CSS3

Table of Contents:

Introduction

What is CSS?

CSS is short for Cascading Style Sheets and is the primary language used to describe look and formatting for webpages across the Internet and documents of markup (e.g. HTML and XML) in general.

A markup language like HTML was initially designed to provide information about formatting and looks itself, but it soon became clear that it would make much more sense to split this into two layers: Document Content and Document Presentation, with CSS fulfilling the task of the latter. Historically that is why HTML has tags like font, which sole purpose is to adjust font family, color and size locally, a job that is today handled by CSS. This allows the developer to re-use formatting rules across several places in the same document and even across multiple documents. Here's an example to prove my point, and don't worry if it's not entirely clear to you what it does - all aspects will be explained throughout this training:

Old style text formatting, using only HTML:

```
This is a piece of
<font face="Tahoma,Verdana,Arial" color="Blue" size="3"><i><b>text</b></i></font>
with
<font face="Tahoma,Verdana,Arial" color="Blue" size="3">
<i><b>highlighted</b></i>
</font>
elements in
<font face="Tahoma,Verdana,Arial" color="Blue" size="3"><i><b>it</b></i></font>.
```

A more modern approach with CSS:

```
<style>
  .highlight{
    color: Blue;
    font-style: italic;
    font-weight: bold;
    font-size: 110%;
    font-family: Tahoma, Verdana, Arial;
  }
</style>
This is a piece of
<span class="highlight">text</span> with
<span class="highlight">highlighted</span> elements in
<span class="highlight">it</span>.
```

Notice how I simply re-use the same set of rules across several HTML tags. This is already an advantage when using it three times, like we do in the example, but it doesn't end there - put the CSS in an external stylesheet file (more on that later) and you can use the same rules across your ENTIRE website. And how about when you decide that highlighted text should be red instead of blue? With the first approach, you would have to manually edit the tags everywhere you used it - with CSS, just change the single ".highlight" rule!

Summary

CSS allows you to easily apply rules about formatting and layout to your HTML elements and then re-use those rules across multiple elements and even pages. In this introduction, we looked at some CSS code, but we did not talk about how it works and why it looks the way it does - this will be the subject for the next couple of modules, where we start from scratch and explain it all in details.

Hello, CSS world!

After discussing what CSS is and why you should use it, you're probably eager to see it in action. If you're reading this in your webbrowser, then you can already see CSS doing its magic - from header colors and sizes to code sample boxes, menus, lists and pretty much anything else - if it has a background, a border, a different text size or color, then it's likely the thanks to CSS. However, that might be a bit too complex to comprehend at the moment, so let's bring it down to a much more basic level.

Throughout the history of programming, every training with respect for itself has started with a "Hello, world!" example, with the sole purpose of showing the most basic way to bring the text "Hello, world!" to the user's screen. This could however easily be accomplished in pure HTML, without the use of any CSS, so we'll spice it up just a bit with a different color:

```
<style>
  h1{
    color: DeepSkyBlue;
  }
</style>
<h1>Hello, world!</h1>
```

That's it - we just wrote our first CSS rule, targeted toward the H1 tag and used it to change the text color, using the color property and a color value called DeepSkyBlue. You can check the result by pasting the code into your editor or simply by clicking the test button above the code sample.

Summary

As you can probably see, CSS is a fairly simple language, but fear not if you didn't quite understand our first example. In the next module, we'll discuss what the above CSS code actually means and why it's written the way it is.

The Anatomy of a CSS Rule

So, in the previous module, we used the classic "Hello, world!" example to get a glimpse of just how easy it is to style an HTML element with CSS. But why did it look the way it did? In this module, we'll focus on the anatomy and syntax of CSS, to get a deeper understanding of how it works. Let's have a look at the previously mentioned example again, where we had a CSS rule which targeted H1 elements:

```css
h1{
  color: DeepSkyBlue;
}
```

What we have here is a **selector** with one **property** and one **value** - these are the core concepts of CSS and you should try to remember their names as you progress through this training. In this example, **h1 is the selector name**, **color is the property** and **DeepSkyBlue is the value**.

In between these three concepts, you see a variety of special characters: There's the **curly braces** around the property and value, there's the **colon** separating the property from the value and there's the **semicolon** after the value. Each of them makes it easy for the browser to parse and understand your CSS code: The curly braces allow you to group several properties into the same rule (selector), the colon tells the parser where the property ends and the value starts, and the semicolon tells the parser where the value ends.

This might seem a bit too complex for a simple selector like the one we have above, but as soon as we use more complex selector names with more properties and more complex values, it makes perfect sense. Let me illustrate it with a more complex example:

```css
h1, h2, h3{
  color: DeepSkyBlue;
  background-color: #000;
  margin: 10px 5px;
}
h2{
  color: GreenYellow;
}
```

Now we have several selectors, with the first one targeting several elements, as well as several properties and several values - now you can probably see why we need the CSS syntax rules to allow for proper parsing of your instructions.

Formatting and Whitespace in CSS

You may we wondering why I'm formatting the selector the way I do: The initial curly brace is on the same line as the selector name, but the ending one is on a line of its own, properties have been indented and there's a space after the colon separating property and value but not before it. **Why? Because that's how I like it but the truth is that the CSS parser doesn't care about whitespace.**

Some people prefer to have the initial curly brace on its own line and to have a space before the colon as well:

```css
h1{
  color : DeepSkyBlue;
}
```

And that works just as well. In fact, a lot of software has been written to compact/minify CSS to take up the least possible amount of space, like this:

```css
h1{color : DeepSkyBlue;}
```

That will work just fine too, but for more complex rules, it will make it harder to read and edit. However, the parser will understand it just as well as our first example, thanks to the special characters used as separators, as already discussed.

Summary

In this module, we've learned that the basic ingredients of CSS are the selectors, properties and values. A CSS document can contain multiple selectors and a selector can have multiple properties which in turn can have a value consisting of one or several elements. We also learned that curly braces, colons and semicolons separate each of the three ingredients from each other, and that whitespace and formatting is really up to you - the parser generally doesn't care.

We have looked at some basic selectors with some basic properties by now, but selectors is a much deeper subject and there are plenty more properties to work with. We will of course be discussing both in much more detail later in this training.

How CSS works

CSS is interpreted by the browser (the application used to view the webpage, e.g. Internet Explorer or Google Chrome) and then used to decide how the webpage should look. This also means that while there is a very thorough specification of the CSS language, the many browsers across all of the possible devices (desktop computers, tablets, mobile phones etc.) interprets your CSS code in its own way. This means that even though most of your work will likely look and act the way you expect it to, there might be subtle differences if you view your work in some of the many browsers on the market.

Since CSS is just a specification and not a law, browser vendors are free to add their own CSS properties, allowing you to perform more advanced things, but only in that particular browser. This is frequently used by the various vendors to try to persuade W3 (the organization in charge of many Internet related specifications, including HTML and CSS) into adding functionality to the next version of the CSS specification.

Historically, the problem with rendering differences across browsers have been a larger problem than it currently is. Especially Microsoft has had problems following the specification with their Internet Explorer, with version 6 being the worst example of this - competing browsers were following the specification way better, but because of market shares, developers had to implement several nasty workarounds to fully support IE6 and its many quirks. Fortunately for developers all over the world, Microsoft has done a lot to remedy these problems in later versions of Internet Explorer.

However, you will still run into differences in rendering, especially when you test across different browsers on different devices and operating systems. Your webpage might not look entirely the same in Internet Explorer as it does in Chrome, and there might even be differences when looking at it in Chrome on a PC with Linux, OSX or Linux. For that reason, always test your webpage in as many browsers as possible and make sure that your CSS validates (more on that later).

Summary

CSS is interpreted by the client (usually a web browser) on each request, and since different browsers uses different parsing engines, things might not look entirely the same across different devices, platforms and browser versions. Make sure that you test all of your pages in as many browsers as possible and to help reduce the number of problems you should make sure that your CSS code can pass the checks of the W3 validator.

In this training we will only be discussing properties and techniques which can currently be used in the most recent versions of the most popular browsers: Microsoft Internet Explorer, Google Chrome, Mozilla Firefox and Apple Safari. However, the fact that these properties and techniques are understood by the browsers still doesn't mean that they are interpreted AND used in the exact same way, so always remember the golden rule of CSS: Test your work as much as possible!

Linking CSS to HTML

As already explained, CSS contains information about how your markup (usually HTML) should be presented to the end user. That means that the two languages have to be linked together - the browser needs to know that you want to combine a piece of HTML markup with a piece of CSS code. To obtain the highest level of flexibility, there are several ways to accomplish this.

Inline CSS through the Style attribute

Almost every HTML tag includes the Style attribute and using this attribute, you can specify CSS directly for the element. This defeats one of the main advantages of CSS, re-usability, since CSS code applied with this technique only applies to a single element and can't be re-used for other elements.

It is however a great way to test things out, or to specify rules which you do not expect ever to re-use. Here's how it may be used:

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
  <span style="color: Blue; text-decoration: underline;">Hello, CSS!</span>
</body>
</html>
```

Notice that I can define several properties for the same element, in this case rules for text color as well as text decoration. This is by far the easiest way to use CSS, since it doesn't involve extra tags or files - just locally defined CSS. However, for reasons already stated, this is not the preferred way of using CSS.

Another disadvantage is the fact that the style code will have to be downloaded each time the page is requested. This is not really a problem for a single element with a couple of style rules, but if you have a loop which outputs the same style attribute code many times, then it fills up your markup document unnecessarily, and instead of being cached by the browser and only downloaded once per visitor, the code is downloaded again and again!

Document wide CSS through style blocks

The second easiest way to apply CSS to elements in your document is through the use of a style block. HTML includes a tag called style, which can contain CSS code. Here, you can define rules which can then be used all across your document. Here's an example:

```
<!DOCTYPE html>
<html>
<head>
  <title>Style blocks</title>
  <style>
    .highlight{
      color: Blue;
      text-decoration: underline;
    }
  </style>
</head>
<body>
  <span class="highlight">Hello, CSS!</span>
</body>
</html>
```

Notice how I can now define the rule in one place and then re-use it multiple times in the document. You can define multiple style blocks, if you want to, and place them wherever in the document you want to. It is however considered best practice to include the style block(s) in the top of the document, inside the head (<head>) section.

Global CSS through external CSS documents

So, by using the style block as described above, you can re-use your CSS code all over the document, but you still have to include it on all of the pages of your website, which requires the browser to download it on each request instead of just downloading an external CSS file once and then cache it. This is a major disadvantage of the style block approach and why you should normally go for the third approach instead: The external CSS file!

A CSS file is simply a plain text file saved with a .css extension and then referenced in the file(s) where you want to apply the rules. If we re-use the example from the style block part, we can then move the "highlight" rule to a new file (without the HTML part) and save it under an appropriate name, e.g. **style.css**:

```
.highlight{
  color: Blue;
  text-decoration: underline;
}
```

We can then reference it in our HTML document, using the link element:

```
<!DOCTYPE html>
<html>
<head>
```

```
  <title>Style blocks</title>
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <span class="highlight">Hello, CSS!</span>
</body>
</html>
```

This example requires the HTML and CSS file to be in the same directory - if they are not, you need to update the href attribute to match the path.

Now we define all of our CSS code in its own file and then just reference it inside all of our subpages to take advantage of the defined rules! If you want to, you can divide your CSS code into several external files and only include the ones you need - it all depends on how big your website is and how you prefer to organize your code.

Summary

We have talked about different ways for your webpage to consume CSS code in this module.

The first approach was the **inline style attribute**. Its biggest advantage is the fact that its so easy to use, but as a disadvantage, it cuts you off from some of the biggest benefits of CSS: Code re-usability and the ability to make layout changes in a single place and have it applied across your entire website.

The second approach uses the **style block**. It's only slightly less cumbersome to use than the inline style attribute, and allows re-using your CSS code. The biggest disadvantage is that code in a style block is global across the page, but not across multiple pages, meaning that you will have to include it on each subpage of your website.

The third approach is the one you most commonly see on websites: **External CSS file(s)**. It only has the disadvantage of being slightly harder to work with, because you have to place the CSS in a separate file which you will have to open to make changes - a very small price to pay for the ability to re-use your CSS across your entire website!

Please be aware that throughout this training, I will mainly be using the second and the first approach, but only because the examples presented in this training are all small, separate entities and not a big website. It's simply a lot easier to demonstrate the various CSS properties this way, instead of having to separate markup and CSS into distinct files and explaining this fact each time. You are free to use the approach you prefer, but for anything other than small examples and very simple websites, the third approach is usually the superior way!

Introduction to CSS Selectors

We already briefly talked about selectors in the module on the anatomy of CSS, but as you will soon realize, selectors come in many different flavors and in various combinations. Selectors can be extremely powerful and most styling would not be possible without them.

As already mentioned, a selector encapsulates one or several properties, which dictates behavior and look for a certain element. The cool thing about selectors is that they allow you to target an element in different ways, in different states, in various hierarchies and even several elements at the same time.

Naming your selectors

Basically, a selector name must begin with an underscore (_), a hyphen (-), or a letter (*a*-*z*) and then be followed by any number of hyphens, underscores, letters, or numbers. When naming your selectors, there are some characters that you can't use, usually because they have a special meaning - these will be covered later on in this training.

Summary

A selector is a name and one or several properties. In the next modules, I'll show you all the ways you can use a selector and to demonstrate it, I will be using several CSS properties. You might not know them all by now, or perhaps none of them at all, but don't worry - they will all be explained later on. I include them now because empty selectors don’t really give a good impression on how CSS works, but feel free to jump around in this training as you please if you want to know more about the properties used.

The Element Selector

The most basic type of selector is probably the one that simply targets an existing HTML element. For instance, you can target all paragraph elements (<p>) simply by writing the name of it in your stylesheet:

```
p{
  color: Red;
}
```

With this simple rule, we have just changed the text color of all paragraphs to red - the element selector is very strong!

You can target any of the valid HTML elements this way and even non-existing elements can be targeted - if you want a <tiger> tag on your page, you are free to write a CSS selector which targets your tiger-element (although that wouldn't validate!).

So, for most of the time, your element selectors targets your everyday HTML tag. For instance, you may decide that bold-tags should no longer cause text to be bold:

```
b{
  font-weight: normal;
}
```

The internal stylesheet of most browsers dictates that bold tags have bold text, but with the power of CSS, you can easily change that, either locally (more on that later) or globally, like we just did.

Here is a more complete example, where we use what we just learned. Feel free to check it out and play around with it, to see how it works:

```
<style>
  p{
    color: Red;
  }
  b{
    font-weight: normal;
  }
</style>
<p>Here's a paragraph!</p>
<p>Here's another <b>paragraph</b> - the word paragraph would normally be bold
here!</p>
```

Summary

So, the major advantage of element selectors is that they are global - every place where your stylesheet is included, these rules affect your elements. Obviously, this is also the main disadvantage of the element selectors, because sometimes that's really not what you want. Fortunately, there are several other options, including class and ID selectors, which we'll look into next.

The Class Selector

We previously looked at element selectors, which targets all elements (which normally translates to HTML tags) on a page. If we want to be more specific, class selectors are the next step. Instead of targeting all elements with a specific name, they target all elements that has a specific class name specified.

While this normally makes the list of targets narrower, it does give you the opportunity to target elements of various types (e.g. both bold and italic tags) at the same time - with a class selector, the element type/name is no longer the important part.

A class selector looks just like an element selector, but instead of using names that are tied to the names of HTML elements, you make up the name and then you prefix it with a dot (.). For instance:

- .red { }
- .myElements { }
- .navigation { }

Only elements that uses one or several of these class names are targeted. Allow me to illustrate how this works with an example:

```
<style>
  .red{
    color: Red;
  }
  .beautiful{
    font-weight:bold;
    color: Blue;
    font-style: italic;
  }
</style>
<p class="red">
  Here's some text - <span class="beautiful">this part is especially pretty!</span>
</p>
This text is very normal!
```

Try out the example and see the result for yourself. Notice that two similar paragraph elements now look completely different because we have assigned a class to the first one. Also notice how I can name the selectors however I want - the two names are actually a great example of bad and good (or at least better) naming conventions. The name "red" is not a good name, because the rule could easily contain more than the color-related rule, or the color could easily be changed in a re-design from red to blue. The latter name is better, because it is more general and doesn't convey a specific color or look.

Element specific classes

In our first example, all element types could use our classes but, in some situations, you may want to limit the use to a specific element type. This is usually done to indicate how the class is supposed to be used, to allow for more than one class with the same name and avoid conflicts. Element specific classes are used simply by appending the class name to the element name in your selector, like this:

```
<style>
  span.beautiful{
    font-weight:bold;
    color: Blue;
    font-style: italic;
  }
</style>
<p>
Here's some <b class="beautiful">text</b> - <span class="beautiful">this part is
especially pretty!</span>
</p>
```

Try out the example and notice how even though we try to use the *beautiful* class in two places, it only works for the span element, because we now require this.

Multiple classes

Classes are not unique and the class property of an HTML tag allows you to specify more than one class. The cool thing about this is that it allows you to combine the rules for several selectors and use them for the same tag however you want to.

This also means that instead of writing selectors with many rules and then only targeting few elements, you can write fewer specific selectors and simply combine them when it is appropriate. This allows for greater re-usability, which is really what CSS is all about.

Have a look at this example:

```
<style>
  .status{
    padding: 5px;
    margin: 5px;
    border-width: 1px;
    border-style: solid;
  }
  .error{
    color: Red;
    border-color: Maroon;
  }
  .information{
    color: Blue;
    border-color: Navy;
  }
</style>
```

```
<div class="status error">
  This is an error!
</div>
<div class="status information">
  This is information!
</div>
```

Here we use CSS to show status information. We have a common .status selector and then we have a selector for error messages and one for informational messages. Error and information messages obviously share stuff, since they are both a type of messages, but they also have distinct looks. So, we put the shared stuff in a class called .status, and then put the rest in different classes called .error and .information, and then we use them on the div tags, simply by separating their names with a space.

Without the common .status class, we would have to repeat all the properties for each class, which would be a waste of bandwidth and force us to change stuff in multiple places whenever we wanted thicker borders or something like that.

Summary

As demonstrated, CSS classes are very versatile - they can be as specific or non-specific as you want them to and you can use them for all sorts of element, or not, depending on whether the element references the class or not. This makes classes the most flexible type of selector you can use in CSS. In the next module, we'll have a look at the ID selector, which is even more specific.

The ID Selector

We started the Selector-part of this training with the most wide-ranging selector, which targets elements, then we talked about the more specific class-type selector, and now we'll be discussing the most specific selector type: The ID selector. The ID selector is actually so specific that it only targets a single element on the page!

Just as we saw with the class selector, the ID selector uses a specific attribute found on all HTML tags to figure out whether the specific rule applies to the element or not, and that's the ID attribute. The value of the ID attribute should be unique, according to the HTML specification, meaning that it can only be used on a single element per page. As a consequence of that, a CSS ID selector should also be used only when you need to target one, specific and unique element on a page.

An ID selector looks just like a class selector, but instead of having a dot as the prefix, it uses the hash sign (#). Let's see how it works:

```
<style>
  #main-header{
        color: GreenYellow;
  }
</style>
<h1 id="main-header">Hello, CSS!</h1>
```

As you can see, it works just like classes, but instead of using a dot, we use a hash character, and instead of using the class attribute, we use the id attribute - the difference lies in the fact that the ID should be unique, as explained in the introduction to this module.

So why use ID selectors at all? Well obviously, no one will be forcing you - you may use class selectors instead, if you want to. But by using an ID selector, you specify to yourself and everyone else who will be writing markup for your CSS, that the selector should only be used in one single place. This is often used for main page elements, e.g. the tags that creates the structure of the page (menu, side box, main content and so on).

Besides that, you may already have specified ID's for some of your tags, e.g. to reference them from JavaScript code. If so, you may write a CSS ID selector and have it applied automatically.

Element specific ID selectors

Just like the class selector, you may limit an ID selector to a specific element type by putting the name in front of the selector name, like this:

```
<style>
  h2#main-header{
    color: GreenYellow;
  }
</style>
<h1 id="main-header">Text in h1</h1>
<h2 id="main-header">Text in h2</h2>
```

With that in place, this specific ID selector will only apply to a header (h1) tag.

Summary

With the ID selector, you get really specific - from targeting all elements by name/type or class name, the ID selector, by design, only targets a single element on the page. Use it mainly for the larger, structural elements on your page, e.g. the navigation, top area, main content and so on.

Grouping Selectors

So far, all of our selectors have only targeted one class, one ID or one element. However, CSS makes it ridiculously easy to target more than one element at the same time, allowing you to specify the same properties and rules for more than one element at the same time - **just separate the selector names with a comma and you're good to go**. This is another one of those features of CSS which allows for greater code re-usability - there's no reason to specify the same properties for several elements/classes, if you can re-use them. Here's a nifty example:

```
<style>
  h1, h2, h3{
    color: Maroon;
  }
</style>
<h1>Main header</h1>
<h2>Header level 2</h2>
<h3>Header level 3</h3>
```

As you can see, we can now target h1, h2 and h3 elements with one, single rule, instead of having to specify it for each of them. You can mix and match with class and ID selectors too, if you want to:

```
h1#main, h2.sub, h3, .someClass, #anID{
  color: Maroon;
}
```

Now the really cool thing is that thanks to CSS and its cascading nature, you can still add rules specific to one or several of elements and the browser will apply it according to precedence (we'll talk more about that later). Check out this example:

```
<style>
  h1, h2, h3{
    color: Maroon;
    text-align: center;
  }
  h1{
    background-color: Silver;
    padding: 10px;
    text-align: left;
  }
  h2, h3{
    background-color: Gray;
    padding: 5px;
  }
</style>
<h1>Main header</h1>
<h2>Header level 2</h2>
<h3>Header level 3</h3>
```

Try out the example and notice how the browser uses the appropriate properties from the selectors. We are able to group together the common properties in one selector, and then we can add and even modify them in the more specific selectors later on.

Summary

Grouping selectors together makes it so easy to re-use CSS code, while maintaining a very high degree of flexibility - if a common rule is used in several places, you can still overrule it by writing a more specific selector.

Introduction to Advanced Selectors

So far, we have looked at some very basic CSS selectors, with the most advanced stuff being the grouping of multiple selectors into the same. However, CSS selectors can be much more advanced than that, which makes them an extremely powerful tool in your web-designing toolbox.

In the upcoming modules, we'll look into how these advanced selectors work and how you can benefit from them. However, I want to make it clear that while they can be very useful, they are not a requirement for doing most CSS work. What we will be doing with them can be accomplished with basic CSS selectors - just not as easy or syntactically pretty.

So, in the next modules we'll look into the so-called combinators, where we use the hierarchical nature of HTML to shape and limit our selectors.

Summary

We'll look into combinators in the next modules, which are advanced CSS selectors used to limit and filter in the elements targeted by the selector.

If you're more eager to get started using all the interesting CSS properties, you may want to skip this entire section about advanced CSS selectors for now and return to it later on, when you're ready to see some of the cool things you can do with them

The Descendant Selector

So far, we have only used selectors which directly targeted a specific element or element(s) with a specific ID or class. Especially targeting elements of a specific type, e.g. all links or images, is very powerful, but what if you want to limit this, for instance to elements found only in a specific part of the page? This is where combinators come into the picture, a range of selector types which uses the hierarchy of elements on your page to limit the elements targeted.

In this module, we'll look into the **Descendant selector**, which allows you to limit the targeted elements to the ones who are descendants of another element. The syntax is very simple - you simply write the parent(s), separate with a space, and then the actual element you want to target. Here's an example:

```
<style>
  p b{
    color: Blue;
  }
</style>
<p>Hello, <b>world</b> - what a <b>beautiful</b> day!</p>
Hello, <b>world</b> - what a <b>beautiful</b> day!
```

In this example, I want all bold elements to be blue, but only if they are inside a paragraph tag. If you try the example, you will see that while the bold tag is used four times, only the first two of them are blue - that's because they are inside a paragraph, which our descendant selector requires!

This is obviously a very basic example, but think of the bigger picture - for instance, this would easily allow you to change the color of all links inside your main menu, without having to tag them all with a specific class!

Of course, you can use all the usual modifiers to limit your selector to specific classes or ID's, like in this example:

```
<style>
  p.highlighted b{
    color: Blue;
  }
</style>
<p class="highlighted">Hello, <b>world</b> - what a <b>beautiful</b> day!</p>
<p>Hello, <b>world</b> - what a <b>beautiful</b> day!</p>
```

Here, we only target bold elements which are descendants of a paragraph with the class "highlighted".

A descendant doesn't need to be the direct child

With this selector type, you should be aware that not only direct children are targeted - also children of the child (grandchildren) and so on will be targeted, all the way down through the hierarchy. This example should demonstrate that just fine:

```
<style>
  div.highlighted b{
    color: Blue;
  }
</style>
<div class="highlighted">
  <b>Level 0...</b>
  <div>
    <b>Level 1...</b>
    <div>
      <b>Level 2...</b>
    </div>
  </div>
</div>
```

Here, we target bold elements which are descendants of a div tag with the class "highlighted". If you try the example, you will notice that even though we wrap the last set of bold tags in several layers of div tags, it is still affected by the rule about blue bold tags. If you only want direct children to be affected, you need the child selector, which will be explained in one of the next modules.

Summary

The syntax for a descendant CSS selector is extremely simple - just write the parent selectors, a space and then the target selector. Despite the fact that it is so easy to use, it's also extremely powerful. Hopefully you have already learned that from the examples of this article, but if not, just read on, to see more about what the advanced CSS selectors can do for you.

The Child Selector

In the previous module, we saw just how powerful the descendant selector can be, because it allows you to target ALL children and grandchildren (and so on) of one or several elements. However, sometimes this can be TOO powerful - sometimes you only want to target the direct children of an element. Fortunately for us, CSS has a selector for this as well!

In the previous module, we had an example with a descendant selector which automatically targeted all bold tags all the way down through the hierarchy:

```
<style>
  div.highlighted b{
    color: Blue;
  }
</style>
<div class="highlighted">
  <b>Level 0...</b>
  <div>
    <b>Level 1...</b>
    <div>
      <b>Level 2...</b>
    </div>
  </div>
</div>
```

The syntax for using the direct child selector looks like this:
parent > child

So, re-writing the above example to only affect direct children of the div.highlighted tag is very easy:

```
<style>
  div.highlighted > b{
    color: Blue;
  }
</style>
<div class="highlighted">
  <b>Level 0...</b>
  <div>
    <b>Level 1...</b>
    <div>
      <b>Level 2...</b>
    </div>
  </div>
  <b>Level 0 again...</b>
</div>
```

Notice how we now have what we wanted - only direct children of the parent are now affected.

Of course, you can add more requirements to the selector, both of the descendant and the child type. Take for instance this example:

```
<style>
  div.highlighted > ul > li > a{
    color: DarkOrange;
  }
</style>
<div class="highlighted">
  <ul>
    <li><a href="#">List Link 1</a></li>
    <li><a href="#">List Link 2</a></li>
    <ul>
      <li><a href="#">List Link 2.1</a></li>
    </ul>
    <li><a href="#">List Link 3</a></li>
  </ul>
</div>
```

Notice how I've just made the selector rule more specific than what we have previously seen - instead of just targeting e.g. links inside the ".highlighted div" or links inside a list, I target links which is a direct child of a list item tag, which is a direct child of an unordered list which is a child of a div tag with the "highlighted" class. So even though we have a child list inside of the list, its links won't be affected by this rule and if you add another list which is not inside of the highlighted div, it won't be affected either.

Summary

The child selector is a more specific version of the descendant selector, but that doesn't make it less powerful. You are very likely to run into situations where either of them can help you simplify your CSS code!

The Sibling Selector

We just looked at how the child and descendant selectors can be used to target specific children/grandchildren of an element, but what if you want to target siblings instead? CSS has a couple of selector types for that as well, and in this module, we'll check out the general sibling selector.

With the general sibling CSS selector, which takes a selector, followed by a tilde character (~) and then the selector you wish to target, you can target elements by requiring the presence of another element within the same parent element. Another requirement is that the first part of the selector needs to be present in the markup BEFORE the targeted element, even though they are all children of the same parent. Here's an example which will demonstrate it:

```
<style>
  h2 ~ p{
    font-style: italic;
  }
</style>
<div id="content">
  <h1>Hello, world!</h1>
  <p>Some text here</p>
  <h2>Hello, world!</h2>
  <p>Some text here</p>
  <p>More text here</p>
</div>
```

So, all of the text elements are children of the same div element. We then specify that paragraph elements which are siblings to an H2 element should be in the *italic* style. As you can see, if you try the example, this means that the last two paragraph tags will be in italic, but not the first, because it comes before the H2 element in the markup. You will also notice that the sibling selector does not affect grandchildren:

```
<style>
  h2 ~ p{
    font-style: italic;
  }
</style>
<div id="content">
  <h1>Hello, world!</h1>
  <p>Some text here</p>
  <h2>Hello, world!</h2>
  <p>Some text here</p>
  <div>
    <p>More text here</p>
  </div>
</div>
```

Notice how the paragraph inside the second div tag is no longer affected because it's no longer directly related to the h2 element.

Summary

The general sibling selector allows you to target elements based on other elements within the same parent/child scope. However, you will notice that this can be a very broad set of elements, depending on how many elements the parent holds. Using the Adjacent sibling selector, which we'll discuss in the next module, you can limit the elements to those immediately after another element.

The Adjacent Sibling Selector

In the previous module, we discussed the sibling selector, which allows us to select all elements which follows another element within the same parent. However, using the Adjacent sibling selector, you can limit this to only include the first element which comes directly after the first element in the markup. This can be a bit difficult to imagine, so allow me to illustrate it with an example:

```
<style>
  h2 + p{
    font-style: italic;
  }
</style>
<div id="content">
  <h1>Hello, world!</h1>
  <p>Some text here</p>
  <h2>Hello, world!</h2>
  <p>Some text here</p>
  <p>More text here</p>
  <p>Even more text here</p>
  <h2>Hello, world!</h2>
  <p>Text here as well...</p>
  <p>But no more!</p>
</div>
```

With the adjacent sibling selector, we have just specified that the first paragraph element after all H2 elements should use italic text. If we had used the general sibling selector, like we did in the previous module for an example much like this one, all paragraph elements after the first H2 element would be targeted. The syntax for the adjacent sibling selector is just as easy though, as you can see - the two selector parts are simply joined by a plus character (+).

Of course you can use more than simple element selectors - the two parts of the general AND the adjacent sibling selectors can be as specific as you want them to, utilizing all the techniques we have previously looked at in the various modules on selectors. Here's an example, just to prove my point:

```
<style>
  div#content h2.main + p{
    font-style: italic;
    color: Blue;
  }
</style>
<div id="content">
  <h1>Hello, world!</h1>
  <p>Some text here</p>
  <h2 class="main">Hello, world!</h2>
  <p>Some text here</p>
  <p>More text here</p>
  <h2>Hello, world!</h2>
  <p>Text here as well...</p>
</div>
```

The selector is simply made up of two parts in this case, separated by the plus sign, and you can be as specific in each of the parts as you want to.

Summary

The general sibling selector and the adjacent sibling selector allows you to target elements based on which element comes before them, within the same parent element. As we saw in this module, the adjacent sibling selector is a bit more restrictive, allowing you to target only elements which comes right after another element.

Introduction to the CSS Box Model

The CSS Box Model is a very complicated issue, especially because various browsers have interpreted slightly different in the past. The Box Model describes how the (visible or non-visible) box around an element is laid out and how stuff like margins, paddings and borders behave.

Inline vs. Block elements

The first and most important thing you need to know about the box model is the difference between **inline** and **block** elements. Basically, every HTML element has three different states: Block, Inline or not displayed. The last group is reserved for tags which are not supposed to be visually rendered by the browser, e.g. meta tags, style blocks and so on.

The difference between inline and block elements are more interesting. **A block level element will naturally span the entire available width**, without concern about how much horizontal space it actually needs. As a result of that, a block level element will automatically push following content to a new line, so by default, two block level elements can't stand next to each other - the first will push the second one to the next line (floating elements will allow this though, but more on that later).

Contrary to the block level element, an inline element does not break the current flow. An inline element only takes up the space it needs to render its content and after that, more inline elements can be displayed.

A good way to remember the difference is to think of an inline element as a line of text/sentence and a block level element as a paragraph - the sentence can stand together with other sentences to form an entire paragraph, but the paragraph breaks the natural flow to create space around it.

Elements are born as either invisible, inline or block level elements. An example of inline elements include span tags (the generic inline element), links (anchors), images and form fields like input. Common block level elements include div tags (the generic block level element), header tags (h1-h6), paragraphs, lists and tables.

With all this theory taken care of, I would like to present you with a simple example showing the difference in action:

```
<style>
  .box{
    background-color: Orange;
  }
</style>
```

```
<div class="box">Block 1</div><div class="box">Block 2</div>
<br>
<span class="box">This is an inline element... </span>
<span class="box">and so is this</span>
```

Try out this example and notice how the elements, despite using the same CSS class, behaves completely different, just as described above. Most notably, the div elements (which are block level elements by default) uses the entire available width, while the span level elements (which are inline elements by default) can stand right next to each other and be part of the same sentence.

Inline elements and sizes/margins

An important thing to be aware of is the fact that inline elements act differently when using margin and size (width/height) properties. First of all, **the width and height properties are ignored for inline elements**. Instead, you can use the line-height property to make an inline element smaller or bigger, as the space between each line grows or shrinks.

You will also see margins and paddings acting differently than you might be used to, when applied to inline elements. In general, if you need to create space between or inside elements, you may want to use a block level element instead. If you're trying to use inline elements because you want two or more elements to stand next to each other, then a floating block level element might be more suitable - more on that later in this section of the training.

Summary

In this article, we briefly discussed what the CSS Box Model is and then we talked a lot about the difference between inline and block level elements. Remember that while some elements are born as inline elements and others are born as block level elements, this can easily be changed with the **display** property. This will be explained later.

We also talked about concepts like margins, paddings and borders. If you are not familiar with these concepts, then don't worry - they will be fully explained in the upcoming modules.

Margin

You may know the concept of margins from your word processing application, e.g. Microsoft Word, where you can define how wide and tall the margins should be, as in how much space you want from the edge of the page and into the actual content. Margins in CSS work just like that - they allow you to put a bigger distance between your elements by specifying the desired margins. Put another way, **the margin is an outer, invisible border around your element**.

The default value for the **margin** properties is auto, which usually translates to zero. However, for some elements, the browser will likely choose to add a certain amount of pixels to the margin, which you may of course override if you want to. A good example of this is the header elements (h1-h6), the body tag and paragraphs - all of these usually have margins added to them by the internal stylesheet of the browser. This is why you will often see user stylesheets setting the margin to 0 - they are overriding any margin applied by the browser or other stylesheets.

You can specify the margin(s) for an element by using one or several of the four margin-* properties, named **margin-top**, **margin-right**, **margin-bottom** and **margin-left**. Here's an example:

```
<style>
  .box{
    background-color: DarkSeaGreen;
    width: 100px;
    height: 100px;
    margin-top: 10px;
    margin-right: 5px;
    margin-bottom: 10px;
   margin-left: 5px;
  }
</style>
<div class="box">
  Box
</div>
```

Using the margin shorthand property

While it's perfectly fine to use the margin-* properties, a so-called shorthand property exists, simply named **margin**. It allows you to define margin values for all four sides, without having to repeat all the property names each time. The margin property simply directs your values out to the margin-top, margin-right, margin-bottom and margin-left properties, so it's just another way of using the CSS syntax, which will shorten your code in many situations.

The margin property can take from one to four values. This allows you to specify margins for all four sides of an element, but if all four values are the same, you can shorten it to just one value which will be applied to all sides. If you have the same value for top and bottom margin, and another value which should be applied to both the left and right margins, you can just have two values. Confused? Don't be - here's an example where we use all of the mentioned approaches:

```
<style>
  .box{
    background-color: DarkSeaGreen;
    width: 100px;
    height: 100px;
  }
</style>
<div class="box" style="margin: 10px 10px 10px 10px;">
  Box
</div>
<div class="box" style="margin: 10px 10px;">
  Box
</div>
<div class="box" style="margin: 10px;">
  Box
</div>
```

Notice how I start with the most verbose way, where I declare all four values, and then work my way down.
But which order should they be in? And how does the various versions work? When specifying margins, the following rules apply:

4 values:

1. [top margin]
2. [right margin]
3. [bottom margin]
4. [left margin]

3 values (not as commonly used):

1. [top margin]
2. [left/right margin]
3. [bottom margin]

2 values:

1. [top/bottom margin]
2. [left/right margin]

1 value:

1. [top/right/bottom/left margin]

Relative margins

A lot of margins will be specified in absolute values (usually pixels, like we saw in the first example), but just like most other size related CSS properties, you may also use relative values. This is usually done by either using a relative size unit, for instance the **em unit** (1 em equals the size of the current font), or simply by specifying a percentage-based value. In the next example, I have specified a common margin of 1em, and then I have applied a different font-size, measured in em's, for each of the three boxes:

```
<style>
  .box{
    background-color: DarkSeaGreen;
    width: 100px;
    height: 100px;
    margin: 1em;
  }
</style>
<div class="box" style="font-size: 1em;">
  Box
</div>
<div class="box" style="font-size: 2em;">
  Box
</div>
<div class="box" style="font-size: 3em;">
  Box
</div>
```

Now if you try to run this example, you will notice that even though the boxes share the same actual margin value, the margin is now relative to the font-size, meaning that it grows when the font-size is increased. This is useful in many situations!

Negative margins

So far, we have only used positive margins but you can just as easily use negative margins. This can be utilized to pull an element closer to another or to negate the effect of padding. Now, padding will be explained in the next module, but since we'll be using it in the next example, just think of it as an internal margin - space reserved inside of the element, instead of outside like margins.

To illustrate it, have a look at this example:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    padding: 10px;
    background-color: DarkSeaGreen;
  }
  .box-header{
    background-color: CornflowerBlue;
  }
</style>
```

```
<div class="box">
  <div class="box-header">
    Header
  </div>
  Hello, world!
</div>
```

This is a common usage, where we have a box with some text in it, with a header in a different color. A box like that may be used in many places on a page, and in all cases, we would likely add some padding to it, to keep text and content away from the edges of the box.

However, we may want to make an exception for the optional header and let that one go all the way to the edges of the box. We would probably also like to have some distance from the header and down to the following content and all of this can be solved with a proper margin declaration:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    padding: 10px;
    background-color: DarkSeaGreen;
  }
  .box-header{
    background-color: CornflowerBlue;
    margin: -10px -10px 10px -10px;
    padding: 5px 10px;
  }
</style>
<div class="box">
  <div class="box-header">
    Header
  </div>
  Hello, world!
</div>
```

We simply use a combination of negative and positive values for the margin property to adjust the position and compensate for the padding in the parent element.

Summary

Margins are used to adjust empty space between elements, something that is very useful for creating designs that are spacious and easy on the eye. Some elements do come with built-in margins (usually dictated by the browser), but you are always free to define your own margin values, either to decrease or increase them or to simply reset them to zero. Margins are defined in absolute (pixels) or relative (percentage or relative units) sizes and they can be both positive or negative to push or pull an element away or towards other elements.

Borders

By default, most HTML elements doesn't have a border, but CSS gives you plenty of options to define one, with a range of border related properties. In fact, the border can be adjusted using so many properties that it sometimes gets a bit confusing. However, at its most basic level, you will usually want to control the border color, width and style, so let's look at an example of just that:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    border-color: CornflowerBlue;
    border-width: 2px;
    border-style: solid;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

This is pretty much as basic as it gets - by using the **border-style**, **border-color** and **border-width** properties, we can easily give an element the border we want. Now let's talk about these three properties.

Border color

Just a plain old color property, where you can define the color for the border in several different ways, as it is with all color related properties in CSS. Look elsewhere in this training for a complete walkthrough of all the options you have when defining a color in CSS.

Border width

Works much like margins and paddings - can be either an absolute value, like in this example, a relative value, or one of the pre-defined border width values: Thin, medium or thick. If you use the pre-defined values, it's up to the browser to interpret them, which basically gives you less control of how your work will look across all the different devices and browsers.

Border style

For the style of the border, you have a range of options. The most commonly used is probably the solid border, but there are many more to choose from:**hidden**, **dotted**, **dashed**, **solid**, **double**, **groove**, **ridge**, **inset** and **outset**. Wanna know how they all look? Try out this example:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    border-color: CornflowerBlue;
    border-width: 4px;
    margin: 10px;
    float: left;
  }
</style>
<div class="box" style="border-style: dashed;">Dashed</div>
<div class="box" style="border-style: dotted;">Dotted</div>
<div class="box" style="border-style: double;">Double</div>
<div class="box" style="border-style: groove;">Groove</div>
<div class="box" style="border-style: inset;">Inset</div>
<div class="box" style="border-style: outset;">Outset</div>
<div class="box" style="border-style: ridge;">Ridge</div>
<div class="box" style="border-style: solid;">Solid</div>
```

Shorthands

We talked about shorthand properties earlier - properties which allows you to define values for multiple properties at the same time. In the first example, we actually used shorthand properties to define the same color, width and style for borders of all four sides of an element. For instance, border-style actually maps to **border-top-style**, **border-right-style**, **border-bottom-style** and **border-left-style**, and the same goes for border-width and border-color.

This also means that border-style, border-color and border-width can all take from one to four values, allowing you to use different styles, colors and widths for all four sides of the element. Here's an example:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    border-style: solid dashed ridge dotted;
    border-color: CornflowerBlue Chartreuse CadetBlue Chocolate;
    border-width: 1px 2px 3px 4px;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

Now the final result is a pretty odd-looking box, but hopefully you can see how it works. Just like with margins, you can specify one or several values, which will be applied from the top and clockwise around the element (top, right, bottom, left).

Without these shorthands, you would have to use 12 properties to achieve the same result, but it can get even shorter: Using the border property, you can shorten it even more. Here is our very first example, re-written to use the border property:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    border: 2px solid CornflowerBlue;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

We just saved a couple of properties more! When using the border property, the correct order is width, style and color and while the browser may be able to understand it even if you don't get the order right, it's recommended to always use this specific order when using the border property. Notice that you ARE allowed to skip one or two values - in that case, the browser will try to understand which of them you skipped and assign the default values for the missing ones.

Border radius

As an addition to CSS 3, the ability to define border radius was added, effectively giving developers the possibility to make rounded corners on their elements. For new developers, this might seem trivial, but before this was added, the desire for rounded corners generated hundreds of how-to articles!

Fortunately, that was in the past and thanks to the border-radius property, it's now as easy as pie to get rounded corners:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    border: 3px solid CornflowerBlue;
    border-radius: 5px;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

The only disadvantage is the lack of support for this in Internet Explorer 8 and versions below it, but they will simply fall back to regular corners. As you may have guessed by now, border-radius is a shorthand property, short for **border-top-left-radius**, **border-top-right-radius**, **border-bottom-right-radius** and **border-bottom-left-radius**. You can set these individually, or use one or several values for the border-radius property to have different values for the four corners of an element.

The border-radius property takes absolute and relative values, just like most other CSS length units, and in the next example, I'll use that to create corners so rounded that the usual squared box actually becomes a circle:

```
<style>
  .circle{
    width: 100px;
    height: 100px;
    background-color: CornflowerBlue;
    border-radius: 50%;
  }
</style>
<div class="circle"></div>
```

Pretty cool, right?

Summary

Borders are a great tool when designing your webpages and as you can see from the above examples, they are both flexible and easy to use. The amount of shorthand properties can make things a bit confusing, but after a while, you will get the hang of it.

Padding

In a previous module, we talked about margins, which is the amount of extra whitespace added to the sides of an element. Padding is just the same, but on the inside of the element. **In other words, padding is an inner, invisible border around your element.** By default, most block level elements have zero padding, meaning that text and other content will go all the way to the edge of the parent element. This is usually not desirable, especially if the parent element has borders:

```
<style>
  .box{
    border: 2px solid CadetBlue;
    background-color: Gainsboro;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

If you try the example, you will notice the text being very close to the edges of the surrounding box, which is usually not what you want. You could of course fix this by adding margin to the text element itself, but in that case, you would have to do it for every element you put into box-elements. Instead, you can just apply a padding to the box, which will push all child elements away from the borders. This is done with one or several of the padding properties called **padding-top**, **padding-right**, **padding-bottom** and **padding-left**:

```
<style>
  .box{
    border: 2px solid CadetBlue;
    background-color: Gainsboro;
    padding-top: 5px;
    padding-right: 10px;
    padding-bottom: 5px;
    padding-left: 10px;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

With that in place, there will now be a nice amount of whitespace inside the borders of the box.

Using the padding shorthand property

As for margins, paddings can also be applied with a shorthand property called "padding". It allows you to define padding values for all four sides, without having to repeat all the property names each time. The **padding property** simply directs your values out to the padding-top, padding -right, padding -bottom and padding -left properties, so it's just another way of using the CSS syntax, which will shorten your code in many situations.

The padding property can take from one to four values. This allows you to specify paddings for all four sides of an element, but if all four values are the same, you can shorten it to just one value which will be applied to all sides. If you have the same value for top and bottom padding, and another value which should be applied to both the left and right paddings, you can just have two values. Confused? Hopefully not, but here's an example where we use all of the mentioned approaches:

```
<style>
  .box{
    border: 2px solid CadetBlue;
    background-color: Gainsboro;
    width: 100px;
    height: 100px;
    margin: 10px;
  }
</style>
<div class="box" style="padding: 10px 10px 10px 10px;">
  Box
</div>
<div class="box" style="padding: 10px 10px;">
  Box
</div>
<div class="box" style="padding: 10px;">
  Box
</div>
```

Notice how I start with the most verbose way, where I declare all four values, and then work my way down.

But which order should they be in? And how does the various versions work? When specifying paddings, the following rules apply:

4 values:

1. [top padding]
2. [right padding]
3. [bottom padding]
4. [left padding]

3 values (not as commonly used):

1. [top padding]
2. [left/right padding]
3. [bottom padding]

2 values:

1. [top/bottom padding]
2. [left/right padding]

1 value:

1. [top/right/bottom/left padding]

Relative paddings

A lot of paddings will be specified in absolute values (usually pixels, like we saw in the first example), but just like most other size related CSS properties, you may also use relative values. This is usually done by either using a relative size unit, for instance the **em unit** (1 em equals the size of the current font), or simply by specifying a percentage-based value.

In the next example, I have specified a common padding of 1em, and then I have applied a different font-size, measured in em's, for each of the three boxes:

```
<style>
  .box{
    border: 2px solid CadetBlue;
    background-color: Gainsboro;
    width: 100px;
    height: 100px;
    padding: 1em;
    margin: 10px;
  }
</style>
<div class="box" style="font-size: 1em;">
  Box
</div>
<div class="box" style="font-size: 2em;">
  Box
</div>
<div class="box" style="font-size: 3em;">
  Box
</div>
```

Now if you try to run this example, you will notice that even though the boxes share the same actual padding value, the padding is now relative to the font-size, meaning that it grows when the font-size is increased. This is useful in a lot of situations!

Summary

Padding does to the inside of an element what margin does to the outside: It creates extra whitespace. In between the margin and the padding is the border, which we talked about in the previous module, and while the border will usually be visible, margin and padding is not.

One very important thing to remember about padding is that it's added to the size of the element. This doesn't really affect you if your element doesn't have a fixed size, but if it does, e.g. 100x100, and you add a padding of 10 pixels to all four sides, then the element will be 120x120 pixels instead. In other words, if your element has to be of a very specific size and you want to apply padding to it, you need to compensate for this when defining the width and height.

Outlining

We previously had a look at the border properties, which allows you to draw a border around an element. With the **outline** properties, you can get an extra border, for extra visual attention for your element. Outlining is just as easy to apply as a border - just have a look at this example for proof:

```
<style>
  .box{
    background-color: #eee;
    outline: 3px solid LightCoral;
    border: 3px solid LightBlue;
    padding: 5px 10px;
  }
</style>
<div class="box">Hello, world!</div>
```

Differences between border and outline

From our first example, the border and outline properties may look identical, but there are actually a couple of pretty important differences:

· You cannot apply a different outline width, style and color for the four sides of an element, like you can with the border - the values provided will be used for all four sides of the element.

· The outline is not a part of the element's dimensions, like the border is, meaning that no matter how thick an outline you apply to the element, the dimensions of it won't change. This also means that the browser won't reserve the required space for your outline - you will need to make sure that it can fit it, without overlapping other elements.

Shorthand property

The outline property is a shorthand property (a subject we have already covered in greater details in the previous modules) which translates into the **outline-width**, **outline-style** and **outline-color** properties. That obviously means that you can use the properties alone, if you want to:

```
<style>
  .box{
    background-color: #eee;
    border: 3px solid LightBlue;
    padding: 5px 10px;
    outline-width: 3px;
    outline-style: solid;
    outline-color: LightCoral;
  }
</style>
<div class="box">Hello, world!</div>
```

It's just another way of doing the same thing we did in the first example.

Outline color

Just a plain old color property, where you can define the color for the outline in several different ways, as it is with all color related properties in CSS. Look elsewhere in this training for a complete walkthrough of all the options you have when defining a color in CSS.

Outline width

Works much like margins and paddings - can be either an absolute value, like in this example, a relative value, or one of the pre-defined outline width values: Thin, medium or thick. If you use the pre-defined values, it's up to the browser to interpret them, which basically gives you less control of how your work will look across all the different devices and browsers.

Outline style

So far, we've just used the solid outline style, but just like for borders, there are quite a few to choose from:

hidden,**dotted**, **dashed**, **solid**, **double**, **groove**, **ridge**, **inset** and **outset**.

Let's see how they look:

```
<style>
  .box{
    outline-color: CornflowerBlue;
    outline-width: 4px;
    margin: 10px;
    float: left;
    border: 2px solid LightCoral;
    padding: 5px 10px;
  }
</style>
<div class="box" style="outline-style: dashed;">Dashed</div>
<div class="box" style="outline-style: dotted;">Dotted</div>
<div class="box" style="outline-style: double;">Double</div>
<div class="box" style="outline-style: groove;">Groove</div>
<div class="box" style="outline-style: inset;">Inset</div>
<div class="box" style="outline-style: outset;">Outset</div>
<div class="box" style="outline-style: ridge;">Ridge</div>
<div class="box" style="outline-style: solid;">Solid</div>
```

Outline offset

A cool thing about outlines is that you can create a distance between it and the border, if you want to, by using the **outline-offset** property. It takes a CSS length unit and the empty space between the border (if there is any) and the outline will be transparent and thereby take the background color of the parent element. Here's an example:

```
<style>
  .box{
    background-color: #eee;
    outline: 3px solid LightCoral;
    outline-offset: 3px;
    border: 3px solid LightBlue;
    padding: 5px 10px;
  }
</style>
<div class="box">Hello, world!</div>
```

However, please notice that the support for this property is currently a bit limited. As of writing, it is for instance **not supported by any version of Internet Explorer**, but support will likely be added in an upcoming version.

Summary

Using the outline properties, you can draw (an extra) border around an element, particularly used for extra visual attention. The outline works much like the border does, but with a couple of exceptions: You have to use the same width, style and color for all four sides and the space used by the outline is not reserved as a part of the element, like it is with the border.

Visibility & Display

So far, all the elements we have worked with has been visible, which is probably what you want most of the time. However, CSS does contain a couple of visibility related properties, which can come in handy in a lot of situations. For instance, you want to hide a part of the text until a user clicks a "Show more" button or similar use cases.

Visibility vs. Display

CSS has two properties used to control visibility: The **visibility** property and the **display** property. They are used in different situations and act differently in several ways. In this module, we'll look into both of them and see just how they differ.

The visibility property

The initial value of the visibility property is visible, simply meaning that the element is visible unless you change it - that makes sense. Now try out this example:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    background-color: CornflowerBlue;
  }
</style>
<div class="box">Box 1</div>
<div class="box" style="visibility: hidden;">Box 2</div>
<div class="box">Box 3</div>
```

Three boxes but the middle one has been made invisible by setting the visibility property to **hidden**. If you try the example, you will notice one very important thing: While the second box is not there, it does leave a pretty big hole in the page. Or in other words: The element can't be seen, but the browser still reserves the space for it!

This is one of the most important differences between the **visibility** property and the **display** property - elements hidden with the visibility property still affects layout as if it was visible, sort of like a completely transparent element, while elements hidden by the display property doesn't (the browser will treat it as if it wasn't there).

The display property

Let's try the first example of this article, but instead of the visibility property, we'll use the **display** property, so that you may see the difference:

```
<style>
  .box{
    width: 100px;
    height: 100px;
    background-color: CornflowerBlue;
  }
</style>
<div class="box">Box 1</div>
<div class="box" style="display: none;">Box 2</div>
<div class="box">Box 3</div>
```

If you try the examples, you'll immediately notice the difference: The second box has vanished without a trace, when we used the **none** value for the **display** property.

In the start of this section about the CSS Box Model, we talked about the difference between inline and block elements, how all HTML elements are born as one of them and how this can actually be changed by CSS. The property for doing so is in fact the display property, and while it is used a lot for hiding elements, it is also used for a range of other things - for instance to shift an element between the inline and block type.

In fact, if you have hidden an element by setting **display** to **none**, the way to get it back will often be to set display to either **inline** or **block** (but there are many other possible values, as you can see from this great reference page for the display property).

Remember how we talked about that the **div element** is born as a block level element and the **span element** is born as an inline element? Look how easily this can be changed with the display property:

```
<style>
  .with-background{
    background-color: CornflowerBlue;
  }
</style>
<span style="display: block;" class="with-background">Hello, world!</span>
<div style="display: inline;" class="with-background">Hello, world!</div>
```

Summary

Hiding elements on a webpage is easy with CSS - just use the visibility property if you still want to keep the space that the element would usually occupy reserved, or the display property if you want the browser to treat the element as if it wasn't there. In this module we also learned that the display property is used for more than just hiding elements, for instance to shift an element between inline and block level or one of the other types.

Introduction - width and height Properties

You may not have thought about it by now, but all elements on a webpage have dimensions. Sure, you don't have to specify them, thanks to the fluid layout model used by default - elements take up the space they need, and if there's not enough room for them, they are automatically pushed into a direction with more available space. This all happens thanks to the fact that the **width** and **height** properties of an element is set to "auto" by default, meaning that the element will automatically expand or subtract depending on the content within it.

You should be aware of the difference between inline and block elements, when it comes to dimensions, because they will behave differently: By default, an inline element will only consume the vertical and horizontal space needed to fit the content. A block element, on the other hand, will use all of the available horizontal space but only the vertical space needed to fit the content. Therefore, only block level elements can have custom widths and heights specified, as illustrated in this example:

```
<style>
  .box-look{
    background-color: Silver;
    margin: 20px;
    padding: 10px;
  }
  #box1{
    width: auto;
    height: auto;
  }
  #box2{
    width: 100px;
    height: 100px;
  }
</style>
<div id="box1" class="box-look">
  Box 1<br>
  Some content....
</div>
<div id="box2" class="box-look">
  Box 2<br>
  Long text is automatically wrapped...
</div>
<span class="box-look">
  Inline content. Only uses the space it needs...
</span>
```

If you try out this example, you can see how the first box uses all available horizontal space, while the second box only use the 100 pixels we assign it in both directions. Div elements are by default block level elements. Please be aware that I have only specified width and height for box1 as **auto** to illustrate the difference - these are already the default values, so they may be omitted.

The last element is a span tag, which is by default an inline element and as you can see, it only uses the space actually required. You could try assigning a width and a height to it, but the browser would ignore it - only block level elements can have custom widths/heights.

Relative and absolute width/height

We already saw how we could define widths and heights as an absolute value, in pixels, but that is just one of many options. The width and height properties can take either a [length] or a [percentage] value, meaning that you can use both absolute values (as we did with pixels) or relative values, either as a percentage of the available space or relative e.g. to a font size (like the *em* unit - more on that later on).

In the next example, I'll show you how we can use percentage values to take up a relative share of the available space:

```
<style>
  #parent-box{
    width: 300px;
    height: 300px;
  }
  #box1{
    width: 25%;
    height: 75%;
  }
  #box2{
    width: 75%;
    height: 25%;
  }
</style>
<div id="parent-box" style="background-color: CornflowerBlue;">
  <div id="box1" style="background-color: GreenYellow;">
    Box 1<br>
    Some content....
  </div>
  <div id="box2" style="background-color: Salmon;">
    Box 2<br>
    Long text is automatically wrapped, if needed...
  </div>
</div>
```

In this example, we have a box with an absolute size acting as a parent box, and inside of it, we have two smaller boxes, which uses a relative amount of the available parent space, just to show you how easy it is to do.

Summary

In this module, we talked about how block level elements on a webpage will automatically adjust their sizes and how we can force them to be of a specific size by using the width and height properties. But what happens if we specify a width and a height which doesn't leave enough room for the content? We'll look into that in the next chapter.

Minimum and maximum width/height

So far, we have seen how we can give an element relative and absolute sizes, by using the **width** and **height** properties. However, another possibility exist: Specifying minimum and/or maximum sizes. For this purpose, four properties exist: **min-height**, **max-height**, **min-width** and **max-width**. Their purpose should be easy to defer from their names, but how and when to use them might not be equally clear.

Specifying minimum and/or maximum sizes for an element allows you to take advantage of the fluid nature of elements on a webpage, allowing a certain element to expand and subtract within a limited set of dimensions, instead of defining a constant and absolute size like we did with the *height* and *width* properties. Just like width and height, the min-* and max-* properties allows you to specify an absolute or relative length unit or a percentage based size.

Minimum width and height

Using the min-width and min-height properties, you can define the smallest possible size that an element can have. If the element doesn't have width and/or height defined (auto), or if these values are smaller than the defined minimum height and width, **the minimum height and width values will overrule it**. This can easily be illustrated by an example:

```
<style>
  .box{
    width: 50px;
    height: 50px;
    background-color: DarkSeaGreen;
    padding: 10px;
    margin: 20px;
    float: left;
  }
</style>
<div class="box">
  Box 1 - Default
</div>
<div class="box" style="min-height: 80px; min-width: 80px;">
  Box 2 - Minimum height and width
</div>
<div class="box" style="min-height: 80px; min-width: 80px;">
  Box 3 - Minimum height and width. Only expands to 80px.
</div>
<div class="box" style="min-height: 80px; min-width: 80px; height: auto;">
  Box 4 - Minimum height and width. Only expands to 80px, unless no max-height/height
have been specified (auto)
</div>
```

If you test the example, notice how the second box is bigger by default, even though width and height is originally set to 50px - in this case, the min-width and min-height values take precedence.

Now look at the third box. It has more content than the second, and while the minimum values would normally allow it to grow, the fact that it has now outgrown both its height and min-height values caps it at the biggest value of the two (80px). If we want to allow it to grow beyond this, the height and/or width should not be defined. We illustrated this with the fourth box, where we reset the height value to auto, which is the default value, if it wasn't set by the .box selector - with that in place, the element is now allowed to grow vertically to make room for the content.

Maximum width and height

Sometimes you want to restrain an element to a specific width and/or height. Obviously this can be done with the width and height properties, but this will force the element to stay at this size all of the time - no matter if the content requires it or not. By using the max-width and/or max-height properties instead, you can allow an element to be whatever size it requires to fit the content and then grow along with the content up until a certain point.

Here's an example where we use max-height and max-width:

```
<style>
  .box{
    background-color: DarkSeaGreen;
    padding: 10px;
    margin: 20px;
    float: left;
  }
</style>
<div class="box">
  Box 1 - Default
</div>
<div class="box" style="max-height: 100px; max-width: 100px;">
  Box 2 - Maximum height and width
</div>
<div class="box" style="max-height: 100px; max-width: 100px;">
  Box 3 - Maximum height and width, this time with more content
</div>
```

Notice how each of the three boxes act differently, depending on whether or not max values are specified and how much space the content requires.

Summary

The max and min width/height properties give you an extra layer of control in addition to the regular width and height properties. Please be aware that if both min-height, max-height and height are specified, min-height takes precedence over both, while max-height only takes precedence over height. The exact same is true for the width-related properties.

Overflow

As already explained, a block level element in CSS will, by default, take up all the horizontal space as well as all the vertical space it needs to fit the content inside of it. However, what happens if we define a vertical size that is not enough to contain the content? Let's have a look:

```
<style>
  #box1{
    width: 100px;
    height: 100px;
  }
</style>
<div id="box1" style="background-color: GreenYellow;">
  Some content....
  More content...
  There's probably not enough room for it in this tiny box...
</div>
```

Try out this example and you will soon realize an important thing: Whenever you decide to set a fixed height on an element, you are responsible for making the content fit, and if it doesn't, it will simply "fall out" of its container. Obviously, you wouldn't notice it in this example if it wasn't for the distinct background colors (the green box on the white background), but it's still a very real problem that you will likely face as soon as you try to limit the size of your elements. So, what can be done about this?

The overflow property

With a little help from the **overflow** property, you can control what happens when an element has larger content than it can actually fit. There are several possible values and each of them will make your element behave differently when deciding what to do with overflowing content. Here's an example that will illustrate it:

```
<style>
  .box{
    width: 80px;
    height: 80px;
    background-color: DarkSeaGreen;
    padding: 10px;
    margin: 20px;
    float: left;
  }
</style>
<div class="box" style="overflow: visible;">
  Box 1 - A box with many, many words in it, designed to cause overflow
</div>
<div class="box" style="overflow: hidden;">
  Box 2 - A box with many, many words in it, designed to cause overflow
</div>
<div class="box" style="overflow: auto;">
  Box 3 - A box with many, many words in it, designed to cause overflow
```

```
</div>
<div class="box" style="overflow: scroll;">
  Box 4 - A box with many, many words in it, designed to cause overflow
</div>
```

The overflow has, per the current specification, four possible values: Visible, hidden, scroll and auto. The default is **visible**, which is the behavior we saw in the first example - the content simply just expands beyond the border of the container. This is the value used for the first box, but it could have been omitted, since it is the default value.

For the second box, we use the **hidden** value. This will cause content which expands beyond the borders to simply be hidden, making it invisible to the end-user. This can be practical in some cases, but in the case of a long text not fitting into an absolutely sized element, you would likely prefer the **auto** or **scroll** value.

The **auto** value, used on the third box, leaves it up to the browser how to handle the problem. Most browsers, at least desktop browsers, will handle overflow by adding scrollbar(s) if necessary. This is very often the value you will prefer, because it allows you to have an absolutely sized element and then only have scrollbars rendered if the content actually overflows.

At last, we have the **scroll** value. Most desktop browsers will handle this value by adding vertical and/or horizontal scrollbars to the element, if content overflows. Scrollbars are often what you desire, but with this value, most browsers will force scrollbars upon your element, no matter if the content overflows or not - for this reason, the **auto** value is usually preferred.

Summary

With absolutely sized elements, you are likely to run into problems with overflowing content. You can solve this by using the overflow property, as seen in the examples above, but you can also solve it by using relatively sized elements, for instance by adjusting the size of the parent element according to font size instead of an absolute value. This will help you in cases where your elements contain text and the user changes the font size in the browser. Which solution to go for very much depends on the situation though.

Introduction to Positioning

As mentioned several times in this training already, the positioning of elements on a webpage is very fluid. Unlike printing, you don't have a fixed size that you can adjust everything to, because websites are visited from screens in all sizes, from small cell phones to huge desktop monitors, all running in different resolutions. For that reason, elements in a webpage are automatically laid out right after each other, in the order in which they are specified. **This positioning model is called static and all elements use it by default.** Positioning is controlled mainly by using the **position property**, which can have the following values:

Position: static

Static is the default way of positioning elements, so you only have to set the position property to static to override a previously set position. In the static mode, your elements are placed from top to bottom, within the available space of their parent, while respecting any margins set on the element itself or the surrounding elements.

Position: relative

As the name indicates, relative elements can be moved around within their parent element. By default, if you set the position property to relative, the element is placed as if it were static, but you can then use the **top**, **bottom**, **left** and **right** property to move it around by setting these values relative to the parent element.

Position: absolute

An absolute positioned element is placed exactly where you want it to within the browser window, by adjusting the top, bottom, left and/or right properties. This means that you can place elements e.g. in the top, right corner, even if your main page has been tied to the middle of the screen. The exception to this is if you place an element with an absolute position inside of a relatively positioned element - in that case, the child element will be confined to the space of the parent element instead of the entire window. All of this will of course be demonstrated in the upcoming module about absolute positioning.

Position: fixed

By using the fixed property, your element will be positioned as if it were absolute, with one very important addition: When the user scrolls, the fixed element will remain in this position all the time. This allows you to create, for instance, a top menu which remains on screen even when the user scrolls through down through your page.

Floating

Besides the position property, floating of elements also causes the position to be changed - we use the **float property**, which will also be described in this part of the training.

Summary

So, as you can see, CSS supports several different positioning techniques, all of them which will be discussed in more details in the upcoming modules. Especially absolute positioning, where you take complete control of where an element is placed, is interesting. You should however be very aware of the fact stated above: Your website is very likely to be rendered on screens in MANY different sizes, so if you decide to use a different positioning approach than the default, you should do so carefully! Read on for a thorough walkthrough of all the possible positioning methods.

Relative Positioning

A relatively positioned element will, as the name indicate, be positioned relative to its parent element. The parent is the element containing the child element, which can of course be a specifically defined element or the entire page (in this case, the body tag is likely the parent). By default, if you set the **position** property to **relative**, the element will act as if it were static:

```
<style>
  #parent-div{
    background-color: #6c9;
    width: 200px;
    height: 200px;
  }
  #relative-div{
    background-color: #6495ed;
    position: relative;
    width: 100px;
    height: 100px;
  }
</style>
<div id="parent-div">
  <div id="relative-div">
  </div>
</div>
```

However, the real magic starts happening when you use the position related properties called **top**, **bottom**, **left** and **right**. By default, these are set to **auto**, which basically results in the element being placed in the top, left corner. You can easily change this though:

```
<style>
  #parent-div{
    background-color: #6c9;
    width: 200px;
    height: 200px;
  }
  #relative-div{
    background-color: #6495ed;
    position: relative;
    width: 100px;
    height: 100px;
    top: 50px;
    left: 50px;
  }
</style>
<div id="parent-div">
  <div id="relative-div">
  </div>
</div>
```

Notice how the child element has now been pushed 50 pixels from the top and the left. For relatively positioned elements, the **top** and**left** properties tells the browser how far **below** the normal position you want it, while the **bottom** and **right** properties tells the browser how far **above** the normal position you want the element. This might be a bit confusing at first, especially because the properties work in a more logical way when it comes to absolutely positioned elements, but you will soon get the hang of it.

This also means that while you CAN use negative left and top values to move an element away from its parent, it has the same effect as giving the element the same bottom and right values, but in this case, positive numbers instead of negative. To illustrate, check out the next two examples where I use both approaches and achieve the exact same result:

Using negative values for the left and top properties:

```
<style>
  #parent-div{
    background-color: #6c9;
    width: 200px;
    height: 200px;
    padding: 10px;
  }
  #relative-div{
    background-color: #6495ed;
    position: relative;
    width: 200px;
    height: 200px;
    left: -20px;
    top: -20px;
  }
</style>
<div id="parent-div">
  <div id="relative-div">
  </div>
</div>
```

Using the bottom and right properties instead:

```
<style>
  #parent-div{
    background-color: #6c9;
    width: 200px;
    height: 200px;
    padding: 10px;
  }
  #relative-div{
    background-color: #6495ed;
    position: relative;
    width: 200px;
    height: 200px;
    bottom: 20px;
    right: 20px;
  }
</style>
```

```
<div id="parent-div">
  <div id="relative-div">
  </div>
</div>
```

So, as you can see, which method to use is really up to you!

How does relative positioning affect surrounding elements?

That's an excellent question but the answer is also quite simple: It doesn't. When a browser meets a relatively positioned element, it will simply push it in the direction you instruct it to (by using top/left or bottom/right properties), but it will still just allocate the space that the element was originally meant to take up. This becomes very clear with the next example:

```
<style>
  .box{
    background-color: #6c9;
    padding: 20px 10px;
    width: 200px;
  }
  #relative-div{
    background-color: #6495ed;
    position: relative;
    width: 200px;
    height: 200px;
    left: -10px;
    top: 20px;
  }
</style>
<div class="box">Top</div>
<div id="relative-div">
  Relative element
</div>
<div class="box">Bottom</div>
```

We now have surrounding elements, which uses static positioning (since this is default) and in the middle, an element with a relative position. We use the left and top property to push it away from the top element, and look what happens: The element is pushed away from the top element, as expected, but the bottom element is not pushed away, which means that the two elements will overlap each other. Why? Because the relatively positioned element is still only claiming the space it would have if it was statically positioned!

Summary

As you can see, relative positioning has some advantages as well as some disadvantages. However, in combination with absolute positioning, it can be even more useful, as we will see in the next module.

Absolute Positioning

With static positioning, which is the default positioning model used on a webpage, everything is laid out in the same order as you define them in your HTML code, from top to bottom, and everything stays within its parent container. With absolute positioning on the other hand, you get to dictate exactly where in the browser window you want an element to appear, even on top of other elements, if that's what you want.

Absolute positioning is accomplished by using the **position** property and setting the value to **absolute**. However, this is not enough - you need to use one or several of the position-related properties **left**, **right**, **top** and **bottom**. This allows you to control the precise position of the element and without any of them, an absolutely position element will revert to a behavior very similar to a static element.

So, let's try adding an element with an absolute position, to see how it acts:

```
<style>
  #absolute-div{
    position: absolute;
    width: 100px;
    height: 100px;
    background-color: #6495ed;
    top: 35px;
    left: 35px;
  }
</style>
<div id="absolute-div">
  Absolutely!
</div>
Hello, world!<br>
This is a great example!
Hello, world!<br>
This is a great example!
```

Try running this example and notice two very important things which likely differs from what you're used to from HTML and CSS: While the text is defined after the box, the box is still rendered over the second sentence, and also, it is allowed to be displayed on top of the text. This is absolute positioning in a nutshell - you can place elements anywhere and even have them overlap other elements.

In this case, we used the left and top properties to offset our element from the top, left corner, but bottom and right can also be used, if we want to specify the distance from the bottom and/or right corner. In addition to that, you can of course combine them like you please, as you can see from this next example:

```
<style>
  .box{
    position: absolute;
    width: 100px;
    height: 100px;
    background-color: #6495ed;
  }
  .right{
    right: 10px;
  }
  .left{
    left: 10px;
  }
  .top{
    top: 10px;
  }
  .bottom{
    bottom: 10px;
  }
</style>
<div class="box top left">Top, Left</div>
<div class="box top right">Top, Right</div>
<div class="box bottom left">Bottom, Left</div>
<div class="box bottom right">Bottom, Right</div>
```

Here, we define CSS selectors for top, bottom, left and right elements and then we combine them to place elements in all 4 corners of the screen - pretty easy, right?

Overlapping and the z-index property

So, with all this freedom, what happens if two elements overlaps each other? We already saw this happening in the first example of this article but let me make it very clear with another example:

```
<style>
  .box{
    position: absolute;
    width: 100px;
    height: 100px;
  }
</style>
<div class="box" style="background-color: #5f8426;">Box 1</div>
<div class="box" style="background-color: #6495ed;">Box 2</div>
<div class="box" style="background-color: #68968a;">Box 3</div>
```

If you try the example, you will only see box number 3. Why? Because they all have the same positions (default values for top and left), so the last declared element takes precedence and is placed on top of the other. This might not be what you want though and as luck would have it, CSS has a property for controlling this: The z-index property.

With the z-index property, we take full control of which elements are on top. It holds a numeric value, which defaults to 0, and the higher z-index value an element has, the more on top it will be, as illustrated by this example:

```
<style>
  .box{
    position: absolute;
    width: 80px;
    height: 80px;
    padding: 10px;
  }
</style>
<div class="box" style="background-color:#5f8426;top:10px;left:10px;z-index:3;">
  Box 1
</div>
<div class="box" style="background-color:#6495ed;top:60px;left:60px;z-index:1;">
  Box 2
</div>
<div class="box" style="background-color:#68968a;top:100px;left:100px;z-index:2;">
  Box 3
</div>
```

Notice that by using the z-index property, I can now define the elements in the order I want to and still have them overlap in another order. As a side note, feel free to use more distance between the z-index values, if you want to make room for more elements in between, e.g. 10, 20, 30 and so on.

Absolute, but relatively

Sometimes you may want to position an element absolutely, but instead of have it positioned within the borders of the browser window, you want it to remain within its parent element. The normal behavior, as we have seen, is to use the browser window, but if the parent element has its **position** property set to **relative**, all this changes. Now, I could give you an example with a couple of boxes inside of another box, but instead, I decided to show you how far you can get with positioning, by creating a smiley face:

```
<style>
  #face{
    position: relative;
    background-color: AntiqueWhite;
    width: 200px;
    height: 200px;
    border-radius: 50%;
    left: 20px;
    top: 20px;
  }
  .eye{
    position: absolute;
    background-color: #6495ed;
    width: 50px;
    height: 30px;
    top: 40px;
    left: 40px;
    border-radius: 40%;
  }
  #nose{
    position: absolute;
    background-color: DarkSalmon;
    width: 20px;
    height: 40px;
    top: 85px;
    left: 95px;
  }
  #mouth{
    position: absolute;
    background-color: DarkSalmon;
    width: 110px;
    height: 8px;
    bottom: 50px;
    right: 45px;
  }
</style>
<div id="face">
  <div class="eye"></div>
  <div class="eye" style="left: 120px;"></div>
  <div id="nose"></div>
  <div id="mouth"></div>
</div>
```

Granted, this guy is NOT pretty, but it's still pretty cool how absolute positioning allows you to almost make a small painting, right? Notice how I establish a frame (the face) by setting its position to relative and then place absolutely positioned elements within this frame - this is a very useful technique!

Summary

Absolute positioning is very powerful, especially when used in combination with a relatively positioned element, but also to place elements, for instance, in the corners of the user's browser. If you have overlapping elements then be sure to use the z-index property to control which element gets to be on top.

Fixed Positioning

The fixed position is quite interesting. It works pretty much like absolute positioning, with one important difference: An element with a fixed position stays in the designated place, even if you scroll up or down. This allows you to make "sticky elements", like menus, which are always at the **top**, **bottom**, **left** or **right** of the viewport, no matter how far you scroll.

Another important difference is that while you may have an absolute positioned element be positioned relative to its parent element (see the article on absolute positioning if you're curious), this is not possible with a fixed element - it's always placed in relation to the viewport (usually the browser window).

So, how is it used? Here's an example:

```
<style>
  .top-fixed{
    position: fixed;
    top: 0;
    right: 0;
    background-color: #eee;
    padding: 3px 10px;
  }
</style>
<div class="top-fixed">Fixed element (Top)</div>
<p style="margin-bottom: 12000px;">Filler text</p>
```

Don't worry about the filler paragraph - it's simply there to make sure that the viewport is high enough for you to see the effect of the fixed element when scrolling.

Using the fixed value for the position property, along with the top and right properties, we can now tie our little box to the top, right corner and leave it there, even when the user scrolls. This is excellent for creating e.g. fixed top menus!

Be aware of the fact that just like with absolutely positioned elements or floating elements, a fixed element is taken out of the regular page flow, meaning that the browser no longer automatically allocates space for it. This can be seen in the above example, where our fixed element overlaps other elements on the page. **To remedy this, you simply need to make sure that your page has the available space for your fixed elements, for instance by adding margins and paddings to surrounding elements.** If you actually want the elements to overlap, you can control which element takes precedence using the z-index property, as described in the module about absolute positioning.

Fixed elements on other sides

You can tie your elements to all four sides of the viewport, just as easy as we did in the above example. It's simply a matter of using the top, right, bottom and/or left properties, depending on where you want the element. Here's an example:

```
<style>
  .fixed{
    position: fixed;
    background-color: #eee;
    padding: 3px 10px;
  }
</style>
<div class="fixed" style="top: 0; left: 100px;">Top</div>
<div class="fixed" style="top: 100px; right: 0;">Right</div>
<div class="fixed" style="bottom: 0; left: 100px;">Bottom</div>
<div class="fixed" style="top: 100px; left: 0;">Left</div>
<p style="margin-bottom: 12000px;">Filler text</p>
```

Notice how easy it is - as long as the element has the position set to fixed, you can start using the top/bottom and/or left/right properties to position the element accordingly.

Summary

Fixed positioning is a powerful tool for creating sticky elements, which just means that even when the user scrolls through your page, the fixed elements remain in the same position within the viewport. This can be used in many scenarios, like sticky top menus, sharing buttons to the left or right which remains in place and so on.

Floating Elements

A very powerful positioning technique is floating. It allows you to take a block level element out of the normal order and let other elements float around it. For instance, this can be used to place an image to the left or the right, while letting text float around it, as seen in magazines and newspapers. However, this is not how HTML and CSS works by default, when you place an image next to some text. Just have a look at this example to see what I mean:

```
<style>
  .container{
    width: 300px;
    background-color: Gainsboro;
    padding: 10px;
    text-align: justify;
  }
</style>
<div class="container">
  <img src="../images/logo.jpg">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed auctor placerat
metus, sit amet egestas dui rutrum sed. Aliquam eget fringilla metus, vel pulvinar
turpis. Mauris consectetur purus convallis nibh consectetur, nec mattis nisi
euismod.
</div>
```

As you can see, the text starts at the bottom right corner of the image, instead of just using up all the free space to the right of the image and this would be even worse if you tried to line up other block level content next to the text. The simple solution to this is to allow the image to float, either to the left or the right. This will "pull up" surrounding elements so that they can use the remaining space.

Now let's add a bit of floating magic to the first example, so that you can see the difference:

```
<style>
  .container{
    width: 300px;
    background-color: Gainsboro;
    padding: 10px;
    text-align: justify;
  }
  .image{
    float: left;
    margin: 0 10px 10px 0;
  }
</style>
<div class="container">
  <img src="../images/logo.jpg" class="image">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed auctor placerat
metus, sit amet egestas dui rutrum sed. Aliquam eget fringilla metus, vel pulvinar
turpis. Mauris consectetur purus convallis nibh consectetur, nec mattis nisi euismod.
</div>
```

Notice how the text is now moved up so that it uses the available space to the right of the image. That's because the image is now floating and therefore allows other elements to be displayed next to it - both inline and block level elements. Try changing the value of the **float** property from **left** to **right** and see how easy it is to get that cool, magazine/newspaper-like look!

Clearing after a float

When you start floating your elements, you will very likely soon run into a common problem: When you have a floating element, all content after it will move up next to it, if there's enough space. Let me illustrate it with an example:

```
<style>
  .container{
    width: 300px;
    background-color: Gainsboro;
    padding: 10px;
    text-align: justify;
  }
  .image{
    float: left;
    margin: 0 10px 10px 0;
  }
  .footer{
    background-color: Azure;
    text-align: center;
  }
</style>
<div class="container">
  <img src="http://www.css3-training.net/images/csstraining_logo.png"
class="image">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed auctor placerat
metus, sit amet egestas dui rutrum sed.
  <div class="footer">Footer</div>
</div>
```

In this case, we want the footer element to take up all the space after the image of the text, but because the text doesn't take up enough room, the footer is moved up right below the text and then partly overlapped by the image. The solution to this problem comes from the **clear** property. It's specifically designed to clear existing floats, so that elements are once again laid out like you would expect from HTML and CSS.

The clear property takes one of three different values: **Both**, **Left** or **Right**. This tells the browser if you want to clear only floats from the left or the right or if you want to clear floats from both directions. Look how easy we can fix the before mentioned issue simply by adding the clear property:

```
<style>
  .container{
    width: 300px;
    background-color: Gainsboro;
    padding: 10px;
    text-align: justify;
  }
  .image{
    float: left;
    margin: 0 10px 10px 0;
  }
  .footer{
    background-color: Azure;
    text-align: center;
    clear: both;
  }
</style>
<div class="container">
  <img src="http://www.css3-training.net/images/csstraining_logo.png" class="image">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed auctor placerat
metus, sit amet egestas dui rutrum sed.
  <div class="footer">Footer</div>
</div>
```

The trick here is to utilize the clear property right before the content which you want to stay away from the floated content. In this case, we can simply apply it to the footer element, because it holds the content that we want out of the floated area, but in other situations, you may want to simply add an invisible div element to stop floating - this will be demonstrated in the next example.

The problem with collapsed parent element

Another very common problem with floats is the issue of the collapsing parent element. Consider this example:

```
<style>
  .parent{
    background-color: Gainsboro;
  }
  .child{
    float: left;
    margin: 10px;
    background-color: CornflowerBlue;
    width: 50px;
    height: 50px;
    padding: 5px;
  }
</style>
<div class="parent">
  <div class="child">Box</div>
  <div class="child">Box</div>
  <div class="child">Box</div>
</div>
```

Try it out and see if you can spot the problem. That's right - **the parent element is not visible at all!** You should have been able to see it because of the gray background color, but instead, the three floating elements just sit there as if they had no parent.

Why? **Because floating elements does not reserve any space in its parent element and since there are only floating element inside the parent div, it collapses as if it was empty** . If you add a word below the three child divs, you will actually be able to see the parent element, but it now only reserves space for this word (the only non-floating element), meaning that the parent will only be as high as a single line of text, with the three boxes hanging out of the bottom.

The solution? Actually the exact same one as we used above. By clearing all floats before the end of the container, the browser is forced to reserve space inside of it right down to the clearing element. Here's a working example:

```
<style>
  .parent{
    background-color: Gainsboro;
  }
  .child{
    float: left;
    margin: 10px;
    background-color: CornflowerBlue;
    width: 50px;
    height: 50px;
    padding: 5px;
  }
  .clear{
    clear: both;
  }
</style>
<div class="parent">
  <div class="child">Box</div>
  <div class="child">Box</div>
  <div class="child">Box</div>
  <div class="clear"></div>
</div>
```

Notice how we just added the empty clearing div at the bottom of the parent element to fix this problem. It simply uses the CSS **clear** property to clear floats from both directions and you will likely be using this technique quite a bit when you start working with floating elements.

Summary

Floating is an extremely powerful technique that you will likely end up using quite a bit - it quickly becomes essential when you want to do more detailed layouts. However, it does come with a small price of added complexity. We have already showed you a couple of the "extraordinary" behaviors you will run into when using floats and there are more of them out there.

So, if your webpage starts acting strange, then always look out for any sort of positioning, especially floats - they are often to blame, because they simply make it harder for the browser to interpret what you want.

Fonts & Text - Introduction

It is said that almost 95% of the information on the web is written and therefore it is only logical to say that any web designer should have an extensive knowledge of typography. Whenever you optimize the typography of your website, you also increase the accessibility, readability and usability - and who wouldn't want that?

Previously typography had been reserved for the print world, but in the last couple of years the focus on web typography has gained great attention. And this is the topic of the following modules - how to style the text on your website in order to gain the look and feel you are striving for and adding a few typographic hints, rules and insights into what is considered best practice when it comes to web typography.

Font-family - the basics

The first thing to set the tone of our typography is the font. Generally, there are two types of fonts - serif and sans-serif. The difference? The serif fonts have feet and the sans-serif doesn't. Here is an example

```
<p style="font-family:serif;">This is your browsers default serif font</p>
<p style="font-family:sans-serif;">This is your browsers default sans-serif font</p>
```

There has been a lot of fuzz about whether to use a serif or sans-serif when designing for the web, as there might have been a problem with the readability, but research has shown that with modern screens this is not a problem and you can actually choose the type of font you prefer.

Web safe fonts - More than just serif or sans-serif

As you probably know, there are a lot more than just two types of font out there and all webdesigners have their favorite - I have a thing for Georgia and Century Gothic, where as you might prefer other fonts. Over at CSSfontstack.com they have a neat list of all the *web safe fonts* out there and I have included a couple of them here - oh, and they work on both Windows and Mac!

```
<p style="font-family: Impact, Charcoal, sans-serif; ">Impact - a bold font</p>
<p style="font-family: Century Gothic, sans-serif; ">Century Gothic, a personal
favorite</p>
<p style="font-family: Times New Roman, Times, serif; ">Times New Roman - the classic
choice</p>
```

Setting the font family for the entire page

To set the font family for the entire page, you need to set it for the body of the document - this way all text on your webpage is rendered in this font;

```
body{
  font-family: Times New Roman, Times, serif;
}
```

Here is an explanation of what this snippet of CSS means - This declaration tells the browser to look for the font 'Times New Roman' and if this is not installed, use Times. If the browser doesn't have any of those use whatever generic serif font it does have.

As you can see, some of the properties have more than one value - in this example we've used three values. This is because of the fact that the browser needs a fallback, should the given font not be installed on our computer. So, the font you prefer is the first value and the list ends with a generic font name, such as in this case 'serif'. Oh, and please note that all values are separated by a comma.

What you have learned

- Generally, there are two types of fonts: serif and sans-serif
- Some fonts are considered websafe and you should opt for them in the first place.
- Always include a generic font name when defining font-families, your user might not have the preferred font installed on their device.

Font-weight

If you have ever done any word processing using e.g. Word you definitely know of the opportunity to use bold, italics and underlined text. When writing CSS, these characteristics are spread over several properties and this is why I have grouped them together into a single module.

The font-weight property

The font-weight property defines how bold you text are and there are a lot of possible values; normal, bold, bolder, lighter, 100, 200, 300, 400, 500, 600, 700, 800, 900, and inherit. Let's have a look at the easy-to-understand part - the numerical scale consisting of the numbers 100-900. These values range from lightest (100) to boldest (900).
Here is a rough guide to match the numerical scale with the most common weight terms:

- **100:** Thin, Hairline, Ultra-light, Extra-light
- **200:** Light
- **300:** Book
- **400:** Regular, Normal, Plain, Roman, Standard
- **500:** Medium
- **600:** Semi-bold, Demi-bold
- **700:** Bold
- **800:** Heavy, Black, Extra-bold
- **900:** Ultra-black, Extra-black, Ultra-bold, Heavy-black, Fat, Poster

Here is an example - I've used classes to style the two paragraphs differently but you could just as well have used inline style;

```
<style>
  .weight200{font-weight:200 }
  .weight800{font-weight:800 }
</style>
<p class="weight200">To travel is to live - H. C. Andersen </p>
<p class="weight800"> To travel is to live - H. C. Andersen </p>
```

As you can see, there is a difference between the two sentences. But you should be aware that most webfonts do not have more than two or three weights, and weight such as 'light' or 'semi-bold' is rare. The value '400' equals normal and the value '700' equals bold.

Furthermore, most browsers does not render the font-weight correctly and only differentiate between normal and bold - have a look at the following example and you can see the problem for yourself;

```
<p style="font-weight: 100;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 200;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 300;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 400;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 500;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 600;">To travel is to live - H. C. Andersen </p>
```

```
<p style="font-weight: 700;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 800;">To travel is to live - H. C. Andersen </p>
<p style="font-weight: 900;">To travel is to live - H. C. Andersen </p>
```

If you want to have the opportunity to use more weights, you should have a look at GoogleFonts as a part of their fonts have 3-4 different weights.

The difference between font-weight:bold and font-weight:bolder

What is the difference between font-weight:bold and font-weight:bolder you might ask? The 'bolder' and 'lighter' values select font weights that are relative to the inherited (parent) font weight where as the 'bold' value simply change the font's weight to bold. This comes in handy if you use a font-family with three or more weights.

Below is an example of how you could use the 'bolder' value to emphasize sentences visually.

```
<style>
  p{font-weight: normal;}
  .bolderText{font-weight:bolder;}
</style>
<p>For people could close their eyes to greatness, to horrors, to beauty, and their
ears to melodies or deceiving words. <span class="bolderText">But they couldn't
escape scent</span>.</p>
<p >And scent entered into their very core, went directly to their hearts, and
decided for good and all between affection and contempt, disgust and lust, love and
hate. </p>
<p><span class="bolderText">He who ruled scent ruled the hearts of men</span>.</p>
```

As you can see, the sentences highlighted with the 'bolderText' class is actually bolder.

The font-style Property

The font-style property defines whether or not your text is italic or oblique. But what is the difference between italic and oblique? If typography is a part of your everyday work-life you probably know this, otherwise, here is an explanation: oblique is normally a roman font that has been skewed a certain number of degrees (from 8-12 degrees normally) automatically whereas an italic is created by the font designer to achieve a more calligraphic and slanted version of the font. This affects you only if you want to use an italic - you have to make sure that the font you are using actually have been designed with one, otherwise you get the oblique version and you risk a rather unpleasant sight.

Below is an example of how italic vs oblique is rendered (as you can see, it is same, which means that the font I am using does not have an italic and therefore the result isn't as pleasing for the eye as one might have hoped)

```
<style>
  .styleItalic{font-style: italic }
  .styleOblique{font-style: oblique }
</style>
<p class="styleItalic">To travel is to live - H. C. Andersen </p>
<p class="styleOblique"> To travel is to live - H. C. Andersen </p>
```

If you want to dig deeper into typography and the history hereof, I propose you continue to ILoveTypography as they are some of the best in the field and write terrific articles!

What you have learned

- You use font-weight to define whether or not your text should be bold
- You use font-style to determine whether or not your text should be italic
- Not all of CSS's font-weight property's values are useful, as most free web fonts does not contain nine different weights
- A lot of Google's fonts have three weights and therefore they might be just what you need for headings and such
- Unfortunately, not all web fonts have an italic and therefore you might end up with the oblique which is automatically generated whereas the italic is designed specifically to be skewed towards the right

Font Size

In my opinion, one of the most confusing parts of CSS styling to get the hang of was the font-size property. The large number of possibilities are the problem - in CSS you are given a lot of different measurement units in order to define the size of your font. But which one is the best? I am not to be the judge of which unit is best, but I do have an opinion concerning which unit you should choose.

The 'em' is a scalable unit designed specifically for web media. The 'em' is designed to take the users preset font-size into consideration, this means, that 1em is always the users preferred font-size. 'Why would you want that?' one might ask. Well, a lot of people prefer a larger font-size due to vision impairment and if you try to force a smaller font-size on them, they might leave your site without ever reading a single sentence. So, using 1em as your standard font size you probably heighten the user experience to those with vision impairment.

But why don't all use the em measurement you might ask? Well, there are two problems: some people prefer pixel-perfect layouts and then a fluid measurement as the em isn't a good choice.

How big is an em?

According to W3C 'an em is equal to the computed value of the 'font-size' property of the element on which it is used. The exception is when 'em' occurs in the value of the 'font-size' property itself, in which case it refers to the font size of the parent element.' In other words - an em does not have a fixed size.

Furthermore, it can be hard to get grip of how big a '2em' text is as the measurement is scalable. If the user has not changed any settings regarding font-sizing, then 1em normally equals 16 pixels and 2em equals 32 pixels. If you are used to using pixels when defining your font-size this might be a hard transition. Here is an example for you to explore how the em unit scales:

```
<style>
  .textLarge{font-size: 2em;}
  .textRegular{font-size:1em;}
  .textSmall{font-size:0.7em;}
</style>
<p class="textLarge"> To travel is to live - H. C. Andersen </p>
<p class="textRegular"> To travel is to live - H. C. Andersen </p>
<p class="textSmall"> To travel is to live - H. C. Andersen </p>
```

Font-size - the percentage measurement unit

A lot of people choose to use percentages when defining the size of their font. The reason is obvious: we all know percentage beforehand and it is easy to understand that a font of the size '150%' is 50% larger than a font size '100%'.

If you choose to use percentages as your measurement unit, you can use it in two ways: You can choose to make your font-sizing relative to the users' preferences. You do this by adding a declaration to the body:

```
body{font-size:100%;}
```

When you do this, you tell the browser to use the users preset font size as your standard font size and then you can scale all of your other text according to this.

```
<style>
  body{font-size:100%;}
  h1{font-size:170%;}
  p{font-size: 130%}
  .smallerText{font-size: 50%;}
</style>
<h1>The Perfume by Süskind</h1>
<p>For people could close their eyes to greatness, to horrors, to beauty, and their
ears to melodies or deceiving words. <span class="smallerText">But they couldn't
escape scent</span>.</p>
```

The difficulty with scaling - parent elements!

You need to be aware of one thing! When scaling font-sizes **you scale according to the parent element**! This means, that the current font size is always equal to 100% when you scale the next font size. Let's have an example to show why this might become a problem:

```
<style>
  body{font-size:100%;}
  h1{font-size:170%;}
  p{font-size: 130%}
  .smallerText{font-size: 50%;}
</style>
<h1>The Perfume by Süskind</h1>
<span class="smallerText">A short excerpt</span>.
<p>For people could close their eyes to greatness, to horrors, to beauty, and their
ears to melodies or deceiving words. <span class="smallerText">But they couldn't
escape scent</span>.
</p>
```

As you can see, there are two instances of the class 'smallerText' and you would assume that they had the same size, right? Well, unfortunately that's not how it works. The parent of the first instance is 'body' where as the parent of the second instance is the <p> tag. This means that the first instance is scaled to be **50% of the body** whereas the second instance is scale to **50% of the paragraph**! If you take a look at the example, you immediately see the size difference - the first instance is a lot smaller than the second.

TIP!
Oh, you should be aware that this scaling 'problem' also applies if you choose to use em's instead of percentages.

Font-size - em vs percentage?

So, which on should you choose - ems or percentages? Well, that is entirely up to you and it depends on which measurement system 'feels' right for you. But I believe that you should fiddle around with both units in order to get to know both systems.

What you have learned

- The em measurement unit is specifically designed for web use
- 2 ems is double the size of 1 em
- Many people find it easier to use percentages as this is a unit we are familiar with - but it behave just like ems!
- Whether you use em's or percentages you scale the text based on the parent element. This means, that if you have a lot of elements nested inside of each other you need to tread carefully in order to achieve the result you want

Advanced Typography - wordspacing, letterspacing and textalign!

So now you've got the basics of typography down, let's have a look at how you style the text for your liking. Typography is a very big part of your entire website's look and therefore it is nice to know how you get the text to behave just the way you like it.

Text-align - a neat feature

In the western world, we are so used to left-aligned text, that text-align is a property we hardly notice - except for one place, when we are reading an old-fashioned paper, where the text are justified in order to make the columns of the paper appear more 'clear'.
Let's have a look at the four options; right, left, center or justify;

```
<style>
  p{text-align: right;}
</style>
<p>For people could close their eyes to greatness, to horrors, to beauty, and their
ears to melodies or deceiving words. But they couldn't escape scent.<br />
For scent was a brother of breath. Together with breath it entered human beings, who
couldn't defend themselves against it, not if they wanted to live. And scent entered
into their very core, went directly to their hearts, and decided for good and all
between affection and contempt, disgust and lust, love and hate. <br />He who ruled
scent ruled the hearts of men.
</p>
```

As you can see, this looks quite odd when using the Latin alphabet but if you were to use the Arabic alphabet this would be the kind of text flow that you were used to.

The 'center' value should be used with cause - most people find it harder to read centered text (as they are not used to it), but under some circumstances, such as with e.g. headings, the 'center' value might be a good choice;

```
<style>
  h3{text-align: center;}
</style>
<h3>Breaking news - I'm centered!</h3>
<p>For people could close their eyes to greatness, to horrors, to beauty, and their
ears to melodies or deceiving words. But they couldn't escape scent.</p>
<p>And scent entered into their very core, went directly to their hearts, and decided
for good and all between affection and contempt, disgust and lust, love and hate. He
who ruled scent ruled the hearts of men.
</p>
```

The last value we'll have a look at is the 'justify' option. If you choose to use this, you should be aware that the space between both each letter and the words may be stretched or squeezed and make the entire text seem 'off'. Therefore, you should reconsider your use if you choose to use justify with a text with very long words, short paragraphs, or extremely short sentences - but let's have a look at how the justify works; (the paragraphs of this example are a bit too short for my liking when it comes to justified text, but it is a matter of taste.)

```
<style>
  p{text-align: justify;}
</style>
<p>For people could close their eyes to greatness, to horrors, to beauty, and their
ears to melodies or deceiving words. But they couldn't escape scent.</p>
<p>And scent entered into their very core, went directly to their hearts, and decided
for good and all between affection and contempt, disgust and lust, love and hate. <br
/>He who ruled scent ruled the hearts of men.
</p>
```

Word-spacing - perfect for headings

Word-spacing is one of those features that should only be used for headings, block quotes, or other non body text as it can have a nice visual effect but decreases the readability. CSS treats every character or set of characters with a space around them as a 'word'. The measurement unit for word-spacing is em - as 1 em is the height of the letter x, then a word-spacing of 1.5em equals 1.5 x'es. The following example illustrates how word spacing might be a good choice for headlines and seldom are used for body text;

```
<style>
  h2{word-spacing: 1.2em;}
  .tooMuch{ word-spacing: 1.2em;}
</style>
<h2>Proper use of word-spacing</h2>
<p class="tooMuch">For people could close their eyes to greatness, to horrors, to
beauty, and their ears to melodies or deceiving words. But they couldn't escape
scent.
</p>
```

Letter spacing - also perfect for headings

You use the letter-spacing property to control the amount of space between the letters. You can even use negative values in order to get the letters to stand even closer. But the question is - should you? Just like with word-spacing I recommend only using this feature for headings, block quotes, maybe menu items, and so forth, but never ever body text. Take a look at how the letter-spacing property works;

```
<style>
  h2{letter-spacing: 0.25em;}
  .tooMuch{ letter-spacing: 0.5em;}
</style>
<h2>Proper use of letter-spacing</h2>
<p class="tooMuch">For people could close their eyes to greatness, to horrors, to
beauty, and their ears to melodies or deceiving words. But they couldn't escape
scent.<p>
<p>And scent entered into their very core, went directly to their hearts, and decided
for good and all between affection and contempt, disgust and lust, love and hate. <br
/>He who ruled scent ruled the hearts of men.
</p>
```

Text-transform

As the name says, text-transform transforms your letters - they can be transformed to uppercase, lowercase, or capitalize. The syntax is straight forward:

```
p{text-transform: uppercase;}
```

This is what the three values look like - I have used an inline stylesheet in order to style the text:

```
<p style="text-transform: uppercase;">An example of uppercase text.</p>
<p style="text-transform: lowercase;">An example of lowercase text.</p>
<p style="text-transform: capitalize;">An example of capitalized text.</p>
```

An example of some simple typograpgy

Up until now we have looked individually at some of the effects, you can achieve with the tools of the typographic tools-box, but how can you combine the text-align, word-spacing and all the other neat effects without creating a big mess? In order to give you some inspiration I have assembled an example - it is a nice mix of text-transform and letter-spacing combined with some padding and borders. Have a look at the example;

```
<style>
  h2{
    font-size: 1.7em;
    text-transform:uppercase;
  }
  p:first-line{
    text-transform:uppercase;
  }
  .introduction{
    letter-spacing: 0.25em;
    text-transform:uppercase;
    border-bottom: 1px solid white;
    border-top: 1px solid white;
    padding-bottom: 1em;
    padding-top: 1em;
  }
</style>
<h2>The Perfume - An excerpt</h2>
<p class="introduction">For people could close their eyes to greatness, to horrors,
to beauty, and their ears to melodies or deceiving words. But they couldn't escape
scent.<p>
<p>And scent entered into their very core, went directly to their hearts, and decided
for good and all between affection and contempt, disgust and lust, love and hate. <br
/>He who ruled scent ruled the hearts of men.
</p>
```

As you can see, letter-spacing and text-transform is used to highlight specific parts of your text and hereby making it easier for your users to skim your text in order to get an overview of your text. And this is how all of these features should be used - to make the text easier for your reader to understand!

What you have learned

- You can align your text in four different ways; right, left, center or justify.
- Whenever you choose to use either center or justify you should be aware that it decreases the readability of your text for some users.
- Word spacing should normally only be used for headlines - using it on body text may decrease the readability of your text.
- The same applies for letter-spacing - use these two with care!
- All of these effects should be used with one purpose in mind - to make the text easier for your readers to understand!

Text Shadow

In CSS3 a new feature has been introduced - the text shadow! When using the text-shadow property we have control over three things - the color, the placement and the blur. This means that there are four values:

- color
- x-coordinate
- y-coordinate
- blur radius

You can use either HEX colors of RGBa colors when you define the color. The coordinates are based on the placement of the text and can be either positive or negative. The amount of blur defines how 'washed-out' your shadow is.

Here's a simple example;

```
<style>
  .simpleShadow{
  color: #000;
  font-size: 2em;
  text-shadow: 1px 1px 5px lightgray;
}
</style>
<p class="simpleShadow">Simple shadow effect</p>
```

Okay, so know you know that it is possible to add backgrounds to your text, but it is not really that impressive, is it? Well, this is why I have included *a number of fancy text-shadow effects* to show you some of the possibilities (and they are plentiful!)

Embossed!

Now you're impressed, right? Here is what the CSS looks like for this type of typography. Oh, and please note that you can actually *add multiple text-shadow to every element*. In this example I've added a lightgray one (-1px -1px 1px lightgray) and a black one (2px 2px 1px #000). If you use this styling you need to change the color of the font to one that matches your background!

Text Decoration

The **text-decoration** property is actually quite self-explanatory: It allows you to decorate your text in various ways. It is most commonly used to underline your text:

```
<div style="text-decoration: underline;">Hello, world!</div>
```

However, it actually has several other possibilities. For instance, you can use it to create a strikethrough effect on your text:

```
<div style="text-decoration: line-through;">Hello, world!</div>
```

Now, the cool thing about the text-decoration property is that it allows you to apply several values at once:

```
<div style="text-decoration: overline underline;">Hello, world!</div>
```

Try it out

As the final example, allow me to show you all the possible values in the same example, with a bonus line using all of them simultaneously:

```
<style>
  div{
    margin: 30px 0;
  }
</style>
<div style="text-decoration: overline;">Hello, world!</div>
<div style="text-decoration: line-through;">Hello, world!</div>
<div style="text-decoration: underline;">Hello, world!</div>
<div style="text-decoration: underline overline line-through;">Hello, world!</div>
```

Summary

The text-decoration property makes it easy to add lines around or through your text, but bear in mind that it's not a border - the line(s) drawn by the text-decoration property will always be the same color as the text itself. If you need a line in a different color below or above your text, you should use the border properties.

Text Indent

By using the text-indent property, you can offset the start of the first line of text with a certain amount of whitespace. It's very easy to use, as illustrated by this example:

```
<style>
  p{
    text-indent: 50px;
    width: 200px;
    background-color: Silver;
  }
</style>
<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Sed condimentum augue sed diam semper rhoncus.
  Curabitur porttitor mattis tortor, eget aliquam lectus porta ac.
</p>
```

This first example used an absolute value, but especially if you're already using a relative font size unit like the em unit, specifying the text-indent in the same unit makes sense:

```
<style>
  p{
    font-size: 1em;
    text-indent: 2.5em;
    width: 200px;
    background-color: Silver;
  }
</style>
<p>
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Sed condimentum augue sed diam semper rhoncus.
  Curabitur porttitor mattis tortor, eget aliquam lectus porta ac.
</p>
```

You can also use a percentage-based text-indent. In that case, the offset will be calculated based on the width of the parent element:

```
<style>
  .box{
    width: 200px;
    background-color: Silver;
  }
  p{
    text-indent: 50%;
  }
</style>
<div class="box">
  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Sed condimentum augue sed diam semper rhoncus.
    Curabitur porttitor mattis tortor, eget aliquam lectus porta ac.
  </p>
</div>
```

Summary

The text-indent property will easily give you that cool, first line indentation that many magazines and newspapers have been using for centuries.

The text-align Property

In CSS, you can control the alignment of text much like you can in a word processor like MS Word. This is done using the text-align property, which has several different values. The most commonly used ones allow you to adjust the text to the left, the right or the center. Here's an example:

```
<p style="text-align: left;">Left aligned text</p>
<p style="text-align: center;">Center aligned text</p>
<p style="text-align: right;">Right aligned text</p>
```

Left is the default, so we may as well omit that, but I have included it in the example for consistency. You may also need it to revert to left aligned text, since this is inherited from the parent element, as illustrated with this example:

```
<div style="text-align: center;">
  This div element contains centered text
  <div style="text-align: left;">
    but also left aligned text!
  </div>
</div>
```

As you can see, we specifically instruct the second div element to use left alignment - if we didn't do that, the text in this element would be centered, because it inherits it from the parent div element.

Align justify

Besides the ability to align text to the left, center or right, the text-align property comes with a possible value called **justify**. If you use this option, the browser will try to make all lines of text within the element the same length, usually by adjusting the amount of space between each word. This layout is very commonly seen in newspapers and magazines and is usually implemented to improve readability. Here's an example of the justify option in action:

```
<p style="text-align: justify; width: 200px;">
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a lectus mattis,
consequat mi vitae, tristique ipsum. In hac habitasse platea dictumst. Integer sit
amet aliquet dolor.
</p>
```

When testing this example, you should see that all lines of the text within the paragraph line up on the left and the right side. You might also see why this is not always used - the amount of whitespace being added can be quite high, resulting in some strange looking lines.

Summary

With the text-align property, you can very easily define alignment for your text elements. The most commonly used values are left, right and center, but the justify option can also come in handy in some cases where you want the lines to align in both ends.

Introduction to colors in CSS

Up until now, we haven't really dealt with one of the most basic and yet most important aspects of CSS: Colors! The reason for this it that while it's very easy to define a text or background color in CSS, there are so many ways to denote a color that it requires a bit more explanation.

So, in the next couple of modules, we'll first have a look at how easy it is to alter colors in CSS, and then we'll deal with the theory behind these color notations.

Text & Background color

In this article, we'll discuss just how easy it is to apply both text and background colors to your web pages. Let's start off with the text color.

Changing the text color

If you have already read the modules on text and font in CSS, then you might be tempted to take a guess at the name of the property used for controlling text color - how about font-color or text-color? No, actually, because what we're changing is often referred to as the foreground (text) and background, the CSS property used to control the color of text is simply called **color**.

So, how can we use it? It's very, very simple - just have a look at this example:

```
<p style="color: Red;">Red text</p>
<p style="color: Blue;">Blue text</p>
```

And that's all you need. As you can see, I'm using so-called named colors. CSS has a whole bunch of these, but of course not every color in the world is covered by a name - more on that later on.

Changing the background color

Working with the background color is just as easy. For that purpose, we have a property called **background-color**. It allows us to change the background color of an element just as easy as we changed the text color, and the two properties can of course be combined:

```
<p style="background-color: Silver; color: Red;">Red text</p>
<p style="background-color: Gold; color: Blue;">Blue text</p>
```

As you can see, the two properties go well with each other, but they don't have to be used simultaneously, if you are already happy with the current background or text color. **Since both text and background colors are inherited from their parent element**, you can easily be in a situation where you would only need to use one of them:

```
<p style="background-color: Silver; color: Red;">
This is red text and <span style="background-color: Gray;">and this text has a darker
background</span>.
</p>
```

Notice how the span element keeps the red text color inherited from the parent paragraph element, while we override the background color with a darker variant of the silver color.

Summary

Changing both text and background color is very easy with CSS. So far, we have only shown some basic examples, but to go beyond that and use more advanced color combinations, you need to know a bit more about the way color values are constructed with CSS. We'll discuss that in the next module.

How colors work in CSS

We have already had a peak at how to change text colors and background colors, but so far, we have only used named colors. Obviously there can't be a name for each of the millions of colors that the human eye can perceive, so besides named colors for the most common colors, CSS comes with several different ways of expressing a color value.

RGB colors

Colors on computers are usually made up as a mix of the three core colors: Red, Green and Blue. Each of these can have a numeric value between 0 and 255, expressing the amount of the color in question to use in the mix. In CSS, you can define a color using a specific RGB syntax, like this:

```
<p style="color: rgb(0,0,0);">Black text</p>
```

Zero red, green and blue results in the black color. At the other end of the scale, we obviously have white, which is represented by a red, green and blue value of 255:

```
<p style="color: rgb(255,255,255); background-color: rgb(0,0,0);">
  White text
</p>
```

As you can see, the syntax is fairly straight forward - the word rgb followed by a set of parentheses, which surrounds the three values for Red, Green and Blue, each separated with a comma.

Adding transparency with RGBa

As an extension to RGB, you can use RGBa, which allows you to define a fourth value which defines the level of transparency applied to the color in question. This allows you to get a semi-transparent element and is frequently used to create see-through overlays on websites.

The last part of an RGBa set needs to be a number between 0 and 1, where 0 is fully transparent and 1 is a completely solid color. I'm going to show you an example, but to give you a better impression about how it works, I'm going to use an image background - don't worry, this will be covered in one of the next modules.

```
<style>
  .background{
    padding: 20px;
    background-image: url('../images/random-grey-variations.png');
  }
  .box{
    width: 200px;
    height: 100px;
    background-color: rgba(128,128,128,0.3);
    padding: 10px;
    color: white;
  }
</style>
<div class="background">
  <div class="box">
    This is a semi-transparent box
  </div>
</div>
```

Notice how the box has a transparent layer, on which you can have text and images, allowing you to see the background image behind it - pretty cool, right? You can regulate the last part of the RGBa value to make the box more or less transparent, depending on your needs.

HEX colors

As an alternative to the RGB notation, you may use HEX notation, which consists of a hash character (#) followed by either 6 or 3 characters/numbers. Each set of characters represent the hexadecimal representation of the Red, the Green and the Blue component of the color. Allow me to illustrate with an example:

```
<p style="color: #000000;">Black text</p>
<p style="color: #0000ff;">Blue text</p>
<p style="color: #ff0000;">Red text</p>
```

You can read the values by discarding the hash character and then dividing the remaining 6 characters up into sets of two. Each set defines the values for R, G and B, using the hexadecimal notation.

In cases where all three sets consists of the two same characters, individually, you can use the shorthand notation by only writing the first character for each set. So for instance, black becomes #000, blue becomes #00f and red becomes #f00.

The HEX notation may seem a bit harder to use, because you will have to convert the R, G and B numbers into the hexadecimal values, but most web editors can help you with this. They often do so by offering a color picker to let you pick the color visually and by converting between named color values, RGB colors and HEX colors for you. Also, remember that HEX values are not case sensitive - #F00 is the same as #f00.

Named colors

As already mentioned, CSS comes with support for a wide range of named colors. These can be easier to remember than HEX or RGB values. For a complete list of these colors, please have a look at one of the many online references, e.g. http://www.cssportal.com/css3-color-names/.

Summary

As you can see, there are many ways to define colors when using CSS. Which one to use really depends on the situation and your personal preference. Some people insist that only one type of color notation is used, other people mix and matches depending on the context and the color they're looking for. It's really up to you!

Background Images

We already looked at the ability to define a different background color than what is used by default, but with CSS, we can actually use images for the background as well. Using the background-image property, along with several helper properties, we can completely control this useful aspect of CSS.

But, why would you use a background image on an element instead of just inserting an image through the HTML image tag? That's an easy question to answer, because with CSS background images, you are allowed to put content on top of the image or even use an image as the background for the entire browser window, while only using a fixed portion of it for content.

In the first example, I'm going to show you just how easy it is to use a background image with a regular div element:

```
<style>
  .box{
    background-image: url('../images/logo.jpg');
    width: 350px;
    height: 200px;
    text-align: center;
    font-weight: bold;
    font-size: 2em;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

The essential part here is of course the use of the **background-image** property. Notice how the value prefixed with the **url** keyword and then enclosed inside a set of parentheses as well as a set of quotes. The quotes can, in principal, be omitted, but they are required as soon as you need to have any space characters in the path, so it's usually easier to just make a habit out of using them.

The actual URL can be any type of image that the most common webbrowsers can use, so JPEG, PNG and GIF are obvious and very commonly used.

Defining a fallback background color

In the above example, we use an image hosted on this server, but for it to work everywhere, I have specified the full path. You should however never rely on images hosted on servers outside of your control, for two good reasons: They won't appreciate the fact that you're using their bandwidth, and as a result of that, they might delete or rename the image file, in which case your site won't look as intended.

However, even if you host the image on your own server, as you should be doing, the browser can't display the background until the image has been downloaded. This might be slow, either because your server is under a lot of pressure or because the background image is very big or a combination of the two, and in the meantime, no background will be displayed for your element.

Actually, this is only true if no background has been specified for a parent element, since this value will be inherited, but for instance, if you have defined a dark background image for your body element, the default background color is likely white. This will make the transition from the default background (white in most browsers) to your background image (in this case a dark pattern) seem a bit harsh if the browser is having trouble with fetching the image fast enough.

For this reason, it can be a good idea to specify a background color as well as a background image. By specifying the background color before the image, the browser immediately uses the defined color and then replaces it with the image as soon as the browser has fetched it. Here's an example:

```
<style>
  .box{
    background-color: #ebf1f3;
    background-image: url('../images/logo.jpg');
    width: 350px;
    height: 200px;
    border: 1px solid black;
    text-align: center;
    font-weight: bold;
    font-size: 2em;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

In this case, I have used a background color found in the actual background used - this will make the transition as smooth as possible. You won't likely even see it, but if the image was bigger, the server busier or if the user is on a slow connection, this will be a great improvement. This will also help you in situations where the image can't be loaded for one reason or another - it simply gives you more control.

Summary

In this article we learned how to use an image as a background for an element. Remember that even though we just used a simple div element in these examples, you can use the background-image for pretty much any HTML element, including the body element!

We also learned that it's a good idea to define both a background color and a background image, in that order. The background color will be used as a fallback value, in case the image doesn't exist or if the browser is having problems fetching the image fast enough.

If you download the image used in the first example, you will realize that it's only 100 pixels wide and 100 pixels high, but the box containing the background is twice as wide as that, so why does the background still fill out the entire space? The secret lies in the ability for CSS to repeat a background - in fact, it will do so by default. This entire concept will be discussed in depth in the next article.

The background-repeat Property

CSS can repeat the background image you specify, to fill out the entire container. In fact, this is the default behavior, which is great for a lot of situations. This allows you to create a small, textured background image and then have it repeated as many times as needed, to fill out the available background. We already saw this behavior in the first example, but what are the alternative?

Background repetition is controlled through the **background-repeat** property, which has several possible values. Here's an example showing you them all:

```
<style>
  .box{
    background-image: url('../images/durian.jpg');
    width: 200px;
    height: 150px;
    margin: 10px;
    border: 1px solid black;
    text-align: center;
    font-weight: bold;
    font-size: 2em;
  }
</style>
<div class="box" style="background-repeat: repeat;">Repeat</div>
<div class="box" style="background-repeat: repeat-x;">Repeat-X</div>
<div class="box" style="background-repeat: repeat-y;">Repeat-Y</div>
<div class="box" style="background-repeat: no-repeat;">No-Repeat</div>
```

In the first box, we use the default value of repeat, which basically means that the background image will be repeated in both directions. In the next two boxes, we use repeat-x and repeat-y - the first one repeating the background only horizontally, while the latter only repeats vertically. The last box uses the no-repeat value, which means that the image will be displayed as it is. In any case, if the background is actually bigger than the containing element, then the image will be clipped - only the required portion of it, to fill the containing element, will be used, with the rest being invisible.

Summary

The background-repeat property is very powerful, allowing you to define exactly how and even if you want a background image to be repeated inside an element. In the fluid world of web design, where elements usually don't have a fixed size, this is really great. However, when you don't want any repeating to occur, you may need more control of where the background is placed. This can be achieved with the background-position property, which we'll discuss in the next article.

The background-position Property

In the previous article, we discussed all about the background-repeat property and how repeating the background image in both directions is the default behavior. In case you don't want the background to be repeated, it can be quite useful to control the position of the background. As you can see from the previous examples, the default position is the top, left corner, but this can be changed very easily, through the use of the **background-position** property:

```
<style>
  .box{
    background-image: url('../images/durian.jpg');
    background-repeat: no-repeat;
    background-position: top center;
    width: 200px;
    height: 150px;
    border: 1px solid black;
    text-align: center;
    font-weight: bold;
    font-size: 2em;
  }
</style>
<div class="box">
  Hello, world!
</div>
```

Notice how I use two values, separated by a space, to indicate the desired position. The reason is that the background-position property takes a value of the type [position], which is used to indicate a 2D location. This means that we can use either a single keyword to indicate a side, e.g. top, a two keyword value, like I did in this example, or even a percentage or length value.

In the example above, I want the background to be placed in the center position, aligned to the top of the element, but there are many other possibilities when positioning the background. Just check out this next example, where I show you some of the many combinations:

```
<style>
  .box{
    background-image: url('../images/durian.jpg');
    background-repeat: no-repeat;
    background-position: top center;
    width: 200px;
    height: 150px;
    border: 1px solid black;
    text-align: center;
    font-weight: bold;
    font-size: 1.5em;
    margin: 10px;
  }
</style>
```

```
<div class="box" style="background-position: top;">
  Top
</div>
<div class="box" style="background-position: top right;">
  Top, right
</div>
<div class="box" style="background-position: bottom left;">
  Bottom, left
</div>
<div class="box" style="background-position: 10% 50%;">
  Percentage based
</div>
<div class="box" style="background-position: 10px 30px;">
  Length based
</div>
<div class="box" style="background-position: 50% bottom;">
  Combination
</div>
```

Notice how we can use either one or two values, and we can use the position keywords (top, bottom, left, right and center) as well as percentages and lengths - we can even combine them like in the last example!

Summary

The flexibility of the background-position property allows you to position your backgrounds very precisely. Since the property accepts length and percentage-based values, negative values are of course accepted as well. This is used frequently when combining several images, e.g. icons, into a large image (to save bandwidth and browser connections) and then only showing the relevant part of the image.

CSS3 Transitions - Introduction

This is one of the modules I have been most excited about - *the introduction to CSS3 transitions*! CSS transitions are a presentational effect which allow you to create property changes which occur on e.g. mouse-over or mouse-click.
The CSS3 transitions modules introduces four new properties;

- Which property or properties to target - transition-property
- The duration of the given transition - transition-duration
- The timing function of the transition - transition-timing-function
- And an optional possibility to delay the transition - transition-delay

CSS3 transition-property

The transition property specifies which property the transition is applied to - be it color, background, border, and so forth!

CSS3 transition-duration

This property specifies how many second or milliseconds a transition takes to complete. The default is 0 and a good advice is to never specify a duration shorter than 5-6 milliseconds as the human eye will have troubles recognizing this as transition instead of just a change.

Css3 transition-timing-function

With the transition timing function, you can change the speed-curve of the transition effect.

- Linear
- Ease-in - Specifies a transition with a slow start
- Ease-out - Specifies a transition with a slow end
- Ease-in-out - Specifies a transition with a slow start and end.
- Cubic-beizer(n,n,n,n) - Defines your own transition timing. Possible values ranges from 0 to 1. As this is one of the functions you probably won't ever have to use I am not going to go deeper into it.

Luckily, all of these four transitions can be combined into just one property - *the transition property*.

I think it is about time to show an example, right? The following example is a transition from the color 'Crimson' to 'CornFlowerBlue' with a transition which takes 2 second, eases in (speeding up) and no delay. If we used the individual transition properties, it would look like this:

```
transition-property: all;
transition-duration: 2sec;
transition-timing-function: ease-in;
transition-delay: 0;
```

Using the combined method, the transition property, it looks like this;

```
transition: all 2s ease-in 0;
```

Using the individual transition-declarations is a good idea when you are new to CSS3 as it tells you exactly which value you are working with, but using the combined declaration may be timesaving in the lon run. (If you are using a smart webeditor, you will be writing code so efficiently that it doesn't really matter which option you choose.)

Simple background transition

```
<style>
  .bg-transition{
    padding: 15px;
    text-align: center;
    background-color: Crimson;
    width: 90%;
    min-height: 25px;
    transition: all 2s ease-in 0;
    -webkit-transition: all 2s ease-in 0;
  }
  .bg-transition:hover{
    background-color: CornFlowerBlue;
  }
</style>
<p class="bg-transition"> </p>
```

This simple color transition can be used on text and borders as well and adds a nice detail to e.g. link-hovers. Instead of using the background-color property you just use the color or the border property - and remember to change it in the :hover pseudo class too.

Styled background transition

```
<style>
  .bg-transition{
    padding: 15px;
    border: 10px solid #2f2f2f;
    font-size: 30px;
    text-align: center;
    font-family: 'Yanone Kaffeesatz', sans-serif;
    background-color: rgba(163,169,169,0.3);
    transition: all 2s;
    -webkit-transition: all 1.2s ease-in;
  }
  .bg-transition:hover{
    background-color: rgba(47,47,47,1);
  }
</style>
<p class="bg-transition">P-O-W-E-R-!</p>
```

Tip!: Adding a 20-30 millisecond delay to your transitions is a good idea if you have a group of transitions right next to each other. Your user will not actually see the delay but it will make the transitions seem steadier if all of the transitions are triggered within few milliseconds.