3
CSS

In the previous chapter, we learned how to create HTML documents using HTML elements and attributes. We can even include images and links to other documents and images. But when you look at the result on a screen, you are probably disappointed. I hope you are, because that was done on purpose (oops, I almost said by design, but the design part is what this chapter is all about). When I wrote my first web pages, I was disappointed too, in particular when discovering how hard it was to do something that should be simple: putting a photograph on a page and some text right next to it. Well now, it's time to turn disappointment into excitement!

In this chapter, we will learn how to add the presentation part—in other words the layout—to our web pages, using **Cascading Style Sheets** (**CSS**). Style sheets are a common feature in Desktop Publishing software. They allow you to specify (or modify) the style of a section of a certain kind in your document: for example, every paragraph of text. When I developed my first book in Adobe InDesign, I knew exactly what I wanted every component to look like, so I modified the letter type and size of all of them by hand. I did not want to spend the time learning how to create style sheets. I have, however, since regretted that decision as at that point it had become a time-consuming affair.

Today, I love style sheets and not only do I recommend using them but to let them be the first thing that you create when you start a new project. A style sheet is like a plan for a plan, where you can fill out the details later.

By using style sheets you can separate the design part from the content part. You can even have this done at separate times or by separate people and it will give all of your pages a consistent look and feel. Simply switch two style sheets and your entire site will look completely different. Are we excited now?

Let us start with a sample piece of CSS code:

```
/* selector - by the way this is how to do comments in CSS */
p.red
/* properties */
{
color:red;
font-family:baskerville, cambria,  serif;
font-size:12px;
font-style:italic;
}
```

Comments in CSS can be found in-between the strings `/*` and `*/` similar to that used within the C programming language. So, to encourage good behavior, we included some comments in our first example. Let's analyze the rest of this code.

The part before the curly brace is called the **selector**. It represents one or several elements in our page. In our example, that would include all `<p>` elements with class red.

In-between the curly braces, we find the CSS rules we want. In this example, we want all text inside paragraphs to be red, size 12 pixels, in italics, and in the **Baskerville** typeface. On systems where that font is not available, we want **Cambria** to be used instead.

Note that every rule ends with a *semicolon* and consists of two parts separated by a colon: a **property** and a **value**. Color is a property and in our example the value chosen is `red`.

In old versions of HTML, the same could have been achieved by placing `<font>` elements inside your `<p>` tags. Imagine having 40 `<p>` sections in your document and someone wants to change the `red` into `maroon`! You would have to change your HTML file in 40 different places, and no- a global find and replace would not even help you, as there might be other red "things". By using CSS, you only need to change one line.

It is more common to find CSS code similar to this one:

```
p {
font-size:12px;
font-family:Baskerville,cambria, serif;
}
p.red {
font-style:italic;
color:red;
}
```

This will not only have the same effect for red `<p>` s , but it will also put all other paragraphs in the same typeface and size. This is the C in CSS: the properties of the overall `<p>` flows into the subset of red ones, like in a real cascade.

Adding styles to our documents

So how do the CSS rules become part of our document? There are three ways:

- External style sheets
- Internal CSS
- Inline styles

External style sheets

This is the recommended way to add CSS to your site. It should be your goal for all style sheets of the production version of your site to be external. Simply add a line to the `<head>` section of your site, which should look like the following:

```
<link rel="stylesheet" type="text/css" href="css/style.css">
```

`<link>` is an HTML element we had not yet introduced. Its attributes are listed below:

- `rel`, to indicate the relationship between the HTML document and the linked file.
- `type` specifies the MIME type of the document so the browser knows how to load it.
- `href` is used to specify the location of the file. You may expect a `src` attribute here, like is used for `<img>` tags, but the attribute to specify the file name in a `<link>` element is `href`. For the file name, we recommend that you always use relative pathnames. We suggest that you collect all your style sheets together in a folder with a meaningful name, like `css` or `styles`. Of course, the file itself should have a meaningful name too.

When the file does indeed exist, it will be loaded. That is why it is important that your `<link>` element resides in the `<head>` section of the document so all the CSS rules are read before the body of your document.

Internal CSS

For small projects, or projects you would like to limit to a single HTML file so that it is easy to email to someone, you can use internal CSS. All the CSS rules can then be placed inside the `<head>` section of your document, inside a `<style>` tag. That tag needs to, at a minimum, include a `type` attribute, as in the following example:

```
<style type="text/css">
p.red {
color:red;
}
</style>
```

Inline styles

Styles can be given to an individual HTML element by using the `style` attribute inside the HTML element itself, as in the following example:

```
<h3 style="color:green;">Congratulations</h3>
```

We do not recommend using this in your final product, but it is extremely useful to instantly see the effect of a change during development. If, for some reason, you leave one of the inline style attributes inside a page, it might take you forever to find out why your cool style sheet is not doing what it is supposed to be doing, on this one line of this one page.

On the other hand, I use it everyday, as I introduce and test new elements on a page while not disturbing anything else on the site, as modifying the external `.css` file would.

The Document Object Model (DOM)

As we learned in the previous chapter, an HTML document consists of a tree structure of nested tags, with HTML as the root. In programming, the contents of that tree can be stored in a large object and subtrees can be accessed, modified, or added by using smaller objects.

The whole lot is referred to as the **Document Object Model** (**DOM**). In subsequent chapters, we will learn a programming language (JavaScript) to do exactly that and a JavaScript library (jQuery) to make it easier. In this chapter, we will not be learning how to change our content, but we will learn how to change the style of our content. In all cases, we need a way to access our document. This is where **selectors** fit in.

Selectors

The first part of a CSS rule, the part before the opening curly brace, is the selector, or several selectors separated by commas. A selector represents a *collection* of elements in the page to which the subsequent rules apply. The simplest selector is a single tag name, such as the one we already used in a previous example. Following is a code snippet as another example:

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

The selector `p` means all paragraph elements in the entire page. Applying this rule will result in all paragraphs in the entire page being rendered in blue. Similarly, we could use a class name. Refer to the following example:

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

The selector `.blue` represents all the elements in the page that have the class `blue`, whether they are paragraphs, headings, or so on. Now we can combine the two, as shown here:

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

This selector represents the collection of all paragraph elements on the page with the class set to blue. For those of you that like set theory, this is the intersection of the `p` collection and the `.blue` collection.

Let's go for some more set theory in the next simple example:

```
#errorbox {
color:red;
}
```

The set of elements that matches this selector is, at best, a singleton, as it matches the element with its **id** set to `errorbox`, if present. I cannot remind you often enough that no two elements can share the same id. Equally valid, but slightly more restrictive, is the following rule:

```
div#errorbox {
color:red;
}
```

The former rule was about any element with an id `errorbox`: the latter only applies to a `div` element with the id `errorbox`.

Multiple classes

We learned that several different classes can be assigned to an element by assigning a space separated string to its `class` attribute, for example:

```
class="green cool awesome"
```

If you want to create a CSS rule for an element like this, the selector could look like the following:

```
h2.green.cool.awesome
```

When used as a selector in a CSS rule, the rule would apply to all `h2` elements of class `green`, as well as `cool` and `awesome`. So the style would be changed to the following:

```
<h2 class="green cool awesome">The Hulk</h2>
```

This is completely different from a rule with the following selector:

```
h2  .green .cool .awesome
```

This would be part of a rule applying to elements that have the class `awesome`, are descendant of elements with the class `cool`, which are in turn descendants of tags with the class set to `green`, while being themselves a descendant of an `h2` element.

This is a very important distinction, so it is very important to remember: if you allow me to paraphrase a famous song: *What a difference a space makes*!

Descendants

The sample selectors we used so far are all top level selectors: an element, a class, or an identifier. Now we are going to add more detail, and more complexity. Look at the following example:

```
#container h2 {
color:grey;
}
```

This rule applies only to the `h2` elements that are inside an element with the `id="container"`, no matter how many levels deep they are. So if this was a chunk of your HTML file, the rule would apply to the `h2` element that contains it:

```
<div id="container">
<div id="header"></div>
```

```
<div id="main">
<h2>Title of this Document </h2>
</div>
</div>
```

The following CSS rule will make the heading `grey` as well, but if by any chance there are other `h2` elements inside the `#header` div, they would not be affected by this rule:

```
#container #main h2 {
color:grey;
}
```

Selecting children or siblings

In some cases, you want to be more specific in your selection and have a rule for elements that are the **children** of other elements, not descendants of arbitrary numbers of differing levels down in the document tree. For this purpose you can use the **child selector**, as in the following example:

```
#main >  h2 {
color:grey;
}
```

`h2` is a child of the div element with the id `main`, so with this rule also, the heading in our little chunk of code will be displayed in `grey`. Now consider the following example: On the other hand:

```
#container > h2 {
color:grey;
}
```

In this case, there will be no effect as `h2` is not a child of `div#container`.

There is a similar syntax for specifying an element that is adjacent to another one, or a **sibling** as this would be called in a family tree. Refer to the next example:

```
h2 + p  {
margin-top:0px;
}
```

This would not give a top margin to a `p` element that immediately follows a `h2`, but would not apply this rule to subsequent `p` tags. Look at the following small piece of HTML:

```
<h2>Story</h2>
<p>First paragraph</p>
```

```
<p>Second paragraph</p>
<p>Third paragraph</p>
```

The first paragraph would not have a top margin; all others would have margins based on how the browser renders the paragraphs by default. Of course, this is true if this CSS rule is the only one and not overruled by others, and this brings us to the next topic: **specificity**.

Specificity

From time to time, as a CSS beginner or an experienced web developer, you will be frustrated if you just added a new CSS rule to your style sheet and discovered that it has no effect. More often than not, this happens because there is another rule with a higher specificity that has priority.

We already mentioned that inline styles take precedence over external style sheets. So it seems more than logical that the order of internal CSS and links to external style sheets will influence how the page will end up looking. Now consider the following rule:

```
p.warning {
color:red;
}
```

Suppose the previous stated rule is followed by this next rule:

```
p.warning {
color:orange;
}
```

In such a scenario, all paragraph elements with the class `warning` will appear in `orange`, not `red`, because the orange rule appeared later.

But these two rules happen to share a common selector: it is a `p` with the class `warning`.

Things would be different if you had rules with selectors such as the following:

```
p#error  {
color:red;
}
```

And along with that, if you also had the next rule:

```
body div.container div.main p.error {
color: orange;
}
```

Let us now consider the following:

```
<div class="container">
<div class="main">
<p id="error" class="error">
What you typed is incorrect
</p>
</div></div>
```

Logic or intuition, but not the order in the CSS, would make us think that the text `What you typed is incorrect` would be displayed in red and not in orange. This is, in fact, correct, but we do not want to rely on intuition, do we? Fortunately, there are formulas to determine which CSS rule wins if there are several that could influence the layout of a particular element. It is slightly mathematical and referred to as **specificity**.

The specificity of a CSS rule is a sequence of four numbers that are calculated as follows:

- If the rule is an inline style, the first number is 1, otherwise it is 0
- Add 1 to the second number for every occurrence of an **identifier**
- Add 1 to the third number for every **class** specified
- Increase the fourth number by one for every **element** present

When two rules are compared, the first number is looked at first. If one of them is higher, then that rule has more **weight**. Next, the second number is looked at, and if needed, the third, and finally the last. The specificity of our two sample rules is: 0,1,0,1 and 0,0,3,4.

So our intuition is now confirmed by mere arithmetic.

Block elements and inline elements

Before we finally get into a description of the most important CSS properties by category, we need to say a few words on two categories of elements: **block** elements and **inline** elements.

Think of block elements as rectangular areas of your screen or page. They can contain text, data, and other block elements, as well as inline elements. Typical block elements are the `<div>` and `<p>` tags. Before and after every block element, a new line of text is created.

Inline elements can only contain inline elements and not block elements. Also, they cannot be given a width. They inherit the width from the container that they are inside of. The most popular inline element is the `<span>` element, typically used to change the look of a chunk of text inside more text.

For starters this can be confusing, because block elements like `<div>` may be block elements, but when they have no content inside of them, they appear not to have any width or height. Moreover, you can change the way these elements are displayed with the CSS `display` property. Consider the following lines of code and have a browser render it:

```
<p>paragraph1</p>
<p>paragraph2</p>
<p>paragraph3</p>
<p>paragraph4</p>

<p style="display:inline;">paragraph1</p>
<p style="display:inline;">paragraph2</p>
<p style="display:inline;">paragraph3</p>
<p style="display:inline;">paragraph4</p>
```

The first four lines are displayed as block elements and are true paragraphs: there will be a new line in-between them, and the distance between these paragraphs will be determined by the default `font` and `margin` values the browser has chosen. Font and margin belong to the most important CSS property families. We will learn about them when we discuss the font properties and the so called `box` model for `margin`, `border`, and `padding`.

The second set of paragraphs will all appear on one line, if permitted by room in the viewport, because we explicitly declared them to be inline elements.

Before we discuss the box model, let us look at one more example. Experiment with the first half set of divs as it will, as is, display nothing. Once you insert text in between the `<div>` tags, that text will appear along with the background colors. The `<div>` elements in the second set actually have a width and height, so you will see four perfect colorful squares, but maybe not the way you expected them. We grouped them two by two, yet they all appear stacked on top of each other. Welcome to the world of browser indifference!

```
<style type="text/css">
.redbg {
background-color: red;
}
.greenbg {
background-color: green;
```

```
}
.yellowbg {
background-color: yellow;
}
.bluebg {
background-color: blue;
}
.sq200 {
width: 200px;
height: 200px;
}
</style>

<div>
<div class="redbg"></div><div class="yellowbg"></div>
</div>
<div>
<div class="greenbg"></div><div class="bluebg"></div>
</div>

<div>
<div class="sq200 redbg"></div><div class="sq200 yellowbg"></div>
</div>
<div>
<div class="sq200 greenbg"></div><div class="sq200 bluebg"></div>
</div>
```

Colors

What can make a site look instantly more pleasant is the proper use of colors. It is also a useful tool for doing some debugging. By adding different background colors to some block elements, you are increasing your chances of finding the location of that one missing, closing tag that messes things up. I just used it today.

There is a world of difference between publishing printed books, documents, and even packaging components or media faceart through a professional manufacturing company, and web publishing. In the world of printing, you have full control over the colors you use and, also extremely critical, the fonts or typefaces. You should be very exigent and expect the resulting product to match exactly in color and typeface to what you specified. Color information is exchanged in either RGB (**red-green-blue**) or CMYK (**cyan-magenta-yellow-black**) values. In CSS also, you can specify the desired color through its RGB values. But this is where the comparison begins and ends.

Because of this, we know that colors of a quality printed item should look exactly the way we expect them to. However, when we prepare web content, we cannot tell how the colors on our site will look to our visitors. It depends on too many things. For example, what device do they use: a computer, a tablet, or a mobile phone? The screen they look at can be large or small, be of high or low resolution, and support millions of colors or just a small number. Once we realize that, we are fine. I therefore recommend using colors that are sufficiently distinct, and to not bother using extravagant RGB combinations.

Colors can be specified in several ways. The two most common ones are by **name** and by **RGB** value. Names are easier, but may be more subject to browser interpretation. So I only use them for quick and temporary things, or when they are *black* or *white*. In all other cases, I prefer to use the RGB values, which are written as the # sign followed by three two digit hex numbers, for example, `#FFDEAD`, one of my favorite colors. It is called **Navajo**.

You can use colors for the foreground and the background, with the **color** and **background-color** properties. Color indicates the color that will be used to display the text inside an element or the descendants of an element; background-color, as the property name suggests, will set the background color of a block element.

Fonts

I am a typography enthusiast. I even spent money buying fonts, just for personal use. So I would like to point out that, even more so than with colors, there is a huge difference between using typography for print publishing and web publishing.

In print publishing, you are in full control of `fonts` or `typefaces` (this is not a typography book, so allow me to use these terms interchangeably) that you use in your work. You just have to make sure that all the fonts that you plan to use are installed on the system that holds your `Desktop Publishing` program, so you can use them for your development. Next, make sure that they are **embedded** in the final document you send to your printer, and that both you and your printing company have a legitimate license for all the fonts used. Then all the people who will read the printed copy or even a PDF version online, will see it just the way you designed it.

When publishing on the web, the text that the visitor will see on your page can only be displayed in a font that is actually installed on the device he or she is using. The worst case, yet the best looking, scenario is the one where you use a very nice, expensive font that is installed on the system used to design and test your site. It is possible that is it installed on no other computer on the planet.

That website will look absolutely fabulous on your own computer but potentially dreadful where it matters most. That is why in CSS we work with **font families**, and not single fonts.

Before getting into more detail, one thing to note about the web **middle ages**, is that there used to be a tag `<font>`—simply think—ah! middle ages and do not use it. It is one of those presentational HTML elements we talked about.

So what are fonts?

In simple terms, fonts are a series of pictures, or **glyphs**, determining how letters, numbers, and other characters should be displayed. In the early days of printing, some 500 years ago, Gutenberg, Plantijn, or Moretus created metal casts, one letter at a time. These metal letters would be placed in wooden cases, the larger ones in the upper and the smaller in the lower case. This is how the terms uppercase and lowercase were born.

To compose a single page of text, metal letters had to be put in a frame to hold them together and then inserted in the world's first printing press. Add some ink and paper and there you had it: you had one page of one copy of a book. Of course, for every different size, thickness, and style (for example *italic* versus *normal*), there had to be a separate case with letters.

The people who created those metal letters were called punch-cutters and the fonts were named after them. A famous one was Claude Garamond. You can spot the original Garamond metal fonts, wrapped in paper to protect for posterity, at the Plantijn-Moretus Museum in Antwerp (it is my favorite museum in Belgium). You will discover there that the 16th century equivalent of a combined architect, portraitist, and "web designer" was Peter Paul Rubens, known to most other people only as a painter.

Today, the fonts we talk about are no longer small blocks of metal, but computer files with information on the font layout. The more glyph collections a font contains (more sizes, styles, or weight or thickness), the more accurately a letter will be reproduced.

Font families

Fonts can be divided into different categories or families. The three most important ones are called **serif**, **sans-serif** and **monospace** fonts.

Serif fonts

Serif fonts are typefaces that have glyphs with small decorations or serifs at some of the edges of the letter. For example, the short lines at the bottom of each leg in the letter *m*.

The font designed by Garamond, mentioned in the previous section, is an example of a serif font. Serif fonts are popular because they make reading text pleasant. Publishers use well crafted serif fonts, combined with high quality printing paper, in the hardcover versions of their books. **Times New Roman**, designed for a British newspaper The Times, is a very commonly used serif font. The very text you are now reading is in Times New Roman. The creators used a font called Plantin as a model, which in turn was named after **Plantijn,** the printer person I mentioned earlier.

Baskerville is another example of a serif font and is used a lot for eBooks.

Using serif fonts for the text portion of a website is subject to discussion. As glyphs have to be displayed using pixels on a screen, the decorations that cause a pleasant read in print may result in just the opposite on a screen,. This is because on lower resolution screens, those same decorations may look too pixilated. I personally recommend at least experimenting with some serif fonts for the main text part of your site before making that call.

Sans-serif fonts

Sans-serif fonts are the counterpart of serif fonts. They do not contain the small decorations or serifs at the end of the stroke, hence the name sans (French for without) serif. In print, sans-serif fonts are used for headings, titles, and so on. This is in contrast to the serif fonts used for the body of the text. Sans-serif fonts, for the reason mentioned previously, are now more common for text that needs to be displayed on computer screens. Common sans-serif typefaces are: **Arial**, **Open Sans,** and **Helvetica**. Headings in this book are in Arial.

Monospace fonts

In the typefaces we have discussed so far, not all characters have the same width: the letter `m` is clearly wider than the letter `i`. A monospaced font, also called a fixed-width, is a font where letters and characters do indeed have the same width. The first monospaced typefaces were for typewriters, as the carriage always moves the same distance forward with each letter typed. They were also used in early computers and computer terminals. Software text editors still use them today, as it makes it easier to align source code when every character has the same width.

The source code displayed on web sites is usually displayed in a monospace font, as are the code examples in this book. The most common monospace font is probably **Courier**. Another example is **Lucida Console**.

Let us now get back from our trip into the science and the history of typography, to our CSS story.

The font-family property

To specify how the text of an element needs to be displayed, we use the **font-family** property in the CSS style sheet. For example, consider the following:

```
p {
font-family: "Open Sans", "Helvetica Neue", Verdana, Helvetica, sans-
serif;
}
```

So why is there more than one font specified: a letter can only be in one font, can't it?

This is how it works. When someone visits your site, with this sentence in your style sheet, a paragraph of text will be displayed in the `Open Sans` typeface, if it is installed on the visitor's computer. If it is not found, `Helvetica Neue` will be looked for; if that one is not there then `Verdana` is attempted, and so on. If everything fails, some default sans-serif typeface will be used. For this example, the last scenario is very unlikely, as I cannot imagine a system that would not have Verdana or Helvetica installed. Yet, it is recommended to always conclude the list with sans-serif, serif, or monospace. Note the use of quotes (For example, for 'Helevtica Neue') when font names consist of more than one word.

Font-weight and font-style

When you select a font using one of the desktop applications you use, you can also select style from `picklist`. A typical `picklist` could include: normal, italic, semi-bold, semi-bold italic, bold, and bold italic. The number of items on that list typically matches the number of glyph collections that came with the font. You can visit a website of a company that sells fonts, to get a better feel for how many variations there can be on the theme of a single font.

In CSS, the equivalent of this is provided through two different properties: **font-weight** and **font-style**.

The two most common values for font-style are **normal** and **italic**. The ones for font-weight are **normal** and **bold**, and the numbers `100`, `200`, and so on, through `900`. `400` is the same as normal and `700` the same as bold. All others can have unpredictable results depending on the presence of, let's say, a semi-bold version of your font and the browser used. So until we reach the second part of the book, the message should be loud and clear—only use the basic normal or italic, and normal or bold.

Font-size

Finally, as in our desktop applications, we can specify now which font our text needs to be displayed in, and what size all these letters should have.

Size can be specified in `pixels`, `percentages`, or `ems`. Another one that exists is called `rem`.

When no font-size is specified anywhere, a typical browser will set that size to `16px`. The `em` unit, a term that comes from typography to refer to the size of the letter `m` in a font, is, in CSS, nothing other than the calculated size of the font for the current element. Its use is highly recommended over fixed pixel sizes. Let's illustrate this with an example. Look at the following CSS code:

```
h1 {
font-family: "open Sans", Arial, sans-serif;
font-size:2em;
font-weight:bold;
}
h2 {
font-family:Arial, sans-serif;
color:#999999;
font-size:1.5em;
font-weight:600;
}
```

If no font-size was specified for the body element, we know that this would be `16px`, with all descendants inheriting it. This means that our `h1` heading would become `32px` and the `h2` would be `24px`. Simply changing the font-size for the body element to, let us say, `20px` would proportionally change the sizes of our `h2` and `h1` headings to `30px` and `40px` respectively. If we had given them a size of, for example, `20px` and `24px` we would have ended up with headings the same size as the font used for regular text.

The same is true when the user uses the browser to zoom in or zoom out. We retain the proportions. In the interests of creating responsive designs, using proportional sizes rather than fixed ones is the way to go. Of course you have to be careful and realize that when you change the font-size of an element, all children will inherit it, and when you change that, by mistake or not, of a child element, the size of `em` will no longer be what it was a minute ago.

Consider the following code:

```
<div class="insert">
<p> A paragraph of text that represents what could be an insert in a
book</p>
</div>
```

Following are the CSS rules to go with it:

```
div.insert {
font-family:Baskerville, "Times News Roman", serif;
font-size:0.8em;
color:brown;
}
.insert p {
font-size:0.8em;
}
```

You have just made the font-size twice as small, so that the letters in this paragraph of text will be `0.64`.

In general, I like using percentages, as shown next:

```
.insert p {
font-size:80%;
}
```

Line-height

There is one more important property for dealing with text: the **line-height**. In practice, this determines how much vertical space there will be between two lines of text. Line-height can be specified as a number, which is multiplied with the font size, a `pixel` value, a `percentage`, or the word `normal`. Normal is typically determined by the browser and is usually somewhere between `1.2` and `1.4`. So the height of every line is `1.2` or `1.4` times the font size. That way there is some room for white space above and below the letters.

For a font with size `16px`, the following three lines of CSS would have the same effect:

```
p {
line-height: 24px;
}
p {
line-height: 1.5;
}

p{
line-height: 150%;
}
```

Note that this specifies the space between lines of a paragraph, not the space between paragraphs. That would be determined by the margin, and the margin is one aspect of the single most important concept of CSS: the **box model**.

The box model

All HTML elements can be treated as boxes. In CSS, the term box model is used while talking about design and layout. It is essentially a box that wraps around HTML elements and that can consist of, from outside to inside: margins, borders, padding, and the actual content.

So far in this book, we have given only short examples so that you could study away from a computer, and we will treat this as a textbook for as long as we can. However, to illustrate the box model, and for you to understand it, it is essential to take our examples and check them in a browser. Consider the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Paul Wellens - California anecdotes</title>
<link rel="stylesheet" href="styles/packtpubch3_1.css"
type="text/css" />
</head>
<body>
<div id="box">
June Lake, often called the gem of the Eastern Sierra, is a beautiful
place that I visit as often as I can.
</div>
<div id="box2">
```

```
Mono Lake, saltier than the Black Sea, features tufa formations that
makes the place look like it could be on the moon
</div>
</div>
</body>
</html>
```

And next is the content of the style sheet file, where we use some of what we just learned, combined with introducing all the box model properties:

```
body {
font-size:12px;
background-color: #ffdead;
font-family:Arial, Verdana, Helvetica, sans-serif;
color:#999;
line-height:1.3;
margin:0;

}

#box {
  width:150px;
  height:150px;
  background-color:teal;
  color:white;
  border: 5px solid orange;
  margin:40px;
  padding:20px;


}
#box2 {
  width:150px;
  height:150px;
  background-color:blue;
  color:white;
  border: 5px solid yellow;
  margin:40px;
  padding:20px;

  }
```

When we display this is in a browser, we will see two square boxes nicely stacked on top of each other. Both boxes have text in them; one is colored green with an orange border, the other one blue and yellow. There is an equal distance between the left of the viewport and the boxes, the top of the window and the top box, and in-between the boxes. Feel free to change the values in the CSS and the HTML of box `#box` and see what happens.

We have our content where our text goes, which we have specified as `150` by `150` pixels. If we had not specified width and height, we would have ended up with a thick, green rectangular area with text and border across our viewport. If we remove only border and padding, we just see the text with a background-color; if we only remove the text, we see a bordered, colored rectangle; and if we remove both, we end up seeing nothing. Finally, if you put just the width and height back, we have a green square of exactly 150 x 150 pixels.

So starting from the inner side, we have our content with a *specified* or *calculated* size. That is the size of our element: `150` by `150` pixels in our example.

Next we can have **padding**. This takes the same background-color as the element and increases the inner portion of our box. In our example, `20px` are added on all four sides of the element.

Then we can have a **border**. This puts a border around our element with the thickness, color, and shape we specify. We used a solid border of `5px` around all the sides, and thus, so far our box is already (5 + 25 + 150 + 25 + 5 = 210) x 210 pixels.

Finally, there is the **margin**. The margin, if we want one, is the area on the outer side of our element box. It is transparent, so it has the background color of the parent element. It merely creates a distance between the box and the adjacent box(es). In this example, we used a margin of `40px`. This makes the total size of our box (40 + 210 + 40 = 290) x 290 pixels.

If you try this out and are an attentive kind of person, you may notice something that does not compute. Good catch! We will explain this in a little while. We will now go over all the box model properties you can use.

Padding

You can specify, or not, padding on all four sides of your element. For this purpose, there are four properties you can use:

- padding-top
- padding-right

- padding-bottom
- padding-left

Here is an example:

```
.menulabel
{
padding-left: 8px;
padding-right: 7px;
padding-top: 4px;
padding-bottom: 5px;
}
```

Did you notice that I changed the order? I did that on purpose. The first order I used is the one supported by the shorthand version of the `padding` property. The second one is the order I think about when I do my design: what is it horizontally, next vertically. So, the same four lines of CSS can be replaced by the following single one:

```
.menulabel {
padding: 4px 7px 5px 8px;
}
```

In the first example we used a single value, which means four times the same. There are also two and three value variants of the shorthand version as well. For example, the following will set both top and bottom to `10px`, and left and right to `15px`:

```
.menulabel {
padding: 10px 15px;
}
```

Border

For the border property also, you have the option to specify different values for top, right, bottom, and left, but there is more than just the width. You can also specify the shape, style, and the color. So, following are the properties for all three:

- **border-width**
- **border-style**
- **border-color**

There is even a property you can use for any of these three in any direction, such as **border-top-style**, so there are many properties to choose from- not that having so many would be practical. I do not believe a border with a different shape, color, and size on each side would make for a nice design!

The most common shapes or styles to choose from are:

- **none**
- **solid**
- **double**

There are, of course, more. Double gives you a double border and can be quite decorative at times. Solid is what I recommend you use most of the time.

```
myimg {
5px solid white;
}
```

So why would you need none? If I do not want a border, I just do not specify one, right? *Wrong!* Some browsers, such as Internet Explorer, automatically put a white border of 1px around any `img` element. So be glad you have the option to deal with that:

```
img {
border:none;
}
```

We already used the shorthand notation in our examples, so I only have to remind you of the order of things to include; I constantly forget them myself: **width style color**.

Having the ability to put borders around things is a very cool feature, in particular for photographs. The web equivalent of the "mat" part of a matted photograph can simply be a well-crafted border for the `img` element.

Margin

Finally, there is the margin property, which clears an element around its borders. It has no background color because it is transparent. The five properties are:

- margin
- margin-top
- margin-right
- margin-bottom
- margin-left

You can specify margin sizes in pixels, in percentages, and so on, just like you can with padding. However, there is one extra, extremely useful value you can set the margin to: **auto**. Change the margin setting in our example to:

```
margin: 40px auto;
```

Like magic, your two square boxes will be centered horizontally. If you make your browser window smaller or bigger, they will still be centered. With the margin set to auto, the browser will calculate the left and right margin for you, relative to the parent element. Many websites have a main `div`, child of the body element, styled similar to the following:

```
#container {
margin: 0px auto;
border:none;
max-width:980px;
}
```

Classical web development uses a canvas with a fixed width, and often fixed height as well, to place everything inside. This example uses `980px`. Thanks to the `auto` margin, there will be an automatic margin on left and right, calculated as half of the remaining horizontal space. The `max-width` is new and differs from `width`. Width will always give you 980 px, max-width only when 980 horizontal pixels are available. If that is not the case, on smartphones for instance, the (smaller) full width of the viewport will become the width. This is one tiny step towards **responsive design**.

Collapsing margins

You have been learning about the box model, have used our example, and may have been wondering why two square boxes with top and bottom margins of 40px are only `40`, not 80 pixels apart.

Well, this is not a bug, but a feature. The W3C specification stipulates that when the vertical margins of two elements are touching, the larger of the two will take effect and the other one is reduced to 0. Some other CSS settings can change this, but this is the default. Once you have done a lot of web development and have gotten used to it, you may decide that it actually makes a lot of sense. We will finish this section by adding two more boxes, with pictures inside of them, to our HTML and CSS, basically creating the first two entries of a photo gallery. We will incorporate what we have learned and discover what is still missing.

Here is the HTML:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8" />
<title>Paul Wellens - California pictures</title>
```

```
<link rel="stylesheet" href="styles/packtpubch3_photo.css" type="text/
css">
</head>
<body>
<div class="entry">
<div class="picturebox">
<img  class="picture" src=http://www.paulpwellens.com/packtpub/images/
junelakefall.jpg alt="junelake" />
</div>
<div class="textbox">
June Lake, often called the gem of the Eastern Sierra, is a beautiful
place that I visit as often as I can.
</div>
</div>

<div class="entry" >
<div class="picturebox">
<img class="picture" src="http://www.paulpwellens.com/packtpub/images/
monolake.png" alt="monolake" />
</div>
<div class="textbox">
Mono Lake, saltier than the Black Sea, features tufa formations that
makes the place look like it could be on the moon
</div>
</div>
</body>
</html>
```

Put the following in your `stylesheet`:

```
.picture {
  width:200px;
  height:130px;
  border:5px solid white;
  margin-left:25px;
  margin-top:40px;

}
.entry {

  width:600px;
  margin:0 auto;

}
.picturebox {
```

```
    background-color:teal;
    color:white;
    width:270px;
    height:220px;
    text-align:center;
    vertical-align:middle;
    /* float:left;  */
    border-bottom:1px solid #FFDEAD;

  }
  .textbox {

    background-color:teal;
    color:white;
    width:250px;
    height:180px;
    padding:40px 10px 0px 10px;
    /* float:left; */
    border-bottom:1px solid #FFDEAD;
    text-align:left;
  }
```

If you run this example in a browser, you will notice that despite wrapping them with another `div` tag, the pictures and the text that goes with it are still not side by side, but stacked on top of each other. The solution is already there, but placed inside the `/*` and `*/` string as a comment. Uncomment those lines, and like magic everything will look the way you want it to.

Positioning

There are several CSS properties you can use to alter the position of an element on the page. The one with clearly the most impact is called **float** (each time I use it, it reminds me of the clown character in the Stephen King novel and movie *It!* when it says: *And they all float!*)

Float

I interpret the CSS float property as the CSS way to stack elements horizontally. If you give all of them a:

```
float:left;
```

You stack them from left to right. With a:

```
float:right;
```

You stack them from right to left. This can become very handy when you want to put the first part of your page, an introduction for example, on the right if room is available, and on top if not. In our above example, changing the float left into a float right will put the pictures on the right.

position:relative

The CSS **position** property can be used to position elements in a spot on the page that is different to where they would normally go. "Normally" is the same as `position:static`. Look at the following code:

```
#redsquare
{
width:100px;
height:100px;
background-color:red;
}
```

This produces a red square in the upper-left corner of its parent element. Now, try the following:

```
#redsquare
{
position:relative;
width:100px;
height:100px;
background-color:red;
left:10px;
top:100px;
}
```

The red square moves `100px` down and `10px` to the right.

position:absolute

Let's add:

```
#container
{
  width:700px;
  margin:50px;
  background-color:teal;
  height:500px;
}
```

and

```
<body>
<div id="container">
<div id="redsquare">

</div>
</div>
</body>
```

The red square box is now inside the teal box, `100px` down, and `10px` to the right. When you replace relative by **absolute**, the red box will be `100px` down and `10px` to the right, relative to the ancestor element instead. This is typically the browser window itself.

Styling lists

One element that you will end up using a lot is the `<ul>` tag: the unordered list. By default, every item in the list will be shown with a round bullet in front of the text. With CSS, you can change the style of your list.

list-style-type

Using this property, you can change the shape of the bullet. Some of the values you can use are: `none` (no bullet at all), `square` (a square), `circle` (a small circle), or `disc` (the default).

list-style-image

You can provide your own image for the bullet by using the **list-style-image** property. The default value is `none`, which means that the bullet image is determined by the value of **list-style-type**. However if you specify `url`, followed by a path to an image, that image will be used instead, for example:

```
url('smiley.gif')
```

list-style-position

By default, the bullets appear outside the content flow. If you specify inside as the value of **list-style-position**, the bullets will move to the inside and the text more to the right.

Styling anchors – pseudo-classes

We conclude our selection on CSS properties with the introduction of some pseudo-classes, typically but not solely used with `<a>`, the **anchor** tag. The anchor tag is used mainly for links. To make it visible that they are indeed links, the default styling of `<a>` happens to be a blue color and the text is underlined, which is not very attractive.

Using pseudo-classes, you can give an anchor tag, in theory any tag, a different look depending on where the cursor is, relative to the link and whether or not the link has already been visited. Here is an example:

```
a:link {
text-decoration:none;  /* switches off the underline */
}
a:hover {
color:white;   /* changes color to white when the curser hangs over it
(hover)  */
}
a:visited {
color:yellow;   /* changes the color to indicate that you already
visited that link */
}
```

Firebug

No matter how well you studied this chapter and various online references about CSS, from time to time things will not look as expected. This is where a debugging tool like **Firebug** comes in handy. Firebug is an extension to the Firefox browser. It lets you click on parts of your page, and then the program will show you the HTML and CSS that is involved and even a picture of the box model showing the padding, border, and margin. Most other browsers, Safari and Chrome in particular, have comparable counterparts.

Summary

In this chapter, we gave you an overview of CSS. This is not a complete reference but we did include the most frequently used and useful CSS properties that should be supported by all browsers. We did not include (on purpose) some of the newer ones that were introduced in CSS3. One important new CSS feature that will be introduced in the second part of this book will be **media queries**. This is essential for building responsive designs, but this topic earns at least one chapter of its own.

So far, we have learned two languages we need to create websites: HTML and CSS. Without further delay, we now move on to the next one, a true programming language: **JavaScript**.