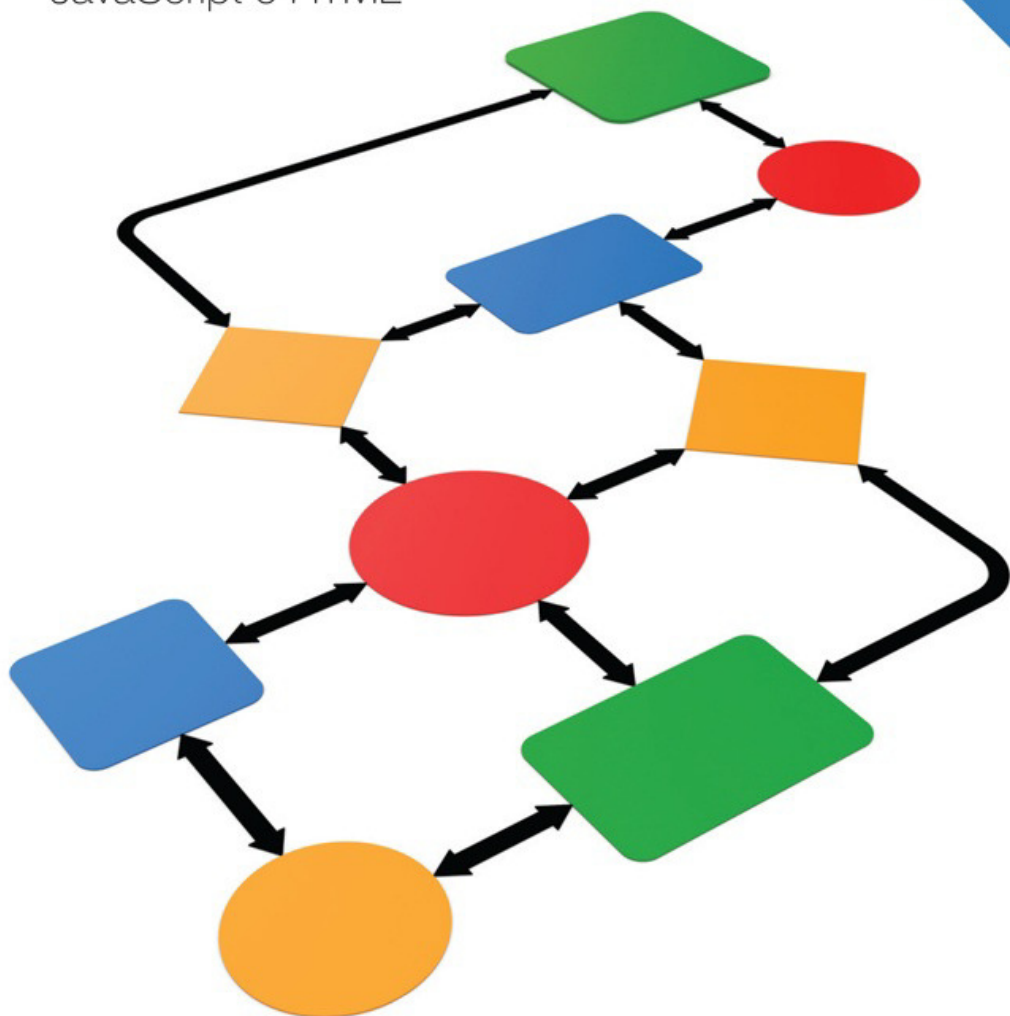


Lógica de Programação

Crie seus primeiros programas usando JavaScript e HTML



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-66250-22-0

EPUB: 978-85-5519-008-7

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

DEDICATÓRIAS

"Aos meus pais, sempre caminhantes" – Paulo Silveira

"Aos meus pais, irmão e irmã que sempre me mostraram o caminho" – Adriano Almeida

AGRADECIMENTOS

Esperamos que você aproveite o livro. Foi feito com muita atenção, para que o nível de dificuldade sempre suba, sem desanimar o iniciante. Ao término, você será capaz de criar suas próprias estruturas gráficas e até mesmo um simples jogo de computador.

Muitas decisões não foram fáceis: preterir o `console.log` em favor do desengonçado `document.write`, explicar funções antes de `ifs` e `fors`, definir funções através de atribuição em vez da sintaxe mais comum. Esses são apenas algumas das questões que apresentavam vantagens e desvantagens para uma melhor abordagem didática. Realizamos nossas escolhas depois de alguns testes com pessoas leigas em computação.

Um agradecimento especial ao Leonardo Wolter e Francisco Sokol pela base dos exercícios com animações. Outro ao Erich Egert, pelo teste do livro com alunos e diversas sugestões para novos desafios.

Sumário

1 Comece a programar hoje	1
1.1 Converse com seu navegador	2
1.2 Criando seu próprio arquivo HTML	6
1.3 Um pouco mais de HTML	8
1.4 Dê olá ao mundo	11
1.5 Revise o código: seu primeiro programa	13
1.6 Utilize o Chrome e o Notepad++	15
1.7 Socorro! Meu programa não funciona. Conheça e use o console do Chrome	16
1.8 Socorro! A acentuação não está funcionando corretamente	20
2 Comunique-se com o usuário	22
2.1 Dê olá ao mundo de outras formas	22
2.2 Trabalhe com números	23
2.3 Revisando o seu código: trabalhando com números	26
2.4 Organize seus dados em variáveis	28
2.5 Reescrevendo a média de idade dos seus amigos	30
2.6 Revisando o seu código: organize-se com variáveis	30
2.7 Pare de escrever BR tantas vezes!	33

2.8 Revise o código: crie sua primeira função	36
2.9 Funções passando informações e chamando outras funções	
2.10 Revise o código: usando a função mostra	39 ³⁷
2.11 Mostrando mensagens secretas, apenas para o programador	
2.12 Para saber mais: comentários	43 ⁴¹
2.13 Compartilhe seu código com seus amigos!	44
3 Pratique resolvendo problemas do seu dia a dia	47
3.1 Como está seu peso? Saudável?	47
3.2 Utilize uma função para calcular o IMC de cada amigo	48
3.3 Revise o código: calculando o IMC	51
3.4 Trabalhe com dados capturados: pergunte a altura e peso do usuário	53
3.5 Exercícios: pergunte os dados do usuário para calcular o IMC	
3.6 Descubra quantos dias seus amigos já viveram	55 ⁵⁴
3.7 Você já entendeu a ordem das chamadas das funções?]	57
3.8 Utilize o console do Chrome para fazer testes!	58
4 Execute códigos diferentes dependendo da condição	61
4.1 Quantos pontos tem seu time de futebol?	61
4.2 Verifique a situação do seu time de futebol	64
4.3 Revisando nosso código: pontos do campeonato	66
4.4 O seu IMC está ideal?	68
4.5 Jogo: adivinhe o número que estou pensando	71
4.6 Revisando seu código: o jogo da adivinhação	75
5 Como repetir tarefas do programa?	78
5.1 Quando serão as próximas copas do mundo?	78
5.2 Realize o loop somente em determinadas condições	81

5.3 Revise seu código: mostre os anos de copas até cansar	84
5.4 Caracteres e números, qual é a diferença afinal?	86
5.5 Revise seu código: transforme texto em números	89
5.6 Praticando mais um pouco: faça tabuadas	91
5.7 Aprenda uma forma diferente de mostrar a tabuada: o comando for	93
5.8 Reescrevendo a tabuada com o for	97
5.9 A média de idades, mas de uma forma mais interessante	100
5.10 Jogo: mais chances para adivinhar o número que estou pensando	104
5.11 Revisando nosso código: o jogo da adivinhação dos números	108
5.12 Exercícios: trabalhando com um loop dentro do outro	110
6 Arrays: trabalhe com muitos dados	115
6.1 Integre o JavaScript com HTML	115
6.2 Revisando uso de HTML e criando o jogo	118
6.3 Facilite o jogo da adivinhação colocando mais números!	121
6.4 Evite os número repetidos no Bingo	126
7 Apêndice — Gráficos para deixar tudo mais interessante	130
7.1 Desenhe linhas e figuras	130
7.2 Criando todo tipo de imagem	134
7.3 Não vou conseguir lembrar de tudo isso! APIs e bibliotecas	
7.4 Revise seus primeiros passos com o canvas	141 ¹³⁷
7.5 Cansei de repetir código! Funções novamente	146
7.6 Loops e funções para nos ajudar	147
7.7 Para saber mais: passe uma função para uma... função!	151

8 Apêndice – Animações e pequenos jogos	160
8.1 Crie uma lousa capturando o movimento do mouse	160
8.2 Exercícios para nossa tela de desenho	165
8.3 Crie animações	165
8.4 Revise e faça novas animações	168
8.5 Desafio: o jogo do tiro ao alvo	171
9 Últimas palavras – Além da lógica de programação	174
9.1 Objetos	174
9.2 Boas práticas que foram violadas durante o aprendizado	176
9.3 Pratique muito!	178
9.4 Continue seus estudos	179

Versão: 21.0.10

COMECE A PROGRAMAR HOJE

Não importa sua idade, profissão e objetivo: programar é mais do que divertido, é um constante desafio. Queremos tornar o programa mais rápido, mais legível, mais elegante e mais útil. Prepare-se para encontrar problemas, quebra-cabeças e questões o tempo todo.

Seu aprendizado aqui será útil não apenas para começar a criar uma página web. Você estará preparado para enfrentar as fórmulas do Excel, desenvolver suas próprias pequenas ferramentas, entender o funcionamento das aplicações que utiliza na internet e, quem sabe, criar um programa para seu próprio celular e dos seus amigos.

O segredo, sem dúvida, é praticar. Não se limite apenas com os exemplos e exercícios do livro. Vá além, deixe sua curiosidade guiar a criação de novos programas.

Siga os capítulos passo a passo. Assim que começar a aprender mais, ofereceremos exercícios e desafios oficiais. Não fique apenas na leitura! A prática é o segredo. Faça todas as sugestões e veja você mesmo o resultado. Caso fique curioso, troque os dados,

modifique as rotinas, experimente, invente. A programação nos dá esse poder de criação. Crie!

Está com dúvidas? Tem dois locais para resolver seu problema. Um é uma lista de discussão, criada especialmente para este livro: <http://forum.casadocodigo.com.br>.

O outro é usar o maior portal de programação do país, o G.U.J.: <http://www.guj.com.br/>.

Em ambos os casos, lembre-se de ser bem específico na sua pergunta, dando detalhes dos problemas e a lista organizada do seu código.

1.1 CONVERSE COM SEU NAVEGADOR

Há muito o que aprender. O mais importante é que você possa rapidamente escrever seus próprios programas, e depois **executá-los** para ver o resultado. Existem muitas linguagens de programação, e todas são alternativas viáveis para um primeiro contato. Escolhemos uma linguagem para você: o **JavaScript**, que possui vantagens e desvantagens como todas as outras, mas para o aprendizado ela é muito adequada: não haverá necessidade de instalar nada para começar a programar.

Abra uma página qualquer. Pode ser o site da casa do código (<http://www.casadocodigo.com.br>), o site da Caelum (<http://www.caelum.com.br>), ou do seu portal preferido. Para fazer isto, você utilizou algum navegador (*browser*), como o Internet Explorer, o Firefox e o Chrome, mas como ele fez toda essa magia de mostrar essa página bonita para nós?

Na realidade, não há mágica nenhuma envolvida nisso. O navegador apenas obedece a instruções que alguém deu para ele. Alguém disse para ele que aquelas imagens deveriam aparecer, que determinados textos precisavam ser exibidos, que a cor de fundo da página é algum tom de cinza, e que quando você clica em um botão, uma mensagem surge para você, e assim por diante.

Quem fez isso? O programador, muitas vezes também chamado de desenvolvedor, que é justamente quem sabe dar as instruções para o navegador realizar. Todos esses navegadores vão pegar essas instruções na forma de um **código**, e vão apresentá-lo na tela de uma forma agradável. *O que viria a ser um código?*

Nesse mesmo site que você entrou, vamos ver qual foi o código utilizado. Para isso, você deve seguir passos diferentes de acordo com o navegador.

No Chrome, clique no ícone da ferramenta, depois *Ferramentas e Exibir código fonte*. CTRL+U é o atalho para isso.

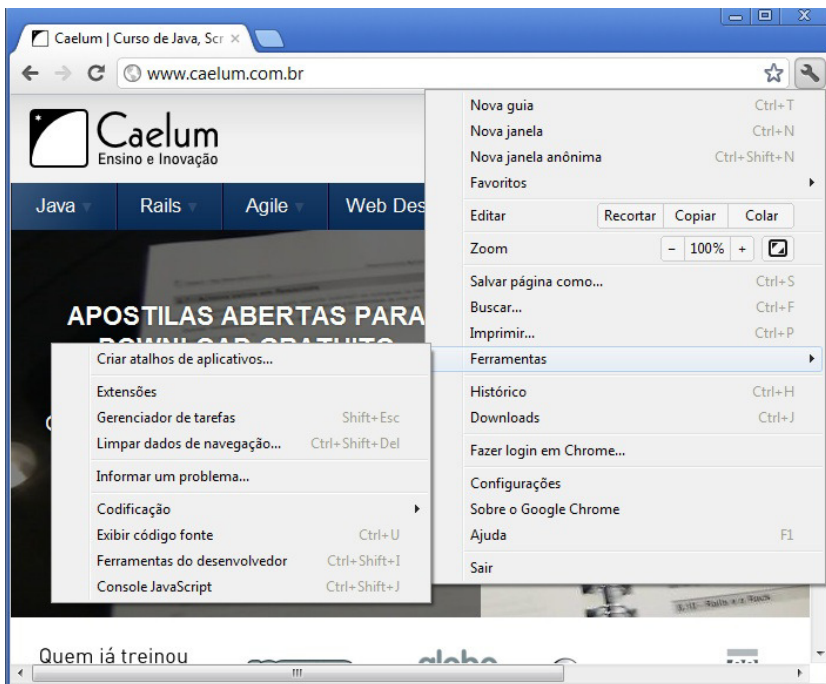


Figura 1.1: Menu para exibir código fonte no Chrome

No Firefox, clique no botão do menu, *desenvolvedor web*, e depois *código fonte*. CTRL+U também funciona nesse navegador.

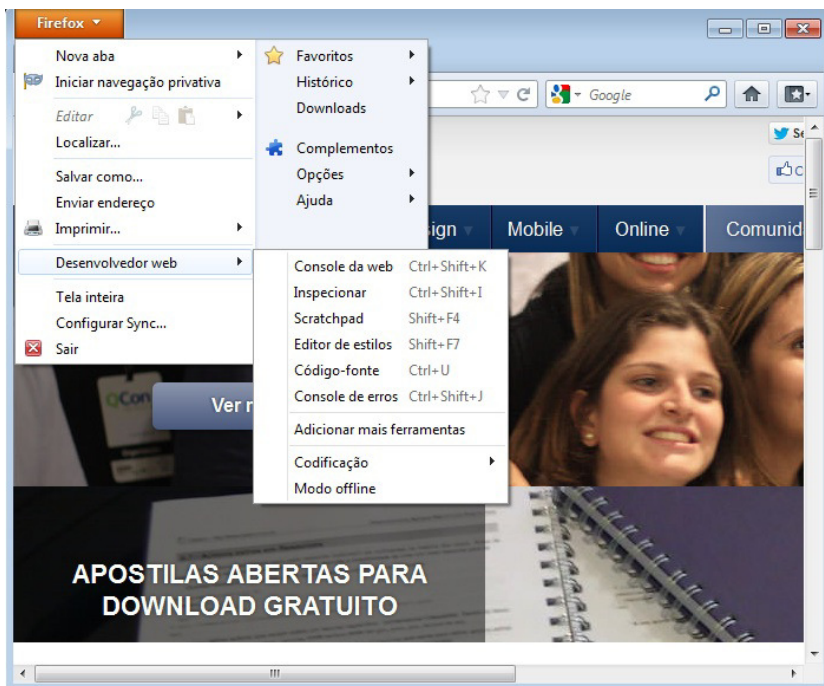


Figura 1.2: Menu para exibir código fonte no Firefox

No Internet Explorer 9, dê um clique com o botão direito no fundo da página e escolha *Visualizar código fonte*.

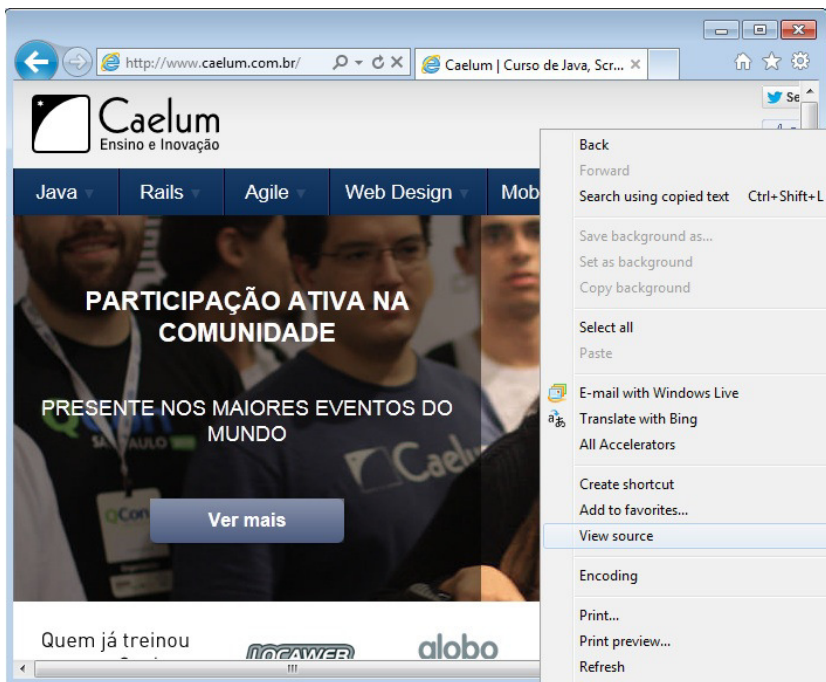


Figura 1.3: Menu para exibir código fonte no Explorer

Parece complicado? O navegador (*browser*) pegou todo esse código, e gerou aquela representação, agradável aos nossos olhos. Podemos nós mesmos criar algo assim para apresentar os dados que desejarmos. Isto é, podemos **criar nossa própria página**, que vai interagir com quem a estiver acessando, o **usuário**.

1.2 CRIANDO SEU PRÓPRIO ARQUIVO HTML

Para criar nosso primeiro código, abra um editor de texto comum, como o Bloco de Notas (*Notepad*) do Windows, ou o gedit do Linux. Editores de texto como o Word não ajudam muito neste caso, pois eles gravam o arquivo de uma forma diferente da

qual o seu navegador está preparado. Digite o seguinte texto:

Meu primeiro teste!

`<h3>Seria isso um programa?</h3>`

Salve o arquivo em uma pasta de fácil acesso, como Meus Documentos ou a própria Área de Trabalho (Desktop), com o nome `minha_pagina.html` .

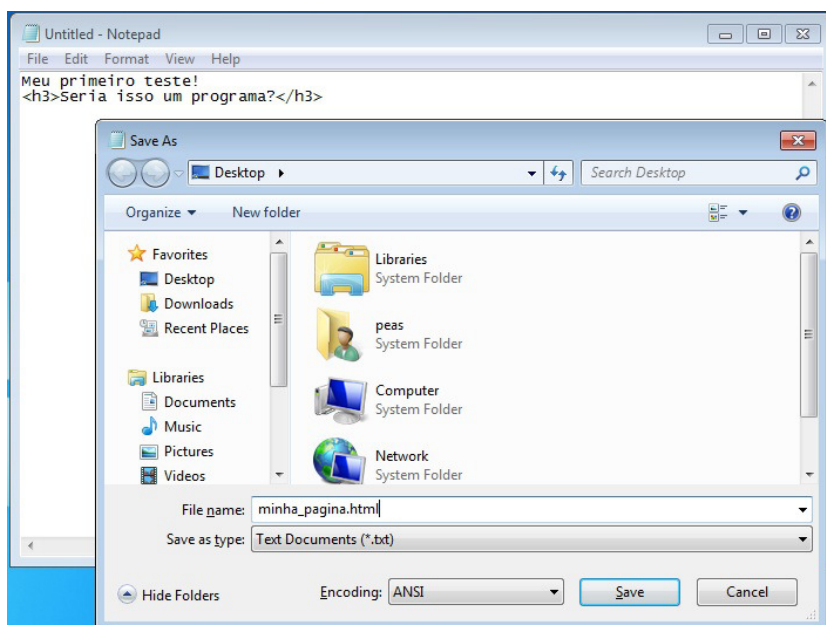


Figura 1.4: Notepad criando o arquivo

Vá ao diretório onde você gravou o arquivo, e dê dois cliques sobre ele. O seu navegador abrirá e teremos o resultado:

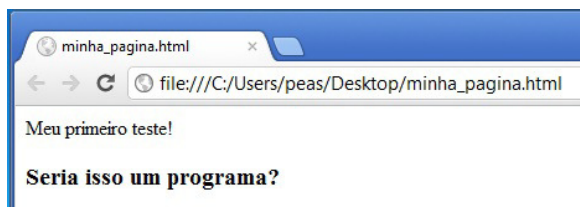


Figura 1.5: Resultado dentro do navegador

Esse é o resultado usando o Google Chrome. Caso você tenha mais de um navegador instalado, pode utilizar o clique da direita no arquivo e selecionar o *abrir com* para escolher um outro navegador

Um arquivo HTML nos permite não apenas apresentar informações que foram colocadas dentro dele, como também realizar operações, pedir informações e **executar comandos**. Veremos isso em breve.

1.3 UM POUCO MAIS DE HTML

Nossa página, por enquanto, só exibe textos fixos (chamados de *conteúdo estático*). Podemos alterá-la para exibir informações diferentes, usando **tags** para mudar a forma com que os dados são representados. Abra novamente seu arquivo `html`. Você pode até deixar o editor aberto o tempo inteiro, pois modificaremos o arquivo com frequência. Vamos alterá-lo:

Meu primeiro teste!

```
<h3>Seria isso um programa?</h3>
```

```
<strong>Não</strong> consigo fazer nada além de mostrar conteúdo  
fixo?
```

Salve o arquivo e abra novamente a página (se ela já estiver

aberta, basta clicar em atualizar no seu navegador). O resultado é previsível:



Figura 1.6: Resultado das nossas mudanças

Não se preocupe com os nomes das diferentes tags (tags são, por exemplo, `` , `<h3>` etc). O importante, nesse momento, é ver como funciona um HTML no geral: por meio das tags, realizamos marcas (*markups*) no texto para enriquecê-lo.

Esse tipo de texto é conhecido como hipertexto, por permitir navegar entre diferentes páginas e sites. Podemos, por exemplo, adicionar um *link* (ponteiro) para um site, como o da Casa do Código:

Meu primeiro teste!

`<h3>`Seria isso um programa?`</h3>`

``Não`` consigo fazer nada além de mostrar conteúdo fixo?

Conheça o site da nossa editora:

``Clique aqui``!

Salve o arquivo com essas duas novas linhas e abra a página no seu navegador. Lembrando de que, para abri-la, dê dois cliques no arquivo, ou se ele já estiver aberto, atualize a página clicando no ícone de recarregamento (*reload*).

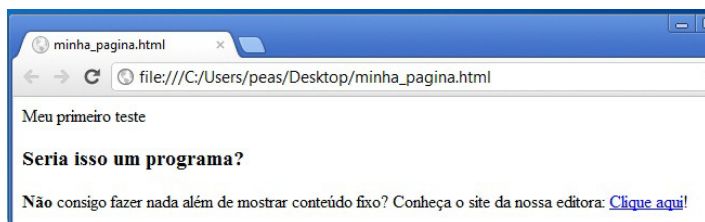


Figura 1.7: Criando um link para outra página

A forma de escrever HTML com aqueles sinais de `<` e `>`, também conhecido como a sintaxe do HTML, pode assustar a primeira vez. Assim como o uso do `<h3> </h3>`, a tag `<a ...> ` envolve um pedaço do nosso texto. Mais ainda, ela possui um *atributo*, nesse caso o tal do `href`, indicando para onde o navegador deve ir se alguém clicar neste link. É comum que tags HTML possuam diversas propriedades diferentes, modificando a visualização e o comportamento de determinados trechos do documento. Uma tag mais o seu conteúdo forma um **elemento HTML**, por exemplo, o `Não`.

Há muitas tags. Não se preocupe em decorá-las. Você perceberá que, com o passar do tempo e a prática, decidir qual tag deverá ser utilizada se tornará um processo natural.

Você já reparou que o navegador não pulou de linha onde sugerimos? Uma das tags que pode ajudar nisso é a `
`. Experimente. Diferente das outras tags que vimos até aqui, ela deve ser usada sem o estilo de abrir e fechar, aparecendo apenas uma única vez para cada uso. Vamos utilizá-la com frequência mais adiante.

Está curioso com a sigla HTML? Ela significa Hypertext Markup Language (linguagem de marcação para hiper texto). Ou, em uma tradução bem aberta, uma linguagem que possui tags para marcar documentos do tipo hipertexto.

Documento hipertexto é um que pode ter links para outros. O HTML sozinho nada mais é que uma forma de marcação (por meio das tags). Veremos agora como incrementá-la.

1.4 DÊ OLÁ AO MUNDO

Para ter essa interação com o usuário, utilizaremos o **JavaScript**, uma linguagem que nos permite escrever códigos interessantíssimos. Altere seu arquivo `html` , adicionando as seguintes linhas no final dele:

```
<script>
alert("podemos fazer mais com JavaScript!");
</script>
```

Salve o arquivo. Abra-o no seu navegador, dando um duplo clique na página ou recarregando-a. O resultado deve ser que uma caixa de texto aparece com uma mensagem. Essa caixa é conhecida também como *pop-up*. Além dela, o texto que já conhecemos será mostrado:

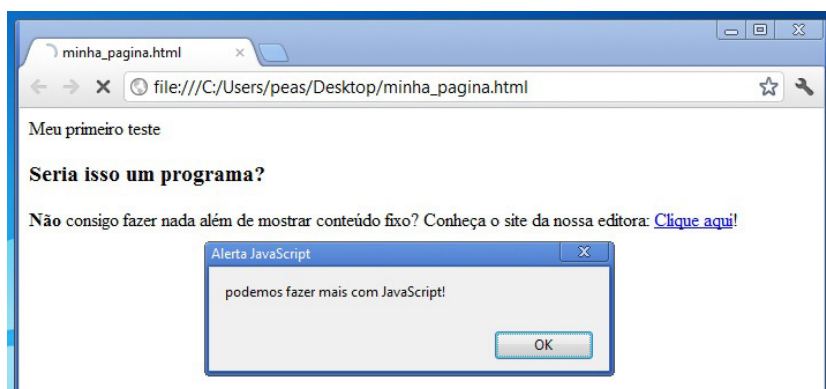


Figura 1.8: Começando a interagir mais

Vai ser muito frequente repetir o processo de alteração do nosso código. Esse processo é: adicionar novas linhas no arquivo html, salvá-lo, e abri-lo novamente no navegador. Como vimos, em vez de dar dois cliques na página, há também o botão de *atualizar* (*Reload*), que puxará as novas informações do nosso arquivo, executando tudo de novo.

Sim, você já está programando! Parabéns, esse foi seu primeiro código usando a linguagem JavaScript. Passos pequenos e muita prática durante a leitura deste livro o levará mais longe do que imagina.

Algo deu errado? No final dessa lição, aprenderemos a usar algumas ferramentas do navegador para nos auxiliar na descoberta dos problemas que podem acontecer. Além disso, todas as lições possuem seções de revisão de código, onde você poderá comparar o código que você fez com o que era esperado.

Repare que nosso código possui uma série de caracteres não usuais, como parênteses e ponto e vírgula. Será que não

funcionaria fazer um simples `alert "olá mundo"` ? Cada linguagem tem seu conjunto de regras, sua **sintaxe**, e que devem ser respeitadas para que o código possa ser executado. No caso do JavaScript, há toda uma especificação que os navegadores seguem (ou deveriam seguir).

Caso você esteja estudando por uma versão digital do livro, pode ter ficado tentado a copiar e colar o código desse `alert` . Não faça isso. Uma parte muito importante do aprendizado é errar a sintaxe de uma linguagem, além de praticá-la mesmo quando óbvia. Caso você tenha errado algo, o navegador emitirá uma mensagem de erro. Às vezes, ela pode ajudar bastante, mas dependendo do navegador, pode levá-lo ao desespero.

1.5 REVISE O CÓDIGO: SEU PRIMEIRO PROGRAMA

Vimos como escrever nosso primeiro código. Durante todas as lições, sempre teremos seções de revisão. É por elas que você pode verificar o que fez até agora, e também seguir novamente os passos, de maneira mais sucinta, para consolidar seus novos conhecimentos.

Vamos fazer novamente? Abra o editor de texto e digite o seguinte código, que já conhecemos:

Meu primeiro teste!

```
<h3>Seria isso um programa?</h3>
```

```
<strong>Não</strong> consigo fazer nada além de mostrar conteúdo  
fixo?
```

Conheça o site da nossa editora:

```
<a href="http://www.casadocodigo.com.br">Clique aqui</a>!
```



```
<script>
alert("podemos fazer mais com JavaScript!");
</script>
```

Agora, salve-o. Como é um arquivo novo, o editor vai perguntar para você onde e com que nome quer gravá-lo. Escolha uma pasta de fácil acesso e um nome significativo. No nosso caso, usamos como `minha_pagina.html` . Para não perder o que fez até aqui, pode gravar essa nova versão com outro nome, como por exemplo `meu_primeiro_programa.html` .

Agora encontre o arquivo e dê dois cliques nele. O navegador deve abrir, interpretando as tags HTML e executando nosso código JavaScript!

Vamos fazer alguns exercícios, baseado no que já aprendemos:

1) Edite o seu arquivo e adicione mais um alert. Além da mensagem `podemos fazer mais com JavaScript!` , coloque um outro `alert` escrevendo a data que você começou a programar. Lembre-se de salvar o arquivo e abri-lo no navegador. Caso seu navegador já esteja aberto com a sua página, basta atualizá-la.

2) Realize outros testes. Você pode ter mais de uma seção com a tag `<script>` e `</script>` , colocando outros `alert` lá? É importante ser curioso com seu próprio programa, ir além do que foi sugerido, explorando as outras possibilidades e limites.

Você pode realizar esses testes em outros arquivos para não misturar com os exercícios que já estão como você quer. Para isso, basta criar um novo arquivo, com outro nome, como `meus_testes.html` . Lembre-se de evitar a tentação de copiar e

colar um código que você já fez. É importante que você pratique escrever seus códigos.

1.6 UTILIZE O CHROME E O NOTEPAD++

Podemos utilizar qualquer um dos navegadores para aprender a programar. Mesmo assim, durante o livro, usaremos o Chrome como base para nossas imagens, atalhos, menus e dicas. Recomendamos fortemente que você faça o mesmo. Não há problema utilizar um outro, porém as mensagens de erro e a forma de apresentar o resultado podem variar um pouco.

Você pode fazer a instalação pelo site: <https://www.google.com/chrome>.

A instalação é muito simples, basta, depois do download, seguir os passos ao executar o programa de instalação. Há versões para Windows, Linux e Mac.

Para o editor de texto, o Bloco de Notas (Notepad) seria suficiente, porém é muito mais interessante usar um editor que nos ajude mais, colorindo alguns termos para facilitar a visualização do nosso código fonte, além de oferecer outros truques.

Para o Windows, recomendamos o Notepad++, que é gratuito e de código aberto: <http://notepad-plus-plus.org/>.

Clicando em *downloads*, há logo uma opção Notepad++ v6.1.3 Installer (pode ser uma versão mais atual).

Basta baixar este executável e abri-lo. Durante a instalação, há a opção de escolher pelo português. Agora você pode editar nosso html através desse editor, dando um clique da direita no arquivo

e escolhendo *Edit with Notepad++*:

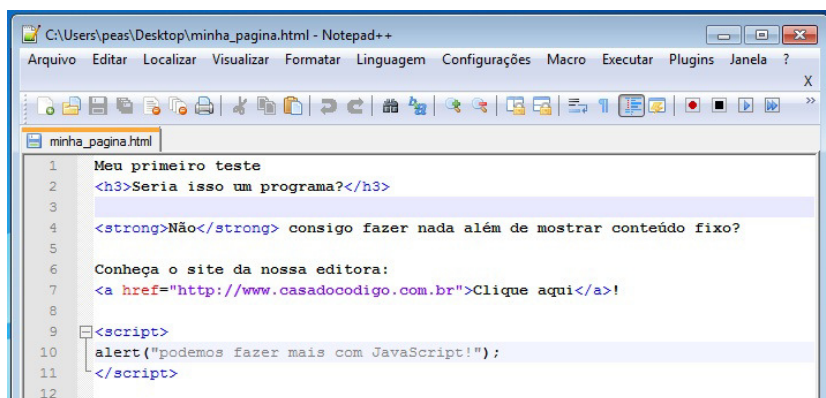


Figura 1.9: Utilizando o Notepad++ para facilitar nosso trabalho

Muita diferença, não? Esse editor utiliza cores diferentes para tornar nosso código mais legível, além de oferecer abas para abrir múltiplos arquivos, numeração de linhas e muito mais.

Usaremos o Notepad++ nas imagens deste livro, juntamente com o Google Chrome e o Windows 7. Caso você utilize Linux ou Mac, provavelmente já possui um editor de sua preferência. Recomendamos o Sublime para Linux e Mac. Claro, você pode usar o vim, emacs ou qualquer outra ferramenta que achar mais adequada.

No livro, usamos algumas imagens capturadas no Windows, outras no Mac e outras no Linux. Há poucas diferenças notáveis.

1.7 SOCORRO! MEU PROGRAMA NÃO FUNCIONA. CONHEÇA E USE O CONSOLE DO CHROME

Não se apavore com os erros. É importante saber enfrentá-los. Entender uma mensagem de erro é fundamental. Há sempre também fóruns e listas de discussão onde você pode pedir ajuda. Lembre-se de descrever muito bem seu problema e qual é a mensagem de erros, mas ao mesmo tempo, é necessário ser sucinto.

O fórum do GUJ é bastante conhecido na comunidade de desenvolvedores brasileiros de diversas linguagens. Utilize-o para tirar suas dúvidas: <http://www.guj.com.br>.

Há alguns erros bem comuns. Um exemplo é não usar os parênteses em um `alert`. Se você fizer isso, nada que estiver dentro desta tag de `script` funcionará e não haverá uma mensagem de erro! Faça o teste, adicionando uma segunda chamada ao `alert` de forma errada:

```
<script>
alert("podemos fazer mais com JavaScript!");
alert "chamando sem parenteses";
</script>
```

O navegador não vai nem mostrar o primeiro `alert`, onde não há erro nenhum! Como então descobrir o problema? Há uma ferramenta do Chrome para nos auxiliar nessa tarefa.

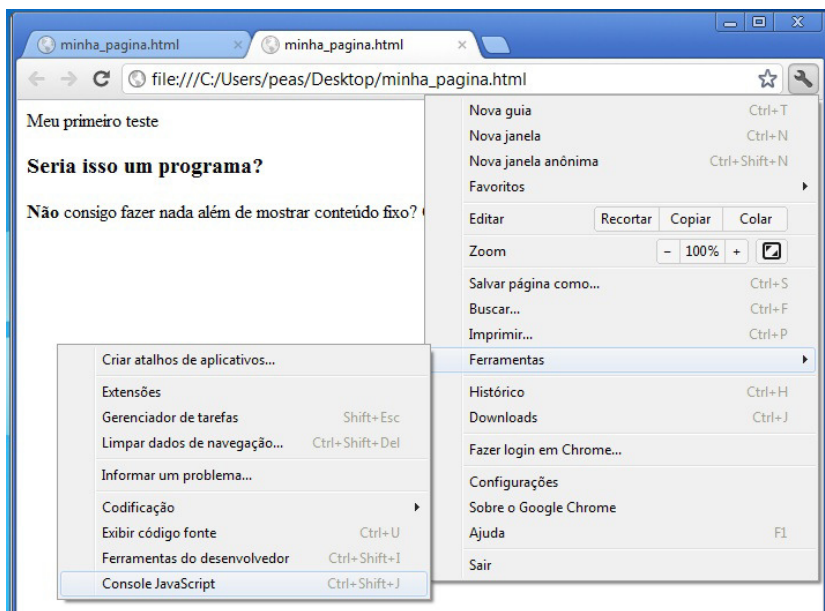


Figura 1.10: Selecionando o menu para abrir o Console JavaScript

Clique no ícone de menus/ferramentas. Ele fica no topo superior direito do Chrome. Depois, acesse o menu *Ferramentas* (*Tools*) e, por último, *Console JavaScript*, como na figura. A seguinte janela aparecerá, indicando o **erro de sintaxe**:

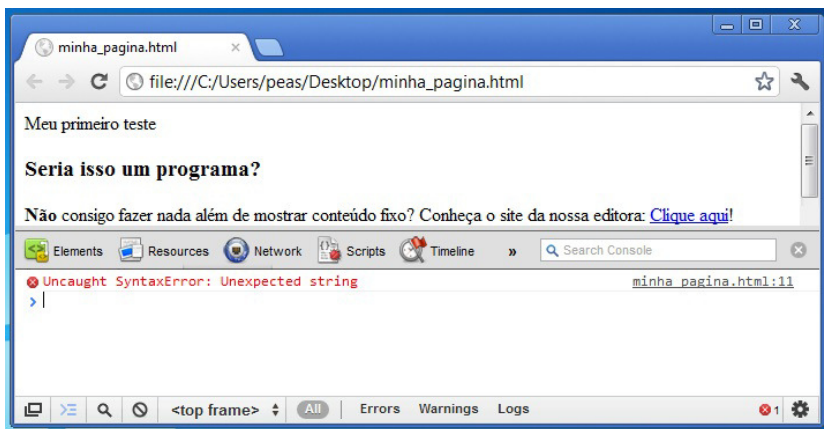


Figura 1.11: Console JavaScript do Chrome aberto

Neste caso, ele está avisando que temos uma *"string não esperada"*. Ajudou? Algumas vezes, as mensagens de erro podem não ser muito claras, mas já ajudam bastante indicando o local do problema. Repare que, à direita da mensagem de erro, o navegador diz em que arquivo e em qual linha houve o problema. Você pode até mesmo clicar ali para ir diretamente ao trecho de código problemático.

Com essa mensagem de erro, o Chrome quer indicar que o "chamando sem parenteses" apareceu em um lugar não esperado, como se estivesse falando em uma "língua" que ele não entende. Esse tipo de erro é o que chamamos de **erro de sintaxe**, um erro na forma de se comunicar com quem entende da linguagem. Nesse caso em particular, ele não esperava essa mensagem fora de parênteses. Corrija o problema.

Faça testes com outros erros. Escreva `alert` de maneira errada, por exemplo, `alertar`.

Ao mesmo tempo, há algumas mudanças que não impactariam na execução do código. Por exemplo, poderíamos ter dado espaço entre `alert` e a abertura dos parênteses. E o ponto-e-vírgula, poderíamos omiti-lo? E as aspas? Teste, descubra. É importante experimentar além das nossas lições.

Para o Firefox, há um console semelhante acessando o menu *Ferramentas (Tools)*, *Web Developer*, e depois *Console Web*. No Internet Explorer 9, você pode encontrá-lo clicando no ícone de engrenagem e escolhendo *F12 Ferramentas de Desenvolvedor (Developer Tools)*.

1.8 SOCORRO! A ACENTUAÇÃO NÃO ESTÁ FUNCIONANDO CORRETAMENTE

Caso você esteja utilizando o Mac ou o Linux, haverá um pequeno problema: a acentuação não aparece corretamente. Quando gravamos arquivos, há diferentes formatos de armazenar os caracteres em *bytes*. O Chrome, por padrão, tentará lê-lo em uma codificação conhecida por *latin1*.

Se gravamos nosso arquivo em outro formato, precisamos colocar essa informação no HTML que possui uma série de tags que nem mesmo marcam o texto, mas dão meta informações ao arquivo. Caso você esteja usando um desses sistemas operacionais, acrescente a seguinte linha como sendo a primeira do seu arquivo:

```
<meta charset="UTF-8"> .
```

Pense nisso como uma propriedade de um arquivo do Word. São informações que servem não para o usuário final diretamente, mas sim para o navegador poder trabalhar melhor com aquele

arquivo.

COMUNIQUE-SE COM O USUÁRIO

Acabamos de conseguir fazer o primeiro programa nos obedecer e mostrar uma mensagem para o usuário, o que já é um grande passo. Vamos escrever códigos mais elaborados, divertidos e interessantes a partir de agora. Você vai começar a perceber o poder que a programação pode ter e já será um bom passo para que comece a exercitar sua própria criatividade, além de ganhar o hábito de programar. Preparado?

2.1 DÊ OLÁ AO MUNDO DE OUTRAS FORMAS

O `alert` é nossa primeira forma de comunicação com o usuário. Como podemos fazer para enviar duas mensagens? Basta executarmos duas vezes essa instrução. Crie um novo arquivo, que será gravado `comecando_javascript.html`, e coloque o seguinte conteúdo:

```
<script>
alert("olá mundo!");
alert("esse é meu segundo programa");
</script>
```

Você poderia ter utilizado o mesmo arquivo da lição anterior,

mas é bom aqui criarmos um novo para poder acompanhar nosso aprendizado. Lembre-se de salvá-lo e depois abri-lo no seu navegador.

Verifique o resultado. Pode ser um pouco trabalhoso ter de ficar clicando no botão de *OK* a cada novo `alert` . Imagine se tivéssemos 15 mensagens para serem mostradas? Você teria que dar 15 cliques no botão *OK*. Chato, não?

Há várias formas de se comunicar com o usuário pelo JavaScript, e uma delas é através do `document.write` . Crie um novo arquivo, o `programa.html` com o seguinte código:

```
<script>
document.write("olá mundo!");
document.write("esse é meu segundo programa");
</script>
```

Acesse a página e verifique o resultado.

Para não aparecer tudo em uma única linha, você pode usar a tag `
` , que já conhecemos, para quebrá-la. Em vez de fazer `document.write("olá mundo!");` , faça `document.write("olá mundo!**
**");` . Salve o arquivo e atualize a página. Entendeu a função da tag `
` ?

2.2 TRABALHE COM NÚMEROS

Por enquanto, só trabalhamos com mensagens fixas, estáticas, e com **sequências de caracteres** definidas entre os sinais das aspas.

"Casa do código" é uma sequência de caracteres. "Olá Mundo" é uma outra sequência de caracteres, assim como "Eu tenho 25 anos" . Mas podemos também trabalhar com números:

```
<script>
document.write("Minha idade é: ");
document.write(25);
</script>
```

Apenas as aspas se foram na segunda linha. Se estiver usando o *Notepad++*, verá que ele colore o número de forma diferente do que está entre aspas. Será então que não precisaríamos dela para mostrar as sequências de caracteres? Faça o teste sem elas e verá que, se não for apenas números, teremos um erro.

Mas por que utilizar um número sem as aspas? Afinal, poderíamos muito bem ter feito assim:

```
document.write("25");
```

Qual é a diferença de 25 para "25" ? Quando usamos as aspas, estamos dizendo ao JavaScript que queremos que isto seja encarado como uma sequência de caracteres, não como um simples número. A grande diferença estará na forma que a linguagem tratará cada um deles. Experimente fazer uma conta com números:

```
document.write(25 + 25);
```

Agora vamos fazer algo muito parecido, utilizando o mesmo operador, porém com duas sequências de caracteres:

```
document.write("25" + "25");
```

Esse teste é fundamental para você entender a diferença dos dois conceitos. No segundo caso, o operador `+` junta as duas sequências de caracteres. Esse processo de juntar sequências de caracteres é chamado de **concatenação**.

Em muitas linguagens, assim como no JavaScript, uma

sequência de caracteres entre aspas é chamada de `string` . Dizemos, portanto, que o `+` , além de somar números, concatena strings .

Você vai trabalhar com números ou sequências de caracteres (`string`)? Depende do que quer fazer. Com números, podemos trabalhar as operações matemáticas. Para saber uma estimativa do ano em que você nasceu, subtraímos o ano atual desse valor:

```
document.write("Eu nasci em: ");  
document.write(2012 - 25);
```

E se em vez de ter feito `document.write(2012 - 25)` , tivéssemos colocado `2012 - 25` todo entre aspas, fazendo `document.write("2012 - 25")` ? Qual é o resultado?

Apenas por uma questão de concisão, às vezes vamos omitir a tag `script` dos próximos programas, como acabamos de fazer. Você deve sim utilizá-las. Aliás, o que aconteceria com nosso programa no caso de não colocarmos essas instruções dentro da tag `script` ? Faça o teste.

Além do operador de subtração (`-`), há o de soma (`+`), multiplicação (`*`) e divisão (`/`). Você pode somar a sua idade a dos autores. Paulo tem 32 anos e Adriano tem 26:

```
document.write("A soma das nossas idades é: ");  
document.write(25 + 32 + 26);
```

Para calcular a média, basta dividirmos o resultado da soma

por 3:

```
document.write("A média das nossas idades é: ");  
document.write(25 + 32 + 26 / 3);
```

Verifique o resultado. Não é o esperado! A conta de divisão é calculada antes da soma, como na matemática da escola, logo o primeiro valor a ser calculado é $26 / 3$. Podemos utilizar parênteses para forçar a ordem desejada do cálculo, realizando primeiramente as somas:

```
document.write((25 + 32 + 26) / 3);
```

Os parênteses são usados mesmo quando a precedência dos operadores já trabalha conforme esperamos, pois pode facilitar a legibilidade do que queremos fazer.

Também é possível misturar números com strings, mas sempre com cuidado. O que acontece ao somá-los?

```
document.write("Minha idade é: " + 25);
```

A sequência de caracteres "Minha idade é" vai aparecer junta ao número 25, isto é, serão concatenadas! Repare também que precisamos tomar cuidado com os parênteses. Vamos misturar strings e números mais uma vez:

```
document.write("A média das nossas idades é: " +  
((25 + 32 + 26) / 3));
```

2.3 REVISANDO O SEU CÓDIGO: TRABALHANDO COM NÚMEROS

Crie um arquivo `testando_idades.html` e vamos revisar o que já aprendemos. Coloque o código que calcula a média das

idades:

```
<script>
document.write("Minha idade é: " + 25);
document.write("A soma das nossas idades é: ");
document.write(25 + 32 + 26);
document.write("A média das nossas idades é: " +
    ((25 + 32 + 26) / 3));
</script>
```

Cada revisão dessas sempre oferecerá exercícios importantes para que você pratique e fixe o que vimos. Não deixe de ir além e realizar seus próprios testes. Sua curiosidade será importante para seu aprendizado.

1) Esse código não está usando o `
`, então toda a saída está na mesma linha! Fica praticamente impossível de acompanhar o programa. Altere-o, adicionando `
` ao final de cada linha. Onde há aspas, basta colocá-lo lá dentro. Onde não há, você precisará concatenar. Por exemplo, nesta linha:

```
document.write("Minha idade é: " + 25);
```

Você precisa adicionar um `+ "
"` :

```
document.write("Minha idade é: " + 25 + "<br>");
```

2) Quantos anos você tem de diferença do seu irmão? Adicione uma nova linha de código, imprimindo a mensagem "Nossa diferença de idade é", concatenando com o resultado da subtração da sua idade com a do seu irmão (ou de um amigo, claro!). A resposta pode dar negativa, sem dúvida.

3) Se, em média, um casal tem filho quando eles têm cerca de 28 anos, quantas gerações se passaram desde o ano zero? Imprima esse número, dividindo o ano atual por essa média.

2.4 ORGANIZE SEUS DADOS EM VARIÁVEIS

Podemos imprimir o ano do nascimento de cada um de nós três, utilizando o recurso de juntar (concatenar) uma sequência de caracteres (uma *string*) com números.

```
document.write("Eu nasci em : " + (2012 - 25) + "<br>");  
document.write("Adriano nasceu em : " + (2012 - 26) + "<br>");  
document.write("Paulo nasceu em : " + (2012 - 32) + "<br>");
```

Além desse `
` que apareceu muitas vezes, tanto aqui quanto na seção anterior, o número 2012 é bastante repetido. O que acontecerá quando precisarmos atualizar esse número para 2013 ? Ou quando descobriremos que Paulo tem, na verdade, uma idade diferente? Precisaríamos substituir todos esses valores, um a um.

Mesmo utilizando um atalho do seu editor para procurar/substituir, essa não é uma opção tão elegante. Além disso, esses números 2012, 25, 26 e 32 aparecem sem um sentido muito claro no seu código: quem lê-los provavelmente terá de se esforçar bastante para compreender o que você desejava expressar por meio deles.

Como facilitar a mudança desses números e também tornar nosso código mais compreensível? O ideal seria poder ter uma forma de dizer 2012, sem precisar repeti-lo. Podemos fazer isso **atribuindo** o valor 2012 a, digamos, ano :

```
var ano = 2012;  
document.write("Eu nasci em : " + (ano - 25) + "<br>");  
document.write("Adriano nasceu em : " + (ano - 26) + "<br>");  
document.write("Paulo nasceu em : " + (ano - 32) + "<br>");
```

O que faz o trecho de código `var ano = 2012` ? Ele **atribui**

2012 a ano . Chamamos ano de **variável**. Uma variável pode guardar praticamente o que você quiser: um número, uma string, um outro pedaço de código. Podemos fazer o mesmo com a soma das idades:

```
var eu = 25;
var adriano = 26;
var paulo = 32;

var total = eu + adriano + paulo;
document.write("A soma das idades é: " + total);
```

var é uma palavra especial no JavaScript. Chamamos esse tipo de palavras de **palavras-chave** de uma linguagem. Ela tem um tratamento diferenciado, nesse caso criando uma variável. Não se preocupe com a sintaxe, com essa forma diferente de escrever. Ficará mais claro no decorrer das lições.

O operador igual (=) não funciona exatamente como na matemática. Por exemplo, 2012 = ano não funciona, não é o mesmo que ano = 2012 . Dizemos que o operador = **atribui** o valor 2012 à variável, que fica a esquerda do = .

Repare que o uso das aspas define o que será impresso. Se não há aspas dentro dos parênteses do document.write(...) , o JavaScript buscará o valor daquela variável. Caso contrário, utilizará o que está dentro das aspas apenas como uma sequência de caracteres (string), como vimos anteriormente. É muito importante você mesmo testar e ver essa diferença:

```
var ano = 2012;
document.write("ano");
document.write(ano);
```

O que acontece?

Você utilizará variáveis o tempo inteiro. Vamos praticá-las!

2.5 REESCREVENDO A MÉDIA DE IDADE DOS SEUS AMIGOS

Com essas mudanças no seu código, você pode calcular a média de idade dos seus amigos de uma forma mais organizada, sem copiar os números para dentro do `document.write` :

```
document.write((eu + adriano + paulo) / 3);
```

Uma outra forma seria quebrar esse processo em passos. É bastante comum criar algumas variáveis a mais para ajudar a legibilidade. Um programador costuma trabalhar em uma equipe, onde outros colegas estarão sempre lendo, modificando e trabalhando com as mesmas linhas de código. A ideia aqui seria criar uma variável para a soma, e outra para a média:

```
var total = eu + adriano + paulo;  
var media = total / 3;  
document.write(media);
```

2.6 REVISANDO O SEU CÓDIGO: ORGANIZE-SE COM VARIÁVEIS

Na seção anterior, escrevemos nosso código de forma mais legível. Caso queira, crie o arquivo `testando_idades_com_variaveis.html` para praticarmos esse código uma última vez, depois passaremos para um outro problema.

Inicialmente, coloque a declaração da idade de cada pessoa:

```
<script>
var eu = 25;
var adriano = 26;
var paulo = 32;
```

Depois, calculamos os dados que precisamos: o total e a média.

```
var total = eu + adriano + paulo;
var media = total / 3;
```

Por último, imprimimos a média e fechamos a tag de `script` :

```
document.write("A média de idade é " + media);
</script>
```

Seu código está funcionando? Cuidado com os nomes das variáveis. Você precisa utilizá-las da mesma forma como as declarou. O JavaScript diferencia, inclusive, maiúsculas de minúsculas. Erre o nome de uma variável para você ver qual é a mensagem de erro que aparecerá no JavaScript Console.

Por exemplo, mostre `med` em vez de `media` , como havia sido declarado:

```
document.write("A média de idade é " + med);
```

É comum digitarmos o nome de uma variável errada. Fique atento e habitue-se a utilizar o JavaScript Console que vimos no fim da lição passada. As mensagens de erro serão em inglês, e algumas vezes não serão tão específicas quanto gostaríamos, mas sempre dizendo com exatidão em que linha do código houve o problema.

Você está cansado de ver um número com tantas casas decimais? Você pode arredondá-lo com o `Math.round(numero)` . Altere seu código:

```
document.write("A média de idade é " + Math.round(media));
```

O `Math.round` pega o valor que está dentro dos parênteses e o arredonda, utilizando esse novo valor para juntar (concatenar) com o restante da frase que queremos mostrar.

Agora, vamos fazer um exercício completamente novo:

1) Crie um novo arquivo, o `calcula_consumo.html` , colocando um título de destaque na primeira linha: `<h3>Álcool ou Gasolina?</h3>` . Logo abaixo, insira as já conhecidas tags de `<script>` e `</script>` . O arquivo está pronto para adicionarmos o código do nosso programa. Salve e abra-o no navegador.

2) Seu carro tem um tanque de 40 litros. Com gasolina, e usando todo o tanque, você fez um `caminhoComGasolina` de 480 quilômetros. Qual é o `consumoDeGasolina` ? Para calculá-lo, divida a distância percorrida pela quantidade de litros gasto. Imprima esse valor, organizando suas contas em variáveis.

É comum utilizar uma variável dessa forma, como `consumoDeGasolina` . A letra `D` e `G` facilitam a leitura. Compare `consumoDeGasolina` com `consumodegasolina` . E fique atento: se você errar o maiúscula/minúscula depois que criar a variável, o código não funcionará como esperado.

3) Já com álcool, o mesmo tanque de 40 litros fez um `caminhoComAlcool` de 300 quilômetros. Qual é o `consumoDeAlcool` ?

4) Os números são todos quebrados, cheios de dígitos. Utilize o `Math.round` para arredondá-los. Funciona bem?

5) Arredondar um número pequeno, como `0.314178473` , vai

dar 0 ! Faz sentido, mas não é o que queríamos. Gostaríamos de arredondar 0.314178473 para ter apenas duas casas decimais, por exemplo. Para fazer isso, você deve utilizar `numero.toFixed(2)`, sendo que o 2 é um parâmetro indicando quantas casas decimais queremos.

Repare que estamos chamando uma função de maneira bem diferente. Antes fazíamos `Math.round(numero)`, e agora fazemos `numero.toFixed(2)`. Além do nome da função ser diferente, dessa vez a variável `numero` aparece "na frente" da chamada. Isso aparece bastante no JavaScript e tem sim uma diferença que entenderemos em capítulos posteriores.

6) Agora um desafio. Hoje, o `precoDaGasolina` está R\$2.90 e o `precoDoAlcool` R\$2.40. Qual é o `precoPorKilometro`, tanto do álcool quanto da gasolina? Dica: dividindo o preço do litro pelo consumo, temos o preço por quilometro. Qual é o menor deles?

2.7 PARE DE ESCREVER BR TANTAS VEZES!

Vimos que utilizar variáveis já ajudou de duas formas: não precisamos mais copiar e colar tanta informação e também alguns trechos ficaram mais legíveis. As variáveis podem ir além, economizando na repetição de linhas de código.

Atenção! Nesta seção, não precisa alterar seu código, apenas acompanhe o que poderia ser feito. Você fará tudo o que há de novo aqui durante a revisão, que virá logo a seguir.

Repare como ficou aquele nosso código que mostra as idades de cada pessoa:

```
var ano = 2012;
```

```
document.write("Eu nasci em : " + (ano - 25) + "<br>");
document.write("Adriano nasceu em : " + (ano - 26) + "<br>");
document.write("Paulo nasceu em : " + (ano - 32) + "<br>");
```

Poderíamos organizá-lo um pouco mais, removendo o `
` para uma outra chamada do `document.write`.

```
var ano = 2012;
document.write("Eu nasci em : " + (ano - 25));
document.write("<br>");
document.write("Adriano nasceu em : " + (ano - 26));
document.write("<br>");
document.write("Paulo nasceu em : " + (ano - 32));
document.write("<br>");
```

Já é um passo. Mas e se quiséssemos, em vez de pular uma única linha entre cada resposta, passar um traço? Ou pular duas linhas? Teríamos de modificar nosso código em **todos** os pontos que há `document.write("
");`. Note que isso é quase o mesmo problema que já vimos antes, quando queríamos que nosso programa calculasse automaticamente a idade dos nossos amigos!

É uma repetição e podemos tratar ela da mesma forma. Em vez de ter todo esse trabalho, podemos **separar esse trecho de código e dar um nome para ele**. É o que chamamos de **função**. A diferença entre uma função e uma variável é que as variáveis guardam apenas valores, enquanto funções guardam linhas de código que serão executadas quando chamarmos tal função.

Vamos criar uma função que executa isso e chamá-la de `pulaLinha`:

```
function pulaLinha() {
    document.write("<br>");
}
```

Opa! Agora apareceram muitas coisas novas. Temos a palavra

function , as chaves ({ e }), além do document.write estar estranhamente um pouco mais para a direita. Não se preocupe em entender todos os detalhes agora.

O function indica que queremos criar um novo procedimento (trecho de código) para não termos mais de copiar e colar código. As chaves indicam o começo e o fim desse procedimento: tudo o que está dentro delas faz parte dessa função.

O document.write está mais à direita por uma questão fundamental de legibilidade. Todo programador escreverá o código dessa forma, para deixar claro que esse trecho está dentro da função que declaramos. Você pode e deve fazer isso com o TAB do seu teclado.

No trecho anterior, nós criamos a função, mas assim como as variáveis, apenas criá-la não é o bastante. Faltou ainda utilizarmos essa nova função pulaLinha ! Diferente de uma variável que guardava um número ou uma string, queremos **chamar** esta função, para que o código que está dentro dela seja executado. Para isso, escreveremos pulaLinha(); , isto é, com parênteses, indicando que é para aquele código, de pular linha, ser executado. Nosso código ficaria então:

```
function pulaLinha() {  
    document.write("<br>");  
}  
  
var ano = 2012;  
document.write("Eu nasci em : " + (ano - 25));  
pulaLinha();  
document.write("Adriano nasceu em : " + (ano - 26));  
pulaLinha();  
document.write("Paulo nasceu em : " + (ano - 32));  
pulaLinha();
```

O código está melhor? Parece até que ficou mais comprido. Vamos ver o que ganhamos com essa abordagem, e onde mais podemos melhorar.

Em vez de pular uma linha, podemos usar um efeito visual mais interessante. Que tal colocar uma linha que cruza o navegador de lado a lado? Para isso, temos a tag `<hr>` no HTML. Como fazer com que nossa `pulaLinha` utilize essa tag em vez de `
` ? Bastaria alterar sua declaração:

```
function pulaLinha() {  
    document.write("<hr>");  
}
```

O que mais precisamos mudar? Nada! É exatamente essa a grande vantagem. Com as funções, conseguimos deixar um código em um único ponto, sem ter de ficar alterando muitos lugares para obter um resultado diferente do anterior.

2.8 REVISE O CÓDIGO: CRIE SUA PRIMEIRA FUNÇÃO

A partir de agora, vamos escrever nosso código. Crie um novo arquivo, que será o `mostra_idades.html` .

Começamos com a tag de `script` , e depois declarando a função `pulaLinha` . A forma de escrever (sintaxe) pode parecer estranha no começo:

```
<script>  
  
function pulaLinha () {  
    document.write("<br>");  
}
```

Pronto. A função `pulaLinha` agora se refere a uma função que pode ser chamada. **Como chamá-la?** Da mesma forma que você já fez com `alert`, por exemplo. Isto é, **usando os parênteses** após o seu nome:

```
var ano = 2012;
document.write("Eu nasci em : " + (ano - 25));
pulaLinha();
document.write("Adriano nasceu em : " + (ano - 26));
pulaLinha();
document.write("Paulo nasceu em : " + (ano - 32));
pulaLinha();
```

Toda vez que aparece `pulaLinha()`; o navegador vai executar o código da função `pulaLinha`. Dessa forma começamos a evitar código duplicado. Isso é apenas o início, vamos usar mais das funções para facilitar o nosso código.

2.9 FUNÇÕES PASSANDO INFORMAÇÕES E CHAMANDO OUTRAS FUNÇÕES

Vamos novamente nos concentrar no texto. Na revisão, faremos o código que está sendo descrito aqui.

Já é possível enxergar bem onde ganhamos: podemos mudar o comportamento do nosso programa alterando apenas um único lugar: a função que criamos para a `pulaLinha`. Porém nosso código continua um pouco grande, e toda hora temos de chamar essa nossa função.

Quando percebemos que estamos sendo muito repetitivos, sempre podemos considerar a criação de uma nova função. Nesse caso, está fácil enxergar que toda vez que jogamos uma frase para o navegador com `document.write`, logo em seguida

pulamos uma linha. Podemos unificar isso em um único lugar? Isso é possível criando mais uma função, uma que mostra uma frase e põe também o `
`. Por exemplo:

```
function mostra() {  
    document.write("alguma frase<br>");  
}
```

Mas essa função não é tão útil: ela sempre mostra a mesma frase. Não serve para o que nós queremos, pois a frase que desejamos mostrar depende do momento. Não se desespere, há sim como resolver esse problema.

Quando **declaramos** (criamos) uma função, podemos fazer de tal forma para que recebamos algo a mais, alguma informação que seja importante para nós. No caso da função que mostra alguma frase, o que seria esse valor importante? A frase que queremos mostrar! Fazemos isso declarando dentro dos parênteses, como `function(frase)`, e depois utilizamos `frase` normalmente como as variáveis que já conhecemos:

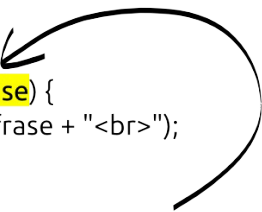
```
function mostra(frase) {  
    document.write(frase + "<br>");  
}
```

Pronto. Mas como o `mostra` saberá que `frase` deve ser colocada no navegador? Isso será feito durante a chamada da função. Diferente do `pulaLinha`, que chamamos usando os parênteses sem nada dentro, o `mostra` será invocado com a `frase` dentro deles:

```
mostra("Usando funções para melhorar o código");
```

Essa string que queremos mostrar vai ser **passada** para a função que criamos, e lá dentro será a nossa variável `frase`.

Variáveis que são passadas para funções são frequentemente chamadas de **parâmetros** ou **argumentos**.



```
function mostra(frase) {  
    document.write(frase + "<br>");  
}  
  
mostra("Usando funções para melhorar o código");
```

The diagram illustrates the flow of data from an argument to a parameter. A curved arrow originates from the string argument `"Usando funções para melhorar o código"` in the function call `mostra(...)` and points to the parameter `frase` in the function definition `function mostra(frase) {`. Both the parameter and the argument are highlighted in yellow.

Figura 2.1: O texto que passarmos entre os parênteses vai parar na variável frase função

Podemos mudar um pouco a nossa função `mostra` para que ela também se aproveite da nossa velha `pulaLinha`, em vez de concatenar o `
` por si só:

```
function mostra(frase) {  
    document.write(frase);  
    pulaLinha();  
}
```

É isso mesmo: uma função pode chamar outra função. É algo que ocorre com muita frequência. Qual é a vantagem aqui? Agora, o nosso `mostra` também acompanhará as mudanças do `pulaLinha`. Se quisermos pular linha de uma forma mais visual, seja com `
`, `<hr>` ou outro recurso do HTML, a função `mostra` vai se beneficiar disso, sem nem mesmo precisar ser modificada!

2.10 REVISE O CÓDIGO: USANDO A FUNÇÃO MOSTRA

Fizemos bastante durante essa lição. Vamos revisar como está

seu arquivo agora. Organize-o, criando o arquivo `mostra_idades2.html` . Logo no começo, temos a definição das nossas funções. Começamos pela `pulaLinha` :

```
<script>

function pulaLinha() {
    document.write("<br>");
}
```

Logo a seguir, vamos ter nossa segunda função, a `mostra` que, por sua vez, faz uso da `pulaLinha` . Diferente da anterior, ela recebe um **parâmetro**, que será a frase a ser apresentada no navegador:

```
function mostra(frase) {
    document.write(frase);
    pulaLinha();
}
```

Lembre-se de colocar o código dentro de uma função sempre mais à direita, usando o `TAB` do seu teclado. Esse é o processo de **indentar** o código (neologismo do inglês *to indent*). É importante que sua **indentação** esteja correta para facilitar a leitura do programa.

Após as duas funções declaradas, vamos utilizá-las no nosso código para imprimir quantos anos tem cada um dos envolvidos:

```
var ano = 2012;
mostra("Eu nasci em : " + (ano - 25));
mostra("Adriano nasceu em : " + (ano - 26));
mostra("Paulo nasceu em : " + (ano - 32));
</script>
```

Vamos a alguns exercícios, começando por uns baseados nesse código.

1) Altere sua função `pulaLinha` para que ela pule duas linhas! Isto é, faça dois `
` s.

2) Como vimos, há uma tag HTML que pode ser ainda mais interessante para separar um resultado do outro, é o `hr` . Altere a função `pulaLinha` para que ela escreva no navegador um `<hr>` entre os dois `
` s que você já fez.

3) A fonte do nosso programa talvez ainda não seja adequada. Há uma tag HTML que se chama `big` . Faça com que a função mostra coloque a frase entre `<big>` e `</big>` .

4) O que acontece se você esquecer a palavra `function` na hora de declarar uma de suas funções? E os parênteses na declaração da função `pulaLinha` ? Faça os testes e veja as mensagens de erro no console JavaScript do Chrome.

2.11 MOSTRANDO MENSAGENS SECRETAS, APENAS PARA O PROGRAMADOR

O `alert` joga uma mensagem dentro de um pop-up. Usá-lo extensivamente pode acabar com a paciência do usuário, que precisará clicar em `OK` a cada nova mensagem. O `document.write` é menos intrusivo, mas você já reparou que as mensagens são jogadas diretamente na página, sem nem mesmo um espaçamento entre as linhas. Isso porque o próprio documento HTML é alterado. Se você quiser pular uma linha via o `document.write` , precisará utilizar tags HTML, como o `
` , fazendo `document.write("olá mundo!
");` , por exemplo.

Mesmo colocando o `document.write` em uma função, muitas vezes queremos que algumas mensagens não apareçam

para o usuário, porém gostaríamos de poder vê-las durante o desenvolvimento de nosso programa. Isto é, uma mensagem que, de alguma forma, fosse visível apenas para você, programador. Isso é muito útil para descobrir erros (o que chamamos de **bug**), aprender novos truques e testar recursos. Guardamos esses dados em **logs**. É comum usar o neologismo **logar**, assim mesmo, em português. Para logar informações com JavaScript, há a função `console.log`. Faça um teste:

```
<script>
document.write("olá mundo!");
document.write("esse é meu segundo programa");
console.log("esta mensagem aparece apenas no log");
</script>
```

Qual é o resultado?

A mensagem passada ao `console.log` não apareceu! Quando utilizamos essa função, o navegador guarda todas as mensagens em um local especial, longe da vista do usuário comum. Para ver o resultado, precisamos habilitar a visualização do console, exatamente como fizemos na lição anterior para verificar erros.

No Chrome, você faz isso clicando no pequeno ícone de ferramentas/menus, escolhe a opção *Ferramentas (Tools*, se estiver em inglês), e depois *Console JavaScript*. É o mesmo console que você usou para ver as mensagens de erro do seu código:

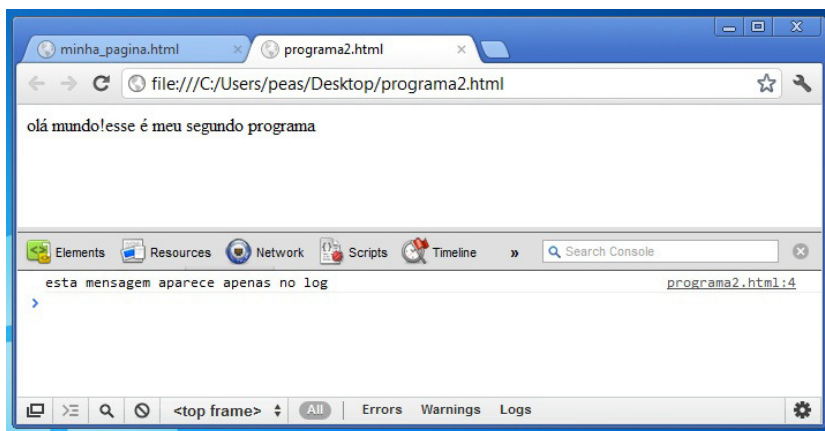


Figura 2.2: Console de JavaScript do Chrome com o log

Há a tecla de atalho `CTRL+SHIFT+J` no Windows, e no Linux para abrir essa aba (`Command+Option+J` no Mac) e depois clicar no Console, caso outra opção esteja selecionada. Ele realmente é importante e você estará visitando-o com frequência.

Você pode combinar as três funções da melhor forma que encontrar: `alert` para destacar uma mensagem, `document.write` para adicionar informações dentro da própria página e o `console.log` para mostrar dados apenas a nós, programadores.

Também veremos no decorrer do nosso aprendizado outras formas e técnicas que nos auxiliam a descobrir problemas no nosso código. Maneiras de remover os **bugs**, isto é, como **debugar** o nosso código.

2.12 PARA SABER MAIS: COMENTÁRIOS

É também comum querermos colocar uma frase dentro do código que sirva apenas como referência para os programadores. Em JavaScript, podemos fazer isso usando `//`. Tudo o que vier após o `//` vai ser ignorado pelo navegador. Repare:

```
// esta função mostra uma frase no navegador e pula uma linha
function mostra(frase) {
    document.write(frase);
    pulaLinha();
}

// agora vamos colocar no navegador o ano em que nasci:
var ano = 2012;
mostra("Eu nasci em : " + (ano - 25));
```

Os comentários podem ajudá-lo a organizar melhor o código. De qualquer maneira, é sempre mais importante ter um código bem escrito, com nomes de variáveis expressivas que façam bastante sentido, do que ter de usar muitas linhas de comentários.

Há também a opção de colocar comentários entre `/*` e `*/`. Dessa forma, tudo o que estiver entre esses dois identificadores será ignorado pelo navegador, inclusive se houver quebras de linha.

2.13 COMPARTILHE SEU CÓDIGO COM SEUS AMIGOS!

Até agora você rodou seus programas no seu próprio navegador. E se quisesse que um amigo pudesse ver o que está fazendo? Caso você já conheça um pouco mais de internet, poderia colocar seus arquivos `.html` em um servidor web. Mas há uma forma bem fácil de compartilhar seus exercícios e mostrar suas recém-adquiridas habilidades de programador.

Alguns sites permitem que você escreva código HTML e JavaScript dentro de formulários e veja rapidamente o resultado. Qual é a vantagem de escrever dentro de um site em vez de no nosso próprio computador? É que esses sites permitem compartilhamento dos programas! Acesse nosso programa que calcula idade aqui:

<http://jsfiddle.net/55vSR/>

Esse site vai apresentar 4 diferentes espaços: HTML, CSS, JavaScript e o resultado. O JavaScript é o que nos interessa. Como essa é uma página especial, você não precisa (nem deve) usar a tag `script` dentro desse formulário. Você pode clicar em *Run* (no menu superior), ou pressionar `CTRL+ENTER` para rodar o código.

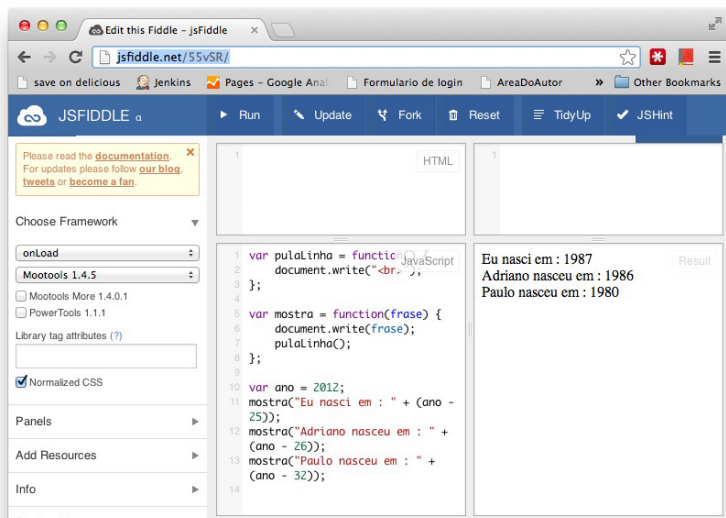


Figura 2.3: Código compartilhado pelo site JSFiddle.net

Você pode editar esse mesmo código e clicar em *Save*. O site JSFiddle vai gerar um novo endereço que você pode compartilhar com seus amigos e familiares. É uma forma interessante de mostrar sua evolução na programação para todos os conhecidos!

Por uma questão de como os navegadores carregam páginas HTML, o jsfiddle funcionará melhor se você usar `console.log` em vez de `document.write`. Ele vai até avisar para não usar o `document.write`.

Também é possível criar uma conta para que você tenha todos os seus códigos organizados.

Compartilhe seus códigos e desafios na nossa lista de discussão! Não deixe de participar, seja tirando dúvida ou mostrando suas conquistas: <http://forum.casadocodigo.com.br>.

Há outras alternativas, talvez um pouco mais complexas e completas, como o <http://tinkerbin.com/>, o <http://playground.html5rocks.com> e o <http://codepen.io/>.

PRATIQUE RESOLVENDO PROBLEMAS DO SEU DIA A DIA

No capítulo anterior, fomos capazes de criar nossas próprias variáveis, e até variáveis especiais, que continham funções. Mas já havíamos trabalhado com outras funções, antes mesmo de conhecer o que eram. A `document.write` e a `alert` são duas das muitas funções que todo navegador vai disponibilizar para nós.

Vamos praticar mais a utilização de variáveis e funções, além de conhecer outras que podemos usar nos navegadores.

3.1 COMO ESTÁ SEU PESO? SAUDÁVEL?

Índice de Massa Corporal, o IMC, é um indicador do grau de obesidade de uma pessoa. Pode ser uma primeira forma de ter um sinal de que alguém está muito gordo ou muito magro. A fórmula do IMC é simples: peso dividido pelo quadrado da altura.

Será que conseguimos traduzir isso para JavaScript? Abra o editor de texto e crie um novo arquivo, que gravaremos como `imc.html` . Para calcular o quadrado da altura, basta

multiplicarmos esse valor por ele mesmo, e depois utilizarmos para dividir o peso.

```
var imc = peso / (altura * altura);
```

Mas onde estão definidas as variáveis peso e altura? Ainda não estão. É comum você ver códigos de exemplo escritos dessa forma, isto é, sem estar por completo. Se você tentar rodar um código assim, o JavaScript reclamará de que `altura` e `peso` não existem. Experimente.

Vamos então dar valores e imprimir o resultado do `imc` :

```
var peso = 75;
var altura = 1.72;
var imc = peso / (altura * altura);
mostra("meu IMC é: " + imc);
```

Estamos usando a nossa função `mostra` . Para isso, ela precisa estar declarada no começo do arquivo. Lembre-se também de que precisamos das tags `<script>` e `</script>` envolvendo nosso código JavaScript; caso contrário, ele seria mostrado no navegador, sem outros efeitos. Fique tranquilo! Você rapidamente se ambientará a esse processo. E logo a seguir haverá uma seção de revisão, na qual você poderá fazer esse código novamente, passo a passo.

Salve o arquivo `imc.html` e abra-o no navegador.

Usamos nosso primeiro número que possui casas decimais, o `1.72` . No JavaScript, sempre usaremos o ponto como delimitador da parte inteira, e não a vírgula.

3.2 UTILIZE UMA FUNÇÃO PARA CALCULAR

O IMC DE CADA AMIGO

Mas e se agora precisarmos calcular o IMC de seu primo? Podemos criar uma outra variável, `imcDoPrimo`, para guardar esse resultado. Para isso, precisaríamos também ter o `pesoDoPrimo` e `alturaDoPrimo`. Nosso código ficaria assim:

```
var pesoDoPrimo = 83;
var alturaDoPrimo = 1.76;
var imcDoPrimo = pesoDoPrimo / (alturaDoPrimo * alturaDoPrimo);
mostra("o IMC de meu primo é: " + imcDoPrimo);
```

Algo em comum com o código para calcular nosso próprio IMC? Sim! A fórmula continua sendo a mesma, e a chamada à função `mostra` também é bastante similar. Quando temos código sendo repetido, devemos sempre pensar na possibilidade de criar uma função. Vamos criar uma que seja responsável por calcular o IMC.

Assim como a nossa função `mostra`, essa precisará receber parâmetros para poder calcular seu resultado. Que parâmetros precisamos para calcular o IMC? Serão dois: `peso` e `altura`, que também declararemos dentro dos parênteses, usando a vírgula como separadora:

```
function calculaIMC(altura, peso) {
    var imc = peso / (altura * altura);
    mostra("IMC calculado é " + imc);
}
```

Repare que declaramos uma variável `imc` dentro da nossa função. Dessa forma, essa variável só existe aí dentro, não podendo ser acessada do lado de fora das chaves. Parece estranho, mas isto é útil para organizar o código. Afinal, quem vai usar essa nossa função não precisa, e nem quer, ficar sabendo como esse cálculo é

feito.

Com a função declarada, já podemos usá-la:

```
var pesoDoPrimo = 83;
var alturaDoPrimo = 1.76;
calculaIMC(alturaDoPrimo, pesoDoPrimo);
```

Salve seu arquivo `imc.html` e veja o resultado!

Um resultado abaixo de 18.5 indica que a pessoa está muito magra, já acima de 35, a obesidade é severa. Agora queremos mostrar o quão distante o índice está de 18. Poderíamos fazer isto dentro da função `calculaIMC` que acabamos de criar, mas seria isso tarefa para uma função que calcula o IMC?

Em vez disso, podemos fazer com que a `calculaIMC` não mostre o IMC e, como o nome diz, apenas calcule seu valor, **retornando-o** para quem a chamou. Altere sua função, removendo a linha do `mostra` e adicionando um `return`:

```
function calculaIMC(altura, peso) {
    var imc = peso / (altura * altura);
    return imc;
}
```

Agora, na linha que faz a chamada a essa função, vamos pegar o que ela retorna, para depois utilizar. Veja:

```
var pesoDoPrimo = 83;
var alturaDoPrimo = 1.76;
var imcDoPrimo = calculaIMC(alturaDoPrimo, pesoDoPrimo);
mostra("O imc do meu primo é " + imcDoPrimo);
mostra("Ele ainda está " + (imcDoPrimo - 18.5)
    + " pontos acima do limite da magreza.");
```

A variável `imcDoPrimo` vai receber o valor que foi **retornado** pela função `calculaIMC` por meio da palavra-chave `return`.

Teremos muitas funções que trabalham dessa forma, recebendo alguns parâmetros, realizando uma tarefa e depois devolvendo um resultado que será útil para quem a chamou. Esse valor que volta para quem **chamou** a função é conhecido como **retorno da função**.

3.3 REVISE O CÓDIGO: CALCULANDO O IMC

Vamos revisar como deve estar seu código do `imc.html`. Primeiramente, precisamos declarar as funções que vamos utilizar. Temos tanto a função `mostra` quanto a `pulaLinha`:

```
<script>
function pulaLinha() {
    document.write("<br>");
}

function mostra(frase) {
    document.write(frase);
    pulaLinha();
}
```

Agora teremos nossa função que, dados altura e peso, calcula o IMC e o retorna:

```
function calculaIMC(altura, peso) {
    var imc = peso / (altura * altura);
    return imc;
}
```

Pronto, já podemos utilizar essa função, pegando o seu retorno, seja para você ou para o seu primo:

```
var peso = 75;
var altura = 1.72;
var imc = calculaIMC(altura, peso);

mostra("Meu imc é " + imc);
```

```
mostra("Ainda estou " + (imc - 18.5)
      + " pontos acima do limite da magreza.");
</script>
```

Vamos modificar um pouco nosso código, praticando com exercícios.

1) Adicione uma linha de código para dizer o quão distante você está da obesidade severa, que é o índice de 35.

2) Há uma forma mais curta de escrever essas duas linhas:

```
var imc = calculaIMC(altura, peso);
mostra("Meu imc é " + imc);
```

Como estamos atribuindo o resultado do cálculo à variável `imc`, e depois usando-a na chamada ao `mostra`, poderíamos ter feito isso tudo dentro de uma única linha, sem declarar a variável:

```
mostra("Meu imc é " + calculaIMC(altura, peso));
```

Faça o teste. Mas será que é vantajoso? A linha que mostrava o quão distante você estava da magreza ficava logo embaixo. O que acontecerá com ela agora que não temos mais a variável `imc`? Não vai funcionar. Você poderia fazer o mesmo nas outras linhas, colocando `calculaIMC(altura, peso)` em vez da variável, mas o navegador vai, a cada chamada, recalculando o IMC, sendo que da forma anterior esse número foi calculado apenas uma única vez e reaproveitado.

3) Você se lembra do `Math.round`? Ele também é uma função que todos os navegadores já possuem definida. E ela é uma função que retorna um valor, por isso podemos fazer:

```
mostra("Meu imc é " + Math.round(imc));
```

Essa função retorna o número arredondado. Uma outra opção

seria utilizá-la no momento da declaração da variável `imc` . Isto é, em vez de:

```
var imc = calculaIMC(altura, peso);
```

Você pode fazer:

```
var imc = Math.round(calculaIMC(altura, peso));
```

Difícil? É uma forma comum de escrever código. Repare nos parênteses: primeiro será calculado o IMC, para depois, o `Math.round` arredondar esse valor.

Qual dos dois você vai usar? Depende. Se você quer sempre trabalhar com o `imc` arredondado, vale utilizar essa última forma. Se algumas vezes usará arredondado e outras da forma quebrada, é melhor utilizar o `Math.round` só quando convier.

3.4 TRABALHE COM DADOS CAPTURADOS: PERGUNTE A ALTURA E PESO DO USUÁRIO

Somos capazes de trabalhar com números e palavras, mas todos eles estão dentro do nosso código, mesmo que organizados em variáveis e funções. Se quisermos calcular o IMC de um outro amigo, precisaremos mudar esse valores, salvar o arquivo e atualizar a página. Seria bastante interessante poder ter informações sobre o usuário do nosso programa, sem ter de alterar o código.

Quando usamos o computador, é natural aparecer uma caixa de diálogo (*dialog box*) perguntando alguns dados do usuário. Podemos fazer o mesmo via JavaScript. Assim como já criamos uma caixa de alerta para exibir nossos resultados com o `alert` ,

podemos usá-la para receber dados. Isso é feito com a função `prompt` .

Crie um novo arquivo que será salvo como `perguntas.html` e coloque:

```
<script>
var nome = prompt("Bom Dia! Qual é o seu nome?");
</script>
```

Simples, não? A função `prompt` retorna exatamente o que for digitado pelo usuário nessa caixa. Com o nome do usuário em mãos, podemos utilizá-lo para conversar com ele:

```
<script>
var nome = prompt("Bom Dia! Qual é o seu nome?");
document.write("Bem vindo, " + nome);
</script>
```

Você também pode perguntar a `idade` do usuário, e utilizar seu próprio `nome` nessa nova pergunta:

```
<script>
var nome = prompt("Bom Dia! Qual é o seu nome?");
document.write("Bem vindo, " + nome);

var idade = prompt(nome + ", quantos anos você tem?");
document.write(nome + " tem " + idade + "anos.");
</script>
```

3.5 EXERCÍCIOS: PERGUNTE OS DADOS DO USUÁRIO PARA CALCULAR O IMC

Em vez de ter de ficar alterando toda vez o nosso arquivo `imc.html` para calcular o IMC de uma outra pessoa, vamos utilizar essa função `prompt` para capturar a altura e o peso do usuário.

1) No seu arquivo `imc.html` , depois da declaração das três funções (`pulaLinha` , `mostra` e `calculaIMC`), pergunte a altura e o peso do usuário:

```
var alturaDoUsuario = prompt("Bom Dia! Qual é a sua altura?")
;
var pesoDoUsuario = prompt("E o seu peso?");
```

2) Agora, com esses dois dados, fica fácil calcular o IMC do usuário:

```
var imcDoUsuario = calculaIMC(alturaDoUsuario, pesoDoUsuario)
;
mostra("O seu imc é " + imcDoUsuario);
</script>
```

3) Altere seu código para perguntar o nome do usuário, usando um `prompt` , e usar esse nome capturado para perguntar tanto a altura quanto peso .

4) O que acontece se você digitar algo que não é um número quando te perguntarem o peso e a altura? Ou utilizar a vírgula em vez de ponto? Veremos como contornar esses problemas em outras lições.

3.6 DESCUBRA QUANTOS DIAS SEUS AMIGOS JÁ VIVERAM

O `prompt` pode capturar quantos anos tem o usuário, e depois podemos utilizar isso para saber quantos dias ele já viveu. Crie o arquivo `dados_vitais.html` e faça a conta de quantos dias o usuário já viveu:

```
<script>
var idade = prompt("Quantos anos você tem?");
var dias = idade * 365;
```

```
document.write("Você já viveu " + dias + " dias de vida");  
</script>
```

Adicione também, logo abaixo desse `document.write`, o cálculo de batimentos cardíacos dessa pessoa. Faremos isso multiplicando o número de dias de vida por 24 horas, e para cada hora vezes 60 minutos, e para cada minuto consideraremos 80 batimentos cardíacos:

```
var batimentos = dias * 24 * 60 * 80;  
document.write("Seu coração já bateu " + batimentos  
    + " vezes. Haja coração!");
```

Essa seção está muito fácil! Vamos fazer alguns exercícios para ir além dessas contas simples.

1) Em vez de deixar o código `var dias = idade * 365` dessa forma, crie uma função que faça esse cálculo. Você vai precisar declarar, lá em cima, uma `var calculaDiasDeVida` que vai ser atribuída a uma `function` que recebe `idade` como parâmetro. Com esse dado, você deve retornar o número de `idade` vezes 365.

Com isso, seu código passará de ``var dias = idade * 365;`` para ``var dias = calculaDiasDeVida(idade);``.

2) Faça o mesmo com a variável de batimentos. Crie uma função `calculaBatimentos` que recebe quantos dias a pessoa viveu e faz o cálculo.

3) Agora que temos as duas funções, podemos fazer assim:

```
var idade = 34;  
var dias = calculaDiasDeVida(idade);  
var batimentos = calculaBatimentos(dias);  
document.write("Seu coração já bateu " + batimentos  
    + " vezes. Haja coração!");
```

Um pequeno desafio: como fazer, na mesma linha, uma chamada para `calculaDiasDeVida` pegar esse resultado e passar para a chamada de `calculaBatimentos` ?

3.7 VOCÊ JÁ ENTENDEU A ORDEM DAS CHAMADAS DAS FUNÇÕES?]

Às vezes, pode ser complicado enxergar qual trecho do código está chamando que outro trecho.

Para facilitar, vamos escrever um simples código com três funções: a primeira função chama a segunda função, e a segunda chama a terceira. Além disso, dentro de cada função, imprimimos uma mensagem antes e depois de chamar a próxima. Dessa forma, será possível que você veja como as chamadas são "empilhadas" e, quando a terceira função é chamada, o caminho inverso começa a ser percorrido.

Crie, então, um arquivo `funcoes.html` e coloque:

```
<script>
// funcoes
function primeira() {
    console.log("1 - antes");
    segunda();
    console.log("1 - depois");
}

function segunda() {
    console.log("2 - antes");
    terceira();
    console.log("2 - depois");
}

function terceira() {
    console.log("3");
}
```

```
// codigo principal
primeira();

</script>
```

Rode e veja o resultado!

3.8 UTILIZE O CONSOLE DO CHROME PARA FAZER TESTES!

Às vezes, pode ser trabalhoso digitar um programa inteiro para descobrir que determinado comando não funcionava da maneira que esperávamos. O Chrome possibilita fazer testes dentro do próprio Console.

Faça o seguinte teste. Abra uma nova aba no Chrome, e então abra o Console, conforme vimos na seção *Mostrando mensagens secretas, apenas para o programador* do capítulo anterior. Você pode fazer isso pelo menu ou já se habituar com a tecla de atalho: CTRL+SHIFT+J no Windows e no Linux (Command+Option+J no Mac), e depois clicar no Console.

Repare que, dentro do console, você pode digitar o que quiser, que o JavaScript será executado! Digite, por exemplo, `console.log("teste");` . O que acontece? Ele primeiro imprime o que a função retorna (nesse caso `undefined` , indicando que não retorna nada). Logo depois, vem a saída da nossa função, conforme esperado!

Faça mais alguns testes. Declare uma variável `idade` valendo 30. Na linha seguinte, digite apenas `idade + 5;` . Não há muito sentido em fazer uma conta em JavaScript sem atribuir o resultado

a uma variável, pois o resultado seria perdido, ficaria sem uso. Dentro do console você pode fazer isso, pois ele exibirá o resultado para você, conforme a figura:

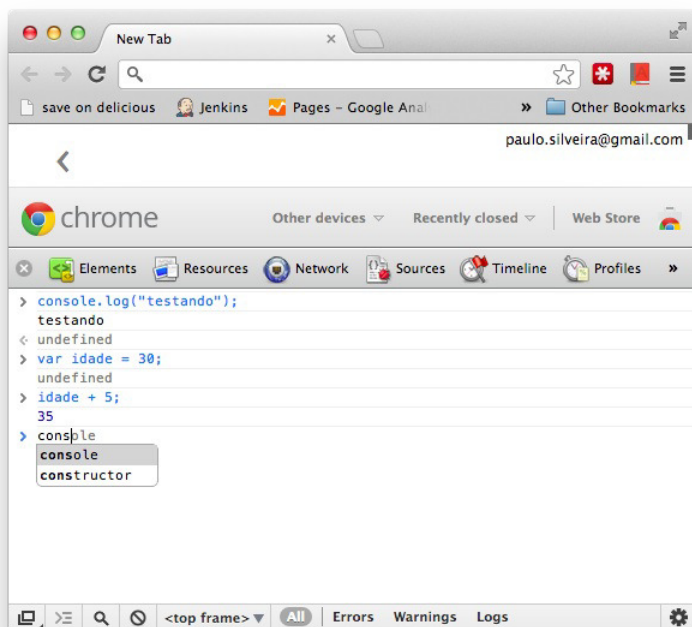


Figura 3.1: Fazendo testes no Console

Outra ajuda que o console dá se chama *code completion*, ou *auto complete*. É a capacidade dele de tentar adivinhar o que você quer escrever, tornando a codificação mais rápida. Repare que, na figura, já começamos a digitar `console` e paramos no meio, como indicado pelo cursor, e o console já deu duas sugestões. Basta escolher a correta e pressionar `enter` ! Essa também é uma

forma de conhecer mais as funções e variáveis que o JavaScript nos fornece.

Cuidado! Se você esquecer de fechar chaves ou parênteses, pode ser bem difícil perceber. O efeito final será inesperado e facilmente pode se confundir entre as diversas mensagens de erro. Caso o código seja grande, prefira criar seu próprio arquivo `html`, ou então o <http://www.jsfiddle.net>, conforme visto na seção *Compartilhe seu código com seus amigos!* do capítulo anterior.

PARA SABER MAIS: OUTRA FORMA DE DECLARAR FUNÇÕES

Em javascript, há quem prefira declarar as funções dessa outra forma...

```
var pulaLinha = function() {  
    document.write("<br>");  
};
```

... do que a maneira que vimos no livro, que diferencia bastante as funções das variáveis:

```
function pulaLinha() {  
    document.write("<br>");  
}
```

Na maioria das vezes, não faz diferença usar de uma forma ou de outra. A forma que atribui a função a uma variável é usada mais comumente quando estamos usando *programação funcional*. Não se preocupe com isso neste momento.

EXECUTE CÓDIGOS DIFERENTES DEPENDENDO DA CONDIÇÃO

Nem sempre queremos executar um código. Como fazemos para decidir se uma linha deve ser ou não executada? Como expressar uma condição? Vamos ver novos comandos e nos aprofundar na lógica de programação.

4.1 QUANTOS PONTOS TEM SEU TIME DE FUTEBOL?

O Casa Do Código Futebol Clube completou seu 15º jogo no campeonato brasileiro. Como estamos indo? Depende do número vitórias, empates e derrotas. Por exemplo, se foram 7 vitórias, 4 empates e 4 derrotas, podemos calcular a situação atual do time como $(7 * 3) + 4$, totalizando 25 pontos.

Isto é, a conta é realizada por meio da expressão `(vitorias * 3) + empates`, já que cada vitória vale 3 pontos e o empate vale 1 ponto. Podemos capturar a qualquer momento quantos jogos

foram vencidos e quantos empates houve, com uso da função `prompt` .

Vamos criar um novo arquivo, o `pontos_futebol.html` , com as nossas já conhecidas funções `mostra` e `pulaLinha` e depois temos:

```
var vitorias = prompt("Quantos jogos o CDC FC venceu?");
var empates = prompt("Quantos jogos o CDC FC empatou?");
var pontos = (vitorias * 3) + empates;

mostra("Nosso time tem " + pontos + " pontos!");
```

Abra o arquivo `pontos_futebol.html` e faça um teste, colocando 3 vitórias e 1 empate. Qual é o resultado? Deveriam ser 10 pontos, mas o resultado sai 91!? Por que?

O JavaScript multiplicou $3 * 3$, resultando `9` . Mas ao somar com o número de empates, em vez dele fazer isso matematicamente, ele juntou os dois valores, concatenando, como acontece com as *strings*.

Por que a concatenação e não a soma?

A explicação é bem simples. Quando usamos o `prompt` , o JavaScript considera o que digitamos na caixa como sendo um texto, ou seja, uma *string*.

Lembre-se de que aprendemos que, quando somamos em um texto, seja o que for, o conteúdo acabava sendo concatenado? Por isso, se fizemos `var teste = 9 + "1";` e mostrarmos a variável `teste` , veríamos que o resultado seria `91` .

Por que não aconteceu isso na conta com `vitorias` ? Pois como era a multiplicação, o JavaScript tentou converter para

número, sem precisar que nós falássemos nada.

Também tivemos essa "sorte" na nossa função que calcula o IMC, porque não fizemos somas com as variáveis que pegamos ao fazer o `prompt`. Vamos precisar resolver isso agora, pois esse tipo de problema, que pode aparecer ao misturar números e *strings*, é bem frequente.

Como resolvo? Não queria concatenar com empates!

Já percebemos que o problema acontece por tentarmos somar um número com um texto (uma *string*). Se queremos fazer uma soma, precisamos que as duas variáveis contenham valores que sejam números. Queremos que eles sejam do **tipo** número.

Você não é o primeiro a passar por esse problema. Justamente para isso, o JavaScript possui uma função própria, chamada `parseInt()`. Não se assuste com esse nome estranho, apenas lembre-se de que o seu objetivo é "transformar" texto em número, para que possamos fazer as operações matemáticas corretamente.

Vamos fazer um primeiro teste para enxergar bem esse problema. Experimente criar uma *string* que contenha apenas números. Somando-a com um número, ocorrerá concatenação:

```
var vitorias = 3;
var empates = "1"; // string!
var pontos = vitorias * 3 + empates;
alert(total);
```

Mas se invocarmos a função `parseInt`, o retorno dela será um número, permitindo a correta manipulação matemática. Repare:

```
var vitorias = 3;
```

```
var empates = "1"; // string!
var empatesComoNumero = parseInt(empates);
var pontos = vitorias * 3 + empatesComoNumero;
alert(total);
```

Agora esse código funciona conforme o esperado, mas criamos uma variável a mais e o código pode começar a ficar poluído. Uma forma comum de resolver isso seria utilizando a mesma variável:

```
var vitorias = 3;
var empates = "1"; // string!
empates = parseInt(empates);
var pontos = vitorias * 3 + empates;
alert(total);
```

Uma outra forma seria chamar o `parseInt` na mesma linha que recebemos o valor dos pontos. É a forma que você encontrará com mais frequência:

```
var vitorias = 3;
var empates = parseInt("1"); // já converte string para int
var pontos = vitorias * 3 + empates;
alert(total);
```

Então, já podemos modificar nosso código que calcula os pontos em `pontos_futebol.html` para, na hora que receber o que o usuário digitar, já converter a String digitada para um inteiro:

```
var vitorias = parseInt(
    prompt("Quantos jogos o CDC FC venceu?"));
var empates = parseInt(
    prompt("Quantos jogos o CDC FC empatou?"));
var pontos = (vitorias * 3) + empates;
```

4.2 VERIFIQUE A SITUAÇÃO DO SEU TIME DE FUTEBOL

Finalmente vamos ao novo tópico: verificar condições e tomar atitudes diferentes de acordo com cada possível resultado.

O nosso grande rival, Livros Velhos Futebol Clube, é o favorito e está indo muito bem. Ele tem 28 pontos. Será que ele está melhor ou pior que o CDC Futebol Clube?

Podemos verificar isso fazendo uma pergunta para o JavaScript, algo como: *"se pontos é maior que 28 , mostre que estamos bem"*. Fazemos isso da seguinte forma:

```
if(pontos > 28) {  
    mostra("Nosso time está indo MELHOR que o Livros Velhos!");  
}
```

O `if` é uma palavra chave do JavaScript. Ele faz a comparação que está dentro dos parênteses e, no caso de ela ser verdadeira, executa o que está dentro das chaves.

Faça o teste. Rode seu código, primeiro colocando 7 vitórias e 4 empates, totalizando 25 pontos. O que acontece com o nosso `if` ? E se você colocar que houve 10 vitórias e 1 empate?

Além do operador `>` , temos também o `<` . Podemos ver se nosso time está pior que o Livros Velhos:

```
if(pontos < 28) {  
    mostra("Nosso time está indo PIOR que o Livros Velhos!");  
}
```

Falta só uma comparação: a de igualdade. Como saber se estamos empatados com o Livros Velhos? Tudo indica que o `=` deveria resolver o problema. Infelizmente, o operador `=` não vai funcionar como esperamos, pois como vimos até aqui, ele serve para **atribuir** valores. Ele pega o valor que está à direita do sinal de

igual e copia para a variável da esquerda.

A solução é utilizar o operador que faz a comparação de igualdade, que é o `==`. São dois sinais de igual, um seguido do outro!

```
if(pontos == 28) {  
    mostra("Nosso time está EMPATADO com o Livros Velhos!");  
}
```

Pode parecer estranho, mas é usado em muitas linguagens. Faz parte da **sintaxe** do JavaScript. É a forma como a linguagem trabalha e cada uma tem suas peculiaridades e detalhes. Há também um outro detalhe da sintaxe que pode ter chamado a sua atenção: no caso de um comando como o `if`, no final do seu fecha-chaves (`}`), não utilizamos o ponto e vírgula!

4.3 REVISANDO NOSSO CÓDIGO: PONTOS DO CAMPEONATO

Nesse momento, temos um arquivo chamado `pontos_futebol.html`, que descobre a situação do nosso time no campeonato de futebol. Vamos rever o código. Ele começa declarando as duas funções que utilizamos bastante:

```
<script>  
function pulaLinha() {  
    document.write("<br>");  
}  
  
function mostra(frase) {  
    document.write(frase);  
    pulaLinha();  
}
```

Perguntamos para o usuário a quantidade de vitórias e empates

do time por meio da função `prompt` :

```
var vitorias = prompt("Quantos jogos o CDC FC venceu?");
var empates = prompt("Quantos jogos o CDC FC empatou?");
```

Adicionamos a essas mesmas linhas o `parseInt` para já transformar o texto digitado em um número inteiro:

```
var vitorias = parseInt(
    prompt("Quantos jogos o CDC FC venceu?"));
var empates = parseInt(
    prompt("Quantos jogos o CDC FC empatou?"));
```

Em seguida, calculamos a quantidade de pontos, considerando que cada vitória vale 3 pontos e cada empate vale 1 ponto:

```
var pontos = (vitorias * 3) + empates;
```

Vamos considerar que o outro time, Livros Velhos, está com 28 pontos. Precisamos, então, fazer o nosso programa mostrar se estamos na frente em número de pontos, atrás ou empatados. Para realizar essas condições, usamos o `if` :

```
<script>
    var vitorias = parseInt(
        prompt("Quantos jogos o CDC FC venceu?"));
    var empates = parseInt(
        prompt("Quantos jogos o CDC FC empatou?"));

    var pontos = (vitorias * 3) + empates;

    if(pontos > 28) {
        mostra("Nosso time está indo MELHOR que Livros Velhos");
    }

    if(pontos < 28) {
        mostra("Nosso time está indo PIOR que Livros Velhos");
    }

    if(pontos == 28) {
        mostra("Nosso time está EMPATADO com Livros Velhos");
    }
</script>
```

```
}  
</script>
```

Agora podemos incrementar ainda mais o nosso programa que compara os dois times no campeonato. Faça as alterações a seguir:

1) Nesse momento, o número de pontos do outro time está fixado em 28. Faça seu programa perguntar quantas vitórias e empates o outro time também possui, e utilize essas informações para calcular sua pontuação. Para isso, você fará mais duas perguntas com `prompt` e calculará uma nova variável `pontos_adversario`.

2) Crie uma função `calculaPontos` que recebe o número de vitórias e de empates, retornando o total de pontos. Declare-a no topo do seu código, dessa forma:

```
function calculaPontos(vitorias, empates) {  
    return (vitorias * 3) + empates;  
}
```

Agora altere o restante do seu código para que ele passe a utilizá-la, em vez de escrever o cálculo toda vez. Perceba que você deve já passar os empates depois de ter aplicado o `parseInt` na variável que você pegou do `prompt`.

3) E se a regra do campeonato brasileiro mudasse? Para deixar os times mais agressivos, uma derrota poderia valer -1 ponto. Altere seu programa para que isso seja contabilizado. Para isso, você vai precisar perguntar também o número de derrotas do nosso time.

4.4 O SEU IMC ESTÁ IDEAL?

Podemos aplicar o `if` no nosso código do IMC para dar um diagnóstico de acordo com seu valor. Abra o seu código do `imc.html` e, depois do cálculo do IMC do usuário, vamos verificar se ele está abaixo do índice de magreza:

```
var imcDoUsuario = calculaIMC(alturaDoUsuario,
                              pesoDoUsuario);
mostra("O seu imc é " + imcDoUsuario);

if(imcDoUsuario < 18.5) {
    mostra("Seu IMC indica que você está ABAIXO do peso.");
}
```

Rode o seu programa e faça o teste com 60 quilos e 1.82 de altura. Faça com alguém mais baixo, e a mensagem não deve ser mostrada.

Vamos verificar a condição oposta, fazendo o mesmo para o índice limite de obesidade:

```
if(imcDoUsuario > 35) {
    mostra("Seu IMC indica que você está ACIMA do peso.");
}
```

Faça o teste com o peso de 99 quilos e 1.6 de altura.

E se o `imcDoUsuario` está acima de 18.5 e, ao mesmo tempo, abaixo de 35 ? Você pode verificar essas duas situações, uma de cada vez. Primeiro fazemos o `if` para saber se está acima de 18.5 :

```
if(imcDoUsuario > 18.5) {
    // nao basta estar acima de 18.5
    // preciso verificar se está abaixo de 35
}
```

Mas, aí dentro, preciso verificar se está abaixo de 35. Como verificar uma condição? Com um `if` . Então posso escrever um

outro aí dentro:

```
if(imcDoUsuario > 18.5) {  
    if(imcDoUsuario < 35) {  
        mostra("OK! Seu IMC está entre os dois limites.");  
    }  
}
```

Parece complicado. Temos um `if` dentro de outro! Repare que usamos o recurso de *indentação* para deixar bem claro que o segundo `if` está dentro do primeiro. Se não tivéssemos usado o `TAB`, ficaria difícil de perceber isso. Quer ver? Olhe só o exemplo de uma péssima indentação, mal formatado:

```
if(imcDoUsuario > 18.5) {  
if(imcDoUsuario < 35) {  
mostra("OK! Seu IMC está entre os dois limites.");  
}  
}
```

Apesar de esse código funcionar perfeitamente, é bastante difícil lê-lo e saber o que acontece. Esforce-se desde agora a manter a indentação correta. Devemos sempre ter o código um `TAB` a mais para a direita quando ele pertence ao trecho de código (**bloco de código**) de fora. No código bem indentado, fica claro que a função `mostra` pertence ao bloco do `if(imcDoUsuario < 35)` que, por sua vez, pertence ao bloco do `if(imcDoUsuario > 18.5)`.

Muitas vezes precisamos verificar se mais de uma condição é verdadeira. Essa situação é tão comum que há uma forma mais curta de fazer dois `if`s, como se fosse um dentro do outro. Há uma forma de escrever "*quero que isso E isso seja verdadeiro*". Esse `E` é feito com um operador que parece estranho, o `&&`:

```
if(imcDoUsuario > 18.5 && imcDoUsuario < 35) {  
    mostra("OK! Seu IMC está entre os dois limites.");  
}
```

```
}
```

Bem mais legível que precisar escrever dois `if` s. Lemos essa condição como *"se o `imcDoUsuario` é maior que 18.5 E o `imcDoUsuario` é menor que 35, faça..."*.

Há ainda um pequeno problema com nosso código. E se o IMC for exatamente igual a 18.5 ou a 35 ? Repare que nenhuma das nossas condições será verdadeira, e nada será impresso. Para isso, temos os comparadores `>=` e `<=` . Se considerarmos que esses valores exatos fazem parte do IMC sadio, podemos então escrever a condição com esses novos operadores de comparação:

```
if(imcDoUsuario >= 18.5 && imcDoUsuario <= 35) {  
    mostra("OK! Seu IMC está entre os dois limites.");  
}
```

4.5 JOGO: ADIVINHE O NÚMERO QUE ESTOU PENSANDO

Vamos criar agora nosso primeiro jogo! Será bem simples, mas o suficiente para garantir alguns instantes de diversão. Faremos o computador "pensar" em um número e a pessoa que estiver jogando tentará descobrir qual é esse número.

Para conseguirmos escrever nosso jogo, crie o arquivo `jogo_adivinha.html` . A primeira tarefa que precisamos fazer é pedir para o computador "pensar" em um número aleatório. Conseguimos isso por meio da função que já vem pronta no JavaScript, chamada `Math.random()` .

Não se assuste com o nome, essa função simplesmente devolve um número aleatório entre 0 e 1 . Assim, ela pode devolver, por

exemplo, 0.5 .

```
<script>
    var numeroPensado = Math.random();
</script>
```

Ficar adivinhando números com casas decimais não é uma tarefa muito agradável. Imagine se o número pensado pelo computador fosse 0.534345, não seria fácil, nem divertido, adivinhar um número assim.

Vamos fazer o programa pensar necessariamente em um número entre 0 e 100 . Infelizmente, não há uma função pronta que faça o sorteio desta forma, então vamos aproveitar e utilizar a `Math.random()` que já conhecemos. Para isso, multiplicaremos o resultado devolvido por 100 . Então, se ele pensar 0.5 , multiplicando por 100 , o resultado será 50 .

```
<script>
    var numeroPensado = Math.random() * 100;
</script>
```

Ainda assim, podemos continuar com números cheios de casas decimais. Se o computador pensar em 0.5372 , ao multiplicar por 100, teremos 53.72 . Podemos arredondar esse valor por meio da função `Math.round()` . Vamos primeiro guardar o número com casas decimais em `numeroPensadoComCasasDecimais` , e depois arredondar e guardá-lo na variável `numeroPensado` :

```
<script>
    var numeroPensadoComCasasDecimais = Math.random() * 100;
    var numeroPensado =
        Math.round(numeroPensadoComCasasDecimais);
</script>
```

A variável `numeroPensadoComCasasDecimais` não vai ser mais utilizada. Nós a criamos apenas para poder calcular o número

sorteado arredondado, que está em `numeroPensado` . É muito comum usarmos algumas variáveis rapidamente, apenas por um curto tempo, para depois ter o resultado importante em uma outra. Algumas pessoas se referem a essas variáveis como **variáveis temporárias**.

Repare que poderíamos ter feito as duas operações (sortear e arredondar) em uma mesma linha de código, sem a ajuda da variável `numeroPensadoComCasasDecimais` . Para isso, passamos o que o `Math.random() * 100` retorna diretamente para o `Math.round` :

```
<script>
    var numeroPensado = Math.round(Math.random() * 100);
</script>
```

Qual das duas abordagens é a melhor? Não há uma resposta precisa, mas é importante evitar que linhas de código fiquem muito complicadas. Às vezes, adicionar variáveis temporárias ajudam na legibilidade do seu código.

Pronto, agora temos um número inteiro que será de 0 a 100. Basta perguntar para o usuário qual é o número que ele acha que o computador pensou e comparar para saber se acertou ou não.

```
<script>
    var numeroPensado = Math.round(Math.random() * 100);

    var chute = prompt("Já pensei. Qual você acha que é?");
    if(chute == numeroPensado) {
        mostra("Uau! Você acertou, pois eu pensei no "
            + numeroPensado);
    }
</script>
```

Mas e quando o usuário errar o número? Vamos mostrar uma mensagem indicando que o chute dele foi errado! Para isso, vamos

precisar verificar **se** o usuário errou, ou seja, se o chute foi diferente do número pensado.

Aprendemos que, para verificar se um valor é igual a outro, podemos utilizar o `==`, no entanto, agora temos de verificar se eles são diferentes. Para isso, podemos usar o `!=` (exclamação e igual). Esse sinal significa que queremos saber se um valor é diferente do outro:

```
var numeroPensado = Math.round(Math.random() * 100);

var chute = prompt("Já pensei. Qual você acha que é?");
if(chute == numeroPensado) {
    mostra("Uau! Você acertou, pois eu pensei no "
        + numeroPensado);
}
if(chute != numeroPensado) {
    mostra("Você errou! Eu tinha pensado no " + numeroPensado);
}
```

Repare que fizemos dois `if` s, sendo que um é o caso contrário do outro. Se os números não são iguais, eles então só podem ser diferentes. Não existe outra possibilidade!

Para essas situações, em que a condição de um `if` é exatamente a oposta do outro, podemos utilizar a palavra-chave `else` (*senão, caso contrário*). Usando o `else`, nem precisaríamos escrever a segunda condição:

```
var numeroPensado = Math.round(Math.random() * 100);

var chute = prompt("Já pensei. Qual você acha que é?");
if(chute == numeroPensado) {
    mostra("Uau! Você acertou, pois eu pensei no "
        + numeroPensado);
} else {
    mostra("Você errou! Eu tinha pensado no " + numeroPensado);
}
```

Você pode ler esse código como: se `chute == numeroPensado` , imprima que ele acertou; **caso contrário**, imprima que errou. Repare que no `else` não escrevemos a condição, pois ele só será acionado no caso da condição escrita no `if` for falsa.

Está difícil de acertar? Você pode diminuir o intervalo de números a serem sorteados, simplesmente mudando a multiplicação. Em vez de multiplicar por `100` , multiplique por `10` , assim o número gerado será de `0` a `10` .

No próximo capítulo, também teremos uma boa ajuda para acertarmos os números: poderemos chutar mais vezes!

4.6 REVISANDO SEU CÓDIGO: O JOGO DA ADIVINHAÇÃO

Acabamos de fazer nosso primeiro jogo! Podemos deixá-lo ainda mais interessante e divertido, mas vamos ver isso no próximo capítulo. Vamos revisar o código que escrevemos e fazer alguns ajustes.

Primeiramente, no arquivo `jogo_adivinha.html` , pedimos para o computador "pensar" em um número aleatório por meio do `Math.random()` , e multiplicamos esse valor pensado por `100`, assim teremos um número entre `0` e `100`, e por fim, o arredondamos, para termos um número inteiro:

```
<script>
    var numeroPensado = Math.round(Math.random() * 100);
</script>
```

Em seguida, perguntamos para o usuário um número, seu

chute , para ele tentar adivinhar o que o computador pensou. Com esse número, verificamos se o usuário estava certo.

```
<script>
    var numeroPensado = Math.round(Math.random() * 100);

    var chute = prompt("Já pensei. Qual você acha que é?");
    if(chute == numeroPensado) {
        mostra("Uau! Você acertou, pois eu pensei no "
            + numeroPensado);
    }
</script>
```

Lembre-se de ter a função `mostra` no seu código.

Depois, mostramos uma mensagem caso o chute tenha sido errado. Para isso, usamos o `else` :

```
<script>
    var numeroPensado = Math.round(Math.random() * 100);

    var chute = prompt("Já pensei. Qual você acha que é?");
    if(chute == numeroPensado) {
        mostra("Uau! Você acertou, pois eu pensei no "
            + numeroPensado);
    } else {
        mostra("Você errou! Eu tinha pensado no "
            + numeroPensado);
    }
</script>
```

Pronto, agora ao abrir esse programa em seu navegador, você será perguntado por um número. E logo em seguida, será mostrado se era o mesmo número que o computador pensou ou não.

1) Faça com que seu jogo mostre, quando o usuário errar a tentativa, se o número que ele chutou era maior ou menor ao número pensado pelo programa.

2) Você pode criar uma função `sorteia`, que recebe um número `n` e sorteia um número entre 0 a `n`, retornando esse valor. Dessa forma, em vez de escrever `var numeroPensado = Math.round(Math.random() * 100);`, você escreveria `var numeroPensado = sorteia(100);`. Faça essa modificação, criando essa nova função e utilizando-a.

3) E se o usuário digitar algo que não é um número? Se ele chutar, por exemplo, "banana"? Não seria bom ele perder uma chance de chute...

Você pode verificar se o usuário digitou algo que não é um número pela função `isNaN(variavel)`. Por exemplo, `if(isNaN(chute))` deve mostrar uma mensagem *apenas números são válidos*. `NaN` significa *Not a Number*, em português algo como "não é um número". Teste essa função no console do Chrome, fazendo `isNaN("banana");` e `isNaN(123);`.

COMO REPETIR TAREFAS DO PROGRAMA?

Já podemos executar uma linha de código de acordo com uma condição. Por meio do `if` e `else`, conseguimos definir qual trecho deve ser executado. Mas e se precisarmos executá-lo mais de uma vez? Será que o copiar e colar o código pode nos ajudar nessa tarefa?

5.1 QUANDO SERÃO AS PRÓXIMAS COPAS DO MUNDO?

Desde 1930, é disputada a copa do mundo de futebol, que a cada edição possui uma sede diferente. A periodicidade do evento é de 4 anos. Será que em 2030 vamos ter copa do mundo?

Uma maneira que temos de saber isso é tomar como base um ano em que a edição ocorrerá, como 2014, e somar 4 anos. Logo sabemos que 2018 é um ano no qual teremos copa do mundo. Se continuarmos esse processo, o próximo ano de copa será 2022, em seguida, 2026 e pronto, chegamos ao 2030 e descobrimos que sim, haverá copa do mundo. **Poderíamos continuar esse processo infinitamente, enquanto não nos cansarmos.**

Vamos escrever um programa que mostra os anos de copa, desde 1930, de uma forma extremamente simples. Basta termos uma variável cujo valor seja 1930 e que sempre tenha seu valor aumentado de 4 em 4.

```
var anoDeCopa = 1930;
alert(anoDeCopa + " tem copa!");

anoDeCopa = anoDeCopa + 4;
alert(anoDeCopa + " tem copa!");

anoDeCopa = anoDeCopa + 4;
alert(anoDeCopa + " tem copa!");

anoDeCopa = anoDeCopa + 4;
alert(anoDeCopa + " tem copa!");
```

Poderíamos continuar repetindo esse código até a exaustão. Mas repare o trabalho que teríamos. E se quisermos mais copas? Quanto código não teremos de escrever? É muito trabalho repetitivo! Vamos tentar uma abordagem um pouco diferente.

Queremos imprimir os anos da copa, de 4 em 4, para saber todas as copas. Será possível? A ideia é criar um programa que some 4 na variável `anoDeCopa`, alerte-nos esse valor, e volte a executar o mesmo procedimento.

Seguindo esse raciocínio, podemos criar um novo programa, no arquivo `anos_de_copa.html`, que mostra para nós os anos em que teve e terá copa do mundo, desde 1930:

```
<script>
var anoDeCopa = 1930;

while(true) {
    alert(anoDeCopa + " tem copa!");
    anoDeCopa = anoDeCopa + 4;
}
```

</script>

Começamos com a variável `anoDeCopa` iniciando em 1930, que foi o ano do primeiro evento. Em seguida, usamos o comando `while` (*enquanto*), que significa que os comandos dentro das `{ }` serão realizados quantas vezes quisermos, como se fosse uma **repetição**, que também é bastante conhecida como **iteração**.

Então, mostramos que aquele ano possui copa e mudamos o conteúdo da variável `anoDeCopa` para ir 4 anos adiante, simplesmente somando 4 no `anoDeCopa` e guardando o resultado na mesma variável.

Quando executamos esse programa, vemos uma mensagem de alerta aparecendo na tela. Se seguirmos clicando em *OK*, veremos que para cada ano de copa, um novo alerta é mostrado. Se tivermos paciência para clicar várias vezes, vamos até descobrir que não haverá copa no ano de 2100! Repare que podemos clicar infinitamente em *OK*, que uma mensagem sempre aparecerá, ou seja, ficamos em uma repetição infinita, também conhecido como **loop infinito**.

CUIDADOS COM ALERTS E LOOP INFINITO EM BROWSERS No código para exibir os anos de copa, mostramos uma mensagem de alerta para cada um dos anos. Isso faz com que muitas mensagens sejam exibidas sem parar.

Alguns navegadores como o Firefox e o Chrome permitem parar a exibição das mensagens logo que a segunda mensagem aparece. No entanto, o Internet Explorer não faz o mesmo, ou seja, seu navegador exibirá as mensagens e nunca parará, até que você force seu fechamento.

Apesar de muito interessante, talvez estejamos mostrando mais informações do que o necessário. Principalmente, se quisermos saber **apenas** os anos em que houve copa, **até** 2014, ano da copa no Brasil.

Em português, é frequente aparecer o termo **laço** em vez de loop, e dizemos que o laço **itera** (repete) determinado trecho de código que queremos. É a repetição de iterações a que já havíamos dado nomes.

5.2 REALIZE O LOOP SOMENTE EM DETERMINADAS CONDIÇÕES

O `while`, de forma similar ao `if`, aceita que indiquemos uma condição, para que a cada repetição uma avaliação seja feita, para saber se deve executar ou não o conteúdo da nossa repetição (*loop*), o **bloco** de código dentro dele. Na seção anterior, apenas

utilizamos `while(true)` , indicando que o bloco de código deve **sempre** repetir.

Queremos melhorar e evitar o loop infinito. Desejamos que o bloco seja executado somente **enquanto anoDeCopa for menor ou igual a 2014**.

```
<script>
var anoDeCopa = 1930;

while(anoDeCopa <= 2014) {
    alert(anoDeCopa + " tem copa!");
    anoDeCopa = anoDeCopa + 4;
}
</script>
```

Agora indicamos para o `while` que ele deve, a cada repetição, verificar se o valor da variável `anoDeCopa` ainda está menor que 2014. Quando o valor do `anoDeCopa` for maior que 2014, a condição do `while` não será satisfeita e o bloco não será executado novamente, pulando para comando que vier logo após, isto é, após o fechar chaves.

Abra em seu navegador o arquivo `anos_de_copa.html` , e perceba que ele mostrará um primeiro alerta para 1930. Quando clicamos em *Ok*, o programa nos mostra 1934 e assim até chegar em 2014.

Ficar mostrando `alert` já deve estar começando a ficar desagradável para você, certo? Se ainda não ficou, logo ficará! Vamos passar a usar a função `mostra` que fizemos nos capítulos anteriores em vez do `alert` dentro do `while` , como vimos no capítulo [ref-label comunicandose].

Primeiro, vamos escrever novamente o código da função

mostra . Antes também criávamos a função `pulaLinha` , mais por uma questão didática, tendo em vista que não reutilizamos o `pulaLinha` em outros locais, nem ela é muito complexa. Dessa vez, faremos as duas coisas na mesma função: chamar o `document.write` e adicionar o `
` ao final.:

```
function mostra(frase) {  
    document.write(frase + "<br>");  
}
```

Agora troque a linha na qual você estava usando o `alert` para usar o `mostra` .

```
mostra(anoDeCopa + " tem copa!");
```

Podemos inclusive mostrar uma mensagem indicando que acabou:

```
<script>  
var anoDeCopa = 1930;  
  
while(anoDeCopa <= 2014) {  
    mostra(anoDeCopa + " tem copa!");  
    anoDeCopa = anoDeCopa + 4;  
}  
  
mostra("Ufa! Esses foram os anos de copa até 2014.");  
</script>
```

Você pode pensar no `while` como se a cada repetição ele fizesse a pergunta: *Posso continuar?* A condição dentro dos parênteses (no nosso caso `anoDeCopa <= 2014`) será usada para decidir. Se a resposta for `true` , significa que pode repetir o bloco mais uma vez; caso contrário, ele deve parar as repetições e ir para o próximo comando após o bloco do `while` , delimitado através das chaves.

5.3 REVISE SEU CÓDIGO: MOSTRE OS ANOS DE COPAS ATÉ CANSAR

Vamos relembrar o que fizemos com o arquivo `anos_de_copa.html` . Estamos criando um programa que nos facilite descobrir quais são os anos em que haverá ou houve copa do mundo. Nesse instante, temos no nosso código a definição da função `mostra` que exibirá os anos no navegador e a inicialização da variável `anoDeCopa` em 1930, pois foi quando ocorreu o evento pela primeira vez.

```
<script>
function mostra(frase) {
    document.write(frase + "<br>");
}

var anoDeCopa = 1930;
```

Agora queremos imprimir os anos de copa **enquanto for antes de 2014**, lembrando de que elas acontecem de 4 em 4 anos.

```
while(anoDeCopa <= 2014) {
    mostra(anoDeCopa + " tem copa!");
    anoDeCopa = anoDeCopa + 4;
}
</script>
```

No final de tudo, podemos mostrar que chegamos ao fim da exibição dos anos de copa. Quando a condição que temos no `while` não for mais verdadeira, o programa executa o código logo após o bloco do `while` , ou seja, vai para imediatamente após o fechamento das chaves.

Podemos adicionar uma linha para o programa mostrar uma mensagem, logo após o bloco do `while` .

```
mostra("Ufa! Esses foram os anos de copa até 2014.");
```

1) Agora que conseguimos mostrar os anos de copa, parece que nosso computador nem se cansou com essa tarefa. Aumente seu programa para mostrar as copas até 2100. Basta trocar o ano limite na condição do `while` . Abra o arquivo `anos_de_copa.html` no seu navegador e veja se ele demorará mais tempo.

2) Incrível, ele faz todo esse trabalho e não cansa! Imagine se fôssemos nós realizando todas essas contas. Você pode até mesmo testar com números grandes, como 20.000, 50.000 ou 100.000.

```
while(anoDeCopa <= 100000)
```

Você pode até perceber um aumento do tempo, mas provavelmente continuará muito pequeno, apenas o conteúdo mostrado no navegador que deverá ter muito mais informações.

3) Note que, para mudarmos o ano limite, temos de alterar o valor na condição do `while` . Ou seja, sempre temos de ficar alterando o arquivo. Vamos deixar isso mais flexível, perguntando para o usuário qual é o ano limite que ele quer saber se há copa do mundo.

Aprendemos que podemos perguntar informações para o usuário por meio do `prompt` . Capture o `anoLimite` através de uma linha como `var anoLimite = prompt("Qual é o ano limite?");` , e depois utilize essa variável para fazer seu `while` . A condição vai ficar `anoDeCopa <= anoLimite` .

4) Abra o programa novamente no seu navegador e verifique que agora, antes de começar a execução, ele nos pergunta qual é o ano limite. Digite 2050, por exemplo, e veja o resultado.

5) O que será que acontece se pedirmos para mostrar as copas

até o ano 100, ano em que a copa do mundo nem havia sido criada ainda?

Faça esse teste e repare que nosso programa é bem esperto, já sabe até que naquele ano não havia copas do mundo. Mas como ele fez isso?

Simples. Na condição do nosso `while`, estamos falando que o `anoDeCopa` deve ser menor ou igual ao `anoLimite` para que os anos sejam mostrados no navegador. Ou seja, como o ano de copa inicia em 1930, qualquer ano limite que seja inferior a isso vai fazer com que a condição dê sempre falso e, com isso, ele nunca executará o bloco do `while`.

5.4 CARACTERES E NÚMEROS, QUAL É A DIFERENÇA AFINAL?

Nesse momento, com o nosso programa perguntando qual é a data limite que ele deve mostrar os anos de copa do mundo, temos o seguinte código:

```
<script>
function mostra(frase) {
    document.write(frase + "<br>");
}

var anoDeCopa = 1930;
var anoLimite = prompt("Qual é o ano limite?");

while(anoDeCopa <= anoLimite) {
    mostra(anoDeCopa + " tem copa!");
    anoDeCopa = anoDeCopa + 4;
}
mostra("Ufa! Esses foram os anos de copa até " + anoLimite);
</script>
```

Nosso programa sempre começa a mostrar os anos a partir de 1930. Podemos ir além e deixá-lo flexível para que também pergunte qual o ano em que deve começar a mostrar. Por exemplo, podemos querer mostrar apenas as copas do século XXI — entre os anos 2001 e 2100.

Vamos parar de iniciar a variável com 1930 e passar a perguntar qual o ano inicial que ele quer.

```
var anoDeCopa = prompt("Informe o ano inicial");
```

Quando executamos esse programa com o ano inicial para 1930 e ano limite de 2014, algo errado acontece!

Repare que ele só mostrou o primeiro ano, os demais não são mostrados. Nosso programa ficou maluco? Não, há um bom motivo pra obtermos um resultado estranho como esse, e vamos explicar com muita calma agora.

Primeiro, precisamos entender o que está acontecendo. Será a primeira vez que entraremos mais a fundo em como o JavaScript funciona. É normal que cada linguagem de programação tenha características não tão óbvias à primeira vista, e faz parte do aprendizado encará-las.

Quero saber por que não saiu como eu esperava!

Para começarmos a entender o motivo da falha, precisamos analisar o que o programa fez. Repare que, na primeira vez, ele conseguiu entrar no `while`, tanto que ele mostrou a mensagem: 1930 é ano de copa! . Com isso, já sabemos que a condição do `while` foi verdadeira logo no começo, o que faz sentido, pois, 1930 é menor que 2014, que foi o ano limite que informamos.

Mas ainda não entendemos o motivo de não ter continuado, já que estamos somando 4 no ano inicial que foi informado e, com isso, agora ele deveria ser 1934.

Vamos mostrar na tela qual é o valor do ano após a soma, dessa forma, poderemos visualizar o que está sendo comparado. Adicione como última instrução do bloco `while` a linha que mostra o conteúdo da variável `anoDeCopa`, assim saberemos o seu conteúdo após a soma.

```
while(anoDeCopa <= anoLimite) {  
    mostra(anoDeCopa + " tem copa!");  
    anoDeCopa = anoDeCopa + 4;  
    mostra(anoDeCopa + " esse é o valor após a soma!");  
}
```

Abrindo novamente esse arquivo no navegador, temos uma grande surpresa.

O número resultante da soma não é 1934, e sim 19304? Como assim? $1930 + 4 = 19304$??? Nosso programa não somou 4. Ele juntou o texto logo no começo, ou seja, ele realizou a **concatenação**, que vimos no capítulo *Comunique-se com o usuário*.

Vimos o `parseInt()` no capítulo anterior. O problema aqui foi ter trabalhado com a variável que representa o ano como sendo uma `string`, já que a função `prompt` sempre devolve dessa maneira.

Podemos aplicar essa função para resolver o problema do nosso código. Primeiramente, vamos continuar perguntando para o usuário qual o ano limite que ele está usando, mas lembrando de que isso nos é devolvido como uma `string` e precisamos

transformá-la em um número:

```
var anoComoTexto = prompt("Informe o ano inicial");  
var anoDeCopa = parseInt(anoComoTexto);
```

Com essa simples alteração, nosso programa volta a estar pronto para funcionar no navegador.

5.5 REVISE SEU CÓDIGO: TRANSFORME TEXTO EM NÚMEROS

Estávamos com nosso programa `anos_de_copa.html` funcionando corretamente, até o momento em que decidimos perguntar para quem usa nosso programa qual é o ano em que começará a mostrar os anos da copa.

```
<script>  
function mostra(frase) {  
    document.write(frase + "<br>");  
}  
  
var anoDeCopa = prompt("Informe o ano inicial");  
var anoLimite = prompt("Qual é o ano limite?");  
  
while(anoDeCopa <= anoLimite) {  
    mostra(anoDeCopa + " tem copa!");  
    anoDeCopa = 4 + anoDeCopa;  
}  
mostra("Ufa! Esses foram os anos de copa até " + anoLimite);  
</script>
```

Quando abrimos esse arquivo no navegador, e informamos 1930 como ano inicial e 2014 como o ano limite, temos uma surpresa, não funcionou com o esperado. Os valores foram concatenados em vez de ocorrer a esperada soma.

Precisávamos somar número com número, e não número com

texto. Resolvemos isso por meio da função `parseInt` para transformar o texto que recuperamos, correspondente ao ano inicial, em um número.

```
var anoComoTexto = prompt("Informe o ano inicial");  
var anoDeCopa = parseInt(anoComoTexto);
```

Nesse momento, o código completo do programa é o seguinte:

```
<script>  
function mostra(frase) {  
    document.write(frase + "<br>");  
}  
  
var anoComoTexto = prompt("Informe o ano inicial");  
var anoDeCopa = parseInt(anoComoTexto);  
var anoLimite = prompt("Qual o ano limite?");  
  
while(anoDeCopa <= anoLimite) {  
    mostra(anoDeCopa + " é ano de copa!");  
    anoDeCopa = 4 + anoDeCopa;  
}  
mostra("Ufa! Esses foram os anos de copa até " + anoLimite);  
</script>
```

Agora podemos abrir o programa `anos_de_copa.html` no navegador, que ele nos perguntará o ano inicial e o limite e fará todo o processo corretamente.

1) Faça o mesmo programa que mostra os anos de copa, porém para as olimpíadas. Lembre-se que o primeiro ano de olimpíadas foi 1896 e que ela também acontece a cada 4 anos.

2) Faça testes no console do Chrome, como vimos na seção *Utilize o console do Chrome para fazer testes!*. Descubra quanto vale `"10" + 10`, `"10" + "10"`, `"10" * 10`, `"10" * "10"`. Deixe sua imaginação criar os mais variados exemplos e você começará a entender quando o JavaScript faz conta, quando ele

concatena, e quando ele não consegue fazer nenhum dos dois!

5.6 PRATICANDO MAIS UM POUCO: FAÇA TABUADAS

Quem nunca hesitou ao ter de falar a tabuada do 7? 7, 14, 21, 28... e esqueci! O cálculo das tabuadas é um bom exemplo de processo de repetição. Calculamos a tabuada do 7, indo do 7 vezes 1, até o 7 vezes 10. Inclusive, isso fica até mais claro com a forma costumeira com a qual alunos de escola escrevem tabuadas:

- $7 \times 1 = 7$
- $7 \times 2 = 14$
- $7 \times 3 = 21$
- $7 \times 4 = 28$
- $7 \times 5 = 35$
- $7 \times 6 = 42$
- $7 \times 7 = 49$
- $7 \times 8 = 56$
- $7 \times 9 = 63$
- $7 \times 10 = 70$

Repare que multiplicamos o 7, começando no 1, **enquanto** o 7 for menor ou igual a 10. É mais um caso em que conseguimos tranquilamente fazer com que um programa de computador realize todo o trabalho para nós e apenas mostre os resultados que queremos. Então, vamos lá!

O primeiro passo é criarmos um novo arquivo, que podemos chamar de `tabuadas.html`. Nesse arquivo, vamos começar a escrever nosso código JavaScript por meio da Tag `<script>`:

```
<script>

</script>
```

Como na nossa tabuada o cálculo inicia na multiplicação por 1 e vai até multiplicarmos por 10, vamos criar uma variável que guarde esse valor, que vamos chamar de `multiplicador` e iniciará em 1.

```
<script>
    var multiplicador = 1;
</script>
```

Enquanto a variável `multiplicador` for menor ou igual a 10, vamos querer calcular a tabuada. Podemos fazer com que nosso programa realize essas multiplicações por 7, enquanto o `multiplicador` for menor ou igual a 10. Lembre-se de copiar a função `mostra` para o arquivo `tabuadas.html`, assim, conseguiremos exibir os resultados da tabuada no navegador.

```
<script>
    function mostra(frase) {
        document.write(frase + "<br>");
    }

    var multiplicador = 1;
    while(multiplicador <= 10) {
        mostra(7 * multiplicador);
    }
</script>
```

Mas se tentarmos abrir esse arquivo no navegador, ele nunca vai parar de fazer multiplicações. Pior. A multiplicação que ele fará, será sempre 7 vezes 1. Isso acontece porque ainda não estamos aumentando o número do `multiplicador`, após cada conta realizada.

Precisamos que a cada repetição, o `multiplicador` aumente um.

Na programação, costumamos chamar esse processo de **incremento**. Então, precisamos incrementar 1 à variável multiplicador :

```
<script>
    function mostra(frase) {
        document.write(frase + "<br>");
    }

    var multiplicador = 1;
    while(multiplicador <= 10) {
        mostra(7 * multiplicador);
        multiplicador = multiplicador + 1;
    }
</script>
```

Ficou fácil agora, não? Vamos fazer mais um exercício para praticarmos e melhorarmos um pouco mais esse exemplo.

1) Em vez de mostrar apenas o número resultante, mostre uma mensagem mais interessante para cada multiplicação, algo como: "7 vezes 10 é igual a 70". Juntando um pouco de texto com números, fica fácil de fazer. Bastaria substituir o código atual que mostra o resultado da multiplicação por:

```
mostra("7 vezes " + multiplicador + " é igual a "
      + 7 * multiplicador);
```

Veja o resultado final no seu navegador.

2) Em vez de sempre realizarmos a tabuada do 7, pergunte ao usuário qual é a tabuada que ele deseja saber. Basta que, antes das repetições, você use um prompt e guarde o número informado em uma variável, que deverá ser usada para calcular esse resultado.

5.7 APRENDA UMA FORMA DIFERENTE DE

MOSTRAR A TABUADA: O COMANDO FOR

Geralmente, quando temos as repetições que não sejam um *loop infinito*, temos 3 características que a envolvem:

- Um valor inicial, que no caso da tabuada é o multiplicador iniciando em 1;
- Uma condição que determine se a repetição deve ser feita ou não, que no nosso caso é quando verificamos que o multiplicador ainda é menor que 10;
- Uma modificação no valor que cause o fim da repetição, ou seja, que tenha influência na condição que é verificada para a repetição, justamente para que não fique em repetições infinitas. No caso da tabuada, é quando aumentamos o valor do multiplicador a cada repetição, ou seja, o incremento.

Até agora, conseguimos realizar essas repetições pelo `while`, sempre escrevendo o código necessário para essas 3 características, porém, não há nada que nos induza, quando estamos escrevendo o código do `while` a lembrar dessas características.

Justamente com esse intuito, existe o comando `for`, que nos induz a pensar em tudo antes de montar o código da repetição. A sintaxe, ou seja, o jeito de escrever o comando, pode parecer um pouco estranha no começo, mas vamos entendê-lo aos poucos.

```
for( ; ; ) {  
    alert("mensagem para repetição infinita");  
}
```

Sintaxe estranha, não? Primeiramente, ao abrir um programa com esse código em um navegador, perceberemos que será feito

um *loop* infinito, ou seja, o `for` está fazendo sim uma repetição. Mas quando ele termina? Nunca. Um `for(; ;)` é equivalente ao `while(true)` que vimos no começo do capítulo.

Repare que, dentro dos parênteses do comando `for`, há dois `;` (ponto e vírgula), dando a sensação de que há ali um espaço para que algumas três informações sejam dadas. O código a seguir não é válido, é apenas para mostrar onde estão esses três espaços:

```
for(espaco 1; espaco 2; espaco 3)
```

Cada um desses espaços espera uma informação para controlar as repetições que acontecerão. Vamos fazer um programa que mostrará 10 alertas, cada um com um número de 1 a 10. Novamente teremos uma repetição, que agora deve ir de 1 até 10. Você lembra como faria isso com o `while`? Seria um código como o seguinte:

```
var numero = 1
while(numero <= 10) {
    alert(numero);
    numero = numero + 1;
}
```

Pois bem, podemos fazer praticamente idêntico com o `for`:

```
var numero = 1;
for(; numero <= 10; ) {
    alert(numero);
    numero = numero + 1;
}
```

Preenchemos o segundo espaço com a condição do loop, que será verificada toda vez para saber se mais uma repetição deve ocorrer.

Qual é a diferença disso para o já conhecido `while`? Dentro

do `for` , apareceram os pontos e vírgulas, que tornaram a forma de escrever bastante estranha. O resultado final será exatamente o mesmo. Então, para que um outro comando de repetição?

Com o `for` , podemos usar o primeiro espaço antes do ponto e vírgula para realizar inicializações. Um exemplo é inicializar a variável `numero` :

```
for(var numero = 1; numero <= 10; ) {  
    alert(numero);  
    numero = numero + 1;  
}
```

E agora, o que mudou? Quase nada. Seu código ficou mais legível. A parte que diz `var numero = 1` é chamada de inicialização do `for` . Ela será executada apenas uma única vez. A próxima etapa é verificar se `numero <= 10` e decidir se deve ou não executar uma repetição.

E o que podemos fazer no terceiro espaço, logo após o segundo ponto e vírgula? O que estiver nessa terceira parte será sempre executada. É muito usado para executar o código que aumenta, que **incrementa** a variável que usamos durante o loop. Nesse caso, para aumentar o valor da nossa variável `numero` . Com isso, podemos escrever um comando que imprima de 1 até 10 com esse novo comando da seguinte maneira:

```
for(var numero = 1; numero <= 10; numero = numero + 1) {  
    alert(numero);  
}
```

Portanto, podemos perceber que o `for` possui 3 espaços/pedaços: o primeiro, onde dizemos qual é o valor inicial que queremos; o segundo para definirmos qual é a condição que determina se as repetições devem ser feitas ou não — essa parte é

feita da mesma forma que vimos no `while` ; e o terceiro, que modifica o valor de alguma variável que influa na finalização das repetições. As três partes são delimitadas pelo `;` .

```
for([inicialização]; [condição]; [modificação do valor])
```

O importante aqui é perceber que o que acabamos de fazer com o novo comando, o `for` , é o mesmo que conseguimos fazer com o `while` , apenas escrito de uma maneira diferente. Enquanto o `for` precisa que escrevamos as 3 características logo no meio dos parênteses do comando, o `while` apenas pede a condição.

5.8 REESCREVENDO A TABUADA COM O FOR

Com isso, como ficaria se escrevêssemos o mesmo programa que mostra na tela as tabuadas, mas agora utilizando o `for` em vez do `while` ?

Para começar, vamos nos lembrar de como era o código do programa da tabuada, que utilizava o comando `while` :

```
<script>
function mostra(frase) {
    document.write(frase + "<br>");
}

var multiplicador = 1;
while(multiplicador <= 10) {
    mostra(7 * multiplicador);
    multiplicador = multiplicador + 1;
}
</script>
```

Agora, podemos criar o arquivo `tabuada_com_for.html` , onde vamos criar nosso novo programa que vai fazer exatamente o

que o anterior fazia, de uma maneira diferente.

Sabendo que o `for`, precisa primeiro do valor inicial, é fácil identificarmos que ele é a variável `multiplicador`, que começa com o valor 1.

```
<script>
for(var multiplicador = 1; ;) {
}
</script>
```

Agora precisamos preencher as outras duas informações que o `for` precisa. A próxima é até quando as repetições devem continuar, que no caso, é a condição que indica que repetições devem ocorrer enquanto o `multiplicador` for menor que 10.

```
<script>
for(var multiplicador = 1; multiplicador <= 10; ) {
}
</script>
```

Pronto, só falta dizermos de quanto em quanto o `multiplicador` aumentará, que no nosso caso, é de 1 em 1. Chamamos isso de **incrementar** a variável. Para isso, estamos fazendo `multiplicador = multiplicador + 1`. Sempre que queremos somar 1 àquela variável, temos de repetir o nome da variável, indicando que nela queremos atribuir o valor que estava lá mais 1.

No dia a dia, muitos programadores vão preferir fazer isso de uma forma abreviada. A **sintaxe**, as regras da linguagem permitem isso. Essa linha pode ser abreviada como `multiplicador++`. Os dois sinais de soma ao lado da variável nada mais é do que uma maneira abreviada que o JavaScript entende e que significa que você quer aumentar o valor daquela variável em 1.

Vamos usar essa nova forma abreviada em nosso programa.

```
<script>
for(var multiplicador =1; multiplicador <= 10; multiplicador++){
}
</script>
```

Pronto, com isso, dissemos que o `multiplicador` vai iniciar em 1, as repetições vão acontecer enquanto `multiplicador` for menor que 10 e que, a cada repetição, o valor de `multiplicador` vai aumentar em 1.

Agora só falta adicionarmos o comportamento necessário para mostrar o resultado da tabuada do 7, já que já temos o `multiplicador` indo de 1 a 10.

```
<script>
for(var multiplicador =1; multiplicador <= 10; multiplicador++){
    mostra(multiplicador * 7);
}
</script>
```

Lembre-se de ter a função `mostra` no seu arquivo `tabuada_com_for.html` . Quando você abrir esse arquivo em um navegador, o resultado será exatamente igual ao que tínhamos antes com o `while` .

Devo usar o `while` ou o `for`?

Mas e agora? Qual das duas maneiras devo usar? Resposta simples, a que você preferir e achar mais simples de escrever, pois ambas as maneiras, seja com o `for` , seja com o `while` , fazem exatamente o mesmo trabalho. Escolha a forma que você achar mais interessante e fácil de entender e continue com ela. Para a sequência do livro, vamos realizar as repetições de acordo com o caso.

5.9 A MÉDIA DE IDADES, MAS DE UMA FORMA MAIS INTERESSANTE

Agora que entendemos o funcionamento das repetições por meio do `while` e do `for`, podemos partir para problemas mais elaborados. Vamos revisar o cálculo das médias de idades.

Até agora, precisávamos criar uma variável para cada idade que gostaríamos de realizar a soma. Então tínhamos:

```
var idadeAdriano = 26;
var idadePaulo = 32;
var idadeSuzana = 25;
```

Sempre que quisermos incluir mais uma pessoa no cálculo da média de idade, precisamos criar uma nova variável para ela e incluí-la no cálculo da média:

```
var idadeAdriano = 26;
var idadePaulo = 32;
var idadeSuzana = 25;
var idadeMarcela = 28;

var media =
    (idadeAdriano + idadePaulo + idadeSuzana + idadeMarcela) / 4;
mostra(media);
```

Ou seja, sempre que quisermos desconsiderar uma pessoa ou considerar uma nova pessoa no cálculo da média de idades, temos um baita trabalho de modificar esse código. Será que conseguimos fazer esse cálculo de média de idades de uma maneira genérica, ou seja, que funcione para qualquer quantidade de pessoas?

Esse é um problema um pouco mais sutil que os anteriores e vamos resolvê-lo em um arquivo chamado `media_idade_familiares.html`. Crie esse novo arquivo.

Se observarmos bem, temos novamente uma característica de repetição. Nesse caso, repetimos o processo da criação de uma variável. Queremos que esse processo funcione de uma maneira genérica, ou seja, quem estiver usando nosso programa vai dizer quantos familiares ele quer colocar no cálculo das médias. Podemos pedir essa informação pelo prompt .

```
<script>
var totalDeFamiliares = prompt("Quantos familiares são?");
</script>
```

Agora que sabemos quantos familiares existem, podemos perguntar, um por um, suas idades. Dessa forma, se fosse informado que são 4 familiares, precisamos perguntar as 4 idades. Repare que temos uma repetição, que nesse caso, vai de 1 até o número total de familiares.

```
var totalDeFamiliares = prompt("Quantos familiares são?");

var numero = 1;
while(numero <= totalDeFamiliares) {
    numero++;
}
```

Analisando o código, temos uma nova variável, chamada numero . Ela é apenas para nos auxiliar a saber até quando temos de repetir o trecho dentro do while . A cada repetição, aumentamos seu valor em 1, por meio do numero++ .

A cada repetição, precisamos perguntar qual é a idade da pessoa.

```
var totalDeFamiliares = prompt("Quantos familiares são?");

var numero = 1;
while(numero <= totalDeFamiliares) {
    var idadeTexto = prompt("Qual é a idade?");
```



```
    var idade = parseInt(idadeTexto);  
    numero++;  
}
```

Repare que usamos o `parseInt` para transformar o valor digitado no `prompt` em um número. Isso porque vamos somar esse valor `idade`. Se o deixássemos como `string`, os valores seriam concatenados, como já vimos.

É comum fazermos tudo isso em uma única linha, descartando a necessidade de criar a variável `idadeTexto`. Esse é aquele tipo de variável que chamamos de temporária, já que ela só existe para nos auxiliar a calcular a `idade`.

Para fazer direto, usamos `parseInt(prompt("Qual é a idade?"))`. Parece complicado, mas não é. O JavaScript sempre começará a execução pelos parênteses mais internos, executando o `prompt`. Com o resultado do `prompt` em mãos, ele executará o `parseInt`, devolvendo o valor do que foi digitado, porém como número, pronto para ser usado corretamente em operações matemáticas.

Nosso código ficou então:

```
var totalDeFamiliares = prompt("Quantos familiares são?");  
  
var numero = 1;  
while(numero <= totalDeFamiliares) {  
    var idade = parseInt(prompt("Qual é a idade?"));  
    numero++;  
}
```

Para calcularmos a média, somamos todas as idades e dividimos pelo `totalDeFamiliares` que temos, porém, a cada repetição, pegamos uma nova idade e não guardamos o total das idades que já podemos ter pego em repetições anteriores. Para

resolver esse problema, vamos criar uma nova variável, chamada `somaDeIdades`, que vai começar com o valor 0 e a cada repetição vamos somar com a idade digitada.

```
var totalDeFamiliares = prompt("Quantos familiares são?");
var somaDeIdades = 0;

var numero = 1;
while(numero <= totalDeFamiliares) {
    var idade = parseInt(prompt("Qual é a idade?"));
    somaDeIdades = somaDeIdades + idade;

    numero++;
}
```

Com isso, considere que temos 3 familiares. O primeiro com 20, o segundo com 30 e o terceiro com 40 anos. Na primeira repetição, o valor de `somaDeIdades` que estará em 0 será somado com 20, ou seja, seu novo valor agora é 20. Na segunda repetição, `somaDeIdades` que agora tem o valor 20, será somado com 30 e vai para 50. Por fim, `somaDeIdades` que agora é 50, vai para 90, com a soma por 40 da idade do último familiar.

No fim das repetições, o valor de soma de idades foi para 90. Agora que acabamos as repetições, se queremos saber a média de idades, basta dividirmos essa `somaDasIdades` pelo `totalDeFamiliares`. Por exemplo, com a soma 90 e 3 familiares, a média de idade será de 30 anos.

Então vamos fazer o nosso programa: após as repetições serem feitas, ou seja, depois do `while` terminar seu trabalho, calcular a média e mostrá-la. São as duas últimas linhas do código a seguir:

```
var totalDeFamiliares = prompt("Quantos familiares são?");
var somaDeIdades = 0;

var numero = 1;
```

```

while(numero <= totalDeFamiliares) {
    var idade = parseInt(prompt("Qual é a idade?"));
    somaDeIdades = somaDeIdades + idade;

    numero++;
}

var media = somaDeIdades / totalDeFamiliares;
alert("A média é: " + media);

```

Pronto, agora se você abrir esse programa em seu navegador, ele te perguntará primeiro a quantidade de familiares que você tem e, em seguida, perguntará qual é a idade para cada um deles. No final, teremos a média de idade dos familiares.

5.10 JOGO: MAIS CHANCES PARA ADIVINHAR O NÚMERO QUE ESTOU PENSANDO

No capítulo anterior, fizemos o computador pensar em um número aleatório entre 0 e 100 e tentamos adivinhar o seu resultado. Você deve ter reparado que era extremamente difícil acertar, quase desmotivante! Uma chance em 100.

Vamos fazer aquele mesmo programa nos dar mais chances para acertar. Poderemos ter, por exemplo, 3 chances para descobrir qual é o número que o computador pensou. A cada tentativa errada, o programa vai dizer se o número que tentamos é maior ou menor do que o número imaginado pelo computador.

Começaremos relembrando o código que fazia o computador pensar em um número de 0 a 100.

```

<script>
    var numeroPensado = Math.round(Math.random() * 100);

```

```

var chute = prompt("Já pensei. Qual você acha que é?");
if(chute == numeroPensado) {
    mostra("Uau! Você acertou, pois eu pensei no "
        + numeroPensado);
} else {
    mostra("Você errou! Eu tinha pensado no "
        + numeroPensado);
}
</script>

```

Logo no começo do programa, o computador pensa em um número, que é guardado. Em seguida, tentamos adivinhar esse número que o computador pensou e uma mensagem é exibida para o acerto ou erro da tentativa.

O que queremos que aconteça agora é que esse processo de chute **se repita** por 3 vezes. Já vimos que, para realizarmos uma repetição no nosso programa, podemos usar o `while`. Ou seja, enquanto não foi feita a terceira tentativa, você será perguntado novamente caso erre, e caso acerte, uma mensagem de parabéns lhe será mostrada.

Novamente, a primeira ação do programa será pensar em um número de 0 a 100, então, manteremos como a primeira instrução do programa a variável `numeroPensado`. Vamos fazê-lo em um novo arquivo, chamado `jogo_adivinha_com_tentativas.html`:

```

<script>
var numeroPensado = Math.round(Math.random() * 100);
</script>

```

A partir do `numeroPensado`, precisaremos perguntar qual o número que o usuário gostaria de dar o chute. Vamos estipular também que a quantidade máxima de tentativas será 3 e que, quando passar da terceira tentativa, a repetição deverá se encerrar.

Para isso, vamos declarar uma variável chamada `numeroDaTentativa`, onde guardaremos se é a primeira, segunda ou terceira tentativa. Essa variável começará em 1, já que no começo do programa, estaremos na primeira tentativa. Enquanto esse número for menor ou igual a 3, ou seja, enquanto não passou da terceira tentativa, faremos a repetição da pergunta. Vamos usar o `while` para isso e não vamos nos esquecer que, ao final de cada repetição, deveremos aumentar o número da tentativa (caso contrário, entraremos em repetições infinitas):

```
<script>
var numeroPensado = Math.round(Math.random() * 100);

var numeroDaTentativa = 1;
while(numeroDaTentativa <= 3) {

    numeroDaTentativa++;
}
</script>
```

Precisamos perguntar para o usuário do programa qual é o número que ele acha que o computador pensou. Vamos fazer isso por um `prompt`, que colocaremos dentro do bloco de repetição, o `while`, pois ele fará parte de uma tentativa:

```
<script>
    var numeroPensado = Math.round(Math.random() * 100);

    var numeroDaTentativa = 1;
    while(numeroDaTentativa <= 3) {
        var chute = prompt("Qual você acha que é?");

        numeroDaTentativa++;
    }
</script>
```

Pronto, agora, a cada repetição, perguntamos ao usuário qual é o chute que ele quer dar. Precisamos saber se ele acertou ou não.

Para isso, basta compararmos o número do chute com o `numeroPensado`, e ver se eles são iguais. Mostraremos mensagens para o usuário nos dois casos.

```
<script>
var numeroPensado = Math.round(Math.random() * 100);

var numeroDaTentativa = 1;
while(numeroDaTentativa <= 3) {
    var chute = prompt("Qual você acha que é?");

    if(chute == numeroPensado) {
        alert("Parabéns, você acertou!");
    } else {
        alert("Errou. Será que ainda tem mais tentativas?");
    }
    numeroDaTentativa++;
}
</script>
```

Abra o arquivo `jogo_adivinha_com_tentativas.html` em seu navegador e tente acertar. Se estiver difícil, uma dica é diminuir a possibilidade de números que o computador vai pensar, de 100 para 10, por exemplo. Para isso, basta mudar a seguinte linha:

```
var numeroPensado = Math.round(Math.random() * 10);
```

Após algumas brincadeiras, você notará que há algo errado nesse jogo. Acerte um número na primeira ou na segunda tentativa para ver o que acontece. O jogo mostra a mensagem de parabéns, indicando que você acertou, mas vai perguntar de novo qual é o número que ele pensou. Precisamos corrigir isso, mas como? Onde está o problema no nosso código?

Analisando com um pouco de cuidado, percebemos que uma vez que o chute foi acertado, não precisamos tentar novamente, ou

seja, **as repetições devem acabar imediatamente**. Mas como fazemos para interromper as repetições e sair imediatamente do `while` ?

O Javascript (assim como outras linguagens de programação) possui um comando chamado `break` (quebrar), que significa que, quando ele for chamado, a repetição deverá ser quebrada, ou seja, interrompida. É exatamente isso que precisamos. Quando o chute for acertado, precisamos interromper as repetições, já que não faz sentido perguntar novamente para o usuário.

```
if(chute == numeroPensado) {  
    alert("Parabéns, você acertou!");  
    break;  
} else {  
    alert("Errou. Será que ainda tem mais tentativas?");  
}
```

Com essa alteração, tente acertar um chute na primeira ou segunda tentativa. O programa não vai mais ficar pedindo outro chute. Bem melhor, não?

5.11 REVISANDO NOSSO CÓDIGO: O JOGO DA ADIVINHAÇÃO DOS NÚMEROS

Começamos o nosso jogo, no arquivo `jogo_adivinha_com_tentativas.html`, fazendo com que o computador pensasse em um número aleatório entre 0 e 100.

```
var numeroPensado = Math.round(Math.random() * 100);
```

O próximo passo foi perguntarmos no máximo 3 vezes para o usuário qual o número que ele acha que o computador pensou.

```
var numeroDaTentativa = 1;
```

```
while(numeroDaTentativa <= 3) {
    var chute = prompt("Qual você acha que é?");

    numeroDaTentativa++;
}
```

A cada tentativa, precisamos verificar se ele acertou ou errou. Caso tenha acertado, precisamos sair imediatamente das repetições, o que conseguimos pelo comando `break`.

```
var numeroDaTentativa = 1;
while(numeroDaTentativa <= 3) {
    var chute = prompt("Qual você acha que é?");
    if(chute == numeroPensado) {
        alert("Parabéns, você acertou!");
        break;
    } else {
        alert("Errou. Será que ainda tem mais tentativas?");
    }
    numeroDaTentativa++;
}
```

Vamos melhorar nosso jogo com alguns desafios.

1) Faça com que seu programa mostre quando a tentativa for errada, se o número "chutado" é maior ou menor ao que o computador pensou. Isso vai valer como uma boa dica para o usuário conseguir acertar. Lembre-se de que, para isso, você terá de usar um outro `if`, dentro do `else` que fala que o usuário errou.

2) Faça com que o jogo só termine quando ele acertar. Para isso, a condição da repetição do `while` terá de mudar, para que ele faça a repetição infinita. Após essa mudança, teste seu jogo e veja em quantas tentativas você consegue descobrir o número.

3) O jogo está fácil demais? Aumente a possibilidade de números que o computador pode pensar. 1000 pode ser um bom número para começar. Desafie seus amigos e familiares. Veja quem

consegue ficar com o recorde.

4) Há uma forma de escrever esse nosso jogo sem utilizar o `break` , apenas mudando a condição do `while` , adicionando mais uma cláusula por meio do operador que significa E (o `&&`). Consegue adivinhar como?

5.12 EXERCÍCIOS: TRABALHANDO COM UM LOOP DENTRO DO OUTRO

Loops aparecem muito em nosso código. É até comum que eles apareçam um dentro do outro. Imagine que queremos mostrar a seguinte sequência de asteriscos na tela. São 3 linhas com 10 asteriscos cada:

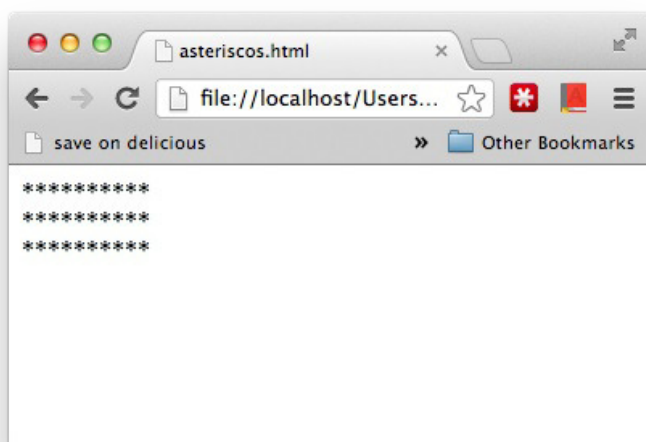


Figura 5.1: Parece fácil fazer esse monte de asteriscos

Você pode pensar em uma solução bem, bem fácil, mas nada desafiadora. Algo como esta:

```
mostra("*****");
mostra("*****");
mostra("*****");
```

Dá para ser mais interessante. Em vez fazer um copiar e colar, podemos usar um `for`, como já visto:

```
for(var linha = 0; linha < 3; linha = linha + 1) {
    mostra("*****");
}
```

Melhorou! Mas queremos ir além. Se precisássemos mudar e imprimir em cada linha 250 asteriscos, teríamos de escrever um monte deles dentro dessa chamada ao `mostra`.

A ideia é, então, usar dois loops. O primeiro fala algo como *"para cada linha, faça"*. O segundo fala *"para cada coluna, imprima um *"*. No final de cada linha, imprimimos um `
`.

O código é curto, mas a primeira vez pode assustar. Crie o `asteriscos.html` e digite as seguintes linhas, dentro das tags de `script`, como sempre:

```
for(var linha = 0; linha < 3; linha = linha + 1) {
    for(var coluna = 0; coluna < 10; coluna = coluna + 1) {
        document.write("*");
    }
    document.write("<br>");
}
```

Repare que declaramos duas variáveis: a variável `linha` para o primeiro loop e a variável `coluna` para o segundo loop. Cada vez que nosso loop das colunas termina, ele pula linha e começa a execução da próxima iteração do loop para começar a impressão

da próxima linha. Exatamente como queríamos. Dizemos aqui que temos um loop encadeado em outro. Em um termo mais técnico, dizemos que temos loops **aninhados**!

Vamos ver outros desafios interessantes.

1) Desenhe um quadrado de 10 linhas por 10 colunas.

2) Agora, utilizando o `prompt`, pergunte ao usuário de que tamanho ele quer o quadrado. Considerando que essa variável chama-se `tamanho`, crie um quadrado de `tamanho` por `tamanho`.

3) Esse aqui é um desafio. Faça um código para obter esse resultado:

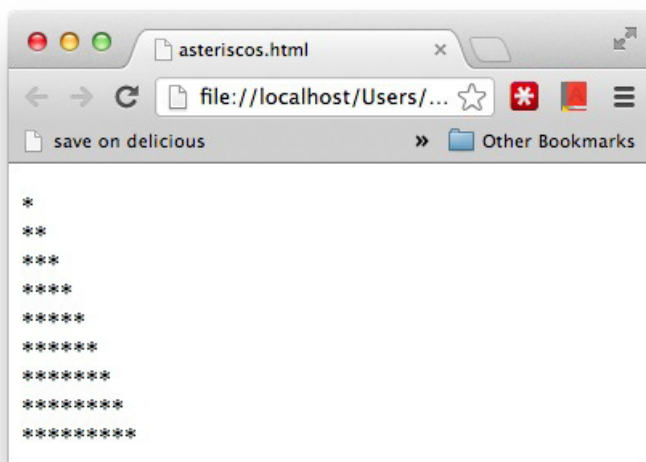


Figura 5.2: Brincando com loops para desenhar

O segredo é pensar assim: na primeira linha queremos apenas uma coluna; na segunda linha queremos duas colunas etc., até a décima linha, que possuirá dez colunas.

Você também precisará de dois loops aninhados. Há duas formas de fazer: uma usando o `break` para parar de imprimir os asteriscos em cada linha em uma determinada condição, ou colocando essa condição direto dentro do `for`.

4) Quer mais? Tente fazer o *V de Vingança*, usando asteriscos e underscores (`_`).

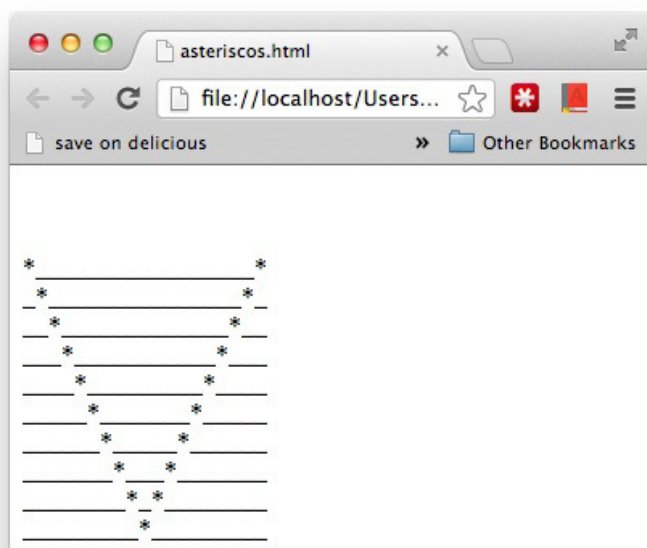


Figura 5.3: Você consegue fazer o V de Vingança?

Analise o padrão. Na primeira linha, o asterisco aparece só na primeira e na última coluna. Na segunda linha, ele aparece na segunda e na penúltima coluna.

No *Apêndice — Gráficos para deixar tudo mais interessante* vamos também desenhar na tela, mas de uma maneira bem mais interessante. Prepare-se!

ARRAYS: TRABALHE COM MUITOS DADOS

6.1 INTEGRE O JAVASCRIPT COM HTML

É comum o JavaScript trabalhar com itens e campos que aparecem no HTML. Crie um novo arquivo, que será uma nova versão do nosso exercício de adivinhação. Ele vai se chamar `adivinha_mais.html` . Como vamos aprender novos conceitos, o jogo vai ser construído aos poucos.

Em primeiro lugar, vamos usar um novo item HTML, chamado `input` . Vamos criar dois deles, um que será um campo texto (`text`) e outro que será um botão (`submit`):

```
<input type="text" />
<input type="submit" value="Compare com o meu segredo!" />
```

Agora abra a página. Veja que temos uma caixa de texto e um botão! Esse é o seu primeiro **formulário**!

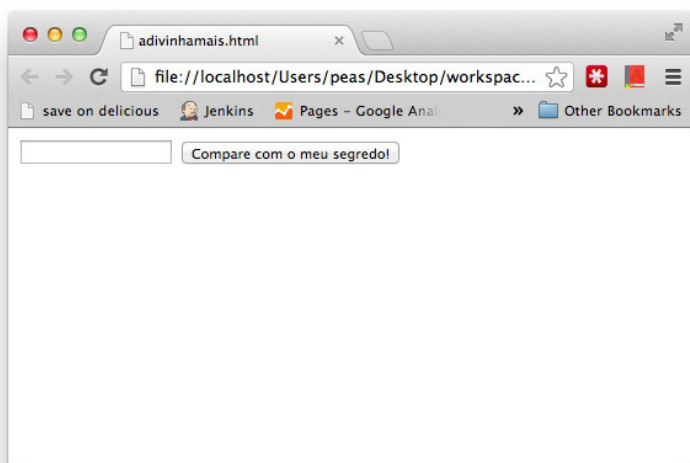


Figura 6.1: Nosso primeiro formulário usando HTML

Clique no botão, o que acontece? Nada!

Sim! É assim que funciona grande parte das páginas da internet em que você coloca dados. Quando você clica em um botão, muitas vezes esses dados são enviados para outros computadores, conhecido como servidores, para que lá sejam armazenados. Mas podemos também trabalhar com esses dados no nosso próprio computador, usando JavaScript.

Vamos fazer com que esse botão veja se o usuário acertar o número que estamos pensando. Antes de continuar, precisamos dar nomes a cada uma das caixinhas que temos. Fazemos isso colocando identificadores (`id`) em cada um dos inputs!

```
<input type="text" id="numero" />
<input type="submit" id="adivinhar"
```

```
value="Compare com o meu segredo!"/>
```

Agora já podemos começar nosso código JavaScript. O primeiro passo que faremos será acessar a caixa de texto que contém o número e dar um `alert`. Eis o código completo:

```
<input type="text" id="numero" />
<input type="submit" id="adivinhar"
      value="Compare com o meu segredo!"/>

<script>
var caixaDoNumero = document.getElementById("numero");
alert(caixaDoNumero.value);
</script>
```

Opa! Aqui aconteceu muita coisa. O `document.getElementById("numero")` é uma forma de fazer com que o JavaScript vasculhe o código HTML e encontre um elemento de `id = numero`. Essa variável ainda não é o valor que digitamos, ela é a caixinha de texto em si! Fazendo `caixaDoNumero.value`, aí sim temos o valor do conteúdo da caixa, isto é, o que foi digitado dentro dela.

Abra o arquivo. O que acontece agora? Sim! O `alert` é executado logo que abrimos a página, sem nos dar tempo de preencher algum valor na caixa, mostrando algo vazio. Não queremos fazer o `alert` logo quando a página for carregada. Queremos chamar o `alert` **quando o botão adivinhar for clicado!**

Para isso, primeiramente vamos criar uma função que queremos que seja executada quando o botão for clicado. Remova o `alert` e adicione:

```
function botaoClicado() {
    alert(caixaDoNumero.value);
}
```


E se abrirmos a página agora, digitarmos o número e clicarmos no botão? Nada novamente!

O que está faltando? Está faltando definir que, quando alguém clicar no botão `adivinhar`, a função `botaoClicado` deve ser chamada! Fazemos isso de uma forma muito simples. Adicione as seguintes linhas:

```
var botaoAdivinhar = document.getElementById("adivinhar");  
botaoAdivinhar.onclick = botaoClicado;
```

Na primeira linha, pegamos o botão e guardamos na variável `botaoAdivinhar`. A segunda linha é a importante. Ela diz que, quando o botão `adivinhar` for clicado (`onclick`), a função `botaoClicado` deve ser chamada.

É muito frequente relacionar um determinado acontecimento à execução de uma função. Esses acontecimentos são comumente chamados de **eventos**, e essas funções a serem executadas em determinados eventos são conhecidas como funções de *callback*. Não se preocupe tanto com nomenclatura, mas ela vai aparecer cada vez com mais frequência nos seus estudos.

E a adivinhação? Bem, ficou faltando guardar o número que pensamos para depois comparar com o digitado. Vamos revisar esse código já colocando essa funcionalidade.

6.2 REVISANDO USO DE HTML E CRIANDO O JOGO

Vamos refazer o nosso código, mas dessa vez com o jogo. Primeiro passo, crie o arquivo `adivinha_mais.html`. Vamos começar declarando dois elementos HTML: um para a caixa que

conterá o número, e outro, o botão que verifica se acertou:

```
<input type="text" id="numero" />
<input type="submit" id="adivinhar"
  value="Compare com o número que estou pensando!" />
```

Agora começamos com JavaScript. Em vez de sortear um número, vamos deixá-lo fixo por uma questão de simplicidade:

```
<script>
var segredo = 8;
```

Pegamos, então, o elemento do HTML por meio daquele código grande, o chamado `document.getElementById` .

```
var caixaDoNumero = document.getElementById("numero");
```

Criamos uma função para verificar se o número digitado (que é `caixaDoNumero.value`) é igual ao `segredo` . Dependendo do que for, imprime o resultado:

```
function botaoClicado() {
  if(segredo == caixaDoNumero.value) {
    alert("Parabéns! Você acertou o número secreto");
  }
  else {
    alert("Infelizmente você errou!");
  }
}
```

Só falta o último passo: falar que, quando o botão `adivinhar` for clicado (o evento de clique), ele deve chamar a função `botaoAdivinhar` :

```
var botaoAdivinhar = document.getElementById("adivinhar");
botaoAdivinhar.onclick = botaoClicado;
</script>
```

Como já vimos, esse tipo de função que recebe a notificação de um evento (por exemplo, o clique do mouse, o teclar de uma letra,

o clique da direita do mouse, o mexer da barra de rolagem etc.) chama-se *callback*.

Seguem dois exercícios simples para você praticar mais:

1) Modifique seu exercício para que o sorteio seja feito realmente. Isto é, não use um número fixo no segredo , e sim randômico.

2) Você reparou que, ao clicar no botão para comparar, o número que você digitou previamente continua lá? Limpe a caixa de texto fazendo `caixaDoNumero.value = ""`; .

3) Aqui também é um bom lugar para fazer testes no console do Chrome, como visto na seção *Utilize o console do Chrome para fazer testes!*.

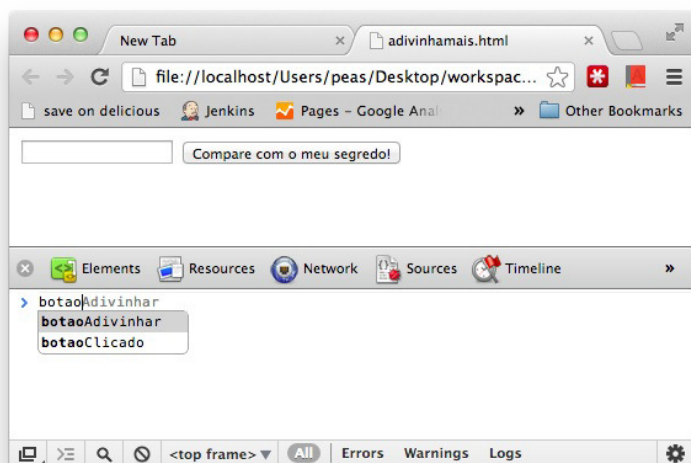


Figura 6.2: Abrindo o console junto com o formulário para fazer testes

Repare que, ao começarmos a digitar `botao`, o Chrome percebe que há duas variáveis com esse nome no nosso JavaScript e já dá as sugestões! É o chamado *code completion* atuando mais uma vez. Se você escolher a nossa caixinha, isso é, o `botaoAdivinhar`, e depois pressionar o ponto e começar a digitar `onclick`, verá que ele possui uma gama de outras funções de callback!

6.3 FACILITE O JOGO DA ADIVINHAÇÃO COLOCANDO MAIS NÚMEROS!

O jogo é um pouco injusto, ainda mais se os números puderem ir de 1 a 60, por exemplo. Algo mais próximo da nossa loteria.

Para ajudar nosso usuário, que tal se tivéssemos mais de 1 segredo? Por exemplo, podemos ter 6 números como o segredo e depois verificamos o chute do usuário. Por exemplo:

```
var segredo1 = 16;  
var segredo2 = 34;  
var segredo3 = 37;  
var segredo4 = 42;  
var segredo5 = 50;  
var segredo6 = 58;
```

Hum, algo não está tão bonito. Para comparar com o chute, precisaremos ter um `if` gigante, verificando cada um deles e fazendo o **ou** (com `||`). Pior: e se precisarmos ter mais segredos? Precisaríamos criar mais variáveis e mexer novamente no nosso `if`.

Muito seria facilitado se houvesse uma maneira fácil de guardar múltiplas variáveis, sem saber exatamente quantas. E há! Vamos criar uma variável `segredos` que guarda vários números:

```
var segredos = [16, 34, 37, 42, 50, 58];
```

A sintaxe é nova para nós. Nunca havíamos visto os colchetes. Neste caso, eles estão criando uma **array** para nós. Sim, tem um nome complicado e você precisa se acostumar a isso.

Abra o console do Chrome e faça alguns testes. O primeiro de todos é criar a variável `segredos` e imprimir seu valor.

```
var segredos = [16, 34, 37, 42, 50, 58];  
console.log(segredos);
```

Repare a saída! Nada surpreendente:

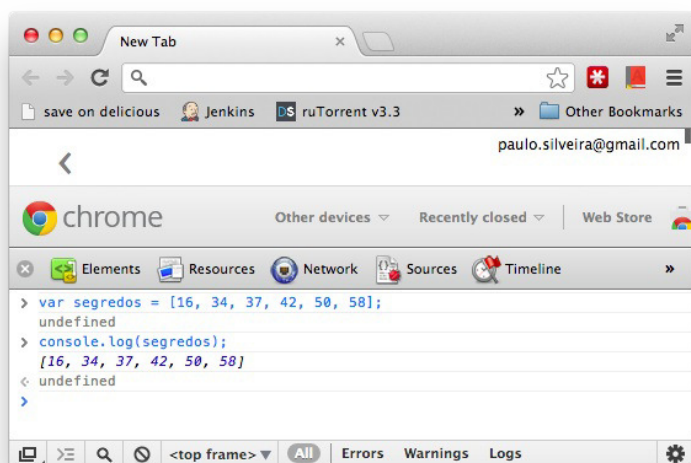


Figura 6.3: Mostrando a nossa array

Isso não parece muito útil. Como pegar o primeiro elemento, isto é, o primeiro número dessa série de segredos? Usamos os colchetes novamente, para indicar que queremos pegar um

elemento, e não a array inteira:

```
console.log(segredos[1]);
```

Opa! A saída deu 34! Por quê? O JavaScript, assim como diversas outras linguagens, começa a contar as posições da array a partir do 0. Então nesse caso temos do 0 ao 5. Para pegar o 16, faríamos:

```
console.log(segredos[0]);
```

Se quiser imprimir todos os números, basta fazer um `for` que vá de 0 a 5:

```
for(var i = 0; i < 6; i = i + 1) {  
    console.log(segredos[i]);  
}
```

O `segredos[i]` acessa a array `segredos` na *i-ésima* posição! Esse número que vai dentro de `[]` chamamos também de índice!

As arrays possuem algumas facilidades. Por exemplo, podemos saber qual é o seu tamanho:

```
console.log(segredos.length);
```

Por esse motivo, é possível escrever aquele `for` sem fixar o número 6, substituindo-o por `segredos.length`:

```
for(var i = 0; i < segredos.length; i = i + 1) {  
    console.log(segredos[i]);  
}
```

Agora está fácil! Crie o seu arquivo `loteria.html`, e vamos declarar o campo de texto e o botão:

```
<input type="text" id="numero" />  
<input type="submit" id="adivinhar"  
    value="Compare com o meu segredo!" />
```

Agora declaramos, no JavaScript, nossos segredos:

```
var segredos = [16, 34, 37, 42, 50, 58];
```

E, como já vimos, pegamos a caixa em que o usuário digita seu chute:

```
var caixaDoNumero = document.getElementById("numero");
```

Vamos preparar a função que será chamada (*callback*) quando o evento de clicar no botão ocorrer. O que faremos dentro dela? Fazemos um `for`, percorrendo todos os números, para ver se ele é o `caixaDoNumero.value`. Se for, imprimimos que ele ganhou. Repare:

```
function botaoClicado() {  
    for(var i = 0; i < segredos.length; i = i + 1) {  
        if(segredos[i] == caixaDoNumero.value) {  
            alert("Parabéns! Você acertou um dos "  
                + " números secretos");  
        }  
    }  
}
```

O código ainda não está tão bom. O que acontece no caso de ele errar? Nenhuma mensagem está sendo apresentada. Podemos tentar resolver esse problema adicionando um novo `alert`, no fim do `for`:

```
function botaoClicado() {  
    for(var i = 0; i < segredos.length; i = i + 1) {  
        if(segredos[i] == caixaDoNumero.value) {  
            alert("Parabéns! Você acertou um dos "  
                + " números secretos");  
        }  
    }  
    alert("Infelizmente você errou!");  
}
```

Pronto, isso funciona no caso de você errar os segredos.

Apenas *"Infelizmente você errou!"* será mostrado.

Mas imagine que o usuário tenha acertado o número, no fim do `for` o seu programa ainda vai mostrar o `alert` de que ele errou! Queremos que essa função não mostre o segundo `alert` no caso de o número estar certo.

Podemos resolver isso com o uso do `return`. O `return` pode ser usado em uma função que não retorna valor algum, para que a execução da função pare naquele momento e retorne para quem a chamou:

```
function botaoClicado() {  
    for(var i = 0; i < segredos.length; i = i + 1) {  
        if(segredos[i] == caixaDoNumero.value) {  
            alert("Parabéns! Você acertou um dos "  
                + " números secretos");  
            return;  
        }  
    }  
    alert("Infelizmente você errou!");  
}
```

Repare que há outras formas de resolver esse problema sem o uso do `return`. Você poderia ter criado uma `var` `achou` e, dentro do `if` que verifica se ele acertou, fazer `achou = true`, além de mostrar a mensagem de acerto. Fora do `for`, antes de mostrar a mensagem de erro, você deveria verificar se `achou != true`.

Qual é o próximo passo? Agora basta registrarmos que essa função vai realmente ser o callback do evento de clicar no botão, como também já aprendemos:

```
var botaoAdivinhar = document.getElementById("adivinhar");  
botaoAdivinhar.onclick = botaoClicado;
```



```
</script>
```

Abra a sua loteria e tente acertar um dos números.

1) Remova o `break` do seu código e veja o que acontece. Por que é necessário ter o `break` aí?

2) Caso o usuário acerte o número, diga qual é a posição do número que ele acertou, por exemplo: *"Parabéns, você acertou o 5o número!"*. Isto é, o índice da array, só tome cuidado com o zero.

6.4 EVITE OS NÚMERO REPETIDOS NO BINGO

Já vimos como é o trabalho básico em uma array, mas como modificar seus valores? Repare na array de `segredos` :

```
var segredos = [16, 34, 37, 42, 50, 58];  
console.log(segredos[0]);
```

Assim como podemos pegar o primeiro elemento por meio de `segredos[0]` , podemos modificá-lo:

```
segredos[0] = 20;  
console.log(segredos[0]);
```

É uma atribuição como qualquer outra que já fizemos.

Uma array também nos fornece acesso a uma série de funções para poder mexer em seu conteúdo. O mais importante é a `push` , que adiciona um elemento no final da array. Observe:

```
segredos.push(60);  
console.log(segredos[6]);  
console.log(segredos.length);
```

Agora nosso array tem tamanho 7! O `push` colocou o `60`

após o 58 , fazendo com que segredos agora contenha [16, 34, 37, 42, 50, 58, 60] .

A sintaxe realmente é um pouco diferente, quando fazemos `segredos.push(...)` . Mas repare que já havíamos feito isso antes, com o `console.log(...)` , o `document.write(...)` . Ainda é cedo para você estudar o que há por trás de tudo isso, mas a verdade é que tanto `segredos` quanto `console` e `document` são variáveis que possuem valores e também outras funções – são os conhecidos **objetos**. Falamos um pouco mais sobre isso no apêndice, mas não se preocupe.

Vamos aplicar esse novo conhecimento. Considere que temos um formulário onde será adicionado cada novo número do sorteio do Bingo. Queremos verificar se todo número digitado é diferente dos números que já saíram, o que indicaria um problema ou fraude! Isto é, se os números 34, 43 e 55 já saíram, 43 não pode sair de novo!

Crie o arquivo `bingo.html` e adicione o HTML para o campo de texto do número e o botão de adicionar:

```
<input type="text" id="numero" />
<input type="submit" value="Adicione e verifique no Bingo!"
      id="verificar"/>
```

Agora, no nosso JavaScript, vamos declarar uma variável com todos os números do Bingo que já foram sorteados. Opa! Mas e se nenhum foi sorteado ainda? Aí usamos simplesmente `[]` , que é uma array com zero elementos:

```
<script>
var sorteados = [];
```

Agora queremos verificar toda vez que alguém clicar no botão de adicionar, se ele já não existe na nossa array de sorteados. Caso exista, exibe-se uma mensagem de alerta; caso contrário, adiciona-se o número na nossa lista de sorteado. Poxa! É muita coisa. Vamos começar simplesmente adicionando na array e fazendo um `console.log` :

```
function adicionarSorteado() {  
    var numero = document.getElementById("numero").value;  
    sorteados.push(numero);  
    console.log(sorteados);  
}
```

Repare que aqui fizemos `document.getElementById("numero").value` tudo em uma linha só, em vez de primeiro guardar `document.getElementById("numero")` em uma variável temporária e depois pegar o `.value` dela. Desde que o código não fique demasiadamente comprido nem complicado, é comum evitar algumas variáveis temporárias.

Ainda está faltando dizer que essa função deverá ser executada quando ocorrer o evento de clicar (`onclick`) do nosso botão `verificar` . Isto é, dizer que essa será nossa função de *callback* do clicar desse botão, como já vimos antes:

```
var botao = document.getElementById("verificar");  
botao.onclick = adicionarSorteado;  
</script>
```

Vá se acostumando aos callbacks, eles vão aparecer com muita frequência no seu futuro como programador.

Agora pode abrir o seu `bingo.html` . Se você adicionar um número repetido, vai ver que ele aparecerá listado duas vezes no

seu console.

Para evitar o número repetido, basta alterarmos a nossa função para procurar se o novo número já não está antes. Faremos um `for` que percorre cada um dos elementos:

```
function adicionarSorteado() {  
    var numero = document.getElementById("numero").value;  
    for(var i = 0; i < sorteados.length; i = i + 1) {  
        if(sorteados[i] == numero) {  
            alert("Número já sorteado!");  
            return;  
        }  
    }  
    sorteados.push(numero);  
    console.log(sorteados);  
}
```

Repare que, se encontramos o número e exibirmos a mensagem de alerta, fazemos um `return`. Como vimos, o comando `return` pode ser usado simplesmente para parar a execução de uma função que não retorna nada. Se você remover esse `return`, perceba que ele vai acabar adicionando o número ao nosso array mesmo se já repetido. Se fosse um `break`, funcionaria?

1) Este é um exercício difícil! Você deve sortear 6 números de 1 a 60 e guardar em uma array. Depois peça para o usuário dizer 6 números, um de cada vez (pode usar um `prompt` ou um `input`) e guarde-os em uma outra array. Aí diga quantos números ele acertou. Sim, é a loteria certinha desta vez!

Tente acertar os 6 números. Depois faça as contas de quantas chances você tem de acertar todos os 6. Você nunca mais vai jogar na loteira!

APÊNDICE — GRÁFICOS PARA DEIXAR TUDO MAIS INTERESSANTE

Ter uma resposta visual ao nosso trabalho pode ser bastante recompensador. Vamos aprender a usar algumas funções do JavaScript para desenhar em uma tela (em um *canvas*) do HTML, para praticar o que aprendemos e ir além. Impressiona seus amigos com imagens interessantes e mostre para nós o que conseguiu fazer! Lembre-se de participar da nossa lista de discussão.

Haverá um pouco de matemática, mas não se assuste. Será bem simples e você não precisa se preocupar com detalhes. O importante é sempre enxergar como as funções, variáveis, `if` e `for` estão trabalhando juntos.

Se você está lendo a versão impressa, não terá as cores exatas para praticar os exercícios, mas não é um problema, você pode escolher a que achar adequada.

7.1 DESENHE LINHAS E FIGURAS

Vamos definir uma área da nossa página para que possamos

desenhar nela via JavaScript. Crie o arquivo `canvas.html` e declare a área que usaremos para pintar da seguinte maneira:

```
<canvas id="tela" width="600" height="400"></canvas>
```

Isso mesmo, apenas uma única linha, sem a tag `script`. Abra seu HTML no Chrome, qual foi o resultado? Isso mesmo, uma tela de 600x400 branca em um fundo... também branco! Não dá para perceber nada.

Vamos desenhar um retângulo verde (*green*) à esquerda, no ponto 0,0 (canto superior esquerdo) de tamanho 200,400 (indo até o ponto inferior, um terço do valor máximo possível, que é 600). Aqui estamos trabalhando em um plano de duas dimensões, onde a vírgula separa a posição X da posição Y.

Para fazer isso, teremos de usar funções do JavaScript que ainda não conhecemos muito. Não se preocupe, você vai aprender bastante sobre elas mais para a frente. É natural (apesar de não ser o ideal) testar novas funções e comandos que ainda não dominamos.

Adicione as seguintes linhas logo abaixo à declaração do seu canvas.

```
<script>
var tela = document.getElementById("tela");
var c = tela.getContext("2d");

c.fillStyle="green";
c.fillRect(0, 0, 200, 400);
</script>
```

Abra seu HTML, temos a seguinte figura:

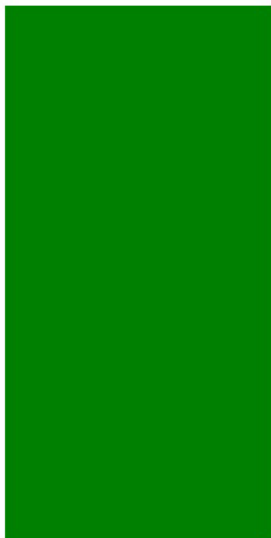


Figura 7.1: Uma simples coluna verde

As duas primeiras não nos importam por enquanto. Você pode imaginar que a variável `c` é como se fosse um pincel. Podemos trocar o estilo do pincel (via variável `fillStyle`) e desenhar diversas formas (no caso, um retângulo, chamando a função `fillRect`).

A função `fillRect` recebe quatro números para pintar um retângulo. Os dois primeiros são a posição `X,Y`, e os outros dois são a largura e a altura do retângulo. Nesse caso, estamos pintando um retângulo na posição `0,0` de 200 de largura por 400 de altura.

Como ter uma ideia melhor de onde isso vai parar na tela? O ponto `0,0` é o canto superior esquerdo. O ponto `600,400` é o inferior direito. Repare em alguns outros pontos espalhados nas nossas coordenadas `X` e `Y`:

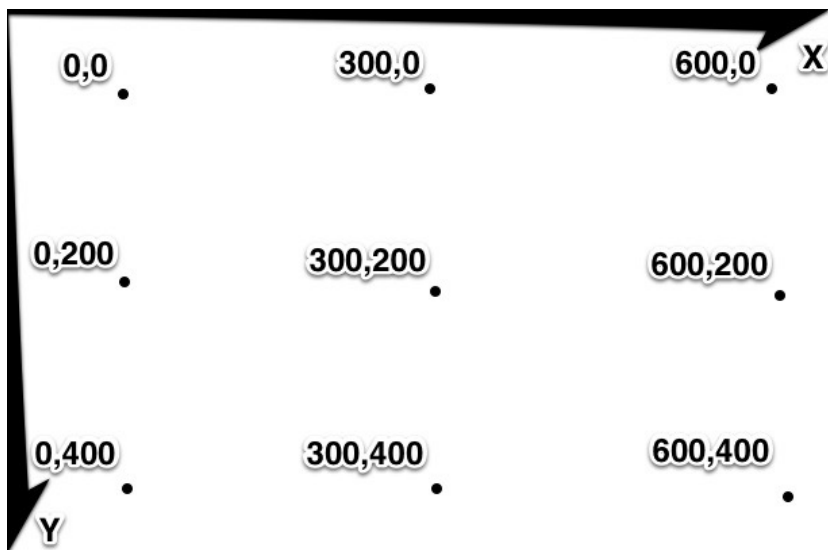


Figura 7.2: Coordenadas do nosso plano. Utilize essa figura como referência para saber onde traçar seus objetos!

Vamos adicionar mais um retângulo, agora em vermelho, começando na posição $400,0$. Adicione logo após o seu retângulo verde:

```
c.fillStyle="red";
c.fillRect(400, 0, 200, 400);
```

O resultado acaba em pizza! Temos a bandeira da Itália.

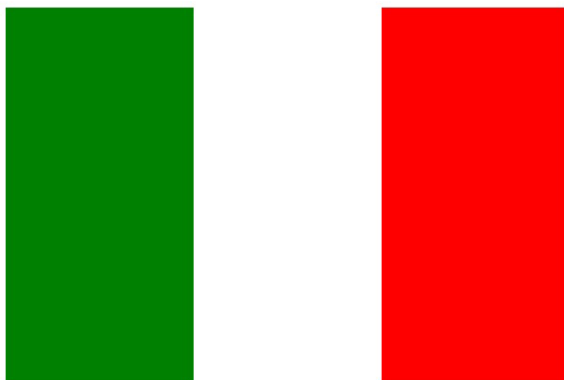


Figura 7.3: Bandeira da Itália

Experimente trocar as cores, como `gray` , `red` , `green` , `blue` , `black` etc.

7.2 CRIANDO TODO TIPO DE IMAGEM

A função `fillRect` é simples e direta. Recebe a coordenada de início, além da altura e largura. Desenhar outras figuras dá um pouco mais de trabalho. Você precisa ir traçando o caminho, para depois chamar a função que preenche o que foi traçado.

Por exemplo, queremos fazer um triângulo no meio da bandeira, em cinza (`gray`), como este:



Figura 7.4: Agora com um triângulo

Para isso, precisamos começar a traçar três lados. Pegamos nosso pincel começando do ponto `300, 200` , que representa o meio da bandeira. Traçamos desse ponto até o inferior esquerdo, que é `200, 400` . Depois traçamos a base do triângulo, movendo o pincel até o ponto `400, 400` . Por último, fechamos o nosso polígono (no caso um triângulo), retornando ao ponto inicial do traçado. Além disso, pedimos para que todo o polígono seja preenchido.

O código fica dessa forma:

```
c.fillStyle="gray";  
c.beginPath();  
c.moveTo(300, 200);  
c.lineTo(200, 400);  
c.lineTo(400, 400);  
c.fill();
```

O código segue fielmente nossa descrição. Primeiro, movemos o pincel para o centro da imagem, para depois traçar os dois lados. O terceiro lado fica implícito quando fazemos a chamada ao `fill` . Podemos ver isso na figura:

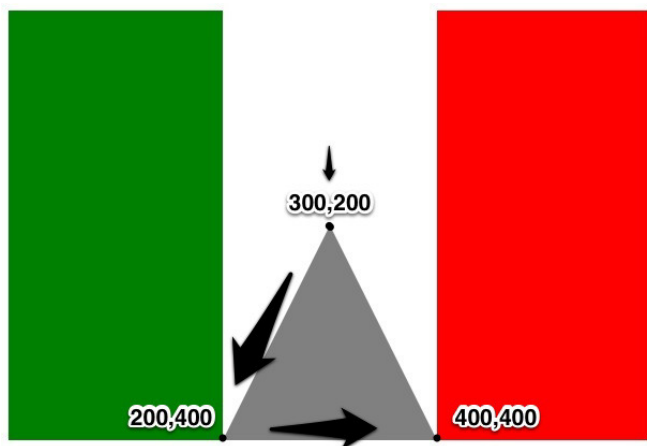


Figura 7.5: Coordenadas do triângulo

Podemos fazer qualquer figura que tenha lados usando essa fórmula, não apenas triângulos. Mas e se precisarmos de algo arredondado? Vamos colocar uma circunferência azul no meio da nossa imagem, para ficar assim:

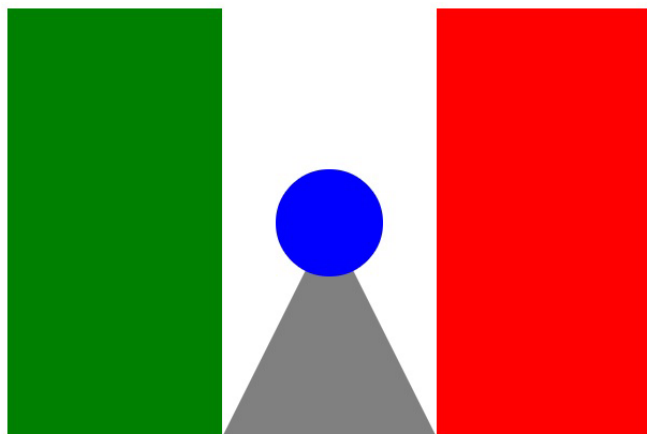


Figura 7.6: Uma circunferência

Existe a função `arc` , que precisa de muitas informações. São 5: as coordenadas X e Y, o raio, o ângulo inicial e o ângulo final, ambos em radianos. Nada simples. Ele precisa dos ângulos iniciais e finais, porque eles também servem para desenhar apenas um pedaço do círculo. Vamos usar de 0 até duas vezes 3.14, que é o valor do PI, representando a circunferência inteira:

```
c.fillStyle="blue";  
c.beginPath();  
c.arc(300, 200, 50, 0, 2*3.14);  
c.fill();
```

O valor de `3.14` é o PI. Estamos usando apenas duas casas decimais. Esse valor de PI pode ser substituído por `Math.PI` , que é uma variável já existente no JavaScript. Faça essa substituição. Valores fixos em variáveis como essa são frequentemente referenciadas como **constantes**.

Vimos muitas funções neste capítulo. Como seria possível se lembrar de todas elas? Ou saber qual é a ordem que devo passar os parâmetros? A linha `c.arc(300, 200, 50, 0, 2*3.14)` parece bastante complicada.

7.3 NÃO VOU CONSEGUIR LEMBRAR DE TUDO ISSO! APIS E BIBLIOTECAS

O nosso código não parece tão difícil, mas como é que vamos saber a existência dessas funções de desenhar no canvas? Se você reparar, vimos mais de 5 funções diferentes, entre elas `fillRect` , `beginPath` , `moveTo` , `lineTo` e `fill` . Além disso, usamos a variável `fillStyle` . É muita coisa para memorizar!

E você nem deve se preocupar tanto em memorizá-las. Claro,

se você usá-las com frequência, isso se tornará fácil. Pense no nosso `document.write`, no `alert` e outras funções que não apareceram apenas uma vez durante nosso aprendizado.

Mas e essas do canvas? E esse tal "contexto 2d", que apareceu no `tela.getContext("2d")`; e mal falamos dele?

Acontece que sempre temos uma documentação das principais funções do JavaScript. Isso também aparece em diversas outras linguagens. Por exemplo, todas as funções e variáveis que usamos para trabalhar com o canvas podem ser encontradas aqui, bem documentadas:

<https://developer.mozilla.org/en-US/docs/DOM/CanvasRenderingContext2D>

O site da Mozilla, responsável pelo navegador FireFox, possui uma das documentações mais completas e fáceis de ler. Nesse site, você pode ver uma documentação como da figura a seguir.

CanvasRenderingContext2D

The 2D rendering context for the drawing surface of a `<canvas>` element. To get this object, call `getContext()` on a `<canvas>`, supplying "2d" as the argument:

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
```

Once you have the 2D rendering context for a canvas, you can draw within it. For example:

```
ctx.fillStyle = "rgb(200,0,0)";
ctx.fillRect(10, 10, 55, 50);
```

See the [Canvas tutorial](#) for more information, examples, and resources.

Method overview

```
void arc(in float x, in float y, in float radius, in float startAngle, in float endAngle, in boolean anticlockwise Optional);
```

```
void arcTo(in float x1, in float y1, in float x2, in float y2, in float radius);
```

```
void beginPath();
```

Figura 7.7: Documentação no site da Mozilla

Repare que ele dá uma breve descrição e depois parte para cada função, cada variável relacionadas ao assunto. Sim, você vai precisar encarar o inglês técnico, mesmo que básico. Em português, documentações desse tipo, que chamamos de referência, são bastante escassas.

A documentação da Mozilla pode conter informações específicas que só funcionam no navegador deles. Para uma documentação oficial, que deveria funcionar em todos os navegadores, porém menos completa, há este site:

<http://docs.webplatform.org/wiki/javascript>

Muitas vezes temos um conjunto de funções que trabalham com um objetivo em comum. Chamamos esse conjunto de **biblioteca**. É frequente alguém se referenciar à *biblioteca do JavaScript que trabalha com gráficos, biblioteca para validação de CPF, biblioteca para fazer drag and drop*. Veremos mais à frente que há casos em que isso pode aparecer com outros nomes mais estranhos ainda, como API ou ainda de objeto.

Procure você mesmo a documentação da função `fillRect` nessa página. Clique sobre ela e você terá mais detalhes.

Também podemos encontrar formatos mais simpáticos dessas documentações, criados por outros desenvolvedores. Este é um bom exemplo de post que um blogueiro criou para divulgar seu PDF que contém uma documentação alternativa:

<http://blog.nihilogic.dk/2009/02/html5-canvas-cheat-sheet.html>

Ele apresenta um resumo das funções que trabalham com o canvas. Segue um pedaço dessa documentação onde podemos ver algumas das que já conhecemos, bem resumidas, explicando quais são os parâmetros que elas trabalham:

CanvasGradient interface	
void	addColorStop(float offset, string color)
CanvasPattern interface	
No attributes or methods.	
Paths	
Methods	
Return	Name
void	beginPath()
void	closePath()
void	fill()
void	stroke()
void	clip()
void	moveTo(float x, float y)
void	lineTo(float x, float y)
void	quadraticCurveTo(float cp _x , float cp _y , float x, float y)
void	bezierCurveTo(float cp _{1x} , float cp _{1y} , float cp _{2x} , float cp _{2y} , float x, float y)
void	arcTo(float x ₁ , float y ₁ , float x ₂ , float y ₂ , float radius)
void	arc(float x, float y, float radius, float startAngle, float endAngle, boolean anticlockwise)
void	rect(float x, float y, float w, float h)
boolean	isPointInPath(float x, float y)

Rectangles	
Methods	
Return	Name
void	clearRect(float x, float y, float w, float h)
void	fillRect(float x, float y, float w, float h)
void	strokeRect(float x, float y, float w, float h)

Pixel manipulation	
Methods	
Return	Name
ImageData	createImageData(float sw, float sh)
ImageData	createImageData(ImageData)
ImageData	getImageData(float sx, float sy, float sw, float sh)
void	putImageData(ImageData imageData, float dx, float dy, [Optional] float dirtyX, float dirtyY, float dirtyWidth, float dirtyHeight)

ImageData interface	
width	unsigned long [readonly]
height	unsigned long [readonly]
data	CanvasPixelArray [readonly]

CanvasPixelArray interface	
length	unsigned long [readonly]

Figura 7.8: Uma documentação não oficial, mas que pode ser bem útil. Há várias como essa na internet

Vamos conhecer outras funções relacionadas ao canvas ainda neste capítulo. Lembre-se de que você pode e deve explorar novas opções através da documentação. O inglês não deve ser barreira: ele exigirá pouco, mas é fundamental conhecer o mínimo.

7.4 REVISE SEUS PRIMEIROS PASSOS COM O CANVAS

Vamos revisar o que fizemos para criar a bandeira da Itália e as nossas outras figuras. Não se assuste com um pouco de matemática e as coordenadas. A prática vai tornar tudo bem mais fácil.

Primeiro, criamos o arquivo `canvas.html` e declaramos nosso canvas:

```
<canvas id="tela" width="600" height="400"></canvas>
```

Logo depois, começamos a colocar o código JavaScript. As duas primeiras linhas são para adquirirmos o tal do contexto 2d, que vai servir como pincel. São duas linhas de código que ainda não dominamos:

```
<script>
var tela = document.getElementById("tela");
var c = tela.getContext("2d");
```

Agora desenhamos o primeiro retângulo, que será verde:

```
c.fillStyle="green";
c.fillRect(0, 0, 200, 400);
```

O segundo retângulo será vermelho, de mesma largura e altura, porém iniciará na posição `400,0` :

```
c.fillStyle="red";
c.fillRect(400, 0, 200, 400);
```

Vamos ao triângulo cinza. Aqui usamos a função que começa o traçado, a `beginPath` . Movemos o pincel com `moveTo` e traçamos com `lineTo` . Por último, preenchemos o miolo com a função `fill` :

```
c.fillStyle="gray";
c.beginPath();
c.moveTo(300, 200);
c.lineTo(200, 400);
c.lineTo(400, 400);
c.fill();
```

Por último, desenhamos nosso círculo bem no meio da bandeira, em azul, lembrando também de fechar a tag de `script` :

```
c.fillStyle="blue";  
c.beginPath();  
c.arc(300, 200, 50, 0, 2*3.14);  
c.fill();  
</script>
```

Vamos praticar mais o conhecimento das coordenadas e testar outros desenhos e formas através das bandeiras.

1) Inicialmente, vamos desenhar duas bandeiras simples: a primeira da Colômbia e a segunda de Madagascar:



Figura 7.9: Bandeira da Colombia

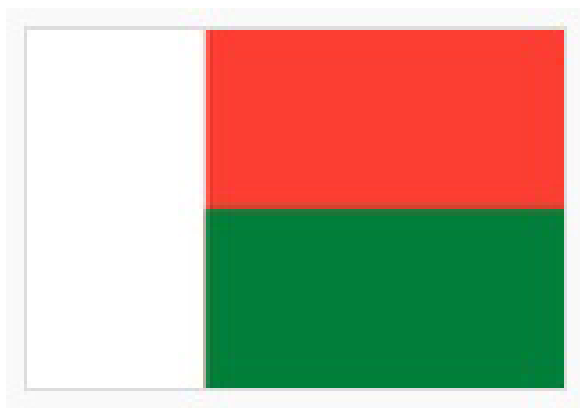


Figura 7.10: Bandeira de Madagascar

2) Agora duas que possuem circunferências e você precisará usar o arco, do Laos e do Niger:

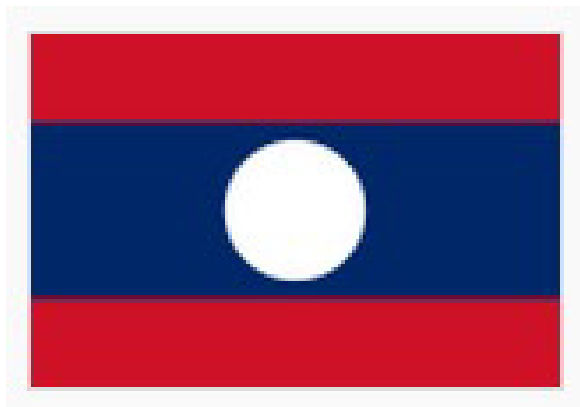


Figura 7.11: Bandeira do Laos



Figura 7.12: Bandeira da Niger

3) Uma bastante interessante é da Noruega. Você pode usar retângulos parar fazer as cruzes.

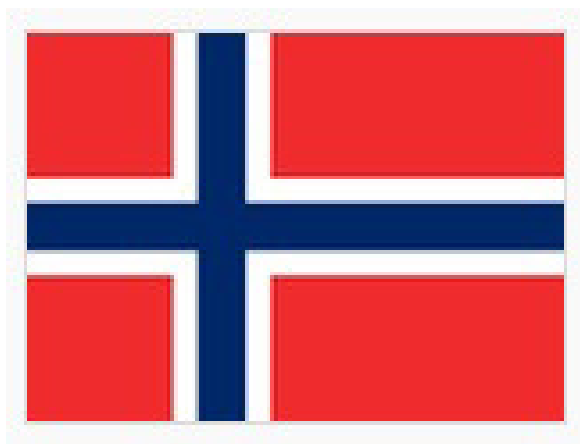


Figura 7.13: Bandeira da Noruega

Repare que há várias formas de criá-la: você pode desenhar um retângulo vermelho grande e uma cruz branca dentro através de dois retângulos, para então fazer as azuis; ou ainda criar quatro retângulos vermelhos e depois somente a cruz azul, utilizando o branco que já há no canvas.

4) Desafio: você agora tem condições de criar a bandeira do Brasil, em especial as partes verde, amarelo e azul. A faixa branca também é possível, mas bastante complicada!

Querendo algo mais próximo para usar em um jogo? Que tal o Pacman? O código pode ser encontrado aqui:

<http://jsfiddle.net/moreira/qRS5c/>

7.5 CANSEI DE REPETIR CÓDIGO! FUNÇÕES NOVAMENTE

Vamos desenhar vários quadrados no nosso canvas e aprimorar nossas figuras. Mas será que precisaremos ficar repetindo tantas vezes os códigos complicados com arcos, quadrados, estilos e cores?

Podemos usar o que já aprendemos e criar uma função para evitar a repetição de código. Faremos isso em um novo arquivo, o `canvas_loop.html`. Dentro dele, vamos declarar o canvas e preparar nosso código para que possamos começar a desenhar os quadrados de 50 por 50. Precisamos de uma função que recebe a posição `x` e `y` onde ele vai ser desenhado. Ela se chamará `desenhaQuadradoVerde`:

```
<canvas id="tela" width="600" height="400"></canvas>
```

```

<script>
function desenhaQuadradoVerde(x,y) {
    var tela = document.getElementById("tela");
    var c = tela.getContext("2d");

    c.fillStyle="green";
    c.fillRect(x, y, 50, 50);
}

desenhaQuadradoVerde(0,0);
</script>

```

Na última linha, estamos chamando a função para testar o que programamos.

Salve o arquivo e abra-o com o Chrome. O resultado é apenas um pequeno quadrado verde.



Figura 7.14: Um quadrado verde... através da nossa função!

7.6 LOOPS E FUNÇÕES PARA NOS AJUDAR

Com o auxílio de um loop, podemos fazer muitas outras imagens. Vamos desenhar diversos quadrados de tamanho 50 por 50, começando da posição zero do eixo X, indo até a posição 600.

Edite seu arquivo `canvas_loops.html` e vamos usar o `while` para fazer isso. Criaremos uma variável `x` que vai aumentando de 50 em 50, até chegar em 600. Delete a chamada para `desenhaQuadradoVerde(0,0)` e adicione as seguintes.

```

var x = 0;
while(x < 600) {
    desenhaQuadradoVerde(x, 0);
    x = x + 50;
}

```

```
}
```

Salve e abra o arquivo no Chrome. Qual é o resultado? Uma linha comprida verde!



Figura 7.15: Uma linha verde. Na verdade, uma série de quadrados verde

Por quê? Pois a cada 50 pontos nós desenhamos um novo quadrado. Não há divisão entre eles, pois eles são verdes em todos seus pontos. Para ficar mais visual, podemos alterar a função `desenhaQuadradoVerde` para que o estilo do traçado da borda (`strokeStyle`) seja preta, adicionando duas linhas:

```
function desenhaQuadradoVerde(x,y) {  
    var tela = document.getElementById("tela");  
    var c = tela.getContext("2d");  
  
    c.fillStyle="green";  
    c.fillRect(x, y, 50, 50);  
  
    c.strokeStyle="black";  
    c.strokeRect(x, y, 50, 50);  
}
```

O resultado agora será a mesma linha, porém com claras divisões de cada um dos quadrados:



Figura 7.16: Marcando bem os quadrados

Vamos trocar esse nosso `while` pelo `for`, que também é um comando de loop, porém mais sucinto. Ele tem uma sintaxe mais

estranha, mas é importante se habituar, ele aparece com bastante frequência! Delete o `while` e faça assim:

```
for(var x = 0; x < 600; x = x + 50) {  
    desenhaQuadradoVerde(x, 0);  
}
```

Percebe como ele fica mais curto? A variável `x` não precisa mais ser declarada antes. E o incremento de 50 em 50 fica definido dentro do próprio comando.

Vamos incrementar um pouco. Como fazemos para desenhar uma linha dessas, só que vermelha, logo abaixo da linha verde? Inicialmente precisamos criar a função que desenha quadrados vermelhos, logo abaixo de onde criamos para os verdes:

```
function desenhaQuadradoVermelho(x,y) {  
    var tela = document.getElementById("tela");  
    var c = tela.getContext("2d");  
  
    c.fillStyle="red";  
    c.fillRect(x, y, 50, 50);  
  
    c.strokeStyle="black";  
    c.strokeRect(x, y, 50, 50);  
}
```

Agora, além do `for` que faz o quadrado verde, podemos criar um `for` que trabalhe para, na linha 50 :

```
for(var x = 0; x < 600; x = x + 50) {  
    desenhaQuadradoVerde(x, 0);  
}  
for(var x = 0; x < 600; x = x + 50) {  
    desenhaQuadradoVermelho(x, 50);  
}
```

E o resultado que temos:



Figura 7.17: Outra linha de quadrados, de outra cor

Mas perceba que os dois `for`s que criamos percorrem exatamente os mesmos valores de `x`, isso é, `0`, `50`, `100`, `150`, ..., `550`. Poderíamos, então, desenhar os dois quadrados dentro de um único `for`:

```
for(var x = 0; x < 600; x = x + 50) {
    desenhaQuadradoVerde(x, 0);
    desenhaQuadradoVermelho(x, 50);
}
```

Vamos incrementar nossos testes artísticos.

1) Crie uma função para desenhar bolas azuis, que receba a posição `x` e `y` e desenhe uma circunferência de 25 de raio. Altere o laço para que, a cada iteração, você invoque essa função para cada `x` na linha 100. Isto é, você vai fazer algo como `desenhaBolaAzul(x, 100);` dentro do `for`.

O resultado deve ser algo parecido com:

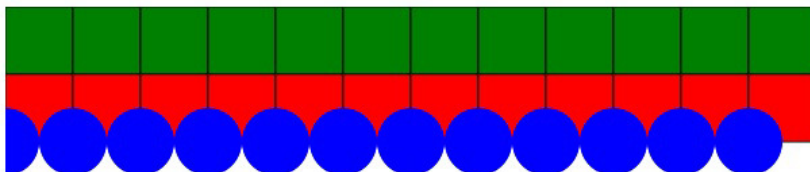


Figura 7.18: Resultado que queremos obter

2) Como arrumar o código para que seja gerada a figura a seguir?

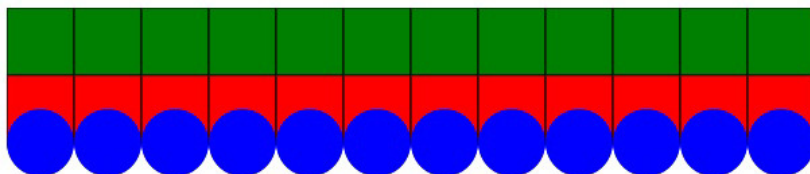


Figura 7.19: E como fazer isso?

3) Você já reparou que as funções `desenhaQuadradoVermelho` e `desenhaQuadradoVerde` são muito parecidas? O ideal seria não ter código repetido nem mesmo dentro de funções. A solução aqui seria criar uma terceira função, a `desenhaQuadrado` que, além de `x` e `y`, recebe uma `cor`. Aí você pode apagar quase todo o código de dentro de `desenhaQuadradoVermelho` e `desenhaQuadradoVerde` para que elas chamem a função `desenhaQuadrado`, passando a respectiva cor. Você encara esse desafio? O resultado será o mesmo, porém o código estará muito mais elegante.

4) Lembra dos nossos exercícios de loop, nos quais desenhamos asteriscos formando um retângulo, um triângulo e até o V de Vingança? Eles estão na seção [ref-label exercicios-asteriscos]. Refaça-os utilizando círculos no canvas e impressione seus amigos!

7.7 PARA SABER MAIS: PASSE UMA FUNÇÃO PARA UMA... FUNÇÃO!

Esse é um tópico opcional para você que está começando,

muito desafiador. Se você está seguindo com facilidade os exemplos e gosta de um pouco de matemática, mais do que já viu nesse capítulo, recomendamos encarar já!

Lembra quando você aprendeu a desenhar funções matemáticas no chamado plano cartesiano? Agora fica fácil fazer isso com um `for` e o nosso `canvas`.

Como desenhar a função matemática $f(x) = x * x$? Podemos começar, de 0 a 600 (nossas coordenadas no eixo x) e fazer a conta para cada ponto, achando a coordenada y . Por exemplo, para o $x = 2$, temos que $f(x) = 2 * 2 = 4$, para $x = 3$, temos que $f(x) = 9$, e assim por diante.

Se for só para imprimir quanto vale cada um, está muito fácil. Repare:

```
for(var x = 0; x < 600; x = x + 1) {  
    var y = x * x;  
    console.log("Para x = " + x + ", o y vale " + y);  
}
```

Mas não é isso que queremos. Queremos colocar o gráfico dessa função na tela. Crie seu arquivo `funcoes.html` e vamos declarar um `canvas`:

```
<canvas id="tela" width="600" height="400"></canvas>  
<script>  
var tela = document.getElementById("tela");  
var c = tela.getContext("2d");
```

Agora fazemos um `for` que, além de colocar no console a posição que vamos desenhar, desenha um ponto na posição x , y usando um pequeno círculo de raio 2:

```
for(var x = 0; x < 600; x = x + 1) {  
    var y = x * x;
```

```

console.log("Para x = " + x + ", o y vale " + y);

c.beginPath();
c.arc(x, y, 2, 0, 2*Math.PI);
c.fill();
}
</script>

```

Abra a sua página e repare no resultado na seguinte figura:

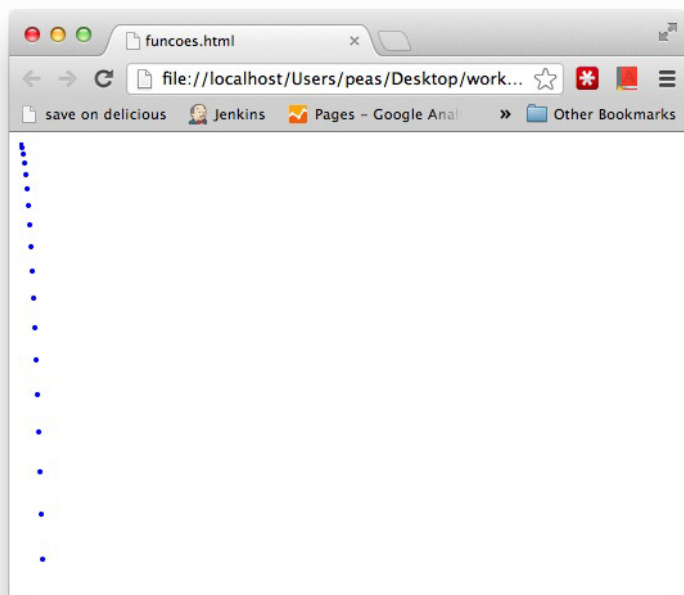


Figura 7.20: A parábola da função quadrática, mas com poucos pontos e ainda estranha

Há duas coisas que não esperávamos aí. Quando estudamos no colégio essa função, sabemos que o gráfico gerado chama-se parábola, e que a direção dela é crescente! Por que está de ponta

cabeça? Pois nosso eixo y tem direção invertida à qual estamos habituados. Para mudar isso, basta alterar o seu código para desenhar o círculo na posição $x, 400 - y$, fazendo `c.arc(x, 400 - y, 2, 0, 2*Math.PI);`.

Essa função cresce muito rápido, por isso só conseguimos ver poucos pontos dela. Só vemos os 20 primeiros pontos, pois para $x = 20$, temos um $y = 400$! Para visualizar melhor o gráfico, desenhe a função $f(x) = 0.01 * x * x$, basta alterar a linha do cálculo do y para `var y = 0.01 * x * x`. Aí sim obtemos a seguinte figura:

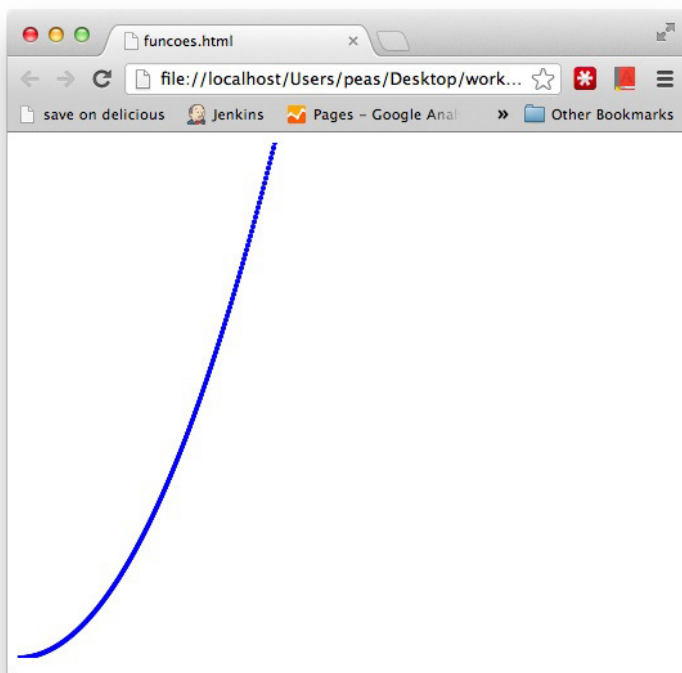


Figura 7.21: A parábola com o visual mais conhecido

Podemos organizar o nosso código e isolar esse `for` dentro de uma função que se chama `desenhaParabola`. Repare no código completo e observe que temos de fazer a chamada à função que criamos na última linha:

```
<canvas id="tela" width="600" height="400"></canvas>
<script>
var tela = document.getElementById("tela");
var c = tela.getContext("2d");

c.fillStyle="blue";
```

```
function desenhaParabola() {
    for(var x = 0; x < 600; x = x + 1) {
        var y = 0.01 * x * x;
        console.log("Para x = " + x + ", o y vale " + y);

        c.beginPath();
        c.arc(x, 400 - y, 2, 0, 2*Math.PI);
        c.fill();
    }
}

desenhaParabola();
</script>
```

Mesmo que você não saiba *matematiquês*, e se agora quisermos desenhar uma outra função, por exemplo, $f(x) = \text{seno}(x)$? Precisaríamos copiar e colar o código da `desenhaParabola` e mudar apenas uma linha de código! É uma péssima prática. E também precisamos aprender a calcular o seno de um ângulo (em radianos) em JavaScript, mas essa é a parte fácil, basta fazer `Math.sin(x)` .

O que podemos fazer aqui, antes de desenhar essa nova função, é transformar o código em algo mais genérico ainda. Em vez de calcular o ponto a desenhar aí dentro, vamos deixar isso para a função da parábola. Vamos inicialmente quebrar o nosso código em duas partes, removendo o `0.01 * x * x` de dentro do `desenhaParabola` :

```
function funcaoQuadratica(x) {
    return 0.01 * x * x;
}

function desenhaParabola() {
    for(var x = 0; x < 600; x = x + 1) {
        var y = funcaoQuadratica(x);
        console.log("Para x = " + x + ", o y vale " + y);
    }
}
```

```

        c.beginPath();
        c.arc(x, 400 - y, 2, 0, 2*Math.PI);
        c.fill();
    }
}

```

O código está ainda mais organizado, mas continuamos com o problema: como desenhar a função $f(x) = \text{seno}(x)$ sem ter de copiar e colar muito código que já escrevemos? Queremos poder escolher qual função desenhar!

Em outras palavras, em vez de chamar `desenhaParabola()`, gostaríamos que nossa última linha de código fosse a seguinte:

```
desenhaFuncao(funcaoQuadratica);
```

Precisamos deixar o código mais genérico para chegar a esse ponto. O truque aqui vai ser bem sutil: em vez de deixar a chamada para `funcaoQuadratica` explícita na função `desenhaParabola`, vamos fazer com que a `desenhaParabola` **receba uma função como argumento**. Para deixar os nomes com sentido, em vez de `desenhaParabola`, ela se chamará `desenhaFuncao` e receberá um argumento chamado `funcaoMatematica`:

```

function desenhaFuncao(funcaoMatematica) {
    for(var x = 0; x < 600; x = x + 1) {
        var y = // O QUE FAZER AQUI?
        console.log("Para x = " + x + ", o y vale " + y);

        c.beginPath();
        c.arc(x, 400 - y, 2, 0, 2*Math.PI);
        c.fill();
    }
}

```

Mas ainda falta a linha para calcular o `y`. Se não queremos uma chamada explícita, como fazer? Não falta muito, pois já recebemos a `funcaoMatematica` como argumento, basta chamá-

la com `var y = funcaoMatematica(x)` .

E para chamar a `desenhaFuncao`, basta fazer a chamada conforme havíamos planejado, `desenhaFuncao(funcaoQuadratica)` . O grande diferencial aqui é que a função `desenhaFuncao` recebe uma outra como parâmetro! E quando ela faz `var y = funcaoMatematica(x)` , ela não sabe exatamente quem está chamando! Nós sabemos que ela está chamando a parábola, pois fizemos a chamada logo depois, mas o código não depende disso, tornando-se muito flexível.

Esse código pode ser visto aqui: <http://jsfiddle.net/paulosilveira/kyuMe/>.

Repare que não é a primeira vez que trabalhamos com funções como sendo uma variável qualquer. No capítulo passado, fizemos `botaoAdivinhar.onclick = botaoClicado` quando configuramos o nosso *callback*. Mas essa é a primeira vez que **recebemos** uma função como argumento. Isso aparece com frequência em códigos JavaScript.

E para desenhar o seno? É bem simples, basta você criar essa função:

```
function funcaoSeno(x) {  
    return Math.sin(x);  
}
```

E depois chamar:

```
desenhaFuncao(funcaoSeno);
```

Simples, não?

O resultado pode não ser satisfatório. A função seno devolve

um número entre -1 e 1 , o que não tem muita graça no gráfico. Experimente uma função mais elaborada, como $\text{Math.sin}(x/20) * 100 + 200$; , o da figura a seguir.

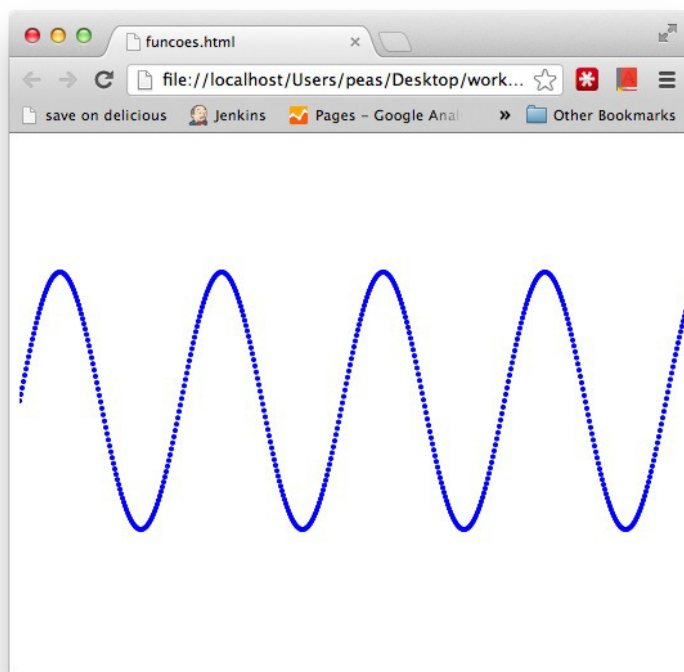


Figura 7.22: A função seno

Faça novos testes com outras funções! Procure raízes, logaritmos etc. Quem sabe até desenhar um fractal?

APÊNDICE – ANIMAÇÕES E PEQUENOS JOGOS

Já temos condições de fazer muita coisa com o que aprendemos. Vamos neste capítulo aprender mais de eventos e da API do canvas para criar até mesmo um simples joguinho. Aproveite para testar sua imaginação!

8.1 CRIE UMA LOUSA CAPTURANDO O MOVIMENTO DO MOUSE

É muito fácil saber o que o usuário está fazendo no nosso programa, para depois tomar decisões. Por exemplo, onde o usuário está clicando na tela? Vamos criar um novo arquivo, o `mouse.html` e desenhar um canvas cinza, que será a nossa tela de teste.

O começo do `mouse.html` é muito semelhante ao que já conhecemos:

```
<canvas id="tela" width="600" height="400"></canvas>
<script>
var tela = document.getElementById("tela");
var c = tela.getContext("2d");

c.fillStyle="gray";
```

```
c.fillRect(0, 0, 600, 400);  
</script>
```

Abra o `mouse.html` no seu navegador. Apenas um retângulo cinza aparece, conforme imaginado.

Queremos saber se o usuário clicou dentro da nossa tela do nosso canvas. Podemos fazer isso definindo uma função para ser chamada toda vez que um clique for feito. Basta atribuírmos uma função para `tela.onclick`. Adicione as seguintes linhas antes do `</script>`:

```
function clicouNoCanvas() {  
    alert("alguém clicou no canvas!");  
}  
  
tela.onclick = clicouNoCanvas;
```

Pronto! Rode seu programa novamente e clique na área do canvas cinza e veja o resultado.

O que fizemos aqui foi definir uma função que será chamada quando um determinado evento ocorrer. Já vimos isso antes, quando pegamos o clique do mouse em um botão do nosso HTML! Esse tipo de função é o que chamamos de *callback*. No nosso caso, definimos que, quando alguém clicar na tela (`tela.onclick`), vamos chamar uma função que, por sua vez, chama o `alert`.

Podemos descobrir a coordenada em que o usuário clicou. Muitas vezes, quando uma função de *callback* é chamada, são passados argumentos descrevendo o evento que acabou de acontecer. Neste caso é passado um evento chamado `MouseEvent`, com o qual podemos descobrir a posição `x,y` do clique por meio de variáveis de dentro desse evento. Altere seu

código da seguinte forma:

```
function clicouNoCanvas(evento) {  
    var x = evento.pageX;  
    var y = evento.pageY;  
    alert("posição do clique : " + x + ", " + y);  
}  
  
tela.onclick = clicouNoCanvas;
```

Repare que agora estamos recebendo como parâmetro uma variável a que demos o nome de `evento`. É comum dar o nome a ela de `mouseEvent`, ou até mesmo um simples `e`. Abra o seu HTML e clique em algum lugar da tela, qual é o resultado?

Isso mesmo! Ele te dá a posição do seu clique. Mesmo assim, perceba que há algo estranho. Tente, por exemplo, clicar no canto superior esquerdo da sua imagem. Observe o resultado abaixo:

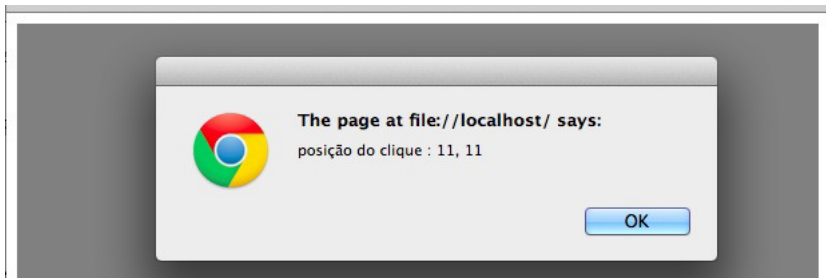


Figura 8.1: Topo esquerdo do canvas não deveria ser 0,0?

No nosso caso, mesmo clicando bem no canto superior esquerdo do nosso canvas cinza, obtivemos `11, 11`. Algumas vezes, com mais precisão, obtivemos `10,10`. Por que o resultado não foi `0,0`? Isso ocorre pois, como as próprias variáveis `evento.pageX` e `evento.pageY` nos dizem, essa é a posição do clique em relação à página! Se quisermos as coordenadas relativas

ao canvas, basta subtrairmos a posição em que o canvas (nossa tela) foi desenhado na página:

```
function clicouNoCanvas(evento) {  
    var x = evento.pageX - tela.offsetLeft;  
    var y = evento.pageY - tela.offsetTop;  
    alert("posição do clique : " + x + ", " + y);  
}  
  
tela.onclick = clicouNoCanvas;
```

Ler apenas essa informação não é tão interessante. Que tal desenhar um círculo azul em cada ponto que o usuário clicar? Basta, dentro dessa função, usarmos aquela função `drawArc` , que já conhecemos:

```
function clicouNoCanvas(evento) {  
    var x = evento.pageX - tela.offsetLeft;  
    var y = evento.pageY - tela.offsetTop;  
  
    c.fillStyle="blue";  
    c.beginPath();  
    c.arc(x, y, 10, 0, 2*3.14);  
    c.fill();  
  
    console.log("posição do clique : " + x + ", " + y);  
}  
  
tela.onclick = clicouNoCanvas;
```

Recarregue seu arquivo no navegador. Clique em alguns pontos na tela e veja o resultado que obtemos!

Veja que, além de desenhar o círculo onde você clica, estamos colocando a informação das coordenadas no console do navegador. Vimos esse recurso no começo, na seção *Mostrando mensagens secretas, apenas para o programador*. É muito mais prático utilizá-lo do que imprimir valores com `document.write` ou o inoportuno `alert` .

Para ver as posições onde estamos clicando, basta abrir o *Console JavaScript* do Chrome. Relembrando: clique no ícone de menus/ferramentas, depois acesse o menu *Ferramentas (Tools)* e, por último, *Console JavaScript*.

Após alguns cliques, você obterá um resultado como o seguinte:

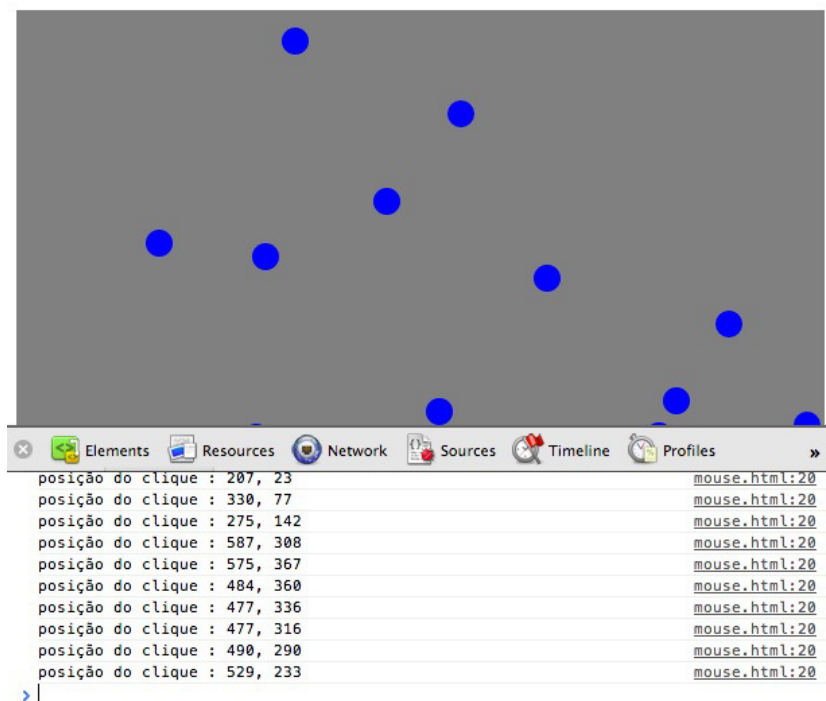


Figura 8.2: Console aberto depois de alguns cliques

Já podemos fazer um software parecido com o Paint, não? Ou como o Photoshop, para os mais ousados!

Opcionalmente, por uma questão de abreviação, poderíamos

ter declarado nossa função diretamente no registro do nosso *callback*. Algo assim:

```
tela.onclick = function(evento) {  
    // código que você já viu ficaria aqui ...  
}
```

8.2 EXERCÍCIOS PARA NOSSA TELA DE DESENHO

1) Como saber se o clique do mouse é o botão da esquerda ou da direita? Pesquise mais na internet sobre os eventos de clique e faça com que o da esquerda desenhe o círculo azul, e o da direita um círculo vermelho!

2) Podemos registrar diversos outros eventos do mouse. Até agora mexemos apenas no `onclick` do nosso `canvas`. Queremos pegar o movimento do mouse e desenhar a cada pequena variação de sua posição. Para isso, atribua a nossa função que desenha círculo não mais para `tela.onclick`, e sim `tela.onmousemove`.

3) Desafio: crie seu próprio Paint! Defina áreas do seu `canvas` em que o usuário pode selecionar algumas cores. Para isso, você deve guardar uma variável `corEscolhida` fora da sua função de desenho. Toda vez que o usuário clicar na região da escolha de uma cor (você pode fazer isso verificando com `ifs`), você deve atualizar essa variável e utilizá-la para fazer `c.fillStyle = corEscolhida`.

8.3 CRIE ANIMAÇÕES

Vamos tentar desenhar um círculo e movê-lo na tela, dando o efeito de uma animação. A ideia parece ser fácil de executar. Podemos utilizar um `for` para desenhar o círculo à esquerda da tela e depois limpar a tela. A cada iteração, vamos avançando o círculo em relação ao eixo `x`, mais para a direita.

Crie o arquivo `animacao.html` e vamos fazer exatamente isso! Primeiro, declaramos nosso canvas:

```
<script>
var tela = document.getElementById("tela");
var c = tela.getContext("2d");

c.fillStyle="green";
c.fillRect(0, 0, 200, 400);
```

Para organizar, vamos começar criando uma função que desenha um círculo na posição `x`, `y` de tamanho `raio`. O código é bastante simples:

```
function circulo(x, y, raio) {
    c.fillStyle = "blue";
    c.beginPath();
    c.arc(x, y, raio, 0, 2*Math.PI);
    c.fill();
}
```

Além dessa, vamos criar a função que limpa a tela:

```
function limpaTela() {
    c.clearRect(0, 0, 600, 400);
}
```

Por último, fazemos o `for` que desenha o círculo na posição `x`, que vai aumentando de 1 em 1. Sua posição `y` será sempre 100 e o `raio` com valor de 10:

```
for(var x = 0; x < 600; x = x + 1) {
    limpaTela();
```

```
    circulo(x, 100, 10);  
}  
</script>
```

Abra a página no Chrome. O que aconteceu? Você viu alguma coisa? Clique para atualizar e tente perceber agora, com mais calma. Nada ainda? Adivinhe o que houve!

O problema é que seu computador é muito mais rápido do que você imagina. Nessa fração de segundo, ele foi capaz de desenhar o círculo e apagar a tela 600 vezes!

Para que possamos ver a animação ocorrer, precisamos que, a cada novo quadro da animação, o computador espere alguns milésimos de segundo. E existe uma função que faz isso: `setInterval`. Ela define de quanto em quanto tempo você quer que uma determinada função seja chamada.

Remova o seu `for` e vamos escrever uma função que queremos ser chamada de tanto em tanto tempo.

```
var x = 1;  
  
function desenha() {  
    limpaTela();  
    circulo(x, 100, 10);  
    x = x + 1;  
}
```

Há dois pontos importantes aqui. Repare que declaramos o nosso contador `x` fora da função, caso contrário, toda vez ele voltaria a valer `1`. Outro ponto esquisito é que não temos mais um laço! A função `desenha` realmente apaga a tela e desenha o círculo apenas uma vez. Quem vai chamá-la várias vezes? É a `setInterval`! Para a mágica funcionar, basta adicionar antes de fechar o `</script>`:

```
setInterval(desenha, 30);
```

O número 30 indica que a função será chamada a cada 30 milissegundos. Pronto! Repare no resultado recarregando a página!

ANIMAÇÃO DE UM JOGO DE VERDADE

Vimos aqui uma forma bem simples de animar nossas figuras na tela. Em um jogo de verdade, há milhares de outros detalhes importantes. Em especial, algumas vezes você pode perceber que a tela "pisca" entre um quadro e outro. Isso ocorre porque dá para ver a tela sendo limpada e depois desenhada. Caso você queira mesmo entrar mais a fundo, você deve pesquisar por *double buffer*, com o qual você usaria dois canvas para não sofrer esse efeito. É um tópico mais avançado.

8.4 REVISE E FAÇA NOVAS ANIMAÇÕES

Crie o arquivo `animacao.html`. Primeiro, declare seu canvas junto com as duas funções: `circulo` e `limpaTela`:

```
<canvas id="tela" width="600" height="400"></canvas>
<script>
var tela = document.getElementById("tela");
var c = tela.getContext("2d");

function circulo(x, y, raio) {
    c.fillStyle = "blue";
    c.beginPath();
    c.arc(x, y, raio, 0, 2*Math.PI);
    c.fill();
}
```

```

}

function limpaTela() {
    c.clearRect(0, 0, 600, 400);
}

```

Agora declaramos a variável `x` que vai mudar onde será desenhado o círculo. Também declaramos a função `desenha`, que será chamada a cada 30 milissegundos, por meio do `setInterval`:

```

var x = 1;

function desenha() {
    limpaTela();
    circulo(x, 100, 10);
    x = x + 1;
}

setInterval(desenha, 30);
</script>

```

Pronto! Pode abrir a sua página no navegador. Aqui há um *jsfiddle* com o código pronto: <http://jsfiddle.net/8SmSS/>.

1) Em vez de fazer o círculo apenas se mover, faça com que ele também cresça. Basta utilizar a variável `x` para definir o tamanho do raio. Você pode, por exemplo, chamar `circulo(x, 100, x/2)`. Agora, remova a invocação ao `limpaTela` e use `strokeStyle` e `stroke` em vez de `fillStyle` e `fill`, dentro da função `circulo`. Você pode ter resultados psicodélicos como este:

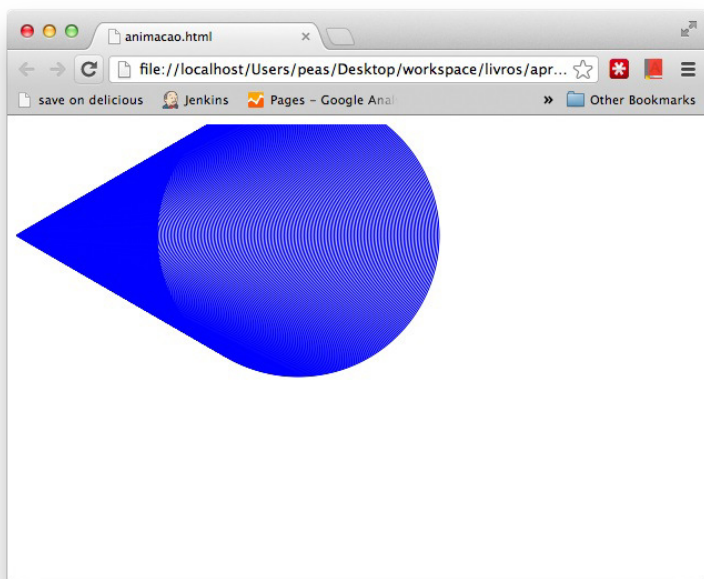


Figura 8.3: Animações estranhas são possíveis!

Utilizando linhas, quadrados e caminhos, junto com o random que aprendemos, é possível criar animações bem interessantes.

2) Em vez de fazer um círculo se mover, você pode trabalhar com imagens! É bastante simples: em vez de chamar a função `circulo`, chame a função `desenhaImagem(x, 100)` (ela não receberá raio, claro). Sua função deve ser assim:

```
var imagem = new Image();
imagem.src = "http://www.caelum.com.br/imagens/"
            + "instrutores/fotos/paulo-silveira-90.jpg";

function desenhaImagem(x, y) {
    c.drawImage(imagem, x, y);
}
```

```
}
```

Você pode trocar a imagem por qualquer outra! Caso a imagem esteja no seu computador, coloque o valor de `imagem.src = "nomeDaImagem.jpg"` e deixe esse arquivo no mesmo diretório que o seu `.html`.

3) É fácil fazer o círculo se mover de outras maneiras. Para ele descer na diagonal, basta chamar `circulo(x, x, 10);`. E se quisermos que ele se mova de outra forma, como numa parábola, vista no capítulo anterior? Ou mesmo de uma forma senoidal?

Quer ir mais longe? Um excelente artigo para você fazer sua animação, com uma imagem que vai mudando a cada quadro é este (em inglês): <http://buildnewgames.com/sprite-animation/>.

8.5 DESAFIO: O JOGO DO TIRO AO ALVO

Este é o seu último exercício. Vamos dar apenas algumas poucas coordenadas para que você mesmo o desenvolva quase que por completo.

Vamos desenhar um alvo, com círculos concêntricos vermelho e branco. Você pode criar uma função parecida com essa, além de uma função `cor` e outra `circulo`:

```
function desenhaAlvo(x, y) {  
  var raio = 40;  
  cor("white");  
  circulo(x, y, raio);  
  cor("red");  
  circulo(x, y, raio-10);  
  cor("white");  
  circulo(x, y, raio-20);  
  cor("red");  
  circulo(x, y, raio-30);  
}
```

```
}
```

A nossa função que será chamada pelo `setInterval` vai sortear uma posição para o alvo (variável `alvoX` e `alvoY`):

```
var alvoX;  
var alvoY;  
  
function desenha() {  
    limpa();  
    alvoX = sorteia(600);  
    alvoY = sorteia(400);  
    desenhaAlvo(alvoX, alvoY);  
}
```

Para isso, você vai ter de criar as funções `sorteia` e `limpa`.

Declaramos as variáveis do alvo fora da função `desenha`, pois você vai precisar acessá-las na função que captura o evento de clicar do mouse. As regiões e blocos por onde uma variável é válida, e pode ser acessada chama-se **escopo** da variável. Apesar de termos usado em diversas vezes, não entramos em detalhes em nenhum momento, visto que há muitos detalhes por trás.

Agora, toda vez que o usuário clicar (evento de `onclick` no canvas), você deve verificar se ele acertou o alvo. Você saberá isso se a distância entre o clique e o centro do alvo for menor que o raio de 40 pixels que definimos. Aqui você usará Pitágoras para calcular a distância!

No final de tudo, você deve chamar o `setInterval(desenha, 1000)` !

Consegue fazer o jogo? Então aprimore! Crie um placar. A cada tiro que acerta o alvo, some 1 em um contador e mostre esse valor usando a função `text`, fazendo algo como `c.text(30,`

30, "Pontuação: " + pontos) .

ÚLTIMAS PALAVRAS – ALÉM DA LÓGICA DE PROGRAMAÇÃO

Aprendemos muito até aqui, mas ainda há um longo caminho. Vamos apresentar, sucintamente, diversos tópicos que vão além dos nossos objetivos, mas que podem ser muito interessantes para você. Alguns deles você vai necessariamente ter de conhecer quando começar a programar profissionalmente.

9.1 OBJETOS

Como falamos no capítulo de estruturas, no JavaScript trabalhamos com objetos. Fica fora do escopo deste livro, mas você vai com certeza vê-los com frequência. Vimos em arrays, vimos no `console`, vimos no `document`, apenas não entramos em detalhes.

É uma forma de organizar nossas variáveis (e ir um pouco além).

Quando trabalhamos com muitos dados, pode começar a ficar complicado trabalhar com muitas informações. Veja só:

```
var nome = "Paulo Silveira";  
var idade = 33;  
var email = "paulo.silveira@casadocodigo.com.br";
```

Até aqui tudo bem. Mas e se tivermos o nome, idade e e-mail do Adriano? Criaremos mais três variáveis e tudo pode ficar bastante confuso.

Há uma forma de agrupar essas informações:

```
var pessoa = {  
  nome : "Paulo Silveira",  
  idade : 33,  
  email : "paulo.silveira@casadocodigo.com.br"  
};
```

A variável `pessoa` é um objeto. Se você fizer `console.log(pessoa)`, vai obter `Object {nome: "Paulo Silveira", idade: 33, email: "paulo.silveira@casadocodigo.com.br"}`. Para acessar cada pedaço da informação separadamente, é fácil e vai lembrar algo:

```
console.log(pessoa.nome);  
console.log(pessoa.idade);  
console.log(pessoa.email);
```

Você pode até mesmo alterar os valores, fazendo `pessoa.idade = 34`, por exemplo.

Usamos objetos diversas vezes durante nosso aprendizado, só não ficamos dando muita atenção ao fato. Essa é apenas a ponta do iceberg. Objetos é um assunto complexo e extenso em JavaScript. Programar orientado a objetos é também peça chave para o desenvolvedor.

Quer fazer um exercício simples para praticar? Crie um formulário com os campos de nome, e-mail e idade. Ao clicar no

botão adicionar, você deve adicionar um objeto que contém esses três dados dentro de uma array de contatos , e listá-la no console.

9.2 BOAS PRÁTICAS QUE FORAM VIOLADAS DURANTE O APRENDIZADO

Há alguns truques que evitamos fazer por simplicidade.

Debug

Não falamos sobre o processo de *debugging* de um programa. Dentro do console do Chrome, você pode abrir o seu código-fonte e executá-lo passo a passo para tentar achar algum problema. Chamamos esse processo de *debug*. Evitamos no livro para que você mesmo saiba fazer esse caminho junto com seu programa, com a ajuda do `console.log` , mas o debug será uma ferramenta importante para seu desenvolvimento.

Callbacks

Aprendemos a mudar nosso callback de clicar em um botão desta forma:

```
var botaoAdivinhar = document.getElementById("adivinhar");  
botaoAdivinhar.onclick = botaoClicado;
```

Mas pode ser que o botão já tivesse uma função de callback registrada. Um evento pode chamar mais de uma função, o que é útil em alguns casos. No dia a dia, você encontrará chamadas como esta, que adicionam um callback a mais, sem remover possíveis existentes:

```
var botaoAdivinhar = document.getElementById("adivinhar");  
botaoAdivinhar.addEventListener('click', botaoClicado, false)
```

Organização do nosso código

Nós sempre misturamos nosso código HTML com o JavaScript. Em vez de trabalhar dessa maneira, o programador experiente sempre vai separar. Podemos fazer isso pela própria tag script :

```
<script src="codigo.js"></script>
```

Para isso funcionar, teríamos todo nosso código JavaScript dentro do arquivo `codigo.js` , que deve ficar, nesse caso, no mesmo diretório que nosso HTML.

Há também algumas boas práticas sobre o carregamento do JavaScript, que não deve atrapalhar o carregamento do HTML, que poderia atrasar a visualização da páginas, mas esse é um outro assunto.

Bibliotecas

Adivinhe: há muito código já pronto que outras pessoas disponibilizaram em arquivos para que possamos usá-los. Costumamos nos referir a esse conjunto de funções que trabalham com determinados recursos de **biblioteca** ou API. Mesmo um conjunto de funções que já faz parte do JavaScript, como a do canvas, às vezes é referenciado como biblioteca.

Sem dúvida alguma a biblioteca que mais aparece com JavaScript é a jQuery: <http://jquery.com/>

Há um livro da Casa do Código basicamente só sobre esse

assunto e detalhes do JavaScript.

Mas há bibliotecas para basicamente tudo! O universo JavaScript é enorme. Uma que pode facilitar o desenvolvimento de jogos é a MelonJS: <http://www.melonjs.org/>.

E se eu quiser, por exemplo, Gráficos 3D?

Sim! Você também pode criar gráficos 3D em um Canvas. É certamente mais difícil do que o que vimos no livro, pois envolve mais uma coordenada e muito mais matemática, mas não é impossível. Para isso, você pode pesquisar por WebGL, uma API JavaScript para o uso do 3D.

Há também uma biblioteca bastante conhecida, a Processing, que possui uma versão em JavaScript. Ela facilita o trabalho com 3D para que você crie interessantes aplicações de visualização de dados:

<http://processingjs.org/>

Aqui colocamos dois exemplos interessantes com o Processing:

- <http://mylifeaquatic.herokuapp.com/>
- <http://videos.mozilla.org/serv/mozhacks/flight-of-the-navigator/>

9.3 PRATIQUE MUITO!

É importante colocar em prática o seu conhecimento.

Com certeza você já tem muitas ideias na cabeça. Você pode

fazer jogos, cadastros e trabalhar com bibliotecas. Ainda está sem ideias? Você pode misturar tudo isso em um único projeto. Vamos a uma sugestão!

Crie uma página HTML, que será o cadastro de uma Academia. Lá, você deve ter campos para perguntar o nome, a idade, o peso e a altura. Ao clicar em *Gerar Relatório*, seu programa deve dar o IMC do usuário junto com um bonito gráfico mostrando onde ele está em relação a escala adequada de peso para a altura dele.

Você pode desenhar uma barra que é mais verde ao centro, perto do IMC ideal e mais vermelha nos extremos! Também pode mostrar uma tabela com a frequência cardíaca recomendada para os exercícios aeróbicos. Para isso, consulte na internet e achará uma fórmula que deverá ser aplicada.

Aproveite e aplique o conceito de objetos para guardar os dados do nosso cliente da academia.

9.4 CONTINUE SEUS ESTUDOS

Gostou do que viu por aqui? Seu próximo passo é praticar bastante. Utilize os exemplos e crie seus próprios desafios. Não deixe de participar da nossa lista de discussão:

<http://forum.casadocodigo.com.br>

E também pode postar dúvidas no fórum do GUJ:

<http://www.guj.com.br/>

Há diversos caminhos por onde você pode continuar.

Um é estudar mais profundamente o HTML e o JavaScript (e também o CSS, que não vimos aqui). Há livros sobre HTML, CSS e sobre JavaScript e jQuery na Casa do Código. Há também cursos sobre esses assuntos na Caelum, o WD-43 e o WD-47.

Você pode ir pelo caminho do Ruby on Rails. O Ruby é uma outra linguagem, e o Ruby on Rails é algo como uma ferramenta para criar sistemas na web. Tem também livros na editora e o curso RR-71 na Caelum.

O Java (sim, é diferente de JavaScript) também é uma opção, que pode abrir portas para você desenvolver aplicativos para celular, no Android. Tem os cursos FJ-11 e FJ-57 na Caelum, além do livro de Android da Casa do Código, mas é necessário conhecer a linguagem Java para lê-lo.