Inheritance and cascade

Introduction

Inheritance and the cascade are two fundamental concepts in CSS, that are important to understand. The two concepts are closely related, yet different:

- Inheritance is associated with how the elements in the HTML markup inherit properties from their parent (containing) elements and pass them on to their children.
- The cascade relates to CSS declarations being applied to a document, and how conflicting rules do or do not override each other.

This article provides an overview of both concepts.

Inheritance

Inheritance is the mechanism by which certain properties are passed on from a parent element down to its children, in the same fashion as genetics: if parents have blue eyes, their children will probably also have blue eyes.

Not all CSS properties are inherited, because it does not make sense for some of them to be. For instance, margins and width are not inherited, since it is unlikely that a child element requires the same margins as its parent. Imagine if you set the content block of a site to be 70% of the browser window width, and then all of its children adopted a width of 70% of their parent elements? Designing page layouts with CSS would be a nightmare.

In most cases, common sense indicates which properties are inherited and which are not: if you are not sure, look up all of the properties present in CSS2 in the CSS 2.1 specification property summary table. For CSS3 properties, consult the individual CSS3 module specifications, available in the CSS current work page.

Why inheritance is useful

CSS has an inheritance mechanism because otherwise CSS rules would be redundant. Without inheritance, it would be necessary to specify styles like font family, font size, and text color individually — for every single element type. The code would become bloated and repetitive.

Using inheritance, you can specify the font properties for the **`html`** or **`body`** elements and the styles will be inherited by all other elements. You can specify background and text colors for a specific container element and the text color will automatically be inherited by any child elements in that container. The background color is not inherited, but the initial value for **`background-color`** is **`transparent`**, which means a parent's background will shine through. The effect is similar to the page's appearance if background colors were inherited.

Note: Consider what would happen if background *images* were inherited. Every child would display the same background image as its parent. The result would look like a jigsaw puzzle put together by someone with a serious drug problem, since the background would be redrawn inside each subsequent child element.

How inheritance works

Every element in an HTML document inherits all inheritable properties from its parent *except* the root element (**`<html>`**), which does not have a parent.

Whether or not the inherited properties will have any effect depends on other things, as described later in the section about the cascade. Just as a blue-eyed mother can have a brown-eyed child if the father has brown eyes, inherited properties in CSS can be overridden.

An example of inheritance

1. Copy the following HTML document into a new file in your favorite text editor and save it as inherit.html.

   ```
   <!DOCTYPE html>
   <html lang="en">
     <head>
       <meta charset="UTF-8">
       <title>Inheritance</title>
     </head>
     <body>
      <h1>Heading</h1>
      <p>A paragraph of text.</p>
     </body>
   </html>
   ```

   If you open the document in your web browser, you will see a rather boring document displayed according to your browser's default styling.

2. Create a new empty file in your text editor, copy the following CSS rule into it, and save the file as style.css in the same location as the HTML file.

   ```
   html {
     font: 75% Verdana, sans-serif;
   }
   ```

3. Link the style sheet to your HTML document by inserting the following line before the **`</head>`** tag:

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

4. Save the modified HTML file and reload the document in your browser.

The font face changes from the browser's default (often set to Times or Times New Roman) to Verdana.

If you do not have Verdana installed on your computer, the text displays in the default sans-serif font specified in your browser settings. The text is also smaller; only three quarters of its size in the unstyled version. The CSS rule applies only to the **`<html>`** element. It does not specify any rules for headings or paragraphs, yet all of the text now displays in Verdana at 75% of its default size. Why? Because of inheritance. The **`font`** property is a shorthand property that sets a whole range of font-related properties.

The CSS rule only specified two properties — the font size and the font family — but that rule is equivalent to the following:

```
html {
  font-style: normal;
  font-variant: normal;
  font-weight: normal;
  font-size: 75%;
  line-height: normal;
  font-family: Verdana,sans-serif;
}
```

All of those properties are inherited, so the **`<body>`** element will inherit them from the **`<html>`** element and then pass them on to its children — the heading and the paragraph text. But notice that there is something unusual occurring with the inheritance of font size.

The **`<html>`** element's font size is set to 75%, but 75% of *what*? Surely the font size of the **`<body>`** should be 75% of its parent's font size, and the font sizes of the heading and the paragraph be 75% of that of the **`<body>`** element? The value inherited is not the specified value — the value typed in the style sheet — but something else called the computed value.

The computed value is, in the case of font size, an absolute value measured in pixels. For the **`<html>`** element, which doesn't have a parent element from which to inherit, a percentage value for font size relates to the default font size set in the browser. Most contemporary browsers have a default font size setting of 16px. 75% of 16 is 12, so the computed value for the font size of the **`<html>`** element is around 12px.

And *that* is the value that is inherited by **`<body>`** and passed on to the heading and the paragraph. (The font size of the heading is larger, because the browser applies some built-in styling of its own. For more details, see the section below about the cascade.)

Consider this example:

1. Add two more declarations to the rule in your CSS style sheet:

   ```
   html {
     font: 75% Verdana,sans-serif;

     /* Add these */
     background-color: blue;
     color: white;
   }
   ```

2. Save the CSS file and reload the document in your browser. Now the background of the whole document is bright blue, and all of the text is white. The white text color is inherited by the **`<body>`** element and passed on to all children of **body** — in this case the heading and the paragraph. The heading and paragraph do not, however, inherit the background but instead default to **`transparent`**, so the net visual result is white text displayed on a blue background.

3. Add another new rule to the style sheet:

   ```
   h1 {
     font-size: 300%;
   }
   ```

4. Save and reload the document: this rule sets the font size of the heading. The percentage is applied to the inherited font size — 12px, as we discussed above — so the heading size will be 300% of 12px, or 36px.

Forcing inheritance

You can force inheritance — even for properties that are not inherited by default — by using the **`inherit`** keyword. Use this strategy with care, however, since Microsoft Internet Explorer versions up to and including version 7 do not support this keyword.

The following rule will make all paragraphs inherit all background properties from their parents:

```
p {
  background: inherit;
}
```

Forcing inheritance is not a common practice. It can be useful for “undoing” a declaration in a conflicting rule, or to avoid hard coding certain values. As an example, consider a typical navigation menu:

```
<ul id="nav">
  <li><a href="/">Home</a></li>
  <li><a href="/news/">News</a></li>
  <li><a href="/products/">Products</a></li>
  <li><a href="/services/">Services</a></li>
  <li><a href="/about/">About Us</a></li>
</ul>
```

To display this list of links as a horizontal menu, you could use the following CSS rules:

```
#nav {
  background: blue;
  color: white;
  margin: 0;
  padding: 0;
}

#nav li {
  display: inline;
  margin: 0;
  padding: 0 0.5em;
  border-right: 1px solid;
}

#nav li a {
  color: inherit;
  text-decoration: none;
}
```

Here the background color of the whole list is set to blue in the rule for **`#nav`**. This also sets the foreground color to white, and this property is inherited by each list item and each link. The rule for the list items sets a right border, but doesn’t specify the border color, which means it will use the inherited foreground color, white. For the links, the **`color:inherit;`** forces inheritance and overrides the browser’s default link color.

Of course, you can specify the border color as white and the link text color as white, but the beauty of letting inheritance do the job is that it is only necessary to update one place to change the colors if you decide to update the color scheme later.

The cascade

CSS an acronym of Cascading Style Sheets, so it is not a surprise that the cascade is an important concept. It is the mechanism that controls the end result when multiple, conflicting CSS declarations apply to the same element. There are three main concepts that control the order in which CSS declarations are applied:

1. Importance
2. Specificity
3. Source order

These concepts are described below.

Importance is the most … er … important. If two declarations have the same importance, the specificity of the rules decide which one will apply. If the rules have the same specificity, then source order controls the outcome.

Importance

The importance of a CSS declaration depends on *where* it is specified. The conflicting declarations will be applied in the following order, with later declarations overriding earlier ones:

1. User agent style sheets

2. Normal declarations in user style sheets
3. Normal declarations in author style sheets
4. Important declarations in author style sheets
5. Important declarations in user style sheets

User agent style sheets

A user agent style sheet is the built-in style sheet of the browser. Every browser has its default rules for how to display various HTML elements if no style is specified by the user or designer of the page. For instance, unvisited links are usually blue and underlined.

A user style sheet is a style sheet that the *user* has specified. Not all browsers support user style sheets, but they can be very useful, especially for users with certain types of disabilities. For instance, a dyslexic person can have a user style sheet that specifies certain fonts and colors to help them read more easily.

An author style sheet is what we normally refer to when we say "style sheet". It is the style sheet that the author of the document (or, more likely, the website's designer) has written and linked (or included).

Normal and important declarations

Normal declarations are just that: normal declarations.

The opposite is important declarations, which are declarations followed by an **`!important`** directive. A dyslexic user might, for instance, want all text to be displayed in Comic Sans MS if he finds that font easier to read. He could then have a user style sheet containing the following rule:

```css
* {
  font-family: "Comic Sans MS" !important;
}
```

Important declarations in a user style sheet will trump everything else, which is logical. In this case, no matter what the designer specified, and no matter what the browser's default font family is set to, everything will be displayed in Comic Sans MS. The default browser rendering will only apply if those declarations aren't overridden by any rules in a user style sheet or an author style sheet, since the user agent style sheet has the lowest precedence.

To be honest, most designers don't have to think too much about importance, since there's nothing we can do about it. There is no way we could know if a user has a user style sheet defined that will override our CSS. If they do, they probably have a very good reason for doing so, anyway. Still, it's good to know what importance is and how it may affect the presentation of our documents.

Specificity

Specificity is something every CSS author needs to understand and to think about. It can be thought of as a measure of how specific a rule's selector is. A selector with low specificity may match many elements (like **`*`**, which matches every element in the document), while a selector with high specificity might only match a single element on a page (like **`#nav`**, which only matches the element with an **`id`** of **`nav`**).

The specificity of a selector can easily be calculated, as we shall see below. If two or more declarations conflict for a given element, and all the declarations have the same importance, then the one in the rule with the most specific selector will "win".

Specificity has four components; let's call them a, b, c and d. Component "a" is the most distinguishing, "d" the least.

- Component "a" is quite simple: it's 1 for a declaration in a **`style`** attribute (also known as inline styling), otherwise it's 0.
- Component "b" is the number of **`id`** selectors in the selector (those that begin with a **`#`**).
- Component "c" is the number of attribute selectors—including class selectors — and pseudo-classes.
- Component "d" is the number of element types and pseudo-elements in the selector.

After a bit of counting, we can thus string those four components together to get the specificity for any rule. CSS declarations in a **`style`** attribute don't have a selector, so their specificity is always 1,0,0,0.

Let's look at a few examples — after this it should be quite clear how this works.

Selector	a	b	c	d	Specificity
`h1`				1	0,0,0,1
`.foo`			1		0,0,1,0
`#bar`		1			0,1,0,0
`html>head+body ul#nav *.home a:link`		1	2	5	0,1,2,5

Let’s look at the last example in some more detail. a = 0 since it’s a selector, not a declaration in a **`style`** attribute. There is one ID selector (**`#nav`**), so b = 1. There is one attribute selector (**`.home`**) and one pseudo-class (**`:link`**), so c = 2. There are five element types (**`<html>`**, **`<head>`**, **`<body>`**, **`<ul>`** and **`<a>`**), so d = 5.

The final specificity is thus 0,1,2,5.

Note: Combinators (like **`>`**, **`+`** and white space) do not affect a selector’s specificity. The universal selector (**`*`**) has no input on specificity, either.

Note #2: There is a huge difference in specificity between an **`id`** selector and an attribute selector that happens to refer to an **`id`** attribute. Although they match the same element, they have very different specificities. The specificity of **`#nav`** is 0,1,0,0 while the specificity of **`[id="nav"]`** is only 0,0,1,0.

Let’s look at how this works in practice.

1. First of all, start by adding another paragraph to your HTML document.

```
<body>
  <h1>Heading</h1>
  <p>A paragraph of text.</p>

  <!-- Add this -->
  <p>A second paragraph of text.</p>
</body>
```

2. Next, add a rule to your stylesheet to make the paragraph text have a different color:

```
p {
  color: cyan;
}
```

3. Now save both files and reload the document in your browser; there should now be two paragraphs with cyan text.

4. Set an **`id`** on the first paragraph so you can target it easily with a CSS selector.

```
<body>
  <h1>Heading</h1>
  <!-- Add the id of "special" to this paragraph -->
  <p id="special">A paragraph of text.</p>
  <p>A second paragraph of text.</p>
</body>
```

5. Carry on by adding a rule for the special paragraph in your style sheet:

```
#special {
  background-color: red;
  color: yellow;
}
```

6. Finally, save both files and reload the document in your browser to see the now rather colorful result.

Let’s look at the declarations that apply to the two paragraphs.

The first rule you added sets a text color of cyan for *all* paragraphs. The second rule sets a red background color and a yellow text color for the single element that has the **`id`** of **`special`**. Your first paragraph matches both of those rules; it is a paragraph and it has an **`id`** of **`special`**.

The red background isn’t a problem, because it’s only specified for **`#special`**. Both rules contain a declaration of the **`color`** property, though, which means there is a conflict. Both rules have the same importance — they are normal declarations in the author style sheet — so you have to look at the specificity of the two selectors.

The selector of the first rule is **`<p>`**, which has a specificity of 0,0,0,1 according to the rules above since it contains a single element type. The selector of the second rule is **`#special`**, which has a specificity of 0,1,0,0 since it consists of an **`id`** selector. 0,1,0,0 is much more specific than 0,0,0,1 so the **`color: yellow;`** declaration wins and you get yellow text on a red background.

Source order

If two declarations affect the same element, have the same importance and the same specificity, the final distinguishing mark is the source order. The declaration that appears later in the style sheets will “win” over those that come before it.

If you have a single external style sheet, then the declarations at the end of the file will override those that occur earlier in the file if there’s a conflict. The conflicting declarations could also occur in different style sheets.

In that case, the order in which the style sheets are linked, included or imported controls what declaration will be applied, so if you have two stylesheets linked in a document **`<head>`**, the one linked to further down will override the one linked to higher up. Let’s look at a practical example of how this works.

1. Add a new rule to your style sheet at the very end of the file, like so:

   ```
   p {
     background-color: yellow;
     color: black;
   }
   ```

2. Save and reload the web page. you will now have *two* rules that match all paragraphs. They have the same importance and the same specificity (since the selector is the same), therefore the final mechanism for choosing which one wins will be the source order. The last rule specifies **`color:black`** and that will override **`color:cyan`** from the earlier rule.

Note how the first paragraph isn’t affected at all by this new rule.

Although the new rule appears last, its selector has lower specificity than the one for **`#special`**. This shows clearly how specificity trumps source order.