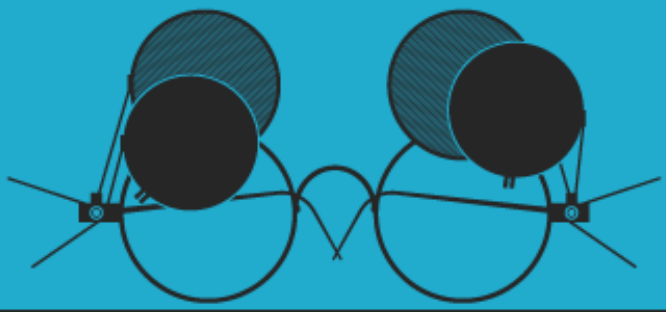


CSS3

Succinctly

by Peter Shaw

Syncfusion | Technology Resource Portal

CSS3 Succinctly

By
Peter Shaw

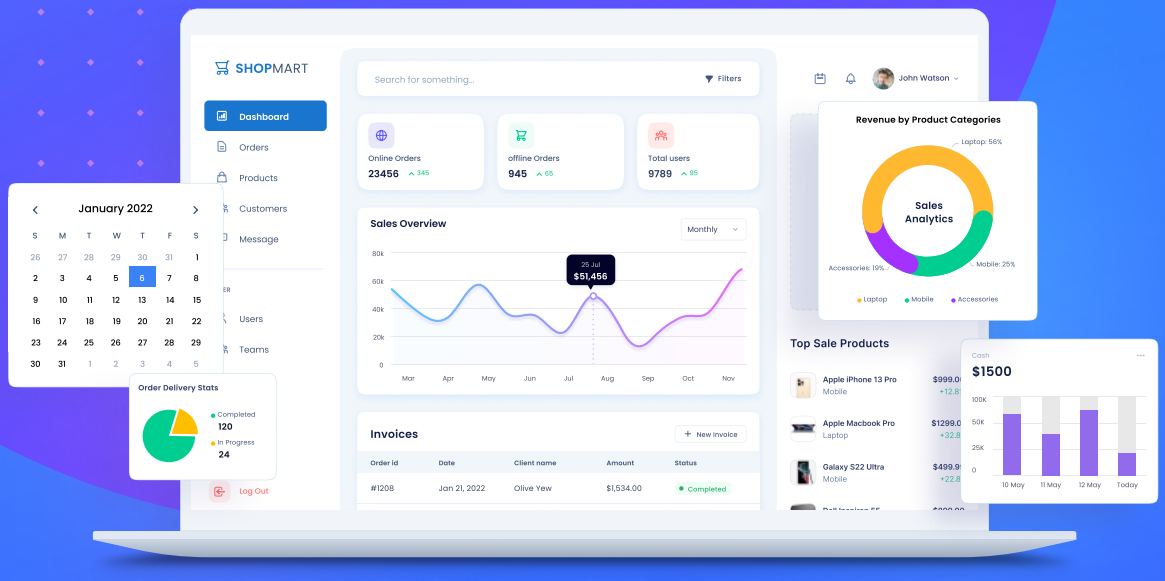
Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Gavin Lanata
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

THE WORLD'S BEST
UI COMPONENT SUITE
FOR BUILDING
POWERFUL APPS

4.6 out of 5 stars



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Apple IBM SIEMENS intel VISA

Syncfusion

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

As an early adopter of IT back in the late 1970s to early 1980s, I started out with a humble little 1k Sinclair ZX81 home computer.

Within a very short space of time this small 1k machine became a 16k Tandy TRS-80, followed by an Acorn Electron, and eventually, after going through many other machines, a 4mb arm powered Acorn A5000.

Eventually, after leaving school and getting involved with DOS-based PCs, I went on to train in many different disciplines in the computer networking and communications industries.

After returning to university in the mid-1990s and gaining a BSc in Computing for Industry, I started my own consulting business in the northeast of England called Digital Solutions Computer Software Ltd. I advise clients at both hardware and software levels in many different IT disciplines, covering a wide range of domain-specific knowledge from mobile communications and networks right through to geographic information systems, banking and finance, and web design.

With over 30 years of experience in the IT industry within many differing and varied platforms and operating systems, I have a lot of knowledge to share.

You can often find me hanging around in the LIDNUG .NET users group on LinkedIn that I help run, and you can easily find me in places such as Stack Overflow and Code Project, or on Twitter as @shawty_ds. I'm also a Pluralsight author and produce various videos that are viewable online.

I hope you enjoy the book, and learn something new from it. In today's fast-moving world of IT, we already have little time to spare trying to find the information we need—these books hopefully help reduce that burden on time.

Please remember to thank Syncfusion (@Syncfusion on Twitter) for making this book (and others in the range) possible, allowing people like me to share our knowledge with the .NET community at large. If you have ideas for a book, please reach out to Syncfusion and let them know what you'd like to see. The *Succinctly* series is a brilliant idea for busy programmers.

Chapter 1 What is CSS3?

If you've been a web developer for any length of time, then CSS won't be strange to you. You might, however, be wondering what the fuss over CSS3 is all about.

Just like the new HTML 5 specifications, CSS3 (or CSS Version 3, to be more precise) is the latest set of specifications designed to mold, shape, and define just what capabilities the newest version of CSS has.

To get the most from CSS3, you do need to have a general understanding of the concept of CSS first. Chapter 2 will cover a very basic refresher to help with this, but if you've never used CSS in any way, then you might feel a little lost reading this book, as it does assume that you have at least a passing familiarity with the technology.

CSS3 is also not a "new CSS" as many people think; like HTML 5, it's backwardly compatible with everything that's come before it. Going forward, however, there's an amazing amount of new functionality, allowing a multitude of creative possibilities that simply did not exist in previous versions.

Code Samples

As we make our way through the book, I'll be creating various code samples for you to study. These code samples won't be available for download, but they will be small enough to type in by hand.

In most cases the code will be given only to highlight a specific topic, especially in the case of mixed CSS and HTML code. With the HTML code (unless it's a quite complex example) in general, I'll just describe what's needed, then show the CSS code as a code sample.

Because of this, I will be assuming that the reader understands HTML, how it's structured, and how to create standard simple HTML tags and elements. If you do not, then I strongly advise that you read at least some background material on the subject before you proceed any further with this book.

To try and make things somewhat easier, however, the following HTML code should serve as a very minimal example of an HTML 5 standards document. This is by no means a complete example; its purpose is to provide a boilerplate starting point for other code in the book.

All the examples I present throughout this book will make an assumption that you're using a similar piece of code, and will seek to build on top of that code.

```
<!doctype html>
<html>
<head>
    <title>Basic HTML 5 document</title>
    <link href="styles.css" rel="stylesheet" type="text/css" />
```

```
</head>
<body>
    <h1>This is a basic HTML 5 document</h1>
</body>
</html>
```

Code Listing 1: Basic HTML 5 document

Pay close attention to the **`link`** tag in the header—this specifies a file with the name of **styles.css**.

It's in a file with this (or a similar) name that we'll be placing most of our CSS code.

Good practice dictates that CSS code be separated from HTML code, and since one of the underlying principles of the *Succinctly* series is to promote good practices, then we shall endeavor to maintain this throughout the course of the book.

Of course, you don't have to call your file **styles.css**; you may call it anything you wish. However, if you do change its name, then please remember that you've used a different name when creating basic HTML documents to experiment with.

You'll be able to tell if you get the file name wrong—you'll see an error in your browser debugging tools, something similar to Figure 1.

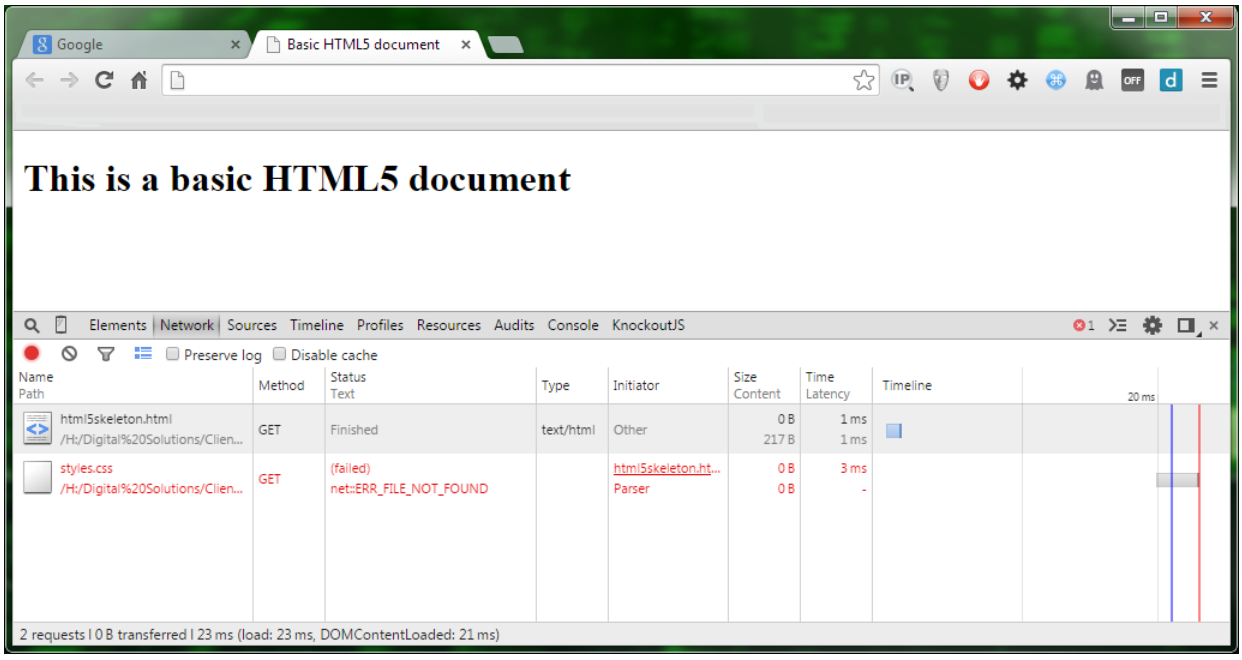


Figure 1: Chrome browser tools showing CSS file failing to load

If your HTML file is unable to find a linked CSS file, then none of the style rules you define will be used, and your result will look nothing like it should.

A quick word on browser compatibility

One final thing that we need to address before moving on is browser compatibility.

Anyone who's done any kind of HTML frontend-driven web development knows all too well the pain involved in making code appear uniform across all browsers.

This won't be more true if you've spent any time working with the CSS2 and HTML4 standards, and the browser wars of not-too-distant years.

Because CSS3 comes in, in-line with the newer HTML 5 standards, then I'll not be going out of my way to ensure the code I present will work without fail on all older browsers.

For the course of this book I'll be working with the following browser:

`Google Chrome 39.0.2171.95 m`

I'll also have the following browser versions available, but won't necessarily be using them:

- Firefox 34.0.5
- Internet Explorer 11.0.9600.17501 Update 11.0.15 (KB3008923)
- Maxthon Cloud Browser 4.4.3.4000

Since I'll mostly be sticking to features that are more or less complete, or have identical support across the four browsers mentioned, then you should have no problems using a different browser. I will, however, note where possible any changes that may be needed.

Be aware that there may be cases, when using a different browser, where a vendor prefix may be needed (but that I've missed due to my use of Chrome).

As a general rule of thumb, I develop using Chrome, then modify to accommodate other browsers. If something does not work as expected, then please do look up the CSS3 syntax for the nonworking code, online, before assuming that the code is broken.

As for minimum versions, I won't be attempting to make any code in this book work correctly you may need to use an appropriate helper library of some kind to fill in the gaps. Due to the sheer number of helpers available, there is unfortunately no room in this book to discuss them all.

Chapter 2 Basic CSS Refresher

Before we get started on the main core of the book, let's go through a basic CSS refresher.

If you've worked with CSS before, or are up to speed with the language and how it works, then you can safely skip this part of the book. If your CSS skills are a little rusty, or you're quite new to the concept of CSS in general, then you should read this chapter, as it will provide the basic groundwork for what's to come in the following chapters.

Defining CSS

So what exactly is CSS?

CSS stands for Cascading Style Sheets, and the idea behind them is a very simple one. CSS documents are created solely for the purpose of styling (that is, managing color, size, layout, position, and much more) the contents of an HTML document.

While HTML code's purpose is to define the structure of the document, CSS is there to define how that content is presented to the end user.

CSS was originally created to reduce the amount of markup that was being used in an HTML document to dictate what a particular chunk of HTML looked like.

Many modern-day web developers may never have seen this older syntax, so to give you an example of how things used to be done, look at Code Listing 2:

```
<body>
    <font family="serif" size="20pt" color="#FF0000">
        <i>
            <b>
                <h1>This is my header in bold italic, 20 point red serif font</h1>
            </b>
        </i>
    </font>
</body>
```

Code Listing 2: A typical example of pre-CSS styling

Today's browsers will still obey this and style it appropriately, as shown in the following image:

Figure 2: The output produced by the pre-CSS code sample in Code Listing 2

As you can see from the HTML code presented in Code Listing 2, vastly more markup was required to produce a simple one-line, bold, red header than most are used to today.

That was just one line—imagine having to do this every time you wanted to produce a similarly styled header for your page.

CSS rules solve two of these problems immediately. First, they typically define the rules that define how the text should look in a separate file. This immediately means that a designer or graphics-only person can easily change these rules without ever having to worry about the code in the HTML file.

Secondly, they promote re-use of a rule on multiple elements through the use of the class attribute, which means you only have to specify the settings once, then tell the markup when to use those settings.

Let's now revisit Code Listing 2, and examine the difference I've just described.

```
<h1 class="boldred">This is my header in bold italic, 20 point red serif font</h1>
```

Code Listing 3a: Our new HTML markup for the bold red style

```
.boldred
{
  font-family: serif;
  font-size: 20pt;
  color: red;
}
```

Code Listing 3b: The bold red style rule now defined in styles.css

You can immediately see the difference here, and to re-use the rule on another line, we simply have to add **`class="boldred"`** as an attribute to our tag. If we then proceed to change the style rule from, say, red to green, every element using that rule will suddenly change to a green color without ever having to touch any HTML markup.

So now we know why CSS was defined; next we'll start to look at how.

Cascading the rules

The astute reader may already be asking themselves what the “cascading” part of CSS stands for. To cascade is to flow or follow downwards, and that’s exactly what CSS style sheets do.

Not considering the subject of specificity for now, which is covered later in this book, rules that are obeyed or discovered further down the chain override anything further up the chain. If you’re not used to this, it can present some interesting bugs and challenges for you, especially when you first start out exploring what is and isn’t available.

Once you see it in action, however, the concept is quite simple.

Take the following style rule:

```
.mystyle
{
  font-size: 10pt;
  color: green;
}
```

Code Listing 4: Simple green text CSS rule

If you were to attach that to an element, you would rightly expect the text to be 10 points in size and green in color.

Now imagine this rule was defined in a style sheet that was already implemented in a CSS document, included elsewhere in your project (as part of a content management system, for example).

Furthermore, imagine you've been asked to make all text using that rule red, but without being able to change the style file previously created.

The reasons why you might not be able to change the original file are many and varied; the style sheet may be also being used by a second site where the text must remain green, or the styles may be being produced dynamically, for example. Whatever the reason, this scenario happens much more than you might anticipate.

However, if you simply create a second style file (let's call it **styles2.css**), then duplicate the link tag (as seen in the basic HTML code in the previous chapter) and change the file name appropriately, we can then proceed to add the following CSS rule to this new file.

```
.mystyle
{
  color: red;
}
```

Code Listing 5: Our duplicate rule created in 'styles2.css'

If you've done things correctly, then you should be greeted by red text in your browser, rather than green text.

In order to continue to use the green version, you simply leave out the second link that loads **styles2.css**, and include it to “adjust” your rule to be red.

If you change the font size in the original file, you'll see that, that one single change will affect both the red and green versions of the text.

This cascading of styles can be used in many different ways; before I explain that however, there's one more important refresher topic you need to understand first.

CSS Selectors

So far you've only seen the use of a “class” attribute-based CSS rule, but there are many other ways that rules can be defined.

Take the following example HTML.

```
<p id="importantInfo">This is a standard text paragraph, with no explicit style
rule defined, all it has is an 'id' attribute that essentially gives the HTML
parser a name for it</p>
```

Code Listing 6a: HTML markup with an ID attribute

```
#importantInfo
{
  color: red;
  font-size: 14pt;
}
```

Code Listing 6b: CSS Style rule that targets a given ID

In Code Listing 6a, I define a standard HTML paragraph tag. It has no associated style information or class selectors applied to it; all it has is a simple **id** attribute, which essentially allows us to give the element a name.

If you look at the CSS rule defined in Code Listing 6b, however, you'll see that instead of a period [.] being placed in front of the rule name, we now have a number sign [#] in front of it.

CSS can use a number of different symbols when defining rules. These symbols are known as CSS selectors, and how you define a selector dictates how you intend the rule to be used.

In the case of Code Listing 6, we've used the ID selector. The purpose of the ID selector is to target one specific, named element in a document.

In Code Listings 4 and 5, the class selector was intended to be used on many different elements, simply by adding a **class** attribute. Because a given ID can only be applied to one element at a time, when we specify a CSS rule that targets an ID, we are for all purposes saying that this rule applies to that element and ONLY that element—never any other.

However (as you'll see in just a moment), ID selectors are often used deliberately to control hierarchy in a CSS document, and make full use of the cascading nature of the technology.

Cascading can be used to great effect when we chain selectors together, because it allows us to drill down deep into the HTML markup, without ever having to make sure that we dictate what the markup should look like.

Let's imagine for a moment that we're creating an HTML document containing news posts.

Every news post in the document has to obey the same set of style rules, and as such, takes on the same look and feel for every post.

Each post should be shown on a different color background to the rest of the page, and should always have blue, bold headers with smaller, red paragraph text.

Based on what we've seen so far, we might be tempted to write the following code.

```
<div id="newspost">
    <h1 class="newsheader">News article 1</h1>
    <p class="newstext">The article text for news article one, this should be
standard paragraph text<p>
        <h1 class="newsheader">News article 2</h1>
    <p class="newstext">The article text for news article 2 goes right here, and
looks the same as article one</p>
</div>
```

Code Listing 7a: Possible news article HTML code

```
#newspost
{
  background-color: silver;
}
```

```
.newsheader
{
  color: blue;
  font-size: 15pt;
}

.newstext
{
  color: red;
  font-size: 10pt;
}
```

Code Listing 7b: Possible news article CSS code

While this will work very well and produce the desired output, we can actually take advantage of CSS's cascading nature here, and actually make things a little more obvious—even to the point where our HTML and CSS are almost describing their own intent.

If we change the HTML in Code Listing 7a so that we remove all the class tags from each of the **h1** and **p** elements, we should end up with the following code:

```
<div id="newspost">
    <h1>News article 1</h1>
    <p>The article text for news article one, this should be standard paragraph
text<p>
        <h1>News article 2</h1>
    <p>The article text for news article 2 goes right here, and looks the same as
article one</p>
</div>
```

Code Listing 8: The code from Listing 7a with the class attributes removed

As you can see, all we have left now is the ID defining the name of **newspost** for the **div** element that wraps all of our articles.

Now change the CSS code as follows:

```
#newspost
{
  background-color: silver;
}

#newspost h1
{
  color: blue;
  font-size: 15pt;
}

#newspost p
{
```

```
  color: red;
  font-size: 10pt;
}
```

Code Listing 9: The CSS code from Listing 7b with changed selectors

You should find that when you render it in your browser, you still get the following output:

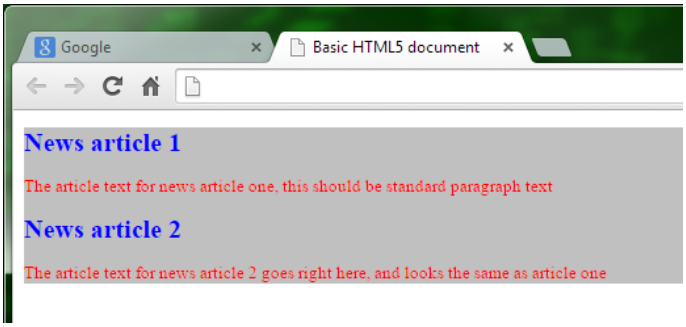


Figure 3: Output produced by Code Listings 8 and 9

One of the really great things about letting the cascading nature take care of things like this, is that you could easily add another article to your **`newspost div`** just by inserting the **`h1`** and **`p`** elements as appropriate. You don't in any way need to worry about assigning the correct class name to each **`h1`** and **`p`** element as you add it, nor do you have to keep checking that both elements are using the correct styles.

As long as the elements are within a section that has the named ID, they will be colored accordingly. Note also that the cascading behavior we've just seen doesn't apply only to ID-based selectors; you can, for example, have a class-based selector for the **`newspost div`**, that you might intend to reuse, and you can still cascade the styles for the other elements off of that, as the following code shows:

```
.newspost
{
  background-color: silver;
}

.newspost h1
{
  color: blue;
  font-size: 15pt;
}

.newspost p
{
  color: red;
  font-size: 10pt;
}
```

Code Listing 10: The CSS code from Listing 9 using class selectors

Using a cascaded, class-based selector like this allows you to reuse your **`newspost`** class in many different places in your HTML, and any **`h1`** or **`p`** elements placed inside of said element will automatically take on the look and feel of the blue headers with red text.

Pseudo Selectors

The next type of basic selector we have to consider when doing a basic refresher is the humble pseudo selector. Just like "pseudo code," the pseudo selector is not a full selector in its own right, but an extension to an existing selector.

The most well-known example of this is the hover selector used on HTML anchor tags.

A pseudo selector works by modifying an existing rule to only be active given a specific scenario, which in the case of hover, is when the user's mouse hovers over the element in question, as Code Listing 11a shows:

```
<a href="#" class="hoverClass">Hover over me I'll change color</a>
```

Code Listing 11a: HTML to demonstrate the pseudo hover selector

```
.hoverClass
{
  font-size: 20pt;
  color: red;
}

.hoverClass:hover
{
  color: blue;
}
```

Code Listing 11b: CSS code to demonstrate the pseudo hover selector

If you render a page containing this code in your browser, you should see something like the following:

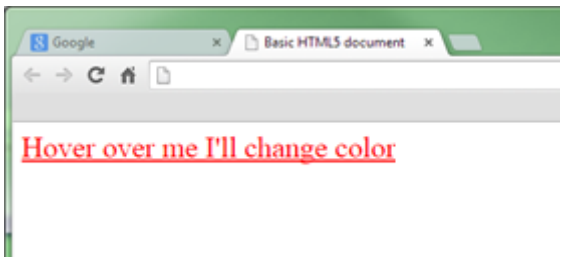


Figure 4: Pseudo hover selector in its default state

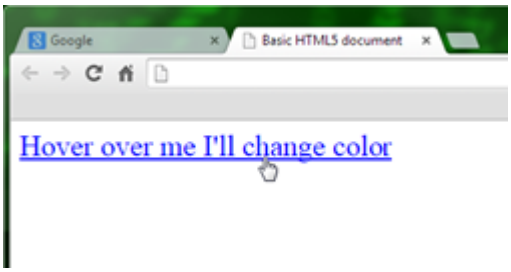


Figure 5: Pseudo hover selector in its hovered state

Prior to CSS3, there were only a few pseudo selectors that could be relied on to work across all browsers; typically these applied only to HTML anchor tags, and were hover (as you've just seen), visited, active, and link, which were used to style anchor tags for visual feedback, depending on whether or not a user had previously visited the link in question. Unfortunately, from CSS3 onwards, even though these styles still exist and work as anticipated, the number of options that can be changed by them has been severely limited. This was in response to a privacy hack that was found in the wild, where unscrupulous website owners were using them to track if users had visited a given site or not.

Direct Selectors

The last selector on the refreshers list is the direct selector. This is the most simple of all the selectors available, and as its name implies, it's used to directly select a known element name. You may also come across the *terms type selectors, element selectors,* and *tag selectors*; there is no agreed terminology for this type of selector, and which one is used depends heavily on the browser manufacturer. For our purposes, we use *direct selector* simply because we are directly targeting a named entity.

For example, if you wanted to target all **p** elements in a given document, you would simply do the following:

```
p
{
  color: green;
}
```

Code Listing 12: Using a direct selector

Once this is added to a style sheet, anything in the HTML markup of the document that consists of a ** ** tag would appear in green, regardless of its class, ID, or anything else.

Don't forget, you can easily combine basic selectors, so you could easily select only **h1** elements that are descendants of **divs**; for example, **h1** type elements have the **boxout** class applied to them and are nested inside a **div** element. This can be achieved using a selector such as **div .boxout h1**.

Summary

CSS3 introduced many new selector types, allowing even more precise control over how you target various parts of your HTML document. We'll be covering these in the next chapter as we start to explore the new things that CSS3 brings to the table for modern day web developers.

In this chapter, we took a quick look at CSS as a technology/language, and had a quick refresher on how it does what it does. Moving forward from here, we'll start to delve into topics specific to this new version of CSS. If you're still having problems making sense of CSS in general, then I suggest working through some of the various sites that are available to you on the internet today.

One particular site I can heartily recommend is HTML Dog. Not only is this site crammed full of all the essential information you might need to know, but it also carries a number of tutorials, reference guides, and resource links to other useful places on the internet that should help you further your knowledge.

Chapter 3 New Selectors

In the previous chapter you saw a brief refresher of the most common types of selectors used in CSS. These three types are known as the class, ID, and descendant selectors, and were all that was available before version 3 of CSS. Now that CSS3 is here, it has brought many new selectors as part of its new functionality.

We won't be covering every new selector in this chapter, but you'll see rather quickly the sheer flexibility now available to target your HTML elements in ways you might never even have imagined.

Many of these new selectors are implemented as pseudo-selectors (selectors that act as an extension to another selector, such as hover), and you could easily write an entire book just on the subject of selectors alone.

In this chapter, we'll look only at the most important of the new selectors available, along with a basic demonstration of their uses. For the rest, I strongly advise you to read many of the excellent resources available on the internet. Aside from the other links I've already mentioned, a great place to start purely for CSS topics is the CSS pages on the Mozilla Developer Network. The handy thing about MDN is the compatibility guides, which will tell you at a glance which selectors work with which browsers, and which ones will give you problems.

The Universal Selector (*)

The *target all* selector does exactly what it says: it targets everything it can in the space in which it is operating.

That means if you have it inside an ID-based selector, then everything that falls under the control of that ID selector will be affected.

Using the * symbol tells you straight away that it behaves like a wild card, being as greedy as possible and selecting as much as possible. At first, this may seem like a great thing, until you suddenly realize just how much it slows your page down on a mobile device, or on a page with millions of nested elements.

The universal selector is designed primarily to make sweeping changes as part of an initial setup, such as setting a pages entire CSS model to a known set of defaults (like 0-pixel margins and no borders).

Typically it's used in CSS resets to start at a known state. Many of the well-known, front-end UI frameworks make extensive use of it to put your page into a known state before applying any custom styles they may provide.

The universal selector should be used sparingly, and its sphere of influence should be made as minimal as possible so as not to cause any drastic performance bottlenecks.

If you wanted to make sure that the margins and padding were set to **0** on every element in your document, then you might apply this selector as shown in Code Listing 13.

```
*
{
  margin: 0;
  padding: 0;
}
```

Code Listing 13: Using the universal selector

After including this somewhere near the top of your style file, you can be sure that immediately after it, none of your rules will have any margins or padding applied to them.

The Adjacent Sibling Selector (+)

One of the simpler selectors to understand, the adjacent selector selects elements that are immediately next to each other, and does so only once for each occurrence.

Imagine you have the following HTML markup:

```
<div>
    <h1>Heading</h1>
    <p>Text line 1</p>
    <p>Text line 2</p>
    <h1>Heading</h1>
    <p>Text line 1</p>
    <p>Text line 2</p>
</div>
```

Code Listing 14a: HTML markup to demonstrate the adjacent selector

If you then applied the following CSS rule to this markup:

```
h1 + p
{
  color: red;
}
```

Code Listing 14b: CSS rule to demonstrate the adjacent selector

Only the **p** element immediately following the **h1** element would be changed to a red color. The second line of text (and any others that might follow) will remain unchanged, as will the **h1** elements themselves, as Figure 6 shows:

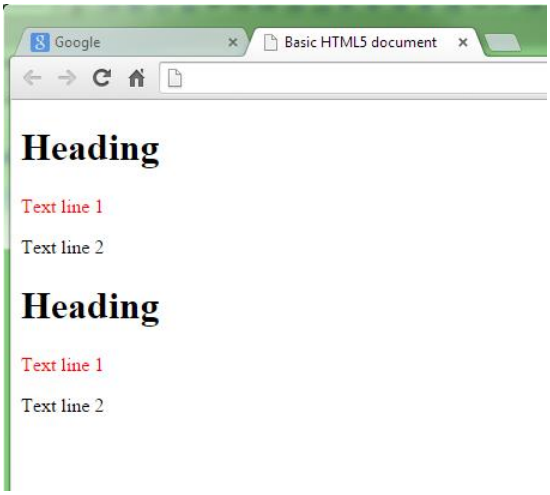


Figure 6: The expected result of Code Listing 14

One of the places that the adjacent sibling selector is most useful is in the creation of lead or opening paragraphs. Consider the following example.

```
<!doctype html>
<html>
<head>
    <title>Basic HTML 5 document</title>
    <link href="styles.css" rel="Stylesheet" type="text/css" />
</head>

<body>

    <section>
        <article>
            <h1>Monstrous Inbox</h1>
            <p>Today in local news, a monstrous email inbox was discovered
belonging to Mr Jones of durham.</p>
            <p>The email inbox was ruomoured to have 11ty billion unread emails in
it, but scientists say that calculating the actual amount is going to be almost
impossible.</p>
            <p>Mr. Jones had no comment, but he did let slip that he had no idea
how his inbox grew so large and so fast.</p>
        </article>
        <article>
            <h1>Jelly Table Legs</h1>
            <p>Furniture Watchdogs are today warning all people to be on the
lookout for tables with Jelly legs.</p>
            <p>In recent months there have been a spate of fake tables released
into the global furniture market, these fake tables are exceptionally dangerous
and could wobble over at any moment.</p>
            <p>People should exercise extreme caution when buying new tables, if
you’re unsure call a professional table tester to ensure you’re getting a genuine
product.</p>
        </article>
    </section>
```

```
</body>
</html>
```

Code Listing 15: Fake news listing

The code in Listing 15 represents what could be a set of news articles. I've marked the code up so that it's using the new semantic tags in HTML 5 to help define document structure and layout. Please note that you don't need to type in all the text found inside the **<p>** elements; I've included a large amount just for demonstration purposes.

If you render the output as is without any styling, you'll get something that looks like the following:

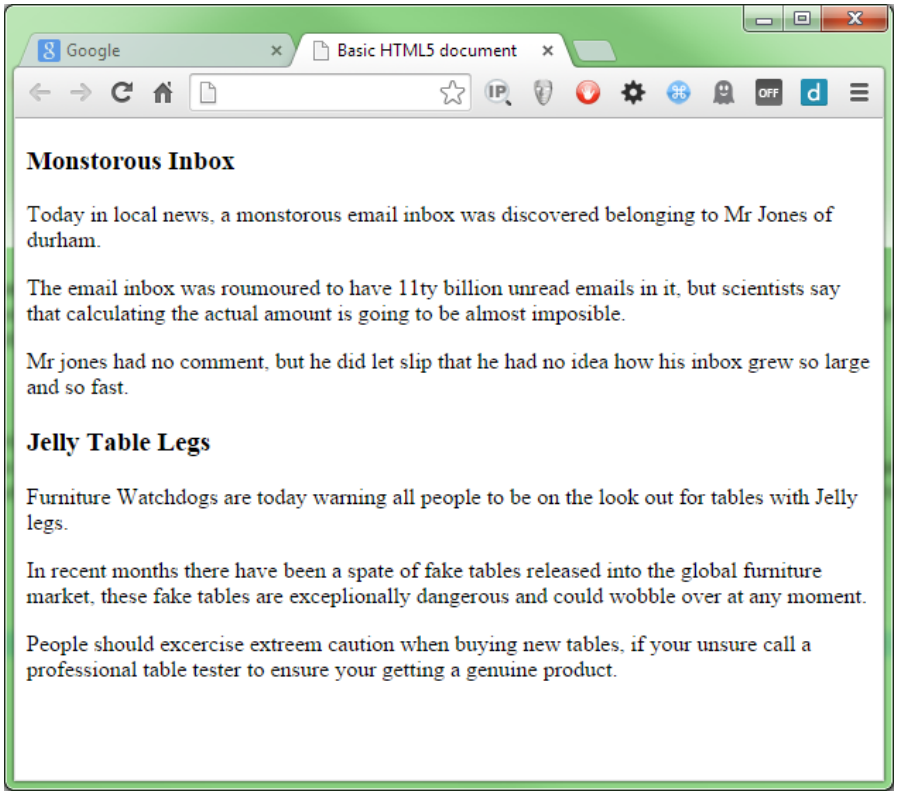


Figure 7: Unstyled news article example produced by Code Listing 15

With the help of the adjacent sibling selector, and some help from the descendant selector, we can easily style this output using a very minimal set of style rules.

The following CSS code shows how we might use the adjacent sibling selector to ensure that our first paragraph stands out from the rest in each article.

```
section
{
	font-family: sans-serif;
	font-size: 12pt;
```

```
}

section h1
{
	color: green;
	border-bottom: 1px solid green;
	font-size: 15pt;
}

section p
{
	color: grey;
}

section h1 + p
{
	color: black;
	font-size: 14pt;
	font-style: italic;
}
```

Code Listing 16: CSS code to style the news articles in Code Listing 15

As you can see in Code Listing 16, the CSS used is very simple, and what's more, it's driven from the **`section`** tag, which means any header or paragraph combinations that are NOT inside a containing section tag can still safely be used, and will not be changed to match the styles in Listing 16.

If we refresh the HTML output with the new styles, and do a little text editing, we should now see something like this:

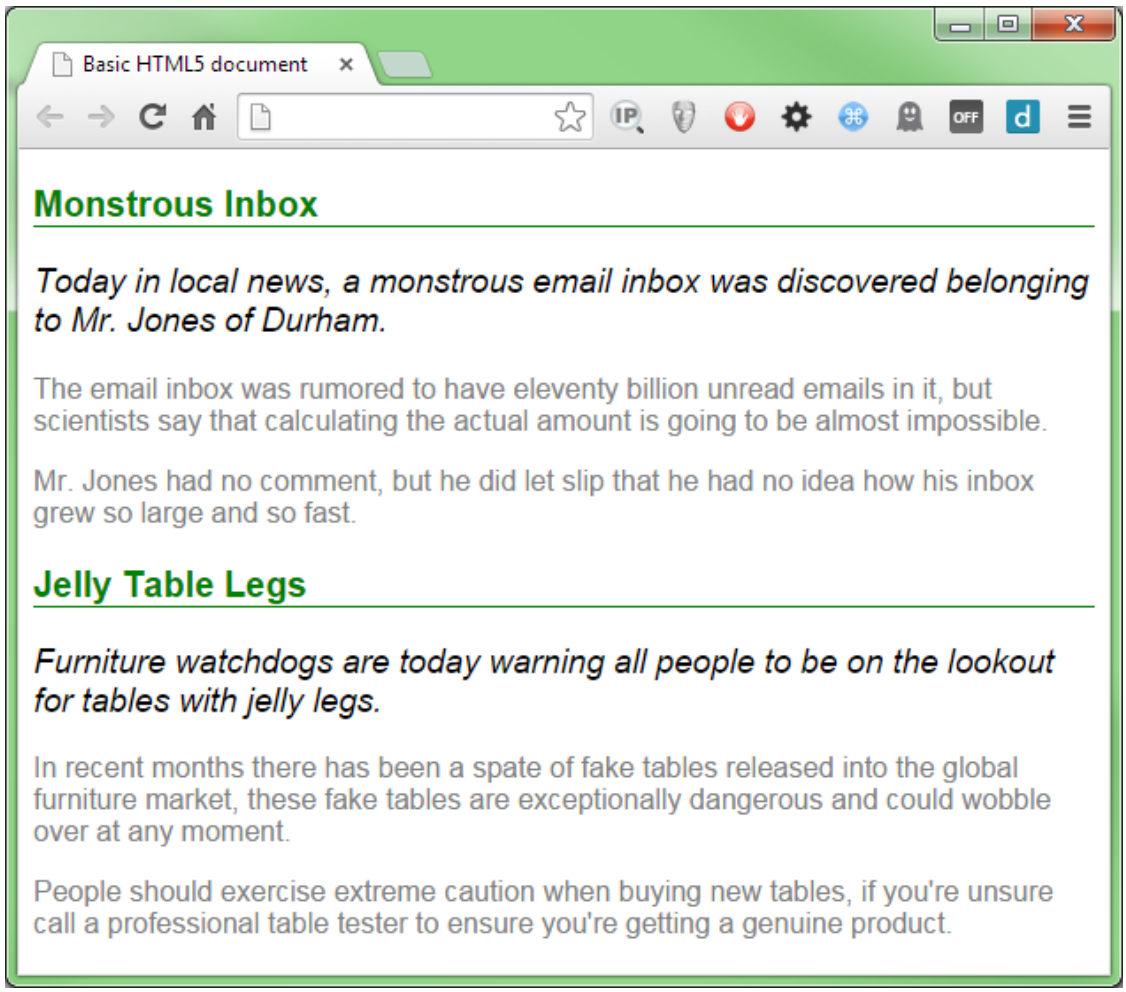


Figure 8: Code Listing 15 with the CSS rules in Code Listing 16 applied

The Direct Child Selector (>)

As you've already seen, the descendant selector allows you to target child elements down the chain of elements in your markup.

If you want to target paragraph tags that reside inside a **`div`** with a class type of **`news`**, for example, then a selector of **`div.news p`** will do the job. There is, however, one drawback: the descendant selector is greedy.

When we say a selector is greedy, we mean that it will attempt to match as much as it possibly can (anyone who has any experience with regular expressions will know what I'm talking about).

What this means in practice is that not only will the elements at the first level of selection be selected, but any that match below that will too.

The direct child selector solves this by being less greedy, and going no further than the first level below the root selector.

Consider the following HTML markup:

```
<div class="test">
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
    <div>
        <p>Paragraph 3</p>
    </div>
    <p>Paragraph 4</p>
</div>
```

Code Listing 17: HTML markup to demonstrate the direct child selector

If we render Code Listing 17 in our browser, we'll get four simple paragraph elements.

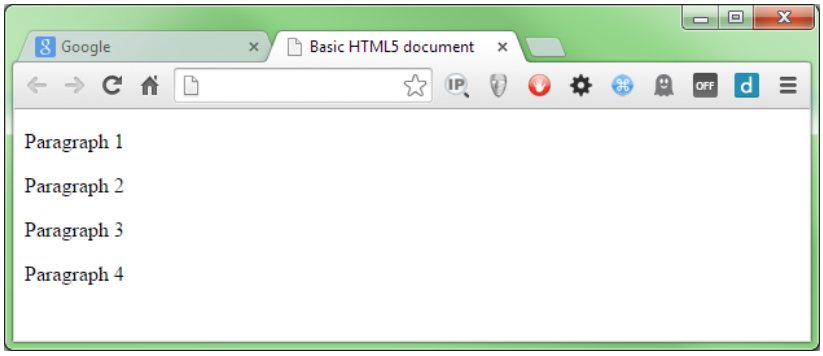


Figure 9: HTML rendered using Code Listing 17

You'll notice from the markup that paragraph 3 is inside a nested **`div`** element; if we style this block using a standard descendant selector, we'll get the following output:

```
div.test p
{
  color: red;
}
```

Code Listing 18: CSS Rule to style the code in Listing 17 using a descendant selector

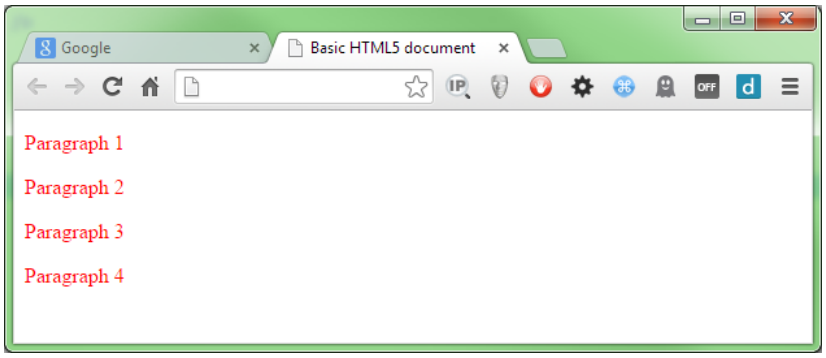


Figure 10: Output produced when combining the HTML in Code Listing 17 with the rule in Code Listing 18

At first glance, this output might seem ok, and in general may actually be what you wanted to achieve. Consider this, however: why is paragraph 3 being embedded in a further **`div`** element?

It may have been created that way with the intention of styling it differently than the other paragraph elements in the block.

Let's change our rule, so it uses a direct child selector as follows:

```
div.test > p
{
  color: red;
}
```

Code Listing 19: CSS Rule to style the code in Code Listing 17 using a direct child selector

You should now see that it skips the paragraph in the nested element, but continues to style the others as expected.

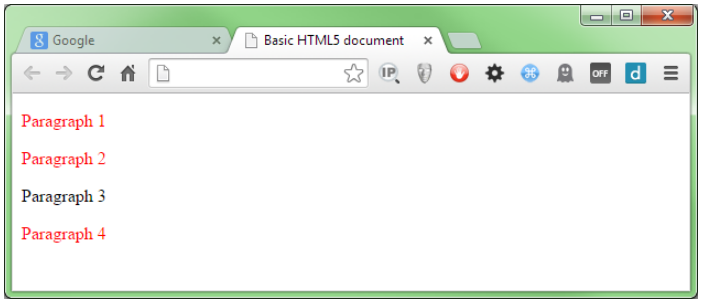


Figure 11: Output produced when combining the HTML in Listing 17 with the rule in Listing 19

This comes in handy when creating things like pop-up menus, where you might like all the top-level elements to take on an unselected look and feel, but where nested items forming sub-menus need to be left alone when a parent element is selected.

The General Sibling Selector (~)

The *general sibling* selector, like its close cousin the adjacent sibling selector, is used to select elements that follow a parent of a given type. However, unlike its cousin, it attempts to select more than one match in the markup hierarchy.

The adjacent sibling selector will at most (as you saw previously) select only the first occurrence of a given match; the general sibling selector, however, will attempt to select that match, as long as the hierarchy continues to match.

If we build two unordered lists as follows:

```
<p>List 1</p>
<ul>
```

```
    <li><a href="#">Link 1a</a></li>
    <li><a href="#">Link 2a</a></li>
    <li><a href="#">Link 3a</a></li>
    <li><a href="#">Link 4a</a></li>
</ul>
<ul>
    <li><a href="#">Link 1b</a></li>
    <li><a href="#">Link 2b</a></li>
    <li><a href="#">Link 3b</a></li>
    <li><a href="#">Link 4b</a></li>
</ul>
<p>List 2</p>
<ul>
    <li><a href="#">Link 1a</a></li>
    <li><a href="#">Link 2a</a></li>
    <li><a href="#">Link 3a</a></li>
    <li><a href="#">Link 4a</a></li>
</ul>
<ul>
    <li><a href="#">Link 1b</a></li>
    <li><a href="#">Link 2b</a></li>
    <li><a href="#">Link 3b</a></li>
    <li><a href="#">Link 4b</a></li>
</ul>
```

Code Listing 20: HTML markup to demonstrate the sibling selector

As with the rest of the examples, if we first render this with no styling, we'll get the default standard rendering:

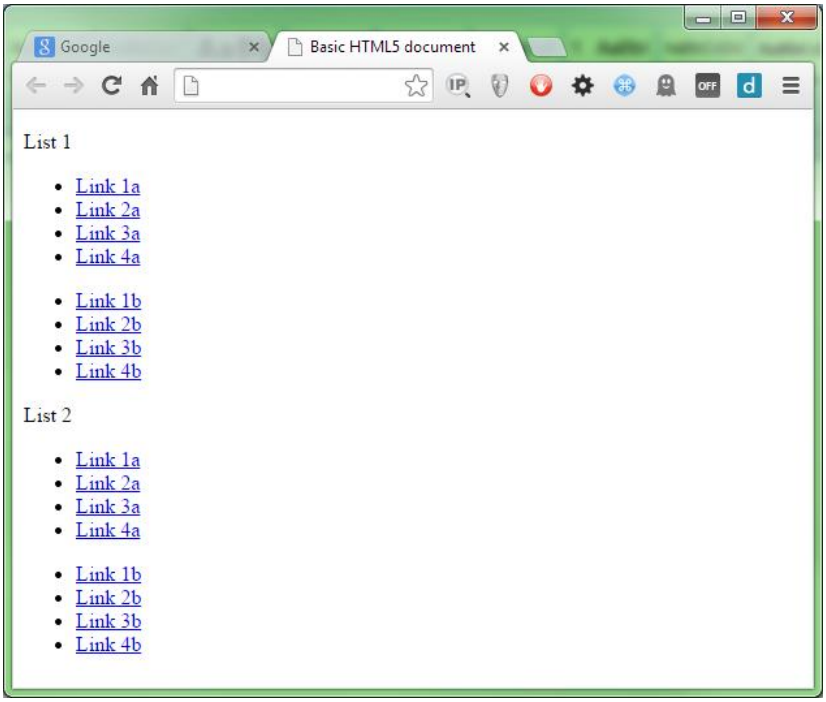


Figure 12: Output produced by the HTML in Code Listing 20 with no style rules applied

To illustrate the difference, we're first going to style the list using the adjacent sibling selector you saw earlier, using the following rule.

```
p + ul a
{
  color: red;
}
```

Code Listing 21: Adjacency selector for Code Listing 20

When we render Code Listing 20, using the rule in Code Listing 21, we get the following output:

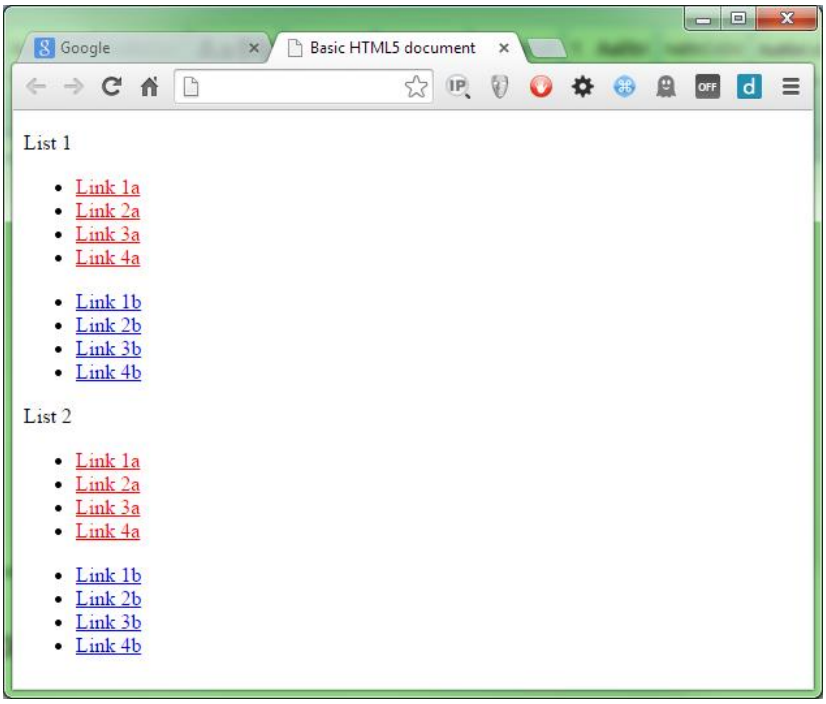


Figure 13: Output produced by Code Listing 20 with the adjacency selector applied

If you've been following along with the code, this shouldn't come as a surprise. The first group is styled, the second is not.

If we now switch our code to use the general sibling selector, however:

```
p ~ ul a
{
  color: red;
}
```

Code Listing 22: Sibling selector for Code Listing 20

We should see a difference in the selection:

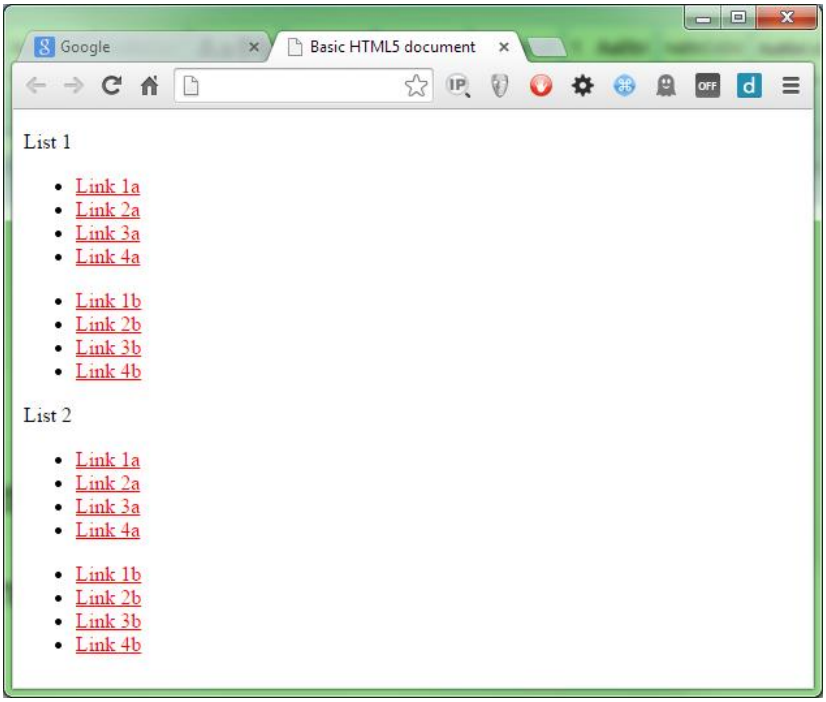


Figure 14: Output produced by Code Listing 20 with the sibling selector applied

While the adjacent sibling selector stops at the first match, the general sibling selector does not; the general sibling selector will continue down the chain, matching anything in the rule as long as the element before it matches.

The difference between adjacency and descendancy

At this point you might be wondering why we have two ways of achieving apparently the same thing. It's important to remember that while we have styled similar scenarios so far in this chapter, we've done this in rather different ways.

When we talk about elements that are adjacent to each other, we mean these elements are rather like neighbors in a street of houses. If you were to draw a line down the alignment of each house, you'd see that everything was at the same level. In the following code, each of the **`div`** elements with a **`class`** attribute of **`house`** are adjacent to one another:

```
<div class="street">
    <div class="house"></div>
    <div class="house"></div>
    <div class="house"></div>
    <div class="house"></div>
    <div class="house"></div>
</div>
```

Code Listing 23: Adjacent elements

When we talk about elements being descendants of each other, we're talking about a parent-to-child-like relationship. If we apply this to our street example in Code Listing 23, we might have "tenants" who occupy each house.

```
<div class="street">
    <div class="house"><p class="tenant"></p></div>
    <div class="house"><p class="tenant"></p></div>
    <div class="house"><p class="tenant"></p></div>
    <div class="house"><p class="tenant"></p></div>
    <div class="house"><p class="tenant"></p></div>
</div>
```

Code Listing 24: Descendant elements

As you can see in Code Listing 24, we now have a tenant inside each house. Tenants are not adjacent to each other because you have to leave the house, move down, and then enter into the next house to reach the next tenant.

However, since you can reach a tenant from a house, then a tenant becomes a descendant of a house.

Child and grandchild terminology are defined as a specialist case of descendancy. Just like in a family tree, a tenant in our example might have children, and those children might have grown up and had children of their own.

A child of a child would still be a descendant of a tenant, but it would not be a child of a tenant; it would be a grandchild, something like the following:

```
<div class="street">
    <div class="house">
        <div class="tenant">
            <div class="tenantsChild">
                <div class="childsChildButTenantsGrandChild"></div>
            </div>
        </div>
    </div>
</div>
```

Code Listing 25: Code showing child relation to dependency

If you were to create more **`div`** elements with a class of **`tenantsChild`** next to each other within the confines of the tenant, those **`tenantsChild`** elements, like the houses, would be adjacent to each other.

It takes a little bit of practice until you get into the habit of thinking this way, but if you think about HTML in its native form as a specialist, XML-based language, then you see that the nesting starts to make sense. Once the nesting makes sense, the flow and layout of the selectors start to fall into place too.

Don't worry if you struggle to make sense of it straightaway. It took me a long time to unravel it all too, and even now I still get it wrong sometimes.

So far we've covered the main base selectors. If you want to read up on the rest of them, you can find the official documentation on the W3C site.

I do think a warning is in order here, though. While the official W3C documentation is the official source of all things HTML, it's also a very difficult site to read.

There are easier sources of information, such as Mozilla and HTML Dog. I would strongly advise you digest those first, then visit the W3C site once you have a better understanding of the topic.

Attribute Selectors

To finish this chapter off, we're going to take a look at something that's been massively expanded on in CSS3: attribute selectors.

There has been some minimal effort to introduce a very small subset of this type of selector in previous versions, but it went largely unnoticed, so most developers never stopped to look at what it had to offer.

In CSS3 however, attribute selectors are all-powerful, and allow you to do some rather smart styling of elements.

First off, we need to define what an attribute selector actually is.

Unlike the selectors you've seen so far, an attribute selector is not a single character construct that sits between two targets; it's a multi-character appendage to a parent rule, which contains its own internal processing rules.

In a small way, attribute selectors are a little bit like having a custom set of search expressions, specifically for hunting out elements based on the value (or part of) of an attribute attached to them.

Take a look at the following HTML markup.

```
<ul>
    <li><a href="#" rel="external">External Link 1</a></li>
    <li><a href="#" rel="internal">Internal Link 1</a></li>
    <li><a href="#" rel="internal">Internal Link 2</a></li>
    <li><a href="#" rel="external">External Link 2</a></li>
    <li><a href="#" rel="external">External Link 3</a></li>
</ul>
```

Code Listing 26: HTML markup with custom attributes

In Code Listing 26, you can see I've created a fairly standard, unordered list containing items with anchor elements in them. It's nothing unusual, and something you'll often see in navigation links in HTML documents.

What I've done, however, is to use the **rel** attribute to mark up the relation of the link, specifying if it's an external or an internal link.

If we wanted to style the two different types in different ways, we could, in theory, just add a class such as **.internal** or **.external**. However, there may be times that HTML markup is generated inside a process over which you have no control, so you may not be able to add classes to the output in this manner.

This is where the attribute selector comes in. You can use it to target a specific named attribute within the element, and as you'll see in just a moment, you can use it to not only target a full string, but part of a string too.

First, let's have a look at how we might apply it to our previous links example. Add the following style rules to your style sheet (assuming you've already added the preceding HTML to your document):

```
a[rel="external"]
{
	color: red;
}

a[rel="internal"]
{
	color: green;
}
```

Code Listing 27: Attribute Selector rules for Code Listing 26

If you render this in your browser, you should see something like the following:

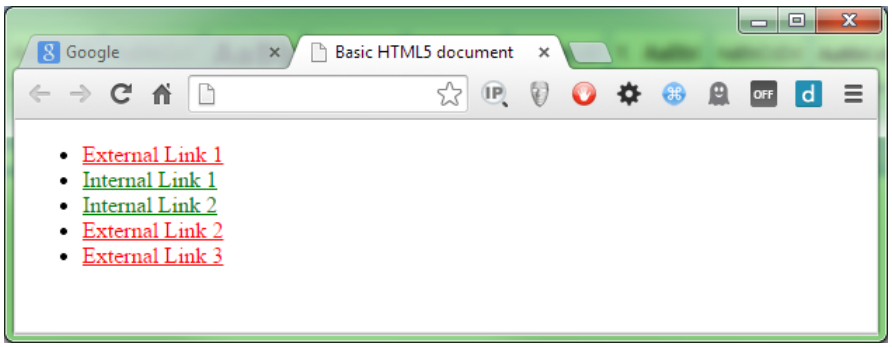


Figure 15: Code Listing 26 with the CSS rules in Code Listing 27 applied to them

Just by using two different rules, we've easily targeted the two different types of links, simply because of the difference in the **rel** attribute.

A real-world example might look something like the following; change the code in Listing 26 to read as follows:

```
<ul>
```

```
    <li><a href="#">Link 1</a></li>
    <li><a>Link 2</a></li>
    <li><a>Link 3</a></li>
    <li><a href="#">Link 4</a></li>
    <li><a href="#">Link 5</a></li>
</ul>
```

Code Listing 28: HTML markup modified from Code Listing 26

By default, most browsers won't make a link clickable that has no **`href`** attribute, and for the default styling at least, they won't be styled using the usual blue underlining.

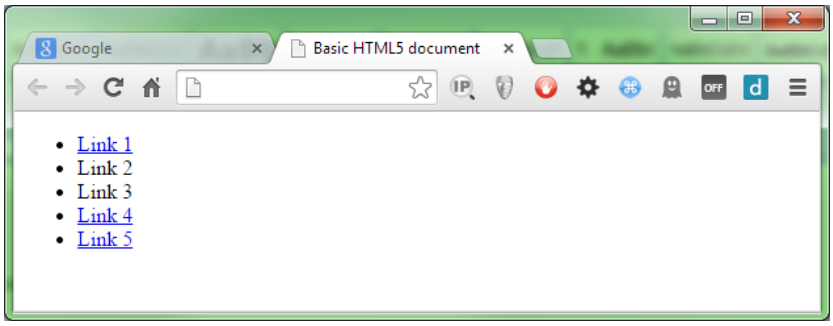


Figure 16: Output from Code Listing 28 with no styling.

However, in many cases (especially if you're using a larger framework), you might find that anchor tags in general have been given a uniform style to maintain a consistent look and feel.

For example:

```
a
{
  text-decoration: none;
  color: green;
}
```

Code Listing 29: Possible rule to give a uniform look and feel to an anchor

This code might be used to apply that uniform style, and would make our list look as follows:

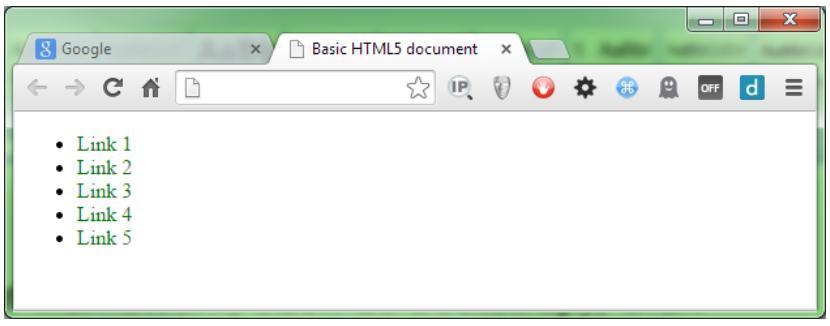


Figure 17: Code Listing 28 with a uniform style applied to the anchor element

You can immediately see that you've lost any contextual information that might suggest to you a link has no destination, and as a site user, you have no idea which links are good, and which are bad.

You could go through your code and add a **`class="nodestination"`** to each link with a missing **`href`**, but what if someone then added links to the database, and all of a sudden the link then has a valid destination? In this case, you're faced with the task of finding all those links that didn't have an **`href`**, and now do, so you can remove the class name and restore normality.

The other way to address this problem is to use just the attribute selector with no extra content on its own, as follows:

```
a[href]
{
  text-decoration: none;
  color: green;
}
```

Code Listing 30: Attribute rule to target only anchors that have an href attribute

When we now refresh our page, we should see that we now get different styling than we did before:

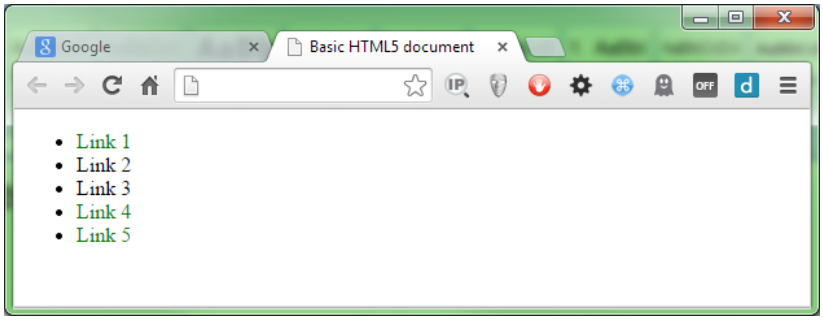


Figure 18: Code Listing 28 with a uniform style, only applied to anchors with valid destinations

Now in order to have the links styled correctly, all you have to do is let your web application platform generate links with **`href`** attributes, and leave the **`href`** attributes out of anything that does not yet have a link.

There is one question remaining, however.

What if we also want to style the remaining elements that don't have the **`href`** attribute, for example, to make them a lighter color so they perhaps look disabled?

CSS3 also has this covered with a pseudo selector. We'll learn more about the new pseudo selectors in the next chapter; for now, we can solve the styling of anchor tags with missing **`href`** attributes quite simply by using the following attribute/pseudo-styling rule:

```
a:not([href])
{
```

```
  text-decoration: none;
  color: silver;
}
```

Code Listing 31: Attribute rule to target only anchors that don't have an href attribute

If we re-render our document after changing our style rule to match Code Listing 31, we should get the following:

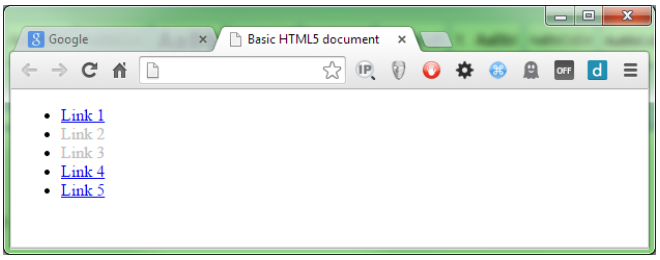


Figure 19: Code Listing 28 with an inverted attribute style

The following rules take advantage of the cascading nature of CSS, and combine things logically:

```
a
{
  text-decoration: none;
}
a[href]
{
  color: green;
}
a:not([href])
{
  color: silver;
}
```

Code Listing 32: Combined rules to style links with missing href attributes automatically

In theory, we could actually reduce that further, and just add the green color to the base anchor rule. This would apply the green as a default, but change it to silver if no **`href`** existed. The idea here is just to show what's going on in the simplest possible terms, and expanding things out often makes concepts clearer to understand.

If we render this now, we should see something like the following:

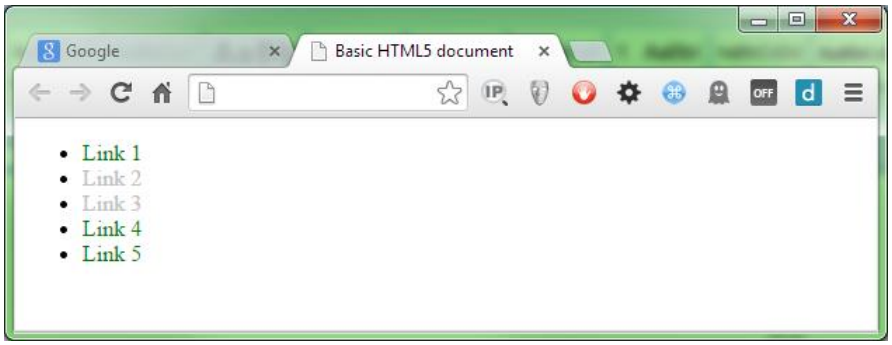


Figure 20: Code Listing 28 with the combined styles applied

The main advantage to take away from this example is that you now no longer have to worry about making sure that a specific class is applied to a given link, making maintenance of the document much easier and less error-prone. If you're running under a CMS such as Umbraco or WordPress, then all you have to care about is whether links have valid destinations or not.

Note: I've previously used an attribute selector in this fashion inside normal paragraph text to automatically highlight links that are in-line with the general text once they become active, and to make them look exactly like the paragraph text when they aren't active.

I mentioned earlier that attribute selectors can also match on partial contents in a given attribute. There are additions to the main selector that allow you to check if an attribute's value begins with, ends with, or contains a given substring.

This can be very useful in the case of external and internal links.

Think back to the code we saw in Code Listing 26; we used the **`rel`** attribute to define an internal or external relationship on an anchor element.

In terms of the W3C specifications, this was probably not the best way to do this, and quite likely, the HTML would fail to validate if it were tested with an HTML 5 validation tool.

This doesn't mean we have to go back to using a class attribute, however.

Imagine for a moment that all your external links had a fully qualified URL, and all your internal links just had a relative page leaf. In this case, Code Listing 26 might now be re-written as follows:

```
<ul>
    <li><a href="http://external.com/link1">External Link 1</a></li>
    <li><a href="internal/link1">Internal Link 1</a></li>
    <li><a href="internal/link2">Internal Link 2</a></li>
    <li><a href="http://external.com/link2">External Link 2</a></li>
    <li><a href="http://external.com/link3">External Link 3</a></li>
</ul>
```

Code Listing 33: Code Listing 26 rewritten to not use rel attributes

If you examine the difference in the **`href`** attributes, you'll see that all the external links start with “http://” and the internal ones do not.

If we use the “attribute value starts with” selector, we can now easily style the external ones with a different color.

We can set a uniform style for anchor tags in general, which would reflect the style for internal links by default, then be altered if it reflected an external link.

To use the starts with attribute selector, we specify a **`^=`** between attribute name and value, as the following rule shows:

```
a[href^="http:"]
{
  color: red;
}
```

Code Listing 34: CSS style rule to change the color of attributes that start with “http”

If we apply that to the HTML markup in Code Listing 33, and then render it in the browser, we should get the following:

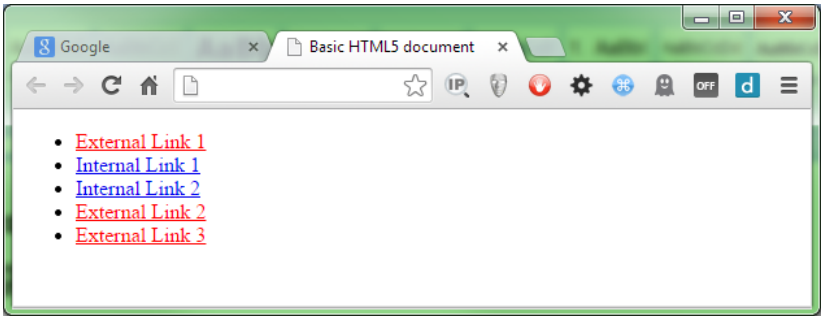


Figure 21: HTML from Code Listing 33 with the rule in Code Listing 34 applied to it

Now if we add in a uniform rule as we did previously, so that links are green by default, we end up with output that looks like the following:

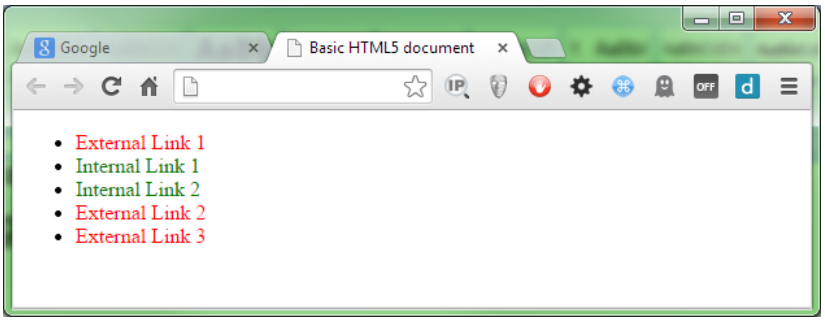


Figure 22: HTML Code Listing 33 with the final styling in place

I'm sure you can quickly realize how easily this could be combined with the rules to detect the presence of the attribute and make further choices on links with no **`href`** attributes, too.

The remaining parts of the attribute selector are the “ends with” and “contains” selectors.

“Ends with” comes in handy if you want to style links based on file type. For example, you could do the following:

```
<ul>
    <li><a href="photo1.png">Photo 1</a></li>
    <li><a href="photo2.png">Photo 2</a></li>
    <li><a href="sound1.mp3">Sound 1</a></li>
    <li><a href="sound2.mp3">Sound 2</a></li>
    <li><a href="document.pdf">Document</a></li>
</ul>
```

Code Listing 35: Code listing 33 rewritten to list file links

Then, create the following rules using the **$=** attribute selector as follows:

```
a[href$=".png"]
{
  color: red;
}
a[href$=".mp3"]
{
  color: green;
}
a[href$=".pdf"]
{
  color: blue;
}
```

Code Listing 36: Ends With attribute rules to style based on file type

When rendered, it should give the following:

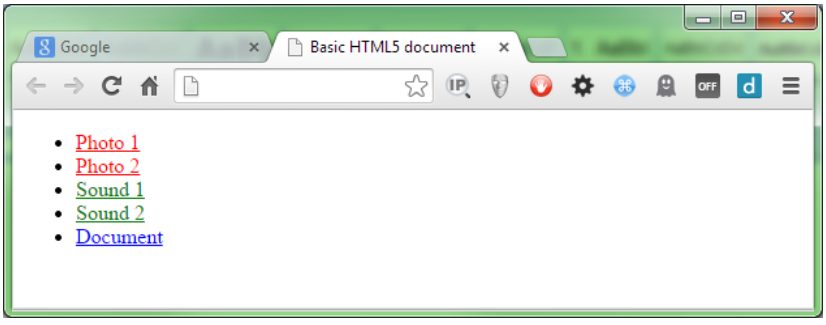


Figure 23: Code Listing 35 showing the output from the “ends with” attribute selector

The final attribute-based selector is the contains selector, and does exactly as it says on the tin.

If an attribute value contains the given substring, then ***=** will match it.

For example, take a look at the following set of anchor tags:

```
<ul>
    <li><a href="/admin/users/list">List Users</a></li>
    <li><a href="/admin/personal/settings">Your Account Settings</a></li>
    <li><a href="/admin/users/edit">Edit Users</a></li>
    <li><a href="/admin/users/report">Users Report</a></li>
    <li><a href="/admin/personal/profile">Your Profile</a></li>
</ul>
```

Code Listing 37: Code Listing 35 rewritten to demonstrate "contains" attribute selector

With the following rules:

```
a[href*="users"]
{
  color: red;
}
a[href*="personal"]
{
  color: green;
}
```

Code Listing 38: Contains string attribute rules to style based on substring

This will produce the following output:

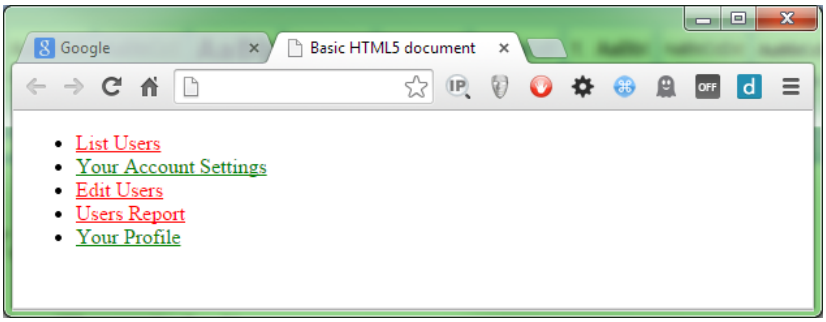


Figure 24: Output produced by Code Listing 37 with the rules in Code Listing 38 applied

What we've covered here hasn't even scratched the surface of the potential available, especially when it comes to attribute selectors.

Under HTML 5 there's a new attribute type called a data-attribute. This new attribute takes the form of:

`data-`

followed by a custom name, for example:

`data-username`

`data-recordindex`

You can use these data attributes on any element, for any purpose, and when combined with the correct selectors, they allow you use HTML 5 to describe its own intentions and style easily.

Summary

In this chapter we took a fairly deep dive into the main new selectors added in CSS3. We've seen how we're no longer limited in how we can target our elements to style, and how we no longer have the restrictions we used to have, allowing almost limitless expression in our rules.

In the next chapter we'll take a closer look at the new pseudo selectors, which, when combined with what we've learned so far, will take the capabilities we have to new heights.

Chapter 4 New Pseudo Selectors

In the previous chapter, we covered the new main selectors added to CSS3. Now, we go further by exploring the new pseudo selectors that have been added.

Pseudo selectors are different than the regular selectors because they can't be used on their own. Instead, they need to be combined with other selectors, such as those in the previous two chapters, in order to do their job. At least, that's how they're described in the mainstream.

The truth of the matter is that many of the pseudo selectors can now indeed be used without any suffixing rule (as you'll see in a moment), but you do still get the best from them when combining them with a regular selector.

Some of the selectors you'll see in this chapter are actually not new in CSS3 and have been around quite some time. In previous versions, however, they either were not well known, or because of the IE6/IE7/IE8 compatibility generation problems, they were actively avoided and not used much.

To kick things off, we'll deal with the oldest ones first.

:focus

As the name suggests, **`focus`** is used when an input element has focus within a document. This is usually used in forms to highlight the active field.

Let's create a mock-up form using the following HTML markup:

```
<form action="#" method="post">
    <label for="txtUserName">User Name</label><br />
    <input type="text" id="txtUserName" name="txtUserName" /><br /><br />
    <label for="txtEmail">Email</label><br />
    <input type="text" id="txtEmail" name="txtEmail" /><br /><br />
    <input type="submit" value="Submit" />
</form>
```

Code Listing 39: Mock HTML form markup

This will give us a very generic looking HTML form.

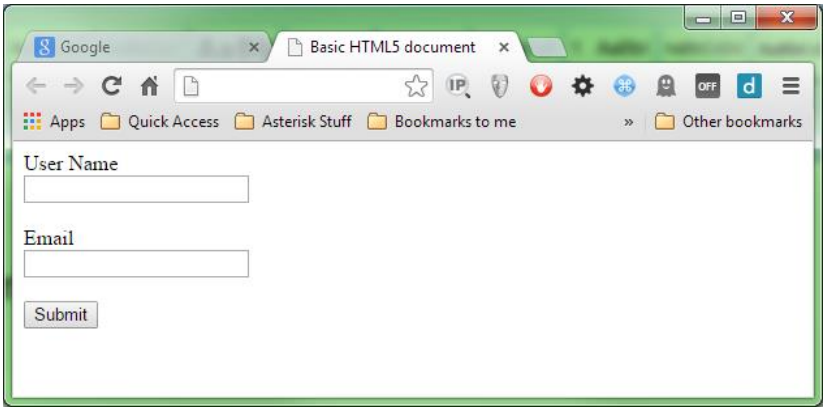


Figure 25: Generic form produced by Code Listing 39

Let's add the following simple rule to our style sheet:

```
:focus
{
  background-color: beige;
  color: green;
}
```

Code Listing 40: Simple Focus CSS rule

If we re-render our browser, and click into one of the form fields so that it gets the focus, we should see that it changes to reflect the selected state.

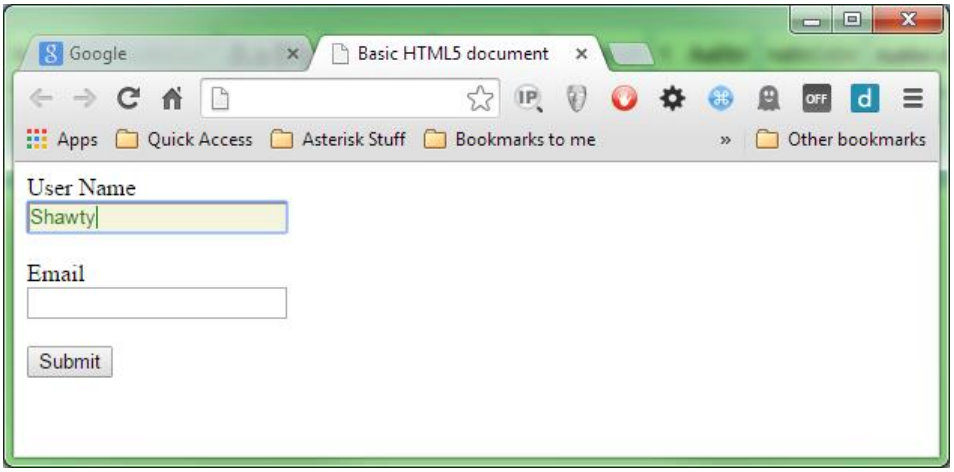


Figure 26: Our simple form with the focus rule applied

You might notice in the example that I've used **`:focus`** on its own without extending anything. In CSS3, many of the pseudo selectors can now be used in this manner; it simply means apply it to anything that understands the selector.

For our example, any element that can receive focus from the user will change its color as specified when focus is given to it.

A slight diversion into visited status

I just want to briefly break off and show another pseudo selector that's been around since V2 of the CSS spec. It can be used alone in the same way as **`focus`**, but can also be used maliciously to find out a person's browsing history.

If you place some external anchor tags in your document as follows:

```
<ul>
    <li><a href="http://www.google.co.uk/">Google</a></li>
    <li><a href="http://www.yahoo.co.uk/">Yahoo</a></li>
    <li><a href="http://msn.com">MSN</a></li>
</ul>
```

Code Listing 41a: External URLs

Then apply the following style rules:

```
:link
{
  color: green;
}

:visited
{
  color: red;
}
```

Code Listing 41b: Styles for the external URLs in 41a

When you render your document, you'll see the following.

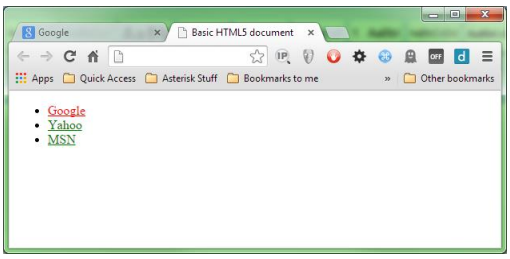


Figure 27: Links rendered by Code Listing 41

You might see that some or even all of the links have a red color. In my case, only the Google link does, and this is because the Google main UK site is in my browser history, whereas the other two are not.

If you look at the CSS, you'll see that the styling is provided by the **`visited`** pseudo selector, giving you, the user of the page, instant visual confirmation that you recently visited that page.

While this might seem fairly innocuous on the surface, if you dig down under the hood and look at some of the new JavaScript API's that have been added in newer specifications, you'll find that there's an API call that goes by the name of **`getComputedStyle`**. Using this call, it's possible to get the actual computed style that's visible on the element in the browser, or at least it was.

In today's modern browsers, this hole is largely patched, and attempting to use this call to get the color of the link will in fact just return whatever the base color is. In our case, all we get back (even for the Google link) is the green color.

I have, however, heard rumors that it's possible to use JQuery and MooTools to retrieve the correct value and make a distinction.

This is just a small example of the fact that anything can be used to breach a user's privacy or security, even something as common as CSS. Always make sure that your rules are specific if possible. Just because you can use them globally without a rule to extend, doesn't mean you should. If you tie them to given rule sets, it means that your style rules can't be abused by people hacking around with the browser tools.

:disabled and :enabled

Among the many new attributes that can be applied to elements in the HTML 5 spec is the **`disabled`** attribute.

You use it like any other attribute by simply adding "disabled" to an element in your markup.

Let's reuse the form mock-up we created in Code Listing 39, but change it so that it looks as follows:

```
<form action="#" method="post">
    <label for="txtUserName">User Name</label><br />
    <input type="text" id="txtUserName" name="txtUserName" disabled /><br /><br />
    <label for="txtEmail">Email</label><br />
    <input type="text" id="txtEmail" name="txtEmail" /><br /><br />
    <input type="submit" value="Submit" />
</form>
```

Code Listing 42: Mock HTML form markup with disabled elements

When you render it, you should see that the first input element can't be used now.

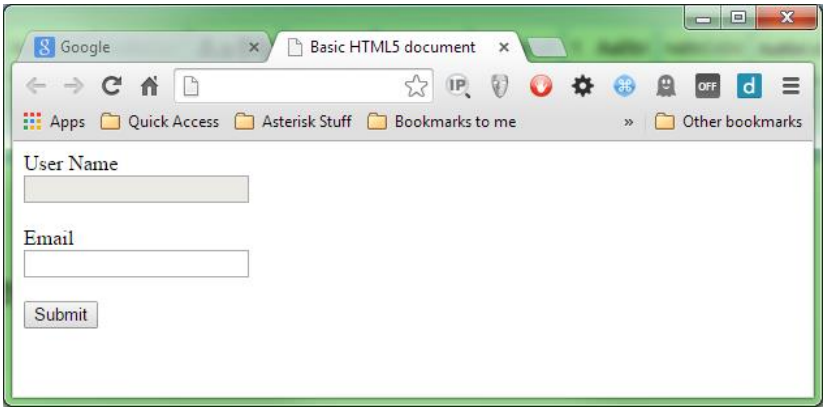


Figure 28: Simple form with a disabled input

Let's add the following rule to our style sheet:

```
input:disabled
{
  background-color: red;
  border: 2px solid blue;
}
```

Code Listing 43: CSS rule to style disabled elements

You'll see a drastic change when you re-render the form.

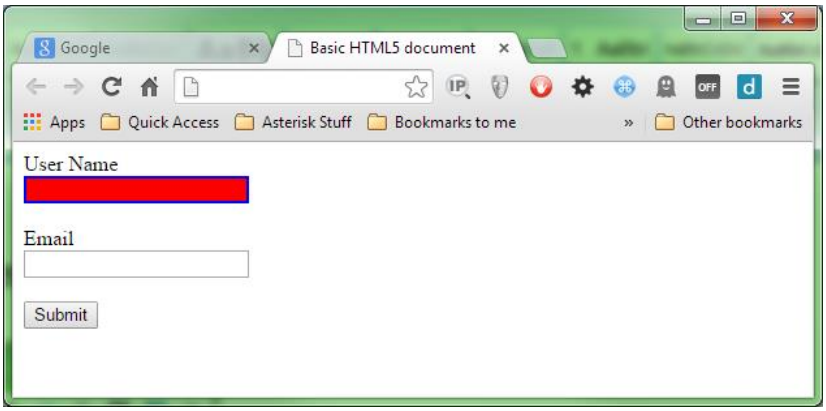


Figure 29: Simple form with disabled pseudo class in Code Listing 43 applied

If you remove the **`disabled`** attribute from the element, the input will revert back to its original state.

This rule is typically used to show read-only fields in a record editor, or to show fields that are not applicable to the current form. For example, you might mark up a form so that address fields are shown only when a tick box to send mail to that address is ticked.

If you use the Bootstrap UI framework, you might have seen that some elements when used in a form have a small, red stop circle displayed over them when you hover over them with a mouse pointer. This is performed in the same way, using the disabled rule to change the style of the mouse pointer.

The **`enabled`** pseudo selector is the logical opposite of disabled, and applies the given style to an element that is NOT disabled, as the following example shows:

```
input:enabled
{
  background-color: yellow;
  border: 2px solid green;
}
```

Code Listing 44: CSS rule to style enabled elements

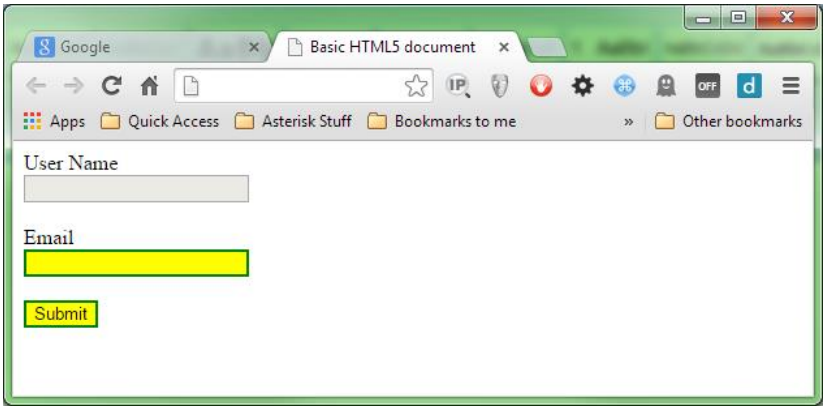


Figure 30: Browser form showing enabled CSS rules applied

An element is classed as enabled when there is no reason to disable it, or specifically, when it does not have a **`disabled`** attribute applied.

Note that an element is also classed as disabled if the element's Boolean **`disabled`** property is set to **`true`** on its document object using JavaScript. In this case, as with other selectors, the application of the disabled and enabled selectors are explicitly applied based on the element's state, meaning you don't have to keep making sure the correct elements have the correct classes applied.

:valid and :invalid

Another of the great features added to HTML 5 is browser-side validation. With browser-side validation, you can make the browser verify the inputs to a form before the form is submitted.

Like the disabled selector, the pseudo selectors here apply to elements only when we use the extra attributes and markup provided in the specification.

Using our form again, let's make another change so that you have the following HTML code:

```
<form action="#" method="post">
    <label for="txtUserName">User Name</label><br />
    <input type="text" id="txtUserName" name="txtUserName" required /><br /><br />
    <label for="txtEmail">Email</label><br />
    <input type="email" id="txtEmail" name="txtEmail" /><br /><br />
    <input type="submit" value="Submit" />
</form>
```

Code Listing 45: Mock HTML form markup with validatable elements

The changes might not be obvious at first, but if you look at the user name box, you'll see that it now has a **`required`** attribute.

The email input, however, no longer has a type of **`text`**; it now has a type of **`email`**.

If we render this, and then click **Submit** without entering any values, we'll see something similar to the following:

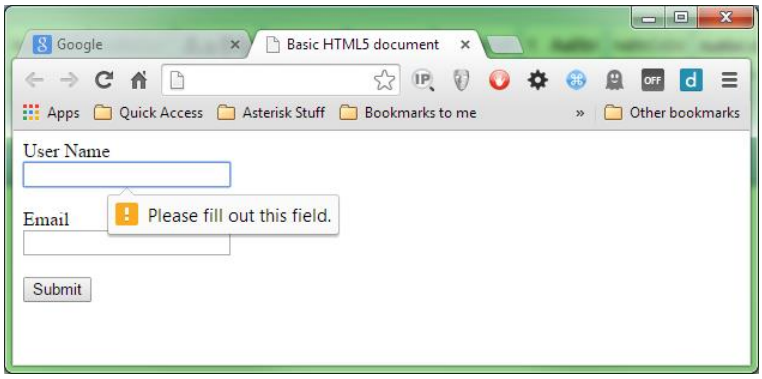


Figure 31: Our basic form showing the required attribute in action

If we add something in the name box, and put something in the email box, but deliberately format the email incorrectly, we'll also get validation on the format of the email.

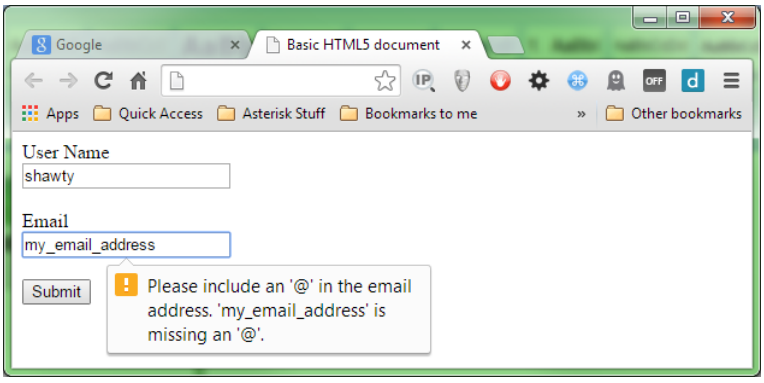


Figure 32: Our basic form showing the email validation in action

There's more to the validation classes that I'll be showing here, but for now, add the following rules to your style sheet:

```
input:valid
{
  background-color: green;
}

input:invalid
{
  background-color: red;
}
```

Code Listing 46: CSS3 Validation rules

If you then render the page, you should get the following:

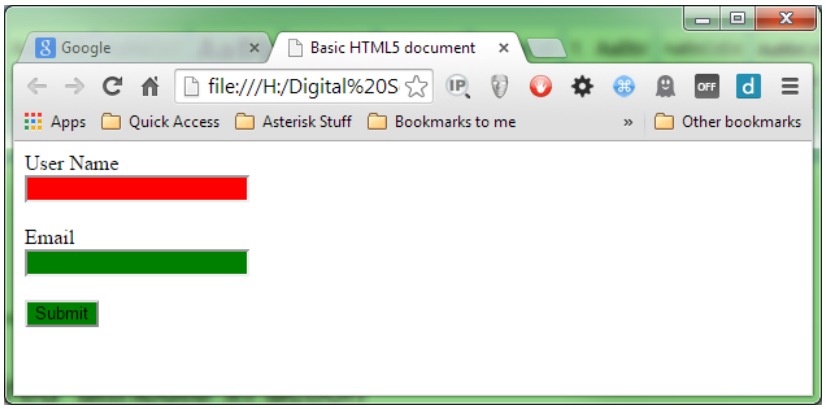


Figure 33: Simple form with validation styles from Code Listing 46 applied

The user name is immediately red because it's required, so until it's filled in, it's classed as invalid.

The email is valid, as we've only told it that it must have its input validated as an email address. We haven't said that it's required as input, so having a blank email is valid. However, if you start typing in it, it will change to red and won't change back to green until the input is either correctly formed, or removed entirely.

Using **`valid`** and **`invalid`** is a great way of giving feedback to the user, but seeing them immediately as soon as the form is displayed is generally not considered good practice.

For this reason, you would usually NOT apply them directly to input elements as I have (in fact, you can see I've just used a very broad selector because the button is green too). Usually, you'll attach them to a class/id name, then check and apply them using JavaScript when the form is submitted.

As this book is about CSS, I'm not going to go into detail about that here. If you want to venture further down the rabbit hole of HTML 5 validation, there's a post on my blog that will get you started.

:in-range and :out-of-range

Following on with the validation pseudo selectors, we have **in-range** and **out-of-range**. These are both used in a similar way to **valid** and **invalid** in the previous section, but they handle a different scenario.

Validation of text types and contents are not the only validation to be added to the new specifications; the HTML 5 spec also brings with it number types and minimum/maximum ranges.

Create another dummy form, but this time, rather than a user name and email input box, let's define a number range.

```
<form action="#" method="post">
    <label for="nmbUserAge">User Age</label><br />
    <input type="number" id="nmbUserAge" name="nmbUserAge" min="16" maximum="100"
/><br /><br />
    <input type="submit" value="Submit" />
</form>
```

Code Listing 47a: Mock HTML form with a number input

Let's apply the following styles to our form:

```
input[type=number]:in-range
{
  background-color: yellow;
}

input[type=number]:out-of-range
{
  background-color: salmon;
}
```

Code Listing 47b: Styles for our number range form in Listing 47a

When we render the form, we'll initially see it show using the out-of-range color, due to the fact that it has no value in it.

If we then start entering values, we should see the color change depending on what value we enter, in range or out.

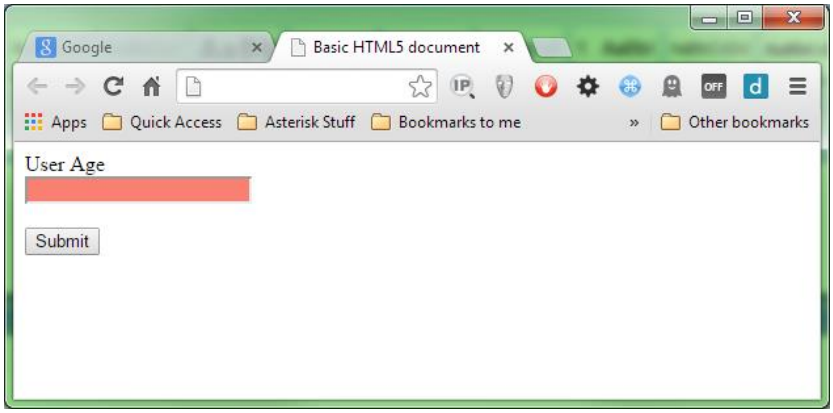


Figure 34: The number range form just after loading

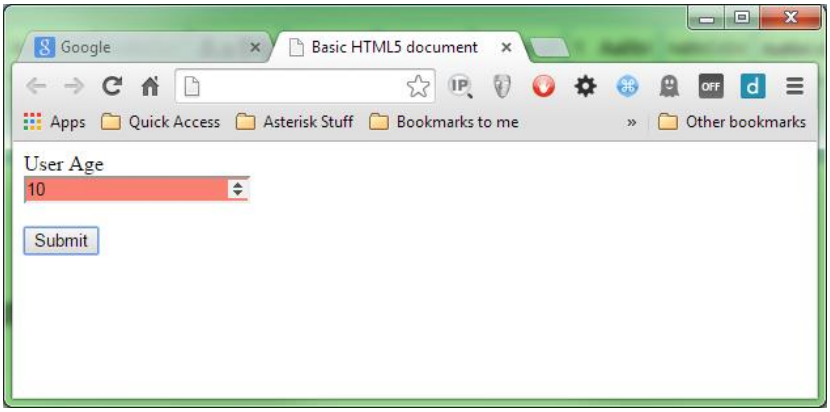


Figure 35: The number range form with an invalid value

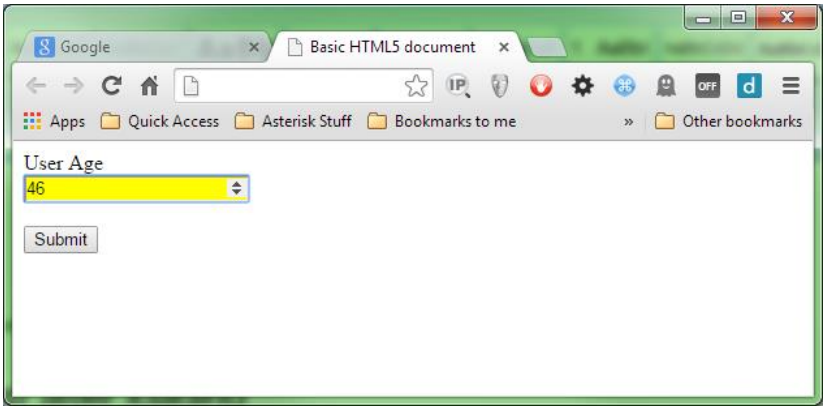


Figure 36: The number range form with a valid value

:empty

The **empty** pseudo selector might at first seem like it's used when an element value is empty, but it's not; it's used when the inner contents of an HTML element is empty. Consider the following HTML:

```
<div class="mydiv"></div>
```

Code Listing 48a: An empty div element

The **div** in Code Listing 48a is empty—it's devoid of any contents whatsoever between the opening and closing tags that surround it.

If we were now to apply a rule such as the following:

```
.mydiv:empty
{
  /* rules go here */
}
```

Code Listing 48b: CSS rule for empty pseudo selector

Any style rules defined in the CSS would be applied to it. However, as there are actually no contents, then applying rules that make color changes is pretty much redundant, unless the parent element already has a rule that shows itself with a fixed size and border.

The empty pseudo element comes in handy when you want to remove empty elements from display. For example, let's imagine you had a message count on the end of a tab in your user interface. If that count was 0 or had no value, then you wouldn't want an empty, red circle showing.

In this case, you'd style your red badge in a separate rule, and then you'd style your empty rule something like the following:

```
.mydiv:empty
{
  display: none;
}
```

Code Listing 48c: CSS rule to make an empty element vanish from display

By defining the **empty** selector using no display, all you need to do to make the count disappear from the display is to make sure there's nothing at all between the opening and closing tags.

You might notice that I'm wording things as "nothing at all," and that's deliberate. When we say that the **empty** selector matches an empty tag, we mean 100 percent empty. A space might look empty to us, but to the CSS selector engine, it's classed as content, as is splitting an empty tag over two lines.

This can get quite frustrating when you're trying to track down why an element you're sure should vanish from display, does indeed not vanish from the display. I know firsthand how much hair-pulling this causes.

If there is anything—anything at all—in between and opening and closing tag, then the empty selector will NOT match. It works best with spans and divs that form small containers whose contents are built and removed using JavaScript.

Because of the way this selector works visually, I'm not going to provide a screen shot. If you construct a simple **`div`** tag and apply the rule in Code Listing 48c to it, then manually add or remove contents using your editor, you'll soon see how it works.

:checked

The **`checked`** pseudo selector is used when dealing with checkbox and radio elements.

The premise is simple: if the element is checked or selected, then this pseudo rule will be applied; otherwise, it won't.

Let's alter our form, this time to show a couple of checkboxes and radio buttons, as shown in Code Listing 49:

```
<form action="#" method="post">
    <label>Check box 1<input type="checkbox" /></label><br />
    <label>Check box 2<input type="checkbox" /></label><br />
    <label>Check box 3<input type="checkbox" /></label><br />
    <label>Check box 4<input type="checkbox" /></label><br />
    <label>Radio Button 1<input type="radio" name="radios" /></label><br />
    <label>Radio Button 2<input type="radio" name="radios" /></label><br />
    <label>Radio Button 3<input type="radio" name="radios" /></label><br />
    <label>Radio Button 4<input type="radio" name="radios" /></label><br />
</form>
```

Code Listing 49: Simple form of check boxes and radio buttons

Rendering this should give us a fairly straightforward bunch of controls.

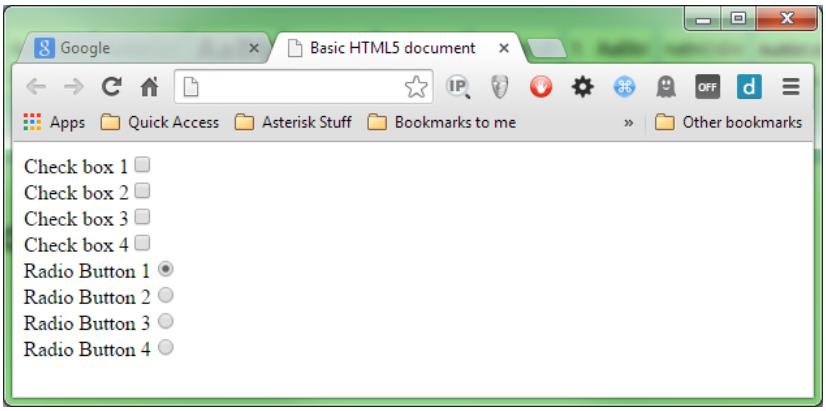


Figure 37: Form showing simple checkbox and radio controls

Since checkboxes and radio buttons don't change color, we'll use a slightly different tactic for this demonstration, and change the size of the controls. Add the following rules to your style sheet:

```
input[type="checkbox"], input[type="radio"]
{
  width: 20px;
  height: 20px;
}
input[type="checkbox"]:checked, input[type="radio"]:checked
{
  width: 25px;
  height: 25px;
}
```

Code Listing 50: CSS Styles do demonstrate using the checked pseudo selector

If you render the form now, and check and uncheck some of the controls, you should see a small size variation on the items that are checked.

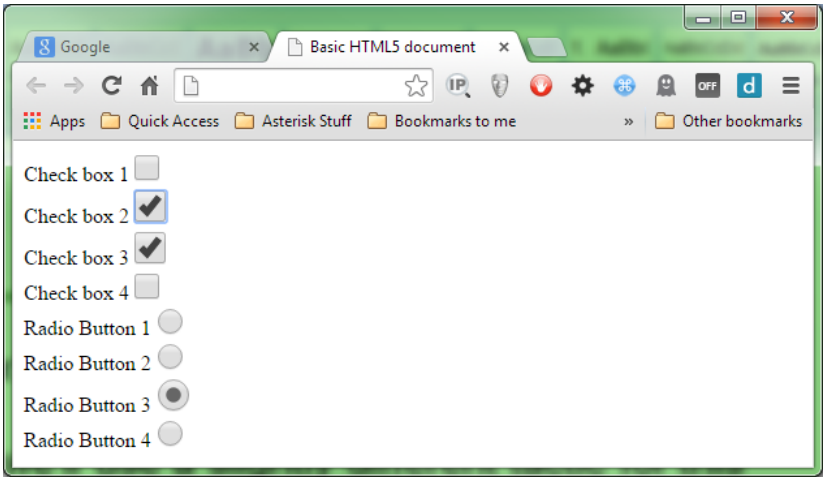


Figure 38: Our check boxes and radio buttons changing size

You might be wondering why you might want to use this pseudo selector if it's difficult to change the properties such as color on a checkbox.

It's not until you start to think about combining it with adjacency selectors that it starts to make sense as to how you might want to use this. Create a line in your form in Code Listing 49 that looks like the following:

```
<input type="checkbox" /><span class="checkboxlabel">This is my check box</span>
```

Code Listing 51a: Line to add to previous form

Then, add the following CSS rule:

```
input[type="checkbox"]:checked + span.checkboxlabel
{
  color: red;
}
```

Code Listing 51b: Rule to style the line in Code Listing 51a

You should find that when you render the document, checking or unchecking the checkbox will make no changes to the checkbox itself, but it will change the text color of the span following it.

It does this because we've told the CSS engine that for any input element that has a type attribute equal to **`checkbox`**, look at the adjacent element and check to see if it's a span, and has a class of **`checkboxlabel`**. If it does, then change the color of that span element to red.

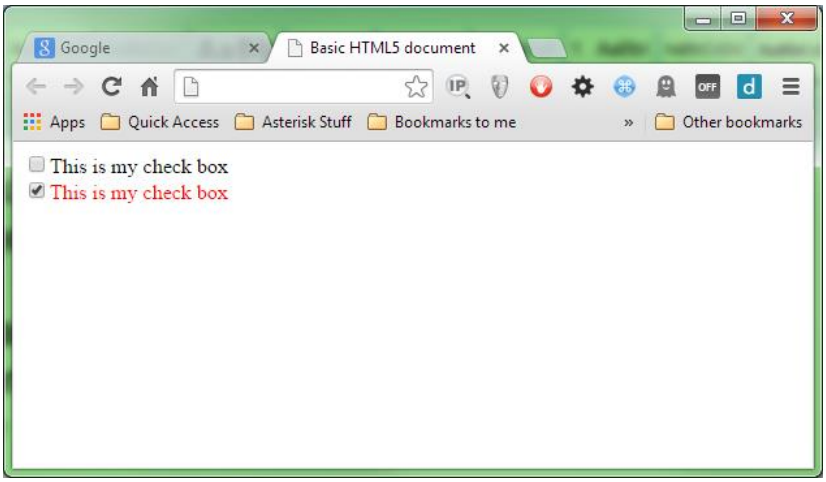


Figure 39: Check box using adjacent selectors to style the label following it

Using checked selectors like this is the method used by UI toolkits such as Foundation and Bootstrap to produce toggle buttons and custom radio and check groups made from lists of icons.

If you're looking to create custom check boxes, then you simply need to combine this pattern with pseudo elements such as before and after to add/remove content dynamically in your markup.

:before and :after

Before CSS3 came along, if you wanted to add or remove elements and content dynamically, you had to resort to JavaScript, or make the changes server side and redraw the document fragment using something like an Ajax call.

Note: Ajax is not something we'll be covering in this book, as it's deeply tied in with JavaScript and is a quite complex subject. In general, however, an Ajax call is the process of asking JavaScript to make a partial request to the web server in order to get

only a small bit of content. It's typically used as a means to get small chunks of data and insert them into a webpage as needed, rather than downloading the entire page in one request.

With the **`before`** and **`after`** pseudo selectors, we can add and remove content in our HTML elements without ever touching a single line of code, and have it all driven using pure CSS rules.

Starting with a simple example, let's imagine we have the following line in our HTML markup:

```
<span class="bracketedLabel">This is a bracketed Label</span>
```

Code Listing 52a: Simple span to demonstrate before and after selectors

There's nothing at all special about this label; if we render it, we just get a simple output.

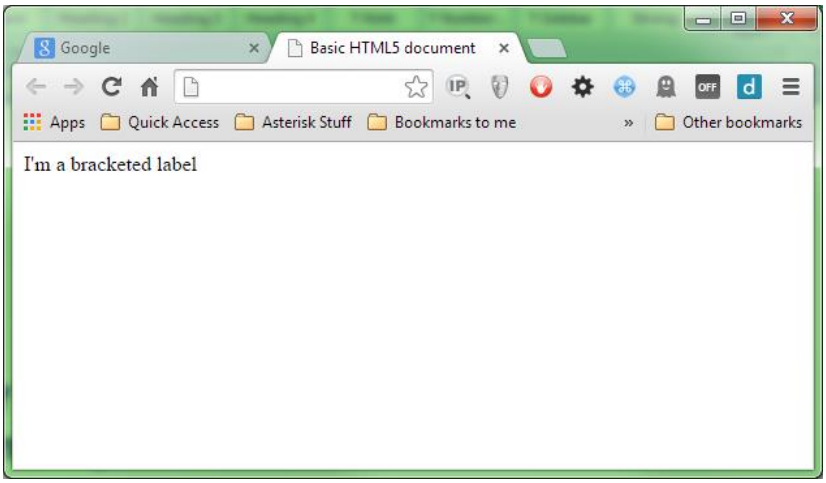


Figure 40: Output from Code Listing 52a

As the name suggests, we'd like any label with this class to have square brackets surrounding it, but we don't want to go and find all occurrences to add them manually.

So, we can add the following CSS rules:

```
.bracketedLabel
{
  color: green;
}
.bracketedLabel:before
{
  content: '[';
  color: red;
}
.bracketedLabel:after
{
  content: ']';
```

```
  color: red;
}
```

Code Listing 52b: CSS rules required to make our label a bracketed one

Then, when we refresh the browser output, we get the following:

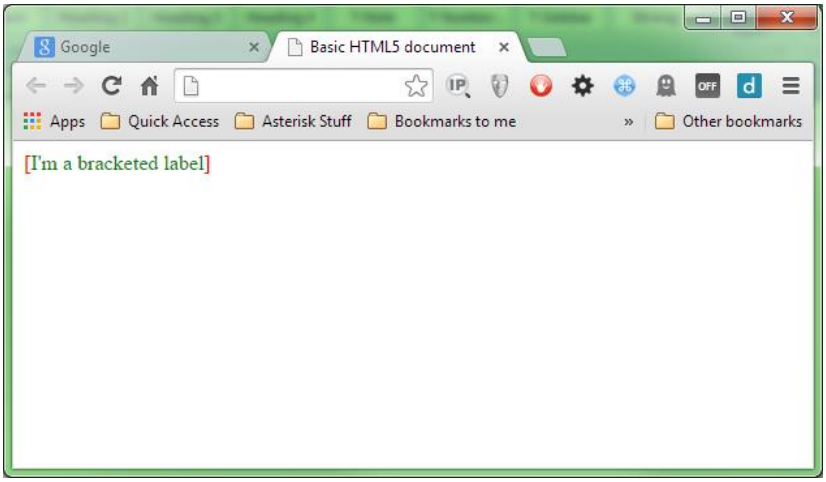


Figure 41: Our bracketed label now has brackets

Before you all start rushing to get the before and after selectors to add and remove HTML tags in your document, I have to tell you that unfortunately, you can't.

While the before and after pseudo selectors are powerful, you're limited to plain text only. If you try to put HTML tags in the rules, then those tags will simply be rendered as is, complete with the tag itself as a text string.

There are a number of reasons for this. Imagine you had an element, that via an CSS rule added an element, which also added an element; if you weren't careful, you could very easily end up in an infinite loop and everything would halt.

CSS rules are also designed to be parsed in one run, so any element that inserted elements that require styling would mean having to stop, go back and reprocess the rules already seen, then come back and continue.

You can, however, still perform some very interesting tricks with them.

One of the better uses is the creation of customized check boxes using nothing more than CSS3 properties and selectors. To illustrate this, create a simple form with two checkboxes in it.

```
<form action="#" method="post">
    <input id="chkCheckboxOne" type="checkbox">
    <label class="checkbox" for="chkCheckboxOne"> Checkbox 1 </label><br />
    <input id="chkCheckboxTwo" type="checkbox">
    <label class="checkbox" for="chkCheckboxTwo"> Checkbox 2 </label>
</form>
```

Code Listing 53: check boxes and radios ready for customization

Like our previous checkbox example, rendering this will just give us our plain form style:

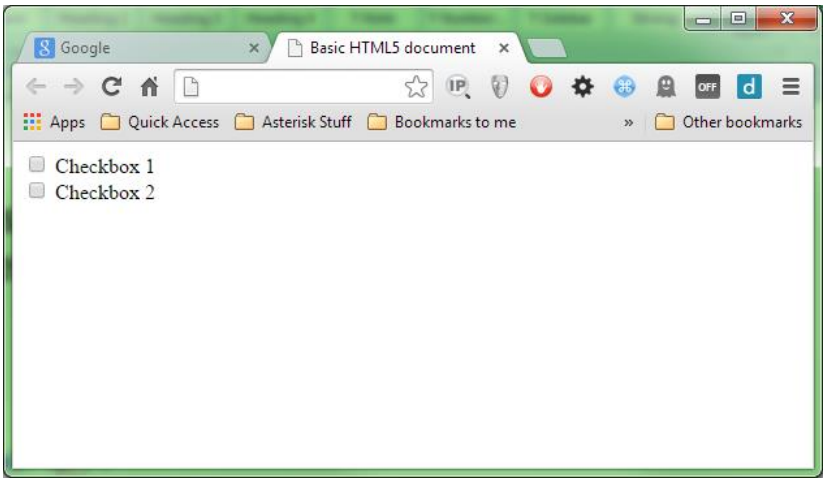


Figure 42: Just another simple form

Let's add the first of our rules, and give the checkboxes some default styling.

```
.checkbox
{
  display: inline-block;
  cursor: pointer;
  font-size: 20px;
  font-family: sans-serif;
}

input[type=checkbox]
{
  display:none;
}
```

Code Listing 54a: Default styles for our checkboxes and radio buttons

If you render the page at this point, you might be surprised to see that we've hidden the actual controls, and just styled our label text. We've done this because we don't want to see the original controls, and since we can't really customize them with colors and such anyway, we just want them to disappear so we can draw our own new ones.

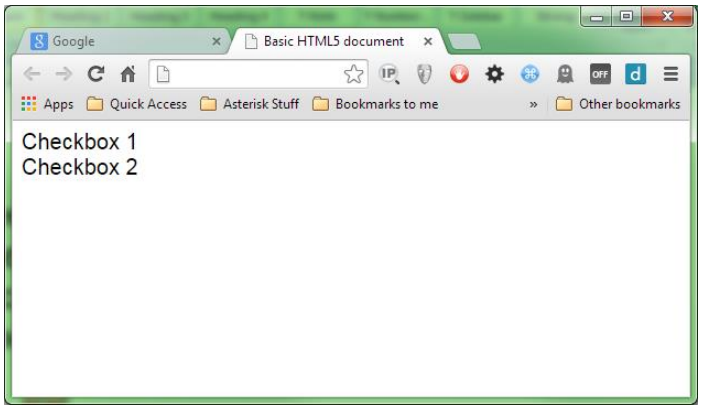


Figure 43: Our checkboxes are still there, just invisible

The next rule we add is the **before** part of the style sheet, but notice we're targeting the label and not the actual checkbox element, as it's really the label that's the driver for all of this.

```
.checkbox:before
{
  content: "";
  display: inline-block;
  width: 18px;
  height: 18px;
  vertical-align: middle;
  background-color: green;
  color: #f3f3f3;
  text-align: center;
  box-shadow: inset 0px 2px 3px 0px rgba(0, 0, 0, .3), 0px 1px 0px 0px rgba(255,
255, 255, .8);
  border-radius: 3px;
}
```

Code Listing 54b: The before part of the rules, which draws the unchecked version of the control

Rendering this, we can see that we now have something that looks a little more like checkboxes now.

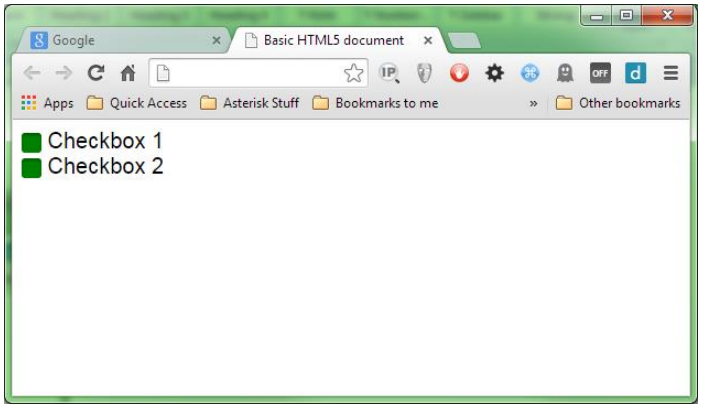


Figure 44: We have green check boxes

We're still not quite finished; we need to add one more rule, which will ensure that we get a good-looking check symbol that appears when the control is in a checked or selected state.

```
input[type=checkbox]:checked + .checkbox:before
{
  content: "\2713";
  text-shadow: 1px 1px 1px rgba(0, 0, 0, .2);
  font-size: 15px;
}
```

Code Listing 54c: The final rule required to finish off our custom checkboxes

When we render that, we should find we can toggle the check boxes exactly like we used to with the default ones.

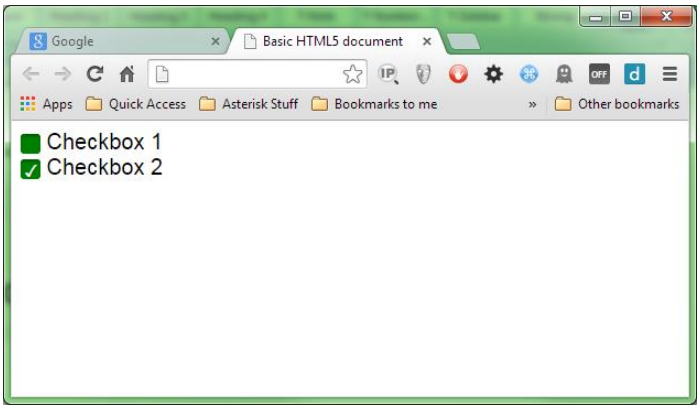


Figure 45: The finished custom checkboxes

This technique can be used in many places; Bootstrap, for example, uses it to position icons in input fields in the Bootstrap UI framework.

Figure 46: Twitter Bootstrap uses the same method for validation icons

Before and **after** are probably two of the most flexible pseudo selectors available, and the development community still hasn't discovered all the ways they can be used.

First-X, Last-X, and nth-X

I've chosen to represent this last group as one collection as they are all related. The first, last and nth pseudo selectors are used for selecting the first, last and user-defined count of elements in a list of elements.

There are two distinct groups in this batch; we'll look at the general selectors (that is, the ones that will select anything), then the type specific selectors. These two groups are listed in the CSS3 spec as:

first-child, **last-child**, **nth-child()**

and

first-of-type, **last-of-type**, **nth-of-type()**

For the first set of examples, we'll use the following HTML markup to illustrate how the selectors work, and what they can do.

```
<ul>
    <li>List Item One</li>
    <li>List Item Two</li>
    <li>List Item Three</li>
    <li>List Item Four</li>
    <li>List Item Five</li>
    <li>List Item Six</li>
    <li>List Item Seven</li>
    <li>List Item Eight</li>
    <li>List Item Nine</li>
    <li>List Item Ten</li>
</ul>
```

Code Listing 55: Unordered list markup for first, last and nth selectors

As we've done with everything else thus far, here's the before rendering of our markup in the browser:

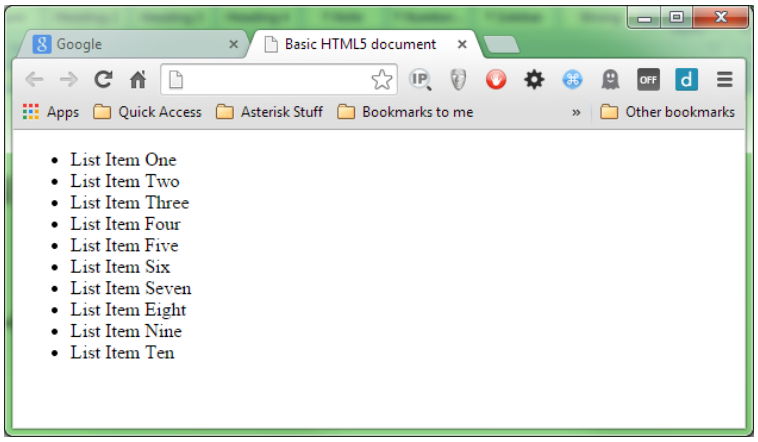


Figure 47: Un-styled list from Code Listing 55

Like many of the selectors available, their names describe without any doubt what they do. Let's apply the following rules to the HTML in Code Listing 55.

```
ul > li:first-child
{
  color: red;
}

ul > li:last-child
{
  color: green;
}
```

Code Listing 56: CSS rules to demo 'first-child' and 'last-child' pseudo selectors

As you might predict, the first and last items in our unordered list change color as anticipated.

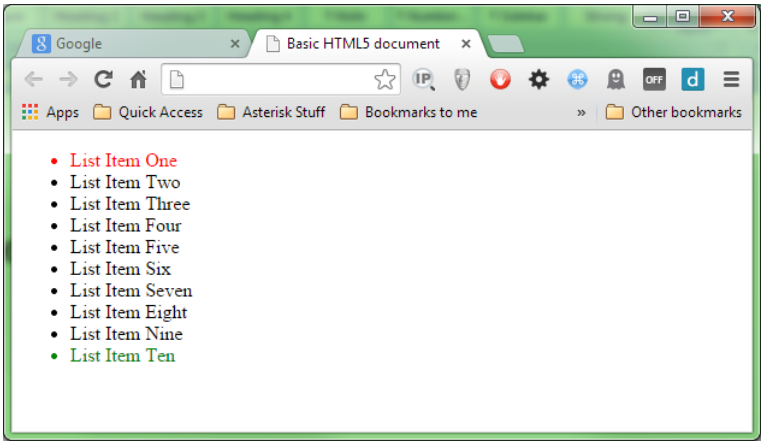


Figure 48: Our list style using 'first-child' and 'last-child'

The first-child and last-child selectors are about as simple as we get; nth-child(), however, requires a little more explanation.

You may notice that unlike the others, nth-child has a set of parentheses following it. This is because unlike the other selectors, nth-child accepts a parameter to control it.

This parameter takes the form of an algebraic equation that describes what to select and how.

Note: For many folks, doing algebra at school is a distant memory. For those who do remember it, they'd probably much rather they didn't. However, even though you might have wondered then when it might ever come in handy, remembering how it works will really help you with algebraic expressions in CSS3. If you are struggling to remember any of it, then now might be a good time to have a quick refresher.

Don't be frightened; an algebraic equation is something that describes how a number calculation should work, but not necessarily what the actual answer should be.

Many folks will have come across something at some stage in their lives that looks like this:
1n+5x=6y

To those who remember, this is called algebra, and is a quite specialist branch of mathematics that many folks study for years on end. Solving these strange equations is somewhat of a hobby for many theoretical mathematicians, and some even devote years of their lives to simply theorizing about these things.

For those who are still scratching their heads, that little bit of algebra means the following:

The equation 1 multiplied by the value of n when added to 5 multiplied by the value of x will be always equal to 6 multiplied by the value of y, where n, x, and y are theoretical instances that hold no current value.

Don't worry, you don't need to know algebra in that kind of depth to understand how an nth-child works, but you do need to know how algebra works.

When you have a discrete value followed by a theoretical instance, that theoretical instance is usually multiplied in some manner by the operator. Operators can also apply additions and subtractions, or just be specified on their own to give a static value.

For example:

1 means just that: 1, an offset of 1, a count of 1, one page, one thing… it really doesn't matter, because it's theoretical.

1n, on the other hand, means 1 amount of **n**, or 1 multiplication of **n**; again, **n** can represent anything because it's all theoretical.

1n+1 means 1 amount of n with an additional 1 added to it.

Perhaps an easier way to understand how these equations work, is to imagine using them to perform multiplication tables. Remember, for example, the 3 multiplication table:

3 x 0 = 0

3 x 1 = 3

3 x 2 = 6

3 x 3 = 9

3 x 4 = 12

And so on…

Now, imagine that the **n** in our **3n** is the multiplicand in the second column (the value 0 to 4); you then start to see how n references any value in the range.

If you then treat the result as a static number, you end up with the following static sequence.

0, 3, 6, 9, 12

Let that sink in for a moment.

Now think back to the list of HTML elements in Code Listing 55. There are 10 list item children in that list, and if we forget about the numbers 0 and 12 in our static list, we have 3, 6 and 9, which could all be offsets to select (0 and 12 are not valid items, as they don't exist).

Well, that's exactly what nth-child() does. It uses the algebraic equation passed to it to determine exactly which elements in a list it needs to select.

Let's apply the following CSS rule to our list from Code Listing 55:

```
ul > li:nth-child(3n)
{
  color: red;
}
```

Code Listing 57a: nth-child CSS rule to select every third element

What will happen is that every third element in the list will be affected by the rule.

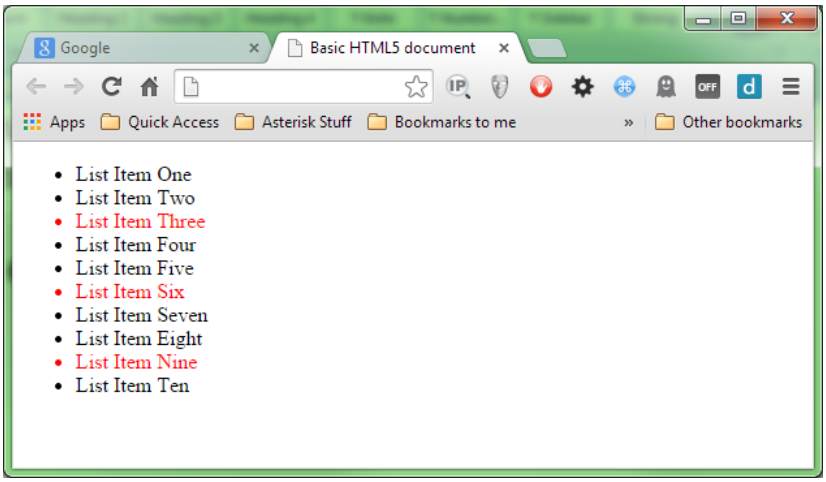


Figure 49: Our nth-child(3n) rule in action

If we start adding a **+** offset to that, we dictate where the multiplication starts:

```
ul > li:nth-child(3n+1)
{
  color: red;
}
```

Code Listing 57b: nth-child CSS rule to select every third element starting at element 1

This will produce the following:

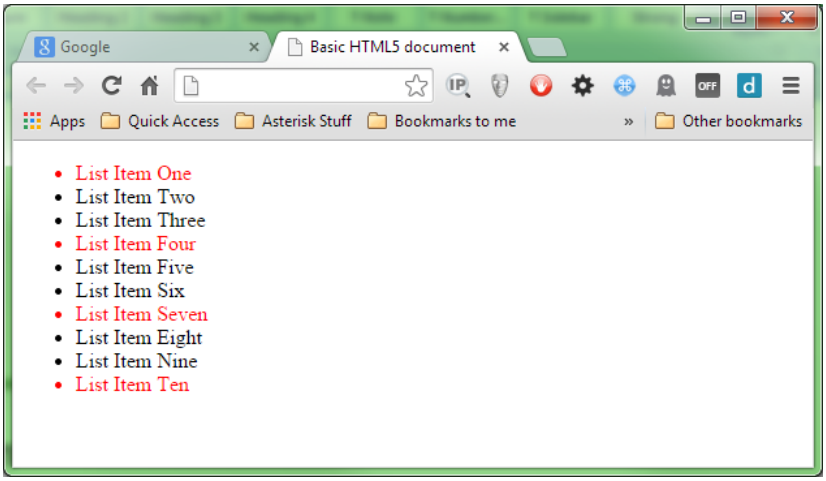


Figure 50: nth-child(3n+1) starting at 1 and selecting every third element

Which, as you can see, starts with element index number 1, then applies the multiplication of 3 each time, to select every third item.

You can use just an **n** on its own too, which, when used with a positive offset, allows you to select ranges. For example:

```
ul > li:nth-child(n+7)
{
  color: red;
}
```

Code Listing 57c: nth-child CSS rule to select everything from 7 onwards

Will start at offset 7 and simply select everything left from that point on.

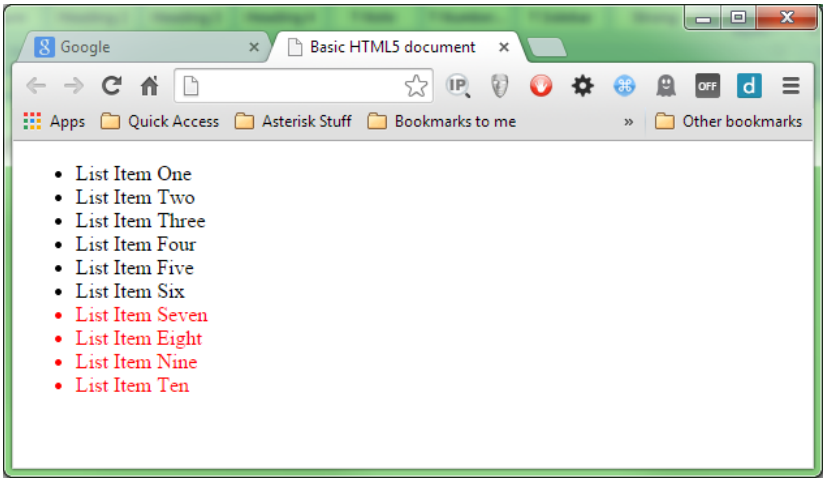


Figure 51: nth child selecting elements from 7 onwards

You can also go backwards by simply prefixing the **n** with a minus symbol.

```
ul > li:nth-child(-n+7)
{
  color: red;
}
```

Code Listing 57d: nth-child CSS rule to select everything from 7 backwards

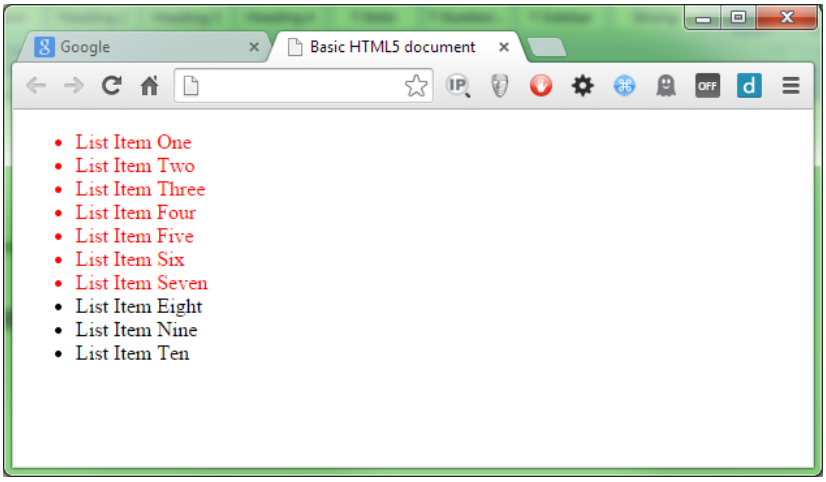


Figure 52: nth child selecting from 7 backwards

If you're just trying to select a single numbered element in a list, you can do the following:

```
ul > li:nth-child(5)
{
  color: red;
}
```

Code Listing 57e: nth-child CSS rule to select just index number 5

Just specifying the number index directly will select only that numbered element.

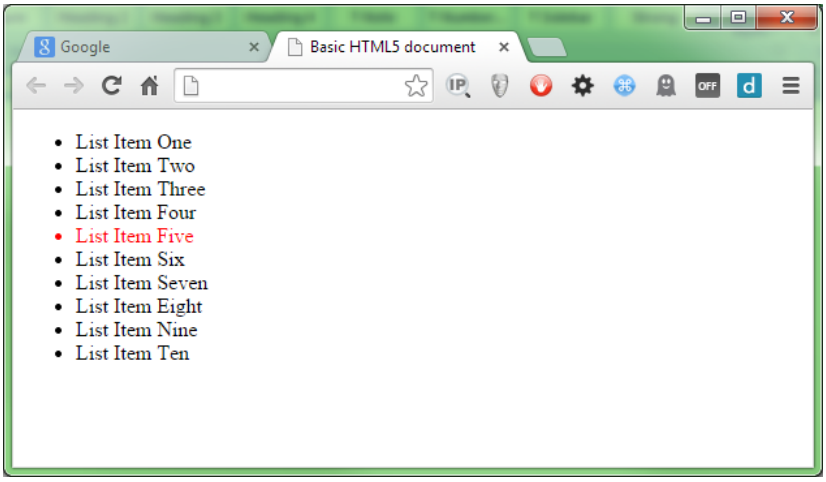


Figure 53: nth-child selecting just the fifth element in the list

I could easily fill up the rest of this book with different combinations of nth-child equations, but I think we've got enough for now.

I can take a guess at the question that most of you are asking yourselves at this point.

"How might I use this to select and stripe all the rows in a table or grid? Can you maybe tell me what the equation for that is?"

The good news here is that the W3C standards committee realized that this was likely to be one of the most common uses of the nth-child pseudo selector, so they added a couple of shortcuts: **even** to select every even numbered element, and **odd** to select every odd numbered element.

If you add the following rule:

```
ul > li:nth-child(odd)
{
  color: red;
}
```

Code Listing 57f: nth-child CSS rule to select every odd numbered element

You should see that every odd list item in the markup gets selected.

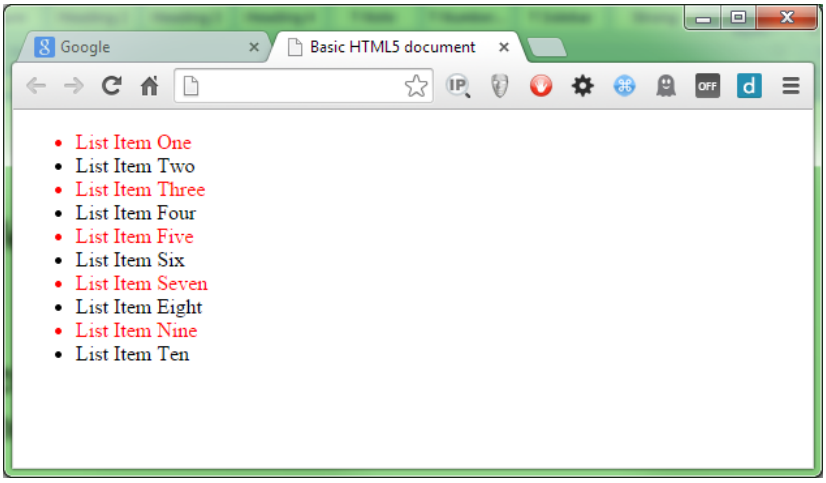


Figure 54: nth-child selecting every odd element

I think by this point you'll likely want to give your brain a rest, so we'll leave the algebra behind for now.

Take note though, that there is still a massive amount we've not covered. For example, you can chain nth-child selectors (just as you can with many others), leading to some crazy selector rules that look like this:

nth-child(3n+1):nth-child(even)

This will select every third element, starting at index 1, but only those that have an even number for the index. For 11 elements, this would select only elements with an index of 4 and 10 (all the others are odd numbers).

You can easily select ranges, then split those ranges up. You can combine those range selections with **`first-child`** and **`last-child`** too, allowing you only to select the start and end of a range. It's a subject in itself that perhaps warrants an entire book, and to date there have been some very clever uses of the **`nth-child`** pseudo selector; for us though, it's time to move on.

The second group of pseudo selectors that fall under the **`first`**, **`last`** and **`nth`** element selections are the **`of-type`** selectors.

The **`first-of-type`**, **`last-of-type`**, and **`nth-of-type`** selectors all work exactly the same way as the general selectors described previously in this chapter, but they target a specific element type.

Let's take our unordered list from Code Listing 55, and change it a bit to turn it into a definition list.

```
<dl>
    <dt>HTML 5</dt>
    <dd>HTML 5 is the markup language used to create documents that are published
on the internet</dd>
```

```
    <dt>CSS3</dt>
    <dd>CSS3 is the markup language used to add style and colors to documents
created using HTML 5</dd>
    <dt>HTTP (Hyper Text Transfer protocol)</dt>
    <dd>HTTP is the mechanism by which HTML 5 documents and their associated
resources are transported on the internet</dd>
    <dt>WWW (World Wide Web)</dt>
    <dd>The World Wide Web is the collective name for the many hundreds of
thousands of HTML 5 documents published via the internet</dd>
    <dt>The Internet</dt>
    <dd>The internet is a massive global computer network, which amongst other
things holds the World Wide Web</dd>
</dl>
```

Code Listing 58: Definition list markup for first, last and nth selectors

A definition list is slightly different to the other types of lists. Rather than just list item elements, there are in fact two different types of child elements present within it: **`dt`** elements or "Definition Titles," and **`dd`** elements, or "Definition, definitions."

If we render Code Listing 58, we can see the default styling for both of these:

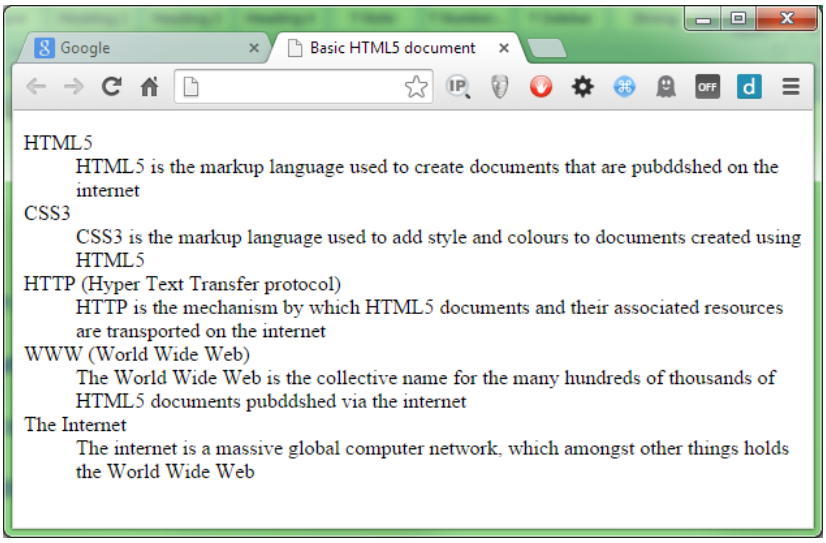


Figure 55: Definition list with default styling

If we try to apply one of our previous rules to this list (modified slightly to account for the different tag types), for example, the CSS rule to select every third child (Code Listing 57a), we'll see that it doesn't quite do what we might expect it to do.

```
dl > :nth-child(3n)
{
  color: red;
}
```

Code Listing 59a: Modified nth-child rule for our definition list

It tries to select every third child element, and even though we can target things specifically by name, we may have to deal with wildly varying list contents.

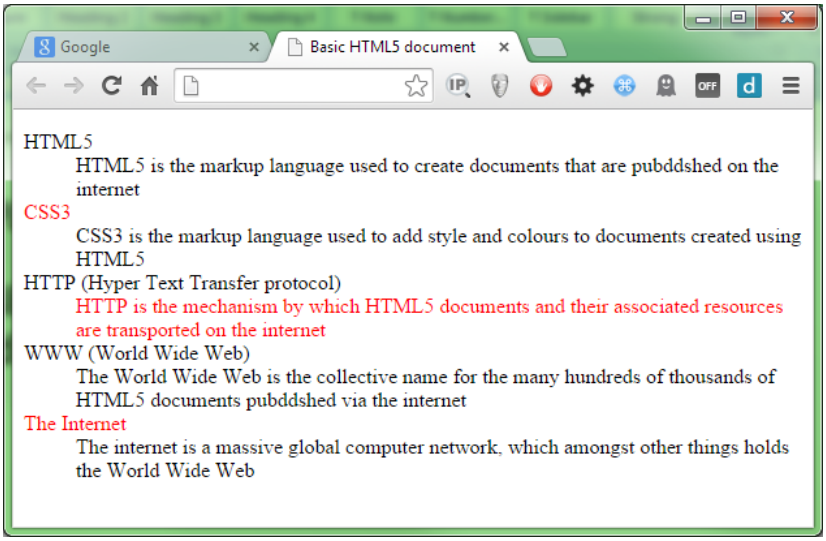


Figure 56: nth-child gets our selection wrong.

However, if we use **`nth-of-type()`**:

```
dt:nth-of-type(2n)
{
	color: red;
}
```

Code Listing 59b: 'nth-of-type' CSS rule to be used instead of 'nth-child'

We quickly see that it effects only the elements we ask it to.

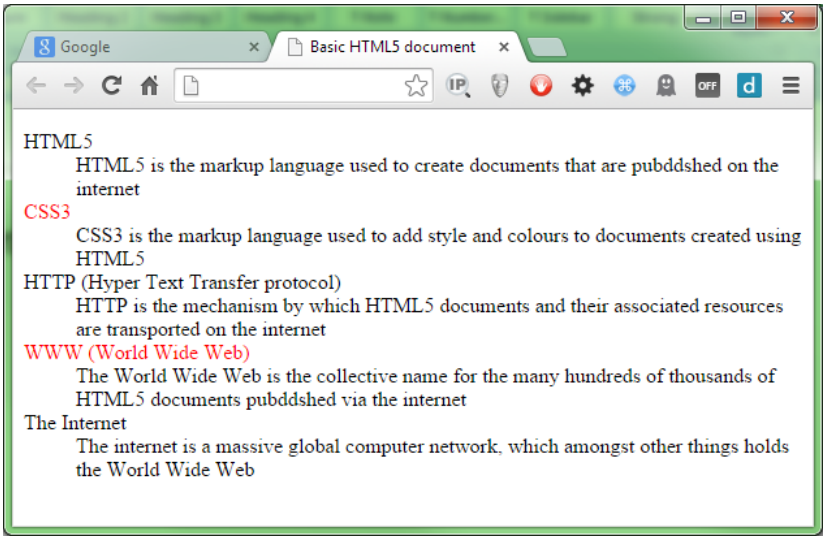


Figure 57: nth-of-type gets our selection right

The **nth-of-type** (and for that matter **first-of-type** and **last-of-type**) only considers targeted elements as specified to be candidates for selection.

In a previous example we saw some markup that might be used for a news page; if you were to use **nth-of-type** on that, you'd get the flexibility of selecting things as you would with **nth-child**, but you would be able to tell the CSS selector engine to ONLY apply that to paragraph elements found in the article.

You could, for example, easily style all the even numbered paragraphs a different color to the odd numbered ones, no matter how many paragraph elements were in each section or how many other elements such as headers might be present.

There are still more pseudo selectors to be covered, and still more that can be done by chaining them all together; however, the potential is far beyond anything we have space here to describe.

Remember too, that generic rules always come before specific rules.

If you define:

```
p:nth-child(odd)
{
  color: blue;
}

p
{
  color: green;
}
```

Code Listing 60: Specific CSS rule first

You'll find that all your paragraph elements will be green. This is because the later rule targets more that the former rule, so any changes that the former rule applies will be over-written by the latter one.

Reversing the order of the two rules will fix this, so the generic one changes everything green first, then the specific one targets odd-numbered elements and changes them from green to blue.

Note: At the time of writing, and with the version of Chrome (and other browsers) being used, the previous description held true in that all <p> elements would indeed have turned green. However, two months later, an update has been made to the specifications, which now means that the nth-child rule has specifity precedence over the standard element selector in the example. This means that the previous example now produces exactly the same result, no matter which order they are in. This does not mean that you should suddenly ignore rule order; there are still plenty of places where this is obeyed. In this, case however, it's a perfect example of the fact that many of these features are still in a state of flux, so as always, choose wisely when deciding which features to use and why.

If you have any problems with styles not being applied when and where you expect them to, the first thing to always check is the order of specificity.

Summary

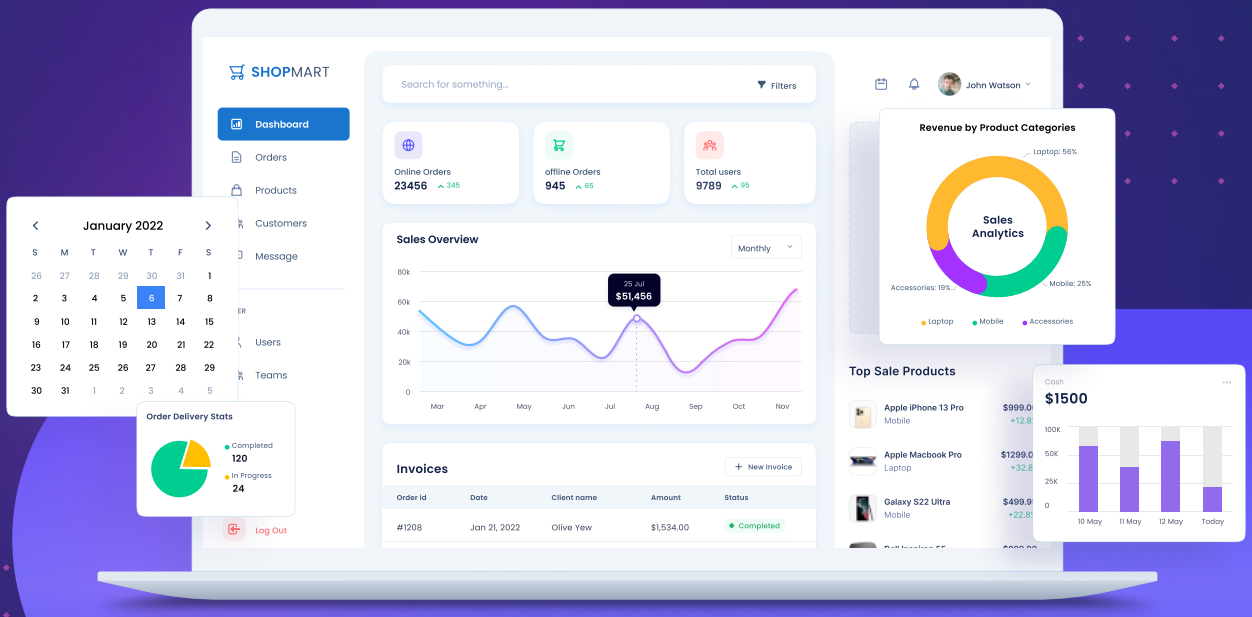
In this chapter we've taken quite a whirlwind (and brain-exercising) tour of the new pseudo selectors available in CSS3. You should by now have a very good idea of how to target and select the elements you need to select, and the methods by which you can do so.

We've learned a huge amount in this chapter, and learned a few clever tricks along the way. Now that the hard stuff is behind us, it’s time to sit back and enjoy the easy life.

Starting in the next chapter, we'll be taking a look at what most of you have been waiting for: the eye candy. We'll start by looking at everyone's favorites: rounded corners, drop shadows, and gradient fills.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 5 Eye Candy

One thing that CSS3 has in abundance are ways to make things look pretty.

For developers (especially those who spend most of their time working server side), they might be wondering what all the fuss is about; we're hired to produce software, and that's exactly what we're doing.

Unfortunately, on the client side and in the browser, things don't quite work that way. Those of our colleagues that don't understand what we do or how we do it love to see things that are flashy, colorful, and sparkly.

When they can see this type of progress, they gauge that visual feedback as proof that there is work being done on the application.

Chief among the things that the non-IT crowd likes to see are the staple eye candy diet of drop shadows and rounded corners.

I'm sure if you've done any serious CSS or design work in the past, you'll be more than familiar with the difficulty of having to use Photoshop to create partially-transparent PNG graphic files, which you then had to line up around the edge of a carefully placed **`div`** element.

If you were off by even so much as a pixel, you'd often have to do it all again from scratch, measuring things one pixel at a time and restricting page layouts to fixed sizes so things didn't break afterwards.

If you're new to this game, then find any veteran of the web development industry and I'm sure that they'll have more than a few horror stories to tell you.

CSS3 solves all of this by adding the ability to define rules that state how rounded a corner should be now, and by allowing you to describe how a drop shadow should look. The CSS3 engine will then apply those instructions automatically to your HTML markup based on the selector rules you provide and workout where everything needs to be.

Rounded Corners

The web is square—or at least it was until very recently.

Until CSS3 came along, all we had were boxes with sharp edges. **`Div`** elements were always square and parallel, tables were square and parallel, and we couldn't make things look a little softer and friendlier (well at least not without a lot of image hacks).

To change this, CSS3 introduced the **`border-radius`** property on all block elements that can use a border.

Using it is simple.

Given the following HTML markup:

```
<div class="niceDiv">
    <p>This is a nice div with no sharp corners</p>
</div>
```

Code Listing 61a: HTML markup to demo border radius

And this CSS rule:

```
.niceDiv
{
  width: 100px;
  height: 100px;
  background-color: beige;7
  padding: 4px;
  border: 1px solid black;
  border-radius: 5px;
}
```

Code Listing 61b: CSS rule to give rounded corners to the HTML in 51a

We get a very nice looking **`div`** element rendered in our browser:

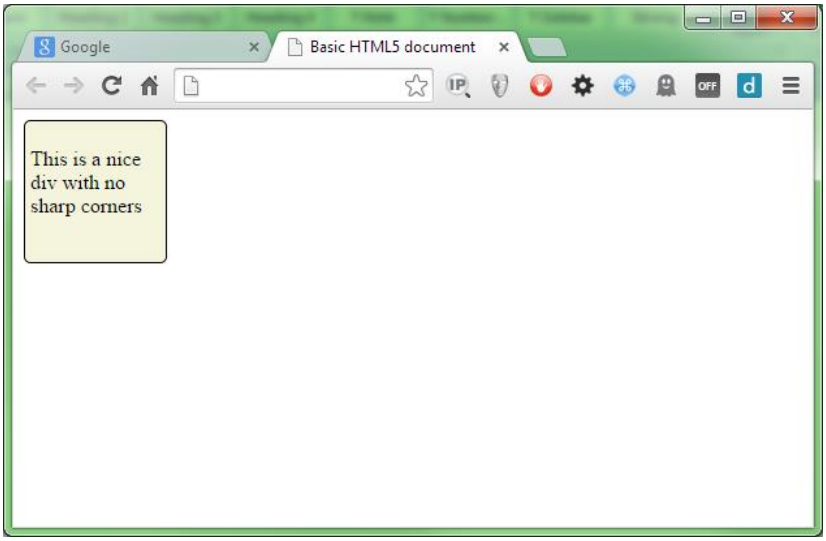


Figure 58: Div with rounded corners

In the rule in Code Listing 61b, we've used the general default for border radius and just specified one value. When you define the rule this way, the same value is used for all four corners, so as you can see in Figure 58, each corner looks the same.

The value specified is how steep the angle of the curve will be; in our case, we've used 5 pixels. If you change it to 25 pixels, you should get something like Figure 59:

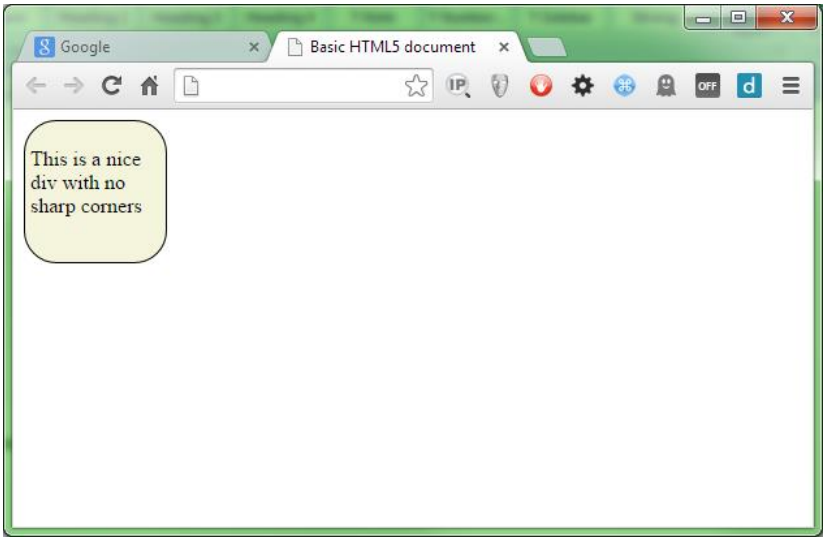


Figure 59: Div with 25 pixel rounded corners

Because each corner is a quarter, and a quarter of a circle is 90 degrees, if we change the value to 90 pixels, we get a perfectly round circle.

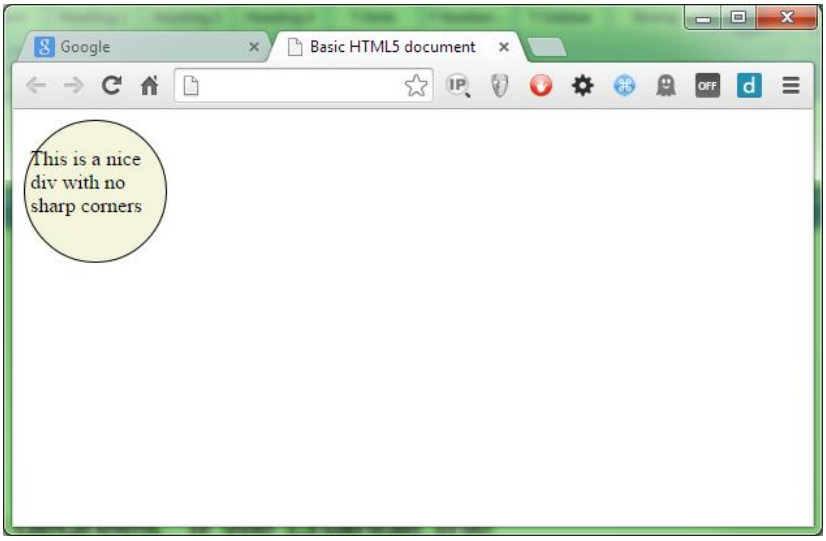


Figure 60: Div with 90-pixel, rounded corners

We've used the one value format in the demos above, but if you wish, you can also specify all four values (one for each corner), change the **`border-radius`** line in Code Listing 61b so that it looks like the following:

```
border-radius: 5px 10px 15px 20px;
```

Code Listing 61c: Altered border radius line for Code Listing 61b

When rendered, you should get a **`div`** with four different-sized corners:

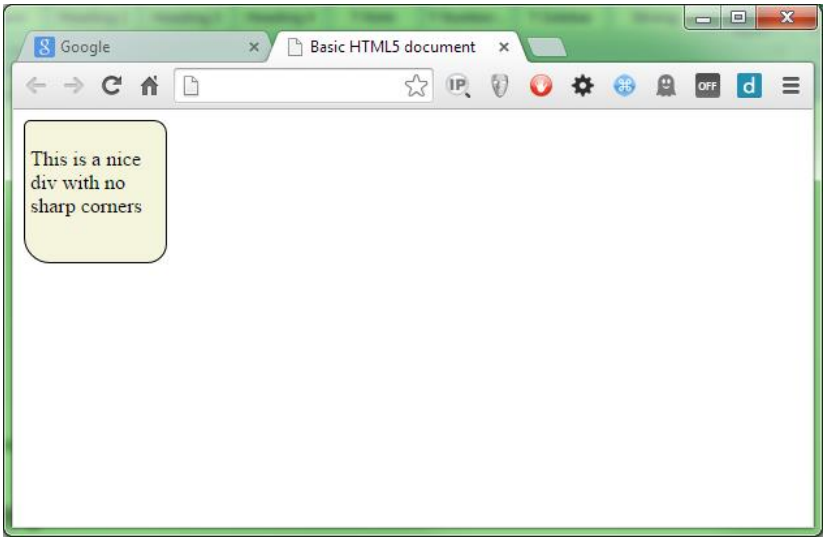


Figure 61: Div with four different size corners

The first value is the top-left corner, followed by the top-right corner, then the bottom-right corner, and finally the bottom-left corner.

If you wish to, you can also use four different rules and write out the size specifications as follows:

```
border-top-left-radius: 5px;
border-top-right-radius: 10px;
border-bottom-right-radius: 15px;
border-bottom-left-radius: 20px;
```

Code Listing 61d: The changes from Code Listing 61c written out

How you choose to do it is entirely up to you; some people like the verbosity of doing things the long way, and to be honest, I find that during development it helps maintain the code. However, when you ship to production, you really want to make these things as small as possible.

A good CSS squisher is a must for optimizing your style files, especially if you do use the long form during development.

Note: Minifying CSS—Minifying is the process of making a file smaller than it is without losing any of its original intent. Unlike a compressed or zip file, a minified file still operates in the manner that it was originally intended—it's just become a lot smaller and more compact. Having a good minifier to squash your files down is a must for web development. One of the biggest speed-ups you can have when your site is being loaded, is for your styles, code, markup, and images to be as small as possible and as few as possible.

There a plethora of tools available to do help you do this. A good place to start is with the Grunt Task Runner. Grunt has a large number of modules for squashing CSS, JavaScript,

images, and even HTML, allowing you to shave off precious bytes everywhere you can. Knowing how to use these tools is a very important skill for any web developer.

You don't just have to give one angle for each corner either. If you do, then the same angle will be applied on both sides, giving you a uniform and equal circle-based curve.

If you specify two values on each corner, you then get to use an ellipse to shape the border's curve radius. Doing this gives you some interesting possibilities.

Change your border radius rules to read as follows:

```
border-top-left-radius: 50px 15px;
border-top-right-radius: 50px 15px;
border-bottom-right-radius: 15px 50px;
border-bottom-left-radius: 15px 50px;
```

Code Listing 61e: Border Radius rules extended to show ellipse-based curves

You should end up with something that looks like Figure 62:

Figure 62: Our div with ellipse-based corners

Drop Shadows

Creating drop shadows is another easy bit of eye candy to master, and it can produce some interesting effects.

Drop shadows are created by using the box-shadow rule, which takes four integer measurements and a color value.

Of the four measurements that are provided, only the first two are required; the remaining two will default to browser defaults if left out.

The first two measurements are the offset x and offset y values, which as the names suggest, set the distance from the element that the shadow protrudes. If positive values are supplied, then the shadow protrudes from the bottom right; if negative values are used, the shadow protrudes from the top left. To get the shadow on other corners, you mix positive and negative offsets as required.

Create an HTML 5 document and add the following markup (or reuse the code from the previous section on border radius, but repeat it four times):

```
<div class="niceDiv">
    <p>This is a nice div with drop shadows</p>
</div>
<div class="niceDiv">
    <p>This is a nice div with drop shadows</p>
</div>
<div class="niceDiv">
    <p>This is a nice div with drop shadows</p>
</div>
<div class="niceDiv">
    <p>This is a nice div with drop shadows</p>
</div>
```

Code Listing 62a: HTML markup to demo drop shadow

Create the following style rule as a basis for the rest of this section:

```
.niceDiv
{
  width: 100px;
  height: 100px;
  background-color: beige;
  padding: 4px;
  border: 1px solid black;
  display: inline-block;
  margin: 15px;
}
```

Code Listing 62b: CSS rule to style the HTML in 62a

This will give us the default beige div you saw in the last section, with no additional styling.

Note: You've seen many instances so far where rules cascade down from less-specific rules above them. This is one of CSS3's (and previous versions) greatest features, which I've not gone into too much detail about. In the previous section we added the border radius directly too the 'niceDiv' rule; in this section I'm going to start using cascading

more to keep the base styles (in this case our beige rectangle) separate from the rules we're currently exploring. You can use the same selector as many times as you like, and the effect will always be cumulative. For example, if you specify '.niceDiv' and '.niceDiv .shadow', you then always have the option to use the base styling by specifying a class of 'niceDiv' to the element, and optionally applying the shadow by adding that to the class list as a secondary ID. There's nothing wrong with breaking things up into groups by repeating class and ID selectors if you wish, but if you do this, make sure you remember the rules of specificity and order things correctly.

As with the previous examples, we'll render this first, so you can see what it looks like before we start.

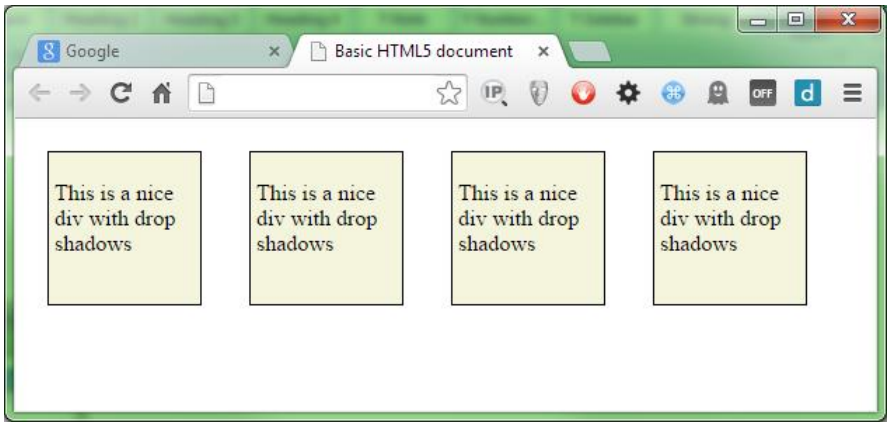


Figure 63: Drop shadow divs before styling

Once we have that working, we'll create some sample rules to apply different drop shadows to our elements.

```
.niceDiv.topLeftShadow
{
  box-shadow: -10px -10px;
}

.niceDiv.bottomLeftShadow
{
  box-shadow: -10px 10px;
}

.niceDiv.bottomRightShadow
{
  box-shadow: 10px 10px;
}

.niceDiv.topRightShadow
{
  box-shadow: 10px -10px;
}
```

Code Listing 62b: CSS rule to apply different shadows to the HTML in 62a

Now we need to make a small alteration to the HTML, and add each of the new class names **topLeftShadow**, **bottomLeftShadow** and so on one to each **div** in the HTML in Code Listing 62a, as follows:

```
<div class="niceDiv topLeftShadow">
    <p>This is a nice div with drop shadows</p>
</div>
<div class="niceDiv bottomLeftShadow">
    <p>This is a nice div with drop shadows</p>
</div>
<div class="niceDiv bottomRightShadow">
    <p>This is a nice div with drop shadows</p>
</div>
<div class="niceDiv topRightShadow">
    <p>This is a nice div with drop shadows</p>
</div>
```

Code Listing 62c: Adding the four drop-shadow classes to our divs

Just in case you're wondering what's going on here, we're using the cascading nature of CSS to our advantage so that we don't have to type out the basic beige **div** style four times in each of the four rules; doing things this way means less typing and better re-use.

If we re-render that, you should now see that we have four default, blocky drop shadows for each of the directions on each of our **div** elements.

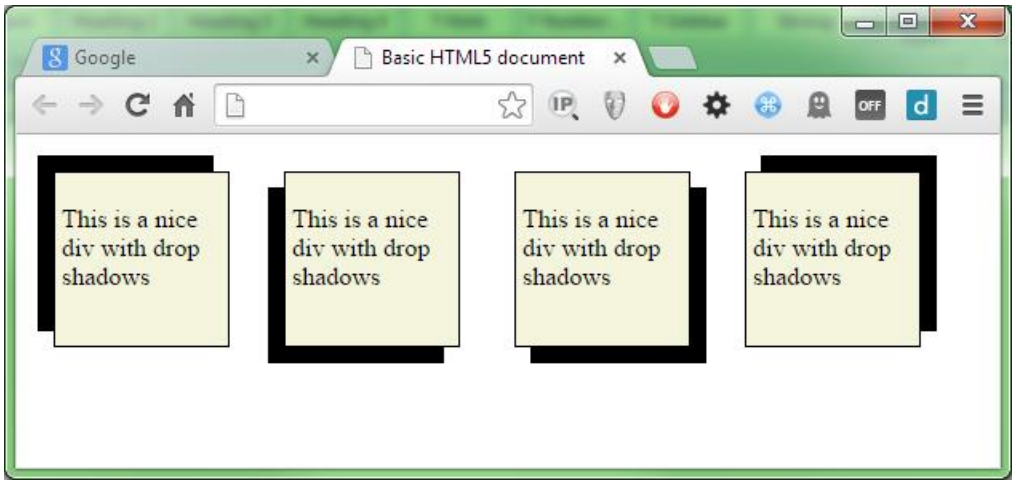


Figure 64: Our div elements with default drop shadows

The default settings for drop shadows, however, are very bold and solid; changing this is where the remaining two optional measurements come into the equation.

The first value is the blur radius; you can't use negative values on this, but any positive values you supply blur the shadow out further and further, giving a much softer appearance with each increasing value.

Change the rules in Code Listing 62b as follows:

```
.niceDiv.topLeftShadow
{
  box-shadow: -10px -10px 5px;
}

.niceDiv.bottomLeftShadow
{
  box-shadow: -10px 10px 5px;
}

.niceDiv.bottomRightShadow
{
  box-shadow: 10px 10px 5px;
}

.niceDiv.topRightShadow
{
  box-shadow: 10px -10px 5px;
}
```

Code Listing 62d: CSS rules changed to blur the shadows

The re-render your test HTML document, you should now see the shadows become slightly softer.

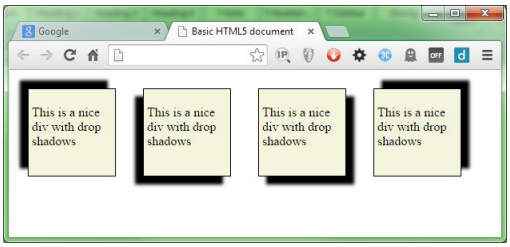


Figure 65: Our div elements with a small blur added

The larger you make the blur, the softer the shadow's edge will be.

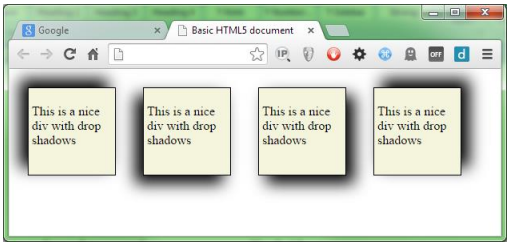


Figure 66: Our div elements with a 30px blur radius

The fourth parameter controls how much the blur spreads away from the edge, and is called the spread radius.

If we add 10 pixels onto the blur radius, our shadow will grow to a total of 20 pixels (we already have a 10-pixel offset), but the final 10 pixels will be used for the blur and its associated fadeout. Change the shadow rules as follows:

```
.niceDiv.topLeftShadow
{
  box-shadow: -10px -10px 30px 10px;
}

.niceDiv.bottomLeftShadow
{
  box-shadow: -10px 10px 30px 10px;
}

.niceDiv.bottomRightShadow
{
  box-shadow: 10px 10px 30px 10px;
}

.niceDiv.topRightShadow
{
  box-shadow: 10px -10px 30px 10px;
}
```

Code Listing 62e: CSS rules changed to blur the shadows

Then re-render, and you should get some very large, very fuzzy shadows with very soft edges:

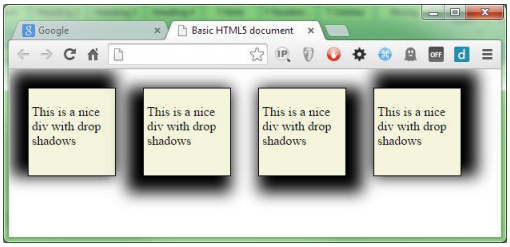


Figure 67: Our div elements with an optional spread radius applied

Getting the right-looking value for the optional values can be very much a case of trial and error, but generally, the larger the spread radius, the more space the blur has to fade things out, and so can make a smoother transition.

Note: If you have some specific Z-ordering (that is, if you're using div layers), then it is possible to cast a shadow onto sibling elements. For this reason it's usually better to try and keep drop shadows as low in the stack of elements as possible, or size the blurs and offsets so that things stay within any margins you have defined on the element.

The final value is the color value, and like any other color property on an element, this can take any standard color value. For now we'll just use the standard 6-digit # symbol way of specifying color, as we'll be taking a closer look at defining colors in the next chapter.

Alter your rules so that they look as follows:

```
.niceDiv.topLeftShadow
{
  box-shadow: -10px -10px 20px 5px #FF0000;
}

.niceDiv.bottomLeftShadow
{
  box-shadow: -10px 10px 20px 5px #00FF00;
}

.niceDiv.bottomRightShadow
{
  box-shadow: 10px 10px 20px 5px #0000FF;
}

.niceDiv.topRightShadow
{
  box-shadow: 10px -10px 20px 5px #FFFF00;
}
```

Code Listing 62e: CSS rules changed to color the shadows

When the HTML is reloaded, you should see your shadows are now much brighter, more colorful, and nicer looking.

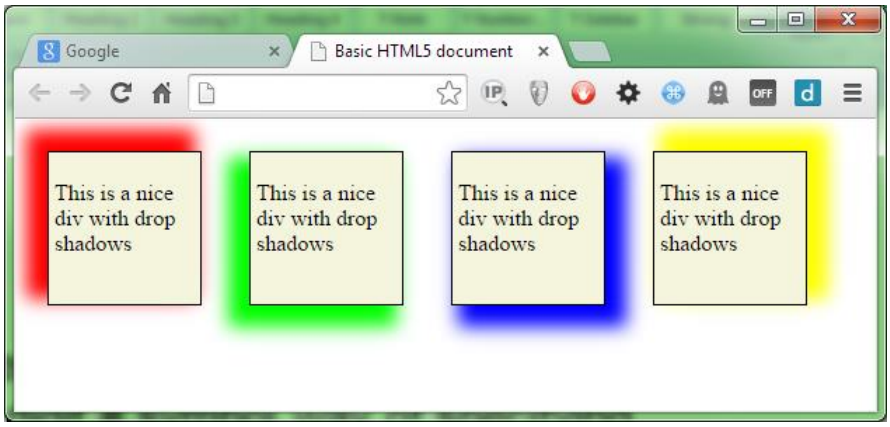


Figure 68: Our shadows are now much more colorful

You can make the shadow appear inside the element (rather than outside as we have been doing) by specifying the optional **`inset`** keyword on the rule. If you use inset, however, you must supply it before any values. Alter our rules one last time:

```
.niceDiv.topLeftShadow
{
  box-shadow: inset -10px -10px 20px 5px #FF0000;
}
```

```
.niceDiv.bottomLeftShadow
{
  box-shadow: inset -10px 10px 20px 5px #00FF00;
}

.niceDiv.bottomRightShadow
{
  box-shadow: inset 10px 10px 20px 5px #0000FF;
}

.niceDiv.topRightShadow
{
  box-shadow: inset 10px -10px 20px 5px #FFFF00;
}
```

Code Listing 62e: CSS rules changed to inset the shadows

And re-render to see the effect.

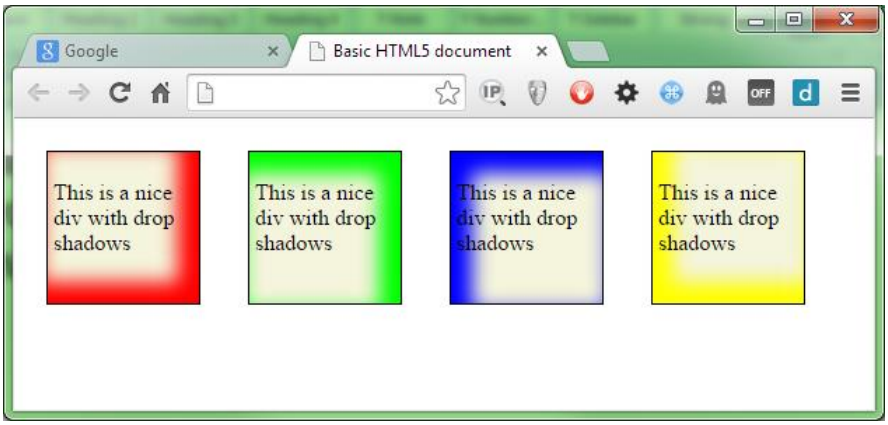


Figure 69: Our div elements with inset shadows

Drop shadows on text

CSS3 also supports adding drop shadows to regular text elements. When you do this, the shape of the shadow will follow the shape of the text rather that it just being the outline of the element's border.

The difference here is that while the parameters are the same (more or less), they are specified in a much different order to the parameters for box-shadow.

When specifying a text shadow, the rule is laid out as follows:

`text-shadow: <color> <offsetx> <offsety> <blurRadius>`

Text shadows don't support spread radius, and the color has to come first in the list rather than last.

By way of a quick example, add the following HTML to your document:

```
<h1>Drop Shadow Text</h1>
```

Code Listing 63a: HTML to demo drop shadow text

Then add the following style rule:

```
h1
{
  text-shadow: red 10px 10px 10px;
}
```

Code Listing 63b: CSS rule to add a drop shadow to our text in code 63a

You should end up with something like Figure 70:

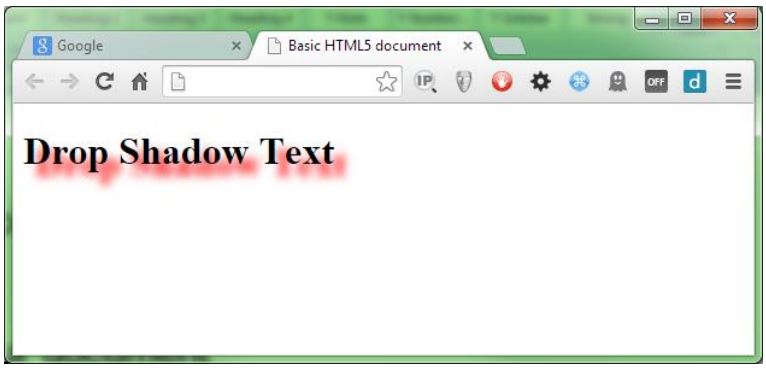


Figure 70: Text with a red drop shadow

The parameters work the same way as for the box shadow, so we'll not go into any lengthy descriptions; instead, I'll leave it as an exercise to the reader to experiment.

The last thing you need to know about both shadow rules is that you can add multiple shadow declarations to an element, separated by a comma.

When you do this, the effect will be a cumulative one, and any shadows that overlay each other will be drawn front to back in the order they are specified in the CSS rule. This can be used to create some rather strange halo-like effects around elements. Remember, however, to use them sparingly: even though browsers these days might be getting faster, effects like this still take power and CPU energy to draw, and a mobile device user might not thank you for draining their device battery all in the name of looking pretty.

Graduated Fills

I'm sure that at some stage in your development of HTML documents you've come across the need for a graduated fill of some description in the design of a page.

It could be a subtle change in color, or a very bold change in a short amount of space. A correctly used graduation of color can make a lot of difference.

As with many graphical effects, in the past you would generally have resorted to an image-editing package such as Photoshop to create the color graduation. Then you would have applied the resulting graduation as a background image and repeated it as needed to cover the full size of the element you were targeting.

With CSS3 you can now use a gradient fill type to create color ranges on the fly without ever using an image editor ever again.

Notice in the previous paragraph I called the gradient fill a “type.” Unlike everything we've seen so far, graduated fills are not a rule all of their own, as shadows and curved borders are. They are in fact a type of image that the CSS engine creates for us, and are typically (but not always) used with any rule where you might normally specify an image.

This means that you don't have rules in your style sheet called gradient rules; instead, you create CSS rules for a given target using normal CSS properties such as color, background-color, background-image, and many others. Then you replace or override all or part of the rules already defined with the areas you wish to apply a gradient to.

This is important for backward compatibility, because it means that you'll always have a fall back, which means CSS color gradients are one of the many new things you can use without having to do anything too special to support older browsers.

In typical usage, you would create your rule (remembering specificity order) to style an element in a way that looks good for the older browsers. Then you'd either add class-specific selectable rules to extend the original rule, or you'd add the gradient generators towards the bottom of the rule in question after the basic properties had already been set.

Producing gradients in this manner allows the CSS engine in your browser to do all the hard work for you, meaning no polyfills or JavaScript hacks to provide backwards compatibility.

Now that you know the basics, let's see how we can apply it.

First we need some HTML markup to play with, so let's create a simple, single **`div`** and put it in an HTML document.

```
<div class="gradDiv">
    <p>I'm a div with a gradient</p>
</div>
```

Code Listing 64a: HTML for our graduated fills

Let’s also define the following rule as a starting point to experiment with:

```
.gradDiv
{
  width: 200px;
  height: 200px;
```

```
  background-color: salmon;
  padding: 4px;
  border: 1px solid black;
  display: inline-block;
  margin: 15px;
}
```

Code Listing 64b: Basic style for our graduated div

The initial render of this should look as follows:

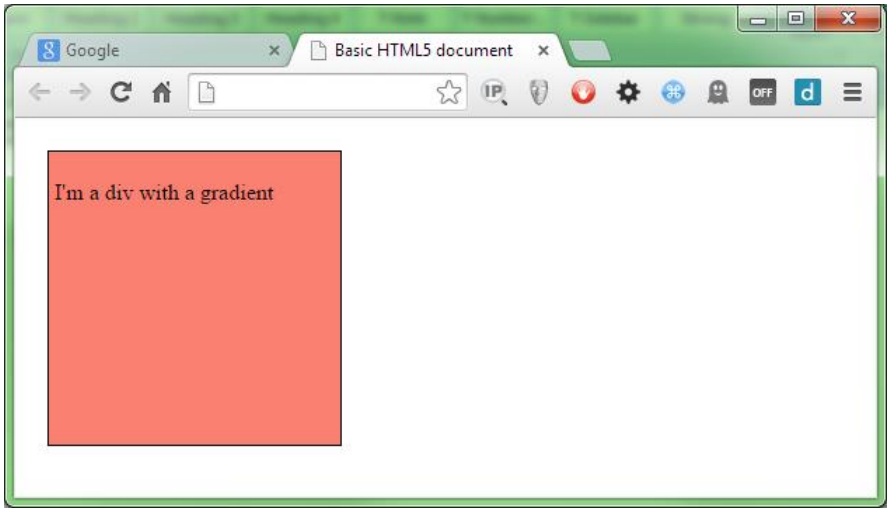


Figure 71: Our un-graduated div element

Gradients come in three types: linear, radial, and repeating, each with its own syntax and format.

Note: Until quite recently, many of the major browsers used vendor prefixes to define things like gradients. Thankfully, all of them now support un-prefixed versions of gradient fills, and each browser has settled on using a standard syntax. If you're tasked with supporting older browsers (and by older here I mean moderately old: IE9, FF20, etc.), then you might find that you need to specify gradient fills (and for that matter, other CSS3 features, too) more than once, and in several different forms. Throughout this book, I'm keeping to properties and functionality that can be used in an un-prefixed manner to keep things simple. Gradient fills have one extra bite that you need to watch for: the actual syntax to specify a fill is drastically different between the different prefixed versions, and only standardised non-prefixed versions.

We'll start with the simplest of these, a linear gradient.

Add the following to your style file (but don't replace the rule we already defined):

```
.gradDiv
{
```

```
  background-image: linear-gradient(red, yellow);
}
```

Code Listing 64b: Extension to the style for our graduated div

When you render that in your browser, you should see it change to:

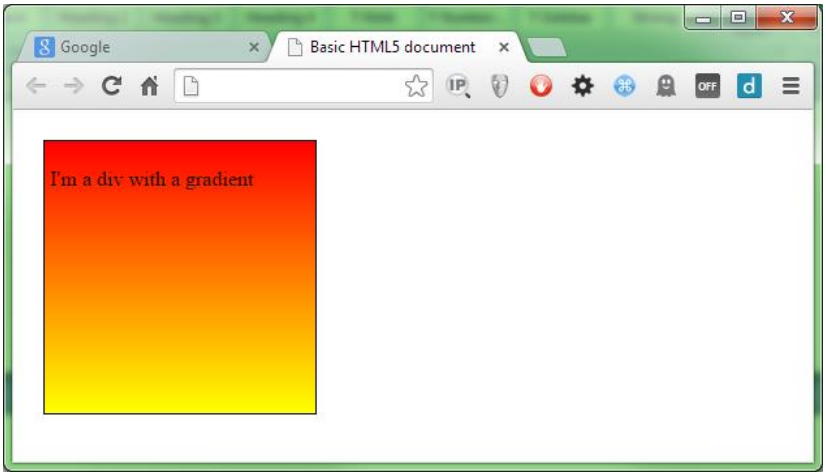


Figure 72: Our first graduated div

As you can see, the linear gradient is very simple to use. We've told the CSS engine to start at red and end at yellow, and it does just that, creating an equal gradient across the entire size of the target element.

We can also change the direction by using the **to** keyword. Change the rule in 64b so that it reads as follows:

```
.gradDiv
{
  background-image: linear-gradient(to left, red, yellow);
}
```

Code Listing 64c: CSS rule with a direction change

When we refresh the HTML, we should find that our gradient now starts at the right and goes toward the left.

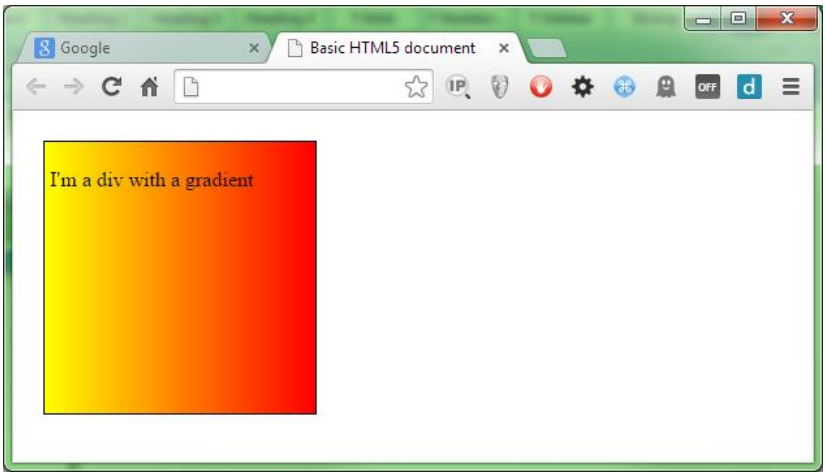


Figure 73: Our gradient now runs right to left

You can combine the directions such that you can have “to top left” or “to bottom right,” allowing you to graduate from corner to corner rather than just top to bottom or left to right.

You can also provide an angle. For example:

```
.gradDiv
{
  background-image: linear-gradient(30deg, red, yellow);
}
```

Code Listing 64c: CSS rule with a direction change

This will give you a gradient traveling from red to yellow at an angle of 30 degrees of the 0 axis.

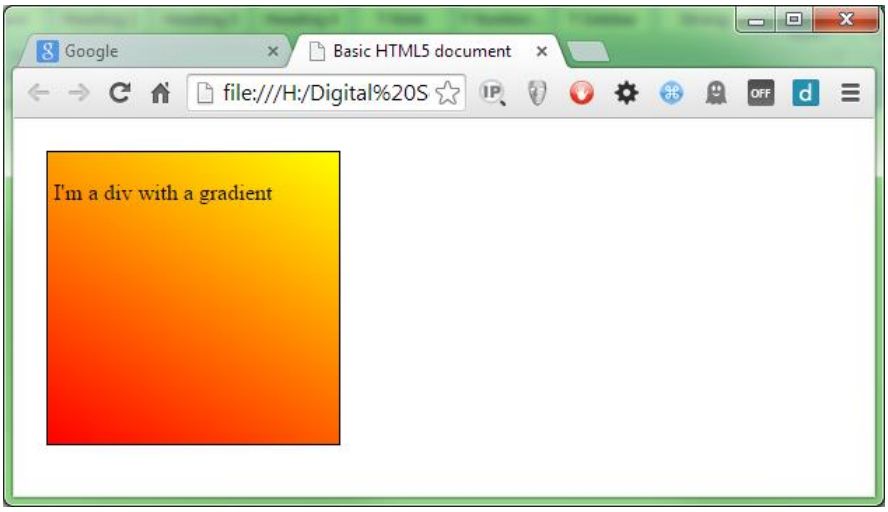


Figure 74: Our gradient at 30 degrees

Note: Depending how far back you need to support, you might still need to use vendor prefixes. I've noted that the vendor-prefixed versions use different syntax to specify the gradients. What I haven't yet mentioned, however, are how the rotations are calculated. Standard syntax sees 0 degrees at the middle bottom of the target element, with positive increments from 0 to 360 degrees traveling clockwise. You can see this in the 30 degree rotation in the previous example as the red starts from the lower-left corner. In the browser-prefixed versions, two different schemes were used—the standard as it is now, and one where 0 degrees was middle right with positive rotations going counterclockwise from 0 to 360.

Linear gradients can also contain more than two colors; in fact, they can contain as many colors as you want. Remember those rainbow-colored backgrounds folks used to be so passionate about? Try the following extended CSS rule.

```
background-image: linear-gradient(red, orange, yellow, green, blue, indigo,
violet);
```

Code Listing 64d: CSS rule with distinct rainbow

Rendering that in your browser might just hurt your eyes!

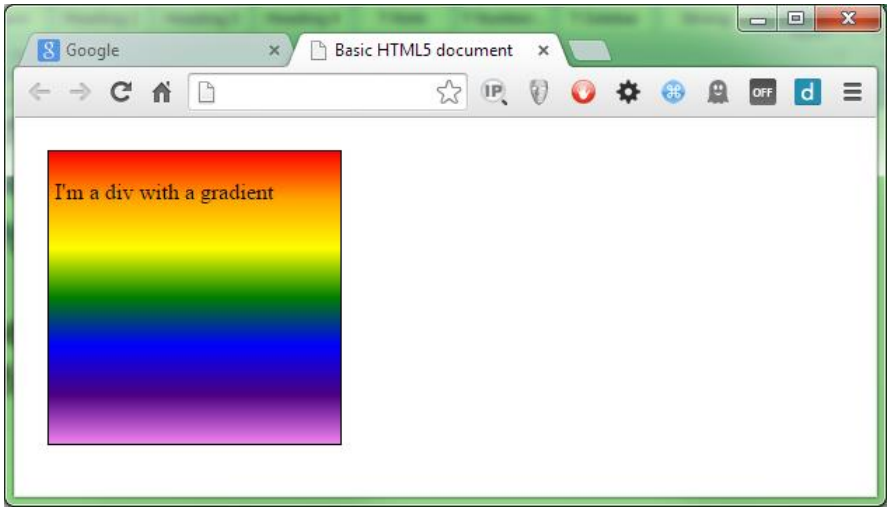


Figure 75: Our div now looks like a rainbow

You'll notice that the color spread is still evenly distributed across the height of the element. We can change that easily using color stops.

If you've ever used an image editor like Photoshop or Gimp, then you'll most likely know what color stops are. They are set values that tell the graphics engine to change color at a given point in the scale.

To demonstrate what I mean, change your extension rule so it reads as follows:

```
.gradDiv
{
  color: white;
  background-image: linear-gradient(black 0%, black 20%, green 30%, black 40%,
black 100%);
}
```

Code Listing 64e: CSS rule now using color stops

You should find that when you render your HTML now, you have a black background with a graduated green bar across it.

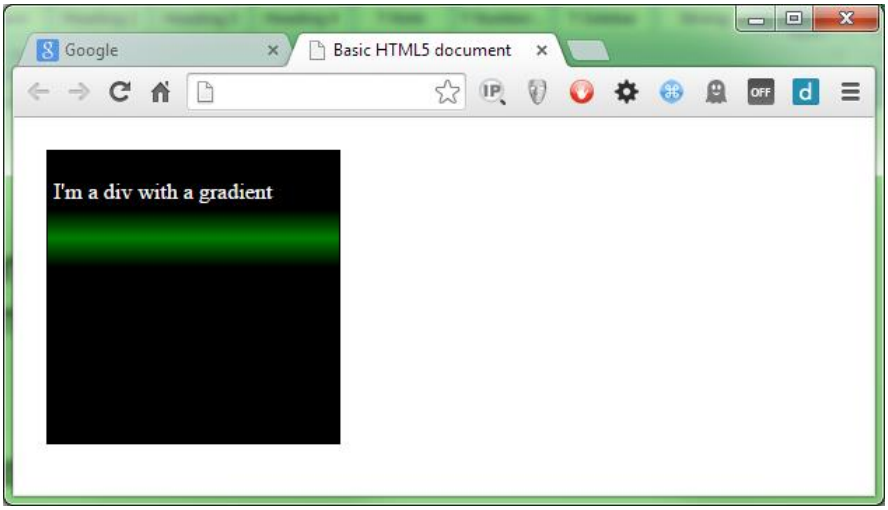


Figure 76: Our div now looks rather dark

You can specify the stops using any value type you like, but for ease of reading I would recommend you stick to percentage values, especially if you need to finely control the values as I've done in Figure 76.

Radial Gradients

Like linear gradients, radials start at a defined point (usually the center of the element) and produce the color graduation in an outward motion. They also take the same set of color settings (and color stops) as linear gradients do.

Starting with a simple example (and re-using the HTML from the previous section), change your CSS rule as follows:

```
.gradDiv
{
  background-image: radial-gradient(red, yellow);
}
```

Code Listing 65a: our CSS rule altered to be a radial gradient

This should produce the output shown in Figure 77:

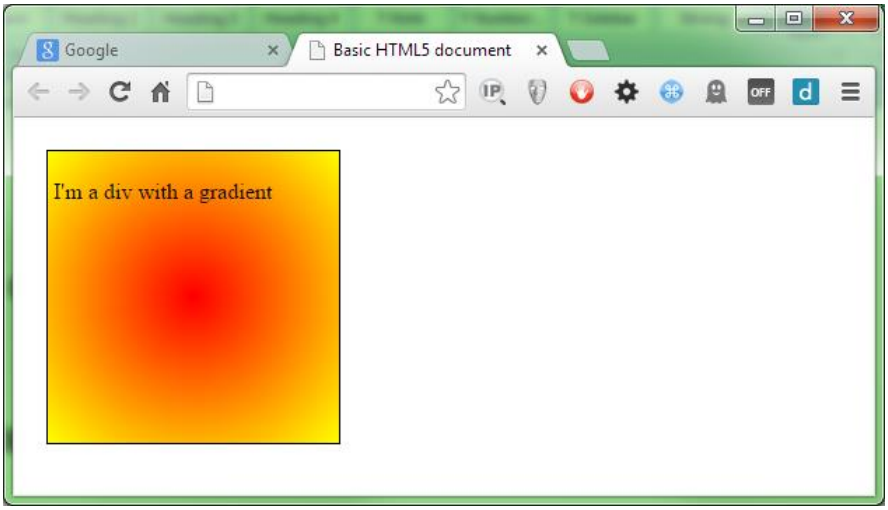


Figure 77: We've changed the gradient to a radial one

Just as with the linear gradient, we can change the start position. So, for instance, we could easily start the graduation from the upper-right corner by doing the following:

```
.gradDiv
{
  background-image: radial-gradient(at top right, red, yellow);
}
```

Code Listing 65b: Radial gradient changed to the top right

This gives us the following:

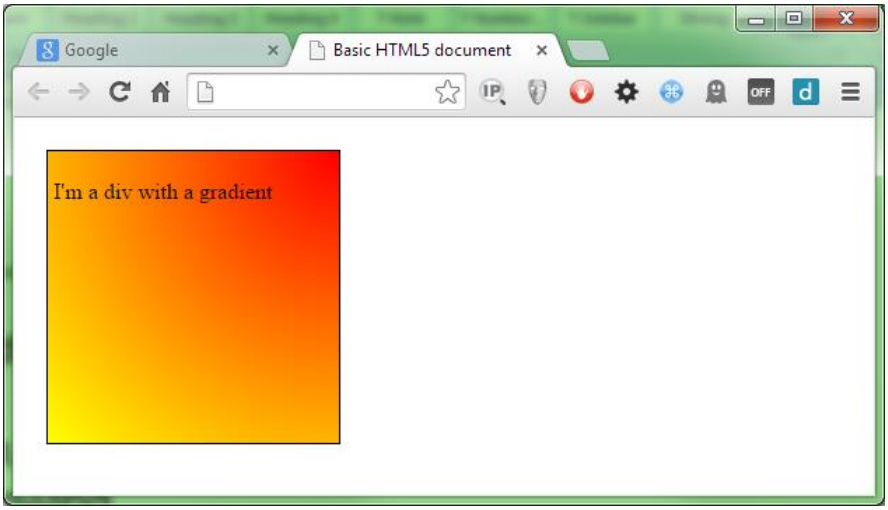


Figure 78: Radial gradient coming from the top right

You can also supply an x and y offset using any valid CSS value. Using percentages here is interesting because you can make positional radials that are dependent on the size of the target element. In the following rule we're starting 75 percent along the x-axis (0 is left) and 60 percent down the y-axis (0 is top).

```
.gradDiv
{
  background-image: radial-gradient(at 75% 60%, red, yellow);
}
```

Code Listing 65c: Radial gradient changed to 75% X and 60% Y

Rendering this should give you something like the following:

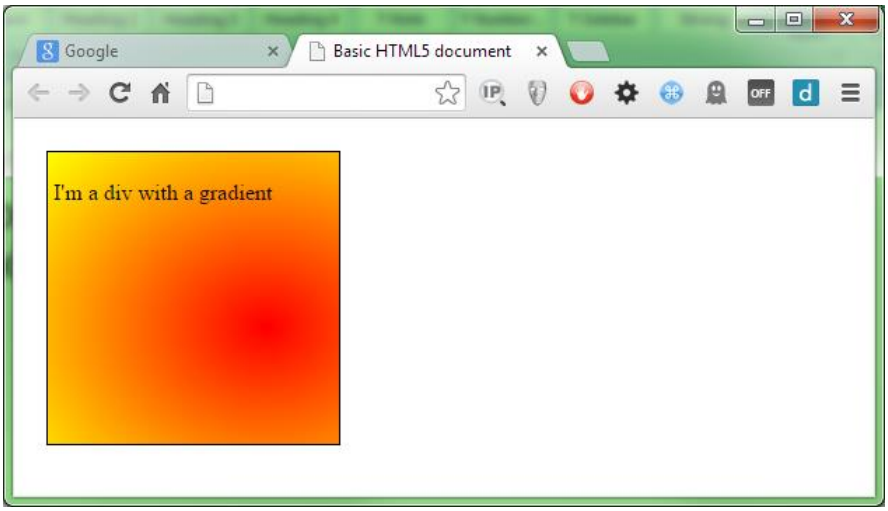


Figure 79: Our div with a gradient centered at 75% by 60%

If you've been looking closely at the rendered output, you may have noticed that the actual gradient is quite circular.

By default the CSS3 engine produces radial gradients with an elliptical shape rather than a balanced, circular shape.

If you need your radial gradient to be circular, then you can force the CSS3 engine to produce circular gradients by using the **`circle`** modifier as follows:

```
.gradDiv
{
  background-image: radial-gradient(circle at 75% 50%, red, yellow);
}
```

Code Listing 65d: Applying the circle modifier to our rule

You might not be able to tell the difference here if you're using the 200px by 200px element we've used so far, so you may want to try changing the width in the parent rule to something like 400px, and then try rendering with and without the circle modifier applied.

At 400 by 200 with the **circle** modifier applied, you should see something like this:

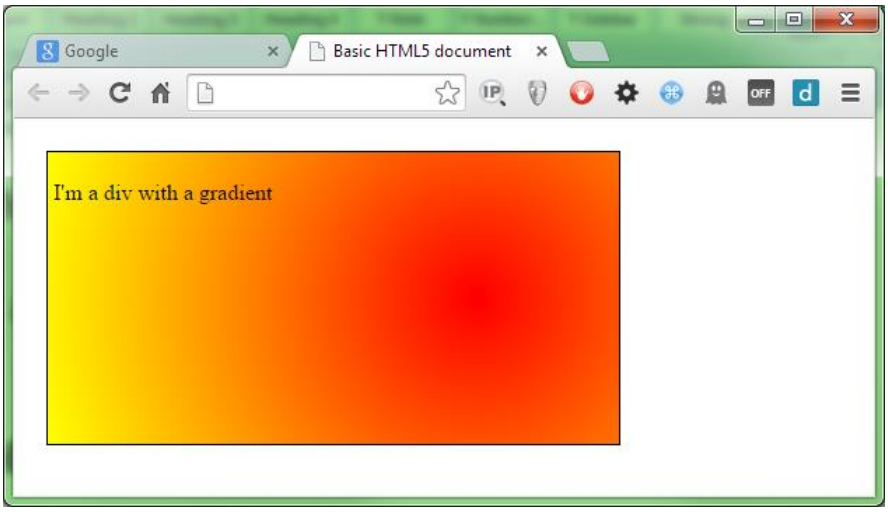


Figure 80: Our gradient with the circle modifier applied

There are other modifiers that you can add too; these are mostly used to control things based on width. For example, if you have a short and wide element, and you're specifying the **circle** modifier, then the largest color range will be along the longest edge. This means that from top to bottom you may only see your start color, but the full fade will be visible across the width. To combat this, one of the modifiers you can supply is **closest-side**, which will force the majority of your gradient into whichever side dimension is the smallest while letting the graduation run out early across the wider side. As with all of the different settings available, you can find these listed along with their explanations on the Mozilla developer network.

Repeating Gradients

The syntax for repeating gradients is identical to that of a linear gradient, the only difference is that you specify them using **repeating-linear-gradient** rather than just **linear-gradient**. However, there is a caveat that you need to watch out for, and that is you MUST supply your colors with color stop values.

If you try to specify a repeating gradient by just supplying non color stop values, you'll simply end up with something that just looks like a standard linear gradient for example

```
.gradDiv
{
  background-image: repeating-linear-gradient(red, yellow);
}
```

Code Listing 66: A repeating linear gradient

This will produce the following image:

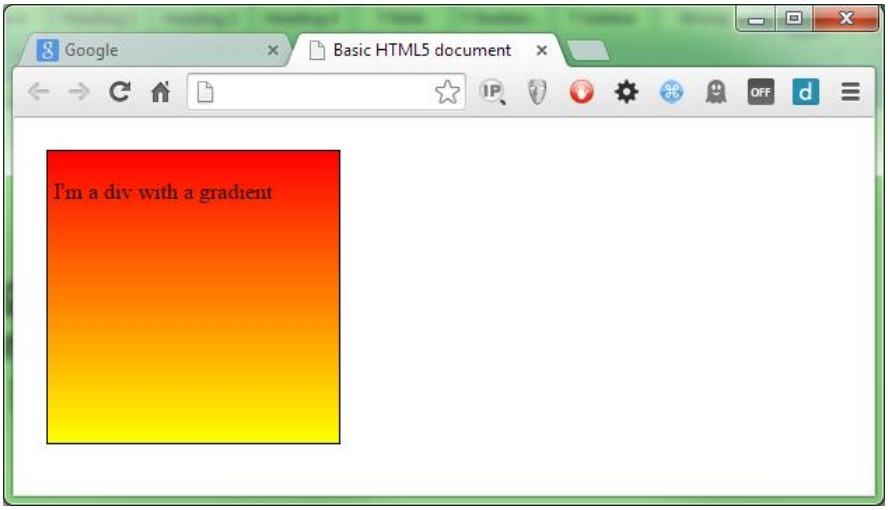


Figure 81: Our repeating gradient doesn't work correctly

If we alter the gradient fill so that it looks like this:

```
.gradDiv
{
  background-image: repeating-linear-gradient(red 0%, red 10%, yellow 10%, yellow
20%);
}
```

Code Listing 67: A repeating linear gradient that works correctly

You'll find that when you render it, you get the following:

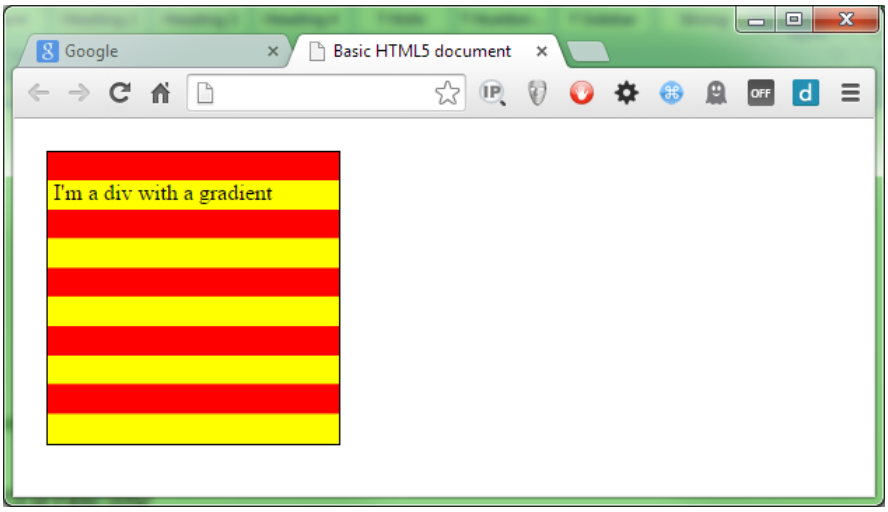


Figure 82: Our gradient now repeats

What's happening here is quite simple: we start at 0 percent and make sure we have red to 10 percent, and then we start yellow at that 10 percent mark and end it at 20 percent.

The CSS3 engine looks at the rule and sees that it runs out of color 20 percent into the element width, and so it repeats the first 20 percent to fill in the remaining 80 percent.

So does this mean we can never use 100 percent of our graduation to repeat? No, not at all; while the 100 percent width in our examples has been the width of the element, the calculations for the gradient are actually the width of the background.

In CSS3, you can now set the size of your background using the **`background-size`** rule.

The relevance to what we do here is that if we set our background size to 10px by 10px, then as far as our repeating gradient is concerned, that 10px would be 100 percent of the size. If that 10px by 10px background was then applied to a 200px by 200px element, the background would repeat 20 times in the X and Y directions unless otherwise instructed.

The only reason we see 100 percent as the width of the element in our examples is because we're leaving the background width and height at the default, which happens to be 100 percent of the target element.

With that in mind, wherever you end a color stop, your repeating gradient will repeat within itself. This means that if you are resizing the background, you need to make sure that any color stops you make repeat also repeat across tiled image boundaries—unless misalignment is what you're aiming for.

Summary

In this chapter, you've seen some of the eye candy that CSS3 introduces. Next, we'll look more closely at the different ways of specifying color in CSS3 and learn about some of the things we can now use, such as alpha channels.

Chapter 6 Color

If you've done any web development at all prior to reading this, then you've almost certainly come across the standard way that colors are defined in HTML/CSS using the 6-digit hex notation: **#rrggbb**

What exactly does this mean? It's actually quite simple. In effect, it's a 24-bit color value made up of three 8-bit components.

In the hexadecimal number scheme, digits are counted from 0 to F rather than 0 to 9 as in the normal decimal number system. When you look closely, you realize that you can fit all of those values into 4 bits, and since the maximum number 4 bits can represent in decimal is 15, then 0 to F actually means 0 to 15.

If you double that up, the 4 bits becomes 8 bits, and the maximum you can then fit is 255, so FF (our largest hexadecimal number) becomes 255 (our largest decimal number). My aim here is not to turn this into a lesson on hexadecimal math; my aim is just to explain to you that a color value is made up of 256 (0 to 255) levels of Red, Green, and Blue values, with 0 being no color and 255 being the max amount of that color.

If you wanted a mid grey, for example, you might use 127 for the values of Red, Green, and Blue, which, when translated would give you **#7F7F7F**.

If you're using Windows it's very easy to convert by using the Windows Calculator. Type **calc** into your run box, or search for it in the Start menu. When it loads, on the **View** menu, choose **Programmer**.

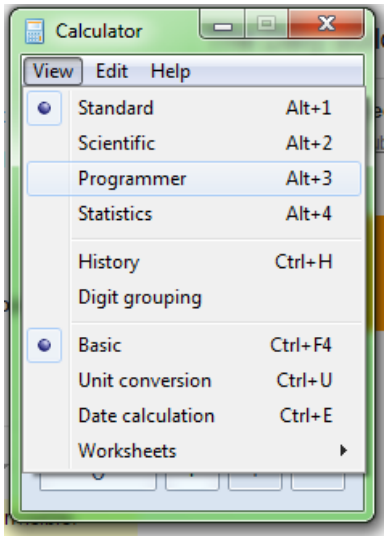


Figure 83: Windows Calculator View menu

You should end up with following:

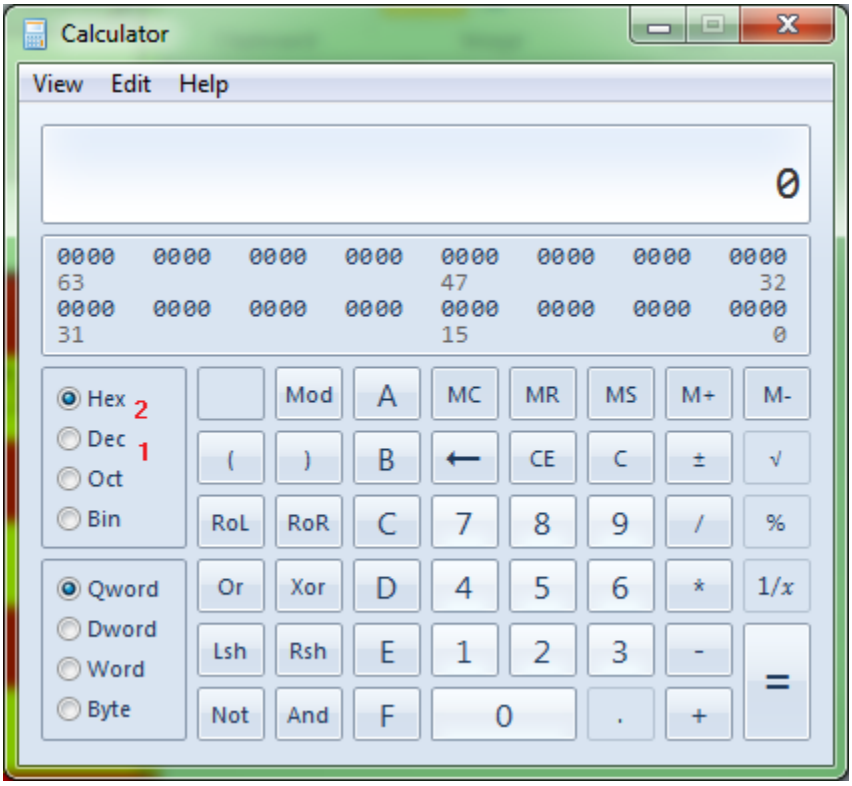


Figure 84: Windows Calculator in programmer mode

To convert to Hex, make sure you select **Dec** (marked “1” in Figure 84) then type in your value. Select **Hex** (marked “2” in Figure 84) and you'll see the value to place in your color value. If your value is only one digit (from 0 to F), then simply prefix it with an extra 0 (01, 02 … 0E, 0F). Do this for each of the three values, and plug them all together to make your final color. Easy, right?

Well no, not really, because you either have to understand hex/dec conversion, or need some kind of tool on hand to make sense of things.

This how it was at the start, and this is how it's always been—at least until CSS3 came along.

Color Names

CSS2 introduced proper names for some colors, but they weren't really publicized; this, however, was greatly expanded with CSS3. You've already in this book that I've used names such as red, green, and blue. There are actually 140 of them altogether, as Figure 85 shows.

aliceblue	antiquewhite	aqua	aquamarine	azure
beige	bisque	black	blanchedalmond	blue
blueviolet	brown	burlywood	cadetblue	chartreuse
chocolate	coral	cornflowerblue	cornsilk	crimson
cyan	darkblue	darkcyan	darkgoldenrod	darkgray
darkgreen	darkkhaki	darkmagenta	darkolivegreen	darkorange
darkorchid	darkred	darksalmon	darkseagreen	darkslateblue
darkslategray	darkturquoise	darkviolet	deeppink	deepskyblue
dimgray	dodgerblue	firebrick	floralwhite	forestgreen
fuchsia	gainsboro	ghostwhite	gold	goldenrod
gray	green	greenyellow	honeydew	hotpink
indianred	indigo	ivory	khaki	lavender
lavenderblush	lawngreen	lemonchiffon	lightblue	lightcoral
lightcyan	lightgoldenrodyellow	lightgray	lightgreen	lightpink
lightsalmon	lightseagreen	lightskyblue	lightslategray	lightsteelblue
lightyellow	lime	limegreen	linen	magenta
maroon	mediumaquamarine	mediumblue	mediumorchid	mediumpurple
mediumseagreen	mediumslateblue	mediumspringgreen	mediumturquoise	mediumvioletred
midnightblue	mintcream	mistyrose	moccasin	navajowhite
navy	oldlace	olive	olivedrab	orange
orangered	orchid	palegoldenrod	palegreen	paleturquoise
palevioletred	papayawhip	peachpuff	peru	pink
plum	powderblue	purple	red	rosybrown
royalblue	saddlebrown	salmon	sandybrown	seagreen
seashell	sienna	silver	skyblue	slateblue
slategray	snow	springgreen	steelblue	tan
teal	thistle	tomato	turquoise	violet
wheat	white	whitesmoke	yellow	yellowgreen

Figure 85: 140 CSS3 color names

There are actually a few more (147 in total), but those are just alternate spellings of colors already specified, such as “darkslategrey” (for those who spell the word “gray” with an “e”).

Anywhere you have to specify a color in your CSS rules, you can use these color names.

If, like many of us, you have trouble remembering them, then you might find this website useful.

The site gives you a random color from the list on its home screen each time you load or refresh the page and lists the color name in big letters. If you click on the small grid icon just below the text, you'll be taken to a page where you can hover over colors, sort them in different ways, and generally examine the entire list.

CSS-Tricks also has a useful list available. It not only lists the colors, but also the #rrggbb values for each of them.

There's one last color name you need to know, and that’s **transparent**.

When you use the transparent color name, you're making the object to which you're applying the color value 100 percent see-through. Unfortunately, there is no way of varying the level of transparency using color names and hex values, but you can set the **`opacity`** CSS rule on an element, unless you want to use direct color values.

Color Values

If named colors are not enough, and the hex notation is too cumbersome for you, then you'll be pleased to know that with CSS3, you can define your colors using one of four new data types.

These types are used in a similar way to those seen previously in the section on gradients, by specifying them as a keyword followed by a set of parameter values in parentheses.

The values inside the parentheses can take either a fixed value or a percentage value. Our medium grey example from the beginning of this chapter could now be represented as follows:

- **`#7F7F7F`**
- **`rgb(127, 127, 127)`**
- **`rgb(50%, 50%, 50%)`**

All three of these values, when used in any place in CSS that a color could be specified, will result in the same color being generated.

You can also now specify colors with an Alpha Channel using the **`rgba`** keyword.

Note: An alpha component is a value that specifies a level of opacity that a given color uses. Think of it like a transparecy value where 0 is fully transparent and 255/100 percent is fully non-transparent with values in between offering differing levels of transparency. It's like a screen door: you can see through the door, but you can't just walk through it, because it's still a physical object. Transparency is similar, and the level of a given Alpha value determines how much you can see through the "door."

The **`rgba`** keyword is used in exactly the same way as the **`rgb`** keyword, but takes an additional value that specifies the Alpha level: **`rgba(127,127,127, 0.5)`**

Unlike the color values themselves, however, you cannot supply a percentage value or an integer from 0 to 255 for the alpha amount.

Instead, you have to use a floating point based number from 0 to 1 where 0 is fully transparent and 1 is fully non-transparent, with 0.5 representing the halfway mark.

Most people just go for the easy values, such as **`0, 0.1, 0.2…. 0.8, 0.9, 1`**

However, you can get finer control if you need it by doubling the digits up, like this: **`0.01, 0.32, 0.84`**

For the common percentage values, you'll often use the following values:

- 0.00 = 0%
- 0.25 = 25%
- 0.50 = 50%
- 0.75 = 75%
- 1.00 = 100%

When you double the numbers up this way, it suddenly becomes easy to correlate percentage values to the 0-to-1 scale used.

There's one more way of specifying colors we need to cover, and this will make more sense to those of you who have been trained in correct color theory as a professional graphic artist or designer. This is the **`hsl`** and **`hsla`** way.

Used in exactly the same way as an **`rgb`** color value, the **`hsl`** method gives different meaning to the first three values.

Rather mixing levels of Red, Green, and Blue, they instead specify Hue, Saturation, and Lightness.

I'm not going to go into the full theory of color here, but the Hue value represents how far along the color chart you are, with 0 degrees being red and 360 degrees being magenta/red; as you go around the 360 degree color wheel from 0 to 360, you get a different root color value.

The Saturation is how strong that selected color value is from 0 percent (very weak) to 100 percent (very strong), and the Lightness value is 0 percent (very dark/usually black) to 100% (very light or bright).

Everything is explained in detail on the W3C color module page. You'll find a very handy chart on this site that shows the various HSL color combinations and more information on how to specify them.

In the case of the **`hsla`** value, the Alpha component is specified in exactly the same way as it is for an **`rgb`** color.

Summary

In this fairly short chapter we looked at how colors are defined in CSS3. I've deliberately not created any code listings, as there's plenty of opportunity to go back to previous samples and replace the values in the **`color`**, **`background-color`**, and **`gradient`** rules if you want to experiment.

We've looked at how CSS3 has made it easier to specify colors in your style rules and introduced a few internet resources that will make remembering and specifying them even more useful.

In the next chapter, we're going to dive into using the new font functionality that's now available for CSS3-based pages.

Chapter 7 Web Fonts

For pretty much the entire life of the HTML/CSS specification, web developers and designers have been restricted to the same core set of fonts for their typography.

I've lost count of the amount of heated debates and discussions I've seen on the subject, and I know firsthand the frustration of having a client wanting text in a specific font only to have to tell them it can't be done.

If you wanted to use a given font, you often had to prepare the text in an art package or word processor that had access to the font you wanted, and then either save the image as a browser-compatible image file or make a screen grab from your word processor to use in your HTML document.

Needless to say, this wasn't easy. Spelling mistakes and simple text changes took an extremely long time to correct (and before anyone asks, I have had at least one client over the years that wanted EVERY bit of body text done this way), and it really was impractical for anything other than simple banners or headlines.

While there was a large amount of frustration surrounding the use of fonts, there were good, sound reasons for restricting things, and these usually boiled down to platform differences.

Early browsers were nowhere near as graphically capable as the browsers of today, and many operating systems only had a limited amount of fonts installed, so the HTML committees formulated a list of known fonts that were pretty much guaranteed to be present on any platform running any graphical browser. For the record, we're not just talking Windows and Mac—back then you had more variations of Unix and Linux, and each of these had their own specific desktop environments. It wasn't until the early 1990s that many of them started to standardize, using things like X11 and windows managers such as GNOME and KDE. Even Windows was available in at least four different versions.

The restricted base font list that was almost always guaranteed to work was:

- sans-serif
- serif
- fantasy
- cursive
- monospace

If you specified your font using one of these, you'd get that platform's version of the font, and while it might not always be exactly identical, the look, style, and feel would be close enough that your design's layout wouldn't be too badly affected.

Depending on your platform, you could also supply overrides to be used. Commonly on sites developed on Windows machines, you would see fonts described as:

Times, "Times New Roman", serif

Doing things this way meant, quite simply: try and see if you have the times font; if not, look for times new roman, and if that fails, then fall back to the 'serif' identifier, and let the platform select the best match.

You still couldn't just use any font you felt like, though.

For example, if you specified **`"Impact Bold", sans-serif`**, then any platform that had **`"Impact Bold"`** installed would display as intended. If the font wasn't available, then sans-serif would be used. It might look nothing like the original font, and so many designs would simply fall apart and fail to render correctly.

CSS3 fixes all of this by providing the ability to now package fonts in different formats with your web application and have the web server download and use them if it detects the target platform does not have them available.

This still doesn't mean that you should not use the web safe fonts list; all it takes is loss of connectivity, and you're back at square one. It does mean, however, that you could specify **`"Impact Bold"`** and have a very good chance of rendering in **`"Impact Bold`**" with display failures apparent now in extreme circumstances.

Using Web Fonts

Now that you know the "why," let's look at the "how."

Creating a suitable font

Perhaps the easiest way of getting a font to use with your design is to use the fantastic Font Squirrel website, which can be used to convert a font from your system into an instantly useable kit of CSS3 rules and style sheets.

Head to Font Squirrel and you should see something like this (valid as of January 6, 2015):

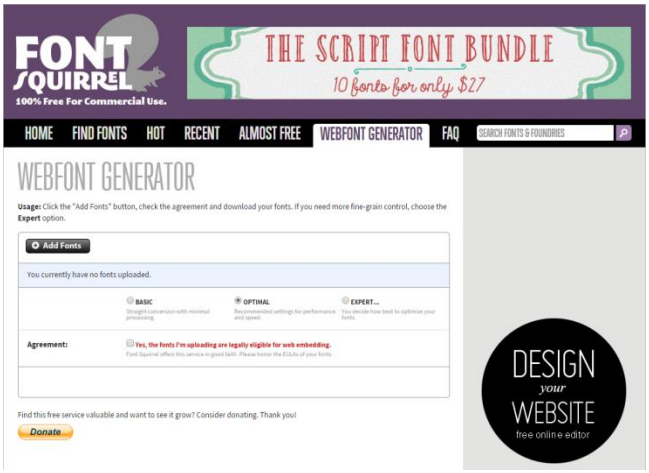


Figure 86: The Font Squirrel Webfont Generator

Using it is simple; you click, add fonts, and select the font files you wish to add. Once you've added the fonts you wish to convert, you MUST click the agreement box to proceed. Font Squirrel will then convert your fonts for you. Once it's finished, it will present you with a download link to a zip file containing your font in several different web-capable formats, along with a number of CSS files and HTML markup examples showing you how to use it.

Note: Be VERY careful about what you convert and use. Many typography providers are exceptionally strict about how you are and are not allowed to use their fonts. Most of them class this process of font conversion for use on a webpage as piracy and are less-than-friendly when pointing this out to website owners. I know of firms that have not even bothered with warning letters and have just slammed unsuspecting website owners with bills for licensing fees with immediate demand for payment. Unless you're using a font that you know you're allowed to use, ALWAYS be sure to check the font's licensing requirements. Many of them make it even more difficult by allowing you to create an image using a font and use that in your webpage, but will not allow the use of the font directly.

Another alternative to using fonts in your webpages is the open source Google Fonts directory.

Google makes a point with this tool of allowing all the fonts listed on the site to be useable in webpage designs.

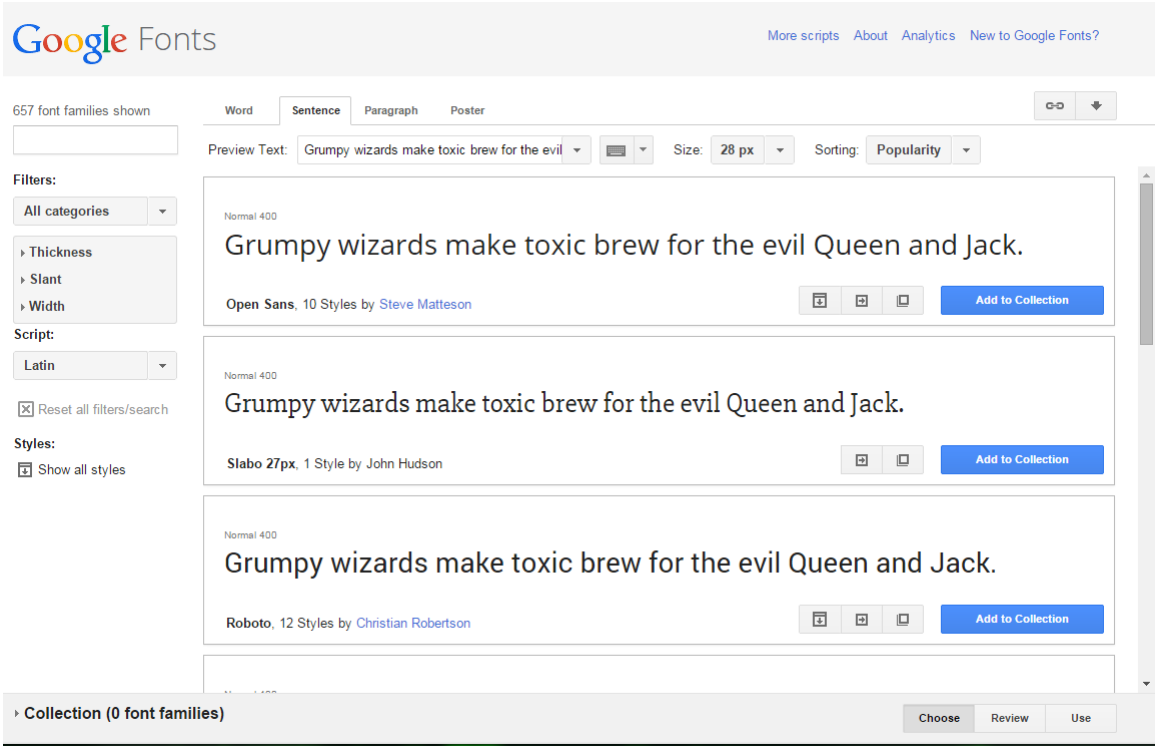


Figure 87: Google Fonts (January 2015)

With Google Fonts, you can search, filter, and compare any of the fonts in the list, then once you've decided on a font you wish to use, you can add it to a collection or view quick examples of its use.

The quick use page is particularly useful because it lists everything you need to know to immediately add that font to your webpage. By following the instructions on the quick use page and copying and pasting the sample code given, you get to use the font directly off Google's global content delivery networks, so it loads fast, and you don't have to go to the hassle of downloading it and adding it to your own web server.

If you choose to add it to a collection, then you can keep adding other fonts, and all will appear in your Collection panel at the bottom of the page.

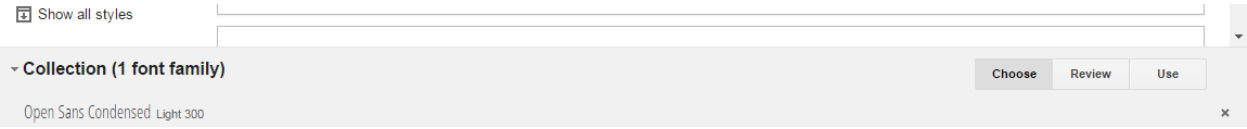


Figure 88: Google Fonts Collection panel

From this panel, you just need to click the **Use** button and you'll be taken to a page that looks the same as the Quick Use page to finish off the integration.

Manually adding fonts

To finish this chapter, I'll take you through an example of adding your own fonts to your webpages. I'll use two fonts that I previously converted using Font Squirrel called Amble and Nobile. If you go to the main page of Font Squirrel and look through the free fonts list, you can find both of these and more. You'll have to download them, unzip them, and then convert them using the web font generator tool.

Once you have the fonts converted, open the zip files that were provided to you, and you should find a number of files going by the name of the font you chose.

Looking at the zip file for the Nobile font you'll see something like the following (I'm using the excellent, free zip program here called 7-Zip):

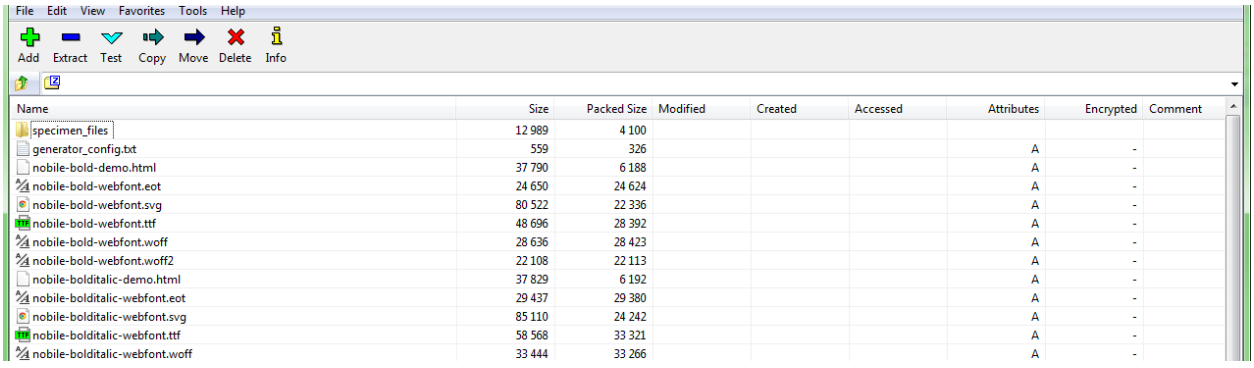


Figure 89: 7-Zip showing a partial listing of our font kit

As well as the expected .html and .css files, you'll see that you have .ttf, .eot, .svg, .woff, and .woff2 files; these five extensions are your actual font files, and it's these files that you need to copy to your web server for your pages to use. I generally put them in a folder called **fonts** in the root so I can keep them all together. How you do it is entirely up to you, but remember that the CSS styles will need to be able to find them, so you may need to put a **..** in path names when using them, depending on how you structure your websites.

Extract the font files from both zips into a location for your test webpage to use, then create the following HTML markup in a demo file to test them with:

```
<h1 class="ambleBold">I'm the header in Amble Bold</h1>
<p class="nobileRegular">
    I'm some general purpose paragraph text that's rendered using the 'Nobile
Regular'
    web font. As you can see I'm nothing special but for comparison...
</p>
<p>I'm some regular paragraph text using the default font</p>
```

Code Listing 68a: HTML markup to demo the usage of custom web fonts

Once you've created the test HTML, create a style sheet with the following rules:

```
body
{
  background-color: aliceblue;
}

@font-face
{
  font-family: 'amblebold';
  src: url('fonts/amble-bold-webfont.eot');
  src: url('fonts/amble-bold-webfont.eot?#iefix') format('embedded-opentype'),
       url('fonts/amble-bold-webfont.woff2') format('woff2'),
       url('fonts/amble-bold-webfont.woff') format('woff'),
       url('fonts/amble-bold-webfont.ttf') format('truetype'),
       url('fonts/amble-bold-webfont.svg#amblebold') format('svg');
  font-weight: normal;
  font-style: normal;
}

@font-face
{
  font-family: 'nobileregular';
  src: url('fonts/nobile-regular-webfont.eot');
  src: url('fonts/nobile-regular-webfont.eot?#iefix') format('embedded-opentype'),
       url('fonts/nobile-regular-webfont.woff2') format('woff2'),
       url('fonts/nobile-regular-webfont.woff') format('woff'),
       url('fonts/nobile-regular-webfont.ttf') format('truetype'),
       url('fonts/nobile-regular-webfont.svg#nobileregular') format('svg');
  font-weight: normal;
  font-style: normal;
}
```

```
.ambleBold
{
  font-family: 'amblebold'
}

.nobileRegular
{
  font-family: 'nobileregular'
}
```

Code Listing 68b: CSS font rules to style and use the HTML in Code Listing 68a

With the appropriate font files copied to a folder called **fonts** in the same place as your test HTML document and style sheet, if you load the document into your browser, you should see the following:

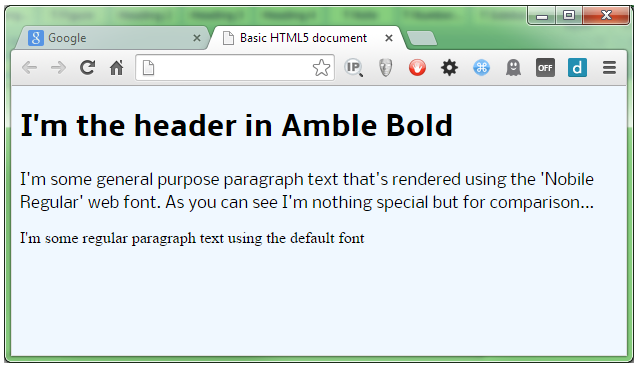


Figure 90: Our HTML rendered using a custom font

You can specify as many **`@font-face`** rules as you wish. Once you've defined them, you can simply use the name you defined them with in any CSS rule that you would use one of the original web-safe fonts in.

As you can see in our demo, however, I prefer to actually define a class-based rule with the font name in it. This allows me to just add the class name to the list of classes for an element; then if I ever need to change the font, I only need to change it in one place rather than in several different rules. Again, it's up to you how you structure this in your own projects; the only thing you have to make sure of is that the various **`url`** keywords in each of the font definitions can be downloaded.

One of the most common problems I see with developers working on the Windows platform under the IIS web server is that sometimes fonts don't render correctly, even though the font files are there and the CSS files are correct. The reason this happens is that by, default, IIS doesn't have a mime type listing for the woff and woff2 font extensions (on some older ones, ttf and svg are not defined, either). You can verify this by requesting the font directly in your browser's address bar. If the font does not have a correct mime type, you'll get a 404 error from the server telling you the font could not be found.

Fixing it is quite easy, and rather than repeat the instructions here, I'll just point you to this article and this article. These two links will give you everything you need to know to solve the problem.

If this occurs on other servers, then you'll need to refer to the appropriate documentation for that server to find out how to add the appropriate mime types.

Summary

In this chapter we saw how to use the font functionality that's part of CSS3, how to convert and add new fonts to our webpages, and also took on board some cautionary notes about using them.

Next up we'll take a look at generated content, counters, and calculations.

Chapter 8 Generated Content and Calculations

You've already seen some of what generated content can do in the form of the :before and :after pseudo selectors; the generated content module, however, can do a bit more than that with just a few simple CSS rules.

You can, for instance, read the value of an attribute from a parent element and then use that in a rule further down the chain.

You can also initialize and use counters, allowing you to use auto-generated sequential numbers for everything from list numbering to element indexing.

Finally, with the calculation module you can actually have your CSS rules perform basic math and workout exact screen sizes, border widths, and all sorts of other useful measurements—all without ever thinking about JavaScript.

Counters

Counters are controlled by normal CSS rules, and while they are very simplistic in nature, they have quite a few uses.

A counter is initialized using the **`counter-reset`** CSS rule, and it's incremented using the **`counter-increment`** rule.

You get the value of the counter by using the **`counter()`** function.

Let's imagine you have the following HTML markup (if you've read the rest of the book so far, you'll recognize this from Code Listing 58).

```
<dl>
    <dt>HTML 5</dt>
    <dd>HTML 5 is the markup language used to create documents that are published
on the internet</dd>
    <dt>CSS3</dt>
    <dd>CSS3 is the markup language used to add style and colors to documents
created using HTML 5</dd>
    <dt>HTTP (Hyper Text Transfer protocol)</dt>
    <dd>HTTP is the mechanism by which HTML 5 documents and their associated
resources are transported on the internet</dd>
    <dt>WWW (World Wide Web)</dt>
    <dd>The World Wide Web is the collective name for the many hundreds of
thousands of HTML 5 documents published via the internet</dd>
    <dt>The Internet</dt>
```

```
    <dd>The internet is a massive global computer network, which amongst other
things holds the World Wide Web</dd>
</dl>
```

Code Listing 69a: Definition list markup for counters demonstration

If we render it, we'll get the following.

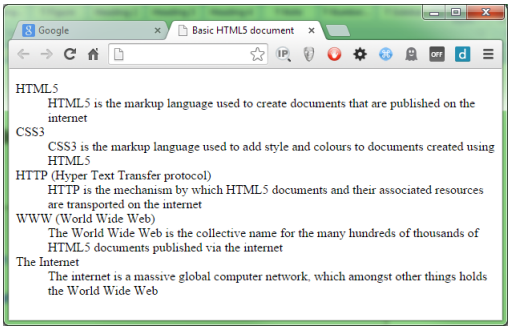


Figure 91: Our definition list ready to be styled.

First let's add a bit of default styling, just to neaten things up a little.

Add the following rules to your style sheet.

```
body
{
  background-color: aliceblue;
  font-family: sans-serif;
  font-size: 12px;
}

dt
{
  font-weight: bold;
  font-size: 1.1em;
  border-bottom: 1px solid grey;
  margin-bottom: 4px;
  margin-top: 8px;
  color: grey;
}

dd
{
  font-size: 0.9em;
  margin-top: 2px;
  margin-bottom: 2px;
  margin-left: 10px;
}
```

Code Listing 69b: Default style rules for our generated content

Nothing specifically new here, we're setting a default font style, size and background color for the page, then we're setting up some default styles for our definition list items.

The output should look like this:

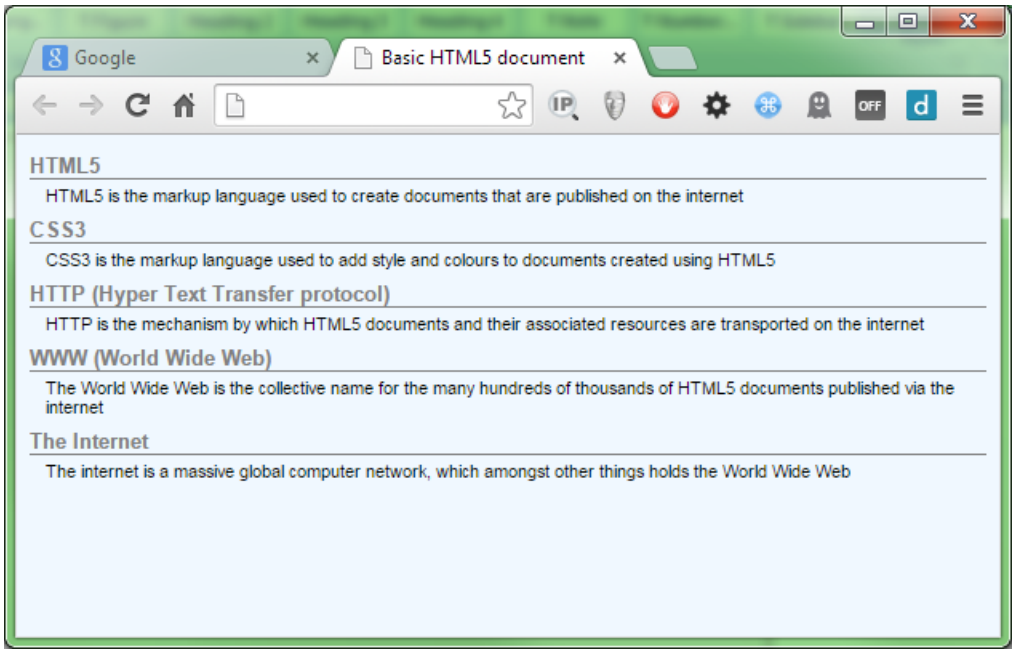


Figure 92: Definition list with default styling

What would be nice, however, would be if we could have each of our definition titles numbered.

Now we could go through each one and prefix the appropriate number on it, but that would be tedious and error-prone, and what if those definitions are published from a database that has no numbers ahead of time?

We could also change our **`dl`** elements to **`ol`** elements, then change all our inner list items to **`li`** elements, but then we'd have to start playing with an nth-child setup to highlight each alternate row as a header. We'd also have the problem of removing the numbers from the definition texts, leaving just the titles with numbers.

Both methods would work, but not without significant development and management work. CSS counters can solve this problem in a couple of small additions to the styles we've already developed.

First, let's reset our counter to **`0`** every time we encounter a **`dl`** element. Append this to your existing style sheet:

```
dl
{
  counter-reset: dtcount 0
}
```

Code Listing 69c: Resetting our counter

Then we need to increment the counter and add it before the contents of our **dt** element:

```
dt:before
{
  content: counter(dtcount)". ";
  counter-increment: dtcount;
}
```

Code Listing 69d: Incrementing and adding our counter

If you now refresh your browser, you should see that each of our **dt** elements now has a number next to it.

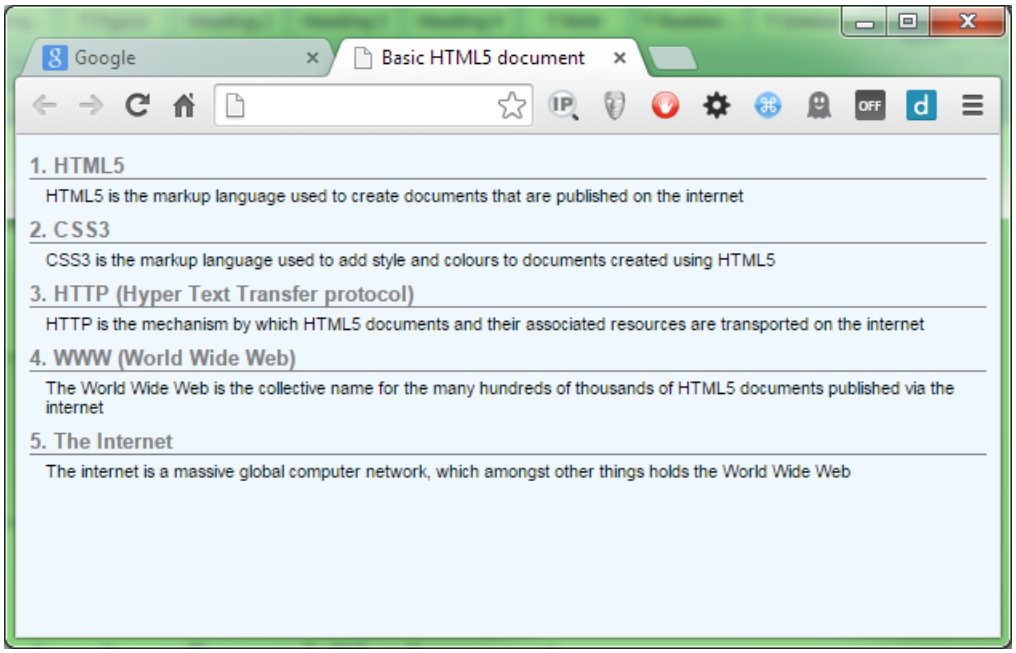


Figure 93: Our definition list with numbers next to the titles

The numbers will increase for however many **dt** /**dd** pairs there are in the list, and will number correctly. You don't have to use the **content:before** pseudo selector, either; anywhere you can use an integer value, you can use the **counter()** function to get the value of your counter. This means you can use it to vary colors, adjust font sizes, increase spacing, and all manner of other things.

What about decrementing a counter? Well, you can do that too, but not quite how you might imagine it.

To decrement a counter, you still use the **counter-increment** rule, but you use a negative offset.

Counter increment rules can take two parameters: the first is the name of the counter to change, and the second is by how much.

Change the code in Code Listing 69d so that it now reads:

```
dt:before
{
  content: counter(dtcount)". ";
  counter-increment: dtcount 4;
}
```

Code Listing 69e: Code from 69d with an offset added

If you render that now, you'll see that your definition lists are now incremented by four each time.

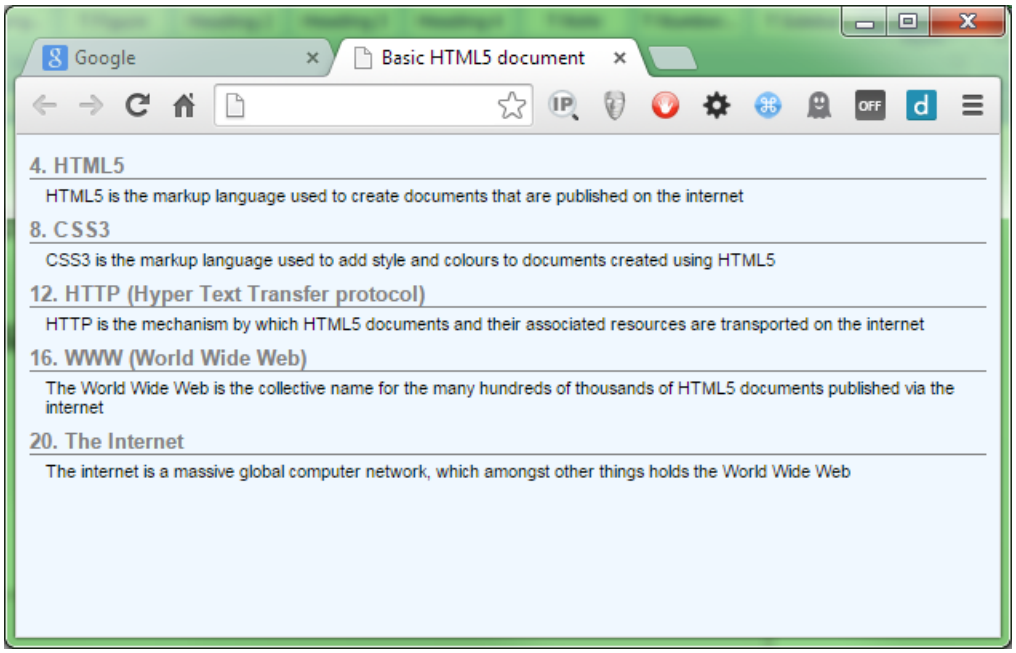


Figure 94: Our counter now increments by 4

Likewise, if you change that 4 to a negative number, then you'll see the numbers next to the **`dt`** elements run backwards.

I'm sure you can think of many more things you can do with these—I'll leave that as an exercise for experimentation.

Calculating Values

One of the biggest problems in CSS over the years has been calculating element sizes accurately. It's mostly because of the lack of ability to do simple tasks like this that there are so many JavaScript UI frameworks and tools such as Sass and Less.

In CSS3, the standards committee has introduced the **`calc()`** function, and as its name implies, this ultra-useful little gem can calculate values for you.

Most of what it can do is really just basic math—it's what you can combine it with that makes it so special.

You can combine values in the **`calc`** function with percentage-based values: em's, px's, pt's, and any other type of value or measurement that CSS understands, and this opens up some amazing possibilities.

Consider the following HTML markup:

```
<div class="header">
    <h1>This is the main Header</h1>
</div>
<p>This is some dummy body text</p>
```

Code Listing 70a: HTML markup to demonstrate the calc function

It's not uncommon to see this type of pattern in most HTML documents, and generally you would mark up the header class so that the header has a fixed size that covers the intended document minimum width using a percentage value, maybe 75 percent.

```
body
{
  background-color: aliceblue;
  font-family: sans-serif;
}

.header
{
  width: 75%;
  height: 60px;
  background-color: steelblue;
  border: 1px solid black;
  color: white;
}
```

Code Listing 70b: Basic body and header rules

The basic rules in Code Listing 70b will give you the following:

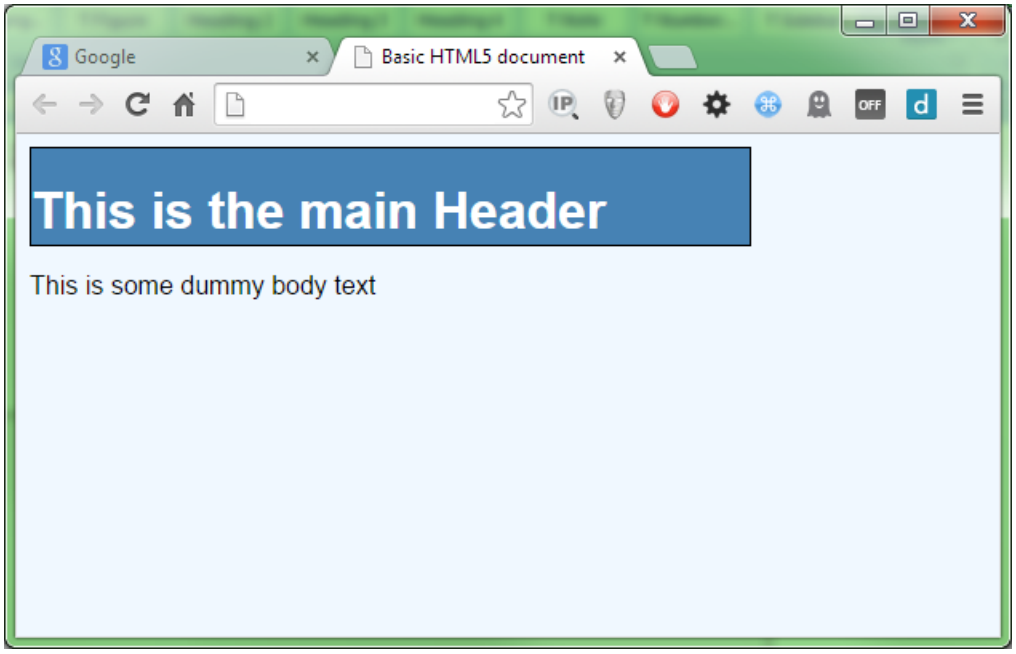


Figure 95: Basic rules applied to our header

Because we've used a percentage value for the width of the header element, it will always be set at 75 percent. That works fine, but what if we want to center the header? That's not difficult, either. Add the following two lines to the .header rule in the basic styles:

```
margin-left: auto;
margin-right: auto;
```

Code Listing 70c: Additional rules to center our header

As expected, this centers our header.

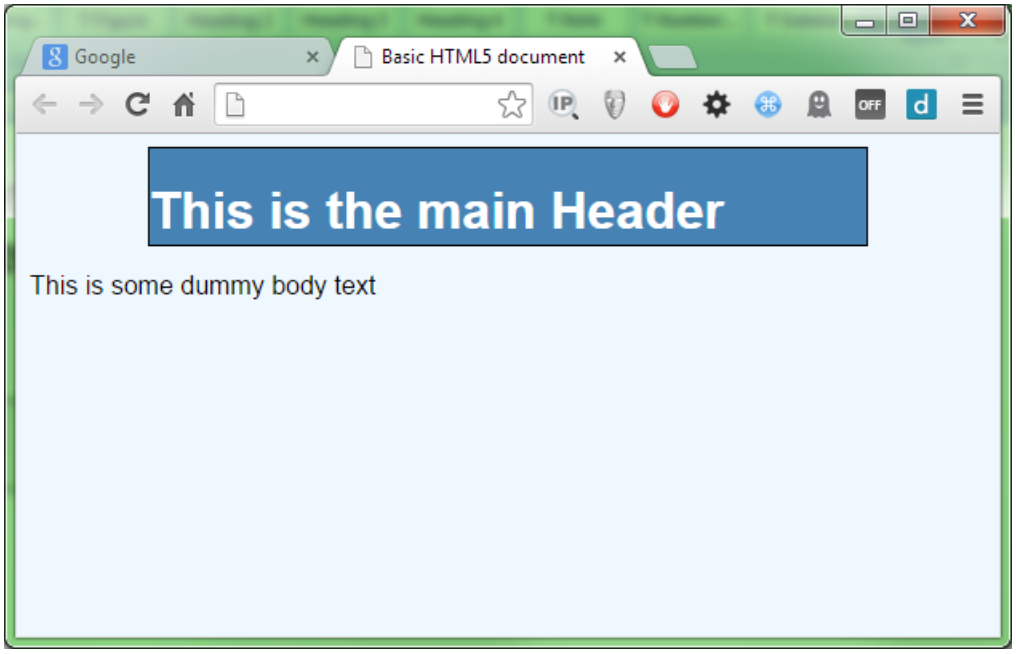


Figure 96: Our header is now centered

Unfortunately, we now have quite a bit of dead space at each side of the header, and because this is a percentage-based measurement, when this gets resized to the size of a wide screen monitor, the gaps will look huge, not to mention how cramped they might look on a mobile device.

You may decide to compensate by changing the percentage value. So you change it to 95 percent, 100 percent, or 98 percent. You might spend quite a few hours adjusting it and then testing it in all sorts of different screen sizes and on different devices, but no matter how hard you try, you just can't get it to look absolutely uniform on every possible device.

If only you had a way of saying, “I want my header to always be 100 percent in width, but to always have exactly 20 pixels worth of space on the left and the right, no matter how small or large my display is.”

This is exactly where the **`calc`** function comes into play.

If you want exactly 20 pixels at either side, then the total of that would be 40 pixels. If you then centered a 100 percent-width object and subtracted 40 pixels, that would give you exactly what you need.

Change the **`width`** rule in **.header** so that it reads as follows:

```
width: calc(100% - 40px);
```

Code Listing 70d: Width rule now using calc to work out its width

When you render it, you should see the following:

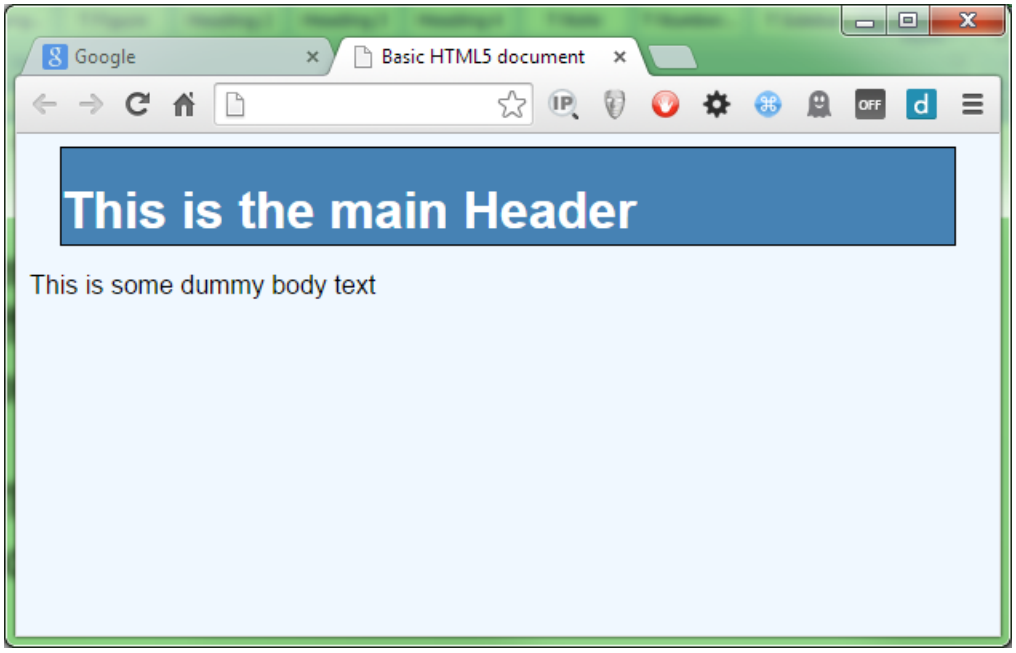


Figure 97: Our header with 20px on either side

Now, it renders with 20px either side, and it doesn't matter what the width is because the CSS engine will always know what 100 percent of the target width is, and so can do the math for us automatically.

Like **`counter()`**, you can use **`calc()`** anywhere you can use a straight forward number. A word of warning, though: not all the browsers implement it in all of the properties. From an implementation point of view, all the current browsers support it, but not all of them support it for things like color values or rotations.

The best place to check is Can I Use, which will list the current compatibility charts for all the new features as they are standardized. One useful trick to know is that you can often type something like **"caniuse calc()"** or **"caniuse web fonts"** into a Google search and get an instant link to the compatibility chart for the feature you'd like to learn more about.

Element Attributes

To round this chapter off, I'm going to introduce the ability to get at an element's attribute in order to use it within the rules you're using.

If you recall, earlier in the book I mentioned something called data attributes; one of the great things about data attributes is the ability to embed custom attribute data in an HTML 5 element and still have it validate.

The following HTML shows how this might be used:

```
<div class="header">
    <h1 data-headersuffix=" (Hello)">This is the main Header</h1>
</div>
<p>This is some dummy body text</p>
```

Code Listing 71a: HTML markup to demonstrate parent attributes

If you render this, you'll get exactly the same output as seen in Figure 97 (assuming you still have the same style rules present). If you don't have the same styles present or have started a new file, then you'll just get a plain black text on a white background.

The point here is that these extra data attributes will have no meaning on anything in the HTML or the CSS, and will be silently ignored.

This really comes in handy—especially for a lot of the JavaScript libraries available—because it means that an HTML developer can add custom attributes to a page, which the JavaScript code then later processes in some way. The Bootstrap UI framework makes extensive use of this facility so that the HTML developer creating pages never needs to know any JavaScript. He or she just has to describe what the intention is, and the Bootstrap library takes care of the rest.

Make sure your style file is the same as Code Listing 70b, and that you're using the HTML code in Code Listing 71a. Render the HTML in your browser and make sure your output looks like Figure 95.

Now add the following extension rule to the style sheet (you should see the header text change to blue):

```
.header h1:after
{
  content: " (Hiya)";
}
```

Code Listing 71b: Extended header rule that puts "(Hiya)" after the header text

Render this and make sure that the header text now has "(Hiya)" appended to it. If it does, now change the rule so that it looks for the data attribute named **`headersuffix`** and uses the parameter from that as the source of its data.

```
.header h1
{
  color: attr(data-headersuffix);
}
```

Code Listing 71c: Extended header rule that uses the data attribute to set the text

The result should be that the header text is now extended using the contents of the custom data attribute present on the **`h1`** element that the selector targets.

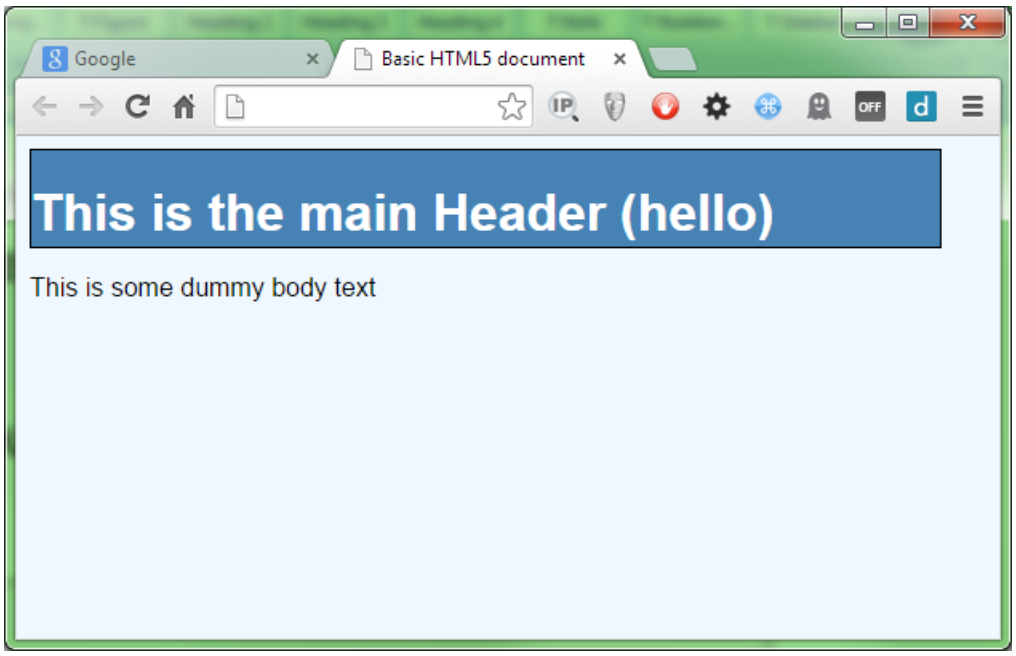


Figure 98: Our header with the text from the data attribute

There is, unfortunately, some bad news. If you read the spec closely, you'll see that the ability to read parent attributes and place them into generated content fields was actually defined in CSS2.1, and the ability to use the **`attr()`** function to set properties, such as color, size, etc., was defined in CSS3. As a result, all major browsers can use **`attr()`** to set content in a :before or :after pseudo selector, but none of them as of yet have implemented them in a general property setting scenario.

This is not to say they won't; as many of you might know, the most popular features on the W3C standards track are the ones that developers request, and **`attr()`** is not getting much attention at the moment. There will come a day when this is finalized and the browser producers feel it's getting enough attention to actually finish implementing it. For now, it's only useable for content generation, despite what the documentation says.

It is useful for one scenario, however (and one I've previously used it for), and that's to append greetings onto headers and welcome banners that are localized for a given locale. Because the value is taken from an attribute, changing that attribute is very easy to do in JavaScript, as there are native calls to access data attributes.

Another place it becomes useful is in printing. If you have a style sheet that's used only when a page is printed, you can use this to grab the contents of an **`href`** from an anchor element, then print it out after the anchor tag. Doing this means that pages with links printed to a PDF or onto paper are still visible, while the ones on screen remain clickable.

Summary

In this chapter we've learned how to generate arbitrary content using nothing more than CSS rules, a task that previously required a large amount of JavaScript to get right. We've learned a bit more about how data attributes work and how we might use them to hold custom content.

In the next chapter we'll do a roundup of some of the things we didn't have space to cover, and a few of the smaller, new rules.

Chapter 9 The Mixed Bag of Tricks

And so we come to the last chapter of the book.

It's always sad when you reach the end of a book like this, especially on a subject as diverse as CSS3, but this is not the end. In fact, we couldn't be further away from the end—we're still just beginning this adventure.

What I've shown you so far is only the tip of the iceberg. If you're in any doubt, take a look at the main CSS index on the Mozilla Developer Network. I haven't even begun to explore CSS Layout or CSS Animation yet; both of these topics would easily take up a full book all of their own, as would an exhaustive rule/property followed by description list style text.

However, books in the *Succinctly* series are designed to be compact and easy-to-read with just enough information to get you in the right direction, and it's always a difficult decision to decide what to leave out.

As with any new technology, the best way to learn it is to play with it. Playing is an invaluable skill and learning tool that often gets lost in the modern-day IT industry with tight deadlines and tricked-out release schedules. What I hope I've presented here is enough information to at least prevent you from having to waste time understanding what it is you're playing with.

For the rest of this chapter, I'm going to round things off with a few notable mentions of easy-to-understand CSS3 functionality that I felt deserve a mention.

CSS Columns

We've all been there, tearing our hair out trying to get the column widths just perfect so we can get a nice and balanced multiple-column layout in our page design.

Hours are spent tweaking, measuring, and tweaking again until we get an almost-perfect combination of rules and content. Then someone decides to drop an image in the sidebar that pushes things out by 1 pixel, and BAM! The entire layout crumbles into a smoldering heap.

Well this should no longer affect you, thanks to CSS columns.

Create a **`div`** with a very long paragraph of text, something like the following:

```
<div class="container">
    <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas
pellentesque urna nec eros ornare, ac tristique diam porta. Donec fermentum velit
eget dignissim condimentum. Sed rutrum libero sit amet enim viverra tristique.
Mauris ultricies ornare arcu non adipiscing. Sed id ipsum vitae libero facilisis
pulvinar id nec lacus. Ut lobortis neque et luctus mattis. Morbi nunc diam,
elementum rutrum tellus non, viverra mattis diam. Vestibulum sed arcu tincidunt,
auctor ligula ut, feugiat nisi. Phasellus adipiscing eros ut iaculis sagittis. Sed
```

```
posuere vehicula elit vel tincidunt. Duis feugiat feugiat libero bibendum
consectetur. Ut in felis non nisl egestas lacinia. Fusce interdum vitae nunc eget
elementum. Quisque dignissim luctus magna et elementum. Cum sociis natoque
penatibus et magnis dis parturient montes, nascetur ridiculus mus. Sed nunc lorem,
convallis consequat fermentum eget, aliquet sit amet libero.</p>
</div>
```

Code Listing 72: A nice, long paragraph of text inside a div element

If you render this, you should see something like:

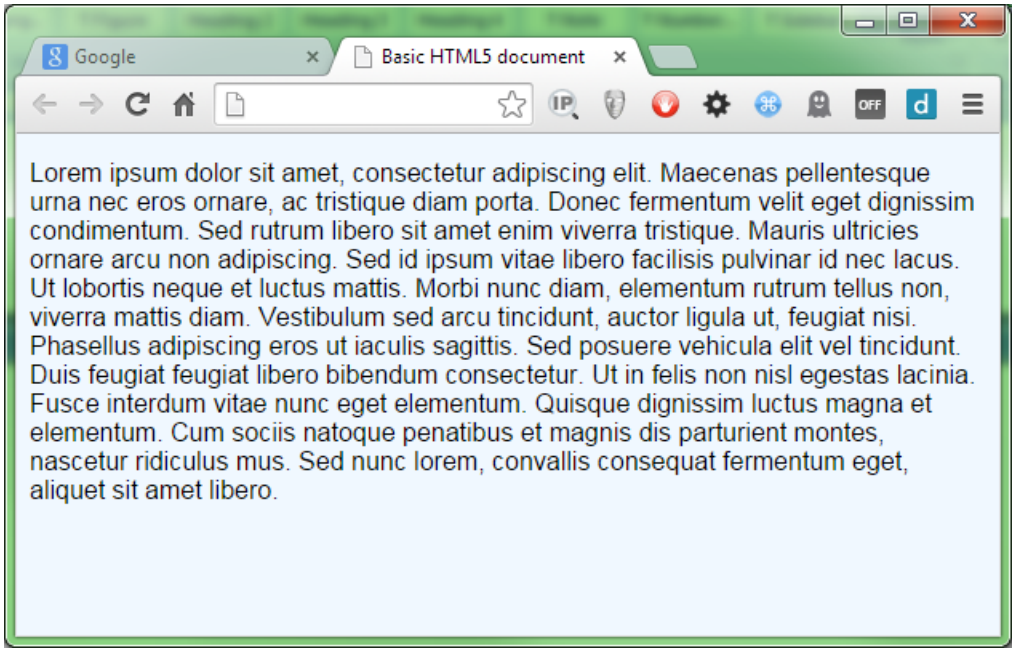


Figure 99: A rather large paragraph with default styling

It's certainly not the nicest of layouts, but if you resize the display, it'll flow and word wrap correctly. If you wanted to split it up into three columns (like a newspaper might), you'd have a rather difficult job on your hands.

You'd almost definitely need to use JavaScript, because you'd need to work out how much text could fit into each box. Then you'd need to make sure the columns were resized appropriately, a task which **`calc()`** could help you with—but it's still a lot of work.

Instead, you could use the CSS3 columns rule. Add the following two rules to a style sheet and attach them to the previous HTML.

```
.container
{
  width: 500px;
  margin: 0 auto;
}
```

```
.container p
{
  -moz-columns:3;
  -webkit-columns:3;
  columns:3;
}
```

Code Listing 73: CSS rules to enable multi-column layout

Rendering this instantly takes care of the layout and the flow for you. Notice, however, that this rule still requires vendor prefixes, and while the general layout works, some of the extended rules in the module are either incomplete or not yet implemented. You can find the full details here.

If you have a compatible browser, then rendering the HTML in Code Listing 72 with the rules in Code Listing 73 should give you this:

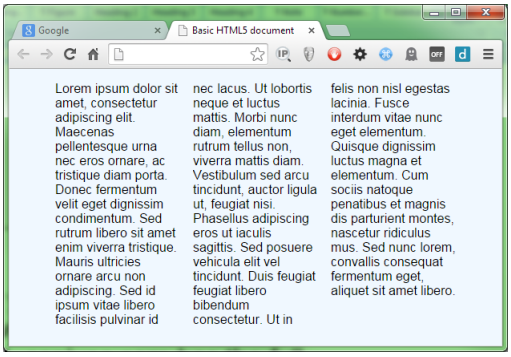


Figure 100: Multi-column layout made easy

CSS Transformations

While transformations are yet another topic that could fill an entire book, there are some basics that are easy to experiment with. Create an HTML document and add the following markup and CSS rules to it:

```
<div class="transDiv">
    <p>I'm a div ready to be transformed.</p>
</div>

body
{
  background-color: aliceblue;
  font-family: sans-serif;
}

.transDiv
{
  width: 200px;
```

```
  height: 200px;
  background-color: beige;
  border: 1px solid black;
  padding: 4px;
  margin-left: auto;
  margin-right: auto;
}
```

Code Listing 74: HTML and CSS rules to create a beige, centered div

When rendered, this should give you the following:

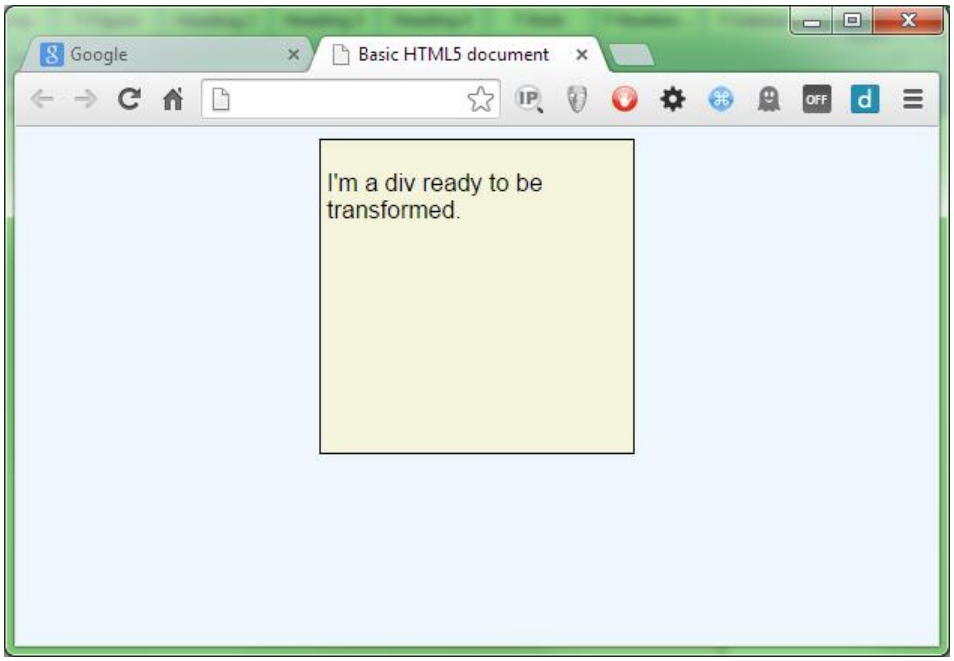


Figure 101: It's just a standard, beige div

Now add the following extended rule to the style sheet and re-render the output:

```
.transDiv
{
  transform: rotate(45deg);
  -webkit-transform: rotate(45deg);
}
```

Code Listing 75: Extended CSS rule to rotate our div

You should have a **`div`** rotated at 45 degrees:

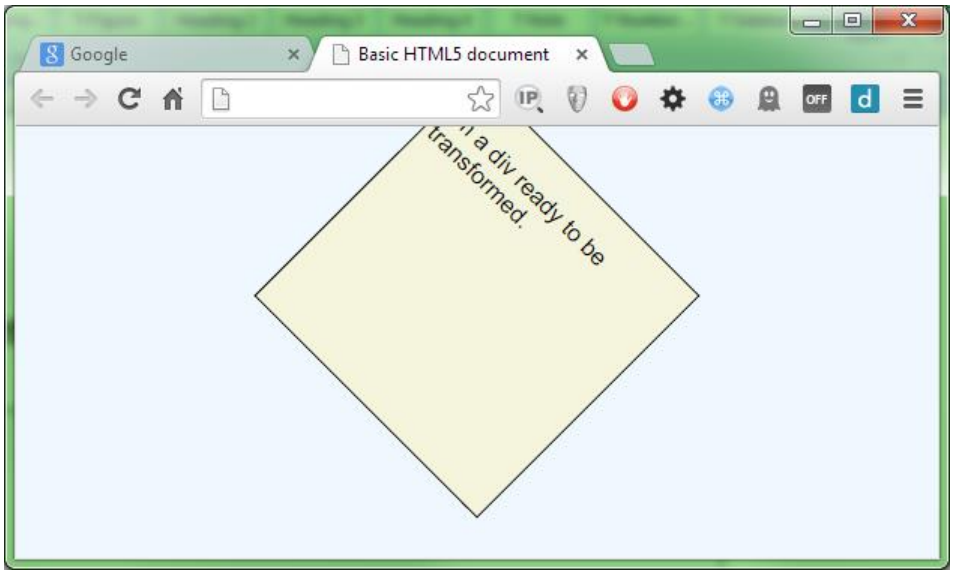


Figure 102: Our div is now rotated.

As with CSS columns, there is still one case where vendor prefixes are needed, and that's on Safari. IE, Firefox, and Chrome all work without needing prefixes in current versions.

You can also apply the following:

- **`scale(x,y)`**
- **`scalex(value)`**
- **`scaley(value)`**
- **`skewx(value)`**
- **`skewy(value)`**
- **`translate(x,y)`**
- **`translatex(value)`**
- **`translatey(value)`**

The actual definition of each function's purpose can be found on MDN, but scale allows you to grow and shrink an element, skew allows you slope, an element and translate allows you to move an element. There are others that allow you to produce all kinds of great effects, and many of these properties can be animated, too, allowing even more to be achieved.

There's still more, such as multiple backgrounds, layered elements, 3D transforms, flex boxes, CSS shapes, and the list goes on.

For now, however, I hope you've learned a little of the power you now hold in your browser. When this is combined with the new element types, layout possibilities, and JavaScript API's available in the full HTML 5 spec, it's easy to see how far the HTML platform has progressed.

It wasn't that long ago that most websites were still fairly static and black and white; now we're starting to see some truly amazing visual affects being made possible, with developers every day finding new and engaging ways of bending this functionality.

It's a platform that's only going to get better, and one that's destined to become the universal user interface for most software. CSS3 is something that every web developer needs to learn both now and for the future—if you don't, then you risk getting left behind.