

OS 3 CONCEITOS FUNDAMENTAIS DE CSS

PARA ENTENDER COMO AS FOLHAS DE
ESTILO REALMENTE FUNCIONAM



DESENVOLVIMENTOPARAWEB.COM

Isenção de Responsabilidade

Todas as informações contidas neste livro são provenientes de nossas experiências pessoais e profissionais com o aprendizado e participação em projetos de desenvolvimento web ao longo de vários anos de estudos e práticas. Embora tenhamos nos esforçado ao máximo para garantir a precisão e a mais alta qualidade dessas informações -- e, acredite, todas as técnicas e métodos aqui ensinados são altamente efetivos para qualquer estudante de desenvolvimento web, desde que seguidos conforme instruídos --, nenhum dos métodos ou informações foi cientificamente testado e/ou comprovado (talvez isso nem seja possível) e não nos responsabilizamos por erros ou omissões. Sua situação e/ou condição particular pode não se adequar perfeitamente aos métodos e técnicas ensinados neste livro. Assim, talvez você deva utilizar e ajustar as informações aqui presentes de acordo com sua própria situação e necessidades.

Todos os nomes de marcas, produtos e serviços mencionados neste livro são propriedades de seus respectivos donos e são usados somente como referência. Além disso, em nenhum momento há a intenção de difamar, desrespeitar, insultar, humilhar ou menosprezar você, leitor, ou qualquer outra pessoa, cargo ou instituição. Caso qualquer conteúdo seja interpretado dessa maneira, gostaríamos de deixar claro que não houve intenção nenhuma de nossa parte em fazer isso. Caso você acredite que alguma parte deste livro seja de alguma forma desrespeitosa e/ou indevida e deva ser removida ou alterada, pode entrar em contato diretamente conosco através do e-mail dpw@webfatorial.com.

Direitos Autorais

Este livro está protegido por leis de direitos autorais. Todos os direitos sobre o livro são reservados. Você não tem permissão para vender este livro nem para copiar/reproduzir seu conteúdo em sites, blogs, jornais ou quaisquer outros veículos de distribuição e mídia. Qualquer tipo de violação dos direitos autorais estará sujeito a ações legais.

Um pouquinho sobre o dpw

O blog **desenvolvimento para web** (carinhosamente conhecido como **dpw**) compartilha gratuitamente conteúdos e recursos sobre desenvolvimento web, como este que você está lendo exatamente agora.

Inicialmente, ele surgiu como um blog pessoal; era escrito mais como “hobby” e concentrador de recursos para uso próprio. Assim como você, no princípio, buscávamos conteúdos que estavam esparsos pela internet. Nós realmente os achávamos, mas demorava bastante para separar os bons dos ruins. Uma das principais funções do blog era ser a coletânea só dos bons.

Isso continua acontecendo até hoje, mas já há muitos anos, entretanto, ele evoluiu: tornou-se um blog empresarial; ganhou estrutura e gerenciamento profissionais; expandiu-se; alcançou presença forte no cenário de webdev nacional.

Hoje, é uma das melhores e mais tradicionais fontes de recursos e consultas sobre desenvolvimento web do Brasil.

Então, as dicas que estão aqui não são "aleatórias" ou foram colocadas sem um propósito. Neste livro, não consta outra lista que você poderia encontrar em qualquer site de fundo de quintal da internet. Essas dicas realmente funcionam se você

se comprometer a estudar com dedicação e disciplina; elas são altamente efetivas para qualquer estudante de desenvolvimento web que as leve em consideração em seus estudos.

Esperamos que este singelo presente do **dpw** seja útil e ajude você a adquirir conhecimentos e melhorar suas habilidades.

É com esse objetivo que há mais de **14 anos** compartilhamos nossos conteúdos, conseguindo ajudar e cooperar com mais de **3,5 milhões de pessoas!**

E não pretendemos parar. :-)

<https://desenvolvimentoparaweb.com/>

ATENÇÃO

Sabia que é possível alcançar níveis mais avançados e maduros na carreira de front-end através da estruturação do CSS de maneira profissional?

Para saber mais, veja o link no final do livro.

Quando você começou a aprender CSS?

Se está no início de sua carreira como desenvolvedor front-end, talvez consiga responder a esta pergunta mais facilmente; mas, se já se passou algum tempinho, poucos anos que seja, talvez você tenha um pouco de dificuldade em responder.

Há muitos anos atrás, quando, pela primeira vez, tive contato com CSS -- a primeira proposta oficial de CSS foi feita em 1994, pelo norueguês Håkon Wium Lie, mas eu só fui mesmo conhecer lá pelos anos 2000 -- foi um verdadeiro choque para mim.

As coisas eram estilizadas (se é que se pode chamar assim) de maneira muito diferente na era pré popularização de CSS. E, também, o que se podia fazer com CSS era muito pouco perto do que é possível agora.

Dei a sorte de já existir o [site do Maujor](#) naqueles idos, e foi lá mesmo que eu comecei minha “formação em CSS”. Foi assim que eu comecei nas folhas de estilo.

E você? Caso não se lembre quando foi que teve contato com CSS pela primeira vez, não tem problema; mas, talvez, você consiga se recordar de **como** começou a aprender.

No meu caso, foi já vendo um código CSS na frente e tentando entender a sintaxe da coisa e como isso poderia ser melhor que colocar atributos em tags HTML -- o que eu entendi rapidinho.

A partir disso, fui conhecendo outras propriedades, outros valores, outras técnicas, enfim, muito mais possibilidades de usar CSS para usá-lo direitinho para fazer exatamente o que ele tinha sido criado para fazer.

O motivo de contar essa história e tentar fazer você recordar um pouco do seu passado como desenvolvedor é o seguinte: se aconteceu com você algo parecido com o que aconteceu comigo, então você foi introduzido no mundo das folhas de estilo já vendo uma regra CSS, com seu seletor, propriedades e valores.

Claro que foi muito bom você ter começado de alguma maneira. Do contrário, talvez nem estivesse lendo este Livro em busca de como se aprimorar, certo?

Mas começar a estudar CSS por suas propriedades e valores talvez não seja a melhor maneira de começar.

Ou melhor: talvez não seja a melhor maneira de começar e continuar, a partir daí, sem determinado aprofundamento teórico.

Claro que, no começo, a gente quer mesmo é conhecer as propriedades, colocar cores, alterar as fontes, criar blocos e layouts. E não há nada de errado nisso.

Mas é tão importante quanto -- alguns diriam até mais -- também saber o que acontece por detrás das cortinas; o que há por detrás daqueles tapumes de madeira fina que sustentam a beleza da propaganda no palco.

Você tem que conhecer determinados conceitos fundamentais para entender como as folhas de estilo realmente funcionam.

Nada mais de coçar a cabeça quando uma regra escrita não entrar em ação ou brincar de tentativa-e-erro para ajustar um layout que, por algum motivo, não está exatamente do tamanho que deveria estar.

Esses conceitos são:

- 1. Cascata;**
- 2. Herança;**

3. Especificidade.

Conhecer esses conceitos fundamentais é essencial para você continuar avançando na carreira. Ao compreendê-los, você começará verdadeiramente a **tomar posse** de CSS.

Há mais de 20 anos atrás, este é o tipo de conteúdo que eu gostaria que tivessem me dito: “estuda isso aqui primeiro; é muito importante”.

Este Livro é nossa maneira de dizer isso para você.

Se ficar com qualquer dúvida ou quiser conversar sobre qualquer assunto abordado, basta enviar uma mensagem em qualquer uma de nossas redes sociais:

- [Blog dpw](#)
- [Facebook](#)
- [YouTube](#)
- [Instagram](#)
- [Twitter](#)

Esperamos que este conteúdo possa ser útil para você.

PRELIMINARES

Antes de entrar nos tópicos principais do livro, propriamente ditos, há alguns assuntos importante a serem vistos.

Trata-se de abordar conhecimentos básicos de CSS que, infelizmente, muitos devs front-end negligenciam -- ou, mais tristemente, sequer ouviram falar.

Recapitulando o que foi visto no capítulo anterior, comentou-se que há um grande problema quando o assunto são os estudos -- e, consequentemente, os profissionais que trabalham -- com folhas de estilos:

Muitos devs não sabem os conceitos fundamentais de CSS.

As causas desse problema podem ser várias; para encontrá-las todas, um estudo à parte deveria ser realizado.

Mas há 2 (fortes) indícios que, certamente, vão ao encontro de explicar o fenômeno:

- 1. Estudos em materiais de baixa qualidade;**
- 2. Pressa de aprender (pular conhecimentos teóricos).**

Este breve Volume não é maculado por nenhuma das 2 possíveis causas. Pelo contrário.

Em relação ao ponto nº 1, procurou-se, na maior medida possível, sintetizar-se os conteúdos propostos, alinhando-os a uma didática de ensino de tecnologias Web que vem se mostrando eficiente ao decorrer de mais de 10 anos de compartilhamento de conteúdos os mais variados.

Evidentemente, não há como passar o conhecimento de 2 décadas de atuação em CSS em algumas dezenas de páginas, mas, tenha certeza, há boas “reverberações” dessa experiência nas páginas que seguem.

Quanto ao ponto nº 2, este Livreto tem como razão de ser justamente passar o conhecimento teórico referente aos 3 conceitos fundamentais de CSS para entender como as folhas de estilo realmente operam.

Esclarecidas possíveis objeções e dado como solução ao que fazer sobre o grande problema apresentado -- ainda que sucinta; não-definitiva -- aprender os 3 conceitos fundamentais para evitar bugs e escrever CSS melhor, agora você vai saber **COMO CSS FUNCIONA**.

Como um navegador renderiza estilos

Mesmo se está começando agora no maravilhoso mundo das folhas de estilo, provavelmente, já faz mais ou menos parte da sua rotina escrever CSS.

Criar regras, com suas propriedades e valores, montando, pedaço a pedaço, a UI (*User Interface*) que precisa ser feita.

Para começar a entender como as folhas de estilo realmente funcionam, um bom primeiro passo é saber como um navegador transforma HTML e CSS em uma página na internet.

Sim, estamos falando de **renderização Web**.

Renderizar é o ato de compilar e obter o resultado final de um processamento digital.

Quando um navegador renderiza um documento (HTML), ele o combina com suas informações de estilo (CSS).

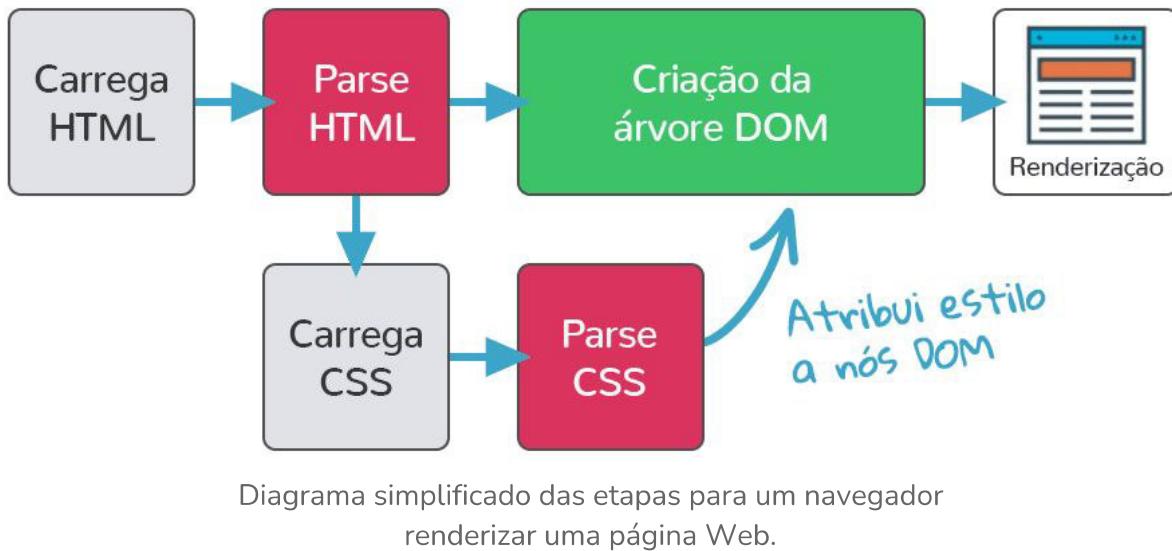
Explicando de maneira simplificada -- e lembrando que cada navegador pode ter suas peculiaridades --, o documento é processado através dos seguintes em estágios ou etapas:

1. O HTML é carregado;
2. O HTML é convertido em DOM (*Document Object Model*), que é uma representação desse documento na memória do computador;
3. O navegador requisita a maioria dos recursos que estão referenciados no documento HTML -- imagens, vídeos, folhas de estilo etc.;
4. O navegador interpreta as diferentes regras do CSS e as atribui a cada respectivo nó do DOM;
5. A árvore de renderização é organizada na estrutura e aparece depois das regras de estilo serem aplicadas ao documento;
6. Finalmente, a parte visual da página é mostrada na tela -- estágio chamado de *painting*.

Como informado, essa a versão simplificada -- beeem simplificada -- do quê acontece.

Uma explicação mais aprofundada até seria possível, mas, provavelmente, não tão proveitosa, tendo em vista que esta é somente uma revisão preliminar para que as explicações dos 3 conceitos fundamentais seja mais bem aproveitada.

Para facilitar o entendimento esquemático da coisa, veja o diagrama a seguir:



O DOM e como navegadores aplicam estilos

Se você já teve a oportunidade de usar ferramentas como Chrome DevTools para inspecionar códigos, certamente, já deve ter visto algo parecido com:

A captura de tela do Chrome DevTools mostra a guia 'Elements' selecionada. O DOM tree é visualizado, com节点高亮显示。节点包括

、aside class="c-tip c-tip--translated c-tip--right wp-block-acf-translated-article"、h2、p等。

```

<div class="c-article__entry-content l-entry-content">
  <p>...</p>
  ...
  <aside class="c-tip c-tip--translated c-tip--right wp-block-acf-translated-article">...</aside>
  <p>...</p>
  <p>...</p>
  <h2>Introdução</h2>
  <p>...</p>
  <div class="wp-block-image">...</div>
  <p>...</p>
  <p>...</p>
  <p>...</p>
  <h2>O que é a Cascata?</h2>
  <...>

```

Isso é o **DOM** (*Document Object Model*): a representação em memória de um documento HTML. Trata-se de uma estrutura

de árvore, em que cada nó do DOM pode ser encarado como uma ramificação; cada elemento, atributo, ou fragmento de texto no HTML se torna um desses nós inter-relacionados.

Cada nó pode ser filho de outro nó e pode ter seus próprios nós-filho, o que dá forma à estrutura que é possível ver.

Por exemplo, o nó que representa a classe

`c-article__entry-content` no HTML tem diversos filhos, `<aside>`, `<p>` e `<div>`; `<aside>`, por sua vez possui nós-filho, com `<p>`, `<a>` e assim por diante.

Esquematicamente, seria algo como:

```
DIV
└ P
   └ LOREM IPSUM
└ ASIDE
   └ A
      └ CLIQUE AQUI
└ P
   └ LOREM IPSUM
└ DIV
   └ IMG
└ P
   └ LOREM IPSUM
└ ...
```

Um navegador não “enxerga” e atua diretamente no documento HTML; ele “enxerga” o DOM, esse monte de

nós-pai e nós-filho e sua relação hierárquica mútua, retornando uma saída visual.

Então, para, efetivamente, aplicar os estilos necessários, o navegador verifica as regras CSS que podem corresponder aos nós presentes na página em questão.

Por exemplo, uma regra deste tipo:

```
p {  
    background-color: lime;  
    border: 1px solid red;  
}
```

Irá aplicar os estilos em cada nó `<p>`, e então pintar (*paint*) o resultado final na tela.

O que acontece se um navegador não entende o CSS encontrado?

Existem muitos navegadores disponíveis e eles não implementam todas os recursos de CSS ao mesmo tempo -- fora o fato de as pessoas não usarem sempre a última versão.

Neste caso, o que acontece se um browser encontra um seletor ou uma declaração CSS que ele não reconhece?

Resposta: **nada!**

O navegador vai simplesmente ignorar o que ele não conhece e partir para a interpretação do restante do CSS.

Isso acontece se algum erro de CSS foi cometido ou alguma propriedade ou valor foi escrito incorretamente -- ou, devido ao constante e incessante fluxo de novas possibilidades de CSS, se tal propriedade ou valor é recente e o browser ainda não o reconhece.

Se você estranhou esse procedimento, pense melhor. Isso é muito bom para quem faz o site e para quem usa.

Não há motivos para um erro que vai bloquear toda a página acontecer e o visitante ficar impossibilitado de acessar -- mesmo que sem todo o visual e funcionalidades que seriam os ideais.

Em outras palavras, é possível usar tecnologia de ponta de CSS sabendo que não haverá bloqueios se um navegador não reconhecer completamente o que está escrito.

Interpretação de seletores

A maioria dos desenvolvedores front-end desconhece alguns conceitos base sobre as folhas de estilo -- este Livro foi criado para ajudar a sanar este gap no mercado brasileiro, diga-se de passagem.

Em função disso, a escrita de CSS no dia-a-dia acaba não resultando em tanta eficiência e performance quanto poderia, caso algumas coisas fossem mais claras.

Por exemplo, você sabia que há uma **eficiência inerente** de seletores CSS?

Eficiência inerente de seletores CSS

Recapitulando rapidamente, uma regra CSS é composta por **seletor, propriedade e valor**.

Por exemplo, nessa regra:

```
div {  
    font-size: 1rem;  
}
```

Você consegue identificar os 3 elementos que a compõe?

São eles:

1. Seletor: `div`;
2. Propriedade: `font-size`;
3. Valor: `1rem`.

Também é possível haver seletores múltiplos e várias definições de estilo simultâneas, como em:

```
h2, p, span {  
    color: lightgreen;  
    font-size: 18px;  
}
```

Neste caso:

1. Seletor: `h2, p, span`;
2. Propriedade/Valor #1: `color: lightgreen`;
3. Propriedade/Valor #2: `font-size: 18px`.

Quer dizer, a regra define cor e tamanho de texto para cada título secundário, parágrafo e span.

Recapitulada a “anatomia” de uma regra CSS, pode ficar mais clara a afirmação de que existe uma eficiência inerente de seletores CSS.

E o que queremos dizer por eficiência é que o navegador terá mais facilidade em interpretar e aplicar os estilos corretamente a este(s) seletor(s).

Em outras palavras: um CSS que é escrito com atenção à eficiência dos seletores de suas regras é um **CSS de melhor performance!**

A eficiência inerente de seletores é a seguinte (do mais eficiente para o menos eficiente):

1. ID;
2. Classe;
3. Tipo;
4. Irmão Adjacente;
5. Descendente Direto;
6. Descendente;
7. Universal;
8. Atributo;
9. Pseudo-classe/Pseudo-elemento.

Quer dizer, o seletor mais eficiente é o de ID; o 2º seletor mais eficiente é o de classe; o 3º, de tipo e; assim por diante.

Caso não esteja associando nome a tipo de seletor, vamos a exemplos bem rapidinhos de cada um deles para刷新 a memória.

Tecnicamente, um ID é um tipo de seletor mais rápido e de mais performance que classe, mas essa diferença é ínfima.

Conforme explicado em nosso artigo “[Não use IDs como seletores em CSS](#)”, um seletor de ID e um seletor de classe mostram pouquíssima diferença no quesito performance.

ID

Herói para uns, vilão para outros, o **ID** (de sintaxe com `#`) foi, por muito tempo, mal compreendido.

Tecnicamente, é o tipo de seletor mais eficiente, mas, infelizmente, um dos tipos mais mal usados e entendidos.

Suas principais características são:

- IDs são únicos (cada página pode ter apenas 1 elemento com determinado ID);
- Cada elemento pode ter, no máximo, 1 ID.

Exs:

```
#l-nav {...}  
#l-hero {...}  
#l-comments {...}
```

Classe

Atualmente, as **Classes** (caracterizadas por seletor com `.`) são usadas muito mais corretamente, tanto em quantidade, quanto em qualidade.

Existem [diferenças entre IDs e classes](#), mas, para o momento, importa saber:

- Pode-se usar a mesma classe em vários elementos;
- Pode-se usar várias classes para um mesmo elemento.

Qualquer informação de estilo que precise ser aplicada a múltiplos elementos em uma página deve ser feita através de classe.

Exs:

```
.c-button {...}  
.c-card {...}
```

```
.c-input { ... }
```

Tipo

Lembra-se do DOM? Seletores de **Tipo** correspondem a elementos pelo nome do nó -- se preferir, pelo nome da tag HTML.

Principalmente quando se está começando, é comum não prever as consequências de estilização ao se optar por seletores de tipo. Portanto, muito cuidado e atenção no uso.

Exs:

```
a { ... }  
button { ... }  
div { ... }
```

Irmão Adjacente

Levando-se em consideração o esquema $E + F$, o seletor **Irmão Adjacente** corresponde se E e F compartilham o mesmo pai e E precede imediatamente F .

A regra de estilo é aplicada ao elemento que é correspondido pelo seletor F .

Exs:

```
h2 + p {...}  
li:first-of-type + li {...}  
input:checked + .c-custom-check {...}
```

Descendente Direto

O seletor **Descendente Direto** (de sintaxe que usa `>`), como sugere o próprio nome, corresponde a qualquer elemento que seja um filho direto (descendente de primeiro nível) de outro elemento.

É causador de muitos bugs em estilos, mas, se usado corretamente, pode ser uma excelente pedida em determinados casos mais difíceis.

Exs:

```
div > a {...}  
#l-hero > .c-button {...}  
.c-card > p {...}
```

Descendente

Um dos mais usados (principalmente por iniciantes), o seletor **Descendente** corresponde a qualquer elemento que seja um descendente (esteja “dentro”) de um elemento, em qualquer nível de sua hierarquia -- na sintaxe, usa-se `>` (espaço).

Também é preciso ter bastante cuidado com esse tipo, já que tem potencial de afetar muitos elementos ao mesmo tempo.

Exs:

```
h1 em {...}  
.c-widget a {...}  
#l-comments .c-button {...}
```

Universal

O seletor **Universal** (com sua inequívoca `*`) aplica estilos a elementos de qualquer tipo.

É bastante usado para aplicar regras genéricas gerais a muitos elementos -- e, por isso mesmo, causador de muitas noites em claro quando usado levianamente.

Exs:

```
* {...}  
.u-float + * {...}  
*.is-hidden {...}
```

Atributo

O seletor de **Atributo** -- de sintaxe com `[]` (colchetes) --, conforme o próprio nome revela, corresponde a elemento(s) com base em valor(es) de atributo(s) que nele(s) consta(m).

O match é possível ao simplesmente indicar a existência de um atributo e, até mesmo, mirando em valores deste -- através do uso de determinados caracteres, como `~, |, ^, $` e outros mais.

Exs:

```
a[lang] {...}  
span[lang="pt"] {...}  
a[href$=".io"] {...}
```

Pseudo-classe/Pseudo-elemento

Por fim, seletor **Pseudo-classe (:) e Pseudo-elemento (::)**, imprescindíveis em muitas situações de estilização.

Exs:

```
a:hover {...}  
.c-input--textarea:focus {...}  
p::first-line {...}
```

Uma observação é que não é porque se trata do tipo de seletor menos eficiente que deve ter seu uso abolido. Pelo contrário: ele é muito útil para conseguir determinadas configurações visuais e efeitos com CSS.

Como navegadores lêem seletores CSS

E, agora, depois de aprender sobre a eficiência inerente, vem uma informação que pode abalar o seu mundo! :D

Preparado?

Navegadores lêem seletores CSS da direita para a esquerda!

É isso, mesmo, caro leitor. Pode parecer bem estranho e um pouco difícil de entender de primeira, mas seletores CSS são lidos da direita para a esquerda pelos browsers.

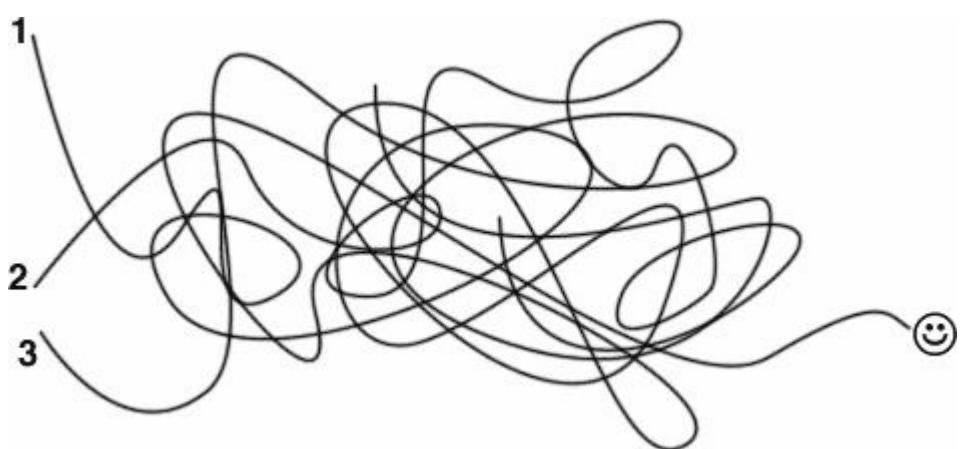
Para entender porquê é assim, rememore aqueles jogos de “encontre o caminho” da sua infância. Quer dizer, dependendo da sua idade, talvez você não saiba bem o que é isso...

Se for o caso, force sua imaginação a conceber um mundo em que brincadeiras de crianças existiam em revistas (de papel) e eram feitas usando lápis de cor, giz-de-cera e canetinha.

Acredite ou não, nosso mundo já foi assim. :)

E uma das brincadeiras dessas revistas recreativas infantis era a de descobrir o caminho correto dentro de um emaranhado de possibilidades.

Algo desse tipo:



Sem querer ser estraga-prazeres, mas aí vai um mega spoiler: é mais fácil se você fizer “de trás para frente”, ou seja, começar a partir do objetivo final e traçar a linha até encontrar o ponto de partida correto.

E adivinha só: os navegadores também gostam de brincar desse jeito.

É mais eficiente para um browser começar sua procura por combinações a partir do elemento mais à direita -- o que ele sabe que irá receber o estilo, também conhecido como “seletor-chave” -- e trabalhar seu “caminho de volta” através da árvore do DOM.

[Existe toda uma explicação técnica para isso](#), mas, para os propósitos desse tópico, só tenha em mente que os navegadores lêem seletores CSS da direita para a esquerda.

O seletor-chave

O **seletor-chave**, como citado, é a parte mais à direita de um seletor CSS. É ele que o browser procura em primeiro lugar.

Lembre-se da ordem dos seletores mais eficientes, vista há pouco. Seja qual for o seletor-chave da vez, ao escrever CSS, é **este seletor que contém o segredo do alto desempenho!**

Veja este exemplo de seletor:

```
#l-main a {...}
```

Quando nós, devs, nos deparamos com um seletor desses, pensamos em algo como “dentro do elemento `#l-main`, pegar todos os `a`”, ou seja, *fazemos a leitura da esquerda para a direita*.

Mas você já sabe que o navegador faz diferente, lê *regras da direita para a esquerda*, significando que ele irá procurar **todas as instâncias** de `a` (da página toda) e, depois, começar a subir no DOM até encontrar o elemento `#l-main` para, aí sim, ter certeza de quais links devem receber a estilização.

Agora você entendeu o tamanho do problema, né?

Existem páginas com centenas de links; até com milhares! E se um seletor parecido com o deste exemplo for usado, mesmo que com a melhor das intenções, o dev que fez isso acabou de criar um monstrinho devorador de performance...

Imagine, então, seletores parecidos com:

```
body .wrapper #content ul li a {...}
```

Esses são os chamados **seletores superqualificados**.

Neste caso, o navegador sabe que deve aplicar o estilo ao seletor-chave `a`. Para cumprir sua missão, terá que olhar para todos os elementos de link da página para verificar se, respectivamente, cada um deles está dentro de um `li`; depois, dentro de um `ul`; depois, dentro de `#content`; depois, dentro de `.wrapper`; depois, dentro de `body`.

Você já deve ter encontrado monstros como esse por aí. Em bibliotecas e frameworks conhecidos, inclusive!

Aprimoramento de seletores é um assunto extenso, à parte do que estamos querendo tratar neste Livreto, mas, só para não “deixar no ar”, uma maneira muito mais eficiente de estilizar os links do exemplo acima seria:

Atribuir uma classe a cada link daquele lista não-ordenada específica. Suponhamos, “`c-list__link`”.

Que occasionaria no seletor:

```
.c-list__link {...}
```

O importante a se considerar é: **a estilização afetaria exatamente os mesmos links da regra anterior**, mas sem superqualificação e sem ter que percorrer todo o DOM da página para fazer o serviço.

Findados os conhecimentos preliminares necessários, chega o momento propício de entrar nos tópicos principais do Livro: **os 3 conceitos fundamentais de CSS para entender como as folhas de estilo realmente funcionam.**

Mas, confesse: só com essa revisão, agora você já entende muito mais. ;)

CONCEITO FUNDAMENTAL #1

HERANÇA

Você já recebeu ou irá receber uma herança?

Herança é a parcela (ou o todo) do patrimônio de alguém, transferida a certas pessoas elencadas na lei como titulares desse direito -- os sucessores.

Para uns, é algo desejado há tempos, vista como uma maneira de elevar o padrão (material) de vida; para outros, nem tanto, já que sua transmissão se dá em circunstâncias tristes, após a despedida de um ente (geralmente) querido.

Fato é que uma herança transfere um patrimônio de uma pessoa para outra(s): o que pertencia àquela, passa a pertencer a esta(s).

Pode parecer curioso, mas algo parecido acontece com as CSS. Inclusive, usa-se o mesmo nome para o fenômeno: **Herança**.

Mas, diferentemente do quê acontece no chamado mundo real, em CSS, o “patrimônio” é composto por características visuais e comportamentos e a herança é passada ainda “em vida”.

Vejamos, então, como se dá a herança em CSS, quais “sucessores” são beneficiados e que tipo de “patrimônio” eles podem receber.

Como funciona a herança nas folhas de estilo

Explicando bem diretamente, a **herança**, em CSS, quer dizer que *alguns* valores de propriedade definidos em elementos são herdados por seus descendentes.

Por exemplo, se você definir a cor de um elemento, todos os elementos dentro dele (descendentes) também serão estilizados com essa cor, a menos que você, explicitamente, especifique uma cor diferente.

Exemplificando em código, uma estrutura HTML como:

```
<p>Lorem ipsum dolor, <em>sit amet  
consectetur</em> adipisicing elit. Accusantium,  
<strong>atque</strong>!</p>
```

Estilizada com:

```
p {  
    color: blue;  
}
```

Fará com que este parágrafo tenha a cor azul.

Perceba que o parágrafo tem filhos (descendentes diretos), `` e ``, e que eles, automaticamente, **herdaram a cor azul de seu pai**.

A maioria dos desenvolvedores front-end não parou para pensar nisso por 2 segundos e meio, não cogitando que a herança CSS está em ação o tempo todo -- e que foi projetada para permitir a escrita de menos regras CSS.

Lembre-se que, em CSS, a herança é passada “em vida”. Isso significa que o elemento mais ancestral de todo o documento (`<body>`) transmite determinadas características a seus descendentes (absolutamente, todas as tags presentes).

Por esse motivo, é muito comum, quando é preciso que um site/app tenha uma fonte diferenciada, a definição de fonte ser feita no elemento `<body>`, já que todo o documento herdará a exibição da fonte personalizada.

O que também remete ao que foi comentado acima, que, caso não haja uma sobrescrita explícita de valores das propriedades que têm a capacidade de serem transmitidas, eles entram em ação.

A sobrescrita explícita se deu com `color: blue`; caso não tivesse sido feita, a cor default teria sido herdada do ancestral `<body>` -- e também se refletiria em `` e ``.

`<body>` é um sujeito generoso: não deixa de fora do “testamento” nem seu tataaaaaaaaaaaaaaaaaaaaaaaaaaaaaaneto. :)

Propriedades herdadas

Talvez você já tenha visto o valor de propriedade `inherit`. Não é bem ao pé-da-letra, mas considere que essa palavra significa **herdado**.

O valor-chave `inherit` faz com que o elemento para o qual é especificado obtenha o valor calculado (*computed value*) da propriedade de seu ancestral.

Ou seja, quando uma propriedade tem esse valor, significa que, o que quer que seja que ela controla no elemento, é para ser aplicado o que foi herdado de seu elemento ancestral.

O que, não necessariamente, significa que será o estilo do elemento-pai, pois esse pode ter herdado do avô, bisavô e assim sucessivamente.

Propriedades não-herdadas

O valor inicial de uma propriedade CSS é seu valor inicial, que pode ter 2 comportamentos diferentes entre propriedades herdadas e propriedades não-herdadas.

Para propriedades não-herdadas, o valor inicial é usado em cada elemento. Quando nenhum valor para uma propriedade não-herdada for especificado, o elemento obtém o valor inicial diretamente.

Exemplo:

```
<p style="border: 1px solid black">  
    Parágrafo com <em>texto enfatizado</em>.  
</p>
```

O termo "texto enfatizado" não terá uma borda própria.

Como `border` é uma propriedade não-herdada e não há propriedade de borda especificada para `em`, então ele obtém o valor inicial de borda `none`.

Segundo informado, quando nenhum valor para uma propriedade não-herdada é especificado, o elemento obtém o valor inicial desta propriedade.

Controle de herança: `inherit`, `initial` e `unset`

É possível interferir no modo padrão de como a herança funciona em CSS.

Para isso, é possível usar algumas palavras-chave (ou valores-chave), como `inherit`, `initial` e `unset`.

Para começar, é possível extinguir a dúvida mais comum, que é a diferença entre `inherit` e `initial`, somente pela tradução dos termos.

`inherit` significa “herdado”; `initial`, “inicial”.

Ou seja, `inherit` faz com que a propriedade herde o valor computado de seus ancestrais, enquanto `initial` usa o valor inicial (ou padrão) de uma propriedade -- que é definido pelo próprio navegador.

Mesmo sendo esta uma explicação simplificada, pode ser que não tenha ficado claro. Então, vamos a um pouco de código.

Exemplo:

```
<p>
  <span>Texto em vermelho.</span>
  <em>Texto na cor inicial (preto).</em>
  <span>Texto em vermelho de novo.</span>
</p>
```

```
p {
  color: red;
}

em {
  color: initial;
}
```

Como indicam os próprios conteúdos do parágrafo do exemplo, o `em` terá sua cor inicial mostrada (a que é definida pelo navegador) devido ao uso explícito do valor `initial`.

O valor `unset` é ainda menos comum de ser visto, mas também pode ajudar em determinadas circunstâncias.

Ele serve para restaurar a propriedade com seu “valor natural”: se a propriedade é herdada naturalmente, ele age como `inherit`; caso contrário, como `initial`.

Um exemplo comparativo deixará tudo mais claro:

```
<ul>
  <li>
    <a href="#">Cor de link</a> default;
  </li>

  <li class="u-inherit">
    <a href="#">Cor de link</a> com inherit;
  </li>

  <li class="u-initial">
    <a href="#">Cor de link</a> com initial;
  </li>

  <li class="u-unset">
    <a href="#">Cor de link</a> com unset;
  </li>

  <li class="u-revert">
    <a href="#">Cor de link</a> com revert.
  </li>
</ul>
```

```
body { color: green; }
.u-inherit a { color: inherit; }
.u-initial a { color: initial; }
.u-unset a { color: unset; }
.u-revert a { color: revert; }
```

A renderização final será:

- Cor de link default;
- Cor de link com inherit;
- Cor de link com initial;
- Cor de link com unset;
- Cor de link com revert.

Mas, espera aí! O que é esse `revert`?!

Ah, sim. É o novo valor possível para controle de herança. :)

O `revert` redefine (ou **reverte**) o valor de cascata de uma propriedade para o padrão especificado pelo navegador, tal como se não houvesse nenhum CSS tivesse sido incluído.

Em muitos casos, `revert` funciona exatamente como `unset`. A diferença é se os valores para as propriedades foram definidos pelo navegador ou por folhas de estilo de usuário -- mais informações sobre isso na parte sobre Cascata.

`revert` não afetará regras aplicadas a filhos de um elemento que foram resetadas, mas removerá os efeitos de uma regra-pai em um filho.

Vamos a mais um exemplo comparativo, para ficar mais clara a diferença entre `unset` e `revert`:

```
<p style="margin: revert;">  
    Texto do primeiro parágrafo.  
</p>
```

```
<p style="margin: unset;">  
    Texto do segundo parágrafo.  
</p>
```

```
p {  
    margin: 50px;  
}
```

Como resultado, `revert` cancelará a margem de `50px` e colocará de volta a margem default aplicada pelo navegador; `unset` simplesmente definirá a margem para o valor inicial (que é `0`).

Veja a [lista completa de propriedades CSS](#) para saber quais são herdadas e quais não são.

Dica: regra geral, estilos que não são herdados geralmente estão relacionados à aparência dos elementos.

A propriedade `all`

Apesar de não ser nenhum segredo de CSS, pouco se vê o uso da propriedade `all`.

`all` é uma forma abreviada de escrita (*shorthand*) para aplicar algum desses valores de controle de herança a quase todas as propriedades de uma vez -- excetuadas `unicode-bidi`, `direction` e CSS Custom Properties (Variáveis CSS).

É uma maneira conveniente de desfazer alterações feitas nos estilos para que seja possível voltar a um ponto de partida conhecido/seguro antes de iniciar novas alterações.

Imagine que há várias propriedades definidas, cujos valores você queira sobrescrever todos de uma vez.

E, por “sobrescrever”, entenda usar um daqueles 4 valores possíveis para fazer controle de herança.

É justamente para isso que `all` existe; para economizar escrita de CSS, garantindo que, ao usar 1 só propriedade, esse controle possa ser feito da maneira mais ampla possível.

Por exemplo:

```
<blockquote>
  <p>Citação estilizada.</p>
</blockquote>

<blockquote class="u-total-fix">
  <p>Citação não estilizada.</p>
</blockquote>
```

```
blockquote {
  background-color: red;
  border: 5px solid green;
  color: #fff;
}

.u-total-fix {
  all: unset;
}
```

Experimente definir o valor de `all` para algum dos outros valores de controle de herança e analise as diferenças.

CONCEITO FUNDAMENTAL #2

ESPECIFICIDADE

Antes de mais nada, você sabe o que é especificidade?

Especificidade é a qualidade daquilo que é específico.

Como não poderia deixar de ser, em CSS o conceito quer dizer mais que sua definição de dicionário, trazendo consequências práticas para a escrita de estilos do dia-a-dia.

Tal como todos os outros conceitos vistos até o momento, importa saber mais sobre a **Especificidade**, um dos conceitos fundamentais de CSS para entender como as folhas de estilo realmente funcionam.

Como o browser interpreta

Fazendo uma analogia, você pode comparar a especificidade a como a mente humana interpreta instruções.

Por exemplo, veja essas 2 caixas:



(a)



(b)

Se eu lhe dissesse “Por favor, pegue a caixa vermelha para mim”, qual delas você me entregaria?

Você poderia perguntar: “Qual das caixas, a ou b?” ou até mesmo pegar as 2 caixas -- afinal, ambas são vermelhas, certo?

Este é um paralelo que ilustra muito bem a situação em que os navegadores se encontram ao lidar com a Especificidade CSS.

Quando você passa a instrução “estilize o parágrafo com uma cor de fundo vermelha”...

```
p {  
    background-color: red;  
}
```

Como pode haver muitos elementos de parágrafo na página, o navegador tem que “adivinar” a qual parágrafo a instrução se refere.

O browser não pode fazer perguntas complementares aos desenvolvedores para decidir sobre a renderização, então, ele segue em frente e tenta estilizar cada parágrafo da página com fundo vermelho.

Entretanto, se, ao invés, a instrução tivesse sido “Estilize o parágrafo de classe ‘reddy’ com um fundo vermelho”...

```
p.reddy {  
    background-color: red;  
}
```

Esse é um pedido mais... **Específico!**

Agora, o navegador estilizará o(s) elemento(s) parágrafo específico(s) que você solicitou. Simples assim.

Tecnicamente, o navegador dá uma olhada em cada seletor que tem como alvo um elemento específico e atribui pontos (ou pesos; ou níveis) a cada um e, aquele com uma pontuação de especificidade mais alta, “vence”.

Cada tipo de seletor recebe pontos diferentes, por exemplo, IDs recebem uma pontuação/peso/nível, classes e atributos outra, e assim por diante.

Pode ter acontecido com você, pode não ter acontecido, mas é algo bastante comum -- principalmente quando se está começando com CSS -- acontecer de se criar uma regra e, por algum motivo, ela não “entrar em ação”.

Isso acontece, justamente, por causa da Especificidade. Algum outro seletor “ganhou” daquele que você estava mexendo e os valores das propriedades, que você imaginou que entrariam em ação, não serviram de nada.

Por exemplo, a partir desse código, que cor terá o parágrafo?

```
p {  
    background-color: white;  
    color: black;  
}  
  
div.c-tip--warning p {  
    color: red;  
}  
  
body#l-home .container p {  
    color: white;  
}
```

A resposta é: **branco**.

Para entender o porquê, você precisa aprender a calcular a especificidade dos seletores.

Como calcular Especificidade

Então, você já sabe: em “briga” de seletores, ganha aquele que tiver o maior valor/peso/nível.

Depois de entender tudo isso que acabou de ser apresentado, você precisa saber como o navegador faz esse cálculo de especificidade.

Não que, a partir de agora, você precise fazer cálculos sempre que escrever regras CSS. A intenção é fazer você **ficar ciente** do quê acontece “por debaixo dos panos” para, em caso de necessidade, você saber como resolver problemas.

Fique sabendo do seguinte: **toda regra CSS tem uma especificidade implícita**. Portanto, mesmo que não soubesse sobre a Especificidade até agora, você já estava trabalhando com ela desde o início!

A ideia é que, a partir de agora, você esteja consciente, sendo capaz de escrever CSS fácil de ler, limpo e manutenível. :)

Para entender a Especificidade CSS

Existem 4 categorias que definem o nível de especificidade de um seletor. Preste atenção na ordem, pois é importante.

Para facilitar as explicações, vamos atribuir letras a cada possibilidade:

- a. Estilo inline;
- b. ID;
- c. Classe, pseudo-classe e atributo;
- d. Elemento e pseudo-elemento.

Por exemplo, seria possível esquematizar a especificidade de um elemento `<h1>` da seguinte maneira:

```
h1 = (a=0), (b=0), (c=0), (d=1)
```

Quer dizer, no seletor `h1` existe **0** estilo inline; **0** ID; **0** classe, pseudo-classe e atributo e; **1** elemento ou pseudo-elemento.

Para ficar menos verboso, isso poderia ser mostrado como:

```
h1 = 0,0,0,1
```

Para facilitar ainda mais, também é possível “converter” essa representação em um número inteiro removendo as vírgulas:

```
h1 = 0001
```

Removendo os zeros à esquerda, o valor final seria **1**.

Para ficar mais bonitinho:

```
h1
```

0

Estilo inline

0

ID

0

Classe, pseudo-classe e atributo

1

Elemento e pseudo-elemento

Se esse título estivesse dentro de outro elemento, a especificidade se alteraria, já que outros tipos de elementos comporiam o seletor.

#l-content h1

- 0 Estilo inline
- 1 ID
- 0 Classe, pseudo-classe e atributo
- 1 Elemento e pseudo-elemento

Neste caso, seria o equivalente a 0101 -- 0 estilo inline; 1 ID; 0 classe, pseudo-classe e atributo e; 1 elemento ou pseudo-elemento.

Removendo o zero à esquerda, **101**.

Então, qual desses 2 seletores ganha a briga?

1. `h1`
2. `#l-content h1`

Acertou se você disse que é a opção 2: **101 é maior que 1**.

Isso significa que, se ambos os seletores existirem em um conjunto de estilos de uma página, `#l-content h1` teria precedência em relação a `h1`, dado que **101 > 1**.

Atente-se ao fato de que a matemática envolvida é somente para demonstrar a lógica que acontece “nos bastidores”. Seria

completamente incoerente se, por padrão, CSS priorizasse a estilização de um elemento puro em detrimento à estilização deste mesmo elemento dentro de um contêiner, certo?

Para fixar, responda: qual é o peso desse seletor?

- `#l-sidebar ul li a.c-menu__link:hover`

Acertou se você disse **123**. Ele tem 0 estilo inline; 1 ID; 2 classe, pseudo-classe e atributo e; 3 elemento e pseudo-elemento.

`#l-sidebar ul li a.c-menu__link:hover`

- 0 Estilo inline
- 1 ID
- 2 Classe, pseudo-classe e atributo
- 3 Elemento e pseudo-elemento

Em um eventual conflito, tudo o que você tem que fazer é garantir que a regra CSS que você quer ver em ação tenha o seletor de maior peso.

E em caso de empate?

Caso ocorra um “empate”, ou seja, 2 ou mais seletores em um mesmo conjunto de folhas de estilo atuando, com o mesmo peso, a Cascata CSS entra em ação -- mais sobre Cascata a seguir.

Agora só falta conhecer o 3º conceito fundamental de CSS, considerado por muitos como o mais importante: **Cascata**.

CONCEITO FUNDAMENTAL #3

CASCATA

CSS é a sigla para **Cascading Style Sheets**, que se convencionou traduzir para português como **Folhas de Estilo em Cascata**.

Você já pensou nesse termo? Já refletiu sobre o que ele significa? A maioria não. Então, vamos a este breve exercício de análise de termos.

“**Folhas**”, aqui, quer indicar os arquivos. Uma metáfora para indicar que cada arquivo CSS é como uma folha, com seus próprios conteúdos -- forçando a barra um pouquinho, você também pode pensar como uma espécie de *esqueumorfismo metafórico*.

Esqueumorfismo é um princípio de design em que objetos derivados retêm ornamentos e estruturas necessários apenas nos objetos originais.

“**Estilo**”, neste contexto, refere-se à aparência das coisas; a como será a cor, tamanho, formato, proporção etc.

“**Folhas de Estilo**”, portanto, é um termo que significa arquivos que contêm instruções (ou regras) para a definição da aparência de elementos.

Mas ficou faltando um pedaço; uma letrinha... O que será que significa “**Cascata**”?

Dessa vez, não será possível recorrer a definições de dicionário. No mundo do desenvolvimento front-end, a Cascata tem seu próprio significado. E que significado!

Na verdade, não se trata somente de uma explicação de termos; a Cascata é muito mais um **modo de funcionamento** de estilos na Web; tem muito mais a ver com como as coisas acontecem em CSS.

E, sabendo dos mecanismos internos de funcionamento de CSS, automaticamente se torna possível começar a entender mais de suas capacidades e potencialidades, além de começar a ser possível antecipar/evitar problemas.

Indo mais além: **compreender como a Cascata funciona é a chave para entender CSS!**

E então, preparado para desvendar a última parte desse quebra-cabeça léxico? :)

Afinal de contas, o que é a Cascata?

Tecnicamente falando, a **Cascata** pega uma lista não-ordenada de valores declarados para uma dada propriedade em um determinado elemento, classifica-os pela precedência de sua declaração e produz um único valor em cascata.

Em outras palavras, Cascata é o algoritmo pelo qual o navegador decide quais estilos CSS aplicar aos elementos.

A Cascata engloba muitos dos fundamentos e conceitos que foram vistos até então, e vai mais além.

Quem não entende a Cascata, não entende CSS.

E, para entendê-la, é preciso conhecer, além do que já foi mostrado anteriormente no Livro, alguns outros conceitos imprescindíveis.

O nome CSS foi tão bem escolhido, que cada um de seus termos faz sentido separadamente. Então, para entender os critérios da “Cascata”, imagine, realmente, uma... Cascata.

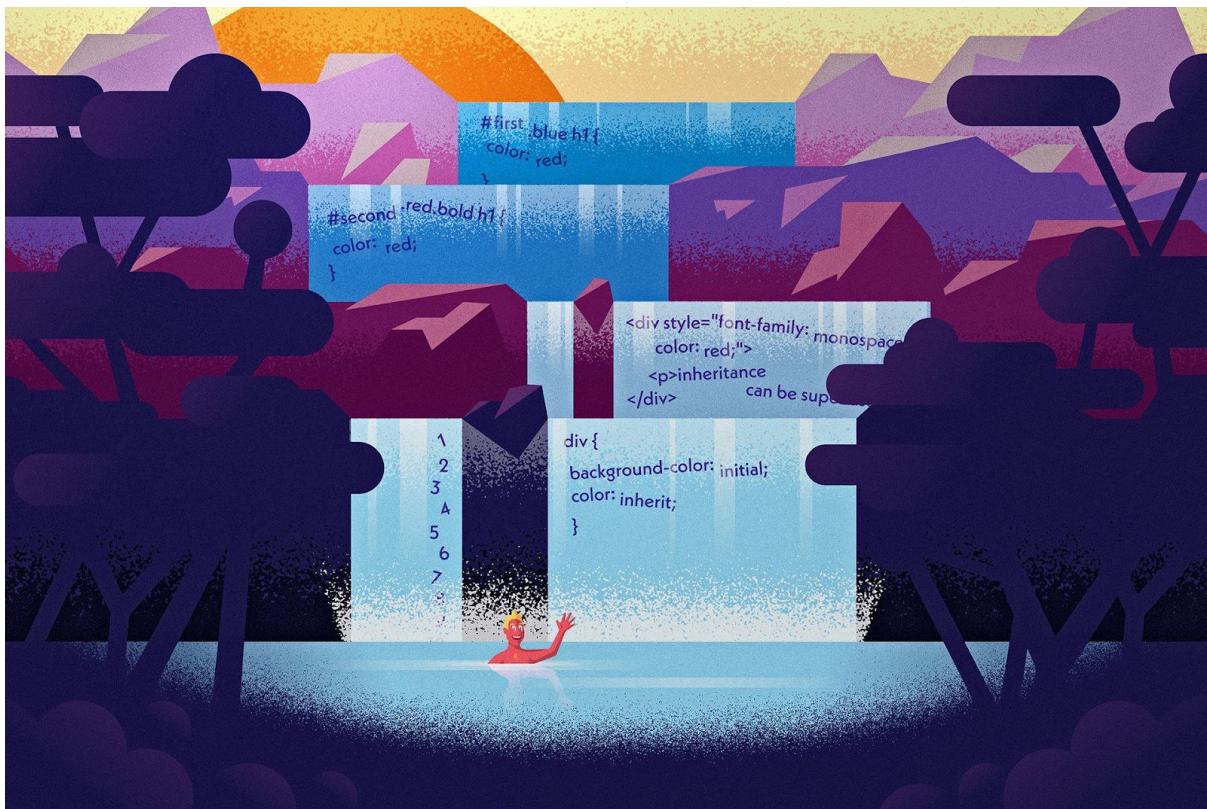


Imagen retirada do LogRocket Blog.

Cada vez que escrevemos uma regra CSS, ela entra na Cascata, que determinará se terminará ou não surtindo efeito no estilo final.

Conforme mostra a ilustração, existem diversos patamares ou **níveis** na Cascata. Quanto mais abaixo na cascata uma declaração vai, menos provável que entre em ação no estilo final aplicado.

Logo, é importantíssimo conhecer quais são esses níveis da Cascata, ou seja, os critérios efetivos pelos quais estilos serão aplicados na página ou não.

Origem

A **Origem** representa a nascente da Cascata (ou seu 1º nível); é onde se inicia o curso d'água que vai fluir regras de estilo para a página Web.

Trocando em miúdos, dentre os diversos critérios levados em consideração, a origem da Cascata é um deles.

Existem 3 origens principais:

1. Navegador (*User-Agent Origin*)
2. Usuário (*User Origin*)
3. Página Web (*Author Origin*)

User-Agent Origin

User-Agent Origin, ou **UA Origin**, refere-se ao próprio navegador em que ocorre o acesso à página Web.

Os UAs devem aplicar uma folha de estilo padrão (ou se comportar como se o fizessem). A folha de estilo padrão de um UA deve apresentar os elementos da linguagem do documento de forma que satisfaça as expectativas gerais de apresentação -- por exemplo, o elemento HTML `` é apresentado usando uma fonte em itálico.

É por isso que, mesmo que em uma página sem 1 linha sequer de CSS, um título se parece com um título, uma lista com uma lista, um parágrafo com um parágrafo e assim por diante.

Aliás, cada navegador se arroga o direito (e dever) de especificar quais serão seus estilos, o que explica porque, [não tomados os devidos cuidados](#), pode haver pequenas discrepâncias visuais de navegador para navegador, como uma margem ligeiramente diferente, um entrelinhas maior etc.

User Origin

A pessoa que usa o navegador -- comumente chamada de **usuário** -- também pode escolher usar estilos próprios, optando por usar uma folha de estilos personalizada.

Por exemplo, é possível que uma pessoa defina uma regra de usuário que faz com que todas as fontes fiquem com o dobro do tamanho, em todos os sites que acessar.

Author Origin

Também poderia ser chamada de **Dev Origin**; é essa origem que nós, desenvolvedores, nos valemos no dia-a-dia da codificação CSS, tornando-as o tipo mais comum.

São folhas de estilo que definem estilos como parte do design de uma determinada página web. O autor/desenvolvedor da página define os estilos do documento usando uma ou mais folhas de estilo, que definem a aparência do site.

3 maneiras de se usar Author CSS

Importante destacar, também, que é possível usar folhas de estilo de autor de 3 maneiras diferentes:

1. **Inline.** Usa-se a propriedade `style` diretamente em tags HTML para especificar os estilos;
2. **Incorporada.** As regras são colocadas dentro da tag `<style>`, no próprio documento;
3. **Externa.** Arquivos de extensão `.css` à parte, que são aplicados em uma página usando `<link>`.

A Cascata nas origens

Algo que deve ser levado em consideração é que pode haver conflitos de especificidade em seletores de origens diferentes. Não é tão comum de causar efeitos desastrosos, mas vale prestar atenção.

Dentre as 3 origens de Cascata principais, a ordem de precedência de aplicação de estilos (Especificidade) é:

1. Author
2. User
3. User-agent

Como foi visto anteriormente, é possível indicar ênfase total em determinado valor de propriedade usando `!important`.

Para casos em que haja conflitos em regras que tenham `!important`, a ordem se inverte:

1. User-agent
2. User
3. Author

Importância

A **Importância** pode ser considerado o 2º nível da Cascata -- e “importância”, aqui, realmente se refere a declarações com `!important`.

Quando do uso de `!important`, o valor da respectiva propriedade ganha prioridade sobre quaisquer efeitos da Especificidade.

`!important` no valor de uma propriedade CSS é uma vitória automática! Ele substitui até mesmo estilos inline da marcação.

Considerando a mesma origem, a única maneira de um `!important` perder é com outra regra com `!important` sendo declarada posteriormente no CSS, com valor de especificidade igual ou maior.

Mas também é possível haver conflitos em origens diferentes. Lembre-se: a Cascata é influenciada por diversos fatores, e aspectos como Especificidade, Importância, Origem e outros, são todos levados em consideração no algoritmo.

A precedência das várias origens é, em ordem decrescente:

1. Declarações de transição (`transition`)
2. Declarações de User-Agent com `!important`
3. Declarações de Usuário com `!important`
4. Declarações de Autor com `!important`
5. Declarações animação (`animation`)
6. Declarações de Autor
7. Declarações de Usuário
8. Declarações de User-Agent

Nesta lista, declarações de origem anteriores prevalecem sobre declarações de origens posteriores.

Devido a seu imenso potencial de “quebrar” a Cascata, evite a todo custo usar `!important` em suas regras CSS.

Com os conhecimentos adquiridos neste Livro, mesmo em casos de conflito de regras, isso certamente vai ficar mais fácil de fazer a partir de agora.

Especificidade

No 3º nível da Cascata, é levada em consideração a Especificidade dos seletores das regras CSS.

Já abordamos a Especificidade com mais detalhes na parte anterior, então, para uma revisão, basta voltar algumas páginas.

O importante a saber é que este também é um quesito levado em consideração nos cálculos da Cascata para decidir quais estilos CSS aplicar aos elementos.

Posição

O 4º e último nível da Cascata é referente à **Posição**, em uma análise da ordem em que as regras foram definidas.

Basicamente, regras definidas posteriormente em folhas de estilo externas ou incorporadas vencerão, desde que todo o resto na Cascata seja o mesmo.

Em outras palavras, **a última declaração na ordem do documento vence**, levando em consideração que:

- Declarações de folhas de estilo importadas são ordenadas como se fossem substituídas no lugar de `@import;`
- Declarações de folhas de estilo independentemente vinculadas pelo documento de origem são tratadas como se fossem concatenadas em ordem de vinculação;
- Declarações de atributos de estilo são ordenadas de acordo com a ordem do documento do elemento em que o atributo de estilo aparece e todas são colocadas após qualquer folha de estilo.

Visto que CSS considera a ordem da origem na Cascata, a ordem em que as folhas de estilo são carregadas realmente importa.

Se você tiver 2 folhas de estilo externas vinculadas ao documento, a segunda substituirá as regras da primeira -- esse também é o motivo pelo qual, se você estiver usando um reset/normalize ou algum framework CSS, convém carregá-lo antes de seus próprios estilos.

Um exemplo ilustrativo seria:

```
div { border: 1px solid red; }
div { border: 1px solid blue; }
```

Neste caso, a borda seria azul, dado que a segunda regra, pela ordem, substitui a primeira.

A não ser pelo uso de...

```
div { border: 1px solid red !important; }
div { border: 1px solid blue; }
```

`!important` causa vitória automática para a primeira regra.

Mas, como você já sabe neste ponto, muitas coisas são levadas em conta nos diferentes níveis da Cascata.

Como entender a Cascata pode ajudar a escrever CSS melhor?

Como a Cascata é uma das partes mais incompreendidas de CSS -- e, por isto mesmo, fonte de muitos bugs --, saber como ela funciona dá uma grande vantagem para manter suas folhas de estilo mais fáceis de ler, limpas e manutáveis.

Saber como aproveitar a Especificidade de seletores CSS a seu favor é uma habilidade muito desejável -- infelizmente, é bastante comum encontrar CSS que vai direto para o `!important` quando uma breve análise e um seletor de maior especificidade teria feito o serviço.

Como alertado anteriormente, quem não entende a Cascata, não entende CSS.

Agora você entende. ;)



CSS

ALÉM DO SENSO COMUM

**Alcance níveis mais avançados e maduros
na carreira de front-end através da
estruturação do CSS de maneira profissional**

Tão importante quanto saber sobre os conceitos fundamentais, é conhecer metodologias, convenções, arquiteturas e boas práticas de CSS para organizar e manter um projeto front-end de qualidade.

Todo o conteúdo desse Livro foi somente um pedacinho do que oferecemos em nosso curso **CSS Além do Senso Comum**.

Cadastre seu e-mail para ser avisado quando abriremos vaga para uma próxima turma:

[**CADASTRAR E-MAIL**](#)

Se ficar com qualquer dúvida ou quiser conversar sobre qualquer assunto abordado, basta enviar uma mensagem em qualquer uma de nossas redes sociais:

- [Blog dpw](#)
- [Facebook](#)
- [YouTube](#)
- [Instagram](#)
- [Twitter](#)

Esperamos que este conteúdo tenha sido útil para você.

Seja sempre sincero e honesto com você mesmo, com seus pares e com seus clientes. Em 100% dos casos, manter sua integridade é mais importante que continuar em um job ou manter seu emprego.

Confie sempre em si mesmo.

Equipe dpw