Chapter 1

Advanced Selectors

Web designers commonly get into the habit of using only a very small collection of basic CSS selectors, most notably the `id`, `class`, and `descendant` selectors, because that's all they need…or so they think.

It's true that you can accomplish almost everything with the `class` selector, but why compromise your nice, clean markup by adding unnecessary classes when you can choose from a range of more practical and efficient alternatives?

CSS3 has brought with it a whole host of such alternatives, some of which are simply wonderfully convenient and others that can make you excitedly ponder all the possibilities! What? CSS3 selectors *are* exciting!

In this chapter, you will learn about the different types of selectors, from sibling combinators to nth-child expressions. With the help of practical examples, you'll be able to understand precisely what they do and how you can put them to use. I'll finish the chapter by combining many of the various selector types that will be described between now and then to create some truly advanced CSS selectors.

Child and Sibling Selectors

The child and sibling combinators are actually among the more mature features in the Selectors module; however, despite this they've had trouble in finding the same kind of mainstream attention as `id` and `class` selectors enjoy. It's about time they got a little nudge into the spotlight.

Child Combinator

Generally speaking, the cascade aspect of CSS is awesome, but sometimes you just don't need or want it. Have you ever found yourself undoing your own styles in a nested unordered list, for example? This is where the child combinator can help out. Consider this example:

```
nav ul {
    background: blue;
    border: 2px solid red;
}

nav ul ul {
    border: none;
    background: none;
}
```

If you find that you have to re-declare your styles to fight against the cascade, there is almost definitely a better way of going about the task. As shown in the following snippet, the child combinator allows you to select only the `ul` that is a direct child of the `nav` element so that the styles aren't applied to any nested `ul` elements:

```
nav > ul {
    background: blue;
    border: 2px solid red;
}
```

Simple stuff, right? Even better, this capability is supported in all major browsers, including Internet Explorer 7!

A few of the selectors discussed in this chapter, including the child combinator, have been around for a while and were actually first defined in the CSS 2.1 selector specification. However, I still see a staggering number of experienced web designers who remain oblivious to their existence, opting to stick with their trusty class and descendant selectors.

Adjacent Sibling Combinator

The adjacent sibling combinator targets the sibling that immediately follows an element. This selector has a number of uses. One common use is when you need to target, for example, the first `p` element to follow an `h1` or an `h2` in your body text, as follows:

```
p {
    margin-top: 1.5em;
}

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

Nice and easy, and a practical solution to targeting elements that could otherwise be really awkward to style, particularly if some of your markup is generated by a content management system.

Major browser support for the adjacent sibling combinator is universal, again including IE7+!

General Sibling Combinator

The general sibling combinator selector is similar to the adjacent sibling combinator, but more [wait for it]… general! Whereas the previous selector targets only the sibling that *immediately* follows an element, this selector targets *any* sibling that follows an element.

Consider the following example:

HTML

```
<h2 class="important">Everything below is very important</h2>
<p>This text is very important.</p>
<p>This text is also very important.</p>
```

```
CSS
h2.important ~ p {
    color: red;
}
```

This selector targets all `p` elements that follow an `h2` with a class of `important`, allowing you to apply a text color of `red` to all paragraphs under that particular heading.

If your first thought regarding this selector is anything like mine was, you probably are thinking "Hmm, pretty cool, but to be honest I can see this sitting at the bottom of the virtual toolbox gathering layers of virtual dust."

The reality, though, is *quite* the contrary! In fact, the general sibling combinator will prove to be one of the most vital ingredients in some of the most complex solutions you will read about in this book. Keep an eye out for its return towards the end of this chapter.

The general sibling combinator is much newer than those discussed earlier. Although it is compatible with all major browsers, Internet Explorer supports it only from version 9.

Attribute Selectors

Another method of selecting elements is through HTML attributes, in terms of whether an element has a certain attribute applied to it and/or the value of the attribute.

All the following attribute selectors have full major browser support, including IE7+.

Selecting Based on the Existence of an HTML Attribute

You can use a selector to target only those elements that have a specific attribute applied to them. The following example shows the use of a selector to target only those `a` elements that have a `title` attribute applied:

```
HTML
<a title="Contact Us" href="contact.php">Contact Us</a>
<a href="mailto:info@domain.com">Email Us</a>

CSS
a[title] {
    text-decoration: underline;
}
```

The selector would apply an `underline` only to the `Contact Us` link in this example because the `Email Us` link doesn't have a `title` attribute.

Selecting Based on the Exact Value of an HTML Attribute

More often than not, you would want to select an element based on the attribute's *value* rather than simply the fact that it exists. The most basic way of achieving this is to select based on the exact value of the attribute, as shown in the following example:

HTML

```
<input type="text">
<input type="submit">
```

CSS

```
input[type="submit"] {
    background: blue;
    color: white;
}
```

This selector targets any `input` element with a `type` attribute value that matches an exact string. In this case, the string is `submit`, allowing you to apply styles to your Submit button freely, without affecting your text input. Very useful.

Selecting Based on the Partial Contents of an HTML Attribute Value

Sometimes you might not know the exact attribute value you need to match, so you want a bit more leniency in terms of matching your HTML attributes. Fortunately, you can use a straightforward solution that allows you to select based on the HTML attribute *containing* a particular string, as shown in the following example:

HTML

```
<a href="http://twitter.com">Twitter</a>
```

CSS

```
a[href*="twitter"] {
    padding-left: 30px;
    background: url(twitter-icon.jpg) left no-repeat;
}
```

The simple addition of an asterisk means that the `href` attribute value in this example has to contain only the string `twitter` to match and apply the styling to the element. In this case, a `background-image` of the Twitter icon is added to any links that go to a Twitter URL.

Selecting Based on the Beginning of an HTML Attribute Value

You can also select based on what an HTML attribute value begins with. For example, if you want to add an icon to all external links, you select all elements with an `href` value that starts with `http`, as shown in the following code:

HTML

```
<a href="http://google.com">Google</a>
```

CSS

```
a[href^="http"] {
    padding-left: 30px;
    background: url(external-icon.jpg) left no-repeat;
}
```

As previously described, the selector targets all the `a` elements with an `href` value starting with `http`, simply by changing the asterisk to a caret (^) symbol.

Selecting Based on the End of an HTML Attribute Value

Now, perhaps you want to show an icon, such as a country's flag, depending on the domain suffix in an external link. You could select all links that direct to a New Zealand domain name, for example, by selecting all the `a` elements that have an `href` value that ends in `.nz`, as shown in this example:

HTML
```
<a href="http://google.co.nz">Google New Zealand</a>
```

CSS
```
a[href$=".nz"] {
    padding-left: 30px;
    background: url(nz-flag.jpg) left no-repeat;
}
```

By changing the caret symbol to a dollar sign, you can now match the string with the *end* of the `href` attribute, allowing you to apply a New Zealand flag to all links that go to a New Zealand domain name.

Selecting Based on Space-Separated HTML Attribute Values

Some HTML attributes allow you to specify multiple, space-separated values; one such example you likely used frequently in the past is, of course, the `class` attribute, which allows multiple `class` values.

With the help of the tilde character (~), you can select an element with an attribute containing multiple values, based on one of those values, as shown in the following example:

HTML
```
<img data-group="Gallery Dog" src="terrier.jpg" alt="Scottish Terrier">
<img data-group="Gallery Dog" src="spaniel.jpg" alt="Springer Spaniel">
<img data-group="Gallery Cat" src="white-cat.jpg" alt="White Cat">
<img data-group="Gallery Cat" src="black-cat.jpg" alt="Black Cat">
```

CSS
```
img[data-group~="Dog"] {
    border: 3px solid brown;
}
```

In this example, the HTML5 `data-` attribute enables you to store some information on each of the images in terms of the image's purpose (in this case, a gallery image) and to which gallery the image belongs (cat or dog).

The selector used here simply looks to see whether one of the values of the `data-group` attribute is `Dog` and then applies the appropriate styling.

Pseudo-Classes

Pseudo-classes have been around for a while in the form of `:hover` and `:visited`, among others, but now CSS3 offers a sizeable selection of brand new pseudo-classes to make the lives of web designers just that much easier.

Firsts and Lasts

First of all (excuse the pun) I touch on the ways of selecting the first or last element in a parent container—without using any classes, of course.

Picture this scenario: you've just coded a nice vertical navigation and given it a `border`. You've also applied a `border-bottom` to each of the links to act as a divider. Of course, you now have the classic issue of a double border at the end of the navigation, as illustrated in Figure 1-1 (we've all been there). You can also see the issue on the companion website, `www.wiley.com/go/ptl/css3`, in the demo file `0101-double-border.html`.

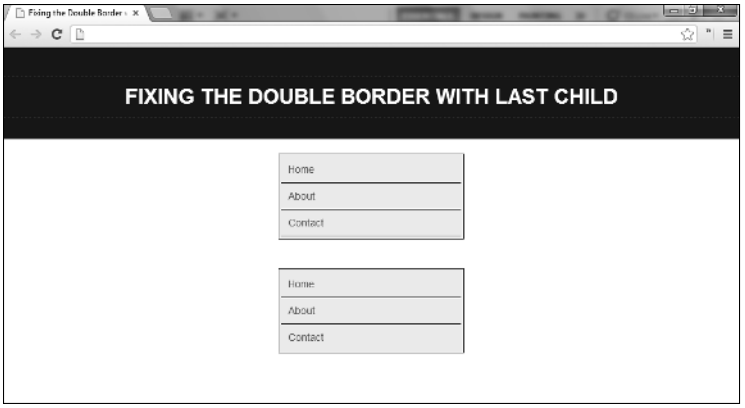

Figure 1-1 The dreaded double border (top), and corrected (bottom)

You could add a class to the last navigation item and use that to remove the `border-bottom` styling, but that approach feels dirty and wrong. Why? Because it's inefficient, adds unnecessary bytes to the markup, and would prove to be a pain when the code requires maintenance.

Enter the `:last-child` pseudo-class:

HTML

```
<ul>
    <li><a href="index.php">Home</a></li>
    <li><a href="about.php">About</a></li>
    <li><a href="contact.php">Contact</a></li>
</ul>
```

CSS

```
ul {
    border: 1px solid #000;
}

ul li {
    border-bottom: 1px solid #000;
}

ul li:last-child {
    border-bottom: none;
}
```

In this example, the pseudo-class looks for the last child in the `ul`, which is, of course, the last navigation list item, and it removes the `border-bottom` styling.

There is little need to go into any more depth with the various cousins of the `:last-child` pseudo-class because they all function in much the same way, so I just touch on them briefly. Note that all these selectors have universal support in major browsers, including IE9+, but not in earlier versions of IE unless otherwise stated.

- `:first-child` selects the first child of an element, regardless of the element type. This particular pseudo-class was first defined in CSS 2.1 and has universal major browser support, including IE7!
- `:first-of-type` looks for the first of a particular type of element in a container. For example, `p:first-of-type` would select the first `p` element, regardless of whether or not it is the first child in the parent.
- `:last-of-type`, as you might expect, does exactly the same as the preceding selector, except it looks for the *last* of a particular type of element in a container.
- `:only-child` selects an element if it is the sole child in its container. For example, in the markup for a standard navigation, each `a` element would generally be the only child of the `li` element that it inhabits.
- `:only-of-type` does the same as the preceding selector, except it selects elements if they are the only one of their type in the container. For example, if a `div` contains only an `h1` and a `p`, each of these children would be the only one of its type.

Nth Child Selectors

The nth child selectors are great because sometimes you need a bit more flexibility than just first or last; sometimes you want to select the fifth child or the eighth child…or the 279th child. When you use `:nth-child()`, it's a cinch.

The most basic way to use `:nth-child()` is to explicitly state which child you want to select, as demonstrated in the following example:

HTML

```
<ul>
    <li>List Item One</li>
    <li>List Item Two</li>
    <li>List Item Three</li>
    <li>List Item Four</li>
</ul>
```

CSS

```
ul li:nth-child(3) {
    color: red;
}
```

In this example, the third list item text would be red. All very simple.

But what if you have a really long list with an unpredictable number of list items and you want to select the fifth-from-last item? Fortunately, another pseudo-class does just that job.

The `:nth-last-child()` selector works in exactly the same way as `:nth-child()`, except that it counts from the last child rather than the first. So `li:nth-last-child(5)` would select the fifth-from-last list item.

Furthermore, you can swap out the `-child` aspect of these selectors for `-of-type`, much like the examples discussed earlier, when you require a more specific match.

For example, consider the following code:

HTML

```
<article>
    <h1>This is a Top Level Heading</h1>
    <p>This is the first paragraph.</p>
    <p>This is the second paragraph.</p>
    <p>This is the third paragraph.</p>
    <p>This is the fourth paragraph.</p>
</article>
```

CSS

```
article p:nth-of-type(3) {
    color: red;
}
```

This selector would target the third paragraph, whereas if you use the `p:nth-child(3)` selector, it would target the third child of the container regardless of type, which would be the second paragraph.

Again, you can use `:nth-last-of-type()` to start the count from the end of the parent container, so if you use the preceding example, `:nth-last-of-type(2)` would select the third paragraph.

Taking Nth Child to the Next Level with Expressions

As well as a simple number, the nth child pseudo-class also accepts expressions for times when you require much more complex selecting patterns. These expressions take the form of `an+b;` for example, `2n+1`, which selects every odd child.

So how does this expression work? Well, think back to those exciting algebra lessons you remember with such fondness and consider the expression as a simple algebraic equation, where *n* increments by 1, starting from 0.

(2×*n*) + 1

(2×0) + 1 = 1 (selects the first child)

(2×1) + 1 = 3 (selects the third child)

(2×2) + 1 = 5 (selects the fifth child)

(2×3) + 1 = 7 (selects the seventh child)

[Etc.]

With the expression broken down, you can clearly see how it selects every odd child. However, I find using a different method of breaking things down makes it a little easier to immediately understand an expression at a glance.

Consider the example `5n+3`. Here, the first number (5) states that every fifth child should be targeted. The second number (3) states the location where the sequence should start. So in this case, every fifth child would be selected, starting from the third child. This example would select the 3rd, 8th, 13th, and 18th child (and so on).

Using Keywords with Nth Child

In addition to integers and expressions, the nth child pseudo-class also accepts a choice of two shorthand keywords, which are, quite predictably, `odd` and `even`.

The `odd` keyword is shorthand for `2n+1`, as shown in the earlier example, and the `even` keyword can also be written as `2n+2` or `2n+0`—or simply, `2n`. You might have noticed that `2n` doesn't exactly need a shorthand option, but the `even` keyword does have the benefit of being immediately obvious to those who aren't familiar with expressions.

Using Negative Numbers with Nth Child

The nth child pseudo-class doesn't just accept positive integers; you can use negative numbers too. Your mind may be searching for a situation in which you might need to do so, but you can actually use negative numbers in some very clever ways.

Figure 1-2 shows a basic league table with 10 teams. The top spots—1 and 2—are promotion places, 3–6 are play-off places, and 9–10 are relegation spots. You need to target each of these zones to style them appropriately, so how do you do it?

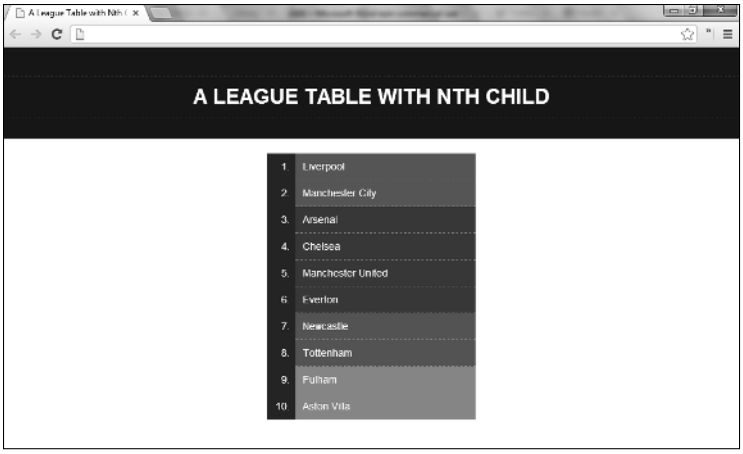

Figure 1-2 A league table showing the different zones

You could just use `:nth-child()` with basic integers to target each position, as demonstrated in the following code. Note that generally you would code a league table using a `table`, but for the sake of simplicity, an ordered list suffices for this example.

HTML

```
<ol>
    <li>Liverpool</li>
    <li>Manchester City</li>
    <li>Arsenal</li>
    <li>Chelsea</li>
    <li>Manchester United</li>
    <li>Everton</li>
    <li>Newcastle</li>
    <li>Tottenham</li>
    <li>Fulham</li>
    <li>Aston Villa</li>
</ol>
```

CSS

```
ol li:first-child,
ol li:nth-child(2) {
    background: green;
}

ol li:nth-child(3),
ol li:nth-child(4),
ol li:nth-child(5),
ol li:nth-child(6) {
    background: blue;
}

ol li:last-child,
ol li:nth-last-child(2) {
    background: red;
}
```

This code would do the job just fine, but that's a lot of selectors—eight in total. With the help of negative numbers, you can reduce that total to just three—one per rule-set.

To select the top two spots, you can use the expression `-1n+2`. Remember, the selection process starts at the second number; in this case, the second child. The first number (-1) states that every child (starting from the second child) should be selected, but in the reverse direction, therefore selecting the second item and then the first item. You can see how this equation works in the following example:

(-1×n) + 2 (you can shorten this to just `-n+2`)

(-1×0) + 2 = 2 (second child is selected)

(-1×1) + 2 = 1 (first child is selected)

(-1×2) + 2 = 0 (No selection)

To select the play-off positions (items 3–6), you need to use a bit of imagination as the desired range is isolated in the middle of the list. You can select the first six places using the method described previously, but in order to be more specific you can actually attach a *second* `:nth-child()` expression onto the selector! The required expression in this second pseudo-class is `n+3`, which ensures the selection begins at the third item, and the original `–n+6` expression ensures the selection *ends* at the sixth item. Take a closer look in the following code example.

```
ol li:nth-child(-n+2) {
    background: green;
}

ol li:nth-child(-n+6):nth-child(n+3) {
    background: blue;
}

ol li:nth-last-child(-n+2) {
    background: red;
}
```

Much nicer. You can see the finished example, `0102-league-table.html`, on the companion website.

The Best of the Rest

With the subject of nth child selectors all but exhausted, there are a few more pseudo-classes to discuss that can prove extremely useful, most of which haven't found their way into the mainstream just yet.

All the following selectors have universal support in major browsers, including IE9+:

- `:empty` selects any elements that have absolutely zero content or children.

  HTML

  ```
  <div></div> <!-- This div is empty -->
  <div> </div> <!-- This div is not empty -->
  ```

  CSS

  ```
  div:empty {
      display: none;
  }
  ```

- `:not(x)` is referred to as the negation pseudo-class. You can use it to select all elements except for `x`, where `x` is another selector.

  ```
  p:not(.important) {
      color: black;
  }
  ```

- `:target` selects an element that is the target of a fragment identifier in the document's URI. For example, if `#bio` sits at the end of the URI in your address bar, you can use `:target` to select the element in your HTML with an `id` of `#bio`.

HTML
```
<div id="bio">This is a bio</div>
```

CSS
```
div {
    background: white;
}

div:target {
    background: yellow;
}
```

Very useful and very convenient. In the case of `:target`, though, you gain a bit more than just convenience. I often use `id` attributes as anchors, allowing me to direct a link to another area on the same page, which adds the `id` anchor to the end of the URI, meaning it can then be styled using `:target`.

Have you clocked what that means? That's right—with `:target`, you can alter CSS styling for a particular element on-click! Using the preceding example, if you have a link with an `href` value of `#bio`, when a user clicks this link, the background of the `#bio div` changes from white to yellow! Huge potential there, I'm sure you will agree.

The last few pseudo-classes to address all relate to form elements, allowing you to style these elements based on their current state.

Again, all the following are supported in all major browsers, including IE9+:

- `:disabled` selects elements that can't be focused or selected in any way—for example, text inputs that use the `disabled` HTML5 attribute.
- `:enabled` selects elements that are in their default state and are ready to be selected. Basically, this includes anything that cannot be selected using `:disabled`.
- `:checked` targets radio buttons or check boxes that have been selected by the user.

Again, the last of these selectors stands out as a potential star performer in the brave new world of CSS3 experimentation. With a bit of help from the other advanced selectors that have been discussed, the `:checked` pseudo-class can be a really powerful tool that you can use to trigger CSS alterations on-click.

Also, `:checked` doesn't suffer from the drawbacks that can make `:target` awkward to work with in terms of triggering CSS changes on-click, such as ugly `id`'s and page jumps.

Bringing It All Together

The time has come to combine some of these advanced selectors and put them to use to create a simple, interactive multiplication table (see Figure 1-3).

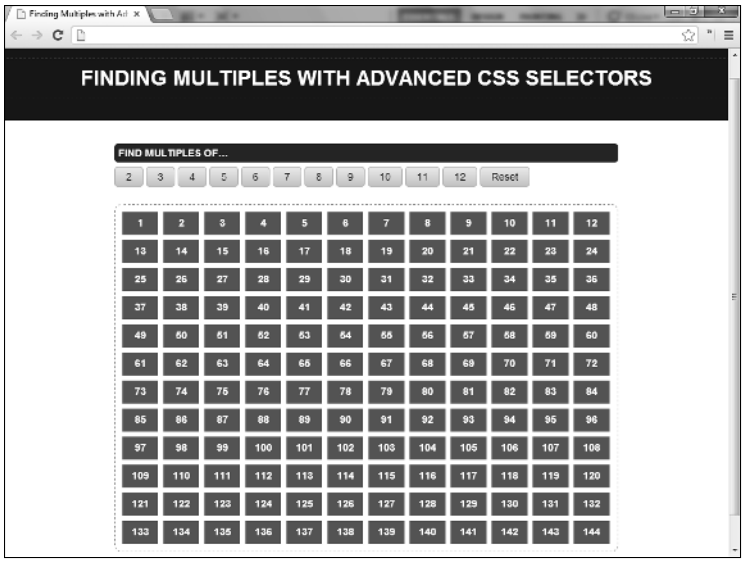

Figure 1-3 Interactive multiplication tables using only CSS

I explain how to develop this solution using only the code necessary to make it function correctly, so to see the full working demo with all the bells and whistles, open `0103-multiples-grid.html` on the companion website at `www.wiley.com/go/ptl/css3`.

The grid is simply a long list of numbers and so has been marked up using an ordered list. The button options are actually `label` elements, styled to look like clickable buttons. So how do they function as clickable buttons? Well, the `label` elements have been grouped with `input` elements (`radio` buttons to be precise) using the `for` attribute, allowing the `label` to be clickable on behalf of the hidden `radio` button.

```
<input value="2" id="two" name="multiples" type="radio">
<label for="two">2</label>

<input value="3" id="three" name="multiples" type="radio">
<label for="three">3</label>

<input value="4" id="four" name="multiples" type="radio">
<label for="four">4</label>
<!-- etc. -->

<ol>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <!-- etc. -->
</ol>
```

Now that the markup structure is in place, the buttons need to trigger the appropriate action when clicked. For example, when a user clicks the 5 button, the styling should change to highlight all numbers that are a multiple of 5.

The first step is to select the appropriate radio button:

```
input[value="5"]
```

You use an attribute selector here, so only an `input` element that has an exact `value` attribute of `5` is selected. Next, you need to modify this selector so that it targets the element only after it has been clicked:

```
input[value="5"]:checked
```

As you read earlier, radio buttons and check boxes allow use of the `:checked` pseudo-class when a user selects them, subsequently making the buttons behave like event triggers. However, now you need to somehow select the *list items* when the button is checked, which, with a glance at the markup, looks as though it could be troublesome based on the general laws of CSS.

Fear not! Remember the general sibling combinator (~)? This simple tilde character is all that you need here because, if you need reminding, it selects any siblings that follow a particular element in the parent container.

```
input[value="5"]:checked ~ ol li
```

The `ol` comes after all the `input` elements in the markup and they share the same parent, so the general sibling combinator works great in this example. As it stands, the preceding selector targets all the `li` elements when a user clicks the 5 button.

Now, you need to make it target only the numbers that are multiples of 5. You may have guessed that this is where the handy nth child pseudo-class makes a return!

```
input[value="5"]:checked ~ ol li:nth-child(5n)
```

The `5n` equates to "select every fifth child"—and that's it! With that selector, every multiple of 5 in the list of numbers is selected when a user clicks the 5 button, allowing you to highlight these numbers in whatever way you want. This is what I've done in the demo.

```
input:checked ~ ol li {
    opacity: 0.3;
    transform: scale(0.7);
}

input[value="2"]:checked ~ ol li:nth-child(2n),
input[value="3"]:checked ~ ol li:nth-child(3n) /* etc. */ {
    background: #333;
    opacity: 1;
    transform: scale(1);
}
```

Bear in mind that in the printed code examples throughout the book, I'll be using only the un-prefixed version of properties for brevity. The live demos, however, are optimized for Chrome unless otherwise stated, as I mentioned in the introduction.

The preceding code employs some CSS3 properties to shrink and obscure all the numbers when one of the buttons has been clicked *except* for the numbers you want to highlight, as shown in Figure 1-4. The `opacity` and `transform` properties are explained in more detail in Chapters 4 and 5, respectively.

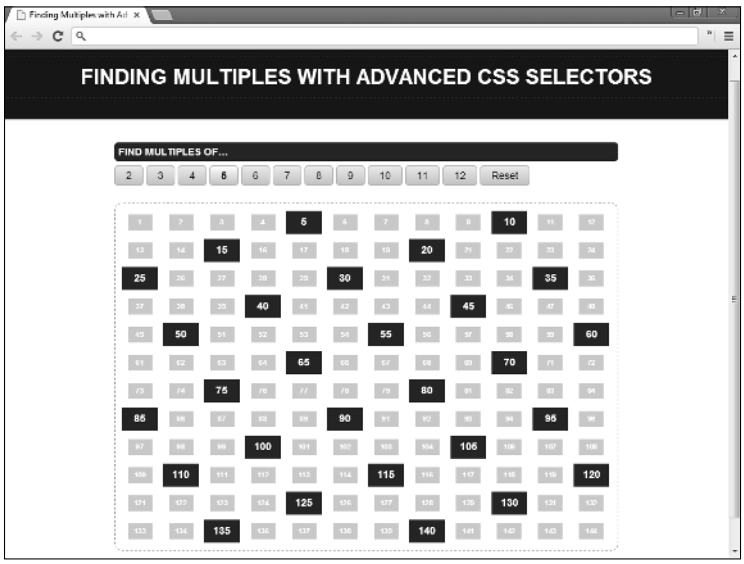

Figure 1-4 The numbers grid after the number 5 button has been clicked

Summary

The best thing about all the selectors discussed in this chapter is that they are supported in all current versions of major browsers! However, you probably know that the phrase "current version" fades into insignificance when it comes to the old enemy—Internet Explorer.

Internet Explorer 9 supports all the Level 3 selectors, which is great, but none of the preceding versions of IE support many of the new selectors…which isn't so great. This is where I bring in the first of many reminders about the term *progressive enhancement*.

The usage stats for IE8 (and possibly IE7 depending on the project and your opinion) are still too significant to ignore, so unfortunately you need to keep that in mind when the opportunity to use one of these selectors arises. Just be conscious of how your site or application will react if it is forced to ignore entire rule-sets in IE8 due to lack of support for the CSS. If it compromises the usability or functionality of the site, use an alternative. If the compromise is simply aesthetic, it is probably safe to stick with the advanced selectors and just deliver a *progressively enhanced* version of the website to the modern browsers that can handle it.

Further Reading

A complete list of CSS selectors and browser compatibility
`http://kimblim.dk/css-tests/selectors/`

Selectors Level 3 Specification
`http://www.w3.org/TR/selectors/`