3 Creating a responsive animated loading screen

This chapter covers

- Creating basic shapes using Scalable Vector Graphics (SVGs)
- Finding out the difference between viewboxes and viewports in SVGs
- Understanding keyframes and animating SVGs
- Using animation properties
- Styling SVGs with CSS
- Styling an HTML progress bar element with appearance properties

We see loaders in most applications today. These loaders communicate to the user that something is loading, uploading, or waiting. They give the user confidence that something is happening.

Without some sort of indicator to tell the user that something is happening, they may try reloading, click the link again, or give up and leave. We should be using some sort of progress indicator when an action takes longer than 1 second, which is when users tend to lose focus and question whether there's a problem. As well as having a graphic showing that something is happening, the loader should be accompanied by text that tells the user what is happening to improve the accessibility of the web page for screen readers and other assistive technologies.

For our animation, we'll be looking into the CSS Animation Module, understanding the animation property, keyframes, and transitions, as well as accessibility and respect for user preferences.

.1 Setup

In this project, we'll be creating rectangles within an SVG. We'll see what SVGs offer and understand the slight differ-

ences between styling HTML elements and SVG elements.

We'll also create a progress bar, which shows the user how much of the task has been completed and how much is left to go. We'll use the HTML `<progress>` element and then look at how we can edit the browser's default styles and apply our own. Overall, we want to create a consistent, responsive loader that works across devices. Figure 3.1 shows the result.

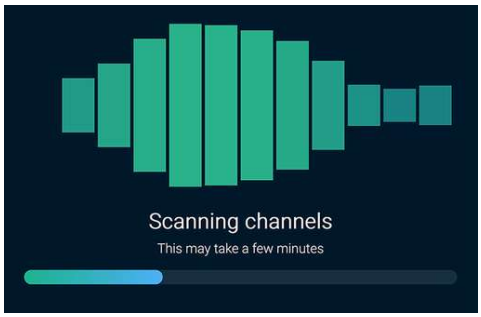

Figure 3.1 Goal for this chapter

The code for this project is in the GitHub repository (**https://github.com/michaelgearon/Tiny-CSS-Projects**) in the chapter 3 folder. You can find a demonstration of the completed project on CodePen at **https://codepen.io/michaelgearon/pen/eYvVVre**.

.2 SVG basics

SVG stands for *Scalable Vector Graphics*. SVGs are written in an XML-based markup language and consist of vectors on a Cartesian plane. Vector graphics can be coded from scratch but often are created in a graphics program such as Adobe Illustrator, Figma, or Sketch. Then they're exported in the SVG file format and can then be opened in a code text editor.

A *vector* is a mathematical formula that defines a geometric primitive. Lines, polygons, curves, circles, and rectangles are all examples of geometric primitives.

A *Cartesian coordinate system* in a plane is a grid-based system that defines a point by using a pair of numerical coordinates based on the point's distance from two perpendicular axes. The location where these two axes cross is the *origin*, which has a coordinate value of `(0, 0)`. Think back to math class; when you were asked to plot lines on a graph, you were using a Cartesian coordinate system. Essentially,

SVGs are shapes on a coordinate plane written in XML.

By contrast, PNGs, JPEGs, and GIFs are *raster images*, which are created by using a grid of pixels. Figure 3.2 illustrates the difference between raster and vector graphics.

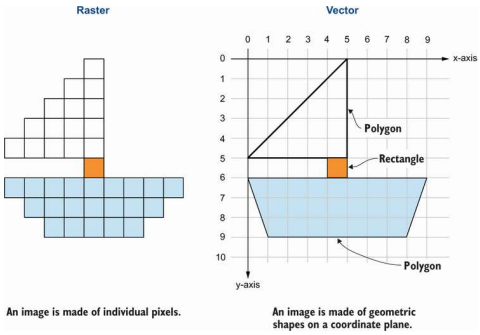

Figure 3.2 Raster versus vector graphics

SVGs have many advantages over raster images, including being infinitely scalable. We can shrink or enlarge the image as much as we want without losing quality. We can't enlarge raster images without seeing *pixelation*, which results from enlarging the grid of pixels that renders the individual squares of the grid visible. By contrast, when we enlarge an SVG, we're setting shapes and lines on a coordinate plane programmatically; the paths between points are redrawn, and the quality doesn't degrade.

Because SVGs are written in XML, we can place SVG code directly in our HTML and access, manipulate, and edit it in much the same way that we do our other HTML elements. SVGs are to graphics as HTML is to web pages.

Rasters, however, are a better choice for dealing with images that are highly complex, such as photos. It's possible to create a photorealistic image by using an SVG, but it wouldn't be practical. The file size and, therefore, load performance are significantly larger for vector graphics than for raster images.

The most common use case for SVGs are logos, icons, and loaders. We use them for logos because logos are often simple images that need to be crisp regardless of the size or medium. Furthermore, it's not uncommon for a company or product to have several versions of a logo for use on a dark background versus a light background. Recoloring, simplicity, and scaling are other reasons why we use SVGs for icons.

We use SVGs for loaders because unlike their raster counterparts, they allow us to add animations inside the image itself. We can isolate an individual element inside the graphic and apply CSS or JavaScript to that individual piece—an exercise that isn't possible with rasters.

Earlier, we mentioned that SVGs are based on a Cartesian plane (a 2D coordinate plane). Let's look into what that means and how it works.

3.2.1 Positions of SVG elements

When we're working with SVG elements, the way to think about positioning is to imagine that we're placing elements on a grid. Everything starts at `(0,0)` (the origin) which is the top-left corner of the SVG document. The higher the `x` or `y` value is, the farther it is from the top-left corner. Figure 3.3 expands on the example of the boat in figure 3.2, adding the origin and the coordinate values for each shape.

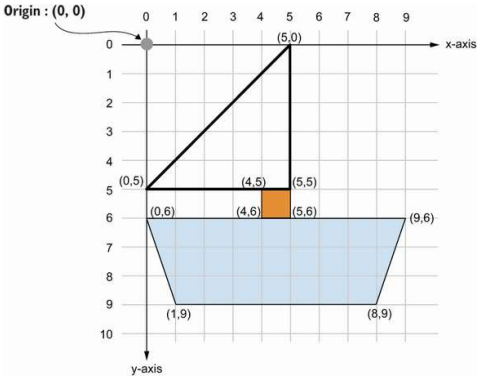

Figure 3.3 Positioning elements on a coordinate plane

The loader in our project is composed of a series of 11 rectangles. To place them, we need to think of their positions on a coordinate plane, taking both their widths and the gaps between them into consideration.

3.2.2 Viewport

The *viewport* is the area in which the user can see the SVG. It's set by two attributes: `width` and `height`. Think of the viewport as being a picture frame: it sets the size of the frame but doesn't affect the size of the graphic it contains. If we place an image inside a picture frame that's larger than the frame, however, we have overflow. The same thing happens to our SVG. As in CSS positioning, viewport measurements have

their origin in the top-left corner of the SVG (figure 3.4).

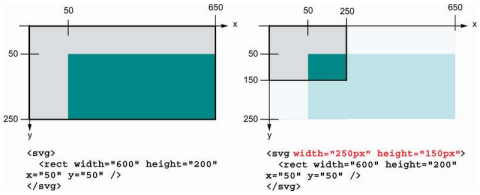

Figure 3.4 SVGs with and without a defined viewport

The viewport for our loader will be

```
<svg width="100%" height="300px"> <!--SVG code --> </svg>
```

The width is set at `100%`, but 100% of what? We're dictating that the loader will take 100% of the available space it's given by its parent item.

The following listing shows our starting HTML. We see that our loader is nested inside a section; therefore, our loader will be the same width as that section.

Listing 3.1 Starting HTML

```
<body>
  <section>
    <svg width="100%" height="300px"></svg>          ①
    <h1>Scanning channels</h1>
    <p>This may take a few minutes</p>
    <progress value="32" max="100">32%</progress>    ②
```

```
  </section>
</body>
```

① Loader with added viewport of 100% width by 300-pixel height

② The progress bar, which we address later in the chapter

We have some starting CSS as well (listing 3.2). The background (`<body>`), `<section>` , header (`<h1>`), and paragraph (`<p>`) have been prestyled to focus on the loader, progress bar, and animations.

Listing 3.2 Starting CSS

```
body { background: rgb(0 28 47); }
section {                                  ①
  display: flex;                           ②
  flex-direction: column;                  ②
  justify-content: space-between;          ②
  align-items: center;                     ②
  max-width: 800px;                        ②
  margin: 40px auto;                       ③
  font: 300 100% 'Roboto', sans-serif;     ④
  text-align: center;                      ④
  color: rgb(255 255 255);                 ⑤
}                                          ⑥
h1 {
  font-size: 4.5vw;
  margin: 40px 0 12px;
}

p {
  font-size: 2.8vw;
```

```
    margin-top: 0;
  }
```

① Start of rule styling the loader's container

② Layout using flexbox to set the child items in the column direction, centering horizontally and setting equal spaces between elements

③ Margin written using the shorthand property: top and bottom, 40px margin; left and right, auto

④ Typography setting the font weight to light, in the Roboto font, with a fallback of sans-serif and centering the text

⑤ Sets the color to white using RGB

⑥ End of rule styling the loader's container

We see that our section has its width capped at 800 pixels. `<section>` is a block-level element, so by default, it will take up the full width available to it. `<body>` and `<html>` are also block-level elements.

Because we don't specify a width, padding, or margin on

`<body>` or `<html>`, they will take the full width of the window. `<section>` will take the full width of the `<body>`. But because we assigned a maximum width to the `<section>`, when the window reaches 800 pixels wide, the section will stop growing with the `<body>` and remain 800 pixels wide. Because the section element has a top and bottom margin of `40px`, it will slightly increase the gap between the browser window and the element.

Our loader is contained within the section. The section will take the full width of the body until it reaches 800 pixels; therefore, our loader will do the same. Figure 3.5 shows how the width of the loader will be affected by the screen size.

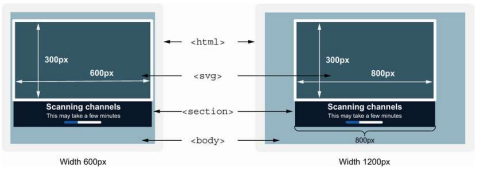

Figure 3.5 Window-width effect on SVG width when using `max-width`

With the viewport set, let's set the viewbox so that the contents of the SVG can scale with

its container. Remember that up to now, we've dealt only with the frame, not its innards.

3.2.3 Viewbox

The *viewbox* sets the position, height, and width of the graphic within the viewport. Earlier, we likened the viewport to a picture frame. The viewbox allows us to adjust the image to fit our frame. It can position the image and also scale the graphic so that it fits inside the frame. We can think of the viewbox as being our pan and zoom tools. To set the viewbox, we apply the `viewBox` attribute to the SVG with the following four values and syntax: `viewBox="min-x min-y width height"`. Listing 3.3 shows `viewBox` applied to our loader.

Dissecting the numbers in order, we start with `min-x` and `min-y`, both of which are set to `0`. We want the top-left corner of the graphic to be in the top-left corner of our frame. `min-x` and `min-y` allow us to adjust the position of the graphic in its frame; it's the pan tool. Because we want it to

be exactly in the top-left corner, we set the values to `0`.

Next, we apply the width, which is set to `710` because our loader has 11 total bars, each created with a width of `60`. 60 × 11 = 660, and we have 10 gaps. The gap width between each bar is 5 × 10 = 50; therefore, our loader's width will be `660 + 50 = 710`.

We'll base the height of the `viewBox` on the height of the bars in our loader. The bars have a height value of `300`, so we also set the viewport height to `300`. Our loader will fit exactly inside its viewport. The next listing shows the `viewBox` applied to the SVG.

Listing 3.3 Declaring the viewbox

```
<svg viewBox="0 0 710 300" width="100%" height="300px">
 <!--SVG code-->
</svg>
```

Notice that both our viewbox and viewport heights equal `300`. This is how we zoom. If the viewbox figures are less than the viewport figures, we're effectively zooming out

of the frame, and the graphic will be smaller. If the viewbox figures are more than the viewport figures, we're zooming in. Because we have equal viewport and viewbox heights, however, we're not zooming.

Now that we've defined the space we'll be working in, we can start adding shapes to the loader.

3.2.4 Shapes in SVG

There are a few standard SVG shapes and elements:

- `rect` (rectangle)
- `circle`
- `ellipse`
- `line`
- `polyline`
- `polygon`

If we want to create an irregular shape, we can also use `path`, but we won't need it for this loader. Most often, paths are what we see when we look at the XML behind logos, icons, and complex animation graphics. For our project, we'll use the basic rectangle shape to create the wave.

To define our rectangles, which will create the bars in

our loader, we'll use the `<rect>` element and add four properties: `height`, `width`, `x`, and `y`. The `x` and `y` attributes determine the position of the top-left corner of the rectangle relative to the top-left corner of the SVG.

We want to create 11 rectangles (listing 3.4) that have a width of `60` and a height of `300`, and we'll use the `x` attribute to move the rectangles across the graphic. We start at `0` and increase the value by the width of our bar (`60`) plus an additional gap of `5`. Each rectangle's `x` value will be `65` more than the previous one. Our 11th rectangle should have an `x` value of `650`.

Listing 3.4 Eleven rectangles

```
<svg viewBox="0 0 710 300" width="100%" height="300">
    <rect width="60" height="300" x="0" />
    <rect width="60" height="300" x="65" />
    <rect width="60" height="300" x="130" />
    <rect width="60" height="300" x="195"/>
    <rect width="60" height="300" x="260"/>
    <rect width="60" height="300" x="325"/>
    <rect width="60" height="300" x="390"/>
    <rect width="60" height="300" x="455"/>
    <rect width="60" height="300" x="520"/>
    <rect width="60" height="300" x="585"/>
    <rect width="60" height="300" x="650"/>
</svg>
```

Now we have our rectangles positioned inside our viewport, and they're resized correctly as we increase and decrease the window size by our `viewBox`. Figure 3.6 shows our SVG in different window sizes. (We added a white border to the SVG and bars to make them more visible in the screenshots.) The contents shrink and grow within their available space without skewing the rectangles that they contain as the width-to-height ratio changes with the resizing of the window.

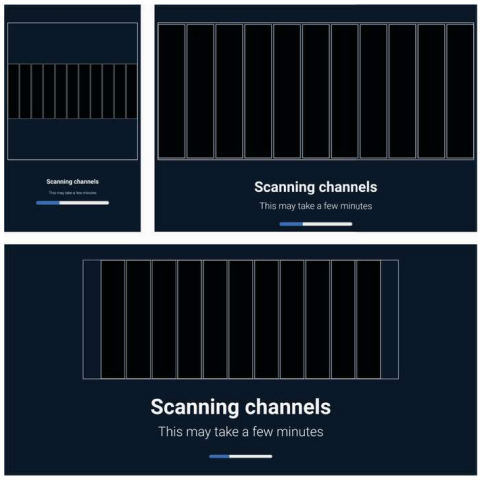

Figure 3.6 Adding 11 rectangles inside an SVG

Notice that our rectangles are black. Our next order of business is to style them.

.3 Applying styles to SVGs

We can apply styles to SVG elements in much the same way that we do in HTML: inline, internally in a `<style>` tag, or in a separate stylesheet. Some minor differences exist, however. First and foremost, how the SVG is imported into our HTML affects where the styles need to live to affect the elements.

The easiest way to add a vector graphic to a web page is to use an image tag. We reference the image file the same way that we would any other image: `<img src="myImage.svg" alt="">`. We can also add it as a `background-image` inside our CSS: `background-image: url("myImage.svg");`.

In both of these cases, our HTML and styles can affect the SVG but not the elements within it. We can affect the size of the image, for example, but we can't change the color of a particular shape inside the SVG. The image is essentially a black box that we can't penetrate to make changes. To manipulate elements within

the image, we'd have to place the styles inside the SVG itself.

Our third option—the one we'll use in this chapter—is to place the SVG's XML inline, directly in our HTML rather than in an external file, preventing the black-box issue we'd encounter if the code were in an external file. The drawback is that our concerns aren't as well separated because now our image code is mixed in with our HTML.

When our SVG is placed inline in our HTML, the standard ways to apply CSS to any other HTML element apply. Therefore, we can place the styles we want to apply to our SVG inside our CSS as though the SVG were any other HTML element.

SVG presentation attributes

In HTML, when we apply styles inline, we need to include a style attribute, such as `<p style="background: blue">`. SVGs, however, have styles that we can add directly to the element as attributes. These styles are called *presentation attributes*.

The `fill` attribute (the SVG equivalent of `background-color`), for example, can be applied directly to the element without a style tag: `<rect fill="blue">`. These properties don't have to be applied inline directly on the element. They can be added inside a style tag or stylesheet the same way that we apply any other CSS style: `rect { fill: blue: }`.

You can find a comprehensive list of SVG presentation attributes at **http://mng.bz/Alee**. Although the techniques for applying styles to our SVG elements remain the same as those for HTML (except for the aforementioned SVG presentation attributes when applied inline), some of the properties we'll use to style our elements will be different. Let's take a closer look at one we'll be using for this project.

To set the background color of the loader bars, instead of using `background-color`, we'll use the `fill` property, as the `background-color` property doesn't work for SVG elements. The `fill` property supports the same values as

`background-color`, such as color name, RGB(a), HSL(a) and hex. So instead of `rect { background-color: blue; }`, we'd write `rect { fill: blue; }`. If no `fill` value is assigned to a particular shape, the `fill` will default to black, which is why our rectangles are black.

Let's add a fill color to our rectangles. Because not all the rectangles are the same color (they have varying colors of blue and green to give the loader a bit of a gradient effect), rather than give each element a class, we'll use the pseudo-class `nth-of-child(n)`, which matches elements based on their positions within the parent. We'll look for the `n` th rectangle, to which we'll apply the fill. Therefore, `section rect:nth-of-type(3)` would find the third rectangle of the section container. Listing 3.5 shows the fill color applied to each of our rectangles.

NOTE A pseudo-class targets the state of an element—in this case, its position relative to the positions of its siblings.

Listing 3.5 Adding a fill color to our rectangles

```
rect:nth-child(1) { fill: #1a9f8c }
rect:nth-child(2) { fill: #1eab8d }
rect:nth-child(3) { fill: #20b38e }
rect:nth-child(4) { fill: #22b78d }
rect:nth-child(5) { fill: #22b88e }
rect:nth-child(6) { fill: #21b48d }
rect:nth-child(7) { fill: #1eaf8e }
rect:nth-child(8) { fill: #1ca48d }
rect:nth-child(9) { fill: #17968b }
rect:nth-child(10) { fill: #128688 }
rect:nth-child(11) { fill: #128688 }
```

Figure 3.7 shows our output. We can see that the bars in the loader are no longer black; color has been applied to them.

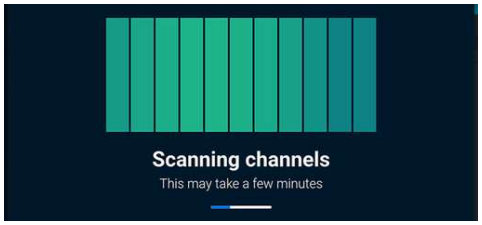

Figure 3.7 Fill applied to loader rectangles

The downside to our declarations is that if another SVG graphic had rectangles, our code could style the wrong graphic. To avoid this issue, we can add a class name to our SVG graphic as an identifier to specify the rectangle we want to style. But because we

have only one SVG in our project, we won't need to worry about that.

.4 Animating elements in CSS

The CSS Animation Module allows us to animate properties using keyframes, which we'll look at in section 3.4.1. We can control aspects of the animation such as how long it lasts and how many times it animates. CSS provides several properties we can use to define our animations' behavior, including the following:

- `animation-delay` —How long to wait before the animation starts
- `animation-direction` — Whether the animation is played forward or backward
- `animation-duratio` n— How long it should take for the animation to run once
- `animation-fill-mode` — How the element being animated should be styled when the animation is done executing
- `animation-iteration-count` —How many times the animation should run

- `animation-name` —Name of the keyframes being applied
- `animation-play-state` — Whether the animation is running or paused
- `animation-timing-function` —How the animation progresses through the styles over time

For our animation, we'll focus on four of these properties:

- `animation-name`
- `animation-duration`
- `animation-iteration-count`
- `animation-delay`

The effect we want to create is the rectangles shrinking and growing, but not in sync. At any given point in time, we want the heights of the elements to be slightly different. When the rectangles are shrinking and growing, we want the tops and bottoms of the rectangle to move toward the center and then expand back to full height. Essentially, we'll be creating a squeezing effect, going from large to small and back to large.

Although we'll apply the same animation to all the rectangles, to stagger their sizes we'll apply a slightly different delay to the start of the animation of each rectangle. As each rectangle starts animating at a different time, each one will be in a different stage of expanding and shrinking, creating a ripple effect.

First, we'll create the animation itself. Then, we'll apply it to the rectangles. Finally, we'll add the individual delays to stagger the size at any given point in time. To create the animation, we'll use keyframes. The `animation` property will reference the keyframes and dictate the duration, the delay, and how many times we want the animation to run.

3.4.1 Keyframe and animation-name

When we create a keyframe, we need to give it a name. The `animation-name` declaration value matches the keyframe name to join the two. With the `animation-name` property, we can list multiple animations separated by commas.

Origins of keyframes

Keyframes come to us from the animation and motion-picture industry. When companies used to do animation by hand, artists composed many individual pictures, with a change within each picture or frame. Over time, they made changes in each frame and gradually got to the end frame. A simple example of this technique is flipbook animation. The more frames you have and the more subtle the tweak is over a short period, the more fluid the animation is.

A keyframe represents the most important (key) changes in your animation (the frame). Then the browser works out the changes over time between defined frames. This process is known as *in-betweening*. Allowing the hardware to do the work, the browser can quickly fill the gaps between the keyframes, creating a smooth transition between one state and another. The in-betweening process is illustrated in figure 3.8.

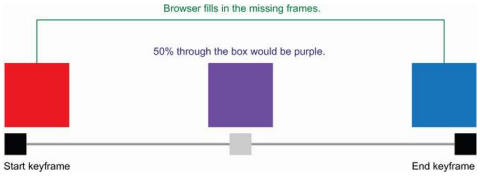

Figure 3.8 In-betweening

In CSS, keyframes are defined using an at-rule called `@keyframes`, which controls the steps within an animation sequence. *At-rules* are CSS statements that dictate how our styles should behave and/or when they should be applied. They begin with an at (`@`) symbol followed by an identifier (in our case, keyframes). We used an at-rule in chapter 2 to create our media query; here, we'll use one to create our keyframes. The syntax is `@keyframes animation-name { ... }`. The code inside the curly braces defines the animation's behavior. Each keyframe inside the `@keyframes` at-rule block is defined by a percentage (percentage of time passed in the animation) and the styles applied when we reach that point in time.

Before we dive into applying animations to our project, let's look at a simpler scenario to get a feel for the syntax (listing

3.6). You can also find this example on CodePen at https://codepen.io/michaelgearon/pen/oNyvbWX, where you can see the animation run (figure 3.9).

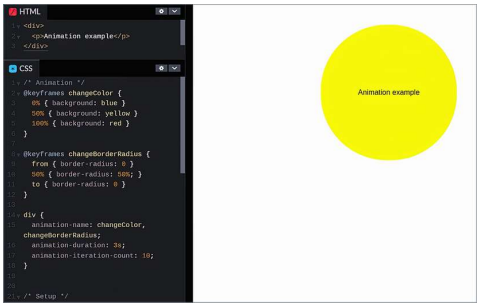

Figure 3.9 Simple animation scenario in CodePen

Listing 3.6 Example animation

```
@keyframes changeColor {                          ①
  0% { background: blue }                         ①
  50% { background: yellow }                      ①
  100% { background: red }                        ①
}                                                 ①
@keyframes changeBorderRadius {                   ②
  from { border-radius: 0 }                       ②
  50% { border-radius: 50% }                      ②
  to { border-radius: 0 }                         ②
}                                                 ②
div {
  animation-name: changeColor, changeBorderRadius;  ③
  animation-duration: 3s;                         ④
  animation-iteration-count: 10;                  ⑤
}
```

① First keyframe, named changeColor

② Second keyframe, named changeBorderRadius

③ The animation-name property referencing both of the animations

④ Sets how long the animation should take to complete

⑤ Sets how many times the animation should run

The example has two sets of keyframes: one named `changeColor` and one named `changeBorderRadius`. We apply both of the animations to a `div`. Then we define how long the animation should take to run (3 seconds) and how many times it should run (10 times). Inside each set of keyframes is code specifying what styles should be applied to the elements. So we have two different types of notation, we have keywords, and we have percentages. Let's break down what we're defining in the first set of keyframes.

We assert that when the animation begins (`0%`), we want to set the background color of the `<div>` to `blue`. By the time we reach `50%` of our animation (half of 3 seconds, or 1.5 seconds), our background

will be `yellow`. And when the animation ends (`100%`, or at 3 seconds), our background will be `red`. In between the keyframes. the color changes smoothly from one state to the next.

In the second set of keyframes, `changeBorderRadius`, instead of percentages we use the keywords `from` and `to`. `from` is the equivalent of `0%`, and `to` is equivalent to `100%`. We can mix the notation we want to use within the same set of keyframes.

When we apply the animation to the `div` ruleset, we also set a duration and iteration count. Notice that these two values are being applied to both of the animations.

Before we take a closer look at these two properties and how they work, let's create the animations for our loader. For our loader, we want to grow and shrink—or *scale*—our rectangles over time. Therefore, we'll call our keyframe `doScale`. Our at-rule will be `@keyframes doScale { }`.

Inside the at-rule, we define the keyframes for the animation. We'll start with the rectangle having its full height. Halfway through the animation, we want the height of the rectangle to be 20 percent of its original height. When the animation terminates, we want the rectangle's height to be back to full size. So we have three steps to define: `from` (or `0%`), `50%`, and `to` (or `100%`).

To change the size of the rectangle, we'll use the `transform` property, which allows us to change the appearance of an element (rotate, scale, distorted, move, and so on) without affecting the elements around it. If we were to reduce the height of an element by using the `height` property, the content below it would move up to fill the newly available space. With `transform`, the amount of space and the location of the element in terms of the page flow don't change—only the visible aspect. Using the same scenario, if we were to decrease the height of that same element using `transform`, the content below it wouldn't move up. We'd have a blank space.

To affect the element, the `transform` property takes a `transform()` function. We'll use `scaleY()`. (You can find a full list of available functions at **http://mng.bz/Zo1N**.)

The `scaleY()` function resizes an element vertically without affecting its width or squishing or stretching it. To define how much an element should be scrunched or stretched, we pass the function a percentage or a number value. The number value maps to the decimal value of its percentage equivalent; therefore, `scaleY(.5)` and `scaleY(50%)` achieve the same result, decreasing the element's height to 50% of its original value. Values above `100%` increase the size of the element, and values between `0%` and `100%` shrink it.

Negative values applied to `scaleY()` flip the element vertically, so `scaleY(-0.5)` would flip the element upside down and shrink its height by 50%. `scaleY(-1.5)` flips the element upside down and makes the height 1.5 times the original value.

For our loader bars, we want our rectangles to be full height at the beginning and the end of the animation, and 20% of the original height halfway through the animation. Our completed keyframe with transforms applied looks like the following listing.

Listing 3.7 Completed keyframe

```
@keyframes doScale {                    ①
  from { transform: scaleY(1) }         ②
  50% { transform: scaleY(0.2) }        ③
  to { transform: scaleY(1) }           ④
}                                       ⑤
rect { animation-name: doScale; }       ⑥
```

① Start of the doScale at-rule

② Starts the animation at full height

③ Halfway through the animation, the height should be 20% of the original value.

④ By the end of the animation, the rectangle returns to full height.

⑤ End of the doScale at-rule

⑥ Applies the animation to the rectangles

If we run the code, we notice that nothing has changed; our rectangles aren't growing and shrinking yet, even though we applied the keyframe to our rectangles. We still need to define the duration and iteration count. Let's dig into those properties a bit further.

3.4.2 The duration property

The `duration` property sets how long we want the animation to happen from start to finish. The duration can be set in seconds (`s`) or milliseconds (`ms`). The longer the duration, the more slowly the animation completes. With accessibility in mind, we want to consider users who are sensitive to motion (section 3.4) and choose a duration that is reasonable.

Animations, seizures, and flash rate

The World Wide Web Consortium (W3C) recommends that to prevent inducing seizures in photosensitive users, we need to make sure that our animations don't contain anything that flashes more than three times in any

1-second period (**http://mng.bz/RldR**).

A lot goes into choosing appropriate animation timing. An animation that's too fast can create changes that are imperceptible or cause seizures, depending on its nature. An animation that's too slow can make our application look laggy. Most microanimations are short and transitional; they animate the change of an element from one state to another, such as flipping an arrow from pointing up to pointing down. A generally accepted duration for this type of animation is around 250 milliseconds.

If the animation is larger or more complex, such as opening and closing a large panel or menu, we can increase the duration to around 500 milliseconds. A loader is a bit different, though. It's not a quick change in response to a user's action; it's a large visual element that the user will focus on for some time.

Most often when determining the "correct" timing for a loader, we use trial and error to find the speed that works

best with our graphic. For our project, we want to set the animation to happen over 2.2 seconds. To apply the amount of time the animation should take, we add the `animation-duration` property to our rectangles, as shown in the following listing.

Listing 3.8 Added animation duration

```
rect {
  animation-name: doScale;
  animation-duration: 2.2s;
}
```

When we run the code, our loader animates once and then never animates again unless we reload the browser window. We also notice that all the bars increase and decrease in size at the same time. First, let's make our loader continue to animate over time; then we'll stagger the animation across our rectangles so that they appear to be different heights.

3.4.3 The iteration-count property

To make our animation restart after it has completed, we use

the `iteration-count` property, which sets the number of times the animation should repeat. By default, its value is `1`. Because we haven't set a value yet, the browser assumes that we want the animation to run once and be done. We want our animation to repeat continuously, so we'll use the `infinite` keyword value.

By applying this value, we're declaring that the animation should keep playing forever. If we wanted to run a specific number of times, we'd use an integer value. After we add our iteration count, our code looks like the following listing.

Listing 3.9 Added animation iteration count

```
rect {
  animation-name: doScale;
  animation-duration: 2.2s;
  animation-iteration-count: infinite;
}
```

When we run the code, we see that all the rectangles grow and shrink in sync, starting from the top, and that the animation restarts after it completes. We still have some work to do to set the anima-

tion to start in the middle of the rectangle rather than at the top, as well as to stagger the animation between our elements. First, though, let's take a quick pause to look at the animation shorthand property.

3.4.4 The animation shorthand property

We currently have three declarations that define our animation: `animation-name`, `animation-duration`, and `animation-iteration-count`. We can simplify our code by combining all three declarations in the `animation` shorthand property, which allows us to define our animation's behavior with a single property. In this property, we can define the values for any of the properties listed in section 3.3. We don't need to provide values for all the properties. If properties aren't defined as part of the shorthand property or individually, they use their default values.

As mentioned earlier, we're defining three properties: `animation-name`, `animation-duration`, and `animation-it-`

eration-count . Refactored to use the animation shorthand property, our declaration looks like figure 3.10.

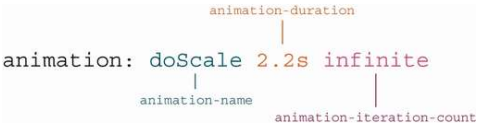

Figure 3.10 Breakdown of the animation shorthand property

This code is functionally identical to the code that currently applies to our rectangle. Using shorthand properties makes our code more concise and can make it easier to read. But if you find that writing out each property is easier for you, either method is perfectly valid. Do what works best for you.

When we use the animation shorthand property, our updated CSS looks like the following listing. After making the change to our code, we notice that our animation hasn't changed.

Listing 3.10 Refactoring to use the shorthand property

```
rect {
  animation-name: doScale;
  animation-duration: 2.2s;
  animation-iteration-count: infinite;
```

```
    animation: doScale 2.2s infinite;
  }
```

Next, let's address the staggering of heights for each of our rectangles.

3.4.5 The animation-delay property

The `animation-delay` property does what its name implies: it allows us to delay an animation on an element. The delay applies to the start of the animation. When the animation starts, it loops normally. As with the `duration` property, we can use seconds (`s`) or milliseconds (`ms`) to set the delay's duration value. The default value is `0` . By default, an animation doesn't have a delay.

To create the staggered effect in our animation, we'll assign different delay values to each of our rectangles, as shown in listing 3.11. The first rectangle's animation will start immediately. We give it a delay of `0` . We could omit this declaration, because `0` is the default value for `animation-delay` ; we added it here for clarity in explaining the code.

The second rectangle gets a `200ms` delay, and we continue to increment the delay by `200ms` for every rectangle thereafter. Notice that on the sixth rectangle, we switch to using seconds instead of milliseconds. We do this to make the code more readable because either second or millisecond values are acceptable.

Listing 3.11 Added animation iteration count

```
rect:nth-child(1) {
  fill: #1a9f8c;
  animation-delay: 0;
}
rect:nth-child(2) {
  fill: #1eab8d;
  animation-delay: 200ms;
}
rect:nth-child(3) {
  fill: #20b38e;
  animation-delay: 400ms;
}

rect:nth-child(4) {
  fill: #22b78d;
  animation-delay: 600ms;
}
rect:nth-child(5) {
  fill: #22b88e;
  animation-delay: 800ms;
}
rect:nth-child(6) {
  fill: #21b48d;
  animation-delay: 1s;
}
```

```
rect:nth-child(7) {
  fill: #1eaf8e;
  animation-delay: 1.2s;
}
rect:nth-child(8) {
  fill: #1ca48d;
  animation-delay: 1.4s;
}
rect:nth-child(9) {
  fill: #17968b;
  animation-delay: 1.6s;
}
rect:nth-child(10) {
  fill: #128688;
  animation-delay: 1.8s;
}
rect:nth-child(11) {
  fill: #128688;
  animation-delay: 2s;
}
```

After adding the delay, we see that we achieved our staggered effect (figure 3.11). But the elements are growing and shrinking from the top rather than from the center.

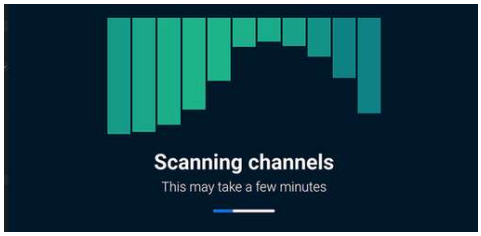

Figure 3.11 Animated rectangles with height change emanating from the top

To say where we want the element to grow and shrink from, we need to tell the browser where on the rectangle the an-

imation should originate. To address this problem, we'll use the `transform-origin` property.

3.4.6 The transform-origin property

The `transform-origin` property sets the origin, or point, for an element's transformations. If we were to rotate the object, the `transform-origin` property would set where on the element we want to rotate from. In our case, we'll use this property to set the position the animation should start from (the point of origin).

If the transform is happening in three dimensions (3D), the value can be up to three values (`x` , `y` , and `z`); if the transform is in two dimensions (2D), we can have up to two values (`x` and `y`). The first value is the horizontal position, or the x-axis; the second value is the vertical position, or the y-axis. When we're working in 3D, the third value would be forward and backward, or the z-axis.

We can declare the value of the `transform-origin` prop-

erty in three ways:

- Length
- Percentage
- Keywords
  - `top`
  - `right`
  - `bottom`
  - `left`
  - `center`

In HTML, the initial value for this property is `50% 50% 0`, which is `center`, `center`, `flat`. For SVG elements, however, the initial value is `0 0 0`, which places it in the top-left corner.

For our animation, we want the rectangle's transform origin to be at the center. We want the top and bottom of the rectangles to shrink rather than having the top fixed and the rectangles expanding and contracting from that point. To do this, we can either apply the keyword value `center` or assign a value of `50%` to the `transform-origin` property for our rectangles. Either way, we're saying that we want the point of origin to be the center of the rectangle. For our project, we'll use the keyword

value `center`. Listing 3.12 shows our updated `rect` rule.

We mentioned earlier that when working with 2D animations, the property takes two values, but we passed only one. When only one value is passed, it is applied to both the vertical and horizontal positions; therefore, `transform-origin: center;` is equivalent to `transform-origin: center center;`.

Listing 3.12 Updated `rect` rule with `transform-origin` property

```
rect {
  animation: doScale 2.2s infinite;
  transform-origin: center;
}
```

We've finished our loader animation (figure 3.12). But we still need to consider how accessible our design is. Section 3.4 dives into some ways we can provide a positive experience for all our users.

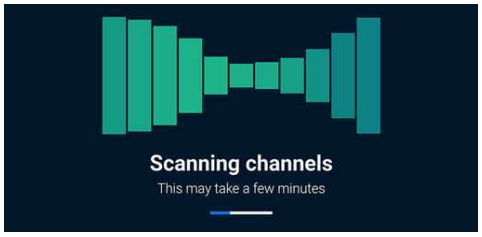

Figure 3.12 Finished loader animation

.5 Accessibility and the prefers-reduced-motion media query

The use of motion, parallax (an effect in which the background moves slower than the foreground), and animations on the web has increased as these effects have become easier to implement and browser support has improved. By using these techniques, we can create richer user interfaces that are interactive and provide richer user experiences.

The use of these techniques comes at a cost, however. For some users, especially those who have vestibular disorders, movement on the screen can cause headaches, dizziness, and nausea. As we mentioned earlier, animations can also cause seizures, especially if they contain elements that flash.

In many operating systems, users can disable animations on their devices. In our applications, we need to make sure that we respect those preferences. To check user settings the level-5 Media Queries Module has introduced the `prefers-reduced-motion` media query. This query is an at-rule, which checks the user's preferences regarding motion on the screen and allows us to apply conditional styles based on those preferences. The query has two values:

- `no-preference`
- `reduce`

We can choose to disable or reduce an animation when a user prefers reduced motion or enable it when they don't specify a preference. A user's preference for reduced motion doesn't mean that we can't use any animation, but we should be selective about which animations we keep. Aspects that may determine which animations to keep enabled include

- How fast it is
- How long it is
- How much of the viewport it uses

- What the flash rate is
- How essential it is to the functioning of the site or understanding of the content

TIP It's worth mentioning that a user may prefer reduced or no animation but may not be aware of the system-preferences settings for opting out of animations. Providing an on-site opt-out button may be useful, depending on how much animation our website has.

Accessibility guidelines for animations

A user should be able to pause, stop, or hide animation that lasts more than 3 seconds and isn't considered to be essential (**http://mng.bz/RldR**). Loaders are a bit tricky in this respect, as they convey important information to the user (the application is doing something and isn't frozen) but can be large and have a lot of motion.

Our loader could be considered to be essential content, but we also provide a progress bar below it to give the user an indication of what the application is doing. Because the

information is conveyed in a different medium, and because the animation is large, has a lot of movement, and could last more than 3 seconds, we're going to disable it for users who prefer reduced motion, using the code in the following listing.

Listing 3.13 Disabling the animation for users who prefer reduced motion

```
@media (prefers-reduced-motion: reduce) {      ①
  rect { animation: none; }                    ②
}
```

① Conditionally applies styles within the at-rule when the user enables prefer-reduced-motion

② Disables the animation previously applied to the rectangles

To check that we successfully disabled the animation, instead of editing our machine's settings, in most browsers we can do the following:

1. Go into our browser's developer tools.

2. In the console tab display, select the rendering tab. (In Google's Chrome browser, if this tab isn't already displayed, click the vertical ellipsis button and choose More Tools > Rendering from the drop-down menu.)
3. Enable the reduced-motion emulation.

Figure 3.13 shows the disabled animation and developer tools in the latest version of Chrome (**http://mng.bz/51rZ**) at this writing.

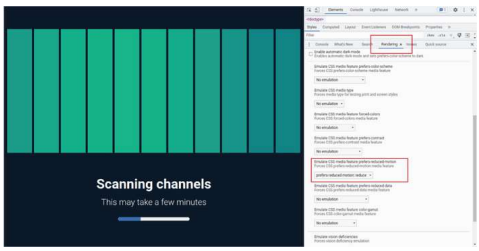

Figure 3.13 Emulating reduced-motion preference using Chrome DevTools

With our loader animation finished and accessibility needs handled, let's turn our attention to the progress bar at the bottom of the screen.

.6 Styling an HTML progress bar

The `<progress>` HTML element can be used to show that

something is loading or uploading, or that data has been transferred. It's often used to show the user how much of a task has been completed.

The default styles of the `< progress>` element vary among browsers and operating systems. Much of the functionality of the progress bar is handled at operating-system level; as a result, we have few properties available to restyle the control, especially when it comes to the colored progress indicator inside the bar itself. In this section, we'll look at some workarounds and their pitfalls. Let's start with an easy one.

Figure 3.14 shows our starting point generated by the HTML in listing 3.14. At this point, no styles have been applied to the control. The figure shows the defaults generated by Martine's machine.

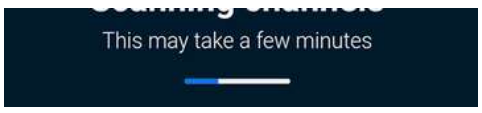

Figure 3.14 Progress bar starting point in Chrome

Listing 3.14 Progress bar HTML

```
<body>
  <section>
    ...
    <progress value="32" max="100">32%</progress>      ①
  </section>
```

① The progress bar

3.6.1 Styling the progress bar

Let's start with changing the height and the width. To increase the width of the progress bar to match the width of the section, we'll give its `width` property a value of `100%`. We also want to increase the height to `24px`.

To change the color of the progress indicator (the colored portion of the control), we can use a fairly new property: `accent-color`. This property allows us to change the color of form controls such as check marks, radio inputs, and the `progress` element. We'll set it to `#128688`, matching the color of the last bar of our loader. The following listing shows our progress rule thus far.

Listing 3.15 Progress rule

```
progress {
  height: 24px;
  width: 100%;
  accent-color: #128688;
}
```

Figure 3.15 shows the styles in listing 3.15 applied to our control.

Figure 3.15 Width, height, and accent color applied to the `progress` element

If we try to add a background color to our element (`background: pink`), we'll notice that the addition doesn't work. As a matter of fact, it fails spectacularly (figure 3.16). It radically changes the appearance of the element and alters the `accent-color` we previously set. Furthermore, the background color changes to gray rather than pink.

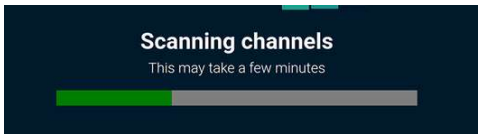

Figure 3.16 `background-color` failure

How do we get around this problem? To restyle the control, we need to ignore the de-

fault and re-create the styles from scratch. To do that, though, we need to use vendor-prefixed properties.

Vendor prefixes

Historically, when browsers introduced new properties, they were added with a vendor prefix before the property name. Each browser's prefix is based on the rendering engine that it uses. Table 3.1 displays major browsers and their prefixes.

Table 3.1 Vendor prefixes and their browsers

Prefix	Browsers
`-webkit-`	Chrome, Safari, Opera, most iOS browsers (including Firefox for iOS), Edge
`-moz-`	Firefox

Vendor prefixes are often incomplete or nonstandard implementations that browsers may choose to remove or

refactor at any time. Although this fact has been clearly documented for years, developers who were eager to use the latest properties regularly used them in production nonetheless.

To prevent this continued behavior, most major browsers moved to shipping experimental features behind a feature flag. To enable the feature and play with it, the user must go into their browser settings and enable that specific flag.

By moving to a flag-based method, the browsers are able to let developers play with experimental, cutting-edge features without fear that a non-standard implementation might be used in a piece of production code. But many vendor-prefixed properties are still available in the wild. For more information about vendor prefixing and feature flags, see the appendix.

The first thing we'll do to fix our `background-color` issue is to remove the default appearance of the control.

To reset the appearance of the `<progress>` element, we use the `appearance` property. By setting its value to `none`, we cancel the default styles provided by the user agent. Because we'll be creating all the styles from scratch, we can remove the `accent-color` property, as it will no longer have any effect.

We'll keep our height and width, and also add a `border-radius` because we're going to have a curved finish. The `appearance` property is supported by all new versions of major browsers, but we still need to include the vendor-prefixed versions, as some of the experimental properties we'll be using require them. The following listing shows our updated rule.

Listing 3.16 Updated progress rule

```
progress {
  height: 24px;
  width: 100%;
  border-radius: 20px;
  -webkit-appearance: none;
  -moz-appearance: none;
```

```
  appearance: none;
}
```

At this point, our progress bar looks the same as when we broke it by adding the background color. This result is to be expected. With `appearance:none` added, we can start altering the control in ways we previously couldn't. First, we'll focus on browsers with a `-webkit-` prefix.

3.6.2 Styling the progress bar for -webkit- browsers

We can use three vendor-prefixed pseudo-elements to edit the styles of our progress bar:

- `::-webkit-progress-inner-element` —The outermost part of the progress element
- `::-webkit-progress-bar` —The entire bar of the progress element, the portion below the progress indicator, and the child of the `::-webkit-progress-inner-element`
- `::-webkit-progress-value` —The progress indicator and the child of `::-webkit-progress-bar`

We'll use all three pseudo-elements to style our element. Let's start from the inside and work our way out. The first part we want to style is the progress indicator, for which we'll need to use `::-webkit-progress-value`. We curve the edges and change the color of the bar to a light blue, as shown in the following listing.

Listing 3.17 Styling the progress indicator in Chrome

```
::-webkit-progress-value {
  border-radius: 20px;
  background-color: #7be6e8;
}
```

Figure 3.17 shows our output in a WebKit browser.

Figure 3.17 Progress value styled in Chrome

Next, we'll edit the background behind the progress indicator by using `::-webkit-progress-bar`. We'll also add rounded corners to the background and change the color to a linear gradient, going from a dark green to a light

blue in keeping with the theme of the whole piece.

The `linear-gradient()` function takes a direction followed by a series of color and percentage pairs. The direction dictates the angle of the gradient; the color-percentage pairs dictate the points within the gradient at which we want to shift from one color to another. We'll use the keyword value `to right` as our direction. Then we'll set a starting color of `#128688` and an ending color of `#4db3ff`. Our gradient, therefore, will go from left to right, fading from our start color to our end color.

CSS gradient generators and vendor prefixes

As gradients can be tedious to write by hand, many CSS gradient generators have been created and are freely available on the web. Many still include vendor prefixes in their generated code. These prefixes are no longer necessary, as gradients are now supported by all major browsers, and browsers that required them are almost completely nonexistent now.

Finally, we add a border radius to the outermost container. The CSS for our progress bar is shown in the following listing.

Listing 3.18 Styling the progress indicator container in Chrome

```
::-webkit-progress-bar {
  border-radius: 20px;
  background: #4db3ff;                                    ①
  background: linear-gradient(to right, #128688 0%,#4db3ff 100%);
}
::-webkit-progress-inner-element {
  border-radius: 20px;
}
```

① Fallback color for the gradient

Our progress indicator looks great in Chrome (figure 3.18). Next, let's take a look at what it looks like in Firefox.

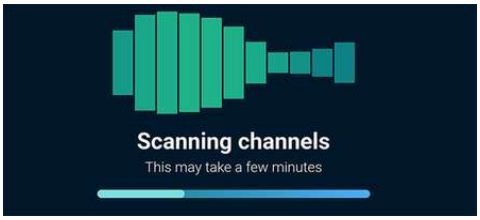

Figure 3.18 Styled progress indicator in Chrome

In Firefox (figure 3.19), we see that our control remains fairly unstyled because instead of

the `-webkit-` vendor prefix, it requires the `-moz-` prefix. Having written code for the `-webkit-` vendor prefix, we need to do the same for browsers that use the `-moz-` vendor prefix.

Figure 3.19 Unstyled progress bar in Firefox

3.6.3 Styling the progress bar for -moz- browsers

We'll approach the styles a bit differently for Firefox because we don't have as many properties to play with. The only `-moz-` prefixed property at our disposal is `::-moz-progress-bar` . Also a pseudo-element, it targets the progress indicator itself. Therefore, we'll style it the same way that we styled `::-webkit-progress-value` for Chrome because we want to achieve the same look in both browsers.

Because we're using the same styles, it's logical to add the `-moz-` selector to the existing rule: `::-moz-progress-bar, ::-webkit-progress-value`

`{ ... }`. It works well in Firefox (figure 3.20), but it will break Chrome (figure 3.21).

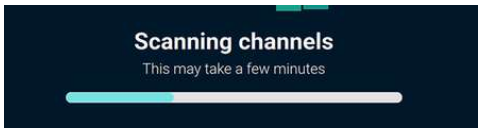

Figure 3.20 Firefox progress bar styled

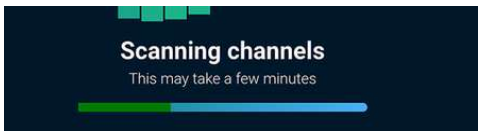

Figure 3.21 Adding both selectors in the same rule breaks Chrome.

Having multiple selectors in the same rule shouldn't cause this side effect, but we're dealing with experimental properties, which sometimes have nonstandard behaviors. To prevent this unfortunate side effect, we'll write two identical rules, one for each selector, as shown in the following listing.

Listing 3.19 Styling the progress indicator container in Chrome

```
::-webkit-progress-value {          ①
  border-radius: 20px;              ①
  background-color: #7be6e8;        ①
}
::-moz-progress-bar {               ②
  border-radius: 20px;              ②
```

```
    background-color: #7be6e8;     ②
  }
```

① Rule for Chrome

② Rule for Firefox

To change the background color for Firefox, we add a background property value to the progress element itself. We use the same gradient we used in the `::-webkit-progress-bar` rule. Figure 3.22 shows our progress in Firefox.

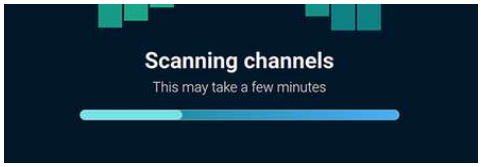

Figure 3.22 Firefox with background applied to the progress element

The last thing we need to do is remove the border, which we'll apply to the `progress` rule. To achieve this effect, we set the border property value to `none` . The following listing shows our final progress rule.

Listing 3.20 Final progress rule

```
progress {
  height: 24px;
  width: 100%;
  -webkit-appearance: none;
  -moz-appearance: none;
```

```
    appearance: none;
    border-radius: 20px;
    background: linear-gradient(to right, #128688 0%,#4db3ff 100%);     ①
    border: none;                                                       ②
  }
```

① Gradient background

② Removes the border

As we can see in figure 3.23, we've achieved the same result in Chrome and Firefox.

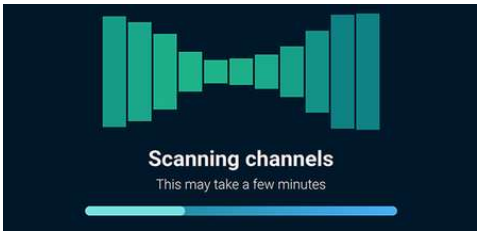

Figure 3.23 Progress bar styles finished in Firefox

We must stress that the styles were achieved by using experimental features that are non-standard and could change in the future. The value here is being able to experiment with new features before they become readily available. It's also an opportunity to get involved in the community; it's not uncommon for the working groups that develop browser features and specifications to request feedback before new standards are ac-

cepted and rolled out for general use.

ummary

- The `animation` property is a way to animate the values of the position, color, or some other visual element with CSS.
- The `@keyframes` at-rule is a way to define keyframes for your animations.
- We can delay the start of an animation by using the `animation-delay` property.
- The `animation-duration` sets how long a single iteration of the animation should take to complete.
- SVGs can be styled with CSS.
- The `prefers-reduced-motion` media query allows us to style animations conditionally per the user's settings.
- The HTML progress bar is a way to show how much of something has loaded.
- By default, the browser applies its own styling to the progress bar, but it can be reset by using the `appearance` property with a value of `none`.

- Our ability to style the `progress` element is fairly restricted unless we use experimental properties.
- Some nonstandard properties are available to style the `progress` element, but they require the use of vendor prefixes. Vendor-prefixed properties are experimental, which means that they sometimes have nonstandard implementations and could change at any time.