), **nodeName** com valor **"_div_"** e **nodeValue** nulo, e o conteúdo desta tag estaria em outros nós da árvore DOM, filhos da mesma.

Para um conteúdo texto simples, ainda com o exemplo da tag **<div>**, este poderia ser obtido com o uso de **firstChild.nodeValue**, como foi feito no exemplo apresentado anteriormente.

Atenção

Neste ponto devemos observar os métodos de navegação. Por se tratar de uma árvore, devemos ter a possibilidade de acessar um nó específico, assim como os descendentes do mesmo.

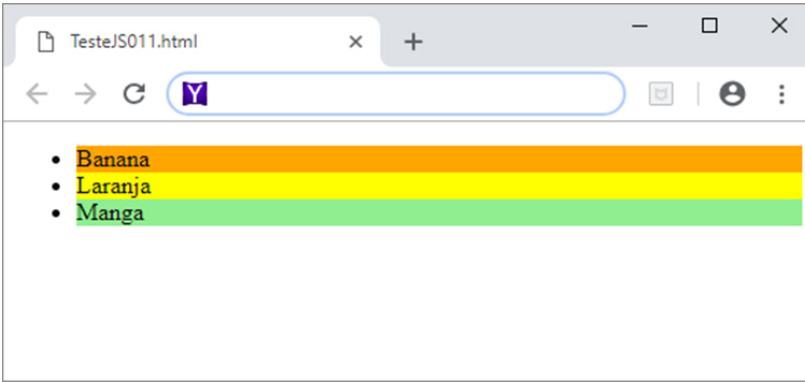
Para o acesso ao nó específico, normalmente a partir do **id** do mesmo, podem ser utilizadas as instruções **getElementById** ou **querySelector**, conforme discutido anteriormente, e a partir do nó localizado, podemos acessar os descendentes do mesmo.

Exemplo

Vamos observar o exemplo:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <ul id="lista">
      <li>Banana</li>
      <li>Laranja</li>
      <li>Manga</li>
    </ul>

    <script>
var obj = document.querySelector("lista")
var filhos = obj.children;
var cores = ["orange","yellow","lightgreen"];
for (i = 0; i < filhos.length; i++) {
  filhos[i].style.backgroundColor = cores[i];
}
    </script>
  </body>
</html>
```

A screenshot of a web browser window titled 'Teste/S011.html'. The browser's address bar shows a purple icon. The page content displays a bulleted list of three items: 'Banana', 'Laranja', and 'Manga'. Each item is highlighted with a different background color: 'Banana' is orange, 'Laranja' is yellow, and 'Manga' is light green.

Neste exemplo, nós localizamos o início da lista com uso de **querySelector("#lista")**, e a partir deste elemento obtivemos os elementos **** internos através do atributo **children** do nó.

Como esses elementos são obtidos no formato **ObjectHtmlCollection**, é possível navegar nos mesmos como um vetor, aplicando as cores disponíveis no segundo vetor ao fundo de cada elemento através de **style**.

Alguns dos atributos que podem ser utilizados pelo nó para a navegação na árvore podem ser observados no quadro:

Atributo	Conteúdo
firstChild	Retorna o primeiro filho do nó corrente
childNodes	Retorna uma coleção de nós contendo os filhos do nó corrente
parentNode	Retorna o nó que ascende ao nó corrente (pai)
firstElementChild	Retorna o primeiro filho do tipo Element para o nó corrente
lastElementChild	Retorna o último filho do tipo Element para o nó corrente
children	Retorna todos os filhos do tipo Element para o nó corrente

De acordo com o tipo de nó existente na árvore, os atributos podem ser diferentes, mas a W3C traz uma documentação muito completa acerca de cada tipo de nó, como para o tipo **Element**, que representa as tags do HTML ou XML.

Saiba mais

A referência pode ser obtida no endereço https://www.w3schools.com/jsref/dom_obj_all.asp
<https://www.w3schools.com/jsref/dom_obj_all.asp>.

Eventos

Podemos definir evento como uma ação pré-determinada que, ao ocorrer, permite que seja iniciada uma ação personalizada, o que certamente será feito através de programação.

Por exemplo, ao clicar sobre o botão, inicie a função somar, o que poderia ser escrito da seguinte forma no HTML:

```
<button id="btn1" onClick="somar()">SOMAR</button>
```

Caso o evento não seja associado ao nível do HTML, ele pode ser atrelado ao objeto através de JavaScript e DOM, como no código a seguir:

```
document.getElementById("btn1").addEventListener("click",
function(event) {
    somar();
});
```

O método addEventListener recebe, como parâmetros, o nome do evento a ser utilizado e uma função callback para resposta ao evento.

Exemplo

Observe um pequeno exemplo de utilização de evento sobre o clique do botão:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"/>
</head>
<body>
<button id="Teste1">Clique Aqui</button>
<script>
  function mostraMensagem(){
    alert("OLA MUNDO");
  }
  var obj = document.querySelector("#Teste1");
  obj.addEventListener("click",mostraMensagem);
</script>
</body>
</html>
```

Neste exemplo, a página contém apenas um botão, e, ao clicar sobre ele, aparecerá a mensagem “OLA MUNDO” em um **alert**.

Podemos observar a associação do evento de clique com a função de tratamento na seguinte linha:

```
obj.addEventListener("click",mostraMensagem);
```

Outra forma de associar seria diretamente no HTML:

```
<button id="Teste1" onclick="mostraMensagem()"> Clique Aqui </button>
```

Modificações dinâmicas

Com o uso de DOM, podemos acessar, alterar ou criar elementos em uma página HTML de forma dinâmica, durante a sua visualização no navegador.

É um processo razoavelmente simples, e que traz grande flexibilidade para a interface visual.

Incialmente, devem ser utilizados alguns métodos de **document** para a criação dos nós de acordo com o tipo correto, e depois os métodos existentes para os nós DOM permitem o acréscimo, substituição ou remoção de outros nós.

A tabela seguinte mostra alguns métodos existentes em **document** para a criação de nós DOM.

Método	
createElement	Cria um nó do tipo Elemento (tag)
createTextNode	Cria um nó de texto
createAttribute	Cria um atributo. O nó pode utilizar com setAttributeNode
createComment	Cria um comentário

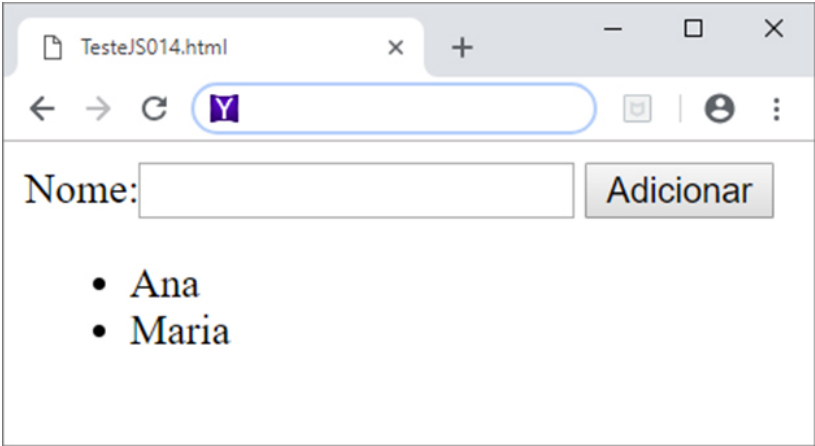
Os nós e atributos criados a partir de **document** podem ser utilizados e anexados a nós de elementos já existentes na estrutura da página, e o nó inicial da página pode ser obtido com o elemento **body** de **document**.

```
var x = document.createComment("Apenas um comentário");
document.body.appendChild( x );
```

Exemplo

O exemplo seguinte demonstra a utilização desta metodologia de inserção de nós:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    Nome:<input type="text" id="nome">
    <button onclick="adicionar()">Adicionar</button>
    <ul id="lista">
    </ul>
    <script>
var lista = document.querySelector("#lista");
var texto = document.querySelector("#nome");
function adicionar(){
  var node = document.createElement("LI");
  var textnode =
document.createTextNode(texto.value);
  node.appendChild(textnode);
  lista.appendChild(node);
  texto.value = "";
  texto.focus();
}
    </script>
  </body>
</html>
```



Neste exemplo, criamos uma função **adicionar** onde, em sua implementação, é criado um nó de elemento do tipo **** e um nó de texto contendo o valor da caixa identificada por “nome”.

```
var node = document.createElement("LI");
var textnode = document.createTextNode(texto.value);
```

Nas linhas seguintes, podemos observar o nó de texto sendo anexado ao nó de elemento, e o elemento **** sendo adicionado à lista da página.

```
node.appendChild(textnode);
lista.appendChild(node);
```

As duas últimas linhas tratam apenas de um refino de interface onde, logo após a inserção, é limpo o texto existente na caixa e colocado o foco na mesma.

Outra forma muito comum de modificação de conteúdo é com o uso do atributo **innerHTML** dos nós DOM. Com ele, podemos colocar qualquer conteúdo HTML dentro da tag representada pelo nó de elemento.

```
document.getElementById(“minhaDiv”).innerHTML = "<h1>TESTE</h1>";
```

Finalmente, podemos remover nós da árvore, a partir de um nó de elemento DOM, com o uso de **removeChild**, ou substituí-los com o uso de **replaceChild**.

Validação de formulários

Um formulário é uma entrada de dados simples, que considera apenas texto e seleções, mas sem grandes críticas acerca de formato e validade destes dados.

Entre diversas outras ações, devemos nos preocupar com a necessidade de:

- ✓ Definir campos obrigatórios;
- ✓ Utilizar tipos de dados específicos;
- ✓ Controlar a visibilidade de campos alternativos.

Podemos utilizar os recursos do HTML5 para efetuar parte dessas críticas, e o JavaScript para definir aquelas que não sejam cobertas por estes recursos.

No entanto, devemos ter em mente que as críticas efetuadas do lado cliente são apenas relacionadas ao formato dos dados, e não à integridade da base. Críticas como violações de chave primária ou inexistência de registro na base só podem ser feitas no lado servidor.

Também precisamos lembrar que o JavaScript pode ser desativado pelo usuário, o que tornaria o conjunto de validações inócuo, exigindo uma nova validação do lado servidor, e que realmente deve ser feita.

Por que validar no cliente se temos que validar no servidor novamente?

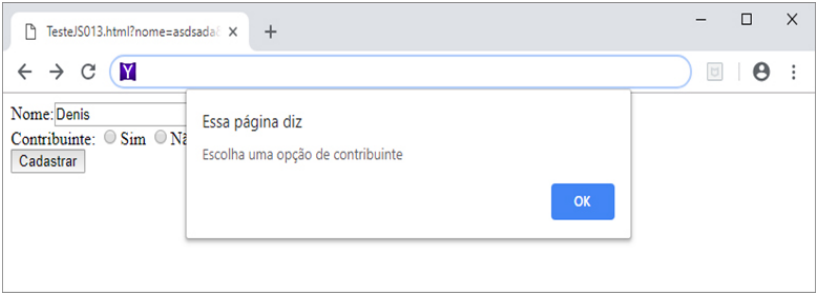
A resposta tem a ver com **usabilidade** e **fluxo de rede**, pois a resposta da validação de formato no cliente é mais rápida, além de diminuir o fluxo de rede com chamadas desnecessárias ao servidor, já que os dados serão criticados antes do envio.

Assim como em todos os demais elementos de interatividade da página com as rotinas JavaScript, também na validação contamos com os eventos para definir o momento de acionamento da crítica ou formatação.

Exemplo

Podemos efetuar uma validação global a partir do evento **onsubmit**, como no exemplo seguinte.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <form method="get" action="//lugarnenhum" onsubmit="return validar();">
      Nome:<input type="text" name="nome" id="nome"/><br/>
      Contribuinte:<input type="radio" name="contrib" id="contrib1" value="S">Sim
      <input type="radio" name="contrib" id="contrib2" value="N">Não
      <br/>
      <input type="submit" value="Cadastrar"/>
    </form>
    <script>
      function validar(){
        var nome = document.getElementById("nome"),
            contrib1 = document.getElementById("contrib1"),
            contrib2 = document.getElementById("contrib2");
        if(nome.value==""){
          alert("Nome é obrigatório");
          nome.focus();
          return false;
        }
        if(!contrib1.checked && !contrib2.checked){
          alert("Escolha uma opção de contribuinte");
          return false;
        }
        return true;
      }
    </script>
  </body>
</html>
```



Devemos observar o formato da função de validação, que deverá retornar **true** ou **false** para o evento **osSubmit**, de forma a permitir ou não o envio da informação para o servidor.

Por este motivo a chamada deste evento é um pouco diferente dos outros.

```
<form method="get" action="//lugarnenhum"
onsubmit="return validar();">
```

Outro elemento interessante neste código é o uso de **focus()**. Caso o nome não seja preenchido, a mensagem **“Nome é obrigatório”** é apresentada e o foco é direcionado para a caixa de texto referente a este dado.

Para o **teste dos componentes do tipo rádio**, devemos verificar se nenhum deles foi marcado. Para isso, utilizamos suas propriedades checked.

Lembrando que, pelo fato de a propriedade ser booleana, a negação será equivalente à comparação com false.

```
if(!contrib1.checked && !contrib2.checked)
```

Outras validações e formatações podem ser efetuadas no momento da perda do foco pela caixa de texto, ou quando seleccionamos o elemento de uma lista de valores, entre diversas outras opções.

Observe, no quadro seguinte, alguns eventos do HTML e suas respectivas aplicações no processo de validação.

Evento	Aplicação
onsubmit	Efetua a validação do formulário imediatamente antes do envio para o servidor. Necessita o retorno booleano, indicando se os valores podem ser enviados ou não
onclick	Normalmente uma chamada explícita de validação. Muito utilizado em botões de rádio e caixas de marcação
onchange	Ocorre quando o valor (value) sofre uma alteração
onfocus	Ocorre quando o componente ganha o foco. Pode ser utilizado, por exemplo, para apagar o valor do campo
onblur	Ocorre na perda do foco pelo componente. É comum a aplicação de máscaras em valores numéricos como CEP e CPF
onsearch	Este evento é iniciado quando um usuário digita algo em um campo de pesquisa (type=”search”)
onselect	Utilizado quando algum texto é selecionado no campo

Orientação a objetos

Com a criação de sistemas cada vez maiores e com grande apelo visual, as técnicas tradicionais de modelagem e programação estruturada começaram a entrar em colapso.

Complexos trechos de código inter-relacionados, junto com a documentação escassa e diversas replicações de processos já existentes, acabam tornando a manutenção dos sistemas extremamente difícil, aumentando o custo e diminuindo as possibilidades evolutivas destes sistemas.

A orientação a objetos surge neste contexto, trazendo uma forma mais organizada de trabalho, onde a modelagem e a implementação mostram uma proximidade muito maior do que nas técnicas ditas tradicionais.



Saiba mais

O termo Programação Orientada a Objetos (POO) foi criado por Alan Kay, autor da linguagem de programação Smalltalk. Mas, mesmo antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar estas ideias foi a linguagem Simula 67, criada por Ole-Johan Dahl e Kristen Nygaard em 1967. Entretanto só veio a ser aceito realmente nas grandes empresas de desenvolvimento de Software por volta dos anos 1990.

Fonte: [Programação Orientada a Objetos/Introdução](#)

https://pt.wikibooks.org/wiki/Programa%C3%A7%C3%A3o_Orientada_a_Objeto/Introdu%C3%A7%C3%A3o#Hist%C3%B3ria

Para podermos adotar esta nova filosofia, devemos deixar de pensar em termos de processos e funções, pois isto é a metodologia estruturada, focada em funcionalidades pontuais e a organização das mesmas.

Agora precisamos pensar de uma forma diferente, em termos de personagens, quais deverão apresentar características físicas e ações ou verbos.

Por exemplo, na **programação estruturada**, diríamos que o projétil partiu no ângulo de 55 graus, sofrendo a ação da gravidade, e atingindo o prédio na altura do quarto andar, pois estamos **definindo um processo**.

Em termos de **orientação a objetos**, consideraríamos que temos um **personagem** chamado tanque de guerra, e que ele é capaz de atirar um projétil.

A mudança de foco é muito grande e tem como objetivo:

Aumento do reuso de código;

Facilidade de manutenção;

Documentação automatizada.

Começamos a utilizar de forma mais ampla a programação orientada a objetos (POO) com o advento das interfaces gráficas, pois ficou muito evidente que a programação estruturada não era a melhor opção para construir ambientes constituídos de janelas.

Antigamente tínhamos que utilizar as APIs do sistema operacional para a construção de cada janela, de forma independente, mas, com a POO, podemos definir um personagem denominado “**Janela**”, que:

Terá atributos como “posição”, “largura” e “altura”; e

Será capaz de efetuar ações como “abrir”, “minimizar” e “maximizar”.

A partir daí, podemos colocar a quantidade necessária de personagens deste tipo para implementar as interfaces de nossos sistemas.

Abstração

Um dos pilares da POO é o conceito de abstração, que se refere à definição de um modelo simplificado de algo maior.

Quando abstraímos algo, estamos preocupados apenas com os detalhes que sejam relevantes para o problema de interesse, e estas abstrações serão representadas como classes na POO, que trazem a definição dos atributos e métodos suportados pelos personagens.

Os **atributos** definem características físicas, como cor, idade, endereço etc., enquanto **métodos** são as ações, ou verbos, que podem ser praticadas, tais como comer, andar ou dormir.

Uma classe funciona como um molde (tipo ou domínio), de forma a definir como serão os objetos criados a partir da mesma, como no exemplo seguinte, em JavaScript.

```
function Pessoa(nome, idade){
  this.nome = nome;
  this.idade = idade;
}
```

Atenção

Neste exemplo, estamos definindo uma classe Pessoa, com o uso de function, segundo a sintaxe utilizada pelo JavaScript. Note que esta é uma abstração que define o modelo Pessoa apenas a partir do nome e da idade.

Qual é este nome e esta idade?

Aqui entram os objetos, pois as classes definem o modelo que será seguido por suas instâncias (ou objetos), e estas instâncias assumem valores para os atributos definidos.

EXEMPLO

Observe o exemplo completo a seguir:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"/>
</head>
<body>
<script>
function Pessoa(nome, idade){
  this.nome = nome;
  this.idade = idade;
  this.exibir = function( ) {
    alert(this.nome+" tem "+this.idade+" ano(s)");
  }
}
var p1 = new Pessoa("Ana",25);
var p2 = new Pessoa("Marcos",36);
</script>
<button onclick="p1.exibir()"/>P1</button>
<button onclick="p2.exibir()"/>P2</button>
</body>
</html>
```

Inicialmente, temos a classe **Pessoa**, criada através de **function** e com o uso do ponteiro **this** para a definição dos atributos **nome** e **idade**, além do método **exibir**, sem parâmetros.

Podemos ler o ponteiro de autoreferência **this** como **“deste”**, ou seja, o atributo idade desta Pessoa (**this.idade**), ou o método exibir desta Pessoa (**this.exibir**).

Atenção

Uma observação a ser feita é que, assim como as classes são definidas com o uso de function, os métodos da mesma também são. Basta observar a definição do método `exibir`.

```

    this.exibir = function() {
        alert(this.nome+" tem "+this.idade+" ano(s)");
    }

```

Com a definição da classe Pessoa, podemos criar os objetos p1 e p2, cada um com seus próprios valores para os atributos nome e idade. Para instanciar os objetos, utilizamos o operador **new**, responsável por alocar a memória necessária para o novo objeto.

```

var p1 = new Pessoa("Ana",25);
var p2 = new Pessoa("Marcos",36);

```

Finalmente, podemos observar a chamada ao método `exibir` de p1 ou p2 a partir dos eventos presentes nos dois botões da página HTML.

```

<button onclick="p1.exibir()"/>P1</button>
<button onclick="p2.exibir()"/>P2</button>

```

Ao clicar no botão com texto “P1” será acionado o método `exibir` do objeto p1, enquanto o botão com texto “P2” acionará o método `exibir` de p2.

Protótipo

O uso de protótipo é uma peculiaridade do JavaScript e não uma característica própria da orientação a objetos.

Na verdade, o JavaScript não apresenta um mecanismo específico para herança, que seria outro dos pilares da orientação a objetos, mas permite a utilização de **prototype** para expandir a funcionalidade de classes já existentes.

Vamos considerar uma classe Pessoa constituída apenas de nome e sobrenome:

```
function Pessoa(nome, sobrenome){
  this.nome = nome;
  this.telefone = sobrenome;
}
```

Com o uso de **prototype**, podemos adicionar a nacionalidade para esta classe já existente. Claro, que exigirá uma inicialização prévia, já que o construtor não tem como prever o novo atributo.

```
Pessoa.prototype.nacionalidade = "Brasileiro(a)";
```

Da mesma forma que podemos adicionar um atributo, podemos adicionar um método a esta classe com o uso de prototype.

```
Pessoa.prototype.nomeCompleto = function( ) {
  return this.nome + " "+this.sobrenome;
};
```

Atividade

1. O uso de DOM permite a modificação dinâmica de partes da página, com o acréscimo, remoção ou alteração de elementos, tratando de um elemento essencial na interação entre o JavaScript e a página.

Observando o trecho de HTML abaixo, qual seria a instrução JavaScript para obter acesso ao elemento DIV e colocar nele a frase “EXERCICIO DOM”?

```
<div id="XPTO">ALVO</div>
```

- a) document.getElementById("XPTO").value = "EXERCICIO DOM";
 - b) document.querySelector("#XPTO").value = "EXERCICIO DOM";
 - c) document.getElementById("XPTO").innerHTML = "EXERCICIO DOM";
 - d) document.querySelector(".XPTO").innerHTML = "EXERCICIO DOM";
 - e) document.querySelector("DIV").value = "EXERCICIO DOM";
-

2. Você está criando um formulário para cadastro de leitores de um jornal na Web, onde devem constar os dados residenciais do leitor. A empresa pediu que dados como rua, bairro, cidade e estado sejam preenchidos automaticamente após o leitor digitar o CEP e sair da caixa de texto. Qual evento deve ser utilizado para efetuar este preenchimento?

- a) onblur
- b) onclick
- c) onexit
- d) onenter
- e) onchange

3. As linguagens da atualidade buscam metodologias mais organizadas para a programação, e a orientação a objetos acaba sendo amplamente adotada com este objetivo. O JavaScript também permite o uso desta metodologia, e, para definir atributos de uma classe, é utilizada uma palavra reservada específica. Qual a palavra utilizada?

- a) function
- b) new
- c) inherited
- d) this
- e) super

Notas

Título modal ¹

Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos.

Título modal ¹

Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos.

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

- Sintaxe JSON (JavaScript Object Notation);
- Bibliotecas JQuery;
- Biblioteca JQuery UI para construção de páginas.

Explore mais

Para entender melhor os assuntos tratados nesta aula, acesse estes materiais:

- [Uso de JavaScript com DOM](https://www.w3schools.com/js/js_htmlDOM.asp) <https://www.w3schools.com/js/js_htmlDOM.asp>
- [Eventos](https://www.w3schools.com/tags/ref_eventattributes.asp) <https://www.w3schools.com/tags/ref_eventattributes.asp>
- [Orientação a objetos em JavaScript](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript) <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript>