IN THIS CHAPTER

» Working with custom properties

» Handling text that overflows its container

» Styling images

» Hiding and showing elements

» Styling bulleted and numbered lists

» Running feature queries

Bonus Chapter 4 Raising Your CSS Game

HTML elements enable web page designers to mark up a document's structure, but beyond trust and hope, you don't have any control over your text's appearance. CSS changes that. CSS puts the designer in the driver's seat.

—HÅKON WIUM LIE

I designed the chapters in Book 3 of *HTML, CSS, & JavaScript All-in-One For Dummies* to give you a top-notch education in the basics of CSS. That education that continues later in the book with the graduate programs of layout (head over to Book 5) and animation (which you'll find in Bonus Chapters 1 through 3 at `www.dummies.com/go/htmlcss&javascriptaiofd`). For now, though, I want to send you to a kind of finishing school for CSS, where you learn some amazing — and amazingly powerful — techniques that you'll turn to again and again throughout your (hopefully long and successful) CSS coding career. With the CSS you're about to learn, you can officially declare yourself to have graduated from being a CSS amateur to a CSS adept.

In this chapter, you unearth some true CSS gems in the form of custom properties, handling text overflow, styling images, learning how to hide stuff, sprucing up your bulleted and numbered lists, and querying the web browser to check whether it supports particular CSS features.

Easier Coding with Custom Properties and Variables

One of the hallmarks of good web design is consistency. For example, if your main page uses a serif typeface for headings and a sans-serif typeface for body text, then all your other pages should do the same. Thankfully, CSS makes this consistency easy because you can define a rule once in your stylesheet and then use a selector to apply that rule wherever it's needed throughout your site. Need to change that typeface? No problem: Just edit your rule and that change will automatically propagate throughout all the pages that use the stylesheet.

That's pretty sweet, but it doesn't solve a major CSS creation and maintenance problem: having to use a particular property value in multiple declarations throughout a stylesheet.

Take a peek at the following CSS code (check out bk03ch08/example01.html in this book's example files):

```
main {
    border: 5px outset hsl(6deg, 100%, 94%);
    padding: 1rem;
    text-align: center;
}
h2 {
    text-shadow: 5px 5px hsl(6deg, 100%, 94%);
    margin-top: 1rem;
}
button {
    background-color: hsl(6deg, 100%, 94%);
    box-shadow:  5px 5px hsl(6deg, 100%, 94%);
    border-radius: 10px;
    font-size: 1.5rem;
    margin-top: 1rem;
}
```

Anything jump out at you? There are actually two things to notice here, one obvious and the other subtle:

- The more obvious thing is that the color value `hsl(6deg, 100%, 94%)` shows up in four different places. That's a lot of occurrences in a code snippet that has only ten declarations.
- The more subtle thing is that the color value `hsl(6deg, 100%, 94%)` itself doesn't tell you anything about why this color is being used in so many places.

These two points are the bane of CSS developers everywhere. The next couple of sections unpack why.

The scourge of repeated property values

When you're adding rules to a website's stylesheet, do you find yourself repeating certain property values — particularly colors — over and over again? I thought as much. Repeating CSS property values sucks the joy out of web design in several ways:

- Repeating a value over and over is mind-numbing.
- If the value is complex, you may have to find the original to remind yourself of the correct syntax or parameters.
- Changing the value means running a find-and-replace operation over all your CSS code.

What drudgery! Wouldn't it be nice if you had a way to define a value just once and then somehow tell the browser to use that value wherever you need it?

The scourge of non-semantic values

When you're reading through your CSS code, coming across a color value such as `hsl(6deg, 100%, 94%)` or a length value such a `1.75rem` doesn't tell you *why* these particular values are being used. Sure, you could add a comment that says why a value is being used, but if it's a value that repeats frequently throughout your code, you're just adding to the drudgery.

Forget that! Wouldn't it be nice if you had a way to somehow give certain property values a semantic name that tells you instantly the significance of those values?

Welcome to custom properties and cascading variables

The scourges I outline in the previous two sections are neatly resolved by two powerful CSS technologies:

- **Provide a value:** You can use any name you like (subject to certain restrictions, as I describe in the next section) for each custom property, so you're free to create semantic, descriptive names.

» **Cascading variables:** These are property values for which you use a special CSS function that effectively says, “Hey, you know that custom property I defined earlier? Yes, that’s the one. Please insert the value of that custom property here, with thanks.” The “cascading” part of “cascading variable” means that the custom property value cascades from the parent element it was defined on; the “variable” part means that the value that gets inserted varies depending on the value you define in the custom property.

In other words, you can set a particular property value once using the custom property, and then use that custom property value throughout your stylesheet. Need to change the value? No problem: Just edit the custom property as needed and the change automatically propagates down to every cascading variable that uses the file. The result? Drudgery-free CSS!

Defining custom properties

Okay, I realize that the preceding discussions must feel very abstract, so it’s time to get real. To define a custom property, begin with two dashes (`--`) followed by the name (which can use only letters, numbers, hyphens, or underscores and can’t start with a digit or hyphen). For example, this declaration defines a custom property named `--accent-color`:

```
--accent-color: hsl(6deg, 100%, 94%);
```

Next you add your custom property to the element that you want to use as the ancestor. Remember, the value of the custom property cascades down to every descendant of whatever element you use. So, for example, if you’re sure that you’ll use the custom property value only in `aside` elements, your rule will look like this:

```
aside {
    --accent-color: hsl(6deg, 100%, 94%);
}
```

However, custom properties are at their most powerful and useful when they’re available everywhere in your page. That means you’ll want the top of the HTML document (that is, the `html` element) to be the ancestor, and in your CSS, you can reference that element using the `:root` pseudo-class:

```
:root {
    --accent-color: hsl(6deg, 100%, 94%);
}
```

Creating a custom property doesn't do anything on its own because you haven't yet applied the custom property value as a variable. That comes next.

Putting custom properties to work with cascading variables

To use the value of a custom property elsewhere in your CSS, you substitute a regular property's value with a cascading variable that holds the value of the custom property.

You create that variable by using the `var()` function:

```
property: var(--custom-property-name [, fallback-value]);
```

- *`property`*: The name of the CSS property you want to modify.
- *`custom-property-name`*: The name of the custom property you defined earlier, preceded by the two leading hyphens (--).
- *`fallback-value`*: An optional value that the browser will apply if, for some reason, it can't use the custom property (for example, if the custom property name is misspelled).

For example, recall my `--accent-color` custom property from the previous section:

```
:root {
    --accent-color: hsl(6deg, 100%, 94%);
}
```

I can now replace each instance of `hsl(6deg, 100%, 94%)` in my stylesheet with a cascading variable that specifies the `--accent-color` property (check out bk03ch08/example02.html):

```
main {
    border: 5px outset var(--accent-color);
    padding: 1rem;
    text-align: center;
}
h2 {
    text-shadow: 5px 5px var(--accent-color);
    margin-top: 1rem;
}
```

```
button {
    background-color: var(--accent-color);
    box-shadow:  5px 5px var(--accent-color);
    border-radius: 10px;
    font-size: 1.5rem;
    margin-top: 1rem;
}
```

There are two big wins here:

- To change the color, I need only change the value of the custom property.
- The code is more understandable because I can now know right away that I'm applying an accent color to the elements.

My Container Overfloweth: Handling Text Overflow

By default, CSS is happy to give any element whatever space it needs to contain all its content. If you don't set a width or height on the element, the element expands as needed to accommodate the content: first the element's width expands until it reaches the width of its parent element; then the element's height expands until there's enough vertical room to fit everything.

What if you set a width on the element? That's not a usually problem because CSS respects your `width` value and wraps the text at the end of each line. To make that wrapping happen, CSS looks for so-called *soft-wrap opportunities*, such as a space or a hyphen (-). Occasionally, CSS comes across a line that presents no word-break opportunities. For example, check out the following code (bk03ch08/example03.html):

HTML:

```
<h3>Shakespeare Quote of the Day:</h3>
<aside>
   I marvel thy master hath not eaten thee for a word;
for thou art not so long by the head as
honorificabilitudinitatibus: thou art easier
swallowed than a flap-dragon.<br>
—<i>Love's Labour Lost</i>
</aside>
```

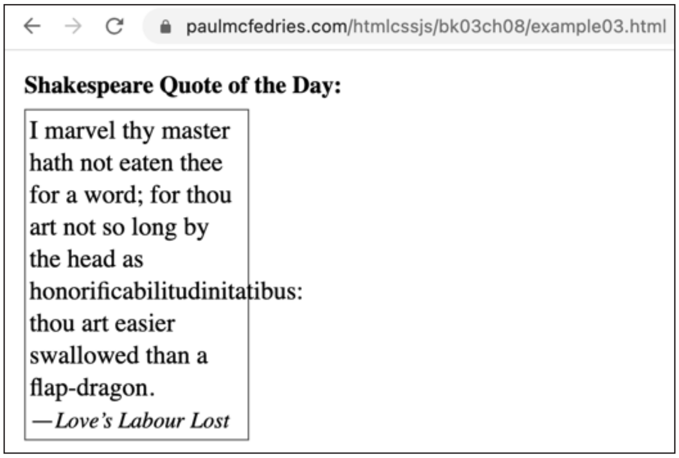
CSS:

```
aside {
    border: 1px solid hsl(0, 0%, 30%);
    width: 175px;
}
```

The `aside` element has a set width, but the text includes a word that's longer than that width, so that line spills out of the `aside` container, as shown in Figure BC4-1.

REMEMBER

Long words are rarely the culprit when it comes to lines spilling out the sides of a container. A more likely suspect — and the bane of web designers everywhere — is a long URL that doesn't include any hyphens.



Shakespeare Quote of the Day:

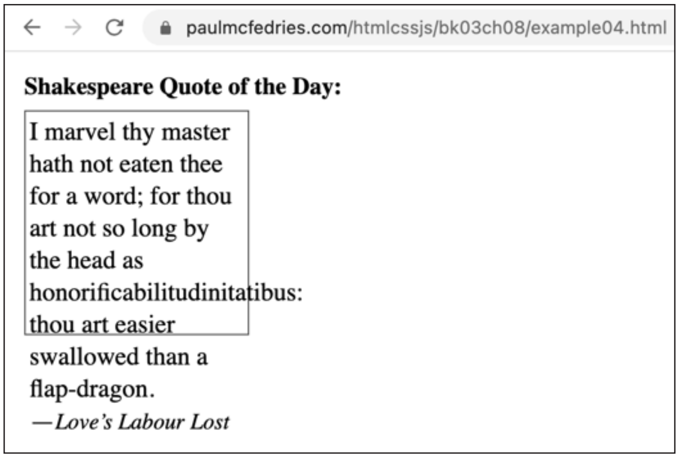
I marvel thy master hath not eaten thee for a word; for thou art not so long by the head as honorificabilitudinitatibus: thou art easier swallowed than a flap-dragon.
—*Love's Labour Lost*

FIGURE BC4-1: If CSS can't wrap a line, it extends the line outside its container.

What if you also set a height on an element? Now CSS has nowhere to expand the element, so if the content is more than can fit inside the constrained element, the text will burst out of the bottom of the element. For example, here I've adjusted the preceding code to add a `height` value to the `aside` element (bk03ch08/example04.html):

```
aside {
    border: 1px solid hsl(0, 0%, 30%);
    height: 175px;
    width: 175px;
}
```

Figure BC4-2 shows the results.



Shakespeare Quote of the Day:

I marvel thy master hath not eaten thee for a word; for thou art not so long by the head as honorificabilitudinitatibus: thou art easier swallowed than a flap-dragon.

—*Love's Labour Lost*

FIGURE BC4-2: If an element doesn't have enough height, its text runs out of the bottom.

Handling overflow

When text runs outside its parent container, that's called *overflow*, and it can create all kinds of problems because the overflow usually flows over something else on the page! Why doesn't CSS just restrict content to its parent? Because doing so would mean truncating the content either vertically or horizontally (or both), and truncating content is a no-no in CSS.

The best solution here is to let the content flow naturally, first by setting the width large enough to avoid having long words spill out horizontally, and second by *not* setting a height on the element.

However, if your page design requires restricting the dimensions of an element, CSS can put you back in charge of your content with the following properties:

```
overflow-x: keyword; /* Handles horizontal overflow */
overflow-y: keyword; /* Handles vertical overflow */
```

- *keyword*: You can use one of the following overflow keywords:
  - `clip`: Truncates the content so that it doesn't extend past the element's padding box.
  - `scroll`: Adds a scroll bar to the element. If you're working with `overflow-x`, a horizontal scroll bar appears if you have content that would otherwise overflow horizontally; if you're working with `overflow-y`, a vertical scroll bar appears if you have content that would otherwise overflow vertically.

- `hidden`: Truncates the content (as with `clip`), allows the content to scroll (as with `scroll`), but doesn't display any scroll bars. Yep, this is a seriously weird behavior! How can you possibly scroll something that has no scroll bars? One possibility is if the element contains tabbable items, such as links or elements that have the `tabindex= "0"` attribute. The user can put the focus inside the element, tab through the content, and it will scroll, if needed.
- `auto`: Lets the browser figure out when to show scroll bars.
- `visible`: Lets the content overflow out of its container (this is the default behavior).

TIP

You can alternatively use the `overflow` property to set both horizontal and vertical overflow keywords simultaneously:

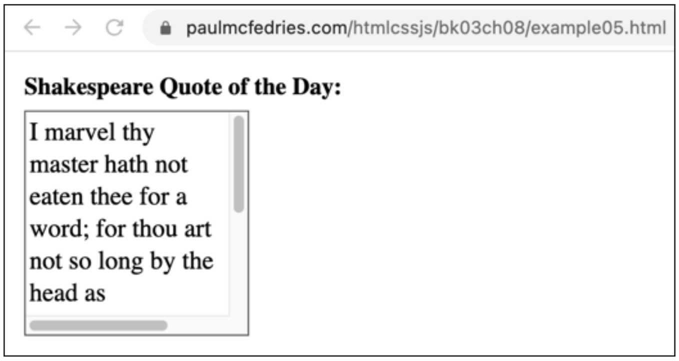
```
overflow: horizontal-keyword vertical-keyword;
```

If you provide a single keyword to the `overflow` property, the browser assigns that value to both the horizontal and vertical behaviors.

In most situations, it's best just to use the `auto` keyword and let the browser handle things for you (bk03ch08/example05.html):

```
aside {
    border: 1px solid hsl(0, 0%, 30%);
    height: 175px;
    overflow: auto;
    width: 175px;
}
```

Figure BC4-3 shows the example elements now with horizontal and vertical scroll bars.


paulmcfedries.com/htmlcssjs/bk03ch08/example05.html

Shakespeare Quote of the Day:

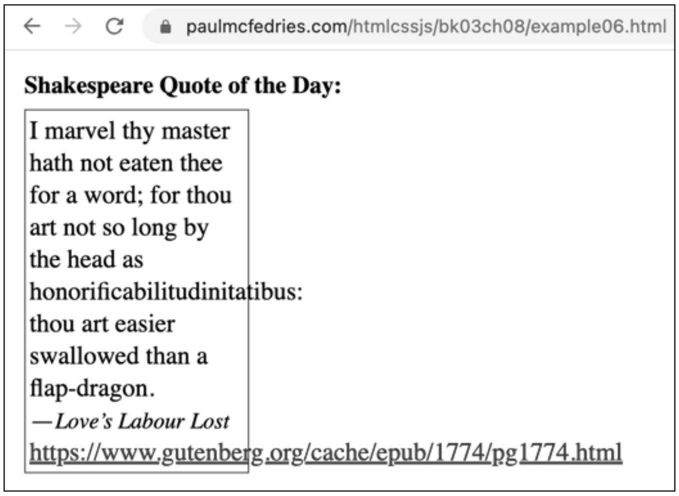
I marvel thy master hath not eaten thee for a word; for thou art not so long by the head as

FIGURE BC4-3: With `overflow: auto`, the browser figures out when scroll bars are required.

Allowing words to break willy-nilly

The soft-wrap opportunities (spaces and hyphens) that CSS uses to wrap text in a container work well most of the time, except when you have a line (such as a really long URL) that offers no such opportunities. Here's an example (bk03ch08/example06.html and Figure BC4-4):

```
<h3>Shakespeare Quote of the Day:</h3>
<aside>
   I marvel thy master hath not eaten thee for a word;
for thou art not so long by the head as
honorificabilitudinitatibus: thou art easier
swallowed than a flap-dragon.<br>
—<i class="small-title">Love's Labour Lost</i>
    <br>
    <a href="https://www.gutenberg.org/cache/epub/1774/pg1774.
  html">https://www.gutenberg.org/cache/epub/1774/pg1774.
  html</a>
</aside>
```


paulmcfedries.com/htmlcssjs/bk03ch08/example06.html

Shakespeare Quote of the Day:

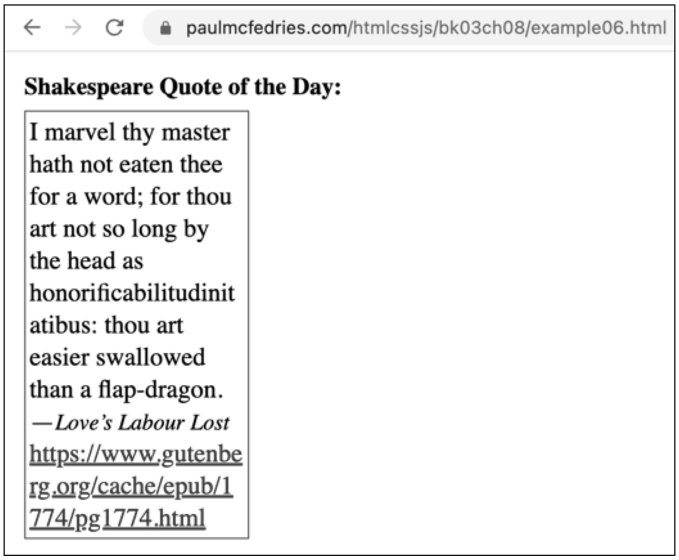
I marvel thy master hath not eaten thee for a word; for thou art not so long by the head as honorificabilitudinitatibus: thou art easier swallowed than a flap-dragon.
—*Love's Labour Lost*
https://www.gutenberg.org/cache/epub/1774/pg1774.html

FIGURE BC4-4: An element with a couple of lines that overflow horizontally.

Rather than use `overflow-x` to introduce a horizontal scroll bar or (worse) to clip the too-long text, you can tell CSS to go ahead and break words anywhere that's necessary to avoid horizontal overflow. You do that by adding the declaration `overflow-wrap: break-word` to your container:

```
aside {
    border: 1px solid hsl(0, 0%, 30%);
    height: 175px;
    overflow-wrap: break-word;
    width: 175px;
}
```

As Figure BC4-5 shows, the browser now wraps the text as needed to prevent overflow, although without adding any hyphens to indicate where the wrapping has occurred. Feel free to add the declaration `hyphens: auto` to fix this.


paulmcfedries.com/htmlcssjs/bk03ch08/example06.html

Shakespeare Quote of the Day:

I marvel thy master hath not eaten thee for a word; for thou art not so long by the head as honorificabilitudinitatibus: thou art easier swallowed than a flap-dragon.
—Love's Labour Lost
https://www.gutenberg.org/cache/epub/1774/pg1774.html

FIGURE BC4-5: The long lines now wrap nicely inside the container.

Handling single-line overflow

Rather than worry about overflow in multiline text, you sometimes have to deal with overflow in text that consists of just one line, such as a text box. That situation is the province of the `text-overflow` property:

```
text-overflow: keyword;
```

» *keyword*: You can use either of the following keywords:

- `clip`: Truncates the text so that it doesn't extend past the element's content box.
- `ellipsis`: Truncates the text within the element's content box, but also adds an ellipsis (. . .) to the end of the visible text.

REMEMBER

For this property to work, you also need to include the following two declarations:

```
overflow: hidden;
white-space: nowrap;
```

The `white-space` property determines how CSS handles whitespace. The `nowrap` value tells CSS to never wrap the text. Other possible values are `pre` (preserves all sequences of whitespace characters and breaks lines at any newline characters in the text); `pre-wrap` (same as `pre`, except lines wrap when they hit the edge of the containing block); and `pre-line` (same as `pre-wrap`, except that sequences of whitespace characters are not preserved).

Here's an example (bk03ch08/example07.html):

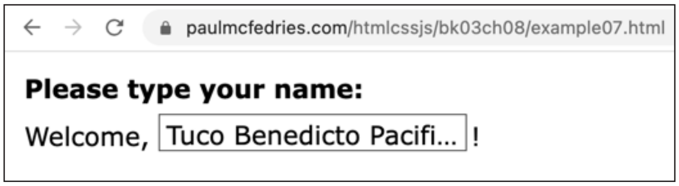
HTML:

```
<label for="name">Please type your name:</label>
<div>Welcome, <span id="name" contenteditable="true">
  </span>!</div>
```

CSS:

```
span {
    border: 1px solid hsl(0, 0%, 30%);
    display: inline-block;
    overflow: hidden;
    text-overflow: ellipsis;
    white-space: nowrap;
    width: 15rem;
}
```

Here you have an editable `span` element prompting the user for their name. You don't want text in this element to overflow (`overflow: hidden`) or wrap (`white-space: nowrap`), and you want any truncated overflow to be represented by an ellipsis (`text-overflow: ellipsis`). Figure BC4-6 shows how the span looks when the user enters a name longer than the `span` width.


paulmcfedries.com/htmlcssjs/bk03ch08/example07.html

Please type your name:

Welcome, Tuco Benedicto Pacifi... !

FIGURE BC4-6: Using an ellipsis (. . .) to represent truncated text.

Styling Images

Images are a weird part of web development because they're a hybrid of both inline and block elements:

- Images are inline elements in the sense that they're rendered along with the text flow.
- Images are block elements in the sense that you can set the `width` and `height` properties on them.

I think you'll find that you'll only rarely want an image to flow along with your text. Therefore, it's always a good idea to add the following rule somewhere near the top of your CSS:

```
img {
    display: block;
}
```

The `display: block` declaration turns all your images into block elements, which makes them easier to work with.

Setting image sizes

With every `<img>` tag you include in your page, it's important to include the `width` and `height` attributes and set these equal to the actual dimensions of the image file you're using. Here's an example:

```
<img src="images/please-brake-for-snakes.jpg" width="704"
   height="1024" alt="">
```

Setting these attributes isn't an instruction to the browser to display the image using those dimensions. Instead, having the width and height enables the browser to calculate the *aspect ratio* of the image, which is the ratio of width to height. Having the aspect ratio allows the browser to set aside the correct amount of space in your layout based on whatever size you set for the image in your CSS. If you don't set the `width` and `height` attributes, the browser sets aside no space for the image, so when it finally loads, your content shifts jarringly to accommodate the image, which can be very annoying for the user.

Near the top of your CSS, besides declaring `display: block` on the `img` element, as I explain in the previous section, you should also set some default width and height values:

```
img {
    display: block;
    max-width: 100%;
    height: auto;
}
```

Setting `max-width: 100%` means that your images never break out of their parent containers; and setting `height: auto` ensures that your images maintain their original aspect ratio if you change the width.

Fitting and positioning images

When you have an image inside a container, you often want to control how that image sits inside the container, which means controlling two things:

- How the image fits within the container's boundaries.
- How the image is positioned within the container.

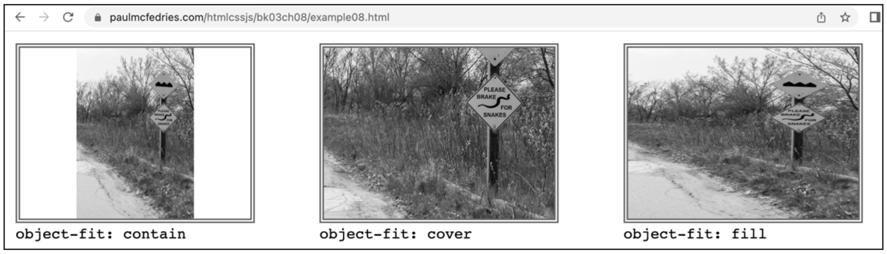
Fitting an image within a container

To determine how the image fits inside the container, use the `object-fit` property:

```
object-fit: keyword;
```

- *`keyword`*: You can use any of the following keywords:
  - `contain`: Preserves the image aspect ratio while scaling the image until it reaches the full width or height (whichever happens first) of the containing element's content box. If there is still space left in the other dimension, the browser leaves that space blank.
  - `cover`: Preserves the image aspect ratio while scaling the image until it covers the full width and height of the element's content box. If the image is larger than the element, the browser crops the image to fit.
  - `fill`: Scales the image until it's the same dimensions as the element's content box. If the image has a different aspect ratio than the element, the image is stretched as needed to fit.

Figure BC4-7 (bk03ch08/example08.html) demonstrates the keywords `contain` (left), `cover` (middle), and `fill` (right).



object-fit: contain
object-fit: cover
object-fit: fill

FIGURE BC4-7: The `object-fit` property in action.

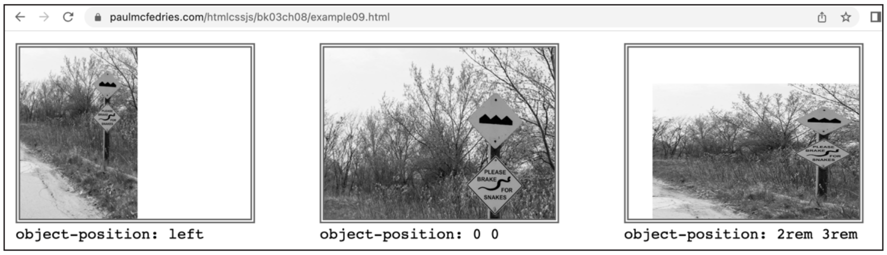
Positioning an image within a container

To set an image's position within a containing element, use the `object-position` property:

```
object-position: x y;
```

- *x*: The horizontal starting position.
- *y*: The vertical starting position.

Both *x* and *y* can be a keyword (`top`, `right`, `bottom`, `left`, or `center`), a percentage, or a length value (such as `50px` or `5rem`). Figure BC4-8 (bk03ch08/example09.html) tries out a few `object-position` values.



object-position: left
object-position: 0 0
object-position: 2rem 3rem

FIGURE BC4-8: The `object-position` property at work.

Applying a filter to an image

One of the more awesome properties you can apply to an image is `filter`, which renders the image using one or more special effects, such as blurring the image or modifying the image colors. Warning! This property is barrel-full-of-monkeys fun and therefore a serious time waster!

Here's the syntax:

```
filter: effect(s);
```

- *`effect(s)`*: Use one or more of the following functions:
  - `blur(`*`radius`*`)`: Blurs the image based on *`radius`*, which is a length value (but not a percentage). The greater the *`radius`* value, the greater the blur.
  - `brightness(`*`value`*`)`: Adjusts the image brightness based on *`value`*. Values under `1` darken the image (`0` produces black) and values over `1` brighten the image.
  - `contrast(`*`value`*`)`: Adjusts the image contrast based on *`value`*. Values under `100%` reduce the contrast (`0%` produces gray) and values over `100%` increase the contrast.
  - `drop-shadow(`*`x y radius color`*`)`: Adds a drop-shadow to the image. The shadow offset is given by *`x`* and *`y`*, the shadow blur is given by *`radius`* (a length value, such as `5px`), and the shadow color is given by *`color`*.
  - `grayscale(`*`value`*`)`: Adjusts the image grayscale based on *`value`*, where `0%` leaves the image as is and `100%` is completely grayscale.
  - `hue-rotate(`*`angle`*`)`: Rotates the image hues around the color wheel by the number of degrees specified by *`angle`*.
  - `invert(`*`value`*`)`: Inverts the image colors based on *`value`*, where `0%` leaves the image unchanged and `100%` is completely inverted.
  - `opacity(`*`value`*`)`: Adjusts the image transparency based on *`value`*, where `100%` is completely opaque and `0%` is completely transparent.
  - `saturate(`*`value`*`)`: Adjusts the image color saturation based on *`value`*, where values under `100%` decrease the saturation (`0%` is completely unsaturated) and values over `100%` increase the saturation.
  - `sepia(`*`value`*`)`: Converts the image colors to sepia based on *`value`*, where `0%` leaves the image unchanged and `100%` is completely sepia.

Figure BC4-9 shows the different `filter` property values applied to an image (bk03ch08/example10.html). To apply multiple effects to an image, include the filter functions you want to use, separating each with a space.

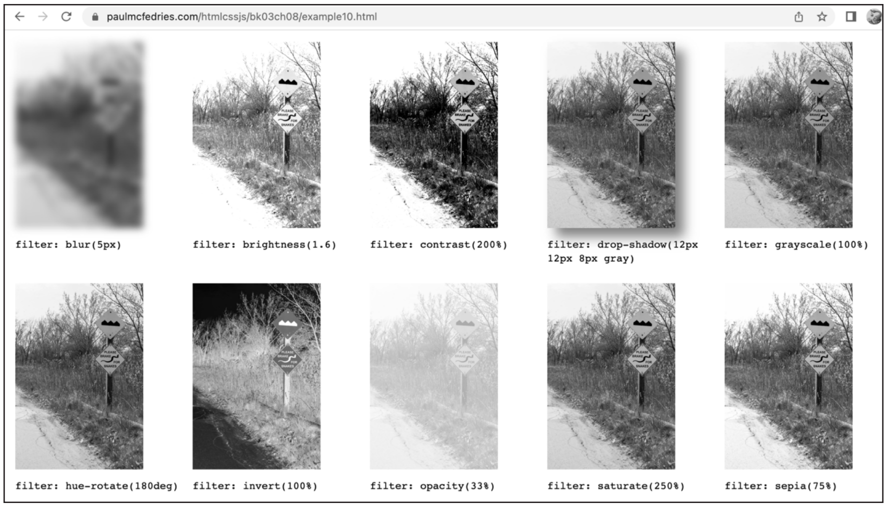


FIGURE BC4-9: The `filter` property doing its thing.

TIP

A similar property is `backdrop-filter`, which, when applied to an element, applies an effect only to the area that lies behind that element. This property supports the same effect functions as the `filter` property. To notice the effect, be sure to apply some transparency to the element's background. As I write this, Safari supports only backdrop filters that use a special `-webkit-` prefix, so if you use this effect, you have to write two declarations (bk03ch08/example11.html):

```
-webkit-backdrop-filter: blur(5px);
backdrop-filter: blur(5px);
```

Keep an eye on the following Can I Use page to learn when this prefix is no longer required:

`https://caniuse.com/css-backdrop-filter`

Hiding (and Showing) Stuff

It may seem like an odd concept, but sometimes you have to hide stuff on your web page. Here are some example scenarios in which you need to hide one or more elements:

- As I discuss in Book 5, Chapter 4, one common strategy for implementing a responsive layout is to hide some nonessential content for users viewing your site on a small (usually smartphone-sized) screen.

- Another responsive layout technique is to have two versions of, say, a `header` element: one for mobile users and one for tablet and desktop users. Depending on the size of the user's screen, you hide one version of the header and display the other.
- Certain types of interface widgets — such as drop-down menus — require some content to be hidden until the user performs an action (such as clicking the menu).
- You may want an element to start off hidden but then gradually become visible via animation (refer to Bonus Chapters 1 through 3).

CSS offers three main ways to hide a page element:

- Remove the element from the page entirely.
- Make the element invisible.
- Make the element transparent.

Removing an element from the page

The most common way to hide an element is to include the following declaration in a rule that targets the element:

```
display: none;
```

This declaration removes the element — and its descendants, if it has any — entirely from the page (technically, it removes the element from the Document Object Model; refer to Book 4, Chapter 6). From the browser's perspective, it's as if the element doesn't exist at all! Here's an example (bk03ch08/example12.html):

HTML:

```
<div>
    First
</div>
<div>
    Second
</div>
<div class="hide-me">
    Third
</div>
```

```
<div>
    Fourth
</div>
```

CSS:

```
.hide-me {
    display: none;
}
```

This example creates four `div` elements, one of which has the `hide-me` class, which declares `display: none` on the element. Figure BC4-10 shows that the third `div` doesn't appear on the rendered page.

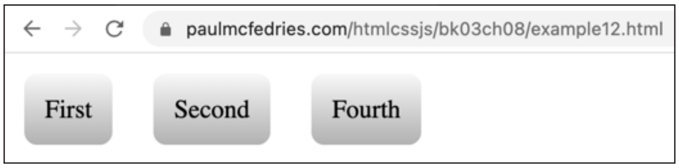


FIGURE BC4-10: The third `div` doesn't appear on the rendered page.

To return a hidden element to the page, you set `display` to the element's intrinsic type:

```
display: block | inline-block | inline;
```

Making an element invisible

Using `display: none` changes the page layout because the subsequent elements fill in the space vacated by the missing element. What if you want to hide an element without changing the page layout? In that case, you'd declare the following property on the element:

```
visibility: hidden;
```

The browser renders the element's box but makes everything inside the box invisible. Figure BC4-11 shows a page that uses the identical code from the previous section, except now the `hide-me` class sets `visibility: hidden` on the third `div` (bk03ch08/example13.html). The third `div` is invisible, but the space for its box remains in the page layout.

To make the element visible again, you use this declaration:

```
visibility: visible;
```

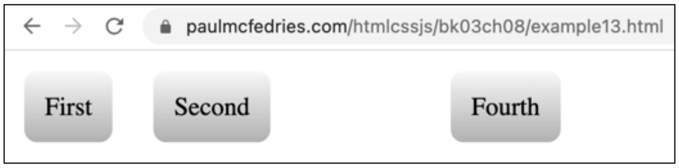


FIGURE BC4-11: The third `div` is invisible, but its box still takes up room in the layout.

Making an element transparent

A third way to hide an element is to make it transparent by setting the element's `opacity` property to `0` (or `0%`):

```
opacity: 0;
```

This declaration is the same as setting `visibility: hidden` on the element. The difference here is that although `visibility` is a toggle (that is, an element's `visibility` property is either `hidden` or `visible`), `opacity` can take any value between `0.0` or `0%` (completely transparent) and `1.0` or `100%` (completely opaque). I make use of this helpful fact when I talk about animating opacity in Bonus Chapters 1 through 3.

Renovating Your Bulleted and Numbered Lists

Bulleted lists and numbered lists (refer to Book 2, Chapter 2) are standard-issue web page features, but that doesn't mean they have to be boring or look the same as every other list on the web. CSS offers a few properties that can inject a little life into your lists.

Customizing bulleted list bullets

The basic bulleted-list bullet is a small, black circle. However, you can customize the bullet in a couple of different ways. The simplest way is to set the `list-style-type` property on the `ul` element, like so:

```
ul {
    list-style-type: type;
}
```

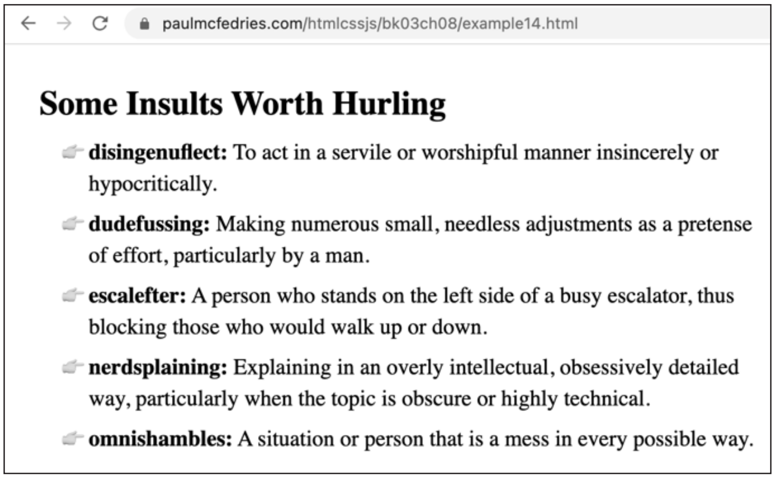
» *type*: Specifies the bullet type:

- **Keyword:** Use `disc` (the standard bullet), `circle`, or `square`. If you don't want any bullet, use `none`.
- **Character(s):** Type the character (or characters), surrounded by quotation marks. For example:

```
list-style-type: ">";
```

- **Unicode number:** Type a backslash (`\`) followed by the character's four- or five-digit Unicode number. (There are lots of Unicode sites on the web, such as Unicodepedia at `https://www.unicodepedia.com/`.) Here's an example (check out Figure BC4-12 and bk03ch08/example14.html):

```
list-style-type: "\1F449";
```


paulmcfedries.com/htmlcssjs/bk03ch08/example14.html

Some Insults Worth Hurling

- **disingenuflect:** To act in a servile or worshipful manner insincerely or hypocritically.
- **dudefussing:** Making numerous small, needless adjustments as a pretense of effort, particularly by a man.
- **escalefter:** A person who stands on the left side of a busy escalator, thus blocking those who would walk up or down.
- **nerdsplaining:** Explaining in an overly intellectual, obsessively detailed way, particularly when the topic is obscure or highly technical.
- **omnishambles:** A situation or person that is a mess in every possible way.

FIGURE BC4-12: Using a pointing finger character as a bullet.

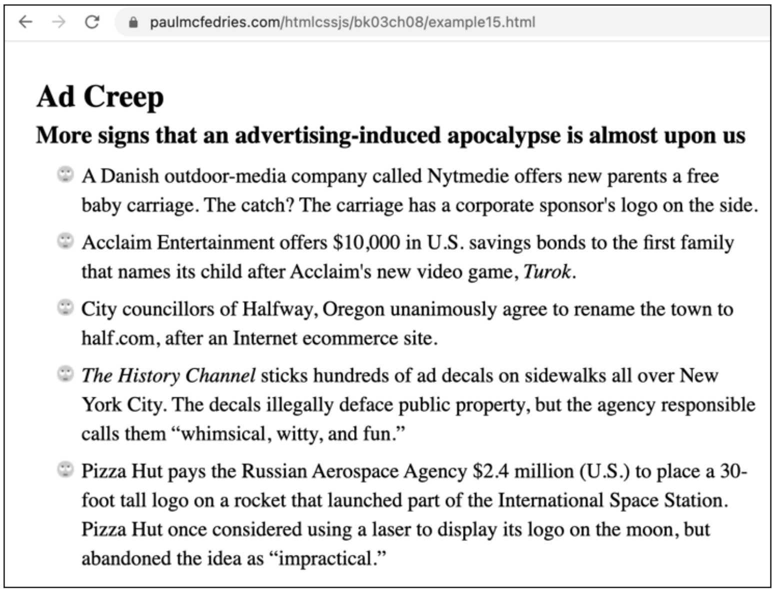
Alternatively, you can use a custom image as a bullet by setting the `list-style-image` property on the `ul` element:

```
ul {
    list-style-image: url(image);
}
```

» *image*: The path and filename for the image you want to use as the bullet.

Here's an example (check out Figure BC4-13 and bk03ch08/example15.html):

```
list-style-image: url(images/eye-roll.png);
```



Ad Creep

More signs that an advertising-induced apocalypse is almost upon us

- A Danish outdoor-media company called Nytmedie offers new parents a free baby carriage. The catch? The carriage has a corporate sponsor's logo on the side.
- Acclaim Entertainment offers $10,000 in U.S. savings bonds to the first family that names its child after Acclaim's new video game, *Turok*.
- City councillors of Halfway, Oregon unanimously agree to rename the town to half.com, after an Internet ecommerce site.
- *The History Channel* sticks hundreds of ad decals on sidewalks all over New York City. The decals illegally deface public property, but the agency responsible calls them "whimsical, witty, and fun."
- Pizza Hut pays the Russian Aerospace Agency $2.4 million (U.S.) to place a 30-foot tall logo on a rocket that launched part of the International Space Station. Pizza Hut once considered using a laser to display its logo on the moon, but abandoned the idea as "impractical."

FIGURE BC4-13: Using a custom image as a bullet.

Customizing numbered list numbers

In most *languages,* the default numbers that precede each item in a numbered list are decimal values: 1, 2, 3, and so on. However, for some variety you can customize the number format by setting the `list-style-type` property on the `ol` element, like so:

```
ol {
    list-style-type: format;
}
```

» *`format`*: Use one of the keywords from the following table:

Keyword	Number format	Example
`decimal`	Standard numbers	1, 2, 3
`decimal-leading-zero`	Standard numbers	01, 02, 03
`lower-alpha`	Lowercase letters	a, b, c
`upper-alpha`	Uppercase letters	A, B, C
`lower-roman`	Small roman numerals	i, ii, iii
`upper-roman`	Large roman numerals	I, II, III

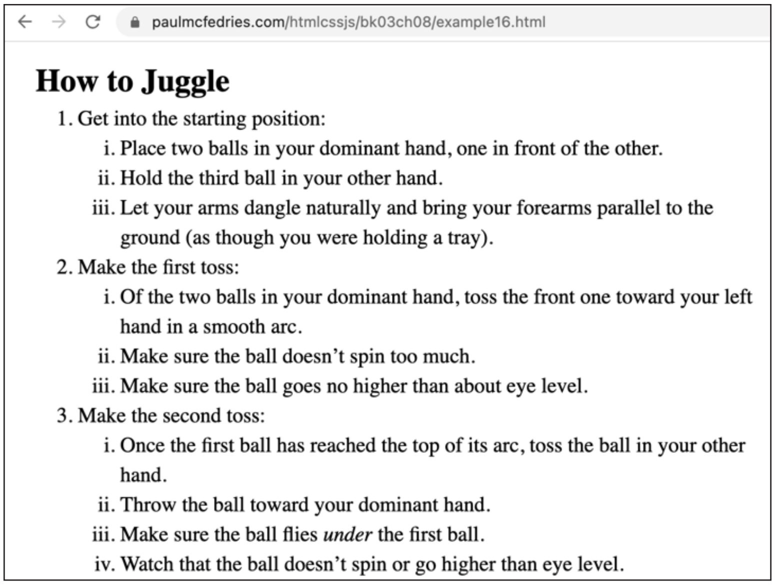
These alternative number formats are particularly useful when you nest one numbered list inside another. The general nesting structure looks like this:

```
<ol>
    <li>First main item</li>
        <ol>
            <li>First nested item</li>
            <li>Second nested item</li>
            <li>Third nested item</li>
        </ol>
    </li>
etc.
```

If you want the nested lists to use small roman numerals, you could use a rule such as the following to target those lists:

```
li > ol {
    list-style-type: lower-roman;
}
```

Figure BC4-14 shows an example (bk03ch08/example16.html).



paulmcfedries.com/htmlcssjs/bk03ch08/example16.html

How to Juggle

1. Get into the starting position:
    i. Place two balls in your dominant hand, one in front of the other.
    ii. Hold the third ball in your other hand.
    iii. Let your arms dangle naturally and bring your forearms parallel to the ground (as though you were holding a tray).
2. Make the first toss:
    i. Of the two balls in your dominant hand, toss the front one toward your left hand in a smooth arc.
    ii. Make sure the ball doesn't spin too much.
    iii. Make sure the ball goes no higher than about eye level.
3. Make the second toss:
    i. Once the first ball has reached the top of its arc, toss the ball in your other hand.
    ii. Throw the ball toward your dominant hand.
    iii. Make sure the ball flies *under* the first ball.
    iv. Watch that the ball doesn't spin or go higher than eye level.

FIGURE BC4-14: Using small Roman numerals with a nested numbered list.

Running Feature Queries

One of the hallmarks of the current age of web development is that new CSS standards make their way into web browsers remarkably quickly. This is particularly true when compared to 10 or 15 years ago, when new features took years to get implemented, if they even got implemented at all (or in the right way)! So, yes, it's awesome that new CSS technologies make it into browsers much faster, but they don't get there instantly. CSS is very complex, and it understandably can still take many months for all the major browsers to support a new feature.

So, as a modern CSS developer, you now have two things you may have to worry about:

- You may have to support very old web browsers that never implemented certain CSS features.
- You may be itching to try very new features that don't yet have wide browser support.

You can safely handle both scenarios by using a CSS technology called the *feature query,* in which you "ask" the web browser whether it supports a particular CSS feature. Two things can happen:

- **The browser supports the feature:** Great! In this case, the browser runs the feature-related code that you've provided.

- **The browser doesn't support the feature:** No problem! In this case, the browser just ignores your feature-related code.

The feature query magic comes from the `@supports` at-rule, which uses the following syntax:

```
@supports (property: value) {
    /* CSS code to run if browser supports property: value */
}
```

- *`property`*: The CSS property that you're querying.
- *`value`*: The value of *`property`* that you're querying.
- CSS code: If the web browser supports *`property`* (or, in some cases, *`value`*), the code inside this declaration block gets executed by the browser.

Note that although you always have to supply both the *`property`* and the *`value`* parameters, you're querying the browser about only one or the other:

- If you want to know only whether the browser supports a specific property, then for the *`value`* parameter you can use anything that's valid for that property. For example:

  ```
  @supports (initial-letter: 2)
  ```

  In this case, you care only whether the browser supports the `initial -letter` property (which, by the way, sets the size of the character targeted by the `::first-letter` pseudo-element), so the value can be anything valid for the property (such as an integer, in this example).

- If you want to know only whether the browser supports a specific value of a property, then for the *`value`* parameter you must specify the value you want to query. For example:

  ```
  @supports (display: subgrid)
  ```

  In this case, you care only whether the browser supports the `subgrid` value of the `display` property.

The way you use feature queries is to write the CSS code that you want the browser to render if it doesn't support the feature, and then use `@supports` to check for feature support and supply some code to run if the browser has implemented the feature. This process is known in CSS World as *progressive enhancement* because you're handling older browsers with code they can work with, and then you add newer code only if the browser supports it. Here's an example (bk03ch08/example17.html):

```
h2 + p::first-letter {
    color: crimson;
    font-size: 32px;
}

@supports (initial-letter: 2) {
    h2 + p::first-letter {
        initial-letter: 5;
    }
}
```

This code styles a `::first-letter` pseudo-element and then uses `@supports` to check for `initial-letter` support. If the browser implements that property, `initial-letter` is set to `5` on the `::first-letter` pseudo element; otherwise, the browser ignores everything inside the `@supports` declaration block.

You can also query on multiple features. For example, to test whether the browser supports two different features, you supply two property/value pairs, separated by the keyword `and`:

```
@supports (property1: value1) and (property2: value2) {
    /* CSS code to run if browser supports both properties */
}
```

To query the browser on whether it supports one feature or another, you supply two property/value pairs, separated by the keyword `or`:

```
@supports (property1: value1) or (property2: value2) {
    /* CSS code to run if browser supports one or both
  properties */
}
```

For example, as I write this, Safari doesn't support the `initial-letter` property, but it does support the prefixed property `-webkit-initial-letter`. So, your `@supports` query should test the browser for one or the other, like so (bk03ch08/example18.html):

```
@supports (initial-letter: 2) or (-webkit-initial-letter: 2) {
    h2 + p::first-letter {
        -webkit-initial-letter: 5;
        initial-letter: 5;
    }
}
```