

**Front-end**  
magazine

Edição 11



**Boas Práticas com JavaScript**  
Evite os maus códigos e entenda  
a linguagem a fundo

**OOCSS**  
Implemente CSS organizado  
e orientado a objetos

# DESIGN RESPONSIVO

**Responsividade na web  
com HTML5 e CSS3**



# MVP

R\$ 1.000.000,00  
INVESTIDOS EM CONTEÚDO  
NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO  
NA SUA CARREIRA...

E MOSTRE AO MERCADO  
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!



 **DEVMEDIA**

## EXPEDIENTE

### Editor

Diogo Souza ([diogosouzac@gmail.com](mailto:diogosouzac@gmail.com))

### Consultor Técnico

Daniella Costa ([daniella.devmedia@gmail.com](mailto:daniella.devmedia@gmail.com))

### Produção

**Jornalista Responsável** Kaline Dolabella - JP24185

**Capa e Diagramação** Romulo Araujo

### Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse [www.devmedia.com.br/central](http://www.devmedia.com.br/central), ou se preferir entre em contato conosco através do telefone 21 3382-5038.

### Publicidade

[publicidade@devmedia.com.br](mailto:publicidade@devmedia.com.br) – 21 3382-5038

**Anúncios** – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

### Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



**DIOGO SOUZA**

[diogosouzac@gmail.com](mailto:diogosouzac@gmail.com)

Analista de Sistemas Java na Indra Company e já trabalhou em empresas como Instituto Atlântico e Ebix L.A. É instrutor Android, palestrante em eventos sobre Java e o mundo mobile e consultor DevMedia. Conhecimentos e experiências em diversas linguagens e ferramentas de programação e manipulação de dados, bem como metodologias úteis no desenvolvimento de Sistemas diversificados.

### Artigo no estilo Mentoring

## 04 – Dominando Design Responsivo com HTML5 e CSS3

[ *Fábricio Galdino* ]

# Sumário

### Conteúdo sobre Boas Práticas

## 18 – Boas Práticas de programação em JavaScript

[ *Fábricio Galdino* ]

### Artigo no estilo Curso

## 28 – Desenvolvimento de jogos web com Pixi.js – Parte 1

[ *Júlio Sampaio* ]

### Conteúdo sobre Novidades

## 39 – Introdução ao desenvolvimento de CSS orientado a objetos

[ *Júlio Sampaio* ]

# Dominando Design Responsivo com HTML5 e CSS3

Crie aplicações web responsivas com técnicas produtivas usando apenas HTML5 e CSS3

ESTE ARTIGO É DO TIPO MENTORING

SAIBA MAIS: [WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI](http://WWW.DEVMEDIA.COM.BR/MENTORING-SAIBAMAI)

**S**em dúvida a responsividade é o conceito mor que define a qualidade de um site, aliada a conceitos como confiabilidade, segurança, performance e preocupação dos criadores para com seu público. Não dá mais para pensar em criar qualquer aplicação web ou website sem priorizar a responsividade como pré-requisito básico e fundamental. O próprio Google, bem como outras ferramentas de busca na web, ranqueiam melhor sites que apresentam seu conteúdo de forma *mobile friendly*, isto é, preparam seus conteúdos para serem exibidos em dispositivos de diversos tamanhos, a incluir smartphones, tablets, notebooks, etc. Em ferramentas e metodologias de SEO (*Search Engine Optimization*) esse conceito é tido como primário para um bom resultado junto aos sites de pesquisa, logo, você deve estar preparado para lidar com isso em seus projetos front-end como um todo.

Frameworks famosos como o Twitter Bootstrap, Angular Material (do AngularJS) e o novo Google Material Design Lite já trazem toda uma especificação de componentes HTML, classes CSS e funções JavaScript prontas para lidar com o desenvolvimento de todo tipo de página responsiva. Elas permitem ainda que você customize facilmente esse aparato de arquivos usando ferramentas de manipulação dinâmica de CSS (Less ou Sass), JavaScript (jQuery, AngularJS, etc.) e HTML puro, via build automatizado, usando o Grunt, por exemplo.

Todavia, existem várias situações onde o projeto em que você estiver trabalhando exija que o design da aplicação como um todo seja completamente diferente

## Cenário

Desenvolver aplicações web hoje em dia muitas vezes se resume à correta escolha do framework de componentes (como Bootstrap, Google Material Design Lite, etc.) para facilitar a construção da estrutura e estilo das páginas. Esse cenário funciona muito bem até o momento em que tais bibliotecas não atendem mais aos recursos específicos do seu projeto: por exemplo, quando precisamos de componentes/design muito específicos ou, pior ainda, quando precisamos misturá-los de duas bibliotecas diferentes numa só aplicação para atender a demanda, aumentando consideravelmente a quantidade de código e arquivos (muitos deles sequer usados).

Uma solução mais adequada seria a criação da sua própria biblioteca de componentes responsivos, com seu próprio estilo. Assim, evitamos o excesso de implementação e, acima de tudo, temos o total controle de todo o website. Este artigo vai de encontro à essa realidade, ensinando a fazer uso dos recursos mais recentes da HTML5 e CSS3 para construir seus frameworks web.

do fornecido por esses frameworks, ou casos em que os componentes das mesmas não sejam suficientes para desenvolver as peculiaridades que seu projeto exige, dentre outros. Nesses casos, o designer, ou desenvolvedor front-end deve ter embasamento o suficiente para lidar com esse tipo de desenvolvimento, de repente criando seu próprio framework de componentes, classes e funções. E são exatamente estes pontos que esse artigo visa tratar, explorando conceitos como grids, desenvolvimento com Sass, tipografia, dentre outros.

## O poder do Sass

Para fins de simplicidade e produtividade, usaremos o Sass para facilitar a implementação do CSS nos exemplos do artigo.

O Sass permite que elementos sejam aninhados para flexibilizar a forma como as regras serão geradas no CSS final.

Por exemplo, vejamos o exemplo demonstrado pela **Listagem 1**. A primeira parte dela mostra um código CSS feito em Sass que define a forma de exibição de um elemento de classe “barra-navegacao” via *display* com valor *flex* (que diz que o conteúdo deve se adaptar às dimensões da tela onde for exibido), bem como o *padding* dos elementos de lista *li* que estiverem inseridos dentro do anterior. Na segunda parte, vemos o resultado da compilação desse Sass, ou seja, o CSS final gerado. Veja como o Sass simplifica a criação de regras de estilo ao evitar que tenhamos de duplicar código sempre que quisermos uma regra em herança.

#### Listagem 1. Exemplo de conteúdo Sass e CSS gerado.

```
// Antes
.barra-navegacao {
  display: flex;
  li {
    padding: 5px 10px;
  }
}
// Depois
.barra-navegacao {
  display: flex;
}
.barra-navegacao li {
  padding: 5px 10px;
}
```

O Sass se baseia num conjunto de vários conceitos para implementar CSS, vejamos:

- Ele pode ser baseado em duas tecnologias diferentes: Ruby ou LibSass (**BOX 1**). Vamos usar neste artigo o Ruby por questões de simplificação.
- Ele é um gem no Ruby, isto é, um pacote usado no Ruby.
- Podemos executá-lo via interface de linha de comando, mas também é possível sua execução via aplicações de terceiros, com a devida importação de suas bibliotecas dependentes.
- Ele é uma linguagem de *scripting* convencional, como JavaScript, CoffeeScript, etc.
- O Sass também contribui para a eliminação das repetições de código quando desenvolvemos CSS puro. Isso se dá por intermédio da sua sintaxe hierárquica que permite o reaproveitamento de regras para os elementos que pertencerem à mesma. Veremos mais detalhes sobre isso na prática adiante.
- Parte do fluxo de trabalho do Sass é “assistir” a um arquivo de tipo SCSS, por exemplo, *estilos.scss*. Quando ele detecta uma mudança nesse arquivo, um novo arquivo *estilos.css* é automaticamente gerado e compilado.

#### BOX 1.LibSass

É uma versão do Sass que possibilita a integração do mesmo em outras linguagens como C/C++, Lua, Java, .NET, etc.. Veja na seção **Links** a URL oficial do projeto.

#### Configurando o Sass

Precisamos de basicamente três passos para ter o Sass instalado:

1. Baixar e executar o instalador do Ruby.
2. Abrir um terminal de comandos referente ao seu SO.
3. Instalar a gem do Sass.

Para verificar se o Ruby já está instalado na sua máquina, digite o seguinte comando no cmd:

```
ruby -v
```

Uma mensagem com a versão do mesmo, bem como a data da *revision* e versão do instalador para o seu SO (32 ou 64 bits) será impressa no console:

```
ruby 2.1.5p273 (2014-11-13 revision 48405) [x64-mingw32]
```

Caso o leitor já esteja com uma versão antiga do Ruby instalada é aconselhado remove-la antes de instalar a nova, a fim de evitar problemas de conflito no ambiente. Para o Windows, especificamente, precisamos do *RubyInstaller for Windows* (vide seção **Links**).

Quando finalizar o download, execute o arquivo, selecione a língua para o passo a passo da instalação, aceite os termos de licença e, na próxima tela, selecione o diretório onde deseja que os arquivos sejam descompactados, bem como marque as opções mostradas na **Figura 1** para que a pasta de binários seja adicionada às variáveis de ambiente e possamos executar comandos no cmd. Clique em *Install* e aguarde.

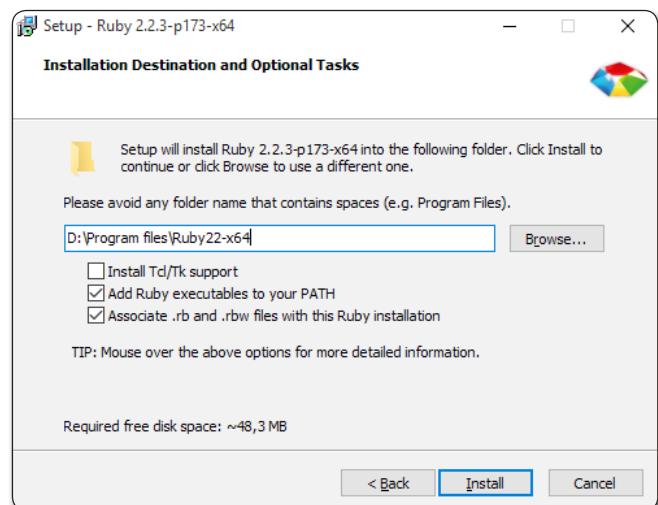


Figura 1. Tela de configuração das opções do Ruby

No final, antes de executar qualquer comando com o novo Ruby precisamos fechar a janela do terminal e abrir uma nova. Então execute mais uma vez o comando de verificação da versão e veja se a mesma corresponde à versão instalada:

```
ruby 2.2.3p173 (2015-08-18 revision 51636) [x64-mingw32]
```

# Dominando Design Responsivo com HTML5 e CSS3

Com o Ruby ok, agora é hora de instalar a gem do Sass no ambiente. Para isso, execute o seguinte comando no terminal:

```
gem install sass
```

Aguarde até que o utilitário do gem faça o download completo e execute o comando `sass -v` para verificar se deu certo. O resultado será algo como:

```
Sass 3.4.20 (Selective Steve)
```

Para trabalhar com o Sass, precisamos ter um mínimo de organização de diretórios possível. Portanto, selecione um diretório de sua preferência para o projeto e crie a estrutura de pastas demonstrada na **Listagem 2**. Veja que temos duas pastas de estilo em detrimento dos arquivos de Sass não compilados, e CSS pós compilados.

## Listagem 2. Estrutura de diretórios do projeto.

```
+--responsive-web
    +--- html
    +--- img
    +--- js
    +--- style
        +--- css
        +--- SCSS
```

Navegue até a pasta `/scss` e crie um novo arquivo de nome `estilos.scss`. No terminal cmd, usando o comando `cd`, navegue até a pasta `/style` e execute a seguinte instrução:

```
sass --watch scss:css
```

Esse comando é responsável por dizer ao Sass que ele deve “assistir” as pastas informadas à frente (`scss/css`), checando quais arquivos do Sass existem, e gerando automaticamente o arquivo CSS correspondente sempre que alterações forem feitas no de extensão `.scss`. Salve qualquer conteúdo Sass no arquivo e veja o arquivo `.css` ser gerado de forma automática na sua respectiva pasta.

## Sass vs SCSS

Existem duas sintaxes padrão para escrever código de estilo usando o Sass: a **Sass** e a **SCSS**. A primeira foi, por muito tempo, a única forma disponível para tal finalidade, porém era considerada muito diferente do CSS original, aumentando assim a curva de aprendizado na tecnologia, mesmo para quem já tivesse familiaridade com o mesmo. Vejamos um exemplo bem simples na **Listagem 3** que define uma propriedade de `float` para um seletor **a**, e uma propriedade de cor (`color`) da fonte para o mesmo seletor e para um outro seletor **b**.

A sintaxe Sass se baseia em “espaços em branco” para definir a hierarquia dos elementos do CSS final.

Quando criamos o seletor **b** com um espaçamento maior que o do seletor **a** dizemos ao Sass que a regra se aplica a ambos. Caso contrário, se o seletor **b** estivesse no mesmo nível do seletor **a**, teríamos a regra aplicada apenas ao primeiro. O CSS de resultado pode ser visto na mesma listagem. Veja como o procedimento não é tão intuitivo quanto usar a sintaxe do SCSS, como podemos ver na **Listagem 4**. Nele temos o uso do novo símbolo `&`, que nos permite adicionar o nome do seletor pai aos seletores aninhados sem ter de digitar o nome completo. Em outras palavras, basta mapear o prefixo do seletor **e**, dentro do mesmo, concatenar cada um dos nomes via operador `&`; assim, o SCSS fará todo o trabalho de montagem que acabará findando no mesmo resultado do exemplo com Sass.

## Listagem 3. Exemplo de CSS gerado a partir de um Sass.

```
// Antes
.seletor-a {
  float: left;

.seletor-b {
  color: red;

// Depois
.seletor-a {
  float: left;
}
.seletor-a, .seletor-b {
  color: red;
}
```

## Listagem 4. Exemplo de CSS gerado a partir de um SCSS.

```
// Antes
.seletor- {
  &a {
    float: left;
  }
  &a, &b {
    color: red;
  }

// Depois
.seletor-a {
  float: left;
}
.seletor-a, .seletor-b {
  color: red;
}
```

## HTML5

A HTML é uma parte essencial de ser assimilada e entendida quando do desenvolvimento de código responsivo. Diante disso, muitas tags padrão da HTML5 que existem para nos auxiliar são simplesmente ignoradas pelos desenvolvedores e designers. Veja na **Tabela 1** algumas delas, suas utilidades e as regras nas quais as mesmas se inserem quando se trata de criar conteúdo responsivo.

Outro conceito importante para a definição de uma correta responsividade nas páginas HTML é o de *roles* do WAI-ARIA

Tag	Descrição	Regras
<main>	Pode ser usada como um container para os conteúdos principais do documento. Geralmente está relacionada à marcação de um tópico central de uma seção/funcionalidade da aplicação. Esse conteúdo deve ser único no documento.	<ul style="list-style-type: none"> <li>O conteúdo principal da página deve ser incluído nessa tag.</li> <li>Deve ser exclusivo e único.</li> <li>Essa tag não deve ser inserida nas tags: &lt;header&gt;, &lt;footer&gt;, &lt;nav&gt;, &lt;aside&gt; ou &lt;article&gt;.</li> <li>Deve haver apenas uma dela por página.</li> </ul>
<article>	Representa uma composição autocontida num documento, página, aplicação, ou site, que deve ser distribuível ou reusável de forma independente. Pode ser um post de fórum, um artigo de revista ou jornal, uma entrada num blog, ou qualquer outro item independente de conteúdo.	<ul style="list-style-type: none"> <li>Um &lt;article&gt; pode ser aninhado em outros &lt;article&gt;'s.</li> <li>Pode existir mais de um por página.</li> </ul>
<section>	Representa uma seção genérica de um documento, isto é, um agrupamento temático de conteúdo, tipicamente com um cabeçalho. Cada seção deve ser identificada incluindo um cabeçalho (<h1><h6>) como elemento filho.	<ul style="list-style-type: none"> <li>Um modo seguro de usar uma seção é colocá-la dentro de um elemento &lt;article&gt;. Além disso, é bom sempre colocar um cabeçalho &lt;h1&gt;&lt;h6&gt; em cada uma.</li> <li>Pode existir mais de uma por página.</li> </ul>
<aside>	Representa uma seção com conteúdo conectado lateralmente ao corpo da página. O mesmo pode ser considerado separado do resto do conteúdo. Geralmente são representadas como sidebars ou inserts. Bons exemplos de uso dela incluem: biografia de um autor, informações de perfil de usuário, ou links relacionados num blog.	<ul style="list-style-type: none"> <li>Se o conteúdo não se encaixa na tag &lt;main&gt;, ele certamente poderá ser inserido num aside.</li> <li>Pode existir mais de um por página.</li> </ul>
<header>	É comum pensar que a seção de topo da nossa página é o header (cabeçalho), e que isso é correto. Entretanto, o nome correto para essa parte do site é masthead, ou seja, o header principal da página, que geralmente contém a logo, alguma navegação, talvez um campo de pesquisa, etc. Já o header pode ser considerado o cabeçalho de qualquer seção, portanto, podemos ter vários deles.	<ul style="list-style-type: none"> <li>Uma boa regra é sempre usar o &lt;header&gt; dentro de uma &lt;section&gt;.</li> <li>Você pode mapear um heading (&lt;h1&gt;&lt;h6&gt;) em um header, mas não é comum, tampouco necessário.</li> <li>Pode existir mais de um por página.</li> </ul>
<footer>	Representa o rodapé para a sua seção mais próxima, ou secciona o elemento raiz da página. Tipicamente contém informações sobre autor da seção, dados de copyright, ou links para documentos relacionados.	<ul style="list-style-type: none"> <li>Deve sempre conter informações sobre o elemento pai que o contém.</li> <li>Apesar do nome “footer” se referir à parte de baixo de uma página, artigo ou aplicação, esse elemento não necessariamente tem que estar nesta parte.</li> <li>Pode ter mais de um por página.</li> </ul>
<nav>	Representa uma seção da página que conecta a outras páginas ou a outras partes da mesma página: uma seção com links de navegação, por exemplo.	<ul style="list-style-type: none"> <li>Deve ser usada para agrupar uma lista ou coleção de links, externos ou internos.</li> <li>É uma prática comum usar listas não-ordenadas (&lt;ul&gt;) dentro desse elemento para melhor estruturar os links, já que é mais simples para o design.</li> <li>Também é comum incluir essa tag em um elemento &lt;header&gt;, mas não é requerido.</li> <li>Nem todos os links são requeridos de estar dentro desse elemento. Por exemplo, se tivermos links no rodapé, não é preciso inserir uma &lt;nav&gt; dentro do &lt;footer&gt;.</li> <li>Pode existir mais de um por página.</li> </ul>

**Tabela 1.** Lista de tags usadas para incutir responsividade

(Web Accessibility Initiative – Accessible Rich Internet Applications), um padrão internacional que visa implementar práticas de acessibilidade nas aplicações ricas da web. As suas *roles* (ou regras), também chamadas de *ARIA roles*, servem para definir determinados comportamentos às páginas, através de tipos pré-definidos. Vejamos um exemplo simples disso:

```
<header role='banner'>
```

Existem muitos tipos diferentes de *roles* que podem ser implementadas, mas vamos focar apenas nas mais importantes para o conceito acessibilidade responsiva. A **Tabela 2** ilustra esta relação, com suas respectivas características.

### Meta tags

As *meta tags* são estruturas importantes, principalmente no SEO, para marcar coisas como autoria das páginas, versionamento, se a página é responsiva ou não, bem como enviar certos comandos aos navegadores que identificam o conteúdo e melhoram a exibição do HTML. Veja na **Tabela 3** uma relação das tags (e suas descrições) mais importantes para implementar design responsivo nas páginas web.

### Exemplo prático

Para exemplificar os conceitos até aqui expostos, vamos implementar uma página inteiramente responsiva, com a inclusão de código HTML5 organizado com base nas tags que explanamos,

# Dominando Design Responsivo com HTML5 e CSS3

bem como estilo definido pelo documento Sass criado. Trata-se de uma página em formato de site convencional, com um cabeçalho (e barra de navegação, menus, caixa de pesquisa), conteúdo principal (com um artigo, cabeçalho interno, formulário de contato) e um rodapé (com *copyright* e links de rodapé).

Comecemos então pelo cabeçalho da página. Crie um novo arquivo HTML de nome index.html no diretório raiz do projeto e insira ao mesmo o código da **Listagem 5**. Ela traz apenas uma implementação simples da tag `<head>` da página HTML, com as respectivas *meta tags* que citamos e foram antes inseridas, um

título (tag `<title>`) e o arquivo de estilo CSS que criamos sendo importado. Ainda precisamos criar o conteúdo deste último. Observe também que, em vez das convencionais tags `<div>` usadas para dividir as seções das páginas, estamos usando a tag `<header>` dentro do corpo HTML para demarcar o cabeçalho único e principal do site. Demos a ele uma classe CSS de nome “masthead” e a *role* de “banner”, já que será único também. Dentro do *header* temos duas *divs*: a primeira contém o título do cabeçalho propriamente dito e a segunda guarda o formulário de pesquisa que, por sua vez, contém a *role* de “search” (única por página).

Role	Descrição
banner	<ul style="list-style-type: none"><li>• É usualmente aplicada à tag <code>&lt;header&gt;</code> do topo da página.</li><li>• Geralmente, o conteúdo que tem um <i>role</i>=‘banner’ aparece constantemente ao longo do site, em vez de apenas em uma página específica.</li><li>• Apenas uma é permitida por página.</li></ul>
navigation	<ul style="list-style-type: none"><li>• É usualmente aplicada ao elemento <code>&lt;nav&gt;</code>, mas pode também ser aplicada a outros containers como <code>&lt;div&gt;</code> ou <code>&lt;ol&gt;</code>.</li><li>• Descreve um grupo de elementos/links navegáveis (internos ou externos).</li><li>• Pode ter mais de um na página.</li></ul>
main	<ul style="list-style-type: none"><li>• É usualmente aplicada ao elemento <code>&lt;main&gt;</code> da página.</li><li>• O container principal/central da página deve ser marcado com essa role.</li><li>• Apenas uma é permitida por página.</li></ul>
contentinfo	<ul style="list-style-type: none"><li>• É usualmente aplicada ao elemento <code>&lt;footer&gt;</code> da página.</li><li>• É a seção que contém informações sobre o documento/site/app.</li><li>• Apenas uma é permitida por página.</li></ul>
search	<ul style="list-style-type: none"><li>• É usualmente aplicada ao elemento <code>&lt;form&gt;</code> que pertence à funcionalidade de busca da página.</li><li>• Se o elemento <code>&lt;form&gt;</code> estiver mapeado dentro de uma <code>&lt;div&gt;</code>, essa role também pode ser aplicada à mesma. Se for o caso, não há necessidade de adicionar a <i>role</i> ao <code>form</code> novamente.</li><li>• Pode haver mais de uma por página.</li></ul>
form	<ul style="list-style-type: none"><li>• É usualmente aplicada ao elemento <code>&lt;div&gt;</code> que contém algum tipo de formulário, exceto o de search que acabamos de ver.</li><li>• Não deve ser aplicada ao elemento de <code>&lt;form&gt;</code> atual, porque o mesmo já tem uma semântica de role padrão que suporta essa tecnologia.</li></ul>
complementary	<ul style="list-style-type: none"><li>• É usualmente aplicada ao elemento <code>&lt;aside&gt;</code>.</li><li>• Deve ser usada em uma região que contém suporte a conteúdo.</li><li>• Pode haver mais de uma por página.</li></ul>

**Tabela 2.** Lista de roles usadas pela WAI-ARIA

Meta Tag	Exemplo	Descrição
viewport	<code>&lt;meta name="viewport" content="width=device-width, initial-scale=1"&gt;</code>	<ul style="list-style-type: none"><li>• É a meta tag mais importante para o design responsivo.</li><li>• A diretiva “viewport” descreve o tipo de <i>meta tag</i>.</li><li>• A propriedade “width” define o tamanho do <i>viewport</i>. O valor “device-width” diz ao browser para ocupar todo o espaço da tela na largura. Também pode ser informado um valor em pixels.</li><li>• A propriedade “initial-scale” define o nível de zoom sob o qual a página deve ser exibida. 1 é igual a 100% de zoom e 1.5 igual a 150%, por exemplo.</li></ul>
X-UA-Compatible	<code>&lt;meta http-equiv="X-UA-Compatible" content="IE=edge"&gt;</code>	<ul style="list-style-type: none"><li>• Visa estabelecer compatibilidade com o Internet Explorer, por isso é aplicada apenas a esse navegador.</li><li>• A diretiva “http-equiv” diz ao IE que uma certa <i>engine</i> de renderização precisa ser usada para carregar a página.</li><li>• A diretiva “contente” diz ao IE que ele deve usar suas <i>engines</i> de HTML e JavaScript mais recentes.</li></ul>
charset	<code>&lt;meta charset="utf-8"&gt;</code>	<ul style="list-style-type: none"><li>• Essa é talvez a mais conhecida pelos desenvolvedores, já que se não inserida na página, os caracteres especiais e acentos gráficos aparecerão distorcidos.</li><li>• Ela diz ao browser que conjunto de caracteres deve ser uso para interpretar o conteúdo.</li><li>• Outro valor muito comum é “ISO-8859-1”, mas o UTF-8 é mais aconselhado pois há mais chances de o seu browser o interpretar.</li></ul>

**Tabela 3.** Lista de meta tags usadas para incutir responsividade

Para todos os campos de texto da página daremos a classe CSS “field” para padronizar no Sass.

Com o HTML pronto precisamos agora criar o conteúdo Sass para lidar com o design inicial da página.

**Listagem 5.** Cabeçalho HTML da nossa página do site.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Design Responsivo com HTML5 & CSS3</title>
  <link rel="stylesheet" href="style/css/estilos.css">
</head>

<body>
  <header class="masthead" role="banner">
    <div class="logo">Design Responsivo com HTML5 & CSS3</div>
    <div class="search" role="search">
      <form>
        <label>Busca:</label>
        <input type="text" class="field">
        <button>Buscar Agora!</button>
      </form>
    </div>
  </header>

  <!-- Restante do conteúdo -->

</body>
</html>
```

Comecemos então pelo Sass de propriedades globais, tais como cor do plano de fundo, logo, cabeçalho e alguns atributos globais que serão comuns a todos os elementos da página. Insira o código da **Listagem 6** no arquivo *estilos.scss*. Veja que no início dela criamos algumas declarações globais. No Sass é possível declarar variáveis, tal como fazemos nas linguagens de programação como JavaScript, por exemplo, e usar seus valores em todo o restante do código.

Isso facilita a padronização uma vez que não precisamos repetir a cor da fonte, por exemplo, para toda nova regra CSS criada. As cores abrangem o plano de fundo, fontes, cabeçalhos, etc. Em seguida, definimos uma função *mixin* (vide **BOX 2**) do Sass de nome *paraTelasPequenas()*, a qual será responsável por traduzir as regras criadas mais abaixo para dispositivos pequenos, por isso a divisão por *16+em* no atributo *@media*.

**BOX 2. Mixins**

Um mixin é uma diretiva que permite a definição de múltiplas regras que serão traduzidas para o CSS dependendo dos argumentos enviados dinamicamente por parâmetro. Pense num mixin como uma função que permite reusar estilo ao longo da folha de estilos sem precisar recorrer a recursos estáticos do CSS em si.

Lembre-se que este atributo é responsável por delimitar o tamanho das telas onde certas regras CSS devem se aplicar. Em seguida, dividimos o restante das regras em algumas subcategorias, a saber:

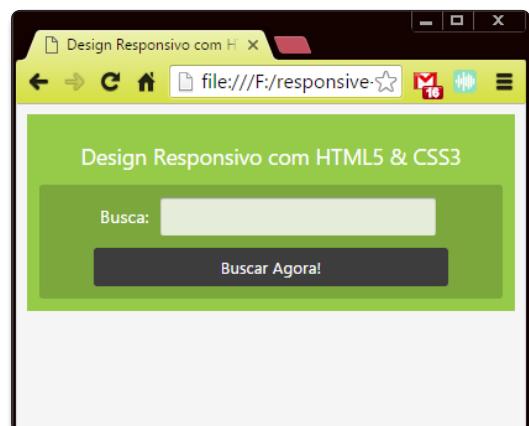
- **Regras globais:** definem propriedades comuns do CSS para fonte do texto, cor de fundo (via variável do Sass, *&bg*), cor dos cabeçalhos *<h1><h6>* e estilo dos *blockquote's* (neste último, chamamos o mixin pela primeira vez, passando uma regra específica para telas de até 420px).
- **Header:** definimos o estilo do cabeçalho, que deve estar alinhado ao centro com seus elementos também centralizados, uma logo (somente texto) e cor de fundo (note que estamos usando o mesmo estilo do portal da DevMedia).
- **Placeholder:** essa é a primeira herança no Sass que usamos. Esse elemento contém regras CSS universais que todos os elementos de seção também terão. Dessa forma, eles só precisam estender do *placeholder* e as mesmas regras serão aplicadas.
- **Campo de busca:** logo na classe “search” já herdamos do *placeholder* que criamos. O restante das configurações é intuitivo.
- **Formulários:** regras básicas para que o formulário ocupe 100% do espaço disponível.
- **Elementos de formulário:** configura o estilo para os campos de input, textarea e botões (repare que para este último definimos seu estilo separadamente, e não dentro de cada regra que tenha um botão qualquer, deixando o mesmo universal).

Como o “watcher” do Sass já foi ativado, basta salvar a página HTML e abri-la em um navegador web.

O resultado da página em um browser desktop pode ser visto na **Figura 2**, enquanto a visualização em dispositivos menores se encontra na **Figura 3**.



**Figura 2.** Tela com cabeçalho responsivo



**Figura 3.** Tela com cabeçalho responsivo em dispositivo pequeno

# Dominando Design Responsivo com HTML5 e CSS3

**Listagem 6.** Código de SCSS global do Sass.

```
//// Declarações Customizadas
// Cores
$bk: #272822;
$rl: #C03;
$g: #429032;
$b: #8cc53e;
$y: #49c5bf;
$bg: #f4f4f4;

// Media Query Mixin - Desktop-first
@Mixin paraTelasPequenas($media) {
  @media (max-width: $media/16+em) { @content; }
}

//Globais
*,
*:before,
*:after {
  box-sizing: border-box;
}

body {
  font-family: "Segoe UI", Arial, "Helvetica Neue", Helvetica, sans-serif;
  background-color: $bg;
}

h1, h2 {
  color: white;
}

blockquote {
  font-style: italic;
  @include paraTelasPequenas(420) {
    margin-left: 10px;
  }
}

// Header
.masthead {
  display: flex;
  justify-content: space-between;
  max-width: 980px;
  margin: auto;
  padding: 10px;
  background: $b;
  @include paraTelasPequenas(700) {
    display: block;
    text-align: center;
  }
}

.logo {
  padding: 10px 0 10px 10px;
  color: white;
  line-height: 1.5;
  font-size: 1.3em;
  @include paraTelasPequenas(700) {
    padding: 10px 0;
    font-size: 1.5em;
  }
  @include paraTelasPequenas(420) {
    font-size: 1.1em;
  }
}

//Placeholder

%highlight-section {
  /*border: white 1px solid;
  border-radius: 3px;
  background: rgba(white, .1);
  background: rgba(black,.2);
  border-radius: 3px;
}

// Campo de busca
.search {
  @extend %highlight-section;
  display: flex;
  align-items: center;
  justify-content: center;
  text-align: center;
  color: white;
  font-size: .9em;
  padding: 10px;
  .field {
    width: 50%;
    margin: 0 5px;
    padding: 7px;
    @include paraTelasPequenas(420) {
      width: 70%;
    }
  }
  button {
    @include paraTelasPequenas(420) {
      width: 90%;
      margin-top: 10px;
    }
  }
}

// Formulários
.field,
.comments {
  width: 100%;
  margin-bottom: 10px;
  @include paraTelasPequenas(420) {
    width: 100%;
  }
}

// Elementos de formulário
input, textarea {
  font-family: "Segoe UI", Arial, "Helvetica Neue", Helvetica, sans-serif;
  padding: 10px;
  border: none;
  background: rgba(white,.8);
  border-radius: 2px;
  box-shadow: inset 0 1px 1px rgba(black,.5);
}

button {
  border: none;
  padding: 7px 10px;
  background: #333;
  border-radius: 3px;
  color: white;
  font-family: "Segoe UI", Arial, "Helvetica Neue", Helvetica, sans-serif;
  cursor: pointer;
}
```

Vamos incluir agora o restante do conteúdo HTML da página, que se divide entre barra de navegação (`<nav>`), corpo da página (`<main>`) e rodapé (`<footer>`). Logo, inclua o conteúdo da **Listagem 7** após a declaração do cabeçalho recém-criado.

**Listagem 7.** Restante do código HTML de navegação, corpo e rodapé.

```
<nav class="main-nav" role="navigation">
  <ul class="nav-container">
    <li><a href="#">Link 1</a></li>
    <li><a href="#">Link 2</a></li>
    <li><a href="#">Link 3</a></li>
    <li><a href="#">Link 4</a></li>
  </ul>
</nav>
<main class="main-container" role="main">
  <h1>Criando conteúdo em HTML5</h1>
  <p>HTML5 é uma linguagem de marcação de hipertexto, muito utilizada na web.</p>
  <article class="article-container flex-container">
    <section class="main-content">
      <header>
        <h1>O elemento <code>&lt;main></code></h1>
      </header>
      <p>Por definição:</p>
      <blockquote>
        <p>O Elemento (<code>&lt;main></code>) representa&hellip;</p>
      </blockquote>
    </section>
    <aside class="side-content" role="complementary">
      <h2>Quer se tornar um programador?</h2>
      <p>Provavelmente você terá um longo caminho, mas pode fazer uso do nosso conteúdo para te ajudar.</p>
    </aside>
  </article>
  <div class="contact-form" role="form">
    <header>
      <h2>Mande-nos sua sugestão/dúvida/questão</h2>
    </header>
    <form>
      <div class="flex-container">
        <label class="label-col">Nome:
          <input type="text" class="field name" id="nome" required>
        </label>
        <label class="label-col">Email:
          <input type="email" class="field email" id="email" required>
        </label>
      </div>
      <label for="comentario">Comentário:</label>

      <textarea class="comments" id="comentario" cols="50" required></textarea>
      <button>Enviar</button>
    </form>
  </div>
  <footer class="main-footer" role="contentinfo">
    <p>Copyright © 2015 | Todos os direitos reservados. </p>
    <ul class="nav-container" role="navigation">
      <li><a href="#">Rodapé Link 1</a></li>
      <li><a href="#">Rodapé Link 2</a></li>
      <li><a href="#">Rodapé Link 3</a></li>
      <li><a href="#">Rodapé Link 4</a></li>
      <li><a href="#">Rodapé Link 5</a></li>
    </ul>
  </footer>
</main>
```

Nela, podemos perceber a inclusão das seguintes tags principais:

- **nav:** cria a barra de navegação com uma lista não ordenada HTML e, dentro de cada item, um link para cada opção de menu. Note que também estamos usando a *role* de “navigation” que debatemos antes.
- **main:** guarda todo o conteúdo principal da página (com a *role* de “main”), que inclui desde o título e subtítulo do post (já que simula a exibição de um artigo – *article*) e a tag `<article>` com o texto do post (no qual também inserimos um `<aside>` com *role* “complementary” para complementar o conteúdo do post).
- **form:** está inserido dentro do *main*, mas traz configurações específicas do formulário de contato (*role* “form”), com uma segunda tag `<header>` (dessa vez interna a uma seção), bem como os campos de `input`, `textarea` e botão de `submit`.
- **footer:** através da *role* “contentinfo”, o rodapé traz apenas um texto de *copyright* e mais uma lista de links de menu tal como tivemos no *header*.

Agora precisamos criar as regras Sass para lidar com o estilo dos novos componentes. Como se trata de muito conteúdo, vamos dividir em duas partes: **Listagens 8** e **9**. Na primeira temos as definições divididas da seguinte forma:

- **Navegação:** mapeia o CSS para a tag `<nav>` definindo o tamanho máximo de largura, cor de plano de fundo (veja que nesta usamos o valor hexadecimal da cor *inline*, isto é, direto no código) e exibição da seção usando o recurso do *FlexBox* (essa propriedade é nova no CSS3 e especifica o tamanho do item relativo ao resto dos itens de igual modo flexíveis que estejam dentro do mesmo container). Outro destaque se reflete na função `rgba()` do Sass definida nas propriedades `box-shadow` e `border-bottom` do link (`<a>`). Essa função recebe quatro argumentos: as cores *red*, *green* e *blue* da cor em si, e o fator *alpha*, que representa o nível de opacidade aplicado ao campo (valor deve estar entre 0 e 1).

- **Container principal:** define propriedades gerais para o corpo da página. Veja que aqui fazemos uso da variável global *&y* criada no início do arquivo para configurar a cor da `border-bottom` do mesmo. Destaque para o elemento `<h1>` que recebe suas propriedades de estilo, aplicando-as a todos os *h1*'s da página. Isso se dá através do código `&>h1` (o caractere & quer dizer “todo o documento” no Sass).

- **Article:** configura as propriedades do artigo que incluem margem, cor de fundo, `padding` e bordas.

Para a segunda listagem certifique-se de incluir seu conteúdo no final do arquivo *estilos.scss*. Nela podemos ver recursos gerais para os demais componentes da página, a saber:

- **Conteúdo lateral:** configura o estilo dos títulos de cabeçalho, links, listas ordenadas e parágrafo que possam existir no corpo do post.
- **Formulário de Contato:** cria uma espécie de delimitação ao redor da div de formulário, configurando propriedades de estilo específicas para os campos de botão, `labels`, `input`, `textarea` desse formulário.

# Dominando Design Responsivo com HTML5 e CSS3

**Listagem 8.** Código Sass para navegação, corpo e artigo.

```
// Navegação
.main-nav {
  max-width: 980px;
  margin: auto;
  padding: 10px 5px;
  border-bottom: rgba(black,.2) 1px solid;
  background: #49c5bf;
  @include paraTelasPequenas(420) {
    padding: 5px 0;
  }
}

// Todas as navegações
.nav-container {
  display: flex;
  justify-content: center;
  list-style-type: none;
  margin: 0;
  padding: 0;
  @include paraTelasPequenas(420) {
    flex-wrap: wrap;
  }
  li {
    display: flex;
    width: 100%;
    margin: 0 5px;
    text-align: center;
    @include paraTelasPequenas(420) {
      display: flex;
      justify-content: center;
      flex-basis: 45%;
      margin: 5px;
    }
  }
  a {
    @extend %highlight-section;
    display: flex;
    justify-content: center;
    align-items: center;
    width: 100%;
    padding: 10px;
    color: white;
    text-decoration: none;
    font-size: 1.3em;
    box-shadow: inset 0 1px 2px rgba(black,.2);
    border-bottom: rgba(black,.6) 1px solid;
  }
}

// Container Principal
.main-container {
  max-width: 980px;
  margin: auto;
  padding: 40px 40px 20px;
  background: #2a2a2a;
  color: rgba(white,.8);
  border-bottom: $y 40px solid;
  @include paraTelasPequenas(420) {
    padding: 20px;
    line-height: 1.5;
  }
}

// Main Headings
&>h1 {
  margin-top: 0;
  padding-bottom: 10px;
  border-bottom: $y 3px solid;
  @include paraTelasPequenas(420) {
    font-size: 1.6em;
  }
}

// Article
.article-container {
  margin-bottom: 20px;
  padding: 10px;
  background: rgba(white,.05);
  border-radius: 2px;
  border: rgba(black,.2) 10px solid;
  box-shadow: 0 5px 15px rgba(black,.3);
}

// Conteúdo principal da página
.main-content {
  width: 75%;
  margin-right: 10px;
  padding: 10px;
  @include paraTelasPequenas(600) {
    width: 100%;
  }
  h1 {
    margin: 0;
    padding-bottom: 10px;
    border-bottom: $g 3px solid;
  }
}
```

**Listagem 9.** Código Sass para conteúdo lateral, formulário de contato e rodapé.

```
// Conteúdo lateral
.side-content {
  width: 25%;
  padding: 10px;
  font-size: .8em;
  border: $b 2px dotted;
  border-left: none;
  border-right: none;
  @include paraTelasPequenas(600) {
    width: 100%;
    margin-top: 12px;
  }
  h2 {
    margin: 0;
  }
}

ol {
  padding-left: 20px;
}
a {
  color: #eee;
}
p {
  @include paraTelasPequenas(420) {
    font-size: 1.2em;
  }
}

// Formulário de Contato
.contact-form {
```

**Continuação: Listagem 9.** Código Sass para conteúdo lateral, formulário de contato e rodapé.

Estas sobrecreverão as demais criadas antes para campos de formulário.

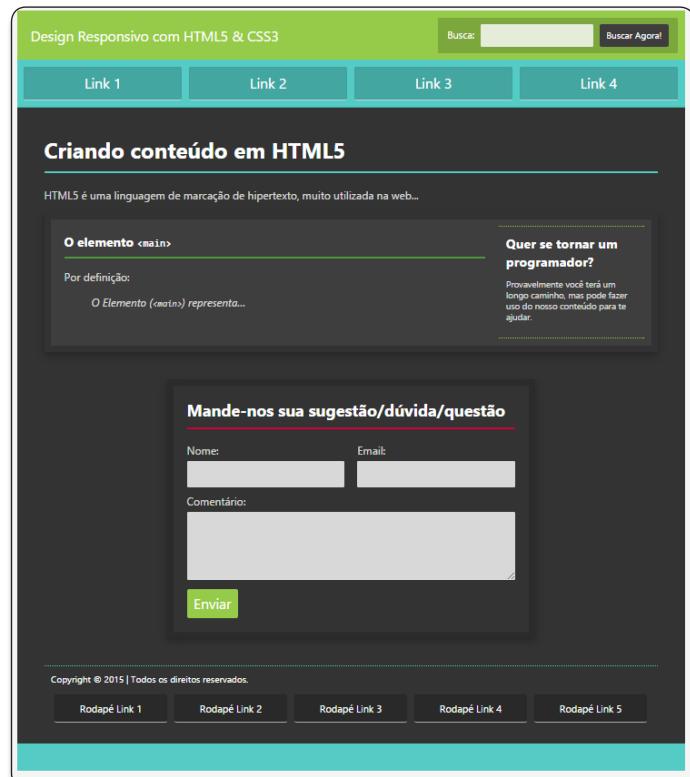
- **Rodapé:** faz uso das variáveis de cores declaradas no início do arquivo para dar cor à borda, ao texto, bem como define o espaçamento padrão do texto.

Após isso, o leitor já pode salvar todos os documentos e verificar o resultado no browser. Na **Figura 4** temos o resultado do site num navegador em modo desktop, e na **Figura 5** a mesma página visualizada a partir de um iPhone 6 Plus. Veja como os elementos se rearranjam em detrimento da diminuição de espaço disponível.

# Grids CSS

Uma grid é um conjunto de guias visuais (verticais, horizontais ou ambas) que ajudam a definir onde os elementos podem ser alocados. Uma vez que eles tenham sido alocados, alcançamos o conceito de **layout**. Uma das maiores vantagens de usar grids é que seus elementos internos terão um fluxo harmonioso ao longo das páginas, incrementando a experiência do usuário em termos de legibilidade, consistência de layout e boas proporções entre os elementos.

Muitos frameworks de componentes já usam esse conceito, como o Bootstrap por exemplo, que mantém o container com uma largura de no máximo 980px, e divide seu espaço interno em até doze divs menores de tamanhos variados.



**Figura 4.** Tela final visualizada de um browser desktop



Figura 5. Tela final visualizada de um iPhone 6 Plus

O conceito se baseia em definir para cada grid as “linhas” (*rows*) que, por sua vez, serão constituídas de colunas (*columns*) em quantidade máxima de 12. Vejamos o exemplo exposto pela **Listagem 10**. No início dela configuramos as propriedades de *box-sizing* para definir uma borda padrão ao documento como um todo. A classe “*container-12*” deve ser a primeira e mais ao topo da *div* que vai conter todo o conteúdo principal. Veja que definimos sua largura fixa em 980px (se o leitor não quiser fazer uso do *@media* para quando esta classe for executada em dispositivos menores pode definir essa largura como máxima permitida – “*max-width*” – e a largura em si (“*width*”) com o valor de 100%). Em seguida, fazemos uso do Sass para imprimir o *float* à esquerda e uma margem para todas as classes de grid, da 1 até a 12. Note que cada grid tem seu valor prefixado de acordo com as dimensões de tamanho máximo da página (de 60px – valor mínimo – até 940px – valor máximo).

Além dessa configuração feita fixamente em pixels, podemos implementar o modelo todo baseado em percentagem. Para isso, precisamos nos assegurar que a largura máxima do container também esteja em percentagem, bem como as linhas e as demais divisórias do documento. Veja na **Listagem 11** como ficaria o

**Listagem 10.** Código de exemplo para sistema de grids.

```
*, *:before, *:after { box-sizing: border-box; } // Container .container-12 { width: 980px; padding: 0 10px; margin: auto; } // Grid >> Global .grid { &-1, &-2, &-3, &-4, &-5, &-6, &-7, &-8, &-9, &-10, &-11, &-12 { float: left; margin: 0 10px; } } // Grid >> 12 colunas .container-12 { .grid-1 { width: 60px; } .grid-2 { width: 140px; } .grid-3 { width: 220px; } .grid-4 { width: 300px; } .grid-5 { width: 380px; } .grid-6 { width: 460px; } .grid-7 { width: 540px; } } .grid-8 { width: 620px; } .grid-9 { width: 700px; } .grid-10 { width: 780px; } .grid-11 { width: 860px; } .grid-12 { width: 940px; } } // Limpa elementos formatados - http://davidwalsh.name/css-clear-fix .clear, .row { &:before, &:after { content: ""; display: table; } &:after { clear: both; } } // Usa linhas para aninhar os containers .row { margin-bottom: 10px; &:last-of-type { margin-bottom: 0; } } // Legado IE .clear { zoom: 1; }
```

**Listagem 11.** Código de exemplo para sistema de grids com percentual.

```
// Container
.container-12 {
  width: 100%;
  max-width: 980px;
  padding: 0 1.02%;
  margin: auto;
}

// Grid >> Global
.grid {
  &-1, &-2, &-3, &-4, &-5, &-6, &-7, &-8, &-9, &-10, &-11, &-12 {
    float: left;
    margin: 0 1.02%;
  }
}

// Grid >> 12 colunas
.container-12 {
  .grid-1 {
    width: 6.12%;
  }
  .grid-2 {
    width: 14.29%;
  }
  .grid-3 {
    width: 22.45%;
  }
  .grid-4 {
    width: 30.61%;
  }
  .grid-5 {
    width: 38.78%;
  }
  .grid-6 {
    width: 46.94%;
  }
  .grid-7 {
    width: 55.10%;
  }
  .grid-8 {
    width: 63.27%;
  }
  .grid-9 {
    width: 71.43%;
  }
  .grid-10 {
    width: 79.59%;
  }
  .grid-11 {
    width: 87.76%;
  }
  .grid-12 {
    width: 95.92%;
  }
}

// Limpa elementos de float - http://davidwalsh.name/css-clear-fix
.clear,
.row {
  &:before, &:after {
    content: "";
    display: table;
  }
  &:after {
    clear: both;
  }
}

// Usa linhas para aninhar os containers
.row {
  margin-bottom: 10px;
  &:last-of-type {
    margin-bottom: 0;
  }
}

// Legado IE
.clear {
  zoom: 1;
}
```

mesmo código com as alterações percentuais. Para calcular esses valores, o leitor pode fazer uso de uma fórmula bem simples:  $(alvo / contexto) \times 100 = resultado\%$ . No nosso exemplo, o alvo seria o valor de cada propriedade em pixels e o contexto seria o valor máximo de largura, ou seja, 980px.

## Tipografia

A tipografia (seleção de fontes) de um site é uma das partes mais importantes da definição responsiva de um design, já que precisamos também considerar como nossos textos, títulos, subtítulos, etc. irão se comportar em diferentes cenários. Dentro dessa questão, muitas perguntas surgem como “deve-se usar pixels, ems, ou rem?”, “que scale devo aplicar?” ou “qual a melhor fonte para cada seção?”.

Dentre outras decisões importantes, a definição do tamanho base da fonte, bem como seu *ratio* precisam ser tomadas no início da criação da página, para se certificar de que as demais sigam o mesmo ritmo. Na seção **Links** você encontra a URL para o site *Modular Scale*, uma ferramenta poderosa para a definição de *ratios* e *bases* do texto com visualização em tempo real e geração de conteúdo em CSS, Sass ou JavaScript.

Veja na **Figura 6** um exemplo de definição na página. Nele selecionamos dois valores padrão para a base, em pixels, e clicamos no botão *Table* que mostra uma relação dos valores de fonte em pixels (primeira coluna), ems (segunda coluna) e o possível valor se o tamanho fosse 16px. O leitor pode ir aumentando ou diminuindo tais valores e ver como o texto se comporta até que tenha uma definição final de qual usará. O mesmo vale para o *ratio*.

Na **Listagem 12** temos um exemplo de página HTML gerada com alguns cabeçalhos de título (`<h1><h6>`) para que possamos entender a aplicação dos valores *base/ratio*. Veja que criamos um segundo arquivo de estilo SCSS para garantir que as regras não se misturem com as definidas no primeiro exemplo. Na **Listagem 13** você encontra o código Sass para aplicar o estilo desejado aos referidos componentes. Note que implementamos o mesmo conceito de função mixin para receber o valor e converter as fontes para telas maiores. No estilo do *body* definimos a base da fonte como 16px e o *ratio* como 1.4, e nos cabeçalhos de títulos os valores proporcionais apontados pela *Modular Scale*. O resultado final pode ser visualizado na **Figura 7**.

Definir um design responsivo é tarefa custosa e muitas vezes envolve vários riscos como a falta de maturidade dos profissionais envolvidos em sua construção, falta de tempo e organização para definir os escopos, falta de conhecimento sobre que tipos de websites receberão aquele estilo, bem como o não entendimento da quantidade de mudanças que esse mesmo site irá sofrer com o decorrer do tempo. Todas estas variáveis devem ser sempre

levadas em consideração na hora de implementar um design responsivo, principalmente se o mesmo for servir como framework base para futuros sites/sistemas. Outro enfoque importante nessa abordagem é o SEO. Um bom design quase sempre implica em melhorias no SEO de uma página. Todavia, se o designer/desenvolvedor não tiver experiência o suficiente para escolher os passos certos, o CSS final poderá ficar gigante e não performático.

The screenshot shows the Modular Scale website interface. On the left, there's a sidebar with input fields for 'Bases' (set to 2px) and 'Ratios' (set to 1,618). Below these are tabs for 'Web', 'Sass', and 'JS'. The main area features a table titled 'Table' with columns for 'Text' and 'px'. The table lists various font sizes in pixels and their corresponding em values, ranging from ms(6) at 8.472px to ms(-6) at 0.111px. At the bottom of the table, there are two lines of Sass code: '\$ms-base: 2px, 2000px;' and '\$ms-ratio: 1.618;'. The browser address bar shows the URL: www.modularscale.com/?2,2000&px&1.618&sass&table.

Figura 6. Definindo scales para fontes na Modular Scale

The screenshot shows a web page titled 'Design Responsivo com H'. The main heading is 'Exemplo de tipografia para Design Responsivo'. Below it is a quote by Steve Jobs: "Você não pode juntar os pontos olhando para frente; você pode conectá-los apenas olhando para trás. Então, você deve confiar que os pontos vão se conectar de alguma forma no seu futuro. Você precisa confiar em alguma coisa — nos seus colhões, destino, vida, carma, qualquer coisa. Essa abordagem nunca me deixou para baixo e fez toda a diferença na minha vida." — Steve Jobs

## Typografia - Fonte H2

Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur adipisci velit...

### Exemplo H3

Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur adipisci velit...

Figura 7. Tela com tipografia aplicada via Modular Scale

**Listagem 12.** HTML para aplicação de base/ratio na fonte.

```
<!DOCTYPE html>
<html>

<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Design Responsivo com HTML5 & CSS3</title>
<link rel="stylesheet" href="style/css/estilos2.css">
</head>

<body>
<h1>Exemplo de tipografia para Design Responsivo</h1>
<blockquote>
<p>"Você não pode juntar os pontos olhando para frente; você pode  
conectá-los apenas olhando para trás. Então, você deve confiar que os  
Pontos vão se conectar de alguma forma no seu futuro.  
Você precisa confiar em alguma coisa — nos seus colhões, destino, vida,  
carma, qualquer coisa. Essa abordagem nunca me deixou para baixo e  
fez toda a diferença na minha vida."</p>
<p>&mdash; Steve Jobs</p>
</blockquote>
<h2>Tipografia - Fonte H2</h2>
<p>Neque porro quisquam est qui dolorem ipsum quia dolor sit amet,  
consectetur, adipisci velit...</p>
<h3>Exemplo H3</h3>
<p>Neque porro quisquam est qui dolorem ipsum quia dolor sit amet,  
consectetur, adipisci velit...</p>
</body>
</html>
```

**Listagem 13.** Código Sass para aplicar base/ratio nos títulos.

```
//Mobile-first Media Query Mixin
@Mixin paraTelasGrandes($media) {
  @media (min-width: $media/16+em) {
    @content;
  }
}
body {
  font: 16px/1.4 Open Sans, Arial, "Helvetica Neue", Helvetica, sans-serif;
  @include paraTelasGrandes(640) {
    font-size: 20px;
  }
}
h1 {
  font-size: 2.454em;
}
h2 {
  font-size: 1.618em;
}
h3 {
  font-size: 1.517em;
}
```

Em relação às bibliotecas de terceiros (*third parties*) é preciso ter muito cuidado com seu uso. Por um lado, elas podem acrescentar em muito ao design e facilitar certas tarefas que demorariam muito se feitas manualmente, entretanto, por outro, podem importar uma carga muito alta de código que, em sua grande maioria, não será usada, logo, perdemos em performance mais uma vez. Tudo é uma questão de conhecimento da biblioteca e da existência de outras substitutas que possam fornecer o mesmo resultado, com melhor produtividade e usando mais recursos da mesma. O único jeito de alcançar isso é expericienciando.

**Autor****Fabrício Galdino**

É um especialista em software e trabalhou com análise de TI e desenvolvimento de negócios por cinco anos. Ter experiência com testes e tecnologias relacionadas ao front-end, como SEO, Responsividade, HTML5 e CSS3.

**Links:****Página de especificação do Ruby.**

<https://www.ruby-lang.org/pt/>

**Página oficial do LibSass.**

<http://sass-lang.com/libsass>

**Página de download do RubyInstaller.**

<http://rubyinstaller.org/>

**Documentação oficial do Sass.**

<http://sass-lang.com/documentation/>

**Página da Modular Scale.**

<http://www.modularscale.com/>

# Boas práticas de programação em JavaScript

Veja neste artigo como otimizar seu JavaScript, evitando os maus códigos e entendendo a linguagem a fundo

A linguagem de programação JavaScript foi concebida como uma linguagem de scripts que proporcionava não somente uma fácil integração entre o browser, o seu documento DOM e a HTML, como também formas de manter o código do cliente organizado, totalmente orientado a objetos (e nisto incluímos vários *design patterns*, boas práticas de codificação, APIs, etc.) e, sobretudo, reutilizável. E isto é o que todos os desenvolvedores mais buscam nos últimos tempos: módulos reusáveis, baseados em componentes e que forneçam uma interface em comum para ser acessada por diferentes clientes e alimentada por diferentes servidores.

Não é raro encontrar pelos fóruns da web ou blogs questões sobre o melhor uso do JavaScript em relação às suas estruturas. A maioria dos desenvolvedores sequer tem conhecimentos aprofundados na mesma ou ao menos entende bem o que cada implementação básica que faz, como a de uma *function*, por exemplo. Não obstante a isso, neste artigo trataremos de introduzir uma série de boas práticas que podem ser usadas para te ajudar a desenvolver código JavaScript mais performático, produtivo e limpo.

## Erros comuns

Para entender melhor como tais práticas podem ser otimizadas, vamos explorar alguns dos principais tipos de erros mais comuns ao se desenvolver código JavaScript. Essa lista é importante pois abrange conceitos tidos como ultrapassados na linguagem e que devem ser entendidos para evitar dores de cabeça futuras.

## Fique por dentro

Este artigo é útil por introduzir uma série de conceitos e boas práticas associadas ao desenvolvimento de código JavaScript. Quando se inicia em uma nova linguagem de programação é essencial conhecer a fundo os detalhes da mesma, bem como as boas práticas que a comunidade construiu com base em experiências e no velho modelo tentativa-erro. Aqui, trataremos de expor alguns dos principais problemas associados ao mau uso do JavaScript, os quais o leitor poderá usar para embasar seus projetos futuros ou criar seus próprios frameworks front-end.

## Não usar “var” para declarar suas variáveis

Quando não as declaramos com essa palavra reservada o código provavelmente irá funcionar, mas fará com que as mesmas sejam criadas em âmbito global, acumulando espaço em memória desnecessário e, possivelmente, culminando em erros nas camadas mais inferiores do código. Um dos problemas mais comuns associados a esse tipo de estratégia é chamado de “vazamento de variáveis globais”. Por exemplo, considere o código a seguir:

```
function teste() { abc = 'abc'; }
teste();
console.log('Ainda consigo ver o valor de abc: ' + abc);
```

Como a variável foi declarada dentro do escopo de um bloco (uma função), pelos termos comuns das linguagens de programação a mesma deveria ser considerada local. Entretanto, a ausência da palavra reservada *var* torna a mesma global, mesmo que ela esteja dentro de um bloco qualquer.

Neste caso, a solução consiste em fazer o devido uso da mesma possibilitando que sua visualização esteja restrita ao bloco em questão:

```
function teste() { var abc = 'abc'; }
teste();
console.log('Agora não consigo mais ver o valor de abc: ' + abc);
```

Outro exemplo clássico desse erro está na implementação da estrutura de repetição *for* do JavaScript:

```
for (i = 0; i < vetor.length; i++) { ... }
```

Neste exemplo, temos uma variável dentro de um bloco (o próprio *for*), porém como não usamos o *var* seu escopo será criado de forma global, logo se tentarmos acessar o valor final de *i* mesmo após finalizado o loop ainda conseguiremos fazê-lo. Uma simples declaração corrige todo o problema:

```
for (var i = 0; i < vetor.length; i++) { ... }
```

Para lidar melhor com este e os demais problemas que serão apresentados, o leitor pode fazer uso da ferramenta online **JSHint** (vide seção **Links**), que é uma ferramenta de análise semântica bem simples e intuitiva que te ajuda a identificar problemas mais óbvios nos seus códigos JavaScript. Veja na **Figura 1** as mensagens de validação do mesmo quanto ao problema do loop que apresentamos.

## Chamar funções de callback antes do previsto

Algumas APIs ou bibliotecas de browsers esperam receber funções como argumentos para seus principais métodos. Mas acontece que a maioria dos programadores acaba chamando a função antes que ela seja de fato enviada ao método como uma callback. Por exemplo, considere o código a seguir:

```
$(document).ready(initializarComponentes);
```

O código mostra a chamada à função **ready()** do framework jQuery que é executada logo após a completa inicialização do

documento DOM na página em questão. Esta função é largamente usada em substituição à função *onload()* do próprio documento HTML, em desuso pela comunidade. Veja que dentro dela chamamos uma função de nome *inicializarComponentes* que, por sua vez, será executada antes que a função *ready()* a reconheça. O grande segredo nesse tipo de implementação está nos parênteses, pois são eles os responsáveis por disparar a execução de qualquer função no JavaScript. No mesmo exemplo, suponha que a implementação da função *inicializarComponentes* seja a demonstrada a seguir:

```
function inicializarComponentes() {
  console.log('Inicializando...!');
}
```

Desta forma, sempre que desejarmos efetuar a execução do código interno a esta função basta chamar o seu nome acrescido dos parênteses, em qualquer lugar do seu código:

```
inicializarComponentes();
```

Já uma função de callback segue o conhecido princípio de “*Don't call me. I'll call you*” (“Não me chame. Eu o chamarei.”). Em outras palavras, as funções ditas de callback não são executadas diretamente, logo não recebem parênteses nos locais onde são chamadas. O mesmo código de inicialização do jQuery deve ficar assim, portanto:

```
$(document).ready("inicializarComponentes");
```

A esse tipo de problema damos o nome de “Invocação Prematura”. Ainda é possível também passar as funções de callback em formato de *string* em vez de *functions*. Vejamos um segundo exemplo:

```
setTimeout("inicializarComponentes()", 2000);
```

A função *setTimeout()* é responsável por executar uma determinada função (primeiro parâmetro) ao final de um certo limite de tempo estipulado (segundo parâmetro). Ao passar a função de callback entre as aspas duplas possibilitamos que a mesma fosse

```
1 // Hello.
2 //
3 // This is JSHint, a tool that helps to detect errors and potential
4 // problems in your JavaScript code.
5 //
6 // To start, simply enter some JavaScript anywhere on this page. Your
7 // report will appear on the right side.
8 //
9 // Additionally, you can toggle specific options in the Configure
10 // menu.
11
12 var vetor;
13 for (i = 0; i < vetor.length; i++) {
14
15 }
```

### CONFIGURE

One undefined variable

13 i

13 i

13 i



About  
Documentation  
Install  
Contribute  
Blog

**Figura 1.** Exemplo de análise do loop com ferramenta JSHint

traduzida automaticamente pelo JavaScript para uma *function* e, sobretudo, não fosse executada imediatamente (como aconteceria caso as aspas não fossem usadas).

Todavia, esse tipo de implementação tem suas limitações: ela só funciona para funções de callback que são esperadas como parâmetros de funções. Se precisarmos configurar uma função associada a um atributo de um objeto, o mesmo procedimento não funcionará. Por exemplo, considere que a função *inicializarComponentes()* deva ser associada ao evento *onload* do objeto *window* do DOM:

```
window.onload = "inicializarComponentes()";
```

O referido código não funcionará porque atributos esperam sempre um valor (tipo primitivo, objeto ou função). Se passarmos uma string, a mesma será ignorada por não atender à regra. Para este mesmo caso, a solução seria remover os parênteses e deixar que a *onload* se encarregue de executar a callback na hora certa.

## O escopo “this” pode não ser o que você espera

Geralmente, quando criamos uma função de callback dentro de outra função, o valor que está associado ao operador *this* (ou o contexto) muda. Esse tipo de implementação requer um conhecimento mais aprofundo no gerenciamento de contextos pelo JavaScript por parte do desenvolvedor, principalmente no que remete ao uso de cadeias de funções aninhadas. De todo modo, a melhor estratégia é sempre salvar seus escopos *this* em variáveis locais dentro da função principal e fora da função de callback.

O grande segredo é saber quem o *this* está referenciando, observando bem quando uma *function() {}* está dentro de outra. Tomemos como exemplo as funções *ready()* e *setTimeout()* vistas há pouco que, quando unidas, podem ser representadas da seguinte maneira:

```
$(document).ready(function() {
  setTimeout(function() { this.getElementById('teste') }, 2000);
});
```

Veja que no exemplo estamos tentando recuperar um elemento de id “teste” no documento HTML que selecionamos através do seletor jQuery *\$(document)*. Todavia, o acesso à variável *this* foi feito de dentro da função *setTimeout()*, logo é a esta que o *this* está referenciando e, portanto, não temos acesso ao *this* externo da função *ready()*. Uma possível solução seria migrar o trecho de código para antes da função *setTimeout()*, mas e se quisermos acessar essa propriedade dentro desta função, isto é, após um determinado *timeout* finalizar? Para isso, basta salvar o valor da variável em uma outra variável local, tal como fizemos na **Listagem 1**.

Existem algumas situações mais específicas que podem gerar confusão quando fizermos uso do *this*. Por exemplo, considere o código apresentado na **Listagem 2** onde temos a declaração de um objeto *pessoa* com somente um atributo: *nome*. Em seguida,

associamos uma função ao mesmo objeto chamada *imprimeNome()* que, quando executada, imprime uma mensagem de alerta com o nome do referido objeto. Veja que para este exemplo fazemos uso do operador *this* através de um processo que chamamos de *binding*, ou associação. Basicamente, o que o JavaScript faz é converter o *this* para *pessoa* no exemplo permitindo assim o devido acesso às suas propriedades.

### Listagem 1. Exemplo com a variável *this*.

```
$(document).ready(function() {
  var teste = this;
  setTimeout(function() {
    teste.getElementById('teste')
  }, 2000);
});
```

### Listagem 2. Exemplo de função simples com o *this*.

```
var pessoa = {nome: 'Fabricio'};

pessoa.imprimeNome = function() {
  alert('Olá, meu nome é ' + this.nome);
}

pessoa.imprimeNome();
// Abre uma janela de alerta com o texto "Olá, meu nome é Fabricio"
```

Contudo, em alguns casos essa implementação pode não trazer o resultado esperado. Se você chamar a mesma função por si só, por exemplo, seu contexto não será associado ao objeto *pessoa*, mas sim ao objeto global *window*:

```
var boasvindas = pessoa.imprimeNome;
boasvindas(); // Abre uma janela com o texto "Olá, meu nome é undefined"
```

O objeto *window*, por sua vez, não tem nenhum atributo “*nome*” em sua constituição o que fará com que a mensagem receba o *undefined* junto. Para solucionar esse tipo de problema podemos fazer uso das funções *call()* e *apply()* do JavaScript que permitem configurar qual contexto deve ser usado para cada chamada de função, seja ela de callback ou não. Dessa forma, podemos garantir que as chamadas ao *this* estarão associadas aos devidos objetos. Vejamos:

```
var pessoa = { nome: 'Fabricio' };
imprimeNome.call(pessoa); // Abre janela com o texto "Olá, meu nome é Fabricio"
imprimeNome.apply(pessoa); // Abre janela com o texto "Olá, meu nome é Fabricio"
```

Ambos os métodos estão disponíveis para todas as funções, recebendo o primeiro argumento com o contexto a ser usado. Para os casos em que a nossa função receber algum parâmetro, a chamada a estas funções deve passar o mesmo logo em sequência ao primeiro, como podemos ver na **Listagem 3**. Nela a função *imprimeNome()* agora recebe uma parâmetro *qtd* referente à quantidade de vezes que a mensagem de boas-vindas deve ser impressa no *alert*. Para a função *apply()*, especificamente, precisamos passar

a lista de parâmetros do método em um vetor, na ordem em que forem definidos na assinatura original da função.

## Uso indevido de callbacks em loops

A criação de funções de callback como *event handlers* (ouvintes de eventos de click, mouse, etc.) dentro de um loop pode não funcionar apropriadamente dependendo da variável contadora do mesmo. A solução para estes casos quase sempre está em criar uma segunda função e fazê-la funcionar como uma *closure*. Por exemplo, suponha que queremos registrar um evento de click para cada um dos botões que tivermos numa página (veja o código da **Listagem 4** para isso).

### Listagem 3. Exemplo de uso do call/apply com função que recebe parâmetro.

```
var pessoa = {nome:'Fabricio'};

pessoa.imprimeNome = function(qtde) {
  for (var i = 0; i < qtde; i++) {
    alert('Olá, meu nome é ' + this.nome);
  }
}

imprimeNome.call(pessoa, 2); // Exibe a mensagem duas vezes
imprimeNome.apply(pessoa, [4]) // Exibe a mensagem quatro vezes
```

### Listagem 4. Função que associa eventos de click para botões na página.

```
var botoes = document.getElementsByTagName('button')
for (var i = 0, len = botoes.length; i < len; i++) {
  botoes[i].addEventListener('click', function(e) {
    console.log('Você clicou no botão de nº' + i);
  }, false);
}
```

O código tem como objetivo vasculhar todos os elementos `<button>` da página e adicionar uma função anônima a cada evento de click dos mesmos que, por sua vez, imprimirá uma mensagem no console do browser informando o id do botão clicado. A lógica da implementação está correta, todavia seu resultado final não atenderá à mesma. Isso porque cada função de callback que criamos (uma para cada iteração no loop) referencia a mesma variável `i`. Em outras palavras, não será criada uma nova variável “`var i`” para cada iteração do `for`, logo a mesma será usada para associar às mensagens do console. Dessa forma, o resultado será a impressão da mensagem “Você clicou no botão de nº 4” em todas as iterações do loop (para os cenários onde temos quatro botões na página de teste).

Uma possível solução para esse problema é envolver o nosso código interno do loop em uma *outer function*, ou função externa. Uma *outer function* é executada assim que for criada e, então, é descartada. Dessa forma, o JavaScript é forçado a criar um novo escopo de variável `i` e a inicializar com o valor atual do loop. Vejamos como fazer isso na **Listagem 5**. Note que estamos passando a variável `i` agora como um parâmetro para a *outer function* no final da mesma, além de mudando o nome do argumento interno para `j` evitando assim confusões de nomenclatura.

### Listagem 5. Loop de eventos dentro de uma outer function.

```
var botoes = document.getElementsByTagName('button');
for (var i = 0, len = botoes.length; i < len; i++){
  ifunction outer(j){
    botoes[i].addEventListener('click', function inner(e){
      console.log('Você clicou no botão de nº' + j);
    }, false);
  }(i);
}
```

## Processamento de tarefas intensivas sem bloqueio do browser

Mais cedo ou mais tarde, casos em que precisamos processar uma grande quantidade de dados via JavaScript aparecerão e exigirão um esforço grande do browser que, por sua vez, pode travar ou não apresentar um tempo de resposta muito bom. Entretanto, ao contrário de outras linguagens de programação *server side*, o JavaScript não tem uma função equivalente à `sleep(1000)` para pausar a execução por um certo período de tempo. A solução, neste caso, consiste em criar filas de tarefas em conjunto com a programação assíncrona via função `setTimeout`. Os primeiros passos consistem em:

- Otimizar o loop para que o mesmo execute em menos de 100 milissegundos: o tempo máximo de resposta para não afetar a experiência do usuário;
- Deixar a maior parte do processamento a cargo do servidor;
- Usar *webworkers*, uma forma simples de executar scripts em threads de fundo sem interferir na interface do usuário. A vantagem é que eles permitem, inclusive, efetuar operações de I/O usando o objeto XMLHttpRequest e se comunicar com o código JavaScript que o criou via mensageria. Porém, nem todos os browsers oferecem suporte a este tipo de recurso, além de o mesmo não estar habilitado a acessar propriedades do objeto DOM;
- Ponha códigos de espera dentro do corpo do loop de forma a permitir que o mesmo respire a cada iteração.

Com base nisso, uma possível solução para o problema seria:

```
for (var i = 0; i < vetor.length; i++){
  setTimeout(processarlens(vetor[i]), 20);
}
```

Há dois problemas nessa implementação: o primeiro se refere às chamadas de callbacks dentro de loops conforme vimos antes; e o segundo se refere ao intervalo de tempo estipulado entre a execução de cada função, isto é, em vez de configurar um intervalo real entre cada chamada, nosso código irá agendar uma lista de *jobs* (um para cada item) para serem executados na fila de eventos todos de uma vez. Assim, nossa fila de eventos terá de trabalhar duro do mesmo modo que teria se não tivéssemos usado a `setTimeout`. Vejamos a solução exibida na **Listagem 6**. O segredo está em remover a estratégia do loop com o `for` e focar no estilo assíncrono da implementação. Na primeira linha criamos uma cópia do vetor uma vez que iremos modificá-lo.

# Boas práticas de programação em JavaScript

Em seguida, dentro da função `processarProxItem()` recuperaremos um por um os elementos do mesmo vetor via método `shift()` dos `arrays` em JavaScript. Se ele for diferente de `null`, o enviamos para ser processado pela função específica e chamamos, em seguida, o método `setTimeout` para dar ao browser uma lacuna de tempo de 10 milissegundos, assim o efeito todo é feito até o fim usando somente recursividade.

## Listagem 6. Solução para loop sem o for.

```
var fila = items.slice(0);

function processarProxItem(){
    var proxItem = fila.shift();
    if (proxItem){
        processarItem(proxItem);
        setTimeout(processarProxItem, 10);
    }
}
processarProxItem();
```

A abordagem melhorou, já que agora o browser não irá mais travar e o usuário pode interagir com outras ações da página normalmente. Porém, ela será 10 vezes mais demorada que a abordagem anterior, pois adiciona um timeout de dez milissegundos ao tempo total de cada execução. Uma forma de melhorar esse tempo de processamento é fazendo uso dos recursos dos *batches* que, por sua vez, executam várias tarefas ao mesmo tempo em blocos aumentando assim a performance do código. Vejamos o código da **Listagem 7**. Note que a estrutura não muda muito, com exceção do loop `while` que configuramos. Sua execução será baseada no tempo de início do processamento que tomará no máximo os 100 milissegundos que definimos antes para resposta máxima ao usuário. Assim, configuramos a execução de vários blocos ao mesmo tempo em vez de somente um por vez.

## Listagem 7. Solução para loop com recurso de batches.

```
var fila = items.slice(0);

function processarProxBatch(){
    var proxItem, tempolnicio = +new Date;
    while(tempolnicio + 100 >= +new Date) {
        proxItem = fila.shift();
        if (!proxItem) return;
        processarItem(proxItem);
    }
    setTimeout(processarProxBatch, 10);
}
processarProxBatch();
```

## Programando sem um compilador

Para os desenvolvedores que estão acostumados com linguagens de programação compiláveis como Java, C# ou C++, a natureza fracamente tipificada do JavaScript pode trazer muitas questões e eventuais frustrações. Dentre as várias reclamações sobre a mesma, temos:

- O JavaScript não captura erros óbvios como incompatibilidade de tipos em parâmetros de funções ou nomes de variáveis/funções digitados de forma incorreta;
- Algumas bibliotecas populares, como o jQuery, silenciosamente ignoram entradas de dados erradas em vez de lançar exceções/erros.

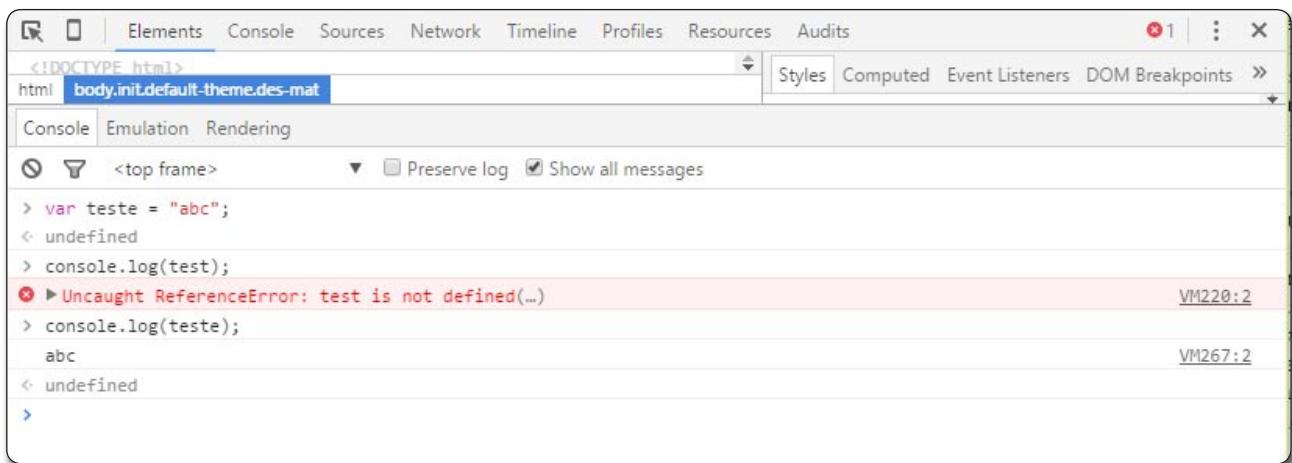
A grande questão é saber que linguagens dinâmicas têm abordagens diferentes das linguagens compiladas e seus usuários tendem a usar diferentes padrões de uso, em consequência. Se estiver programando JavaScript do mesmo modo que fazia com Java, então provavelmente você estará fazendo isso errado. Apesar do processo de compilação ser deveras mais lento, ele pode encontrar erros usando a checagem de tipos, por exemplo, e irá parar a execução do código neste exato momento, com uma mensagem intuitiva explicando o porquê do erro e como o desenvolvedor pode resolvê-lo.

As linguagens dinâmicas como o JavaScript, por outro lado, são interpretadas, isto é, precisam ter seu código executado para verificar os erros do programa. Essa é uma desvantagem em relação às linguagens com sintaxe de erros bem definida e processada em tempo de compilação, já que se assegura que os erros não escapem e atinjam a interface do usuário (UI).

A melhor forma de lidar com isso é através dos Consoles Interativos, comumente chamados de Consoles REPL (de *read-eval-print-loop*), que nada mais são que aqueles disponibilizados nos browsers através de ferramentas do desenvolvedor (no Firefox temos o FireBug, no Chrome a ferramenta proprietária, etc.). Elas funcionam como ferramentas de linha de comando para Unix ou Windows: você digita o comando, pressiona `enter`, ela executa o comando, imprime o resultado e então você faz tudo novamente. Além disso, elas são ótimas para analisar bits de código em pequenas quantidades que apresentam dificuldade para funcionar devidamente, como em expressões regulares ou para examinar o DOM na página usando a API DOM. Também funcionam muito bem para trabalhar com bibliotecas de terceiros como jQuery, por exemplo. Veja na **Figura 2** um exemplo de debug feito na Ferramenta do Desenvolvedor do Google Chrome (via atalho F12), onde criamos uma variável de nome “teste” e depois tentamos imprimir no console com um nome diferente. A ferramenta captura a exceção e imprime uma mensagem intuitiva explicando o que ocorreu: *test is not defined*. Em seguida, efetuamos a impressão com o nome correto e a operação acontece sem erros.

## Objetos e Herança

Enquanto alguns usuários de JavaScript nunca conhecerão sobre prototypes ou a natureza orientada a objetos da linguagem, aqueles que vieram do tradicional modelo OO de programação certamente ficarão confusos quanto à forma como implementamos herança no JavaScript. Talvez a maior confusão venha por parte dos frameworks JS que têm seus próprios *helpers* para escrever código baseado no modelo de classes e, assim, chegamos a seguinte conclusão: não existe apenas um jeito de fazer isso.



**Figura 2.** Exemplo de debug com Console Interativo no Chrome

Além disso, para complementar, os desenvolvedores não entendem por completo os conceitos tão a fundo quanto deveriam.

A essência da herança via prototypes é bem simples e se baseia em alguns conceitos também básicos:

- Um objeto *a* pode herdar de um outro objeto *b*. *b*, por sua vez, é dito protótipo de *a*.
- *a* herda todas as propriedades de *b*, isto é, se o valor *b.nome* for igual a “*devmedia*”, então *a.nome* terá o mesmo valor automaticamente.
- As propriedades nativas de *a* devem sobrepor as de *b*.

Consideremos um exemplo de objeto anônimo de tipo Pessoa e que contenha dois atributos: um *nome* e um *sobrenome*, tal como na **Listagem 8**. Veja que definimos o mesmo atributo de nome para ambos os objetos, porém o segundo não consta de nenhum sobrenome. Todavia, estamos estendendo o segundo objeto do primeiro via operador `__proto__`, propriedade interna a todos os objetos no JavaScript que serve exatamente para definir a herança dos mesmos. Assim, quando acessarmos a propriedade sobrenome do segundo objeto o valor “Galdino” estará automaticamente associado à mesma.

#### **Listagem 8.** Exemplo de herança via prototypes.

```

var fabricio = {
  nome: 'Fabricio',
  sobrenome: 'Galdino'
}
var fabricio_filha = {nome: 'Joana'}
fabricio_filha.__proto__ = fabricio;

```

Supondo que, no exemplo, Joana se case e receba um novo sobrenome. Logo, teríamos o seguinte trecho de código:

```
fabrício_filha.sobrenome = 'Souza';
```

E o valor seria então sobreescrito no objeto *fabricio\_filha*. Caso Joana se divorcie e deseje remover o seu sobrenome para voltar ao

antigo, basta remover a propriedade e a herança se encarregará de fazer a associação automática com o elemento pai novamente:

```
delete fabrício_filha.sobrenome;
```

A implementação parece bem simples, mas temos um problema em relação a ela: não podemos usar o operador `__proto__` ao menos não poderemos em um futuro breve. Isso porque esse recurso não é suportado no Internet Explorer e sequer está presente na especificação da ECMAScript. O navegador Firefox já considera, inclusive, a sua remoção em versões futuras. Bem, então qual estratégia usar, já que o JavaScript não tem nenhum recurso de classes que vemos em outras linguagens? A solução são os **prototypes**.

Em termos de programação genérica, um protótipo é um objeto que fornece um comportamento base para um segundo objeto. O segundo objeto pode então estender esse comportamento base para formar sua própria especialização. Este processo, também conhecido como *herança diferencial*, difere da clássica herança na medida em que não exige uma tipificação explícita (estática ou dinâmica) ou tenta definir formalmente um tipo em função de outro. Enquanto a herança clássica tem reutilização planejada, a verdadeira herança prototípica é oportunista.

Em JavaScript, cada objeto faz referência a um objeto protótipo a partir do qual ele pode herdar propriedades. Protótipos JavaScript são excelentes instrumentos para reuso: uma única instância de protótipo pode definir propriedades para um número infinito de instâncias dependentes. Protótipos também podem herdar de outros protótipos, formando cadeias de protótipos.

Porém, em comparação com a emulação no Java, o JavaScript amarra a propriedade *prototype* ao construtor. Como consequência, mais frequentemente do que não, objetos de vários níveis de herança são alcançados através do encadeamento de protótipos baseados em construtores.

No fim, o encadeamento de protótipos baseados em construtores requer planejamento inicial e resulta em estruturas que se asse-

melham mais de perto às hierarquias tradicionais de linguagens clássicas: construtores representam tipos (classes), cada tipo é definido como um subtipo de um (e apenas um) supertipo e todas as propriedades são herdadas através desse tipo de cadeia. A palavra-chave *class* meramente formaliza a semântica existente. Deixando de lado todas essas características sintáticas, o JavaScript tradicional é claramente menos prototipável do que alguns afirmam.

Numa tentativa de oferecer um maior suporte aos protótipos, a especificação ES5 da ECMAScript introduziu o *Object.create*. Este método permite que um protótipo seja atribuído a um objeto diretamente e, portanto, liberta os protótipos JavaScript de construtores de modo que, em teoria, um objeto possa adquirir comportamento a partir de qualquer outro objeto arbitrário e estar livre das restrições do *typecasting* (conversão dos tipos). Vejamos na **Listagem 9** um exemplo de objeto clássico de círculo com suas respectivas funções.

O *Object.create* aceita um segundo argumento opcional que representa o objeto a ser herdado. Infelizmente, em vez de aceitar o objeto em si (na forma de um literal, variável ou argumento), o método espera uma definição completa da propriedade *meta*. Por exemplo, se quisermos definir a propriedade *radius* do exemplo anterior no objeto pai, precisaríamos instanciar o mesmo da forma como está definida na **Listagem 10**.

Partindo do princípio de que ninguém realmente usa esse tipo de código no mundo real, tudo o que resta é atribuir manualmente as propriedades para a instância depois de ter sido criada. Mesmo assim, a sintaxe *Object.create* só permite que um objeto possa herdar as propriedades de um protótipo.

## Listagem 9. Exemplo de objeto via *Object.create()*.

```
var circulo = Object.create({
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  crescer: function() {
    this.radius++;
  },
  encolher: function() {
    this.radius--;
  }
});
```

## Listagem 10. Exemplo de herança via *Object.create()*.

```
var circulo = Object.create({
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  crescer: function() {
    this.radius++;
  },
  encolher: function() {
    this.radius--;
  },
  radius: {
    writable:true, configurable:true, value: 7
  }
});
```

Em cenários reais, muitas vezes queremos adquirir o comportamento do protótipo a partir de múltiplos objetos: por exemplo, uma pessoa pode ser um empregado e um gerente ao mesmo tempo.

## Mixins

Felizmente, o JavaScript oferece alternativas viáveis para o encadeamento de heranças. Em contraste aos objetos das linguagens mais rigidamente estruturadas, os objetos JavaScript podem chamar qualquer propriedade das funções, independentemente da linhagem. Em outras palavras, as funções JavaScript não precisam ser hereditárias para serem visíveis.

A abordagem mais básica para o reuso de funções é a delegação manual – qualquer função pública pode ser chamada diretamente através das funções *call* ou *apply* que vimos antes. É um recurso poderoso e facilmente esquecido.

Tradicionalmente, um **mixin** é uma classe que define um conjunto de funções que seriam, em outros cenários, definidas por uma entidade concreta (uma pessoa, um círculo, um observador). Entretanto, as classes mixin são consideradas abstratas já que não serão instanciadas por si sós, em vez disso suas funções são copiadas (ou emprestadas) por classes concretas como uma forma de herdar comportamento sem ter de entrar em um relacionamento formal com o fornecedor do mesmo comportamento. No fim, temos uma funcionalidade interessante que nos permite usar objetos (instâncias) que oferecem clareza e flexibilidade: nosso mixin pode ser um objeto regular, um protótipo, uma função, etc. seja qual for, o processo torna-se transparente e óbvio.

Tomando como referência o mesmo exemplo de classe Círculo que criamos, vejamos como transformar nosso código de herança em um mixin (**Listagem 11**). Isto é o mais próximo que podemos chegar ao modelo de classes no JavaScript. Para tanto, o uso da propriedade *prototype* se faz essencial para permitir a definição da estrutura hierárquica no referido objeto.

## Listagem 11. Exemplo de herança com círculo via mixin.

```
var Circulo = function() {};
Circulo.prototype = {
  area: function() {
    return Math.PI * this.radius * this.radius;
  },
  crescer: function() {
    this.radius++;
  },
  encolher: function() {
    this.radius--;
  }
};
```

Também é possível simplificar essa chamada simplesmente declarando o objeto como uma variável via palavra-chave *var*. Assim, nosso código pode ficar ainda mais simples. Veja na **Listagem 12** um exemplo de classe que mapeia os possíveis eventos de um botão.

Como é possível então que um objeto mixin se integre ao seu objeto? Por meio de uma função *extend* que implementa a herança

associando-a aos dois objetos em questão. Normalmente, a função `extend` simplesmente copia (não clona) as funções mixin para o objeto receptor. Existem algumas pequenas variações nesta implementação dependendo do desenvolvedor/empresa que a emprega. Por exemplo, o framework *Prototype.js* omite uma checagem à propriedade `hasOwn` do objeto *Property* sugerindo que o mixin não tem propriedades enumeráveis em sua cadeia de protótipos), enquanto outras versões supõem que o desenvolvedor deseja copiar somente os mixin de protótipo do objeto. Vejamos na **Listagem 13** um exemplo comum dessa função. Note que passamos ambos os objetos de destino e fonte (este último deve ser um *array* em vista da quantidade de objetos filhos que desejam herdar do objeto pai) como parâmetros à função que, por sua vez, se encarrega de iterar sobre a lista de fontes verificando se cada uma tem a propriedade de destino interna ao seu conteúdo (via função JavaScript `hasOwnProperty`) e adicionado a mesma em caso positivo.

**Listagem 12.** Exemplo de mixin simplificado.

```
var clickableFunctions = {
  hover: function() {
    console.log('hover');
  },
  pressionar: function() {
    console.log('pressionando botão');
  },
  soltar: function() {
    console.log('soltando botão');
  },
  disparar: function() {
    this.action.fire();
  }
};
```

**Listagem 13.** Exemplo da função de mixin `extend()`.

```
function extend(destino, fonte) {
  for (var chave in fonte) {
    if (fonte.hasOwnProperty(chave)) {
      destino[chave] = fonte[chave];
    }
  }
  return destino;
}
```

Agora podemos estender um protótipo base a partir dos dois mixins que criamos anteriormente para criar um botão redondo, por exemplo. Vejamos o código da **Listagem 14**. Note que criamos apenas um objeto simples com duas propriedades: uma dimensão de raio do botão e uma label, em seguida, chamamos a função `extend` para associar as funcionalidades implementadas na *Circulo* e *clickableFunctions*, respectivamente. No fim, fazemos um teste básico instanciando um objeto de mesmo tipo e chamando alguns de seus métodos, de forma muito semelhante a que temos em outras linguagens OO.

## Padrões e Modelos

Os frameworks MVC – ou MVW (*Model, View, "Whatever"*) – existem nas mais diversas formas e tipos pela comunidade front-end.

Todavia, apesar de duas diferentes abordagens que dificultam a componentização e o aproveitamento das estruturas modulares, eles nos fornecem componentes fundamentais para um código padronizado: os **modelos**, que “modelam” os dados associados à aplicação. Nas aplicações web baseadas em um cliente, tais modelos geralmente são representados por um objeto de banco de dados hospedado no servidor.

**Listagem 14.** Exemplo de herança com dois mixins criados.

```
var BotaoCircular = function(radius, label) {
  this.radius = radius;
  this.label = label;
};

extend(BotaoCircular.prototype, Circulo);
extend(BotaoCircular.prototype, clickableFunctions);

var botaoCircular = new BotaoCircular(3, 'Enviar');
botaoCircular.crescer(); // Aumenta em 1 o valor da propriedade radius
botaoCircular.disparar(); // Dispara o evento no botão
```

Um bom exemplo disso é o framework MVC minimalista Backbone.js que, apesar de constantemente criticado pela sua camada de visão nada sofisticada, fornece um excelente gerenciamento interno dos modelos. Por exemplo, na **Listagem 15** vemos um exemplo simples de modelo no Backbone. Veja que ele aplica o mesmo conceito de função `extend` que vimos anteriormente, além de usar uma notação de objetos igual à do JSON. Podemos ver também a distribuição uniforme dos atributos com seus respectivos valores iniciais, assim como o atributo externo *idAtributo* que contém o valor do id desse elemento no DOM. A função *nomeCompleto*, por sua vez, retorna os valores concatenados do nome e sobrenome do usuário via variável *this*, conforme também havíamos visto.

**Listagem 15.** Exemplo de modelo definido no framework Backbone.

```
var Usuario = Backbone.Model.extend({
  defaults: {
    usuario: '',
    nome: '',
    sobrenome: ''
  },
  idAtributo: 'usuario',
  nomeCompleto: function () {
    return this.get('nome') + this.get('sobrenome');
  }
});
```

Na **Listagem 16** vemos um código de exemplo que faz uso desse modelo, inicializando-o, além de mostrar como a sua instância deve ser usada na aplicação. Veja que estamos apenas criando um novo objeto de tipo *Usuario* e inicializando suas propriedades para, em seguida, modificar uma delas e analisar como o framework reage a isso. Apesar de simples, esse exemplo é o suficiente para entender como os modelos *cliente-side* funcionam e como eles interagem com os moldes de aplicações MVC.

## Listagem 16. Exemplo de uso do modelo Backbone definido.

```
var usuario = new Usuario({  
    usuario:'fabricio_galdino',  
    nome:'Fabricio',  
    sobrenome:'Galdino'  
});  
  
usuario.nomeCompleto(); // Retorna "Fabricio Galdino"  
usuario.set('nome','João');  
usuario.save(); // Envia as mudanças para o endpoint no servidor
```

Adicionalmente, o Backbone fornece as chamadas classes “collection”, as quais auxiliam os desenvolvedores a manipular facilmente conjuntos de instâncias de modelos comuns. Podemos pensar nelas como espécies de arrays superdotados, carregadas de funções utilitárias, tal como exibido na **Listagem 17**. Veja como é simples criar um tipo de dado a partir de outro, herdando automaticamente todas as suas características. Como se trata de um modelo Backbone baseado no servidor, precisamos referenciar tanto o modelo de entidade (*Usuario*) quanto a URL onde o mesmo está hospedado. O método *fetch()* se encarrega de buscar todos os dados via *request* HTTP, enquanto o método *get()* busca um usuário em específico pelo seu atributo *id*.

## Listagem 17. Exemplo de uso das collections do Backbone.

```
var UsuarioCollection = Backbone.Collection.extend({  
    model: Usuario,  
    url: '/usuarios'  
});  
  
var usuarios = new UsuarioCollection();  
usuarios.fetch(); // Carrega os dados do usuário via HTTP  
var fabricio = usuarios.get('fabricio_galdino'); // Encontra pelo idAtributo
```

Nem todos os frameworks MVC implementam uma classe *Collection* como o Backbone. Por exemplo, o Ember.js define uma classe *CollectionView* que similarmente mantém um conjunto de modelos comuns, mas amarra a manipulação ao objeto DOM. Independente do framework adotado, é sabido que precisamos manipular não somente objetos, classes e suas heranças, como também coleções destes mesmos objetos. E para isso, o uso de um framework adequado pode facilitar muito esse tipo de implementação.

Quando trabalhamos com aplicações de porte grande ou até médio, é comum ter múltiplas instâncias de modelos representando alguns objetos de banco do servidor. Isso geralmente acontece quando você tem múltiplas *views* de algum dado, de modo que um modelo aparece em duas ou mais *views*.

Considere o seguinte exemplo, que introduz duas novas coleções de usuários: *Seguidores*, para usuários que estão seguindo um dado usuário (em uma rede social, por exemplo) e *Seguidos*, para os usuários que estão sendo seguidos por outros. Um usuário que é tanto seguidor quanto está sendo seguido aparecerá em ambas as coleções, neste caso teremos instâncias duplicadas do mesmo modelo. Vejamos o código na **Listagem 18**.

## Listagem 18. Exemplo Seguidores/Seguidos no Backbone.

```
var SeguidosCollection = UserCollection.extend({  
    url: '/segundo'  
});  
  
var SeguidoresCollection = UserCollection.extend({  
    url: '/seguidores'  
});  
  
var seguindo = new SeguidosCollection();  
var seguidores = new SeguidoresCollection();  
  
seguindo.fetch();  
seguidores.fetch();  
  
var usuario1 = seguindo.get('fabriciogaldino');  
var usuario2 = seguidores.get('fabriciogaldino');  
usuario1 === usuario2; // false
```

Ter múltiplas instâncias de um mesmo modelo traz duas desvantagens principais. A primeira é que assim estamos usando memória adicional para representar o mesmo objeto. Dependendo da complexidade do modelo e do tamanho dos atributos que o mesmo contiver, pode não ser interessante consumir kilobytes de memória extra para tal. Se as instâncias forem duplicadas dezenas ou centenas de vezes (um cenário bem possível tratando-se de uma aplicação com alta escalabilidade) elas podem rapidamente consumir toda a memória disponível. A segunda é que se os usuários modificarem um destes modelos no cliente, outras instâncias dos mesmos serão dessincronizadas. Isso pode ser feito de várias formas: como através do usuário mudando o estado do objeto via UI, ou via *update* criado por outro usuário e enviado ao cliente via serviço em tempo real:

```
usuario1.set('nome','João');  
usuario2.get('nome'); // ainda será João
```

Nesse mesmo exemplo onde o mesmo usuário aparece em duas coleções diferentes, torna-se trivial a atualização de ambas as instâncias de forma manual com a nova propriedade. Entretanto, em aplicações reais com vários usuários ao mesmo tempo, é inimaginável a quantidade de vezes que esse mesmo objeto de usuário possa aparecer em dezenas ou centenas de coleções diferentes.

Em função disso, uma solução comum para lidar com a duplicata de instâncias é fazer uso de uma função de fábrica quando criarmos uma nova instância do modelo. Se a fábrica detectar que uma instância do modelo já existe, ela a retornará em vez de criar uma nova. Vejamos o código da **Listagem 19**. Note como é simples definir um objeto em nível global de cache e salvar todas as novas instâncias no mesmo, assim sempre que formos criar um novo objeto do tipo *Usuario*, chamamos a classe *UsuarioFabrica* para lidar com a verificação do cache e respectivo retorno do objeto cacheado ou criação de um novo se for o caso. Toda a checagem gira em torno do atributo *usuario* que é passado por parâmetro ao construtor da fábrica.

#### Listagem 19. Exemplo de fábrica de usuários.

```
var cacheUsuario = {};  
  
function UsuarioFabrica(attrs, options) {  
    var usuario = attrs.usuario;  
    return cacheUsuario[usuario] ? cacheUsuario[usuario] : new Usuario(attrs, options);  
}  
  
var usuario1 = UsuarioFabrica({ usuario: 'fabriciogaldino' });  
var usuario2 = UsuarioFabrica({ usuario: 'fabriciogaldino' });  
usuario1 === usuario2; // true
```

Para fazer um uso eficiente deste padrão é sempre aconselhado usar a função de fábrica em conjunto para criar novas instâncias. Todavia, esse tipo de abordagem é dificultado quando precisamos implementá-la em bases de código que não são de nossa responsabilidade, como em bibliotecas de terceiros e plugins, por exemplo.

Considere, por exemplo, a função `Collection.prototype._preparseModel` do código fonte do Backbone. Ele usa esta função para “preparar” e, no fim, criar uma nova instância do modelo para adicionar à coleção. Ela é invocada para atender a uma série de necessidades, como quando precisamos popular uma coleção com modelos retornados de um recurso HTTP e seu código é apresentado na [Listagem 20](#).

#### Listagem 20. Exemplo de uso da função `prepareModel` do Backbone.

```
// Prepara o hash de atributos (ou outros modelos) para serem adicionados a esta  
// collection.  
Backbone.Collection.prototype._preparseModel = function(attrs, options) {  
    if (attrs instanceof Model) {  
        if (!attrs.collection) attrs.collection = this;  
        return attrs;  
    }  
  
    options || (options = {});  
    options.collection = this;  
    var modelo = new this.model(attrs, options);  
    if (!modelo._validate(attrs, options)) {  
        this.trigger('invalid', this, attrs, options);  
        return false;  
    }  
    return modelo;  
};
```

Atente para o código que faz uso do operador `new` para criar um novo modelo passando os *atributos* e *options* como parâmetros. É ele o responsável por criar a nova instância e associar a mesma à coleção. `this.model` é uma referência ao construtor da classe de modelo que a coleção encapsula. Ele é especificado quando definimos uma nova classe de coleção, tal como:

```
var UsuarioCollection = Backbone.Collection.extend({  
    model: Usuario, url: '/usuarios'  
});
```

O mais interessante nessa abordagem é que em vez de passarmos a classe `Usuario` à definição da coleção, podemos passar a classe `UsuarioFabrica` (a nossa função de fábrica que retorna instâncias de modelo únicas):

```
var UsuarioCollection = Backbone.Collection.extend({  
    model: UsuarioFabrica, url: '/usuarios'  
});
```

Dessa forma, a `UsuarioFabrica` será atribuída ao `this.model` e será invocada pelo operador `new` quando a collection criar uma nova instância:

```
var modelo = new this.model(attrs, options); // this.model é a UserFactory
```

Várias são as técnicas e estratégias que o leitor pode seguir para implementar código em JavaScript. A maioria dos desenvolvedores seguem a sua própria especificação e ignoram muitos conceitos já explanados por outros profissionais que otimizam o uso da linguagem como um todo. A melhor maneira de se aprofundar nestes conceitos é entendendo como o JavaScript funciona de fato, seus principais conceitos e como a orientação a objetos se aplica nessa linguagem em contraste às demais.

A especificação da ECMAScript (marca registrada que especifica linguagens para *client-scripting* na web como JavaScript, JScript e ActionScript) também traz uma série de informações importantes que, além de instruir sobre as melhores práticas associadas ao JavaScript, te manterá atualizado sobre as últimas novidades, padrões e regras da linguagem. O resto é experiência traduzida em prática.

#### Autor



#### Fabrício Galdino

É um especialista em software e trabalhou com análise de TI e desenvolvimento de negócios por cinco anos. Ter experiência com testes e tecnologias relacionadas ao front-end, como SEO, Responsividade, HTML5 e CSS3.



#### Links:

##### Página da ferramenta JSHint.

<http://jshint.com/>

##### Página oficial da especificação ECMAScript.

<http://www.ecma-international.org/publications/standards/Ecma-262.htm>

# Desenvolvimento de jogos web com Pixi.js – Parte 1

Crie jogos e animações gráficas para a web usando a API JavaScript do Pixi.js com HTML5

ESTE ARTIGO FAZ PARTE DE UM CURSO

**D**esenvolver jogos para a web não é uma tarefa fácil. Você provavelmente já ouviu falar no Blender, ferramenta de manipulação e geração de gráficos e animações 2D e 3D extremamente usado por profissionais de games. Um dos maiores desafios de quem a usa ou quer se beneficiar de seus mecanismos, sua engine de renderização, seu forte suporte recebido de uma comunidade aberta e, sobretudo, sua gratuidade, é encontrar um framework de programação web baseado em JavaScript que possibilite uma fácil integração com o Blender. E é sobre isso que vamos falar neste artigo.

Quando se trata de iniciar no universo de jogos para a web, dispositivos móveis e até mesmo desktops, uma das melhores opções para isso é a biblioteca Pixi.js. Ela é extremamente leve e rápida e pode ser usada não somente para o desenvolvimento de jogos como de quaisquer tipos de softwares interativos, usando somente JavaScript e HTML5. E é por isso que precisamos ter conhecimentos nestas duas linguagens como pré-requisito para prosseguir nesse artigo.

O Pixi nos dá ainda a possibilidade de trabalhar com grafos de cenas, isto é, hierarquias de *sprites* (imagens interativas) aninhados, além de permitir atrelar eventos de mouse e click diretamente aos mesmos.

Neste artigo trataremos de explanar acerca desse framework, seus principais conceitos, componentes e como criar seu ambiente de desenvolvimento de jogos

## Fique por dentro

Este artigo é útil por explorar os principais conceitos na prática acerca da biblioteca de desenvolvimento de gráficos 2D Pixi.js. Essa biblioteca ficou famosa por sua capacidade de abstrair o WebGL (padrão baseado no OpenGL ES 2.0 que fornece interfaces de programação de gráficos em 3D) de forma rápida e extremamente leve em comparação com outras ferramentas do tipo. O leitor aprenderá a configurar seu ambiente, extrair os arquivos necessários, inicializar a biblioteca, bem como fazer uso de seus principais recursos: criação de cenários, mapeamento de imagens, tilesets, sprites, dinamização do estilo dos componentes, rotação, scale, dentre outros.

do zero, explorando os recursos ricos que a web fornece quando bem usada. Veremos como mapear tilesets (conjunto de imagens acopladas em uma imagem só, usadas para facilitar o manuseio de desenháveis nos jogos e evitar ter de usar inúmeras imagens ao mesmo tempo), imagens individuais, cenários, plano de fundo, estilização do design de elementos, bem como as principais funções e objetos da API do Pixi.js. Na seção **Links** você encontra a URL oficial do projeto.

## Mãos à obra

Antes de partirmos para o download e configuração do Pixi.js, é importante que o leitor saiba que ele não fará todo o trabalho sozinho. Precisaremos de algumas bibliotecas auxiliares que serão estendidas pelo framework para absorver determinadas funcionalidades como a detecção de colisões de objetos nos jogos, por exemplo. Além disso, também precisamos ter um ambiente de servidor funcional para executar algum código de suporte ao projeto cliente. A melhor opção para isso é fazer uso do Node.js,

um framework robusto e fácil de usar que nos possibilita executar código JavaScript no servidor, além de gerenciar inúmeros pacotes de dependências para adicionar funcionalidades aos projetos. Mas o leitor pode ficar à vontade para usar qualquer outra tecnologia de servidor que lhe melhor convir.

Acesse a seção **Links** e baixe o instalador do Node.js para o seu Sistema Operacional. Verifique de antemão se você já não tem o software instalado na sua máquina executando o seguinte comando na interface de linha de comando:

```
node -v
```

Isso será o suficiente para exibir a versão do Node.js instalado na máquina. Caso contrário, você receberá uma mensagem do tipo “*“node” não é reconhecido como um comando interno*” no console. Além disso, verifique se a versão é a mais recente comparando-a com a apresentada no site oficial. É importante sempre trabalhar com os últimos recursos do Node.js para evitar conflitos com as atualizações que a sua aplicação for receber no futuro. A atualização neste caso também se dará via download e instalação do último executável.

Após concluída a instalação, feche a janela de comandos e abra-a novamente, executando o mesmo comando para verificar se o processo foi feito com sucesso.

Em seguida, precisamos instalar também o pacote *http-server* do Node.js que será responsável por criar um servidor HTTP local e servir nossas aplicações futuras em um browser qualquer. Portanto, execute o seguinte comando no terminal:

```
npm install http-server -g
```

Esse comando faz uso do utilitário *npm* do Node.js. Trata-se de uma extensão responsável por gerenciar as extensões do framework, baixando-as via internet e configurando-as a nível global (flag *-g*).

Em seguida, escolha um diretório de sua preferência e dê-lhe o nome de “pixijs-devmedia”, o qual guardará todos os arquivos do projeto. Acesse o mesmo via terminal de comandos (comando *cd nome\_do\_diretório*).

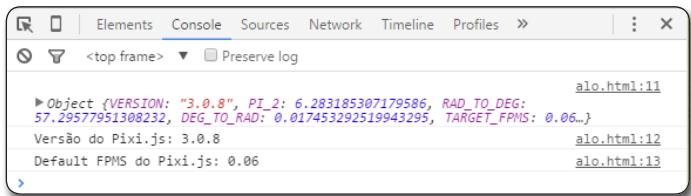
Para baixar o Pixi.js para o nosso projeto, basta clicar no botão *Download* na página home do mesmo e você será redirecionado para a página do framework no GitHub. Clique no botão *Download ZIP* e o projeto inteiro, incluindo os fontes e arquivos de testes, será baixado em um arquivo .zip. De todos os arquivos do projeto, o único que precisaremos é o JavaScript contido na pasta *pixi.js-master/bin*. Existem dois tipos: o com sufixo *-min* e o sem. O primeiro diz que o arquivo tem seu conteúdo interno minificado, isto é, comprimido para facilitar seu download via HTTP no site final. Vamos importar esse para o projeto, mas se o leitor desejar ver a estrutura do JS do Pixi mais a fundo pode importar o outro arquivo.

Crie uma pasta *js* dentro do nosso projeto *pixijs-devmedia*, extraia o conteúdo do arquivo e copie o *pixi.js* para a mesma.

### Criando um template básico

Vamos agora criar uma página HTML básica que apenas importa o arquivo JS na mesma e acessa o seu objeto principal, *PIXI*. Adicione à mesma o conteúdo exposto na **Listagem 1**.

Na página primeiramente importamos o arquivo e acessamos algumas de suas propriedades na tag *<script>*. O objeto *PIXI* é o objeto principal da API que encapsula tudo que há no framework e será o mais usado daqui para frente. Também imprimimos a versão dele e a constante *TARGET\_FPMMS* que guarda o valor da variável de *frames per milliseconds*, muito usada em APIs de jogos. Abra o arquivo em um navegador de sua preferência e, em seguida, o *Console* da ferramenta de depuração (na maioria dos navegadores essa opção está disponível via atalho *F12*). O resultado será semelhante ao exibido pela **Figura 1**.



**Figura 1.** Saída no Console do exemplo

### Listagem 1. Conteúdo HTML da página que testa o Pixi.js.

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>

<body>
  <script src="js/pixi.min.js"></script>
  <script>
    // Testa se o Pixi está funcionando
    console.log(PIXI);
    console.log('Versão do Pixi.js: ' + PIXI.VERSION);
    console.log('Default FPMMS do Pixi.js: ' + PIXI.TARGET_FPMMS);
  </script>
</body>
```

### Trabalhando com Sprites

O *Pixi.js* tem um objeto chamado **renderer** que cria e exibe uma tela. Este, por sua vez, automaticamente gera um elemento do tipo *<canvas>* da HTML5 e entende como exibir suas imagens no *canvas* (vide **BOX 1**). Entretanto, também precisamos criar um objeto do tipo **Container** que representará o *stage* onde tudo acontecerá, isto é, esse objeto basicamente representa o container raiz que contém todas as coisas que o Pixi exibirá. Vejamos o código exibido na **Listagem 2**, necessário para criar um *renderer* e um *stage*.

Veja que a primeira linha do nosso script faz referência ao modelo de modo estrito do JavaScript (vide **BOX 2**), ele é importante para definirmos o acesso à palavra reservada *let* (usada para declarar variáveis locais no JavaScript). A função *autoDetectRenderer()* recebe dois parâmetros para definir o tamanho do canvas a ser renderizado, retornando-o após criado. Em seguida, adicionamos o referido objeto ao corpo do nosso documento HTML.

# Desenvolvimento de jogos web com Pixi.js – Parte 1

## BOX 1. Canvas

O elemento Canvas faz parte da nova API da HTML5 e sua tag (<canvas>) é muito similar a outras como <div> ou <a>, com a exceção de que seu conteúdo será renderizado pelo JavaScript. Para fazer uso da mesma precisamos inserir a tag no documento HTML, acessá-la via JavaScript, criar um contexto (2d ou webgl – 3d), e então usar a sua API para desenhar as visualizações. Vejamos um exemplo básico:

```
// No HTML
<canvas id="canvas" width="200" height="200">
  Este texto é exibido caso o navegador não tenha suporte ao Canvas.
</canvas>

// No JavaScript
var canvas = document.getElementById('canvas'),
    contexto = canvas.getContext('2d');

contexto.fillStyle = 'green';
contexto.fillRect(130,130,130,130);
```

Este exemplo mostra a criação de um canvas simples de id “canvas” que é recuperado em seguida pelo JavaScript e associado a um contexto de desenho 2d. Em seguida, preenchemos a forma com a cor verde e definimos suas dimensões através do método `fillRect()`, que desenha um retângulo de acordo com os valores passados por parâmetro.

## Listagem 2. Criando objetos renderer e stage.

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>

<body>
  <script src="js/pixi.min.js"></script>
  <script>
    "use strict";

    // Cria o renderer
    let renderer = PIXI.autoDetectRenderer(500, 500);
    // Add o canvas ao documento HTML
    document.body.appendChild(renderer.view);

    //Cria um objeto container `stage`
    let stage = new PIXI.Container();
    // Diz ao `renderer` para `render` (renderizar) o `stage`
    renderer.render(stage);
  </script>
</body>
```

Para criar um container chamamos a função construtora `Container()` do objeto `PIXI` e adicionamos o mesmo ao objeto de renderer via função `render()`. Após executar a página no browser teremos um resultado semelhante ao exibido na **Figura 2**.

O método `autoDetectRenderer(500, 500)` já se encarrega de definir o contexto do canvas automaticamente: se é 2d ou webgl. Mas precisamos forçar estas definições através do seguinte código, respectivamente:

```
renderer = new PIXI.CanvasRenderer(500, 500);
renderer = new PIXI.WebGLRenderer(500, 500);
```

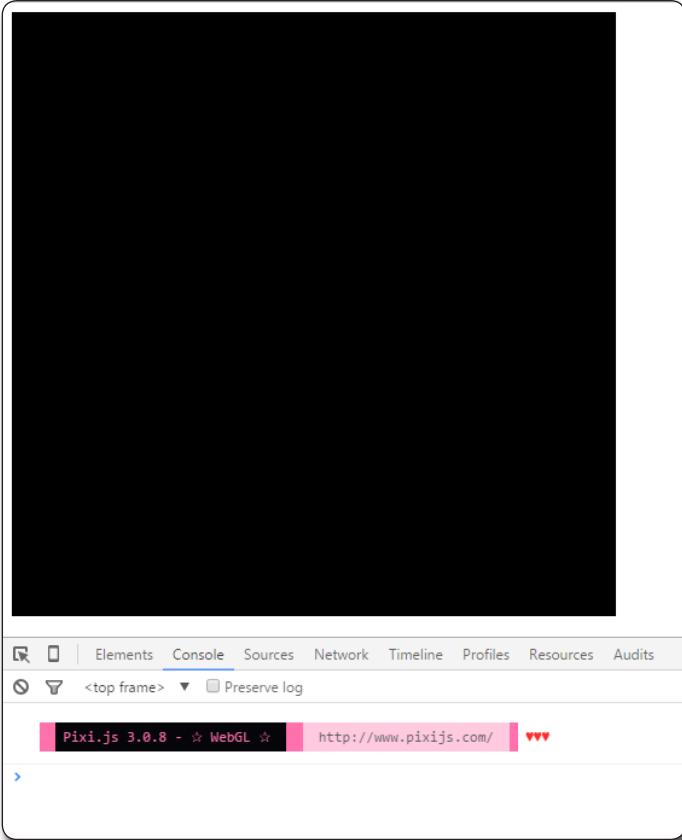


Figura 2. Objeto canvas criado e adicionado à página

## BOX 2. Strict Mode (Modo Estrito)

O modo estrito é uma forma de não silenciar o JavaScript. Em outras palavras, é uma forma de definir comportamentos a nível de browser (que pode ou não suportar tal modo), o qual elimina alguns erros silenciosos no JavaScript, passando a lançá-los. Por exemplo, quando criamos uma variável qualquer e não a declaramos através da palavra chave “var”, ela se torna uma variável válida para o JavaScript de nível global, isto é, visível para todo código JavaScript carregado no DOM. A questão é que para o modo estrito isso é um erro, pois ele força o uso da palavra “var” para definições de variáveis. Portanto, se o browser estiver com tal modo habilitado, receberemos um erro no Console informando isso.

O objeto `renderer.view` é apenas um objeto <canvas> simples, assim podemos controlá-lo da mesma maneira que qualquer outro objeto canvas. Veja como dar à tela uma borda tracejada opcional:

```
renderer.view.style.border = "1px dashed black";
```

Se tiver que mudar a cor do fundo do canvas depois de criado, defina a propriedade `backgroundColor` do objeto renderizador para qualquer valor de cor hexadecimal. Aqui está um exemplo de como podemos configurá-lo para branco puro:

```
renderer.backgroundColor = 0xFFFFFFFF;
```

Veja na **Listagem 3** um exemplo de uso de tais propriedades.

### Listagem 3. Mudando propriedades padrão do renderer.

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>

<body>
  <script src="js/pixi.min.js"></script>
  <script>
    "use strict";

    var renderer = PIXI.autoDetectRenderer(500, 500, {antialiasing: false,
    transparent: false, resolution: 1});
    document.body.appendChild(renderer.view);
    renderer.backgroundColor = 0x8cc53e;

    var scene = new PIXI.Container();

    var render = function() {
      renderer.render(scene);
      requestAnimationFrame(render);
    }

    render();
  </script>
</body>
```

Veja que passamos agora um terceiro parâmetro (opcional) na instanciação do objeto de renderer. Trata-se de um objeto anônimo com três propriedades, a saber:

- **antialiasing**: suaviza as bordas de fontes e gráficos primitivos;
- **transparent**: faz com que o plano de fundo do canvas seja transparente;
- **resolution**: torna o processo de trabalhar com resoluções variantes e densidades em pixels mais fácil.

Após adicionar o renderer ao corpo da página, definimos sua nova cor de fundo (valor sempre em hexadecimal), instanciamos o container (*scene*) e criamos uma nova função de renderização do componente, *render()*. Ela basicamente chamará o método *render()* da API do Pixi, porém efetuando uma chamada consequente ao método *requestAnimationFrame()* que se responsabiliza por desenhar os gráficos de fato na tela. Execute novamente a página no browser e você verá o resultado. Se você quiser fazer com que a tela do seu jogo ocupe todo o espaço disponível no navegador, basta adicionar as propriedades exibidas na **Listagem 4** ao seu renderer.

Note que para cada nova propriedade CSS da página precisamos chamar o mesmo objeto *style* passando o novo valor. Para esse exemplo, especificamente, basta chamar os atributos *innerWidth* e *innerHeight* do objeto implícito *window* do JavaScript que retornarão as atuais largura e altura da tela aberta, respectivamente. Note também que precisamos remover os atributos de *padding* e *margin* padrão que o browser põe nas páginas HTML, através da tag *<style>* recém adicionada.

O leitor pode ainda fazer uso da função *scaleToWindow()* da API do Pixi que se encarregará de redimensionar a tela do jogo da melhor forma possível, quando estiver visualizando o mesmo em padrão *portrait* ou *landscape*.

### Listagem 4. Exibindo o renderer em toda a página.

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
  <style>* {padding: 0; margin: 0}</style>
</head>

<body>
  <script src="js/pixi.min.js"></script>
  <script>
    "use strict";

    var renderer = PIXI.autoDetectRenderer(500, 500, {antialiasing: false,
    transparent: false, resolution: 1});
    document.body.appendChild(renderer.view);
    renderer.backgroundColor = 0x8cc53e;
    renderer.view.style.position = "absolute";
    renderer.view.style.width = window.innerWidth + "px";
    renderer.view.style.height = window.innerHeight + "px";
    renderer.view.style.display = "block";

    var scene = new PIXI.Container();

    var render = function() {
      renderer.render(scene);
      requestAnimationFrame(render);
    }

    render();
  </script>
</body>
```

Voltando agora para os Sprites, precisamos obrigatoriamente ter ao menos um objeto *stage* para que o objeto de renderização possa desenhar os objetos de sprites em um cenário. Os sprites são objetos visuais (geralmente imagens) que podemos controlar usando JavaScript. Quando falamos em controlar nos referimos a animar, exibir/esconder, mover, dentre outras ações comuns em jogos. O *Pixi.js* tem uma classe especializada em lidar com esses recursos chamada *Sprite*. Esta, por sua vez, nos concede três formas de criar sprites, a saber:

- Através de uma imagem comum;
- Através de uma sub-imagem inserida em um *tileset*. Um tileset é uma imagem única e grande constituída de outras imagens menores. No nosso caso, todas as imagens do jogo estarão contidas em um tileset;
- Através de uma *textura atlas*: um arquivo JSON que define o tamanho e a posição de uma imagem num tileset.

### Entendendo de Texturas e Cache

O Pixi renderiza imagens usando a GPU (*Graphics Processing Unit*) do sistema onde o mesmo está sendo executado: um computador, telefone celular ou tablet. A GPU é apenas um chip especializado para a exibição de gráficos de alto desempenho. O navegador da Web se comunica com a GPU usando uma API HTML5 chamada WebGL. Então, para exibir uma imagem ela tem que estar em um formato que a WebGL possa usar para facilmente se comunicar com a GPU.

# Desenvolvimento de jogos web com Pixi.js – Parte 1

Uma imagem WebGL é chamada de **textura**. Antes que possamos fazer um sprite exibir uma imagem, teremos que converter um arquivo de imagem comum em uma textura WebGL. Para manter tudo funcionando rápida e eficientemente o Pixi usa um cache de textura para armazenar e referenciar todas as texturas que seus sprites irão precisar. Os nomes das texturas são strings que correspondem aos locais das imagens que eles referenciam no arquivo. Isso significa que se tivermos uma textura que foi carregada a partir de “img/minhaImagem.png”, podemos encontrá-la no cache de textura através do seguinte código:

```
PIXI.utils.TextureCache["img/minhalmagem.png"];
```

As texturas são armazenadas em um formato compatível com a WebGL e que seja eficiente para o renderizador de Pixi trabalhar. É possível então usar a classe *Sprite* do Pixi para criar um novo sprite usando a textura. Vejamos um pequeno exemplo:

```
let textura = PIXI.utils.TextureCache["img/minhalmagem.png"];
let sprite = new PIXI.Sprite(textura);
```

Para carregar as imagens na tela, tudo que precisamos é do objeto *loader* do PIXI. Ele consegue lidar com qualquer tipo de extensão de imagem. Vejamos o exemplo mostrado pela **Listagem 5**.

## Listagem 5. Usando o objeto loader.

```
PIXI.loader.add("minhalmagem.png").load(setup);
function setup() {
    let sprite = new PIXI.Sprite(PIXI.loader.resources("minhalmagem.png").texture);
}
```

Veja que chamamos o método *add()* do objeto *loader* antes de carregar de fato a imagem. O passo a passo é sempre o mesmo: primeiro adicionamos todas as imagens que queremos usar no jogo no início do script, depois chamamos o método *load()* passando a função que será executada quando as imagens forem carregadas no JavaScript. A função *setup()* instancia um novo objeto do tipo *PIXI.Sprite* passando a textura dessa imagem por parâmetro. É aconselhado que o leitor sempre acesso os recursos de imagens através do objeto *loader* e sua respectiva propriedade *texture*.

Mas antes de visualizar a imagem propriamente dita no arquivo, precisamos adicionar o sprite criado ao objeto *stage* do Pixi que criamos antes:

```
stage.addChild(seuSprite);
```

E em seguida, dizer ao objeto *renderer* para renderizar o *stage*, como já fizemos antes. Veja o exemplo exibido na **Listagem 6**. Nele temos a adição de uma imagem aos recursos do *loader* e, no método *setup()*, seguimos todo o processo de criação do sprite e exibição no *stage*. Salve este conteúdo em um novo arquivo HTML.

## Listagem 6. Exemplo que exibe um sprite na página.

```
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>Hello World</title>
    <style> * {padding: 0; margin: 0}</style>
</head>

<body>
    <script src="js/pixi.min.js"></script>
    <script>
        "use strict";

        let stage = new PIXI.Container(),
            renderer = PIXI.autoDetectRenderer(256, 256);

        document.body.appendChild(renderer.view);
        PIXI.loader.add("img/logo.png").load(setup);

        function setup() {
            // Cria o sprite a partir da textura
            let pixie = new PIXI.Sprite(
                PIXI.loader.resources["img/logo.png"].texture
            );
            // Add o sprite ao stage
            stage.addChild(pixie);
            // Renderiza o stage
            renderer.render(stage);
        }
    </script>
</body>
```

Ao executar, dependendo do browser que você estiver usando, é provável que você receba a seguinte mensagem de erro no Console:

```
Uncaught SecurityError: Failed to execute 'texImage2D' on 'WebGLRenderingContext':
The cross-origin image at file:///D:/tests/pixijs-devmedia/img/logo.png may not be
loaded.
```

Esse erro acontece porque estamos usando o Chrome que, neste caso, bloqueia o acesso a recursos que não venham de um mesmo endereço de origem, ou seja, de um mesmo servidor. Arquivos carregados localmente não funcionam nesse tipo de estratégia porque precisamos adicionar alguns cabeçalhos à resposta HTTP. E é nessa hora onde entra o plugin do *http-server* que configuramos antes. Acesse o diretório do seu projeto via prompt cmd e digite o comando *http-server*. Quando fizer isso, verá uma mensagem semelhante à exibida na **Figura 3** aparecer no Console informando os endereços de IP e porta onde este servidor está executando e que podemos usar para acessar os recursos da aplicação. Veja que a finalidade do *http-server* é fornecer recursos estáticos, isto é, recursos web comuns como páginas HTML, arquivos JS ou CSS, imagens, etc. Dessa forma, caso o leitor deseje executar algum código a nível de servidor com uma linguagem de programação específica como o PHP, Java ou .NET, terá de usar os ambientes de servidor respectivos de cada uma. O Node.js também pode ser usado para tal finalidade apenas com JavaScript se você tiver familiaridade com o mesmo.

Veja que a primeira linha da resposta diz que o *http-server* estará servindo no endereço “./”. Isso quer dizer que todos os arquivos dentro da pasta do projeto pixijs-devmedia estarão disponíveis via browser.

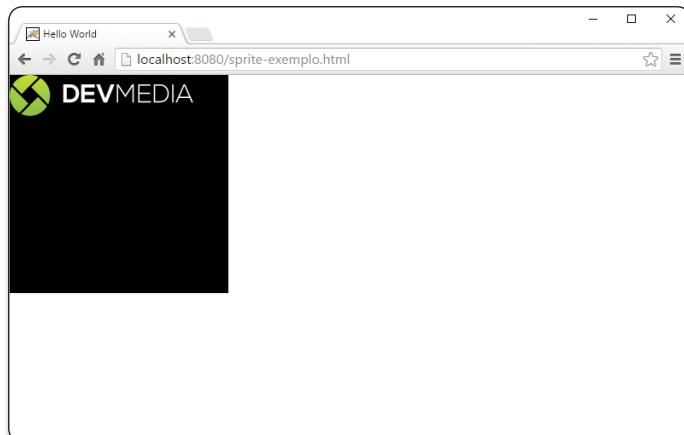
Agora, para testar a nova página criada basta acessar o seguinte endereço num navegador web:

<http://localhost:8080/sprite-exemplo.html>

O *localhost* substitui o 127.0.0.1, o famoso IP universal. Execute a URL no browser e você verá algo parecido com a **Figura 4**.

```
D:\tests\pixijs-devmedia>http-server
Starting up http-server, serving .
Available on:
  http://192.168.56.1:8080
  http://192.168.1.34:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

**Figura 3.** Mensagem indicando IP e porta do servidor local



**Figura 4.** Sprite adicionado ao stage

#### Monitorando progresso de carregamento

O objeto *loader* tem ainda um evento especial de progresso que pode chamar uma função de callback customizada sempre que um arquivo é carregado. Os eventos de progresso são chamados pelos loaders no método *on()*, por exemplo:

```
loader.on("progress", loadProgressHandler);
```

Vamos analisar o exemplo exibido pela **Listagem 7**. Crie uma nova página HTML e adicione o código à mesma.

Vejamos algumas observações:

- Na linha 14 mudamos os valores das dimensões do renderer para 500px de largura e altura. Veja que ambos os objetos são locais pois estão recebendo a palavra-chave *let* separados por vírgula;

**Listagem 7.** Exemplo que lida com o loading dos sprites.

```
01 <!doctype html>
02 <head>
03   <meta charset="utf-8">
04   <title>Hello World</title>
05   <style>* {padding: 0; margin: 0}</style>
06 </head>
07
08 <body>
09   <script src="js/pixi.min.js"></script>
10  <script>
11    "use strict";
12
13  let stage = new PIXI.Container(),
14  renderer = PIXI.autoDetectRenderer(500, 500);
15
16  document.body.appendChild(renderer.view);
17
18  PIXI.loader.add([
19    "img/plantas.png",
20    "img/solo.png",
21    "img/gerador.png"
22  ]).on("progress", loadProgressHandler).load(setup);
23
24  function loadProgressHandler(loader, resource) {
25    // Exibe a 'url' do arquivo sendo carregada
26    console.log(`carregando: ${resource.url}`);
27    // Exibe o percentual dos arquivos atualmente carregados
28    console.log(`progresso: ${loader.progress}`);
29  }
30
31  function setup() {
32    console.log("Todos os arquivos foram carregados!");
33    let plantas = new PIXI.Sprite(
34      PIXI.loader.resources["img/plantas.png"].texture
35    );
36    plantas.y = 40;
37    let solo = new PIXI.Sprite(
38      PIXI.loader.resources["img/solo.png"].texture
39    );
39    solo.x = 40;
40    solo.y = 20;
41    let gerador = new PIXI.Sprite(
42      PIXI.loader.resources["img/gerador.png"].texture
43    );
44    gerador.x = 170;
45    gerador.y = 40;
46    gerador.rotation = 0.5;
47    // Add o sprite ao stage
48    stage.addChild(plantas);
49    stage.addChild(solo);
50    stage.addChild(gerador);
51
52    // Renderiza o stage
53    renderer.render(stage);
54  }
55
56 </script>
57 </body>
```

- Na linha 18 estamos carregando as imagens dos sprites em um vetor e passando o mesmo para o método *add()* do loader, isso porque o mesmo possibilita o envio de vários valores ao mesmo tempo, simplificando assim a implementação. As imagens mencionadas no código, bem como as demais usadas nos outros exemplos, podem ser encontradas no arquivo de código fonte deste artigo;

- Na linha 22 setamos a função `on()` que funciona como um ouvinte de eventos mapeando especificamente o de progresso do carregamento. O segundo parâmetro da função recebe o método que se encarregará de fazer alguma coisa quando este carregamento estiver em andamento. Em seguida, chamamos a função `load()`;
- Na linha 24 temos a função `loadProgressHandler()` que recebe dois parâmetros: o `loader` e o `resource` (contém dados específicos do recurso). Veja que nesta função estamos imprimindo dois valores: a URL do recurso (que imprimirá o mesmo valor que configuramos no método `add()`) e o progresso (quantidade percentual de carregamento dos arquivos relativa ao momento da impressão). Para fazer isto, usamos do recurso de variáveis embutidas nas strings, algo permitido pelo JavaScript via operador `${}`;
- Na linha 31 criamos a função de `setup()` que imprime no console de imediato a mensagem de sucesso no carregamento dos arquivos de sprites. Em seguida, instanciamos nossos três sprites em sequência: da imagem de plantas, solo e gerador, acessando a propriedade `texture` de cada um para tal. Veja que para cada um dos sprites, configuramos suas propriedades `x`, `y` e `rotation` de modo a definir a posição x e y no eixo cartesiano, bem como o nível de rotação que aquela imagem terá ao ser exibida, respectivamente.
- Finalmente, nas linhas 49 a 51 adicionamos cada um deles ao stage e este ao renderer (linha 54).

O resultado pode ser conferido na **Figura 5**. Veja os logs efetuados no Console associados a cada sprite bem como o percentual impresso de cada carregamento total. Na tela do canvas é possível ver os elementos posicionados nos valores de eixos que definimos, caso contrário todos apareceriam sobrepostos uns aos outros no canto esquerdo superior.

## Criando Sprites a partir de Tilesets

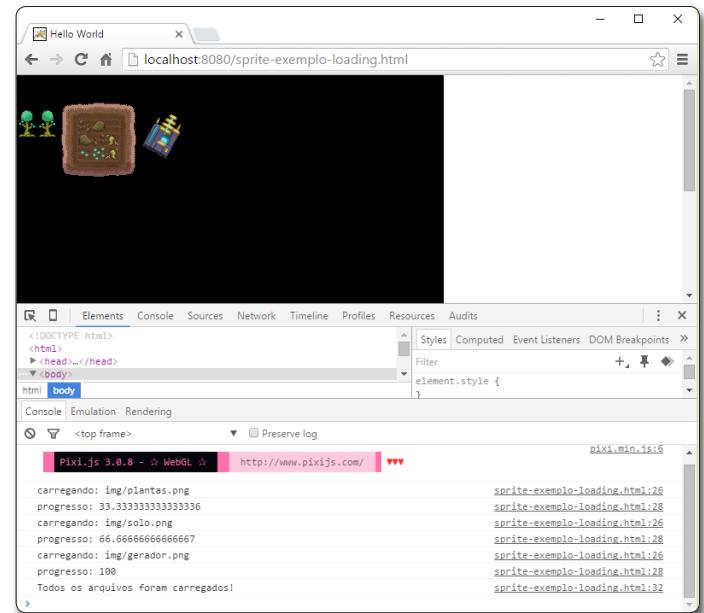
O uso de tilesets para exibir as imagens é a forma mais comum de se trabalhar com sprites em jogos, independente da tecnologia usada. Isso porque ganhamos muito em carregamento, já que todas as imagens estão reunidas em uma só. Imagine ter de carregar centenas ou até milhares de imagens ao mesmo tempo em um cenário de jogo, lembrando que cada imagem individual precisa de uma requisição ao servidor para ser buscada. Isso gera um overhead desnecessário ao aplicativo que pode ser muito mais performático usando tilesets.

Um tileset é uma imagem com várias imagens internas. Veja na **Figura 6** um exemplo de tileset que usaremos no exemplo a seguir (importe-o para o seu projeto).

Agora, para exibir qualquer um destes itens na página precisamos efetuar o mesmo procedimento de carregamento do sprite, bem como inclusão do mesmo no tileset. Vejamos a **Listagem 8**. Crie uma nova página e adicione o seu conteúdo na mesma.

### Observação

As imagens precisam estar com extensão PNG e com fundo transparente, caso contrário quando as cortarmos para exibir individualmente, cada uma aparecerá com o fundo também

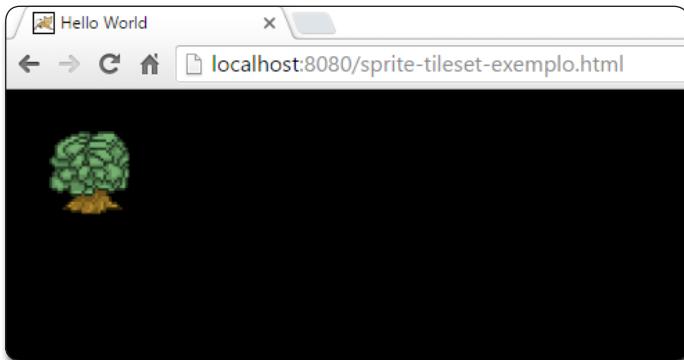


**Figura 5.** Exemplo com vários sprites e progresso



**Figura 6.** Exemplo de tileset com várias imagens internas

Veja que dividimos a lógica de recuperação da textura e adição da mesma no objeto renderer em duas partes. A função `setup()` se encarrega de passar um novo objeto do tipo `Rectangle` para o método `getTexture()`. Para recuperar um pedaço de uma imagem precisamos criar um objeto de retângulo definindo as dimensões (na ordem: eixos x, y e de corte da imagem, largura e altura a partir dos pontos x, y definidos) dessa sub-imagem e passá-lo ao atributo `frame` do objeto de textura. Esse atributo se encarrega de limitar a visualização do sprite, cortando a imagem se for preciso. A função `getTexture()` também recebe os valores do eixo x e y para definir onde a textura deve aparecer no cenário. Logo no início da função criamos um novo objeto do tipo `TextureCache` passando a imagem pré-carregada do tileset. Depois, é só criar um novo `Sprite` passando a textura como parâmetro no construtor e setando as dimensões x e y que recebemos como argumento. No final, adicione a textura ao stage (cenário) via método `addChild()`. O resultado pode ser conferido na **Figura 7**.



**Figura 7.** Página com imagem carregada direto do tileset

**Listagem 8.** Exibindo imagens a partir de um tileset.

```

<!doctype html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
  <style>* {padding: 0; margin: 0}</style>
</head>

<body>
  <script src="js/pixi.min.js"></script>
  <script>
    "use strict";

    let stage = new PIXI.Container(),
        renderer = PIXI.autoDetectRenderer(500, 200);

    document.body.appendChild(renderer.view);

    PIXI.loader.add([
      "img/tileset.png"
    ]).load(setup);

    function setup() {
      // Cria um objeto de retângulo que define a posição e
      // o tamanho da sub-imagem que queremos extrair da textura
      getTexture(new PIXI.Rectangle(125, 0, 60, 60), 30, 30);

      renderer.render(stage);
    }

    function getTexture(retangulo, x, y) {
      // Cria o sprite de 'tileset' a partir da textura
      let texture = PIXI.utils.TextureCache["img/tileset.png"];

      // Diz à textura para usar a seção do retângulo
      texture.frame = retangulo;
      // Cria o sprite a partir da textura
      let sprite_texture = new PIXI.Sprite(texture);
      // Posiciona o sprite no canvas
      sprite_texture.x = x;
      sprite_texture.y = y;

      stage.addChild(sprite_texture);
    }
  </script>
</body>
```

Agora que sabemos como adicionar um objeto independente ao cenário a partir de um tileset, vamos montar nosso cenário com os demais objetos carregados. Comecemos pelo plano de fundo, especificamente criando o chão do nosso jogo. No mesmo arquivo, altere o conteúdo JavaScript pelo apresentado na **Listagem 9**.

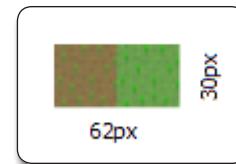
**Listagem 9.** Criando o plano de fundo do jogo.

```

01 "use strict";
02
03 var stage = new PIXI.Container(),
04   renderer = PIXI.autoDetectRenderer(520, 300),
05   TextureCache = PIXI.utils.TextureCache,
06   Texture = PIXI.Texture,
07   Sprite = PIXI.Sprite,
08   Rectangle = PIXI.Rectangle;
09
10 var TILESET_DIR = "img/tileset.png";
11
12 document.body.appendChild(renderer.view);
13
14 PIXI.loader.add(TILESET_DIR).load(setup);
15
16 function setup() {
17   carregarFundoCenario();
18
19   getTexture(TILESET_DIR, new Rectangle(125, 0, 60, 60), 30, 30);
20 }
21
22 function carregarFundoCenario() {
23   let qtde_linha = parseInt(renderer.width/65), cont_linha = 0;
24   let qtde_coluna = parseInt(renderer.height/32), cont_coluna = 0;
25   for (let i = 0; i <= qtde_coluna; i++) {
26     cont_linha = 0;
27     for (let j = 0; j <= qtde_linha; j++) {
28       getTexture(TILESET_DIR, new Rectangle(0, 0, 62, 30), cont_linha, cont_coluna);
29       cont_linha += 62;
30     }
31     cont_coluna += 30;
32   }
33 }
34
35 function getTexture(source, retangulo, x, y) {
36   let texture, imageFrame;
37   // Se source for uma string, uma textura no cache ou um arquivo de imagem
38   if (typeof source === "string") {
39     if (TextureCache[source]) {
40       texture = new Texture(TextureCache[source]);
41     }
42   } else if (source instanceof Texture) {
43     // Se 'source' é uma textura, use-a
44     texture = new Texture(source);
45   }
46   if (texture) {
47     // Faz o retângulo do tamanho da sub-imagem
48     imageFrame = retangulo;
49     texture.frame = imageFrame;
50   }
51 }
52 let sprite = new Sprite(texture);
53 sprite.x = x;
54 sprite.y = y;
55
56 stage.addChild(sprite);
57 renderer.render(stage);
58 }
```

Algumas mudanças estruturais e organizacionais se fizeram necessárias para evitar duplicidade de código, vejamos:

- Na linha 3 criamos uma cadeia de variáveis inicializadas de modo global no script. Trata-se das já conhecidas variáveis de container (renderer) e novas variáveis de *alias*. Um *alias* em JavaScript representa um atalho para determinado “tipo” de objeto já existente em alguma API que estejamos usando. Por exemplo, a API do Pixi.js está cheia de classes aninhadas com as respectivas chamadas a métodos e objetos aninhadas, como PIXI.Sprite, PIXI.utils.TextureCache, etc. Para evitar ter de chamar toda a hierarquia de objetos sempre que precisarmos de determinada classe, criamos uma variável global (via “var”) no início do script e associamos seu nome curto ao nome real, assim simplificamos as futuras chamadas referenciando apenas os alias e o JavaScript interpretará perfeitamente. Faça isso sempre que puder em qualquer API JavaScript;
- Na linha 10 criamos uma constante para salvar o caminho da imagem de tileset;
- Na linha 14 mudamos a forma como carregamos o tileset, não mais passando um vetor, mas sim a imagem por si só;
- Na função setup(), da linha 16, chamamos as novas funções carregarFundoCenario() e getTexture() para adicionar o fundo da tela do jogo e a imagem da árvore, respectivamente;
- Na função carregarFundoCenario(), da linha 22, criamos um mecanismo automático que calcula, com base na largura e altura totais da tela, as dimensões de cada pedaço de terra e grama que exibiremos como chão do nosso plano de fundo. Veja as dimensões de cada pedaço de terra/grama exibidas na **Figura 8**, nela é possível ver a largura/altura da imagem que deverá ser repetida várias vezes no cenário até completar o espaço disponível. A primeira coisa que devemos fazer é calcular a quantidade de vezes que precisaremos “pintar” essa imagem na horizontal (eixo x) e vertical (eixo y). Para a horizontal, dividimos a largura total da tela (ou seja, do *renderer*) pela largura do sprite de terra/grama (acrescido de 2 ou 3 para dar uma margem = 65px); e para a vertical, dividimos a altura total da tela pela altura do sprite (acrescido de 2 ou 3 para dar uma margem = 32 px). Com as quantidades em mãos, podemos iterar sobre as colunas e linhas tal como se a tela fosse uma matriz (x, y) e precisássemos preenchê-la com dois *for*’s. Para cada um, criamos uma variável contadora que será incrementada com o próprio valor adicionado da altura (primeiro *for*) e largura (segundo *for*) do sprite e, em seguida, serão passadas para o método getTexture() que dimensionará em que lugar do cenário cada uma deve aparecer;
- Na função getTexture() da linha 35 mudamos sua assinatura para receber agora o *source* (caminho da imagem), o retângulo que será usado para cortar o tileset e os valores x, y para definir onde a textura deve aparecer no cenário. Também precisamos verificar se a imagem já não está inserida no cache para o caso de não precisar refazer todo o processo e aproveitar o mesmo; caso não esteja, criamos um novo objeto Texture e adicionamos a imagem ao mesmo. Depois, na linha 52 criamos um novo objeto Sprite com a textura selecionada, mudamos suas propriedades cartesianas (linhas 53 e 54) e finalizamos o processo.



**Figura 8.** Dimensões do sprite de terra/grama

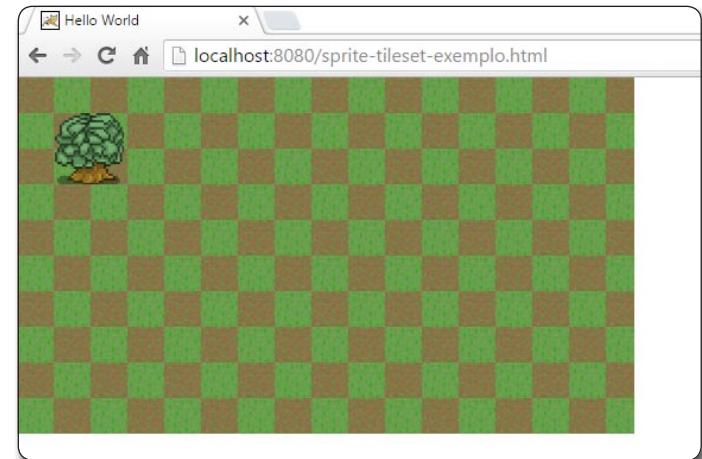
Ao executar o exemplo no navegador você verá algo semelhante à **Figura 9**. Uma vez com o ambiente preparado, modificar as propriedades das texturas é muito simples. Por exemplo, supondo que em vez do efeito de listrado você deseje um efeito quadriculado no plano de fundo. Basta modificar a linha 26 da listagem anterior para a seguinte:

```
cont_linha = (i % 2 == 0) ? 0 : -31;
```

Nela fazemos um teste ternário verificando se o valor de i (variável que itera) é par. Caso seja, setamos zero para a variável contadora, caso contrário definimos seu valor com -31, assim a linha corta metade da imagem na primeira exibição e o efeito quadriculado é aplicado, tal como vemos na **Figura 10**.



**Figura 9.** Tela do cenário com plano de fundo preenchido em listras



**Figura 10.** Tela do cenário com plano de fundo preenchido em quadriculado

Veja como a imagem da árvore se sobressai às do plano de fundo. Isso porque fizemos a chamada ao método que a renderiza após as chamadas ao do plano de fundo. Essa ordem deve ser obedecida, uma vez que o Pixi.js organiza seus sprites em camadas superpostas.

Supondo que queiramos rodear o cenário com arbustos para delimitar as bordas de onde os personagens poderão se locomover, teríamos uma função igual à da **Listagem 10**.

**Listagem 10.** Função que insere arbustos ao redor do cenário.

```
function carregarArbustos() {
    let qtde_linha = parseInt(renderer.width/65), cont_linha = 0;
    let qtde_coluna = parseInt(renderer.height/32), cont_coluna = 0;
    for (let i = 0; i <= qtde_coluna; i++) {
        cont_linha = 0;
        for (let j = 0; j <= qtde_linha; j++) {
            if (i == 0 || i == qtde_coluna) {
                getTexture(TILESET_DIR, new Rectangle(95, 0, 30, 30),
                cont_linha, cont_coluna);
                cont_linha += 31;
            }
            getTexture(TILESET_DIR, new Rectangle(95, 0, 30, 30),
            cont_linha, cont_coluna);
            cont_linha += 31;
        } else if (j == 0 || j == qtde_linha) {
            getTexture(TILESET_DIR, new Rectangle(95, 0, 30, 30),
            cont_linha, cont_coluna);
            cont_linha += 62;
        } else {
            cont_linha += 62;
        }
        cont_coluna += 30;
    }
}
```

Veja que inicialmente criamos a mesma lógica que mapeia a quantidade de linhas e colunas do cenário, criando também dois contadores para cada um. Iteramos sobre as mesmas de igual forma, porém dentro do segundo *for* temos agora algumas condições sendo feitas. Para garantir que os arbustos só aparecerão nas bordas precisamos levar em consideração que:

- Apenas a primeira e última linha devem ser mapeadas;
- Para as demais, apenas a primeira e última coluna devem ser mapeadas;
- Cada sprite de chão é dividido em duas partes: um bloco de grama e outro de terra, ou seja, temos duas imagens em um só sprite.

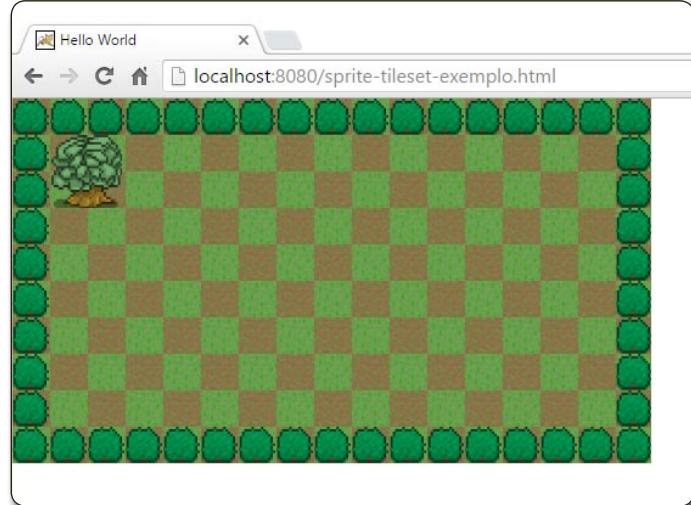
Portanto, o primeiro teste é checar se o valor de *i* (que conta a linha atual) é igual a zero (primeira linha) ou igual à variável *qtde\_coluna* (última linha). Caso positivo, chamamos o método *getTexture()* passando os parâmetros corretos da posição do arbusto na imagem (*x, y, largura, altura*), bem como as posições onde eles devem aparecer no cenário (*cont\_linha* e *cont\_coluna*). Mas este método deve ser chamado duas vezes somente nesta condição, isso porque precisamos atender ao requisito de duas imagens por sprite que vimos antes e é por isso que incrementamos o valor de *cont\_linha* com metade do valor convencional (ou seja, 31).

A segunda condição testa se, caso não estejamos na primeira ou última linhas, estamos na primeira ou última colunas de cada linha. Caso positivo, mapeamos o mesmo arbusto, porém agora só uma vez. Por fim, se não nenhuma das condições forem atendidas, incrementamos o contador de linha normalmente e no final o de coluna.

Quando fizer isso, nosso método estará pronto bastando adicionar a sua chamada logo após a função *carregarFundoCenario()* do método *setup()*. O resultado pode ser conferido na **Figura 11**.

O leitor pode ainda adicionar outros personagens ao cenário, mapeando suas posições no tileset e inserindo os mesmos em posições aleatórias ao longo do mesmo. Veja na **Listagem 11** um exemplo onde mapeamos uma boa parte dos personagens disponíveis no tileset e os adicionamos em posições fixas no cenário. O resultado dessa implementação pode ser conferido em seguida na **Figura 12**.

O leitor pode ainda criar uma espécie de atlas com todos os personagens mapeados em um mapa, bem como suas respectivas posições e dimensões cartesianas, tudo em formato JSON e depois sair chamando via JavaScript cada um em sequência.



**Figura 11.** Tela do cenário com arbustos nas bordas



**Figura 12.** Tela do cenário com demais personagens adicionados

## Listagem 11. Mapeando outros personagens no cenário.

```
function setup() {  
    carregarFundoCenario();  
    carregarArbustos();  
  
    getTexture(TILESET_DIR, new Rectangle(125, 0, 60, 60), 30, 30);  
    getTexture(TILESET_DIR, new Rectangle(125, 0, 60, 60), 170, 70);  
    // Personagem 1  
    getTexture(TILESET_DIR, new Rectangle(95, 32, 30, 30), 340, 90);  
    // Personagem 2  
    getTexture(TILESET_DIR, new Rectangle(95, 63, 30, 30), 340, 180);  
    // Personagem 3  
    getTexture(TILESET_DIR, new Rectangle(95, 95, 30, 30), 40, 230);  
    // Personagem 4 de costas  
    getTexture(TILESET_DIR, new Rectangle(0, 125, 30, 30), 90, 120);  
    // Explosão  
    getTexture(TILESET_DIR, new Rectangle(0, 158, 120, 30), 250, 240);  
    // Bomba  
    getTexture(TILESET_DIR, new Rectangle(125, 63, 30, 30), 330, 30);  
    getTexture(TILESET_DIR, new Rectangle(125, 63, 30, 30), 230, 140);  
}
```

Esse tipo de implementação requer um pouco mais de aprofundamento na API do framework, bem como de JavaScript avançado. Se você não tem conhecimentos sobre JSON é aconselhável estudar um pouco sobre o mesmo pois, além da sua utilidade para o Pixi.js, é uma tecnologia extremamente utilizada e aceita por outros frameworks e linguagens.

O leitor pode agora trabalhar com os recursos aprendidos e criar outros cenários. Um ótimo exercício para fixar os conceitos de sprites, texturas e imagens é buscar na web outros exemplos de tilesets e efetuar o mapeamento de seus sprites, criando diferentes possibilidades de cenários com personagens em tamanhos distintos,

até para que você possa ir se acostumando aos tamanhos de *stage* e às formas como o renderer lida com a renderização de cada um deles. Na seção **Links** disponibilizamos um endereço do site *OpenGameArt* que funciona como um repositório online de conteúdo e arquivos gratuitos para jogos de todos os tipos. Lá você encontrará vários exemplos de tilesets que poderá usar nos seus jogos daqui em diante, ou pelo menos para praticar um pouco mais as definições do Pixi.js. Bons estudos!

## Autor



### Júlio Sampaio

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



## Links:

### Página oficial do Pixi.js.

<http://www.pixijs.com/>

### Página de download do Node.js.

<https://nodejs.org/>

### Projeto do OpenGameArt.

<http://opengameart.org/>

# Introdução ao desenvolvimento de CSS orientado a objetos

Veja neste artigo como criar seu estilo de forma organizada, padronizada, extensível e flexível

**A**lguns desenvolvedores web já devem ter se perguntado como pode uma linguagem de programação estática que é realmente mais voltada à marcação do estilo dos elementos nas páginas fazer uso de objetos e suas peculiaridades? Neste artigo, vamos conhecer a ideia do **CSS orientado a objetos** e ver como ele funciona.

CSS orientado a objetos, ou OOCSS, em sua essência é simplesmente escrever estilo de forma mais limpa. Não é uma linguagem diferente e totalmente nova: continuamos a usar o mesmo velho CSS que todos já conhecemos. É apenas uma “mudança de paradigma”. O que acontece na prática é que, ao usar um CSS orientado a objetos, temos padrões um pouco mais simples e melhores práticas implementadas.

Então por que chamá-lo de orientado a objetos? A programação orientada a objetos (OOP) é um paradigma que usa “objetos” - estruturas de dados que consistem em campos de dados (os *datafields*) e métodos - e suas interações para projetar aplicações e programas de computador. Se tivéssemos que dar uma definição final resumida do que é o OOCSS de fato, poderíamos dizer algo como: CSS orientado a objetos é um paradigma de codificação que usa estilos em “objetos” ou “módulos” – pedaços aninháveis de HTML que definem uma seção de uma página da web – robustos e reutilizáveis.

Isso basicamente significa que teremos um objeto padrão (uma estrutura HTML), bem como classes CSS que aplicaremos ao mesmo e que definem o design e o fluxo de como a página será visualizada pelo usuário final. Confuso? Vamos dar uma olhada na teoria disso tudo.

## Fique por dentro

Este artigo é útil por explorar os principais conceitos na prática acerca do desenvolvimento de código CSS orientado a objetos. Trata-se de um paradigma novo proposto pela engenheira de software Nicole Sullivan como forma de organizar o estilo de nossas páginas com base nos tópicos da metodologia da orientação a objetos: com atributos e métodos, herança, polimorfismo, encapsulamento, etc. Veja como fazer uso de todos estes conceitos através da criação de um modelo de site de blog, além de entender como a estrutura HTML deve conversar diretamente com o seu CSS.

## Teoria do OOCSS

Nicole Sullivan, engenheira de software da empresa Pivotal Software Inc., foi a pioneira dessa ideia. Ela basicamente definiu que a base do paradigma se sustenta na separação da estrutura do *skin* (design base), o que significa que o seu estilo de layout e seu estilo de design devem estar sempre separados. Uma maneira muito prática de fazer isso é usar um sistema de grids; há várias opções disponíveis através de frameworks que já implementam esse tipo de estrutura por padrão (como Bootstrap, Google MDL, etc.), mas você pode criar o seu próprio se desejar. Se você não estiver usando um sistema de grids, provavelmente terá apenas de definir a estrutura do objeto principal da sua página; e isso é o que faremos logo mais na parte prática.

Separar o *container* (recipiente principal) a partir do conteúdo significa que qualquer objeto (o recipiente) deve ser capaz de adaptar-se para aceitar qualquer tipo de conteúdo que seja; por exemplo, não deve ser necessário ter um título de cabeçalho (h1-h6) no topo da página, seguido por uma lista não ordenada para que a aparência do site esteja elegante.

Portanto, esse mecanismo permite flexibilidade e capacidade de reutilização, o que é fundamental.

Há algumas boas razões pelas quais devemos escrever CSS de forma orientada a objetos. Um dos maiores benefícios é que o CSS será mais reutilizável. Mas sua folha de estilos também deve tornar-se muito menor. O OOCSS deve tornar-se, sobretudo, muito mais fácil a manutenção e alteração do design de um site. Além disso, um dos grandes benefícios da orientação a objetos é a possibilidade de mudar partes do seu site, sem que isso implique no não funcionamento de outras.

## Como praticar CSS orientado a objetos?

O primeiro passo é realmente se preparar para o CSS: você deve terminar o seu **objeto HTML**. Geralmente, o objeto terá um cabeçalho, um corpo e um rodapé, embora cabeçalho e rodapé sejam opcionais. Veja na **Listagem 1** um exemplo de objeto básico que representa muito bem isto.

Antes que você presuma que esse tipo de layout está um pouco ultrapassado, em vista da quantidade de novas tags da HTML5 que temos para estruturar tais seções, analise o código da **Listagem 2** que pode perfeitamente substituir o anterior.

**Listagem 1.** Exemplo de template simples para páginas web.

```
<div class="object">
  <div class="head"></div>
  <div class="body"></div>
  <div class="foot"></div>
</div>
```

**Listagem 2.** Exemplo de template simples para páginas web.

```
<article>
  <header></header>
  <section></section>
  <footer></footer>
</article>
```

Por meio da HTML5, temos agora um objeto com significado semântico. Na verdade, este é o objeto que iremos usar para o nosso exemplo. Se vamos escrever algum CSS, vamos precisar de algo para o estilo preparado sob um modelo básico: uma página inicial de um blog e uma única página de post. Para isso, usaremos alguns elementos de estilo da HTML5 e do CSS3.

Com o objetivo de tornar o aprendizado mais fácil e rápido, vamos adaptar alguns dos conceitos do OOCSS na prática, definindo inicialmente a estrutura HTML do nosso blog e entendendo os seus respectivos componentes internos. Para isso, precisaremos de uma estrutura de diretórios e arquivos iniciais que lidará com recursos como HTML, CSS e imagens. Portanto, em um diretório de sua preferência, crie a estrutura demonstrada na **Listagem 3**. Veja que temos dois diretórios, além do raiz /oocss, que conterão os arquivos CSS e as imagens. Estas últimas você pode encontrar no arquivo de download deste artigo.

**Listagem 3.** Estrutura de diretórios e arquivos do projeto.

```
oocss
  | index.html
  | post.html
  |
  +---css
    | style.css
    | texto.css
    |
    \---img
      bullet.png
```

## Criando projeto de exemplo

Para iniciar nossa implementação, abra o arquivo index.html criado e insira o conteúdo da **Listagem 4** no mesmo. Vamos entender aos poucos o seu código, mas desde já é importante que o leitor comprehenda como deve ser montado o esqueleto principal de todas as páginas que serão construídas sob o mesmo design e, consequentemente, farão uso do benefício do OOCSS.

Comecemos então pela importação dos arquivos necessários e definição do cabeçalho da página HTML e corpo (cabeçalho e rodapé). Na referida listagem, teremos a seguinte divisão de responsabilidades no que se refere a organizar o conteúdo final da página:

- Tag **<article>** (de classe *container*): servirá como container geral da página contendo todo o conteúdo do corpo da mesma;
- Tag **<header>**: será usada para conter o conteúdo do cabeçalho do corpo com título, subtítulo e menu principal;
- Tag **<section>**: dentre outras coisas, será usada em todas as páginas para guardar o conteúdo dos posts do blog independentemente de se tratar da listagem deles ou da exibição de apenas um.
- Tag **<aside>**: terá como responsabilidade conter o arquivo do blog (em meses), bem como a lista de últimos comentários. Esse componente deve ser exibido flutuando à direita da página;
- Tag **<footer>**: conterá o conteúdo do rodapé da página.

O leitor já deve ter reparado que preenchemos o conteúdo do cabeçalho e rodapé por serem mais simples e facilitar o foco no corpo (que será apresentado em seguida). Vamos aos detalhes da listagem:

• Linhas 4 a 10: cabeçalho de configuração HTML com a definição da tag meta para configurar o *charset* da página, o título, e os arquivos CSS de estilo. Os dois arquivos de CSS (text e style) serão criados mais à frente. Destaque especial para o import feito na linha 7, cujo atributo *href* aponta para o endereço da API do *Google Fonts*, um site do Google que disponibiliza de forma online uma série de fontes para as aplicações web sem que seja necessário baixar e instalar nenhum arquivo *tff* na máquina do usuário. Na seção **Links** o leitor encontra a URL do site oficial do *Google Fonts*. Fique à vontade para selecionar as que desejar para estilizar sua tipografia a bel prazer. Selecionamos uma fonte primária e duas secundárias:

- *Lora*: será usada para o corpo e demais caracteres do site;
- *Cabin* e *Arvo*: serão usadas para títulos de página (tags h1-h6).

- Linhas 14 a 25: cabeçalho do corpo da página que contém título e subtítulo (este último deve vir com a classe *subtitle* para fins de estilo) e uma tag *<nav>* para criar o menu em forma de lista *<ul>*. É importante seguir um padrão para o site todo, logo para todo título na página defina níveis para cada um e associe-os às tags h1-h6. No item de menu da linha 19 inserimos uma classe *selected* que será usada para dar um estilo diferente ao item selecionado naquele instante.
- Linhas 34 a 36: definição do rodapé apenas com um texto simples.

**Listagem 4.** Estrutura inicial da nossa página index.html.

```

01 <!DOCTYPE html>
02 <html>
03
04 <head>
05   <meta charset='utf-8' />
06   <title>CSS Orientado a Objetos</title>
07   <link href='https://fonts.googleapis.com/css?family=Cabin:500italic|Lora|Arvo' rel='stylesheet' type='text/css'>
08   <link type="text/css" rel="stylesheet" href="css/texto.css"/>
09   <link type="text/css" rel="stylesheet" href="css/style.css"/>
10 </head>
11
12 <body>
13   <article id="container">
14     <header>
15       <h1>CSS Orientado a Objetos</h1>
16       <h2 class="subtitle">(Testando OOCSS em uma página de blog!)</h2>
17       <nav>
18         <ul>
19           <li><a href="index.html" class="selected">Home</a></li>
20           <li><a href="#">Arquivo</a></li>
21           <li><a href="#">Sobre</a></li>
22           <li><a href="#">Contato</a></li>
23         </ul>
24       </nav>
25     </header>
26
27     <section>
28       <!-- Lista de posts do blog -->
29     </section>
30     <aside>
31       <!-- Barra lateral com arquivo do blog e últimos comentários -->
32     </aside>
33
34     <footer>
35       Copyright &copy; Todos os Direitos Reservados
36     </footer>
37
38   </article>
39 </body>
40
41 </html>

```

Dando prosseguimento, adicione o conteúdo da **Listagem 5** diretamente dentro da tag *<section>* que representa a lista de posts da página e o da **Listagem 6** à tag *<aside>* que representa a barra lateral direita da página.

Veja que aqui temos uma lista de três posts com seus resumos, imagens de lead e respectivos links. Cada post deverá ser marcado pela tag *<article>* de classe *post*, muito usada para essa finalidade: demarcar texto principal, o artigo em questão. Cada *<article>*, por sua vez, será dividido em mais três partes:

**Listagem 5.** Conteúdo HTML da listagem de posts da index.html.

```

01 <article class="post">
02   <header>
03     <span class="date">20 de Janeiro de 2015</span>
04     <h2><a href="post.html">Introdução ao JavaScript</a></h2>
05   </header>
06   <section>
07     
08   <p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. </p>
09   </section>
10   <footer>
11     <ul>
12       <li><a href="#">Leia Mais...</a></li>
13       <li><a href="#">Retweet!</a></li>
14       <li><a href="#">Comentários (4)</a></li>
15     </ul>
16   </footer>
17 </article>
18 <article class="post ext">
19   <header>
20     <span class="date">12 de Janeiro de 2015</span>
21     <h2>CSS Orientado a Objetos</h2>
22   </header>
23   <section>
24     
25   <p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit as pernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. </p>
26   </section>
27   <footer>
28     <ul>
29       <li><a href="#">Leia Mais...</a></li>
30       <li><a href="#">Retweet!</a></li>
31       <li><a href="#">Comentários (4)</a></li>
32     </ul>
33   </footer>
34 </article>
35 <article class="post">
36   <header>
37     <span class="date">12 de Janeiro de 2015</span>
38     <h2>Front-End Devs - DevMedia</h2>
39   </header>
40   <section>
41     
42   <p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit as pernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. </p>
43   </section>
44   <footer>
45     <ul>
46       <li><a href="#">Leia Mais...</a></li>
47       <li><a href="#">Retweet!</a></li>
48       <li><a href="#">Comentários (4)</a></li>
49     </ul>
50   </footer>
51 </article>

```

# Introdução ao desenvolvimento de CSS orientado a objetos

- Um `<header>` contendo a data de publicação do post (com classe `date`);
- Um `<section>` para guardar a imagem principal do post, bem como um resumo do texto principal;
- E um `<footer>` para exibir os links com as opções para “ler mais”, “retweetar o post” e “escrever/ler comentários”.

Note também que o segundo post exibido vem com uma classe a mais: “ext”. Isso porque essa classe servirá para exibir a imagem do post à direita na listagem, dinamizando assim o efeito final.

**Listagem 6.** Exemplo de template simples para páginas web.

```
01 <article class="side-box arquivos">
02   <header>
03     <h2>Arquivos</h2>
04     <p>históricos de posts</p>
05   </header>
06   <section>
07     <ul>
08       <li><a href="#">Agosto 2015</a></li>
09       <li><a href="#">Julho 2015</a></li>
10       <li><a href="#">Junho 2015</a></li>
11       <li><a href="#">Maio 2015</a></li>
12       <li><a href="#"> ... </a></li>
13     </ul>
14   </section>
15 </article>
16 <article class="pop-out side-box">
17   <header class="post-it">
18     <h2>Comentários</h2>
19     <p>veja o que os outros estão falando...</p>
20   </header>
21   <section>
22     <ul>
23       <li>
24         <p>Muito bom, parabéns!</p>
25         <p class="meta">Por DevMedia sobre "Introdução ao JavaScript"</p>
26       </li>
27       <li>
28         <p>Muito bom, parabéns!</p>
29         <p class="meta">Por DevMedia sobre "Introdução ao JavaScript"</p>
30       </li>
31       <li>
32         <p>Muito bom, parabéns!</p>
33         <p class="meta">Por DevMedia sobre "Introdução ao JavaScript"</p>
34       </li>
35     </ul>
36   </section>
37 </article>
```

Nessa parte da tela teremos duas divisões: na primeira exibiremos uma lista com os meses do arquivo do blog; e na segunda uma lista com os últimos comentários. Para facilitar o trabalho no design do CSS em relação à divisão de ambas as partes vamos colocá-las cada uma em uma tag `<article>` separada.

Na primeira (linhas 1 a 15), criamos um `header` com título e subtítulo (dessa vez em um parágrafo) e uma `section` com a referida lista. É importante também definir as classes CSS `side-box` (a mesma da segunda parte, usada para impor um estilo equivalente entre as duas) e `arquivos` (para configurações específicas dessa seção). Na segunda (linhas 16 a 37), definimos também

um `header` (de classe `post-it`, isso porque o criaremos no formato de um post-it comum para dar um efeito mais pessoal) e uma `section` que guardará cada um dos itens de comentário. A classe `meta` nos parágrafos dessa seção definirá um estilo diferente para o autor do comentário.

## Embutindo o estilo orientado a objetos

Uma vez com a estrutura da página montada, podemos agora nos voltar para a definição do CSS. Comecemos pela definição do design dos textos da página. Para cada tipo de recurso diferente que for manipulado pelo CSS, é importante que o desenvolvedor/designer separe as responsabilidades em arquivos diferentes para evitar assim confusão na hora de encontrar cada um. Portanto, abra o arquivo `texto.css` e adicione o conteúdo presente na **Listagem 7** ao mesmo. Veja que ele é bem simples e lida apenas com as propriedades de fonte (tamanho, cor, família, etc.), margens e disposição do texto nos elementos principais das páginas.

**Listagem 7.** Conteúdo CSS do arquivo `texto.css`.

```
body {
  font: 13px/1.6 Lora, Helvetica, Arial, FreeSans, sans-serif;
}

h1, h2, h3, h4, h5, h6 {
  color: #333;
}

h1 {
  font-size: 50px;
  text-shadow: 1px 1px 0 #fff;
  font-family: Arvo, arial black, arial;
}

h2 {
  font-size: 23px;
  font-family: 'Cabin';
}

h3 {
  font-size: 21px;
}

h4 {
  font-size: 19px;
}

h5 {
  font-size: 17px;
}

h6 {
  font-size: 15px;
}

p, h1, h2, h3, h4, h5, h6, ul {
  margin-bottom: 20px;
}

article, aside, dialog, figure, footer, header, hgroup, menu, nav, section {
  display: block;
}
```

Essa listagem ainda não traz nenhum código orientado a objetos, trata-se apenas de um conteúdo necessário para organizar as tags principais das páginas do nosso site. Além disso, antes de adentrarmos nesse assunto precisamos definir mais um conteúdo desse tipo no outro arquivo *style.css*. Portanto, abra-o e inclua o conteúdo mostrado na **Listagem 8** ao mesmo. A ideia é editar este arquivo aos poucos, entendendo seu conteúdo parte por parte para não confundir os conceitos, visto que seu tamanho é muito grande.

**Listagem 8.** Conteúdo CSS inicial do arquivo *style.css*.

```
body {  
    background-color: #F2F5F7;  
}  
  
a {  
    text-decoration: none;  
    color: #474747;  
    padding: 5px;  
}  
  
a:hover {  
    background: #8cc53e;  
    color: #fff;  
}  
  
.selected {  
    border-bottom: 1px solid #8cc53e;  
}  
  
ul {  
    list-style: none;  
    padding-left: 0;  
}  
  
.arquivos section ul {  
    list-style-image: url(..//img/bullet.png);  
}  
  
li {  
    padding-left: 15px;  
}
```

Nesta listagem, em específico, definimos alguns estilos básicos como cores de fundo, links, bordas, alinhamento, margem e *padding* dos elementos de lista, corpo e links da página. Destaque especial para a propriedade *list-style-image* que configura a imagem substituta dos pontos nas listas não-ordenadas (<ul>) da seção de arquivos do blog.

Agora, vamos de fato começar a mexer com código orientado a objetos. Para isso, adicione o conteúdo da **Listagem 9** logo abaixo do anterior na *style.css*. Veja que todas as regras nela iniciam pelo seletor de id “container”, definindo valores diversos como cores, margens, tamanhos, planos de fundo, etc. Para todas as demais regras, usamos o famoso **seletor filho “>**. Esse seletor estabelece uma hierarquia automática entre o primeiro e segundo elementos e se encarrega de vasculhar em seus seletores filhos que se encaixem na condição. Por exemplo, no primeiro caso de uso dele temos a regra *#container > header*. A condição aqui diz que o estilo só será aplicado aos elementos *header* que

forem filhos de *#container*. Um exemplo mais complexo seria o caso de *#container > footer li:last-child:after*, onde a regra se aplica somente ao indicador *:after* dos últimos <li>'s de cada <ul>/<ol> que estiverem contidos dentro de algum *footer* e forem filhos de qualquer elemento de id “container”. O uso desse seletor, quando usado corretamente, automaticamente implica no uso de OOCSS em detrimento do recurso de herança que o mesmo implementa por padrão e cujo conceito pertence às diretivas da orientação a objetos.

**Listagem 9.** Conteúdo CSS para estilo do container no *style.css*.

```
#container {  
    margin: 40px auto;  
    width: 960px;  
    border: 1px solid #ccc;  
    background: #ececfc;  
}  
  
#container > header, #container > footer {  
    padding: 80px 80px 80px;  
    width: 800px;  
    overflow: hidden;  
    border: 1px solid #ccc;  
    border-width: 1px 0 1px 0;  
}  
  
#container > header {  
    background-color: #AAEA52;  
}  
  
#container > header li, #container > footer li {  
    float: left;  
    padding: 5px;  
    background: none;  
    font-family: Arvo;  
    text-transform: uppercase;  
    font-size: 12pt;  
}  
  
#container > section {  
    background: #fdfdfd;  
    padding: 0 40px 40px 80px;  
    float: left;  
    width: 493px;  
    border-right: 1px solid #ccc;  
}  
  
#container aside {  
    float: left;  
    width: 346px;  
}  
  
#container > footer {  
    padding: 30px 120px 30px 40px;  
    background: #fcfcfc;  
}  
  
#container > footer li:after {  
    content: "|";  
}  
  
#container > footer li:last-child:after {  
    content: "";  
}
```

# Introdução ao desenvolvimento de CSS orientado a objetos

Também é possível fazer uso dos mesmos recursos nos itens de posts do blog. Acrescente, portanto, o CSS da **Listagem 10** ao final do arquivo para tal. Veja que quando não fazemos uso do seletor de herança o CSS mapeará as características para o primeiro item que encontrar, e não todos. Por isso, certifique-se de mapear as regras certas para cada finalidade. Sobre a listagem, perceba que além das propriedades básicas, algumas se destacam para configurar o estilo individual de cada componente na página:

- **Classe .date**: se encarrega de girar o elemento em 90° para que a data apareça na vertical (via propriedade *transform*), além de posicionar o mesmo alguns pixels à esquerda para não sobrepor o texto principal.
- **Classe .ext**: define que a imagem que tiver como elemento pai essa classe deverá flutuar (*float*) à direita da página.

Vejamos agora o que torna este CSS orientado a objetos. Em primeiro lugar, não limitamos a classe a um elemento específico, já que podemos adicioná-la a qualquer coisa. Isso nos dá a

**Listagem 10.** Conteúdo CSS para estilo dos posts no style.css.

```
.post {  
    overflow: visible;  
    margin-top: 40px;  
    border-bottom: 1px solid #8cc53e;  
    padding-bottom: 20px;  
}  
  
.post > header {  
    margin: 0 0 20px 0;  
    position: relative;  
}  
  
.post .date {  
    padding: 2px 4px;  
    background: #8cc53e;  
    color: white;  
    font-weight: bold;  
    transform: rotate(270deg);  
    -moz-transform: rotate(270deg);  
    -webkit-transform: rotate(270deg);  
    position: absolute;  
    top: 60px;  
    left: -105.5px;  
}  
  
.post img {  
    float: left;  
    margin-right: 10px;  
}  
  
.post.ext img {  
    float: right;  
    margin: 0 0 0 10px;  
}  
  
.post footer {  
    overflow: hidden;  
}  
  
.post footer li {  
    float: left;  
    background: none;  
}
```

maior flexibilidade possível para trabalhar com estes elementos. Em seguida, perceba que não definimos quaisquer alturas ou larguras na folha de estilo; isso faz parte da separação da estrutura do *skin* que conversamos antes. Uma vez que já escrevemos o estilo da estrutura, este objeto irá preencher todo o espaço que a mesma estrutura lhe der.

Além disso, estilizamos todos os elementos envolvidos de forma independente: os elementos pai não exigem certos elementos filhos para que seu estilo esteja ok; e, embora tenhamos denominado elementos filho, nada deixará de funcionar se eles não estiverem lá. Os elementos filhos são, na sua maior parte, não dependentes de seus pais.

Isso já será o suficiente para definir o design padrão da página no que se refere a cabeçalho e lista de posts. Salve todos os arquivos e execute o index.html em um browser qualquer. O resultado será semelhante ao das **Figuras 1 e 2**.



**Figura 1.** Cabeçalho da página index.html no browser

**Introdução ao JavaScript**

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsum voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consecetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

20 de Janeiro de 2015

Leia Mais... Retweet! Comentários (4)

**CSS Orientado a Objetos**

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsum voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consecetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

12 de Janeiro de 2015

Leia Mais... Retweet! Comentários (4)

**Front-End Devs - DevMedia**

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsum voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consecetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem.

Leia Mais... Retweet! Comentários (4)

**Figura 2.** Listagem de posts no corpo da página index.html

Com a simples adição de outra classe ao seu objeto é possível alterar partes menores do *look and feel* da página. Nicole Sullivan, em suas bibliografias e palestras, sempre menciona a criação de uma classe chamada `.postExt` que, em seguida, deve ser aplicada ao objeto. Essa estratégia é interessante porque nos permite usar o mesmo nome da classe (“`ext`”) para todas as extensões, além de deixar o HTML mais limpo. Você apenas tem que se lembrar de não colocar um espaço no seletor: “`.post.ext`”, pois desta forma ele irá procurar por um elemento de classe `ext` dentro de um elemento de classe `post`. Sem o espaço, ele irá procurar por um elemento com ambas as classes.

## Criando a Sidebar

Agora que temos a área de conteúdo principal definida, vamos implementar a barra lateral do blog. Teremos dois objetos na barra lateral: uma lista de arquivos e uma lista de comentários recentes. Para começar, vamos criar uma classe `.side-box` para eles adicionando o conteúdo da **Listagem 11** ao final do `style.css`.

**Listagem 11.** Conteúdo CSS para o painel lateral.

```
.side-box {  
    padding: 20px 80px 20px 40px;  
}  
  
.side-box:not(:last-child) {  
    border-bottom: 1px solid #ccc;  
}  
  
.side-box > header h3 {  
    margin-bottom: 0;  
}  
  
.side-box > header p {  
    text-transform: uppercase;  
    font-style: italic;  
    font-size: 90%;  
}
```

Mais uma vez estamos tomando cuidado para seguir as duas regras: separamos a estrutura do *skin*, não definindo a altura ou largura; assim, o objeto flui para preencher o espaço que ele precisar. Além disso, separamos o container do conteúdo para fazer com que os elementos filhos não sejam necessários para o estilo apropriado. Veja o uso que fizemos do elemento `:last-child` na segunda regra da listagem; este elemento busca o último elemento filho que atenda à condição, porém, como estamos usando-o dentro da função `not()`, automaticamente a regra é negada e passa a analisar todos os filhos, exceto o último. O uso de funções (ou métodos) é extremamente importante para manter a ordem na orientação a objetos. Para finalizar, ajustamos o estilo do `h3` tornando o mesmo dependente da classe `side-box`.

Essa implementação mostrará os comentários apenas em uma lista convencional abaixo do arquivo. Todavia, para dar um ar mais pessoal a esta seção, vamos envolvê-la em uma box de fundo branco e colocar o título com um efeito, simulando um post-it convencional. Para isso, adicione o conteúdo da **Listagem 12** ao final do mesmo arquivo.

**Listagem 12.** Conteúdo CSS para o box dos comentários.

```
.pop-out > section > * {  
    display: block;  
    background: #fefefe;  
    border: 1px solid #8cc53e;  
    padding: 10px;  
    position: relative;  
    width: 110%;  
}
```

Observe que estamos lidando com uma implementação diferente das que fizemos antes. Note que o seletor está basicamente mapeando todo o conteúdo (\*) da `section` que estiver dentro do nosso elemento de classe `pop-out`.

Talvez o leitor esteja se perguntando: você definiu uma largura, mas e aquele conceito de separar a estrutura do *skin*? Neste caso, isso se fez necessário porque precisamos mapear os elementos dentro do corpo do objeto e, uma vez que o objeto tem um `padding` interno, o elemento interior é um pouco estreito por si só. No fim das contas, mesmo configurando dimensões para o objeto em questão, ainda mantivemos a proporção que a percentagem (%) nos proporciona, portanto, as regras continuam valendo e a nossa estrutura OOCSS não foi quebrada.

Trabalhar com OOCSS é o mesmo que aplicar sempre classes aos objetos de nível superior e usar seletores filhos para moldar os elementos internos. Mas parte do CSS orientado a objetos é capaz de misturar e combinar estilos facilmente. É bem possível que você queira utilizar um cabeçalho semelhante em dois objetos que não são realmente relacionados de alguma forma. É ideal, portanto, ter uma coleção de classes de cabeçalhos e rodapés que você possa aplicar diretamente a tais objetos. Dessa forma, você não estará adicionando um código semelhante a várias classes não relacionadas, mas abstraindo isso fora e aplicando nos lugares relevantes. Vamos criar um cabeçalho abstruído, por exemplo, para o caso do `post-it` que mencionamos antes.

Você notará que demos ao cabeçalho do nosso objeto de comentários recentes uma classe chamada “`post-it`”. Vamos criar essa classe agora tal como é demonstrado na **Listagem 13**.

Veja que aqui temos alguns operadores já usados antes em outros estilos, como o `transform` para rotação, por exemplo. Algumas propriedades precisam ser especificadas para aproximar o design de um `post-it` real: como a cor do texto e do fundo (esta última pode ser qualquer uma), o texto em itálico, a sombra ao redor do mesmo para dar o efeito de que está um pouco descolado do papel e, sobretudo, a propriedade `z-index` que aplica um fator numérico ao elemento informando ao browser a sua importância em relação aos outros, sobrepondo-o. As classes `meta` e `subtitle` aplicam estilo aos elementos internos do texto do comentário e do autor em si, respectivamente.

É importante salientar que mesmo que nós planejemos usar isso para um cabeçalho, não o especificamos no seletor. Se essa regra se torna um aspecto comum do projeto do site, podemos acabar querendo este estilo para algo mais, para outros lugares.

# Introdução ao desenvolvimento de CSS orientado a objetos

Por exemplo, podemos estendê-lo com classes que ajustam a cor e a rotação e usá-los em um plano de fundo. É importante lembrar sempre de não prender os seletores a regras ímpares, pois nunca sabemos quando precisaremos desses estilos!

**Listagem 13.** Conteúdo CSS para o cabeçalho de post-it.

```
.post-it {  
    border: 1px solid #8cc53e;  
    padding: 10px;  
    font-style: italic;  
    position: relative;  
    background: #FFDD00;  
    color: #333;  
    transform: rotate(356deg);  
    -moz-transform: rotate(356deg);  
    -webkit-transform: rotate(356deg);  
    z-index: 10;  
    top: 10px;  
    box-shadow: 1px 1px 0px #333;  
    -moz-box-shadow: 1px 1px 0px #333;  
    -webkit-box-shadow: 1px 1px 0px #333;  
}  
  
.meta {  
    font-size: 75%;  
    font-style: italic;  
}  
  
.subtitle {  
    text-transform: uppercase;  
    font-size: 90%;  
    font-weight: bold;  
    letter-spacing: 1px;  
    text-shadow: 1px 1px 0 #fff;  
}
```

Muitas vezes você vai querer ir mais longe do que criar apenas classes de cabeçalho e rodapé; uma biblioteca de componentes é uma parte enorme do CSS orientado a objetos; e é essa a regra básica de ser abstrato: aplicar o mesmo estilo em vários lugares não relacionados, de forma abstrata. Obviamente, um dos maiores e melhores trabalhos em toda a orientação a objetos, independente da linguagem, é a refatoração. Se você reconhece que seu código não está bom e/ou não atende às especificações do OOCSS, então reserve sempre um bom tempo em seus projetos web para refatorar seu código e adequá-lo melhor às diretrizes. Agora, execute a página novamente no browser e o resultado será semelhante ao que temos na **Figura 3**.

## Criando página de post

Agora que finalizamos o estilo da página principal, vamos nos voltar para a implementação da página de post que vai trabalhar um pouco mais o conceito de objetos aninhados. Nessa página vamos implementar a opção de enviar um comentário, bem como exibir os últimos enviados, logo, os componentes precisarão ter uma espécie de comunicação hierárquica uns com os outros. Para isso, no arquivo `post.html` replique o conteúdo da `index.html` e substitua o corpo da `section` principal pelo apresentado na **Listagem 14**. Veja que esta página trará as mesmas configurações

genéricas da index, incluindo a mesma barra lateral, o mesmo cabeçalho com menus e o mesmo rodapé. Inclusive o próprio conteúdo será representado também pelo `<article>` de classe `post`. No `footer` de cada post agora teremos três links para compartilhar nosso post nas redes sociais (linhas 16 a 18).



**Figura 3.** Barra lateral do site com arquivo do blog e últimos comentários

A grande novidade fica a cargo da listagem e formulário de comentários. Para a lista, note que usamos exatamente as mesmas classes CSS que criamos para a exibição na barra lateral, com exceção das classes `reply` (que representa os comentários em resposta a um já existente, aninhados) e `author` (que representa os comentários do autor). Para o formulário, criamos campos simples com uma `label` associada a cada `input` (ambos dentro de um parágrafo) e um botão `submit` no final.

## Comentários

Comentários são um ótimo lugar para usar CSS orientado a objetos por causa dessa característica nata de serem aninhados. A nível de CSS, a **Listagem 15** traz todo o conteúdo que precisaremos adicionar ao `style.css` para os mesmos. Vamos começar a análise pela regra definida na linha 43: nela definimos um seletor de `label` dentro do nosso formulário para o `:after`. Veja que alteramos o conteúdo da `label`, inserindo os dois-pontos após o texto principal, dessa forma não precisamos deixar essa

**Listagem 14.** Conteúdo HTML da página post.html.

```
01 <section>
02   <article class="post">
03     <header>
04       <span class="date">20 de Janeiro de 2015</span>
05       <h2>Introdução ao JavaScript</h2>
06     </header>
07   </section>
08   
10   <p>O surgimento da web ocorreu num cenário no qual os recursos
11     computacionais eram escassos e limitados. Ainda não se tinha
12     conhecimento das
13     dimensões que essa poderosa ferramenta um dia atingiria.</p>
14   HTML e o primeiro navegador, a internet que antes era apenas voltada para
15     comunicação entre universidades e instituições militares, se deparou com um
16     novo mundo para se expandir, sendo nos dias atuais um dos meios mais
17     utilizados para entretenimento, pesquisa, transações comerciais e muitas
18     outras atividades.</p>
19   <p>A evolução da internet e a mudança de seus propósitos trouxeram
20     também novas necessidades. Era preciso tornar mais dinâmicas as
21     páginas que inicialmente eram construídas apenas com conteúdo
22     estático – tags
23   HTML dispondo imagens, textos e links para navegar entre os documentos
24     na rede.</p>
25   <p>Neste contexto, a Netscape, em parceria com a Sun Microsystems, com a
26     finalidade de adicionar mais interatividade às páginas web,
27     criou a primeira versão do JavaScript, denominada JavaScript 1.0.</p>
28 </section>
29 <footer>
30   <ul>
31     <li><a href="#">Facebook</a></li>
32     <li><a href="#">Twitter</a></li>
33     <li><a href="#">Google+</a></li>
34   </ul>
35 </footer>
36 </article>
37 <article class="comments">
38   <header>
39     <h2>Comentários</h2>
40   </header>
41   <section>
42     <article class="comment">
43       <header>
44         <p>Primeiro Comentário</p>
45         <p class="meta">Ago 10</p>
46       </header>
47       <p>Gostaria de saber se vocês tem alguma forma de implementar JS de
48         outro jeito?</p>
49     </article>
50   </section>
51   <article class="comment">
52     <header>
53       <p>Segundo Comentário</p>
54       <p class="meta">Ago 10</p>
55     </header>
56   <section>
57     <p>Gostaria de saber se vocês tem alguma forma de implementar JS de
58       outro jeito?</p>
59   </section>
60   <article class="author comment">
61     <header>
62       <p>O autor</p>
63       <p class="meta">Ago 11</p>
64     </header>
65   <section>
66     <p>Gostaria de saber se vocês tem alguma forma de implementar JS de
67       outro jeito?</p>
68   </section>
69   <article class="comment">
70     <header>
71       <p>Quarto Comentário</p>
72       <p class="meta">Ago 12</p>
73     </header>
74   <section>
75     <p>Gostaria de saber se vocês tem alguma forma de implementar JS de
76       outro jeito?</p>
77   </section>
78 </article>
79 <h2>Deixe um comentário</h2>
80 <section>
81   <form class='comments-form'>
82     <p>
83       <label for='nome'>Nome</label>
84       <input type='text' id='nome' />
85     </p>
86     <p>
87       <label for='email'>Email</label>
88       <input type='email' id='email' />
89     </p>
90     <p>
91       <label for='comentario'>Comentário</label>
92       <textarea id='comentario' rows='4'></textarea>
93     </p>
94     <p>
95       <button type='submit'>Enviar</button>
96     </p>
97   </form>
98 </section>
99 </section>
```

configuração a cargo da aplicação que pode trazer tais labels de qualquer lugar. Essa é uma forma de dinamizar o HTML via CSS e não via JavaScript como comumente é feito. Na linha 20, mais uma vez, fazemos uso dos seletores sem o espaço em branco entre os mesmos. Isso é importante para que ele busque por todos os filhos do primeiro elemento. Na linha 47 definimos uma mesma lista de regras para várias seleções distintas. Isso se assemelha

à declaração de variáveis globais nas linguagens OO, se você precisa de uma mesma regra para vários elementos, mapeie-os um após o outro separando por vírgulas. O mesmo valeria para o botão de *submit* (linha 60) caso ele tivesse um irmão do tipo *reset*, por exemplo.

O resultado final da página pode ser visualizado nas **Figuras 4, 5 e 6**.

# Introdução ao desenvolvimento de CSS orientado a objetos

The screenshot shows a blog post titled "Introdução ao JavaScript". The post content discusses the history of the web and the development of JavaScript. It includes a sidebar with a "Comentários" section and a "Arquivos" section showing a list of posts from August 2015 to May 2015.

Figura 4. Visualização da página de post

The screenshot shows a list of comments for a blog post. The comments are displayed in a threaded format. The first comment is from "Primeiro Comentário" (Ago 10) and the second is a reply from "Reply" (Ago 12). Below them are two more comments from "Segundo Comentário" (Ago 10) and "O autor" (Ago 11), respectively. All comments contain the same text: "Gostaria de saber se vocês tem alguma forma de implementar JS de outro jeito?"

Figura 5. Visualização da lista de comentários na página de post

The screenshot shows a form for leaving a comment. It includes fields for "Nome:" (DevMedia), "Email:" (devmedia@dev.com), and a "Comentário:" text area containing the text "Testando 1, 2, 3...". A green "Enviar" button is at the bottom.

Figura 6. Visualização da página de post

**Listagem 15.** Conteúdo CSS para os comentários do post.

```
01 .comment {  
02   border: 1px solid #ccc;  
03   border-radius: 7px;  
04   -moz-border-radius: 7px;  
05   -webkit-border-radius: 7px;  
06   padding: 10px;  
07   margin: 0 0 10px 0;  
08 }  
09  
10 .comment > header > p {  
11   font-weight: bold;  
12   display: inline;  
13   margin: 0 10px 20px 0;  
14 }  
15  
16 .reply.comment {  
17   margin-left: 80px;  
18 }  
19  
20 .author.comment {  
21   color: #ececce;  
22   background: #474747;  
23 }  
24  
25 .comments-form p {  
26   font-family: Arvo;  
27   padding: 5px;  
28   border-radius: 5px;  
29   -moz-border-radius: 5px;  
30   -webkit-border-radius: 5px;  
31 }  
32  
33 .comments-form p:hover {  
34   background: #ececce;  
35 }  
36  
37 .comments-form label {  
38   display: inline-block;  
39   width: 100px;  
40   vertical-align: top;  
41 }  
42  
43 .comments-form label:after {  
44   content: ":";  
45 }  
46  
47 .comments-form input[type=text],  
48 .comments-form input[type=email],  
49 .comments-form button,  
50 .comments-form textarea {  
51   width: 200px;  
52   font-family: Arvo;  
53   border: 1px solid #ccc;  
54   border-radius: 5px;  
55   -moz-border-radius: 5px;  
56   -webkit-border-radius: 5px;  
57   padding: 2px;  
58 }  
59  
60 .comments-form button[type=submit] {  
61   margin-left: 70px;  
62   color: white;  
63   background-color: #8cc53e;  
64 }
```

Devemos lembrar, contudo, que CSS orientado a objetos não é uma linguagem, mas uma coleção de padrões e boas práticas. E em sua grande maioria, os princípios que vimos aqui são sempre válidos. O segredo é pensar o que é realmente necessário de se otimizar: você pode codificar o HTML minuciosamente com alguns ids para lidar com o mesmo e ser capaz de estilizá-lo facilmente; mas o CSS não será reutilizável, e o site pode quebrar mais tarde quando você mudar algumas coisas.

## OOCSS + Sass

CSS orientado a objetos é muito útil, mas não diríamos o mesmo do desarranjo da sua marcação com classes não-semânticas. Essas classes polvilhadas por todo o seu HTML com certeza vão mudar e se isso trouxer dores de cabeça junto, então seu CSS não foi corretamente codificado. Entretanto, se combinarmos o OOCSS com o Sass obteremos o melhor dos dois mundos: CSS modular sem inchaços e HTML facilmente manutenível.

Comecemos pela diretiva `@extend` no Sass, por exemplo, que nos permite herdar estilos de outro seletor sem duplicar tudo como um *mixin*. Até mesmo as chamadas a `@extend` podem causar inchaço no código se você as aninhar ou usá-las com seletores aninhados.

Felizmente, a última versão do Sass (vide seção **Links**) já conta com um recurso chamado *placeholders*. Tratam-se de seletores que retornam nada menos que suas próprias extensões. Vejamos na **Listagem 16** um exemplo básico de placeholder.

**Listagem 16.** Exemplo básico de placeholder no Sass.

```
%separador
  color: red

hr
  @extend %separador

.separador
  @extend %separador
```

Esse código geraria o seguinte CSS:

```
hr, .separador {
  color: red;
}
```

Veja como foi fácil implementar o conceito OO de herança apenas especificando um componente pai e definindo quais componentes filhos o estenderão. Esse tipo de recurso é perfeito para criar módulos CSS não-semânticos que, no universo Sass, são chamados de “patterns” (padrões).

Vejamos um exemplo mais real: considere um módulo de nome `.teste`. Você provavelmente aplicará este módulo a vários de seus componentes: `.usuario`, `.perfil`, etc. O ponto é que não será mais necessário ter de repetir `.teste` em todo o seu HTML. Especialmente porque você já vai ter de repetir `.usuario` e `.perfil` em vários lugares distintos. Ao usar os placeholders, nosso pattern ficaria tal como demonstrado na **Listagem 17**.

Agora, em vez de ter que repetir `.teste` em todos os seus elementos, você precisa apenas estender o padrão de `%teste` em qualquer lugar que quiser usá-lo (veja o exemplo da **Listagem 18**).

**Listagem 17.** Pattern do Sass para exemplo com `.teste`.

```
%teste
  color: red
  &:first-child
    width: 100%
  &:last-child
    color: red
```

**Listagem 18.** Exemplo de uso do pattern criado com o placeholder.

```
.usuario
  @extend %teste
  // Estilos específicos de usuario aqui...

.perfil
  @extend %teste
  // Estilos específicos de perfil aqui...
```

Isso significa que em seu HTML você só precisa adicionar as classes semânticas: `.usuario` e `.perfil`. Além disso tudo, ainda ganhamos em flexibilidade, já que se você precisar alterar a forma como o usuário enxerga o módulo `.teste` basta remover a chamada ao `@extend` e pronto!

Mas para que estas regras sejam devidamente aplicadas, é necessário que as adicionemos aos patterns. Existem diversos tipos diferentes de patterns no mercado e você pode criar os seus próprios.

No blog, não criamos nenhum mecanismo de exibição de mensagens para o usuário, mas isso é realmente uma ótima maneira de aprender mais sobre o CSS orientado a objetos. Portanto, tente criar uma classe de mensagens na qual você possa aplicar diferentes elementos: como um parágrafo, uma lista de itens, ou um objeto completo. Em seguida, estenda-o a classes de erro, aviso e informação. Veja o quanto flexível e reutilizável você pode fazê-lo e, assim, vá praticando suas habilidades em OOCSS. Essa é a melhor forma de se habituar às diretrizes do OOCSS e ir acostumando sua metodologia de programação a lidar com ele. É sempre complicado migrar para uma nova maneira de programar quando se está acostumado a trabalhar de forma estruturada e sem tantas regras semânticas. É por isso que a prática se faz cada vez mais importante. Bons estudos!

## Autor



**Júlio Sampaio**

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



# Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

**Porta 80** é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz, assim como você.



## Estrutura

100% NACIONAL.  
Servidores de primeira linha, links de alta capacidade.



## Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



## Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



## 1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.  
**Conheça!**



# Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |  
Aplicações | Streaming | Email corporativo

[porta80.com.br](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486