IN THIS CHAPTER

- Learning how transitions work and what properties can be animated
- Setting up some basic transitions
- Penetrating the mysteries of transition timing functions
- Triggering transitions with JavaScript
- Taking accessibility into account when using transitions

Bonus Chapter 2

Animating CSS Properties with Transitions

Our brains aren't really built for things to just happen*. Companies like Apple understand this, and they account for it in their products. If you have an iPhone, watch it carefully as you do things like lock/unlock it, or switch between apps. Notice how much* life *there is in every interaction and transition.*

—JOSH COMEAU

Besides great content, the awesome pages you come across on the web have a few things in common: good typography, solid layout, and a thoughtful approach to responsiveness and accessibility. Chances are, those pages also have one other feature in common: a sense of liveliness and a kind of twinkle that come from the judicious use of animated effects. You may not even notice these effects, but your brain registers them and probably squirts out some feel-good dopamine in response because your brain reacts positively to the aliveness that animation brings to the page.

In this chapter, you discover how to use animation to inject some élan into your otherwise inert, just-sitting-there pages. In particular, you explore the remarkable universe of CSS transitions, which enable you to animate a huge number of properties with, in many cases, a single line of CSS code. With the techniques you learn in this chapter, you'll no longer be stuck just *applying* a CSS property; you'll be able to *animate* that property. Your page visitors, their brains bathed in dopamine, will thank you for it.

What is a Transition?

All the CSS I talk about in this book has been applied by the web browser in, as Shakespeare once said, "one fell swoop." That is, whether it's applying a color, adding padding, or setting a font size, when the browser comes upon a declaration (and assuming that declaration is a "winner" in the cascade; check out Book 3, Chapter 4), the browser implements the declaration instantly. Even transforms (the subject of Bonus Chapter 1) triggered by, say, the `:hover` pseudo-class are applied right away in the sense that the transform immediately changes the element to its new property value.

On the surface, nothing is inherently wrong with the browser's unseemly haste in applying the CSS it comes across. Speed is good, am I right? Well, not always, because when "speed" means "instantly," you often end up with something that appears unnatural because few things in the real world happen instantly. An object is in one state and then it changes to another state, and that change usually goes through a continuum of intermediate states. There is, in short, a transition from one state to another and it's via that transition that we make sense of the change and perceive the change as "natural."

You can achieve that sense of naturalness in your web pages by asking the web browser to apply some properties not right away, as usual, but via a continuous series of intermediate states. This is called a *transition* and, as you learn in the rest of this chapter, with transitions you can control not only which properties are animated but also the animation trigger, the animation duration, and the animation timing.

WARNING

In this chapter's introduction, I use the phrase "*judicious* use of animated effects" with italics here for emphasis. Why? Because, believe me, nobody wants to see fireworks exploding while your page loads. Nobody wants to be inundated by Hollywood-quality special effects when a menu opens. Nobody wants to hover the mouse pointer over a button and sit through some interminable and ultimately pointless transformation of the element.

Go overboard with animations and you'll enrage your visitors. Keep your animations subtle and useful and you'll delight them. Your call!

Knowing which Properties You Can Animate with Transitions

Basically, if a property takes values where it makes sense that changing from one value to another can take place over time in a continuous series of intermediate stages, that property is almost certainly animatable with a transition (or any of the CSS animations that I discuss in Bonus Chapter 3).

For example, the `opacity` property can take a numeric value from `0` to `1` (or `0%` to `100%`), where `0` means the element is fully transparent and `1` means the element is fully opaque. When setting, say, `opacity: 0` on an element, it makes sense that the change from the default of `opacity: 1` could take place over time in a series of intermediate stages. That is, instead of becoming immediately transparent, the element could fade out over the course of a second or two.

By contrast, changing, say, the `font-family` property from one typeface to another is a discrete change, meaning that the idea of having intermediate states that change over time doesn't make sense.

TIP

The Mozilla Developer Network maintains an exhaustive list of the CSS properties that are animatable here: `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_animated_properties`.

How Transitions Work

Defining a transition requires four things in your CSS for a specific element (or selector):

- **The parameters of the transition.** In particular, which element property you want to animate and how long the animation should take.
- **The initial value for the property you're animating.** If the initial value is just the default value for the property, you don't need to specify the value.

- **The state that triggers the transition.** This is usually an action-based pseudo-class such as `:hover`, `:focus`, or `:active`. However, you can also use JavaScript to trigger a transition (check out "Using code to trigger a transition," later in this chapter).
- **The final value for the property you're animating.** You declare this final value within the trigger rule.

Given these items, the animation process goes something like this:

1. **The browser renders the element with the initial (or default) property.**
2. **The user (or a script) triggers the animation.**

   For example, if you used the `:hover` pseudo-class, the browser triggers the animation when the user hovers the mouse pointer over the element.
3. **The browser runs the transition based on the parameters you specified.**

   For example, if the property you're animating is a `transform` using the `scale()` function and your final property value is `scale(1.5)`, the browser begins scaling the element from its initial value, such as `scale(1)`.
4. **When the transition reaches the final property value, the browser ends the animation.**

   The browser runs the animation for the duration you specify.

Setting Up a Basic Transition

Defining a transition is really a two-stage process. In the first stage, you apply the initial property value (unless you're starting with the default property value) and then add the `transition` properties. At its most basic, a transition requires two parameters: the property you're animating and the duration of the animation. You can either use individual properties for this or the `transition` shorthand property:

```
transition-property: property-name;
transition-duration: time;
transition: property-name time;
```

- *`property-name`*: The name of the property you want to animate. You can also use the keyword `all` to tell the browser to transition all the animatable properties that change with the trigger.

» *time*: A value expressed in seconds (`s`) or milliseconds (`ms`) that specifies how long the transition takes to run from the initial property value to the final property value.

For example, suppose you have the following `button` element (check out bk06ch02/example01.html in this book's example files):

```
<button>Hover over me</button>
```

Suppose, as well, that you want to animate a transform. Here's the setup code for that:

```
button {
    transition-property: transform;
    transition-duration: 1s;
}
```

So far, so good. The second stage is to set up the trigger and use that rule to specify the final value of the transition. For the example, you could do something like this:

```
button:hover {
    transform: scale(2);
}
```

This tells the browser that when the user hovers the mouse pointer over the button, the browser should smoothly transition the button to twice its default size. Figure BC2-1 shows the original button, and Figure BC2-2 shows the button at the end of the transition. Note, as well, that when the user moves the mouse pointer off the button, the browser transitions the button back to its original state.

FIGURE BC2-1: The original button.

FIGURE BC2-2: The button scaled to twice its default size at the end of the transition.

Adding transitions to the toggle switch

As another example, in Bonus Chapter 1, I illustrated the translate transform by modeling a toggle switch. In particular, clicking that switch did two things:

- Used the `transform: translateX()` property to shift the toggle button to the left by `24px`
- Used the `background-color` property to change the toggle background color to green

Both properties are animatable, so here's some code that sets up a transition for each property (check out bk06ch02/example02.html):

```
.toggle {
    transition: background-color 0.75s;
}
.toggle::after {
    transition: transform 0.75s;
}
#cb-toggle:checked + .toggle {
    background-color: hsl(102deg 58% 39%);
}
#cb-toggle:checked + .toggle::after {
    transform: translateX(24px);
}
```

Specifying multiple transitions

You don't have to work with one transition at a time. CSS is perfectly happy if you specify two or more transitions for a given element. You define multiple transitions by using `transition-property` to specify a comma-separated list of the

properties you want to animate, and by using `transition-duration` to specify a comma-separated list of the transition durations:

```
transition-property: name1[, name2..., nameN];
transition-duration: time1[, time2..., timeN];
transition: name1 time1[, name2 time2..., nameN timeN];
```

The browser applies the duration *time1* to the transition for the property *name1*, the duration *time2* to *name2*, and so on. Here's an example (bk06ch02/example03.hml):

```
button {
    background: hsl(100, 61%, 75%);
    color: hsl(0, 0%, 10%);
    transition-property: background, color, transform;
    transition-duration: 2s, 1s, 3s;
}
button:hover {
    background: hsl(100, 61%, 25%);
    color: hsl(0, 0%, 90%);
    transform: scale(2);
}
```

REMEMBER

If there are fewer durations than there are properties, the browser repeats the durations as needed so that every property has a duration. This means that if you want every property transition to use the same duration, you need to specify only a single time value:

```
transition-property: name1[, name2..., nameN];
transition-duration: time;
```

Delaying the Transition

It's occasionally useful to delay the start of a transition. When you're running multiple transitions, for example, you may want one transition to end before starting the next one. You can delay any transition by adding the `transition-delay` property:

```
transition-property: property-name;
transition-duration: duration-time;
transition-delay: delay-time;
transition: property-name duration-time delay-time;
```

- *property-name*: The name of the property you want to transition
- *duration-time*: The length of the transition in seconds (`s`) or milliseconds (`ms`)
- *delay-time*: The amount of time you want the browser to wait before starting the transition, expressed in seconds (`s`) or milliseconds (`ms`)

Note, in particular, that when you use the `transition` shorthand and you specify two time values, the first time value is interpreted as the transition duration and the second time value is interpreted as the transition delay. Here's an example (bk06ch02/example04.hml):

```
button {
    background: hsl(100, 61%, 75%);
    color: hsl(0, 0%, 10%);
    transition-property: background, color, transform;
    transition-duration: 1s;
    transition-delay: 0s, 1s, 2s;
}
button:hover {
    background: hsl(100, 61%, 25%);
    color: hsl(0, 0%, 90%);
    transform: scale(2);
}
```

Adding a Timing Function

CSS transitions work by taking the initial value of the animating property, the final value of that property, and then applying a series of intermediate values that change the property from the initial to the final value within the specified transition duration. This process of calculating a series of intermediate values is called *interpolation.*

Intuitively, you'd think that interpolation would work something like this:

1. Subtract the final property value from the initial property value.
2. Divide the result of Step 1 by the number of milliseconds in the transition duration.
3. Run the transition by continually incrementing the property value by the result of Step 2 until the final value is reached.

In other words, intuitively you may think that the "speed" of the transition would be constant from the start of the animation to the end. However, when you make a close examination of any transition, it becomes clear that the speed isn't constant at all. What actually happens is that the transition speeds up quickly to about the halfway point, and then slows down to the finish. What's up with that?

That, my friend, is CSS trying to make transitions appear more natural. After all, when an action occurs in the real world, it's rare for it to run at a constant rate. A car takes a bit of time to get up to speed, and then it slows before it comes to a stop; a snowball rolling down a hill starts slowly and then picks up speed as it goes; a rubber ball dropped from a height bounces a few times when it hits the floor.

CSS enables you to mimic these and similar natural behaviors by defining how the speed of an animation varies throughout its duration. You define this animation *timing* by specifying a *timing function* using the `transition-timing-function` property:

```
transition-property: property-name;
transition-duration: duration-time;
transition-delay: delay-time;
transition-timing-function: timing-function;
transition: property-name duration-time delay-time
  timing-function;
```

- *property-name*: The name of the property you want to transition
- *duration-time*: The length of the transition in seconds (`s`) or milliseconds (`ms`)
- *delay-time*: The amount of time you want the browser to wait before starting the transition, expressed in seconds (`s`) or milliseconds (`ms`)
- *timing-function*: A keyword or function that specifies the timing you want to use for the transition

Timing function keywords

Table BC2-1 lists the keywords you can use with the `transition-timing-function` property.

Because most of these keywords ease (that is, slow down) the transition beginning and/or end, they're also known as *easing functions.*

TABLE BC2-1 **Keywords for the `transition-timing-function` Property**

Keyword	Description	`cubic-bezier()` Equivalent
`ease`	Starts quickly, speeds up to the midpoint of the transition, and then slows to the end. This is the default timing function.	`cubic-bezier (0.25, 0.1, 0.25, 1)`
`ease-in`	Starts slowly and then speeds up to the end of the transition.	`cubic-bezier (0.42, 0, 1, 1)`
`ease-out`	Starts quickly and then slows down to the end of the transition.	`cubic-bezier (0, 0, 0.58, 1)`
`ease-in-out`	Starts slowly, speeds up to the midpoint of the transition, and then slows to the end.	`cubic-bezier (0.42, 0, 0.58, 1)`
linear	The speed is constant throughout the transition.	`cubic-bezier(0, 0, 1, 1)`

Here's an example (bk06ch02/example05.html):

```
button {
    transition-property: transform;
    transition-duration: 2s;
    transition-timing-function: ease-in-out;
}
button:hover {
    transform: translateX(50vw);
}
```

That cubic-bezier() gobbledygook

Do you have deep furrows in your brow over the last column in Table BC2-1 and its scary contents? You're not alone, believe me. The `cubic-bezier()` function is an intimidating beast, for sure, but really it's just another way to express a transition timing function by defining a curve (called a cubic-Bezier curve by math nerds) that the transition "follows." The slope of the curve at a given point determines the current speed of the transition at that point:

- The closer the slope of the curve is to horizontal, the slower the transition speed.
- The closer the slope of the curve is to vertical, the faster the transition speed.

The `cubic-bezier()` function values are coordinates on an x-y plane where x represents the animation time and y represents the progression of the animating property. The point (0,0) represents the beginning of the transition, and the point (1,1) represents the end of the transition. The `cubic-bezier()` function values are points between these two that deform the curve and, hence, the speed of the transition.

Okay, I get it: The `cubic-bezier()` function just isn't easy to grasp, at least in part because you have no way to examine the function's values and intuit the curve they create. So, forget that.

Instead, you can also use your web browser's dev tools to play around with `cubic-bezier()` function values. Right-click the element that has the defined transition and then click Inspect to display the element in the dev tools. Next to the element's `cubic-bezier()` function, click the Open Cubic Bezier Editor icon to launch the editor, shown in Figure BC2-3. (Figure BC2-3 shows the Chrome tool; all the major browsers offer similar tools.)

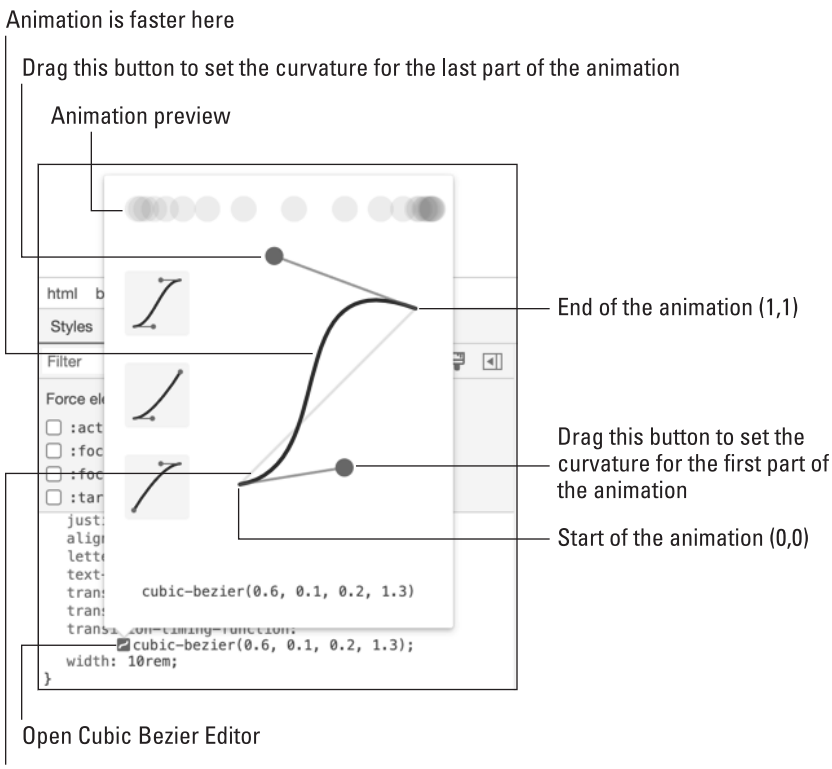

FIGURE BC2-3: In the browser dev tools, use the Cubic Bezier Editor to build your cubic-Bezier curves visually.

The Cubic Bezier Editor is a coordinate plane that shows animation time on the x-axis and animation progression on the y-axis. The lower-left point (0,0) is the start of the animation, while the upper-right point (1,1) is the end of the animation. Your job is to drag the other two points (represented by the circles attached to line segments) in a way that defines the curve you want (remembering that near-horizontal parts of the curve are where the animation runs slowly, whereas near-vertical parts of the curve are where the animation runs quickly). You can also click a preset timing function on the left side.


TIP

Feel free to drag the second curve button (the one attached to the end point of the animation) above the plane, which will give you a "y" value of greater than 1. This makes the animating property go "past" its final value briefly, which adds a kind of "bounce" effect. (It is, admittedly, a pretty lame bounce effect. For the real bounce deal, you need to set up animation frames, as I discuss in Bonus Chapter 3.)

Here's an example (bk06ch02/example06.html):

HTML:

```
<div class="ball" tabindex="0">Click me</div>
```

CSS:

```
body {
    height: 100vh;
}
.ball {
    border-radius: 50%;
    height: 10rem;
    width: 10rem;
    transition-property: transform;
    transition-duration: 1.5s;
    transition-timing-function: cubic-bezier(0.5, 0.96, 0.75,
  1.44);
}
.ball:focus {
    transform: translateY(calc(100vh - 10rem));
}
```

This code turns a `div` element into a ball that, when clicked, shifts the ball from the top of the viewport to just past the bottom of the viewport, and then finally back up to the bottom of the viewport.

Scripting Transitions with JavaScript

In pure CSS, you can trigger an element transition by putting the property (or properties) you want to animate inside an element pseudo-class such as `:hover`, `:focus`, `:focus-within`, or `:active`. (Head for Book 3, Chapter 3, to learn about these and other pseudo-classes.) These triggers are limited, but they may be all you need.

However, if you want a bit more control over your transitions, you need to get JavaScript on the job, which enables you to set up a wider range of transition triggers as well as to run some code when a transition ends.

Using code to trigger a transition

To use JavaScript to trigger a transition, you need to set up your CSS and JavaScript as follows:

- **CSS:** You need to do two things:
  - Style the element you want to animate with the `transition-*` properties you want to use.
  - Create a class that contains the final values of the property or properties you want to transition. So, whereas before you put these declarations in a pseudo-class rule, now you put them in a class rule. Make sure the element you're animating doesn't use this class by default.
- **JavaScript:** You need to set up two things here as well:
  - Create an event listener that will trigger the transition. For example, you may want to listen for clicks on the element you're animating.
  - In the event listener callback function, use the `classList` object's `add` method to add the class on the element. (For the details on the `classList` object, check out Book 4, Chapter 6.) Because the element doesn't have the class by default, the `add` method inserts the class, which triggers the transition.

Here's an example (bk06ch02/example07.html):

HTML:

```
<div class="ball" tabindex="0">Click me</div>
```

CSS:

```
body {
    height: 100vh;
}
.ball {
    border-radius: 50%;
    height: 10rem;
    width: 10rem;
    transition-property: transform;
    transition-duration: 1.5s;
    transition-timing-function: cubic-bezier(0.5, 0.96, 0.75,
  1.44);
}
.move-ball {
    transform: translateY(calc(100vh - 10rem));
}
```

JavaScript:

```
// Listen for clicks on the ball element
document.querySelector('.ball').addEventListener('click',
  (event) => {

    // Add the 'move-ball' class to the element
    event.target.classList.add('move-ball');
});
```

The HTML and CSS are the same as in the example from the previous section, but now the `transform` declaration resides in the `.move-ball` rule. The JavaScript listens for clicks on the element with the `.ball` class. In the event handler, the script uses `event.target` to reference the element (that is, in this case, `event.target` is a reference to the element that was clicked), and then adds the `move-ball` class to trigger the transition.

Here's a slightly more involved example that animates a drop-down menu of links (bk06ch02/example08.html):

HTML:

```
<nav>
    <button class="nav-menu-button">
        Menu
    </button>
```

```
    <div class="nav-menu-dropdown-contents" aria-role="menu">
        <a href="" class="dropdown-contents-link">Home</a>
        <a href="" class="dropdown-contents-link">Products</a>
        <a href="" class="dropdown-contents-link">Blog</a>
        <a href="" class="dropdown-contents-link">Contact</a>
    </div>
</nav>
```

CSS:

```
.nav-menu-dropdown-contents {
    height: 0;
    left: 0;
    opacity: 0;
    position: absolute;
    top: 75px;
    transition: all 0.75s ease-out;
}
.nav-menu-dropdown-contents.show {
    height: fit-content;
    opacity: 1;
}
.nav-menu-dropdown-contents > a {
    display: block;
    border-radius: 5px;
    font-size: 1.25rem;
    padding: 4px 8px;
    text-decoration: none;
}
```

JavaScript:

```
// Get the Menu button
const menuButton = document.querySelector('.nav-menu-button');

// Listen for clicks on the Menu button
menuButton.addEventListener('click', () => {

    // Get the dropdown const menuDropdownList = document.
    querySelector('.nav-menu-dropdown-contents');

    // Transition the dropdown by toggling the 'show' class
    menuDropdownList.classList.toggle('show');
});
```

The HTML defines a `nav` element that include a Menu `button` element and a `div` that includes four navigation links. The CSS styles the drop-down `div` (via the selector `.nav-menu-dropdown-contents`) with `height: 0` and `opacity: 0`, which hides the menu by default. When the `show` class is added (via the selector `.nav-menu-dropdown-contents.show`), the `div` is transitioned to `height: fit-content` and `opacity: 1` to make it appear. The JavaScript listens for clicks on the Menu button, and the event handler uses the `toggle` method to add the `show` class on the drop-down menu to trigger the transition.


REMEMBER

Why did I use the `toggle` method instead of the `add` method to trigger the transition? Because in this case I want the transition to be reversible. That is, if the user clicks the Menu button a second time, I want the menu to disappear. The easiest way to do that is to toggle the class on and off.

Chaining multiple transitions

One of the nice things about transitions is that, especially compared to the keyframe-based animations that I talk about in Bonus Chapter 3, they're simple. A transition gets triggered, it runs, and then it stops. Done!

That simplicity is great if you just want to add some subtle effects to your page interface. However, you may find that you sometimes need a little something extra to get the effect you want. I'm not talking about adding more animatable properties to the transition. Nothing wrong with that (as long as you don't go overboard with it), but I'm after bigger game here: Running an entirely different transition after the first one ends. This is called *chaining* transitions, and it enables you to run one transition after another, which gives you extra scope for all kinds of interesting effects.


WARNING

Before getting to the code, let me stress, again, that you don't want to let things get out of hand here. You can usually get away with chaining two, perhaps three, short transitions, but beyond that you're asking *way* too much of your site visitors.

The secret to chaining transitions is that each transition fires a `transitionend` event when a transition finishes. This means you can set up some JavaScript code that listens for this event. When it fires, your callback function adds whatever class you've defined for the second transition.

Here's an example (bk06ch02/example09.html):

HTML:

```
<div class="ball" tabindex="0">Click me</div>
```

CSS:

```
body {
    height: 100vh;
}
.ball {
    border-radius: 50%;
    height: 10rem;
    width: 10rem;
    transition-property: transform, background-color, color;
    transition-duration: 1s, 750ms, 750ms;
    transition-timing-function: ease;
    width: 10rem;
}
.move-ball {
    transform: translateY(calc(100vh - 10rem));
}
.color-ball {
    background-color: hsl(0deg 75% 60%);
    color: hsl(45deg 100% 90%);
}
```

JavaScript:

```
// Listen for clicks on the ball element
document.querySelector('.ball').addEventListener('click',
  (event) => {

    // Add the 'move-ball' class to the element
    event.target.classList.add('move-ball');
});

// Listen for the transitionend event on the ball
document.querySelector('.ball').addEventListener('transition
end', (event) => {

    // Add the 'color-ball' class to the element
    event.target.classList.add('color-ball');

    // Is this the end of the second transition?
    if (event.propertyName === 'color') {

        // If so, change the ball text
        event.target.innerHTML = 'Thanks!';
    }
});
```

In the `.ball` rule, notice that `transition-property` now specifies three properties, and `transition-duration` now specifies three durations. The CSS also includes a `.color-ball` rule that changes the `background-color` and `color` properties. This rule defines the second transition.

In the JavaScript, the `.ball` element has an event listener for the `transitionend` event that, when fired, adds the `.color-ball` class to the element. Just for fun, the `transitionend` callback function also checks whether the transition event's property name is `color`, which means this is the end of the second transition, so the code then modifies the ball text.

TIP

If you want to run some code when a transition starts, set up a listener for the `transitionstart` event.

Making Your Transitions Accessible

When used with purpose and restraint, transitions can enhance your page interface and delight your visitors. However, you need to be careful here because certain kinds of animation can be extremely annoying for people with cognitive conditions such as attention deficit hyperactivity disorder (ADHD); they can be triggering for people who have epilepsy, vestibular disorders, or migraine sensitivity; and they have been shown to cause problems such as dizziness, headaches, and nausea. The bigger or more frenetic the animated effect, the greater the chance it will cause problems for anyone subjected to it.

So, does this mean you should never use animation on your site? No, that's too drastic because there are ways to give users control over the animations they encounter:

- **JavaScript:** If your site has a "Settings" feature through which users can customize their experience, be sure to add a setting that lets users turn off animations. (If you don't implement settings, an alternative would be a checkbox or switch somewhere on each page.) In your animation code, you could then get the setting from local storage and test it: If the user doesn't want animation, then just before you add the transition class, set the `transition-duration` property to a very small value (bk06ch02/example10.html):

```
event.target.style.transitionDuration = '0.01ms';
```

» **CSS:** The `@media` at-rule has a `prefers-reduced-motion` media feature that will be set to either `reduce` (if the user has set the operating system preference for reduced motion) or `no-preference` (if the user hasn't set such a preference). This means you can wrap your transition declarations in a media query, like so (bk06ch02/example11.html):

```
@media (prefers-reduced-motion: no-preference) {
    .nav-menu-dropdown-contents {
        transition: all 0.75s ease-out;
    }
}
```