The UX Designer's CSS3 Cheat Sheet

You don't have to know *all* of CSS3 to be able to do useful things with it in your prototypes. Here are some of the more useful bits. (For a definitive and exhaustive reference, see MDN.)

The CSS rule

A CSS rule looks like this:

```
selector {
  property: value;
  ...
}
```

For example:

```
div {
  background-color: red;
  color: white;
}
```

This gives all `<div>`s a red background and white text.

Selectors

Selector	Example	What it means
`element`	`div`	Apply the rule to all matching elements (e.g., all `<div>`s).
`.class`	`.sidebar_text`	Apply the rule to all elements with this class.
`#id`	`#signup_form`	Apply the rule to the element with this ID.

Selector	Example	What it means
`[attribute="value"]`	`[type="radio"]`	Apply the rule to all elements with a `type` of `radio`. (Only`<input>`s have a `type` of `radio`, so you'd usually use them like this: `input[type="radio"]` (see below).) For attributes without a value, it can be used like this to select elements that have the attribute: `[disabled]`. There are also ways of matching just part of the value—see this CSS Tricks post.
`*`	`*`	Apply the rule to *all* elements.

Combinators

Selector combination	Example	What it means
`element.class`	`p.sidebar_text`	Apply the rule to the specified type of element with the specified class.
`element#id`	`form#signup_form`	Apply the rule to the specified type of element with the specified ID.
`selector1, selector2`	`section, div`	Apply the rule to all `<section>`s *and* all `<div>`s.
`selector1 selector2`	`section .legal_text`	Apply the rule to all elements with a class of `legal_text` within a `<section>`.
`selector1 > selector2`	`section > p`	Apply the rule to all `<p>`s that are direct children of a `<section>`. So it applies to this `<p>`: `<section> <p>Blah</p> </section>` but not to this one: `<section> <div> <p>Blah</p> </div> </section>`
`selector[attribute="value"]`	`input[type="tel"]`	Apply the rule to all `<input>`s with a `type` of `tel`. For attributes without a value, it can be used like this to select elements that have the attribute: `input[disabled]`.

Selector combination	Example	What it means
`selector1 ~ selector2`	`section ~ div`	Apply the rule to all elements that match the second selector that are siblings of the first selector. So `section ~ div` applies to all `<div>`s that are siblings of a `<section>`.
`selector1 + selector2`	`section + div`	Apply the rule to all elements that match the second selector that are *adjacent* siblings of the first selector. So `section + div` applies to all `<div>`s that are *adjacent* siblings of a `<section>` (i.e., come directly before or after a `<section>`).

Pseudo-classes

Pseudo-class	Examples	What it means
`selector:hover`	`a:hover`	Apply the rule to matching elements on hover (i.e., when the pointer is over them).
`selector:disabled`, `selector:enabled`	`input:disabled`	Apply the rule to matching elements that are disabled/enabled.
`selector:first-child`	`li:first-child`	Apply to the matching element that is the first child of its parent element. So `li:first-child` applies only to the first `<li>` in a `<ul>` or an `<ol>`.
`selector:last-child`	`li:last-child`	Apply to the matching element that is the last child of its parent element. So `li:last-child` applies only to the last `<li>` in a `<ul>` or an `<ol>`.

`selector:nth-child(arg)`	`li:nth-child(odd)`
	`li:nth-child(3)`
	`li:nth-child(4n)`
		Apply to the matching element that is the nth child of its parent element. `arg` can be a number, `odd` or `even`, or a formula.
		
		So `li:nth-child(odd)` applies to the odd `<li>`s in a list (i.e., the first, third, fifth, etc.). Handy for stuff like alternate row striping.
		
		`li:nth-child(3)` applies to the third `<li>` in a list.
		
		`li:nth-child(4n)` applies to every fourth `<li>` in a list. (You can make more complex formulas too.)

More pseudo-classes here.

Specificity

Sometimes you will have more than one rule that sets the same property for the same element. (If you're using a front-end framework, it will have lots of CSS rules, and you will probably need to override some of them.) But how does the browser decide which rule wins in this situation? And how do you make your rule the one that wins?

It works like this: **the more specific rule wins**.

But what does that *mean*? Well, specificity is calculated from a number of things: whether the `!important` keyword is used, inline styles (applied directly to an element using the `style` attribute instead of using a CSS rule), and which of the following the selector includes: an ID, class names (more class names makes it more specific), and/or element names.

This is how that works:

`!important` > inline style > ID > class > element

Let's dig into that a bit more.

!important

`!important` is the most powerful. It *always* beats all the others.

So this:

```
div.callout { color: red !important; }
```

beats this:

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

So why not use `!important` whenever we find ourselves in this situation? Well, OK, what happens when you end up with two rules that both use `!important`? You're still going to have to use the other things (ID, class, element) to make one rule more specific so that it overrides the other. It's OK to use `!important`, but use it sparingly.

Inline styles

Inline styles are where you add a `style` attribute directly to an HTML element, like this:

```
<p style="color:red;">Hello</p>
```

This will always beat any CSS rule that tries to set the color for this element (except one that uses `!important`).

I try to avoid using inline styles (I prefer to keep things separate), but they are handy for trying things out without having to switch to a separate CSS file.

IDs and classes

A rule that includes an ID beats one that uses only classes. Let’s say we have this in our HTML:

```
<div id="container">
   <p class="intro">Hello</p>
</div>
```

This rule:

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

will beat this one:

```
.intro {color: black; }
```

This despite the fact that the first, ID-based rule does not directly target the <p> element.

Element names

A rule with just an element name as the selector will be beaten by one with a class or an ID. So for the same HTML as above, this rule:

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

will beat this one:

```
p { color: black; }
```

Combining selectors

Combining selectors to make them more specific increases a rule’s specificity. So this rule:

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

beats both this one:

```
p { color: black; }
```

and this one:

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

But the above rules still apply—a rule that uses an ID will beat all of these.

Rule of thumb

Make rules as specific as you can, and remember that you can go and add IDs and classes to your HTML if you need to.

For more on specificity, see this CSS Tricks article.

Properties

The following is a list of the properties that I use all the time while prototyping. There are many, many more, but I just don't find myself using them. Some, like `float` and `clear` and media queries (`@media`) are important, but if you use a framework such as Foundation or Bootstrap, you won't really need them—the framework handles all that fiddly layout stuff for you.

Property	Value(s)	What it means	Example
`font-family`	Names of one or more fonts	The font(s) for the selected elements. Multiple font names (a "font stack") give you fallbacks if a font isn't available (e.g., Helvetica on Windows machines).	`font-family: "Helvetica Neue", arial, sans-serif;`
`font-size`	Numeric value plus unit (`px`, `%`, `em`, `rem`, etc.) or keyword (not used much now)	The size of the text for the selected element. `px` is absolute size in pixels. `%` and `em` are relative to the parent element, `rem` is relative to the root (`html`) element.	`font-size: 1.6rem;`
`font-weight`	A keyword or a numeric value	The weight of the text for the selected element. (Most fonts have `bold` and `normal` variants; others have multiple numeric weights: 100, 200, 300, etc. `bold` corresponds to 700, `normal` to 400.)	`font-weight: bold;`
`font-style`	`normal` or `italic`	Makes the text of the selected element italic or not.	`font-style: italic;`
`text-decoration`	`underline`, `overline`, none, etc., plus color and/or style if desired.	Adds or removes underline/overline to the text of the selected element. This is a shorthand rule for: `text-decoration-line`, `text-decoration-color`, and `text-decoration-style`.	`text-decoration: underline;`
`text-align`	`left`, `right`, `center`, `justify`, etc.	The text alignment for the selected element.	`text-align: center;`
`line-height`	Numeric value (usually)	The line spacing of the selected element's text. 1.6 is a good value (browsers usually default to 1.2).	`line-height: 1.6;`

Property	**Value(s)**	**What it means**	**Example**
`margin`	One, two, or four numeric values plus unit (`px`, `%`, `em`, `rem`, etc.). Negative numbers are allowed.	Adds or modifies whitespace outside the bounding box of the selected element. A single value sets the margin on all four sides to the same size. Two values set the top and bottom margin to the first value and the left and right margin to the second. Four values set the top, right, bottom, and left margins, in that order. You can also set them individually using `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`.	`margin: 20px 10px 0px 10px;`
`padding`	One, two, or four numeric values plus unit (`px`, `%`, `em`, `rem`, etc.).	Adds or modifies whitespace *inside* the bounding box of the selected element. The syntax is the same as for `margin`, above. You can also set them individually using `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`.	`padding: 10px;`
`border`	Numeric value plus unit for line thickness, plus style and/or color if desired.	Adds a border around the selected element. This is shorthand for: `border-width`, `border-style`, and `border-color`. You can also set these for each side separately using `border-top`, `border-bottom`, etc.	`border: 1px solid silver;`
`border-radius`	Numeric value plus unit (px, %, em, rem, etc.)	Adds rounded corners to the border and/or background of the selected element. The example adds a small amount of rounding to all four corners. (It is also possible to add elliptical rounded corners, different rounding on each corner, etc.) You can use this to turn a square into a circle by making the radius big enough.	`border-radius: 3px;`

Property	Value(s)	What it means	Example
`color`	Keyword, hex value (e.g., `#FF0000`), RGB (`rgb(255,0,0)`), or RGBa (`rgba(255,0,0,0.5)`)	The color of the selected element's text (including underline, etc.). The example makes the text red and 50% opaque.	`color: rgba(255,0,0,0.5);`
`background-color`	Keyword, hex value (e.g., `#FF0000`), RGB (`rgb(255,0,0)`), or RGBa (`rgba(255,0,0,0.5)`)	The background color of the selected element's bounding box. (Note that padding gets the background color, but margins don't.) The example sets the background color to pale grey.	`background-color: #EEEEEE;`
`opacity`	A numerical value between 0 and 1	The opacity of the selected element (including all of its child elements). Note that it is also possible to set the opacity of just the element's text, just its background, just its border, and so on (by using RGBa values for `color`, `background`, and `border` respectively).	`opacity: 0.5;`
`z-index`	Any whole number (including negative numbers). Default value: 0.	How the element is layered if it overlaps other elements. Normally, if two or more elements overlap, the source order determines which one appears on top. `z-index` lets you explicitly control this—the element with the bigger `z-index` wins. (`z-index` has no effect on elements with `position: static;`.)	`z-index: 10;`
`visibility`	`visible` or `hidden`	Hides or shows the selected element *without affecting layout*. Setting `visibility` to `hidden` does not cause the following elements to move up to where the element was.	`visibility: hidden;`

Property	Value(s)	What it means	Example
`display`	none, `inline`, `block`, `inline-block`, etc.	How to display the selected element: • none—hides the element. Unlike `visibility: hidden;` this *does* affect layout—it's as though the element doesn't exist. • `inline`—the element is displayed on the same line as sibling elements. (You can't specify width, height, margins, etc. for inline elements.) • `block`—the element is placed on a new line, below the previous element. • `inline-block`—the element is placed inline, but otherwise behaves like a block element (you can give it `height`, `width`, `margin`, etc.).	`display: inline-block;`
`position`	`static`, `relative`, `absolute`, or `fixed`	How the element is positioned: • `static` (the default)—the element is positioned normally. You can't use `top`, `bottom`, `left`, `right`, or `z-index`. • `relative`—the element is positioned relative to where it normally would be. So you can use `top`, `bottom`, `left`, and `right` to change its position. • `absolute`—the element is positioned relative to its parent (or the first ancestor element with a `position` other than `static`). • `fixed`—the element is positioned relative to the viewport and does not move when you scroll. (Did not work in older mobile browsers.)	`position: relative;`

Property	Value(s)	What it means	Example
`top`, `bottom`, `left`, `right`	Numeric value plus unit (px, %, em, rem, etc.)	Where the element is positioned, relative to where it would normally be. Each one moves the corresponding edge of the element left or right (for `left` and `right`) or up or down (for `top` and `bottom`). Handy trick: when `position` is set to `absolute` and the parent element's `position` is set to `relative`, you can use these to stretch the element into position (e.g., so it is 100% of the parent's width and height, regardless of the size of its content). Except for this special case, if you use `top` and `bottom` together, `top` wins. And if you use `left` and `right` together, `left` wins (except on RTL pages).	`top: 10px;`
`width`	Numeric value plus unit (px, %, em, rem, etc.)	The width of the selected element. For an image, setting just `width` (and not `height`) scales it proportionately. For elements containing text, setting just `width` (and not `height`) causes the element's height to increase or decrease to accommodate its content.	`width: 100px;`
`height`	Numeric value plus unit (px, %, em, rem, etc.)	The height of the selected element. For an image, setting just `height` (and not `width`) scales it proportionately. For elements containing text, setting just `height` (and not `width`) has no effect on its width (by default, this will be 100% of the width of the parent). Instead, it will either add whitespace at the bottom of the element (if the height is greater than what it would be normally) or cause the content to overflow the element's bounding box (if the height is less than what it would be normally). See `overflow` for ways to handle this.)	`height: 200px;`

`min-width`, `max-width`	Numeric value plus unit (px, %, em, rem, etc.)	Maximum or minimum width for the selected element.	`min-width: 100px;`
			`max-width: 600px;`

`min-height`, `max-height`	Numeric value plus unit (px, %, em, rem, etc.)	Maximum or minimum height for the selected element.	`min-height: 200px;`
			`max-height: 400px;`

Property	Value(s)	What it means	Example
`overflow`, `overflow-x`, `overflow-y`	`visible`, `hidden`, `scroll`, `auto`	Defines what happens when the content is too big for its bounding box (for example, if you set the height of the element to less than what it would normally be). The `-x` and `-y` variants define behavior for just the x or y axis. • `visible` (the default)—if the content is too big, it flows out onto whatever is there. • `hidden`—the extra content is hidden. • `scroll`—the extra content is hidden, but scrollbars are added (on desktop) so that all the content can be accessed. The scrollbars are displayed all the time. (On mobile, the content area can still be scrolled.) • `auto`—if the content is too big, the extra content is hidden and scrollbars are displayed. If the content is *not* too big, scrollbars are *not* displayed.	`overflow: scroll;`