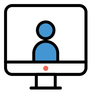
Programação Estruturada

Aula 15 - Tratamento de Erros, Estilo de Código e Boas Práticas de Programação

Videoaula 03: Boas Práticas de Programação

Material Didático do Instituto Metrópole Digital - IMD
Versão 5.3 - Todos os Direitos reservados



Videoaula 03: Boas Práticas de Programação

Você sabia que os seus programas precisarão passar por manutenção, assim como equipamentos do seu dia a dia? Além disso, os seus programas também precisarão evoluir à medida que as necessidades de seus clientes mudarem. A Engenharia de Software utiliza tecnologias, práticas e processos visando à eficiência no desenvolvimento de softwares. Esta disciplina é bastante ampla e tem até cursos de graduação voltados para o seu aprendizado.

Nesta videoaula, pretendo apenas apresentar para você uma dessas práticas utilizadas pela engenharia de software, as "boas práticas de programação", um conjunto de convenções que, uma vez respeitadas, tornam os programas mais legíveis e facilitam a manutenção e evolução dos programas. As "boas práticas de programação" também são, por si só, um tema bastante amplo e existem disciplinas em cursos de graduação voltadas apenas para elas.

Esta videoaula será apenas uma introdução a esse tema. Porém, fique bem atento a tudo que será dito aqui, pois as convenções que eu apresentarei já podem ter um efeito imediato na sua produtividade, melhorando o entendimento, a corretude e a eficiência de seu código.

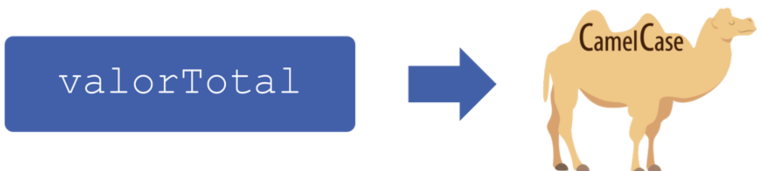
Primeiramente, veremos diversas convenções que lhe ajudarão a TORNAR O SEU CÓDIGO MAIS COMPREENSÍVEL. Conforme vimos na videoaula anterior, a aderência a um estilo de código ajuda bastante nisso. Então, a primeira dica é que você **adote e siga um estilo de código**.

Além disso, **escolha nomes intuitivos para as suas funções, variáveis e constantes** de maneira que eles indiquem imediatamente do que elas se tratam. Você acha mais fácil descobrir o que uma função chamada `f` faz ou uma função chamada

`fatorial`? Essa dica parece óbvia, mas você se surpreenderá ao se deparar com programas que não respeitam essa convenção. E lembre-se: **escolha nomes diretos e curtos!**

Uma outra boa prática é **definir um padrão para o nome das funções, variáveis e constantes**. Um padrão que eu particularmente prefiro é de sempre usar letras maiúsculas para constantes com o sublinhado separando as palavras e de usar o formato *Lower Camel Case* para variáveis e funções. Já falamos sobre esse formato na aula sobre Variáveis e Constantes. Nesse formato, cada palavra, com exceção da primeira, é iniciada com uma letra maiúscula e as demais letras são minúsculas. As palavras são, então, unidas sem espaço. Veja no slide (Figura 1) como fica o nome da variável `valorTotal`.

Figura 1 - *Lower Camel Case*



Sempre **comente o seu código**, explicando do que se trata. Isso se aplica a variáveis, constantes, funções e, até mesmo, trechos de código. Explique o algoritmo que você escreveu e a lógica que usou, mostre o objetivo das variáveis e das funções. Sempre faça isso antes do elemento sobre o qual está comentando. Você poderá se surpreender com a importância de um código bem comentado. Os comentários serão importantes não só para os outros membros de sua equipe, mas para você mesmo quando retornar ao código algum tempo depois de escrevê-lo. Eu posso assegurar que, se você não comentar bem o seu código, no futuro não se lembrará mais da funcionalidade do algoritmo que você mesmo escreveu. Em muitas ocasiões, a boa escolha de nomes ajudará nessa questão, mas o comentário é essencial para qualquer informação adicional.

No entanto, a fim de evitar que os comentários deixem o código poluído, difícil de ler e atualizar, **utilize comentários sucintos e relevantes para a compreensão do código**.

Para facilitar, nos comentários das variáveis, **declare cada variável em uma linha**.

Outras duas dicas que ajudam bastante a comentar o código e, consequentemente, entendê-lo, é **quebrar comandos longos em comandos menores e mais simples e escrever um comando por linha**.

Geralmente, a fim de modularizar melhor o seu código, como fizemos ao longo dessa disciplina, você utilizará funções que resolvem problemas menores para resolver problemas maiores. Alguns programadores chamam essas funções menores de auxiliares, outros preferem declarar as funções auxiliares antes da declaração do código que as usam. No entanto, prefiro fazer o contrário: **declarar primeiramente o código e depois declarar as funções auxiliares** que esse código utiliza. Isto porque, no meu ponto de vista, as pessoas precisam primeiro entender o que o código faz para depois entender como ele faz. Ao escrever o código antes, fica claro desde o início o que ele faz. Depois, é possível até que as pessoas não precisem nem ler as funções auxiliares, principalmente se seus nomes já deixaram claro o que elas fazem.

Quando for escrever programas, lembre-se sempre que outros programadores, e você mesmo, possivelmente precisarão entender o seu código para fazer manutenção ou evolução.

Portanto, **evite escrever códigos complicados** que poderiam ser escritos de maneira mais simples. Algumas vezes, você ficará tentado a fazer isso por acreditar que códigos complicados demonstram que sabemos programar mais. Na prática, isso prejudicará tanto você quanto a sua equipe.

Outra convenção importante é **declarar todas as variáveis no início do *script* ou da função e inicializá-las imediatamente sempre que possível**. Seguindo essa convenção, você obterá códigos mais limpos, além de ter um único local para procurar variáveis locais. Além disso, facilita a identificação (e remoção) de variáveis globais implícitas indesejadas e reduz a possibilidade de redeclarações indesejadas. Lembre-se que o JavaScript move todas as declarações para o topo do *script*, mas isso é feito apenas na interpretação. O código fonte permanece inalterado. Portanto, é importante seguir essa convenção.

Ainda falando sobre variáveis, **evite usar variáveis globais**, pois elas podem ser sobrescritas por outros scripts. Ao invés disso, use variáveis locais. Além disso, variáveis globais podem ser transformadas em variáveis locais (ou privadas), usando fechamentos (*closure* em inglês). Nesta disciplina, não apresentamos fechamentos, mas você pode acessar o QR code que está exposto no slide (Figura 2) se desejar aprender.

Figura 2 - Informações sobre *Closures*

Outra dica: **evite ter mais de um ponto de retorno nas funções**, como vemos no código do lado esquerdo do slide (Figura 3). Ao invés disso, prefira ter um ponto único de retorno no final do corpo da função. Veja no slide (Figura 3) no código do lado direito como foi usada essa convenção no exemplo visto nesta aula com a variável `resultado`.

Figura 3 - Evitando Mais de Um Ponto de Retorno

```
function resultado(m) {
    if (m >= 7) {
        return "Aprovado";
    } else if (m >= 5) {
        return "Recuperação";
    } else {
        return "Reprovado";
    }
}
```

```
function resultado(m) {
    var resultado;

    if (m >= 7) {
        resultado = "Aprovado";
    } else if (m >= 5) {
        resultado = "Recuperação";
    } else {
        resultado = "Reprovado";
    }

    return resultado;
}
```

Sempre que possível, **evite usar negações nas condições de blocos `if-else`**, como visto no slide (Figura 4). As pessoas tendem a se confundir com expressões lógicas como essa. Um pouquinho de lógica proposicional ajudará você a encontrar a expressão lógica mais simples que seja equivalente à condição desejada. Veja como evitamos o uso da negação usando o fato de que, se um número não é menor que outro, então ele é maior ou igual ao outro.

Figura 4 - Evitando o Uso da Negação

Darei agora algumas dicas de como MELHORAR A SUA PRODUTIVIDADE.

Primeiramente, faça uso da modularidade de maneira organizada. A **separação do seu código fonte em pacotes** é uma boa prática em programação. Em outras palavras, seja um programador organizado que mantém seus códigos bem separados em pacotes, facilitando assim a localização e o reuso deles não só por você, mas por toda a sua equipe. Recomendo que você comece a fazer isso a partir de agora. Que tal já organizar todo o material construído ao longo desta disciplina?

Utilize blocos `try-catch-finally`. Essa é considerada uma boa prática por questões de segurança. Por exemplo, vimos várias situações de erro que faziam com que a página HTML ficasse completamente em branco. Claro que você aprendeu a usar a ferramenta de depuração para identificar o erro, mas imagine se você fosse um usuário comum desta página. Ao controlar os erros, você poderá exibir mensagens mais intuitivas para o usuário final, evitando esse comportamento ou aquelas mensagens padrão da maioria das linguagens de programação que são em inglês e incompreensíveis para o usuário comum.

Digite as chaves, parênteses e colchetes (abrindo e fechando), antes de digitar os comandos ou expressões entre eles. Vários editores de texto, como o que utilizamos nesta disciplina, já fazem isso automaticamente. Por exemplo, quando digitamos o abre parênteses, o *Visual Studio Code* já insere o fecha parênteses. Se o seu editor não faz isso, você observará que, ao fazê-lo estará evitando perder tempo procurando qual símbolo você abriu, mas esqueceu de fechar, causando um erro de casamento entre aberturas e fechamentos de blocos.

Por fim, adquira o hábito de **salvar constantemente o código** que você está escrevendo. Para isso, aprenda imediatamente o atalho de teclado com essa função.

E, é claro, não se esqueça de **fazer *backups*** do seu código com frequência. Eu, sinceramente, não desejo que aconteça com você estar no meio de um projeto e perceber que perdeu tudo. Portanto, faça *backups*, preferencialmente na nuvem, para evitar que uma falha no seu hardware cause esse desastre.

Já estamos próximos da conclusão desta aula, mas antes, vamos dar algumas dicas de como AUMENTAR A CHANCE DE SEU CÓDIGO FINAL ESTÁ CORRETO, ou seja, fazer exatamente aquilo que você desejava.

Vamos começar pelo óbvio, **teste o seu código**. Você não terá como estar seguro de que o seu programa está correto se não testá-lo. Isso é essencial para garantir programas de maior qualidade.

Outras dicas farão com que você utilize construtores de uma maneira que diminua a chance da inserção de erros no seu programa. Por exemplo, **evite alterar as variáveis de controle dos laços `for` dentro do corpo do laço**. Eu já deixei essa dica para você na aula sobre esse tipo de laço, mas não custa nada repetir. Outra boa prática é sempre **colocar o `default` no comando `switch`**, mesmo que você ache que não precisa. Isso destaca a ação a ser tomada nos casos excepcionais do bloco.

A próxima convenção é específica do JavaScript: sempre **faça comparações de igualdade e desigualdade usando os operadores === e `!==`**, respectivamente. Isto porque os operadores de comparação == e != sempre convertem os valores envolvidos

em tipos correspondentes antes da comparação. Essa conversão, se acontecer, pode ter comportamentos diferentes daquilo que você espera que aconteça.

Por fim, sempre **trate parâmetros com valores `undefined`**. Talvez você tenha observado durante esta disciplina, quando tentava resolver os exercícios, que se uma função é chamada com um argumento ausente, o valor desse argumento é `undefined`. Isso pode quebrar o seu código. Por isso, é importante que você tenha em mente que os argumentos podem ter valores `undefined` e saber o que fazer em casos semelhantes. Logo, uma boa opção é lançar erros nesses casos.

Agora, vamos ver como MELHORAR A EFICIÊNCIA DE SEUS PROGRAMAS, embora na sua carreira de programador você irá adquirir diversos conhecimentos que ajudarão a melhorar ainda mais a eficiência dos mesmos. Deixarei para você mais três dicas, além daquelas que já foram dadas ao longo da disciplina.

Primeiramente, **evite chamar funções nos testes de laços**. Observe o exemplo do slide (Figura 5). Note que a cada iteração do laço teremos que chamar a função `maximo` para saber se o laço deve parar. Essa função percorrerá o array `numeros` para poder retornar o valor máximo que, neste exemplo, será sempre 4. Portanto, não há necessidade de fazer essa chamada à função `maximo` a cada iteração.

Figura 5 - Chamando Funções nos Testes de Laços

```
function entrada(){
  var numeros = [3, 4, 2];
  var saida = "";
  for (var i = 0; i < maximo(numeros); i++) {
    var m = prompt("Forneça uma entrada: ");
    saida = saida + m + "<br>";
  }
  document.getElementById("resultado").innerHTML = saida;
}
```

Então, por que não calcular esse valor antes do laço, armazenando-o em uma variável, digamos `max`, e usar essa variável na condição do laço? Veja no slide (Figura 6) como isso pode ser feito.

Figura 6 - Evitando Chamar Funções nos Testes de Laços

```
var numeros = [3, 4, 2];
var saida = "";
var max = maximo(numeros);
for (var i = 0; i < max; i++) {
  var m = prompt("Forneça uma entrada: ");
  saida = saida + m + "<br>";
}
```

Note que nesta versão chamamos a função `maximo` apenas uma vez. Isso é muito mais eficiente e pode fazer uma diferença enorme em alguns programas.

Baixe os arquivos **15_5 Loop.html** e **15_5 Loop.js** e faça a alteração necessária para obter essa otimização.

Código 1 - 15_5 Loop.html e 15_5 Loop.js

```
1  <html >
2    <head>
3      <meta charset="UTF-8" />
4      <title>Programação Estruturada - Aula 15</title>
5    </head>
6    <body>
7      <noscript>Seu navegador não suporta JavaScript ou ele está desabilitado.</noscript>
8
9      <h1>Loops</h1>
10
11     <p id="resultado"></p>
12
13     <script src="script.js"></script>
14   </body>
15 </html>
16
```

```
1  entrada();
2
3  function entrada(){
4   var numeros = [3, 4, 2];
5   var saida = "";
6   for (var i = 0; i < maximo(numeros); i++) {
7     var m = prompt("Forneça uma entrada: ");
```

```
8      saida = saida + m + "<br>";
9    }
10   document.getElementById("resultado").innerHTML = saida;
11 }
12
13 function maximo(arr) {
14     var max = 0;
15     for (i = 0; i < arr.length; i++) {
16       if (arr[i] > max) {
17         max = arr[i];
18       }
19     }
20     return max;
21 }
22
```

A próxima dica também diz respeito a laços: **percorra laços apenas o necessário** para resolver o problema em questão. Observe o exemplo do slide (Figura 7). Note que o laço percorrerá todo o array mesmo que ele já tenha encontrado o índice do elemento procurado.

Figura 7 - Um Laço Ineficiente

```
39  function busca_linear(a,e) {
40      var indice = -1;
41
42      for (var i = 0; i < a.length; i++) {
43          if (e === a[i]) {
44              indice = i;
45          }
46      }
47
48      if (indice === -1) {
49          throw "Elemento " + e + " não ocorre no array";
50      }
51
52      return indice;
53  }
```

Baixe os arquivos **15_6 Busca Linear.html** e **15_6 Busca Linear.js** e utilize a ferramenta de depuração para observar que, de fato, todo o array é percorrido mesmo que já tenhamos encontrado o elemento.

Código 2 - 15_6 Busca Linear.html e 15_6 Busca Linear.js

```
1  <html >
2     <head>
3        <meta charset="UTF-8" />
4        <title>Programação Estruturada - Aula 15</title>
5     </head>
6     <body>
7        <noscript>Seu navegador não suporta JavaScript ou ele está desabilitado.</noscript>
8
9        <h1>Busca Linear</h1>
10
11       Entrada: <input type="number" id="entrada" value="">
12       <button onclick="inserir()">INSERIR</button>
13       <button onclick="limpar()">LIMPAR</button>
14       <button onclick="buscar_elemento()">BUSCAR</button>
15       <p id="array"></p>
16       <p id="resultado"></p>
17
18       <script src="script.js"></script>
19    </body>
20 </html>
21
```

```
1  var numeros = [];
2  var elemento;
3
4  function inserir() {
5     var x = Number(document.getElementById("entrada").value);
6     numeros.push(x);
7     imprimir();
8  }
9
10 function limpar() {
11    numeros = [];
12    imprimir();
13 }
14
15 function buscar_elemento() {
16    do {
17       elemento = prompt("Digite um número:");
18    } while (elemento == "" || isNaN(elemento));
19
```

```
20     elemento = Number(elemento);
21 
22     var resposta = "";
23 
24     try {
25         var indice = busca_linear(numeros, elemento);
26         resposta = "Elemento " + elemento + " ocorre no índice " + indice + "<br>";
27     } catch (erro) {
28         resposta = erro + "<br>";
29     }
30 
31     document.getElementById("resultado").innerHTML = resposta;
32 }
33 
34 function imprimir() {
35     var resposta = "Array  = [" + numeros + "] <br>";
36     document.getElementById("array").innerHTML = resposta;
37 }
38 
39 function busca_linear(a,e) {
40     var indice = -1;
41 
42     for (var i = 0; i < a.length /*&& indice === -1*/; i++) {
43         if (e === a[i]) {
44             indice = i;
45         }
46     }
47 
48     if (indice === -1) {
49         throw "Elemento " + e + " não ocorre no array";
50     }
51 
52     return indice;
53 }
54 
55 function strings() {
56     var a = "Olá Mundo";
57     var b = new String("Olá Mundo");
58 }
59 
```

É claro que podemos otimizar esse comportamento. Apresentei a solução no slide (Figura 8). Notou a diferença? Ela é sutil, mas muito importante.

Figura 8 - Um Laço Eficiente

```javascript
function busca_linear(a,e) {
    var indice = -1;

    for (var i = 0; i < a.length && indice === -1; i++) {
        if (e === a[i]) {
            indice = i;
        }
    }

    if (indice === -1) {
        throw "Elemento " + e + " não ocorre no array";
    }

    return indice;
}
```

Eu simplesmente adicionei uma condição no laço que indica que o elemento ainda não foi encontrado. No nosso caso, sabemos que se o índice ainda é -1 é porque o elemento ainda não foi encontrado. Portanto, ao adicionarmos essa condição ao laço, ele só continuará se o elemento ainda não foi encontrado. Caso contrário, ou seja, se o elemento foi encontrado, o índice será diferente de -1 e o laço terminará. Sugiro que agora você altere os arquivos **15_6 Busca Linear.html** e **15_6 Busca Linear.js** e utilize a ferramenta de depuração para observar que, de fato, o laço termina quando o elemento é encontrado.

Código 3 - 15_6 Busca Linear.html e 15_6 Busca Linear.js

```html
<html >
  <head>
    <meta charset="UTF-8" />
    <title>Programação Estruturada - Aula 15</title>
  </head>
  <body>
    <noscript>Seu navegador não suporta JavaScript ou ele está desabilitado.</noscript>

    <h1>Busca Linear</h1>

    Entrada: <input type="number" id="entrada" value="">
    <button onclick="inserir()">INSERIR</button>
```

```
13        <button onclick="limpar()">LIMPAR</button>
14        <button onclick="buscar_elemento()">BUSCAR</button>
15        <p id="array"></p>
16        <p id="resultado"></p>
17
18        <script src="script.js"></script>
19    </body>
20 </html>
21
```

```
1  var numeros = [];
2  var elemento;
3
4  function inserir() {
5     var x = Number(document.getElementById("entrada").value);
6     numeros.push(x);
7     imprimir();
8  }
9
10 function limpar() {
11    numeros = [];
12    imprimir();
13 }
14
15 function buscar_elemento() {
16    do {
17       elemento = prompt("Digite um número:");
18    } while (elemento == "" || isNaN(elemento));
19
20    elemento = Number(elemento);
21
22    var resposta = "";
23
24    try {
25       var indice = busca_linear(numeros, elemento);
26       resposta = "Elemento " + elemento + " ocorre no índice " + indice + "<br>";
27    } catch (erro) {
28       resposta = erro + "<br>";
29    }
30
31    document.getElementById("resultado").innerHTML = resposta;
32 }
33
34 function imprimir() {
35    var resposta = "Array  = [" + numeros + "] <br>";
36    document.getElementById("array").innerHTML = resposta;
37 }
38
```

```
39 function busca_linear(a,e) {
40    var indice = -1;
41
42    for (var i = 0; i < a.length /*&& indice === -1*/; i++) {
43       if (e === a[i]) {
44          indice = i;
45       }
46    }
47
48    if (indice === -1) {
49       throw "Elemento " + e + " não ocorre no array";
50    }
51
52    return indice;
53 }
54
55 function strings() {
56    var a = "Olá Mundo";
57    var b = new String("Olá Mundo");
58 }
59
```

A última dica desta videoaula também é específica do JavaScript: **sempre trate números, textos e booleanos como valores primitivos, nunca como objetos**. Isto porque a declaração desses tipos como objetos diminui a velocidade de execução e produz efeitos colaterais desagradáveis.

Chegamos ao final de nossa última aula. Nela, você aprendeu como funciona o tratamento de erros em JavaScript, conheceu um estilo de código, ou seja, como estruturar o seu código para que ele seja visualmente mais fácil de ler e entender. E, finalmente, conheceu algumas boas práticas de programação, regras simples, mas que terão um impacto importante na sua produção.

Desta vez, deixarei apenas UMA ATIVIDADE: analise todos os exemplos e exercícios feitos por você ao longo da disciplina. Aplique a eles os conceitos de tratamento de erros, estilo de código e boas práticas aprendidos nesta aula.