CSS - Basics

- Dr Nick Hayward

A brief introduction to the basics of CSS.

Contents

Intro

A **Cascading Style Sheet**, or CSS, allows us to define stylistic characteristics for our HTML. In effect, it helps us define how our HTML is displayed and rendered. The colours used, font sizes, borders, padding, margins, links, and so on.

CSS syntax

CSS follows a defined syntax pattern, which may be defined as follows

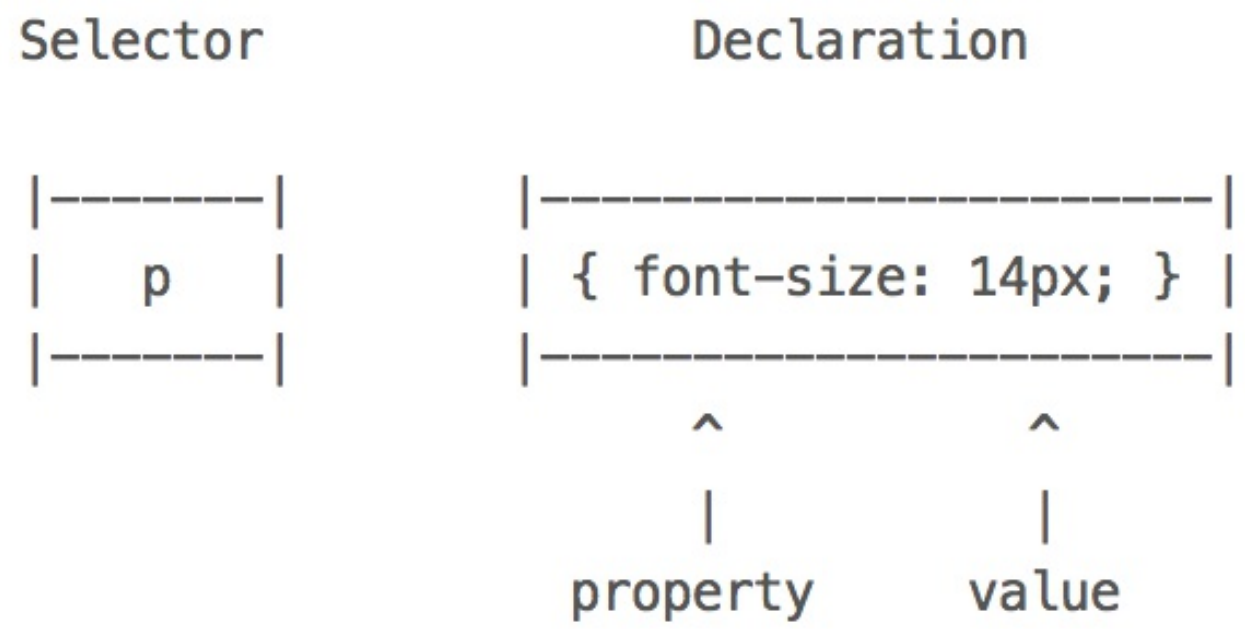
- selector
  - e.g. `body` or `p`
- declaration
  - property and value pairing

For example,

```
body {
  color: black;
  font-family: "Times New Roman", Georgia, Serif;
}
```

So, in this example `body` is the selector, `color` is the property, and `black` is the value.

Image - CSS Syntax



rulesets

In essence, a CSS file is a group of rules for styling our HTML documents. These rules form **rulesets**, which can be applied to elements within the DOM of our HTML documents.

Rulesets consist of the following,

- a selector
  - eg `p`
- an opening brace
- `{`
- a set of rules
- `color: blue`
- a closing brace
- `}`

So, each rule can be considered as containing a specific *property*, followed by a colon, plus our required value (or list of values separated by spaces), followed by a semi-colon. So, as seen earlier, our ruleset might look as follows,

```
body {
  width: 900px;
  color: #444;
  font-family: "Times New Roman", Georgia, Serif;
  }
```

Our example includes a selector for the `body` element, and our required rules. This set of rules will be applied to all content of the `body` element, including any elements contained within the `body` element. This ruleset, therefore, contains three rules, each respectively specifying a width property, a value for the colour property, and the available fonts for our text content.

As you've probably noticed, colour can be set to a named value (e.g. `blue`) or HEX value (e.g. `#444`). There's a fun and useful colour picker at the following URL,

- HTML Colour Picker

comments

We can also add comments to help describe the selector and its properties,

```
  /* add styling to paragraphs */
  p {
    color: blue;
    font-size: 14px;
    }
```

Comments can be added before the selector or within the braces.

display

For the majority of cases, we can display HTML elements in one of two ways. We can use *inline*, which applies to elements such as `<a>` or `<span>`, to display content that will appear contiguously on the same line as the surrounding content.

For example,

```
<div class="content">
  <p>
    <a href="...">Philae</a> is a <span>Ptolemaic</span> era temple in Egypt.
  </p>
</div>
```

However, it's more common to display elements as `block-level` instead of `inline` elements. Basically, this means that the element's content will be rendered on a new line outside the normal flow of our content. A few sample block elements include

- article, div, figure, main, nav, p, section...

It's interesting to note that *block-level* is not technically defined for new elements in HTML5.

Display and elements

We may consider the display of elements using CSS relative to the following:

inline

Current inline elements include:

- b | big | i | small | tt
- abbr | acronym | cite | code | dfn | em | kbd | strong | samp | var
- a | bdo | br | img | map | object | q | script | span | sub | sup
- button | input | label | select | textarea

Source - MDN - Inline Elements

n.b. not all inline elements supported in HTML5

block-level

Current block-level elements include:

- address | article | aside | blockquote | canvas | dd | div | dl
- fieldset | figure | figcaption | footer | form
- h1 | h2 | h3 | h4 | h5 | h6
- header | hgroup | hr | main | nav | noscript
- ol | output | p | pre | section | table | tfoot | ul | video

Source - MDN - Block-level Elements

n.b. *block-level* is not technically defined for new elements in HTML5

CSS selectors

Effective use and manipulation of **selectors** is a crucial aspect of working with CSS and JavaScript as well. We've already encountered basic selectors, such as

```
p {
  color: #444;
}
```

Don't forget , `p` is the selector, `color` is the property, and `#444` is the value

The above ruleset is adding basic styling to our paragraphs, setting the text colour to HEX value *444*, a dark grey. This is simple, and easy to apply, but it applies the same properties and values to all paragraphs. If we want to be more specific, for example, we need to use classes, psuedoclasses, and other selectors.

classes

We can add a **class** attribute to an element, such as a `<p>` or a `<span>`, thereby helping us differentiate elements. We can also add a **class** to any DOM element. For example, we could now add different classes to multiple `<p>` elements,

```
<p class="p1">paragraph one...</p>
<p class="p2">paragraph two...</p>
```

By adding these classes, we can now select our paragraphs by class name within the DOM. We can then apply a **ruleset** for each class. We can be specific, and style this class for a specific element

```
p.p1 {
  color: #444;
}
```

Or, we could simply style the class itself

```
.p1 {
  color: #444;
}
```

The above example will style all elements with the class `p1`, and not just `<p>` elements with that class. Classes are very useful for styling groups of elements together, or abstracting styling to one ruleset and applying to multiple different elements.

pseudoclasses

As with **classes** in the previous example, we can add a class to links or anchors, thereby styling all links with the same ruleset. However, we might also want to add specific styles for different link states. For example, styling links with a different colour depending upon whether a link has already been *visited* or not.

CSS allows us to add an additional ruleset for such an example,

```
a {
  color: blue;
}

a:visited {
  color: red;
}
```

In the above example, `visited` is a CSS **pseudoclass** applied to the `<a>` element. The main difference is that a browser implicitly adds this pseudoclass for us, we just need to provide the styling in the CSS. We don't need to add anything to the HTML for the link, for example.

Likewise, we can use another pseudoclass to help us style our links when a user hovers their cursor over an `<a>` element. So, we can now style our links for **hover**

```
a:hover {
  color: black;
  text-decoration: underline;
}
```

Again, the browser will implicitly set this pseudoclass in the HTML for us.

complex selectors

As we build our websites and applications, our DOM will often become more complicated and detailed. The depth and complexity will require more complicated selectors as well. If we consider lists, and their varying depths

```
<ul>
  <li>unordered first</li>
  <li>unordered second</li>
  <li>unordered third</li>
</ul>

<ol>
  <li>ordered first</li>
  <li>ordered second</li>
  <li>ordered third</li>
</ol>
```

So, we now have two lists, one unordered and the other ordered. We can style each list, and the list items, as we've already seen. We simply create our rulesets for the given selectors.

For example,

```
ul {
  border: 1px solid #ddd;
}

ol {
  border: 1px solid #ccc;
}
```

We could also add a single ruleset for the list items, `<li>`, but we'd be applying the same style properties to both types of lists. If we wanted to be more specific, we could apply a ruleset to each list item for the different lists. For example,

```
ul li {
  color: blue;
}

ol li {
  color: green;
}
```

Now, it might also be useful to set the background for specific list items in each list, therefore making it easier to visualise the first list item, for example

```
li:first-child {
  background: #eee;
}
```

We can take this another step, and set a pseudoclass of `nth-child` thereby allowing us to specify a style for the second, fourth &c. child in the list.

```
li:nth-child(2) {
  background: #ddd;
}
```

For a bit of fun, we could take this even further and style odd and even list items to create a useful alternating pattern. This is particularly useful, as you might imagine, for styling tables. For example,

```
li:nth-child(odd) {
  background: #bbb;
}
li:nth-child(even) {
  background: #ddd;
}
```

For a table, we could simply update the selectors for each pseudoclass to reflect a `<tr>`.

We can take this yet another step, and select only certain list items, or rows in a table &c., such as selecting every fourth list item, starting at the first one. For example, our CSS would be as follows

```
li:nth-child(4n+1) {
  background: green;
}
```

In fact, for the even and odd children we are simply leveraging the above in a pre-configured package. Other examples include

- `last-child`
- `nth-last-child()`

and so on.

Cascading rules

As the name suggests, CSS, or cascading style sheet, employs a set of **cascading** rules, which are applied by each browser if and when a ruleset conflict arises.

For example, consider the simple issue of specificity. If we create a ruleset for a `<p>` element, and then another for a `<p>` element with a class `p1`, the more specific rule, the class, will take precedence.

```
p {
  color: blue;
  }

p.p1 {
  color: red;
}
```

However, there is also the issue of possible duplication in rulesets. If we had two rulesets for the same element, for example

```
h3 {
  color: black;
}

h3 {
  color: blue;
}
```

the **cascading** rules of CSS dictate that the later ruleset in the CSS list will be the one applied.

For this example, we would end up with `<h3>` elements styled in blue.

So, hopefully, you can already see how simple styling and rulesets can quickly become compounded and complicated. Different styles, specified in different places, can interact and affect each other in complex ways. This can become a powerful feature of CSS, but it can also create many issues with logic, maintenance, and design.

Therefore, we can often consider three primary sources of style information, relative to our documents, that form this cascade. They include,

- default styles applied by the browser for a given markup language
  - e.g. colours for links, size of headings, and so on...
- styles specific to the current user of the document
  - often affected by browser settings, device, mode...
- styles linked to the document by the designer
  - as we've seen, such styles can be linked in three ways including an external file, embedded in a definition at the beginning of the document, and as inline styles per element.

The basic cascading nature of these options means that the following applies,

- browser's style will be default
- user's style will modify the browser's default style
- the styles of the document's designer will then modify the overall styles further

So, as we read documents in a browser, our styles might be applied in a cascading nature from multiple sources. The whole becomes the document that we see and use within our browser.

For example, a rendered document may include some styles from the browser's defaults for HTML. Then another part of the rendered style might come from customised browser settings or style definition files. For example, a user may customise their browser preferences or specify custom behaviours, which are then applied by the browser for rendering documents. Finally, we will also see styles applied from stylesheets linked to the document itself, or other embedded styles. It's not quite as simple as just looking at the linked CSS files.

So, to reiterate, for our styles in a cascade, a designer's stylesheets have priority, then user's stylesheets, and then the browser's own defaults.

Inheritance

CSS also includes an interpretation of the concept of inheritance for its styles. So, descendants will inherit properties from their ancestors.

For example, if we create a style on an element, all descendants of that element within the DOM will also inherit that style. This will apply unless the style is then overridden by another ruleset that specifically targets that element.

```
body {
  background: blue;
}

p {
  color: white;
}
```

As `p` is a descendant of `body` in the DOM, it will inherit the background colour of the body. So, we can set our paragraphs' text colour to correctly show against the specified background colour. White text colour against a blue background.

This characteristic of CSS is an important feature, and it helps to reduce redundancy and repetition of styles. It is also another reason why it is useful to maintain an outline of the DOM structure for a given HTML document.

Again, however, there is a small caveat to this characteristic of CSS. Whilst most styles will happily follow this pattern, not all properties are inherited by default. For example, properties related to block-level elements are a notable issue with inheritance. **Margin, padding, and border rules are not inherited from ancestors.**

Fonts

Fonts for our HTML document can be set for the `body` or within an element's specific ruleset. The first thing we need to do is specify our font-family,

```
body {
  font-family: "Times New Roman", Georgia, Serif;
}
```

The value for the font-family property specifies preferred and fall-back fonts for our document. If *Times New Roman* is not available, then the browser will try *Georgia* and *Serif*. We add quotation marks to "Times New Roman" because the font name includes spaces.

However, `""` is now added due to the CSS validator requesting this standard. It's believed to be a legacy error with the validator itself, and not an actual requirement of the CSS standard.

relative and `em`

We've already seen how we can change the colour of our text, but it's also useful to be able to modify the size of our fonts as well. For example,

```
body {
  font-size: 100%;
}
h3 {
  font-size: x-large;
}
p {
  font-size: larger;
}
p.p1 {
  font-size: 1.1em;
}
```

With the above, we begin by setting the base font size to 100% of the font size for a user's web browser. This allows us to scale our other fonts relative to this base size. So, we could use CSS *absolute* size values, such as `x-large`, which scales the size accordingly. Or, we could try *relative* sizes, such as `larger`, to help make our font sizes larger relative to the current context.

However, if we need better control of our font sizes, we can use `em`. These are meta-units, which represent a multiplier on the current font-size. They're derived from standard typography practices, which are based upon the standard width of an uppercase **M** in printing.

So, if the current font size has been set to `12px`, a font-size of `1.5em` will make the font actually an equivalent `18px`. However, this current font size will be relative to the element itself. So, cascading and inheritance will be an important factor in how a font-size is calculated. It might be as simple as referencing the root font size for the body, or an inheritance from a parent element. You'll need to check in the CSS and the rendering.

The obvious benefit of this approach, `em`, is that our text will scale according to the base font size. If we modify the size of the base, all font sizes set to `em` will also adjust accordingly.

Try different examples at

- W3 Schools

Further examples as follows,

- JSFiddle - CSS Fonts

relative and `rem`

Another option is the `rem` unit for font sizes.

This is simpler in concept, but can often provide an easier way to manage font sizes. It's particularly useful for websites and applications that are designed to be responsive or progressive in design and usage. e.g.

```
body {
  font-size: 100%;
}
p {
  font-size: 1.5rem;
}
```

However, there are issues with using both `em` and `rem` specified font-sizes.

custom web fonts

Using Fonts with CSS has often been a limiting experience, problematic at best, and reliant upon the installed fonts on a given user's local machine. There were workarounds, such as wrapping font files in JavaScript, and then serving from a remote server with the applicable HTML documents, but these were notoriously slow and buggy. They only tended to be employed, at best, as a final solution to a difficult problem.

However, with the advent of web fonts, the process of rendering with custom fonts is now considerably easier for designers and developers. We can deliver required fonts via the internet, using services such as Google's custom fonts.

- Google Fonts

From this site, we can pick and choose our custom fonts by selecting the *Quick-use* button. This loads a new page with options and instructions for using your chosen custom font. We then select our required character sets, add a `<link>` reference for the font to our HTML document, and then specify the fonts in our CSS. We need to add this new font as a font-family in our style sheet,

```
font-family: 'Roboto';
```

We can then style our document's fonts as normal.

An example may be found at the following URL,

- JSFiddle - CSS Custom Fonts

References

- CSS Tricks - nth child recipes
- JSFiddle - CSS Basics
- MDN - CSS
- Perishable Press - Barebones Web Templates
- W3 CSS
- W3 Schools - CSS
- W3 Schools - HTML Colour Picker
- W3 Web Style Sheets - Even & Odd