10 Styling forms

This chapter covers

- Styling input fields
- Styling radio buttons and check boxes
- Styling drop-down menus
- Considering accessibility
- Comparing `:focus` and `:focus-visible`
- Using the `:where` and `:is` pseudo-classes
- Working with the `accent-color` property

Forms are everywhere in our applications. Whether they're contact forms or login screens, whether or not they're core to an application's functionality, they're truly omnipresent. The design of a form, however, can easily make or break the user's experience. In this chapter, we'll style a form and look at some of the accessibility considerations we need to make sure to address. We'll look at some of the challenges that come with styling some radio and check-box inputs and drop-down menus, and we'll

cover some options for styling error messaging.

A *form* in this context is a section of code in an HTML `<form>` element containing controls (form fields) that the user interacts with to submit data to a website or application. Because contact forms are so prevalent across applications and websites, we'll use a contact form as the basis for our project.

0.1 Setting up

Our form contains two input fields, a drop-down menu, radio buttons, a check box, and a text area. We also have a header at the top and a Send button at the end of the form. Figure 10.1 shows our starting point—the raw HTML without any styles applied—and what we aim to accomplish.

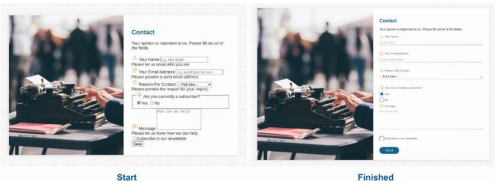

Figure 10.1 Starting point and finished product

Our starting HTML is fairly simple; it contains our form, inside which our labels, fields, error messages, and buttons are

placed. The starting and final code are on GitHub (**http://mng.bz/rWYZ**), on CodePen (**https://codepen.io/michaelgearon/pen/poeoNbj**), and in the following listing.

Listing 10.1 Starting HTML

```
<body>
  <main>
    <section class="image"></section>                                    ①
    <section class="contact-form">
      <h1>Contact</h1>
      <form>
        <p>Your opinion is important to us...</p>
        <label for="name">
          <img src="./img/name.svg" alt="" width="24" height="24">       ②
          Your Name                                                      ②
        </label>                                                         ②
        <input type="text"                                               ②
               id="name"                                                 ②
               name="name"                                               ②
               maxlength="250"                                           ②
               required                                                  ②
               aria-describedby="nameError"                              ②
               placeholder="e.g. Alex Smith"                             ②
        >                                                                ②
        <div class="error" id="nameError">                               ②
          <span role="alert">Please let us know who you are</span>       ②
        </div>                                                           ②

        <label for="email">                                              ③
          <img src="./img/email.svg" alt="" width="24" height="24">      ③
          Your Email Address                                             ③
        </label>                                                         ③
        <input type="email"                                              ③
               id="email"                                                ③
               name="email"                                              ③
               maxlength="250"                                           ③
               required                                                  ③
               aria-describedby="emailError"                             ③
               placeholder="e.g. asmith@email.com"                       ③
```

```
        >                                                              ③
        <div class="error" id="emailError">                            ③
          <span role="alert">Please provide a...</span>                ③
        </div>                                                         ③

        <label for="reasonForContact">                                 ④
          <img src="./img/reason.svg" alt="" width="24" height="24">   ④
          Reason For Contact                                           ④
        </label>                                                       ④
        <select id="reasonForContact"                                  ④
                required                                               ④
                aria-describedby="reasonError"                         ④
        >                                                              ④
          <option value="">-- Pick One --</option>                     ④
          <option value="sales"> Sales inquiry</option>                ④
             ...                                                       ④
        </select>                                                      ④
        <div class="error" id="reasonError">
          <span role="alert">Please provide the...</span>
        </div>

        <fieldset>                                                     ⑤
          <legend>                                                     ⑤
            <img src="./img/subscriber.svg" alt=""                     ⑤
  width="24" height="24">                                              ⑤
            Are you currently a subscriber?                            ⑤
          </legend>                                                    ⑤
          <label>                                                      ⑤
<input type="radio" value="1" name="subscriber"                        ⑤
       checked required>                                               ⑤
Yes                                                                    ⑤
          </label>                                                     ⑤
          <label>                                                      ⑤
            <input type="radio" value="0" name="subscriber" required>  ⑤
            No                                                         ⑤
          </label>                                                     ⑤
        </fieldset>                                                    ⑤
        <label for="message">                                          ⑥
          <img src="./img/message.svg" alt="" width="24" height="24">  ⑥
          Message                                                      ⑥
        </label>                                                       ⑥
        <textarea id="message"                                         ⑥
                  name="message"                                       ⑥
                  rows="5"                                             ⑥
```

```
                    required                                                    ⑥
                    maxlength="500"                                             ⑥
                    aria-describedby="messageError"                             ⑥
                    placeholder="How can we help?"                              ⑥
          ></textarea>                                                          ⑥
          <div class="error" id="messageError">                                 ⑥
            <span role="alert">Please let us know how we can help</span>        ⑥
          </div>                                                                ⑥

          <label>                                                               ⑦
            <input type="checkbox" name="subscribe">                            ⑦
            Subscribe to our newsletter                                         ⑦
          </label>                                                              ⑦

          <div class="actions">
            <button type="submit" onclick="send(event)">Send</button>
          </div>
        </form>
      </section>

    </main>

    <script src="./script.js"></script>                                         ⑧
  </body>
```

① Left image

② Name input with associated
label and error message

③ Email input with associated
label and error message

④ Reason for Contact drop-down
menu and associated label

⑤ Fieldset containing subscrip-
tion radio buttons

⑥ Message textarea

⑦ Subscription check mark

⑧ JavaScript that handles errors

You may have noticed that a JavaScript file is included. We'll use this file to show and hide errors later in the chapter (section 10.8).

So that we can focus specifically on styling form elements, the CSS to lay out the page is provided in the starting project. We use `grid` to place the image and form side by side. We also use a gradient to create the dots in the background. Our theme colors have been set up with CSS custom properties and some basic typography settings, including using a sans-serif font and changing the default text size for our project to `12pt`. The following listing shows our starting CSS.

Listing 10.2 Starting CSS

```
html {
  --color: #333333;                          ①
  --label-color: #6d6d6d;                    ①
  --placeholder-color: #abababab;            ①
  --font-family: sans-serif;                 ①
  --background: #fafafa;                     ①
  --background-card: #ffffff;                ①
  --primary: #e48b17;                        ①
  --accent: #086788;                         ①
  --accent-contrast: #ffffff;                ①
  --error: #dd1c1aff;                        ①
  --border: #ddd;                            ①
  --hover: #bee0eb;                          ①
```

```
  color: var(--color);
  font-family: var(--font-family);
  font-size: 12pt;
  margin: 0;
  padding: 0;
}

body {
  background-color: var(--background);                    ②
  background-image: radial-gradient(var(--accent) .75px,  ②
                    transparent .75px);                   ②
  background-size: 15px 15px;                             ②
  margin: 0;
  padding: 2rem;
}

main {
  display: grid;                                          ③
  grid-template-columns: 1fr 1fr;                         ③
  margin: 1rem auto;                                      ④
  max-width: 1200px;                                      ④
  box-shadow: -2px 2px 15px 0 var(--border);
}

.image {                                                  ⑤
  background-image: url("/img/illustration.jpeg");        ⑤
  background-size: cover;                                 ⑤
  background-position: bottom center;                     ⑤
  object-fit: contain;                                    ⑤
}                                                         ⑤

.contact-form {
  background-color: var(--background-card);
  padding: 2rem;
}

h1 { color: var(--accent); }
```

① Sets up our theme colors using custom properties

② Adds the polka-dotted background

③ Grid to place the two sections side by side

④ Prevents our design from getting too wide and centers it horizontally on the page

⑤ Adds the image to the left side

0.2 Resetting fieldset styles

Fieldsets are purpose-built to group controls and labels. Radio groups are a perfect use case for fieldsets, as they allow us to identify the controls effectively and explicitly as belonging together. They also give us a ready-built way of labeling the group of controls via the `<legend>`. Stylistically, however, we can agree that they're rather unsightly.

Let's reset the styles on the group to make it disappear visually. Programmatically, we want to keep the group, as it's helpful for users of assistive technology, but we're going to make it blend in a little more. To make the `<fieldset>` styles disappear, we need to reset three properties: `border`, `margin`, and `padding`. The following listing shows our rule.

Listing 10.3 Resetting fieldset styles

```
fieldset {
  border: 0;
  padding: 0;
  margin: 0;
}
```

With browser default styles on the `<fieldset>` removed (figure 10.2), let's turn our attention to our input fields.

Figure 10.2 Reset fieldset

0.3 Styling input fields

We have four types of input fields in our form, broken down as follows:

- *Your Name*— `text`
- *Your Email Address*— `email`
- *Yes/No*— `radio`
- *Subscribe to our newsletter*— `checkbox`

HTML has many more types of fields, including `date`, `time`, `number`, and `color`, each with its own semantic meaning and styling considerations. We chose the preceding four types because they're commonly used on the web today.

The unstyled appearance of these fields dictates what we'll do to style them. We'll treat the radio buttons and check box differently from the text input, for example, but we can reuse code across multiple types. We'll group them by how the unstyled controls look, so we'll handle the text and email together and then handle the radio buttons and check box together. Let's start with the text and email inputs.

10.3.1 Styling text and email inputs

The first thing we want to figure out is how to select only the text and email input fields—rather, all input fields that aren't a radio button or check box. One solution would be to add a class to each input we want to handle. This approach is hard to maintain and will get quite noisy, however, especially in a form-heavy application or complex form. Therefore, we'll use the pseudo-class `:not()` in conjunction with the type selector `selector[type="value"]`.

The `:not()` pseudo-class allows us to select elements that don't meet a particular criterion. In our case, we want to select all input fields that don't have a type

of `radio` or `checkbox`. Our selector, therefore, will be `input:not([type="radio"], [type="checkbox"])`. Now we can start styling the input fields, which currently look like figure 10.3.

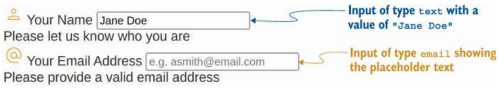

Figure 10.3 Input type `text` and type `email`

We see in figure 10.3 that the font is smaller than the `12pt` size we set on the body. Small font sizes are difficult to read on mobile devices; they're also hard to read for many users, especially young children and the elderly. If we want our form to be easily usable across a wide population and across devices, we'll need to increase it, so we'll set it to `1rem` to match the rest of our application. Inputs don't inherit font styles by default, so we'll also explicitly set `color` and `font-family` to `inherit`.

NOTE `inherit` is a handy property value. It allows an element to inherit a property value from the parent forcibly when inheritance doesn't happen by default.

Next, we're going to give the inputs some padding and custom

borders, as well as curve their corners. In this case, we'll make these changes for stylistic purposes. Most applications have a general style (look and feel). The styles we choose to apply to our fields should be in the same vein as the rest of our application's general theme to help the form blend with the page and look as though it belongs. From a marketing perspective, sticking with our theme also helps reinforce brand recognition.

To create the bottom border gradient effect, we'll use a linear gradient that goes from our primary color to our accent color. Because a gradient is an image we can't assign to the `border-bottom` property, we need to use `border-image`, which allows us to style our borders with images. We'll still provide a color in the `border-bottom` property as a fallback. Our code looks like the following listing.

Listing 10.4 Styling input fields that aren't of type `radio` or `checkbox`

```
input:not([type="radio"], [type="checkbox"]) {

  font-size: 1rem;
  font-family: inherit;
  color: inherit;
  border: none;
```

```
    border-bottom: solid 1px var(--primary);                          ②
    border-image: linear-gradient(to right, var(--primary), var(--accent)) 1; ③
    padding: 0 0 .25rem;
    width: 100%;
  }
```

① Removes all borders from the field

② Adds the border back in, but only on the bottom, with our primary color as a fallback color

③ Adds the gradient for our border

Pixels and rems

Notice that our border uses pixels whereas the rest of our declarations use rems. In some instances, we want some elements of our design to be relative to the text size. In other words, if the text size increased or decreased, we'd want those elements to scale accordingly. Our padding and margin in this case use rems because if the text size increases, we don't want the design to start looking cramped; on the flip side, if the text size decreases, we want to shrink that space accordingly. For these cases, we want to use a relative unit such as rems.

We want to keep the border at 1 pixel, however, regardless of the

text size. Therefore, we use a fixed unit.

We have some basic styles set for our text and email inputs, as shown in figure 10.4. We've started to develop a theme for our form controls.

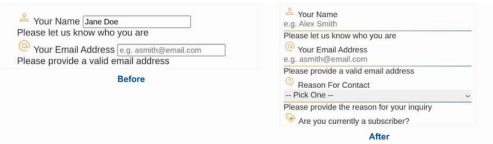

Figure 10.4 Text and email input styles

10.3.2 Making selects and textareas match the input styles

To make sure that the look and feel are consistent across our controls, let's apply the same styles we applied to the input field to the `<textarea>` and `<select>` elements. We're not going to create new rules or copy and paste the code. To keep our styles consistent and maintainable, we'll add `select` and `textarea` as selectors to our existing rules, as shown in the following listing.

Listing 10.5 Adding `textarea` and `select` to existing rule

```
input:not([type="radio"], [type="checkbox"]),
textarea,                                          ①
select {                                           ①
  font-size: 1rem;
```

```
    font-family: inherit;
    color: inherit;
    border: none;
    border-bottom: solid 1px var(--primary);                          ②
    border-image: linear-gradient(to right, var(--primary), var(--accent)) 1; ③
    padding: 0 0 .25rem;
    width: 100%;
  }
```

① Adds textarea and select to our rule

② Adds the border back in, but only on the bottom, with our primary color as a fallback color

③ Adds the gradient for our border

When the rule is applied, we notice that both fields still need a little bit of extra styling. Let's focus on the `<textarea>` first. Figure 10.5 shows our updated `<textarea>`.

Figure 10.5 Updated `<textarea>` styles

By default on the web, users can resize the width and height of `<textarea>`s by clicking and dragging the bottom-right corner. In our layout, increasing or decreasing the height doesn't cause any layout issues.

Changing the width, however, hides our image and eventually makes our form uncentered, as we can observe in figure 10.6.

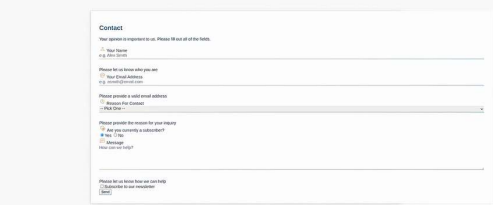

Figure 10.6 `<textarea>` resize issue

The `<textarea>` extends outside the container in an unsightly fashion. When we resize vertically, the container resizes appropriately, but this isn't the case horizontally. By changing the value of the `<textarea>`'s `resize` property from its default setting (`both`) to `vertical` , we limit users' ability to resize the element. Users will continue to be able to change its height but not the width, as shown in the following listing.

Listing 10.6 Updated styles for `textarea`

```
textarea { resize: vertical }
```

Visually, the text box looks the same and still has the resize control in the bottom-right corner (figure 10.7). When the user interacts with the resize control,

however, they'll be constrained to resizing vertically.

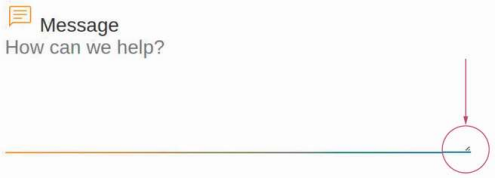

Figure 10.7 `<textarea>` vertical resize only

We still need to address the `<select>`, but this process will be a bit more complicated than editing the `<textarea>`. So let's finish styling the input fields first and then circle back to finish styling the `<select>` control.

10.3.3 Styling radio inputs and check boxes

Some form controls are notoriously difficult to style because the number of styles that can be applied to them are incredibly limited. Radio buttons and check boxes fall squarely into that category. Until recently, no properties whatsoever affected the radio-button circle or the check-box square. Our only option was to replace the native control styles with our own.

Why are some form fields so hard to style?

Some form fields, radio buttons and check boxes included, have

a reputation for being hard to style. This reputation stems from the limited number of CSS properties we have to alter how they look. The reason we have only limited properties is that the bulk of their appearance is driven by the operating system, not the browser.

Now we have the ability to change the native control's color. The `accent-color` property allows us to replace the user agent's chosen color with the color we specify. Applying `accent-color: var(--accent);` to our check box and radio buttons (listing 10.7) yields the results shown in figure 10.8.

Figure 10.8 Accent color applied to radio buttons and check boxes

Listing 10.7 Updated styles for `textarea`

```
input[type="radio"],                ①
input[type="checkbox"] {            ①
  accent-color: var(--accent);
}
```

① Styles are being applied only to inputs that have a type of radio or checkbox.

The elements have taken our set accent color instead of the light blue default color they used before. If we increase the `font-size` in the application, however, the controls don't increase in size (figure 10.9).

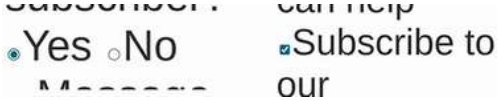

Figure 10.9 Increasing font size on radio buttons and check boxes

Although we can change the color of the element (which is an effective way to style the control quickly and efficiently to fit our styles better), if we want to allow a control to scale with our font size or make any further customizations, we'll need to replace the control's styles with our own. Because we want to keep the functionality of the control and replace only its visual aspect, our HTML stays the same. We're going to hide the native control provided by the browser and replace the visual portion with our own custom styles. To hide the native control, we'll use the `appearance` property and give it a value of `none`. This property allows us to control the native appearance of the control. By setting its property to `none`, we're saying that we don't want it to display the styles provided

by the operating system. We'll also set the `background-color` to our own background color (because some operating systems include a background for the controls) and then reset our margins.

We can remove the `accent-color` declaration we created earlier; we're re-creating the visual aspect of the control from scratch, so the declaration will have no effect. The following listing shows the completed reset.

Listing 10.8 Reset of `radio` and `checkbox` inputs

```
input[type="radio"],
input[type="checkbox"] {
  accent-color: var(--accent);
  appearance: none;
  background-color: var(--background);
  margin: 0;
}
```

Figure 10.10 shows that the radio buttons have disappeared. We can start creating our own styles for those controls.

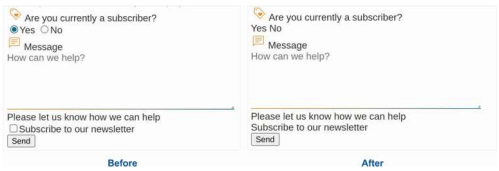

Figure 10.10 Reset radio and check box styles

To start, we want to create a box. For radio-button inputs, we'll give that box a `border-radius` to make it round. At the core, whether an input element is a check box or group of radio buttons, an input needs a box. We'll create one by giving the input a `height` and a `width` of `1.75em`. We use `em` units because they're a percentage of the parent's font size. By setting our height and width to `1.75em`, we're setting them to equal 1 3/4 times the value of the parent's font size. If our label—the container and therefore the parent of our input—has a `font-size` of `16px`, our box will be 28 pixels wide by 28 pixels tall (`16 x 1.75 = 28`).

Next, we'll add a border that inherits our label's font color. This step may sound a little weird: how are we going to make `border-color` inherit from `font-color`? We're going to use the keyword value `currentcolor`, which allows properties to inherit font color when they otherwise could not. We're going to set the border color to `currentcolor` to make the border color match the font color. To set our border width, we'll use `em` to allow the width of our border to scale with the size of our radio buttons.

Because inputs are inline elements by default, to apply our height and width, we'll also need to change the `display` property. We'll set it to `inline-grid` because when we handle the `checked` state for our inputs, we need to center the inner disk or check mark. Grid allows us to do so easily by means of the `place-content` property.

`inline-grid` is to `grid` as `inline-block` is to `block`. `inline-block` has all the same characteristics as `block` but places itself inline in the page flow. `inline-grid` works the same way. We have access to all the features of `grid`, but the element places itself inline in the page flow rather than below the previous content. For our purposes, this fact means that the input will place itself with the text label without our having to create special rules for labels containing radio-button inputs or check boxes.

Finally, we need to handle `border-radius`. This step is where the check box and the radio buttons diverge, because the check box is square and the radio buttons are circular. Because our fields have rounded edges, we're going to add a small `border-ra-`

dius (4px) to the check box. To make the radio buttons circular, we'll add a border-radius of 50% . Our updated rule is shown in the following listing.

Listing 10.9 Styled radio and checkbox inputs

```
input[type="radio"],
input[type="checkbox"] {
  appearance: none;
  background-color: var(--background);
  margin: 0;
  width: 1.75em;
  height: 1.75em;
  border: 1px solid currentcolor;      ①
  display: inline-grid;                ②
  place-content: center;               ②
}

input[type="radio"] { border-radius: 50% }

input[type="checkbox"] { border-radius: 4px }
```

① Sets the border to the same color as the parent element's text color

② Sets up to center the inner disk or check mark when the element is checked

Our unchecked inputs are styled. Now we need to address the styles to use when those inputs are selected. In figure 10.11, selected (checked) and unselected elements look identical.

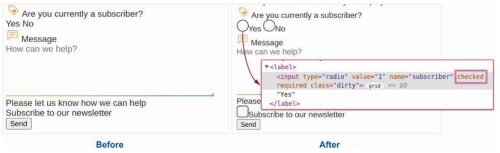

Figure 10.11 Unselected `radio` and `checkbox` styles

10.3.4 Using the :where() and :is() pseudo-classes

At this junction, we're going to look at two pseudo-classes that will help us keep our code clean and concise: `:is ()` and `:where ()`. Both pseudo-classes work similarly in that they take a list of selectors and apply the rule if any of the selectors within the list matches. Both are incredibly helpful for writing long lists of selectors. Instead of writing

```
input:focus, textarea:focus, select:focus, button:focus { ... }
```

we can use `:where` or `:is` and write an equivalent like so :

```
:where(input, textarea, select, button):focus { ... }
```

The `:is()` pseudo-class would be applied in the same manner. The difference between `:is ()` and `:where ()` is in their level of specificity. `:where()` is less specific and therefore easy to override. `:is()`, on the other hand, takes the specificity value

of the most specific selector in the list.

NOTE To see how specificity is calculated, check out chapter 1. We'll go into a bit more depth on calculating specificity with `:where()` and `:is()` in section 10.3.9.

WARNING Use caution in using `:is ()`, because if we have an `id` selector in our list of selectors (`id` selectors are most specific), we can create rules that are difficult to override.

We'll use `:where ()` and `:is ()` in conjunction with pseudo-classes such as `:checked`, `:hover`, and `:focus`, and with the `::before` pseudo-element to finish styling our `checkbox` and radio inputs.

10.3.5 Styling selected radio and checkbox inputs

To add the inner disk of the selected radio button and the check mark for the check box, we'll apply a method similar to the one we used for unselected inputs. We created some base styles that applied to both types of inputs and then added the finishing touches to each element individually when the styles diverged. As before, we'll start by

creating a box. Next, we'll place that box in the center of the existing styles, and then we'll shape it to be a disk or check mark.

To create this second box to be placed inside our current element, we'll use the `::before` pseudo-element. At this point, the `:where()` pseudo class (introduced in section 10.3.4) comes into play; we'll use it to select both of our input types and then add the `::before` pseudo-elements. Our selector will look like this: `:where(input[type= "radio"], input[type="checkbox"])::before { }`.

Our content will be empty, so we'll use a `content` property value of `""` (empty quotes), and we'll give it a `display` value of `block` so that we can assign a `width` and a `height`.

When we created the outer box earlier, we gave it a height and width of `1.75em`. We used an `em` unit so that control would scale relative to the text size. We'll do the same thing here. We want the inner disks and check mark to be smaller than their containers, so we'll set the `height` and `width` to `1em`.

Assuming that the `font-size` applied to the input is `16px`, our box will be `16px` by `16px` (`16 x 1 = 16`).

We don't need to do anything to position our inner box. Remember that earlier, we set the input display to `inline-grid` and then added the `place-content` property with a value of `center` in listing 10.8. The grid layout automatically places the inner box in the center of the input. The CSS for our inner disk and check mark looks like the following listing.

Listing 10.10 Centering the inner box

```
:where(input[type="radio"], input[type="checkbox"])::before {
  display: block;
  content: '';
  width: 1em;
  height: 1em;
}
```

When we apply this code, we see no changes, as demonstrated in figure 10.12. Our inner box does exist but isn't visible yet.

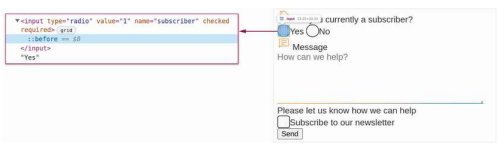

Figure 10.12 Invisible inner box

The box isn't visible because it doesn't have any content or background color. We'll add a background color next.

10.3.6 Using the :checked pseudo-class

We're not going to apply the same background color to our element all the time. We're going to use our accent color when the element is selected and our hover color when the element is being hovered over.

The `:checked` pseudo-class selector can be used on an input of type `radio` or `checkbox`, or on the option element (`<option>`) in a drop-down menu (`<select>`) to apply styles when the element is selected. The ability to use it on `<option>` is browser-dependent.

When we apply the `background-color` for the `checked` and `hover` states, if the selectors have the same level of specificity (as our example will), the order in which we write these rules matters. If we write the checked state rule first and the hover state rule second, the hover color will be applied to a selected input on hover; the hover state rule will override the checked state rule because it appears

later in the CSS file. Therefore, we want to make sure that the hover state rule is placed before the checked state rule in the CSS file. Figure 10.13 illustrates these two scenarios.

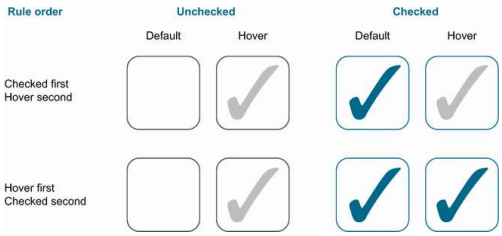

Figure 10.13 Rule order regarding the background for a selected check box on hover

Let's see how we'd go about applying our background colors in our CSS file. The following listing shows our `hover` and `checked` code so far.

Listing 10.11 Inner element background color

```
:where(input[type="radio"], input[type="checkbox"]):hover::before {      ①
  background: var(--hover);                                              ①
}                                                                        ①
:where(input[type="radio"], input[type="checkbox"]):checked::before {    ②
  background: var(--accent);                                             ②
}                                                                        ②
```

① Adds a background color to the inner box on hover

② Adds a background color to the inner box when input is selected

Figure 10.14 shows that we have a box we can shape inside our elements. The box is displayed in our accent color when the element is selected, and when a user hovers over unselected radio-button or check-box inputs, we see a gray box.

Figure 10.14 Setting up for the selected state

Next, we need to shape the inner box, where our code will diverge to create disks and a check mark for the radio buttons and check box, respectively.

10.3.7 Shaping the selected radio buttons' inner disk

Starting with the radio-button inputs, we turn our inner box into a circle by adding a `border-radius` of `50%`, as shown in listing 10.12. We don't differentiate between the `hover` and `checked` states because we want the shape to be a disk regardless of the state of the element.

Listing 10.12 Radio-button inner disk

```
input[type="radio"]::before {
  border-radius: 50%;
}
```

Now we have traditional-looking radio buttons that scale nicely regardless of text size (figure 10.15). With our radio buttons styled, we'll turn our attention to shaping the check mark inside our check box.

Figure 10.15 Styled `radio` inputs

10.3.8 Using CSS shapes to create the check mark

Shaping our radio inputs was simple: we used `border-radius` to achieve a disk shape. Creating a check mark isn't quite as simple. To do that, we'll use `clip-path` .

NOTE `clip-path` allows us to create shapes by creating a clipping region that defines which parts of the element should be displayed and which parts should be hidden. We used `clip-path` in chapter 7.

The shape we'll apply to the `clip-path` to create our check

mark is a polygon. Polygons are created by setting a series of `X` and `Y` percentage-based coordinates between which a line is created. The `(0,0)` coordinate is the top-left corner of the shape. If the shape isn't explicitly closed, it automatically joins the first and last points. Our `polygon()` function will be `polygon(14% 44%, 0% 65%, 50% 100%, 100% 16%, 80% 0%, 43% 62%)`. Figure 10.16 explains the point-by-point construction of the shape.

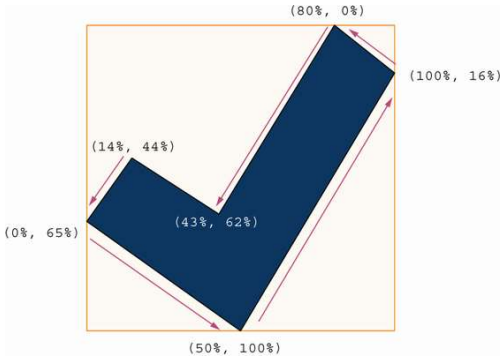

Figure 10.16 Polygon check-mark shape coordinates

NOTE The coordinates for simple shapes are easy enough to figure out. But as shapes get more complex, determining the coordinates manually can be cumbersome. In those situations, we can turn to vector-graphic drawing programs such as Inkscape and Illustrator, or to one of the many CSS shape-generator websites, including https://bennettfeely.com/clippy.

With our shape created, we can create our `clip-path` and apply it to the inner portion of our check box, as shown in the following listing.

Listing 10.13 Check mark in our check box

```
input[type="checkbox"]::before {
  clip-path: polygon(14% 44%, 0% 65%, 50% 100%, 100% 16%, 80% 0%, 43% 62%);
}
```

With the `clip-path` added, we have a fully functional check box. Next, let's add some finishing touches. Notice in figure 10.17 that the outlines of the selected radio buttons and check box are still in our font color rather than the accent color.

Figure 10.17 Styled check mark in the check box

To add the outline color to both the radio buttons and the check box when they're selected, we're going to use the `:checked` pseudo-class again to change the border color to our accent color only when the control is selected. This procedure translates to the code shown in listing 10.14. We

use `:is()` instead of `:where()` for reasons of specificity.

Listing 10.14 Accent-color outline for selected inputs

```
:is(input[type="radio"], input[type="checkbox"]):checked {
  border-color: var(--accent);
}
```

10.3.9 Calculating specificity with :is() and :where()

We mentioned earlier that `:where()` has a specificity of `0`, meaning that it's the least specific selector available to us. We set our default border color in the selector `input[type="radio"], input[type="checkbox"] { ... }`, which has a specificity of `11`, calculated according to table 10.1. In each column, we count the number of each type of selector, with columns A, B, and C forming the specificity value.[1]

Table 10.1 Calculating specificity

Selector	A ID selectors (×100)	B Class selectors, attribute selectors, & pseudo-classes (×10)	C Type selectors, pseudo-elements (×1)	Specificity
`:where(input[type="radio"], input[type="checkbox"])`	Ignores specificity rules and always equals 0			`000`
`:where(input[type="radio"], input[type="checkbox"]):checked`	Ignores specificity rules and always equals 0			`000`
`input[type="radio"]`	`0`	`1`	`1`	`011`
`input[type="radio"]:checked`	`0`	`2`	`1`	`021`
`:is(input[type="radio"], input[type="checkbox"]):checked`	`0`	`2`	`1`	`021`

Because `:is()` bases its specificity value on the value of the most specific selector within it, in this case the specificity will be `11` plus another `10` for the `:checked` state, giving us a specificity of `21`. Because `21` is greater than `0`, we override the styles, and our border becomes our accent color.

Now our radio buttons and check box are styled both when they're selected and unselected, and on hover for both states. Figure 10.18 shows our progress so far.

Let's turn our attention to the drop-down menu next.

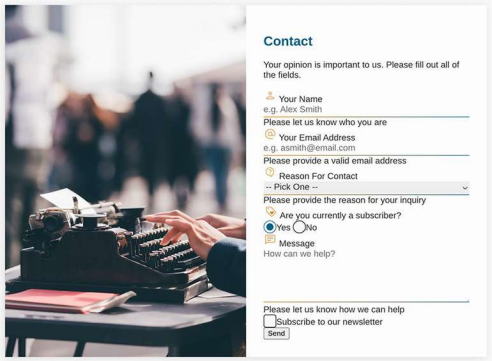

Figure 10.18 Styled check box and radio-button inputs

0.4 Styling drop-down menus

Although we applied the same default styles to `<select>` elements as we did for the text-based `<input>`s and `<textarea>`s (listing 10.5), we see in figure 10.19 that the drop-down menu (`<select>`) is still rough. We also see in the expanded view that our options list doesn't match our theme.

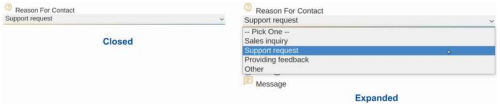

Figure 10.19 Drop-down menu closed and expanded

Let's start by fixing the background color. Although it's not obvious because the background behind our form is white, the input fields have a white background by default. We're going to add a rule to the existing declaration that affects the `<input>`s, `<textarea>`, and `<select>` elements to set the background color to the card background (listing 10.15). That way, should the card background change, our form controls will have the appropriate background color.

Listing 10.15 Default styles applied to `select`

```
input:not([type="radio"], [type="checkbox"]),
textarea,
select {
  font-size: 1rem;
  font-family: inherit;
  color: inherit;
  border: none;
  border-bottom: solid 1px var(--primary);
  border-image: linear-gradient(to right, var(--primary), var(--accent)) 1;
  padding: 0 0 .25rem;
  margin-bottom: 2rem;
  width: 100%;
  background-color: var(--background-card);      ①
}
```

① Adds background-color declaration

With the background color added, we see that the input and options have a white background (figure 10.20).

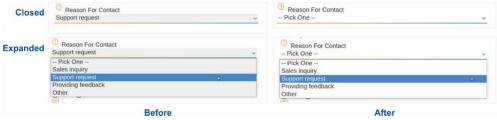

Figure 10.20 `select` element styled

Although it would be nice to update the drop-down menu options to match our theme better, these menus, like the radio inputs and check boxes, get a lot of their styles and functionality from the operating system itself. Therefore, we're limited in what we can style with CSS alone, and for this design, these changes are about as far as we can go. We can use JavaScript and ARIA to replace the entire control, but because this book is about CSS, we're going to style as much as we can with CSS alone.

What is ARIA?

ARIA (which stands for Accessible Rich Internet Applications) is a set of roles and attributes that can be added to HTML elements to supplement missing information about the

use, state, and functionality of an element that otherwise isn't available to the user. For more information, check out https://www.w3.org/WAI/standards-guidelines/aria.

NOTE When creating a custom control, it's important to be mindful of the underlying accessibility information and functionality that the browser provides automatically and to make sure we're re-creating that functionality along with the visual aspects of the control. Libraries or frameworks can be helpful when a custom control is needed, assuming that the library or framework was built with accessibility in mind. Usually, the best place to find out is the documentation.

0.5 Styling labels and legends

To style our labels and the legend, we're going to start by giving them a vertical margin for breathing room between the label and the control. We'll also use Flexbox to align the text and the icons, radio inputs, and check box. Finally, we'll decrease their font size and change their color. Most important here are the values entered by the user, not the labels. By decreasing their size, we diminish their im-

portance in the visual hierarchy. We end up with the code displayed in the following listing.

Listing 10.16 Added margin and updated font size

```
label, legend {
  display: flex;           ①
  align-items: center;     ①
  gap: .25rem;             ①
  margin: 0 0 .5rem 0;
  font-size: .875rem;
  color: var(--label-color);
}
```

① Aligns the label text and the icon

With our labels and legend styled (figure 10.21), let's turn our attention to the placeholders.

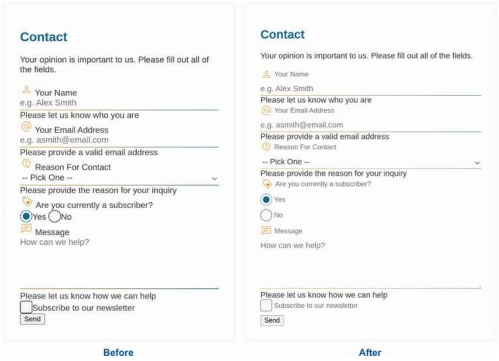

Figure 10.21 Styled labels and legend

0.6 Styling the placeholder text

In our form, it's difficult to distinguish what fields are user

filled from what is placeholder text. As we did for our labels, we're going to deemphasize the placeholder text to make it easier to distinguish from user responses.

Labels and placeholder text

Our project has both labels and placeholder text. Although placeholder text can be helpful to guide the user, it doesn't replace labels. In fact, the Web Content Accessibility Guidelines (WCAG) accessibility standards specifically require form fields to have a label (**http://mng.bz/mVzW**).

Placeholder text disappears after the user enters a value in the field. This arrangement is problematic because the user doesn't have a way to reference the instructions after they enter a value.

Furthermore, labels are required for assistive technologies such as screen readers, which rely on this information to indicate to the user what is expected in the field.

To style our placeholder text, we're going to use the `::placeholder` pseudo-element. Because we want the placeholder to be styled the same way

regardless of the type, we'll write one rule that targets all placeholder text regardless of element type. In this new rule, we'll decrease the size of the placeholder text and lighten its color, as shown in the following listing.

Listing 10.17 Styling the placeholder text

```
::placeholder {                          ①
  color: var(--placeholder-color);
  font-size: .75em;
}
```

① Targets any placeholder text regardless of element type

Figure 10.22 shows our updated fields.

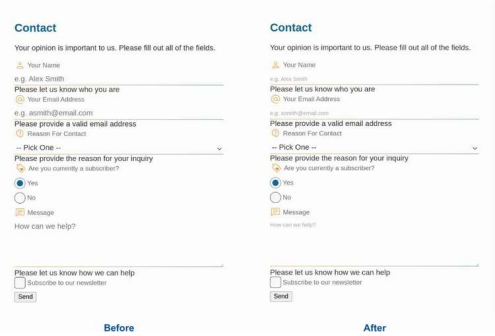

Figure 10.22 Styled placeholder text

Next, let's style the button at the bottom of the form.

0.7 Styling the Send button

We have a Send button at the bottom of our form. Let's make it

a bit more prominent and make it match the rest of our form. We'll create a rule that targets this button.

Next, we'll remove the border, curve the corners, and edit the text and background colors. In the "before" part of figure 10.23, the button text is smaller than our default font size, so we also change `font-size` to `1rem`. Finally, we set our button padding.

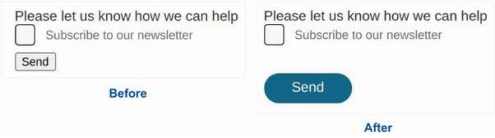

Figure 10.23 Styled Send button

To make our button stand out even more, we'll separate it a little bit from the rest of our fields. The button is located inside a `<div>` with a class of `actions`. We'll give this `<div>` a top margin of `2rem`, which will move the button down a little farther from the Subscribe check box. The following listing shows our new rules, and figure 10.23 shows our progress.

Listing 10.18 Resetting button styles

```
button[type="submit"] {
  border: none;
```

```
    border-radius: 36px;
    background: var(--accent);
    color: var(--accent-contrast);
    font-size: 1rem;
    cursor: pointer;
    padding: .5rem 2rem;
  }

  .actions { margin-top: 2rem }
```

Next, let's style the error messages.

0.8 Error handling

Below the Name, Email, and Message controls are error messages. Currently, they're unstyled, so they aren't easy to identify as error messages or to match them with the fields that the errors describe. Furthermore, we don't want to show this error message until the user has interacted with the control. Nobody wants an error message yelling at them before they've even started.

We're going to style the error messages to look like error messages; then we'll hide them by default and show them only when appropriate. This task is where our JavaScript file comes into play.

We're going to make our text red, like most error messages on the

web, by setting that color in our `--error` custom property. We'll also make the text bold and preface our error with an error icon to present it clearly as such; we don't want to use color alone to convey meaning or intent.

NOTE Color is a great way to differentiate content types. But we should always use something else with it—such as an icon; text; or a change in size, weight, or shape—because people who are color-blind may not be able to differentiate between colors. Furthermore, some colors don't have the same meaning across cultures. For reasons of accessibility and clarity, it's best practice to use more than color alone to convey a message.

So that we can keep our error icon consistent instead of adding it before each error, we'll add it programmatically via CSS, using the `::before` pseudo-element. To size and position the icon, we'll use two relative units: the character unit (`ch`), which we used in chapter 7 and which is based on the font's width; and `ex` , which is relative to the font's X-height, which is the distance between the baseline and mean-line of a font (figure 10.24). We use these particular units be-

cause they're relative to not only the font size, but also the characteristics of the typeface being used. Using `ch` and `ex` units helps make the size and spacing between the icon and the text seem like an extension of the font that's being used.

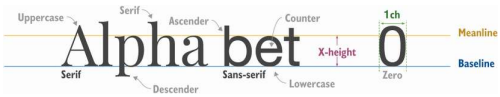

Figure 10.24 A visual representation of typography terms

We'll also add some margin to our error `<div>` to give our input fields some breathing room. Our rules to style errors look like the following listing.

Listing 10.19 Error styles

```
.error {
  color: var(--error);          ①
  margin: .25rem 0 2rem;
}
.error span::before {
  content: url('./img/error.svg');
  display: inline-block;
  width: 1.25ex;                ②
  height: 1.25ex;               ②
  vertical-align: baseline;     ③
  margin-right: .5ch;
}
```

① Makes the text red

② Makes the icon 1.25ex by 1.25ex

③ Aligns the icon to the text's baseline

Notice that when we added the icon before the text, we added it to the `span`, not the error `<div>` itself, because we're going to be showing and hiding the `span` inside the error and the entire error `<div>`. Let's take a closer look at the HTML to understand why.

Listing 10.20 shows the complete control for the Name field, including its label and error message. Notice that the error `<div>` has an `id` of `nameError`, which is referenced by the `aria-describedby` attribute on the input field. The `aria-describedby` attribute tells screen readers and assistive technologies that the element whose `id` it references contains extra information pertaining to the input field.

If we hide the error `<div>` in its entirety by using `display:none`, the element to which the `aria-describedby` is pointing won't exist. Therefore, we hide only the contents (the `span`) so as not to break the programmatic connection between the element and its error. Because we'll be hiding only the `span`, we need to

apply the icon to the `span` so that it can be hidden when we hide the error message.

Listing 10.20 Name-field HTML

```
<label for="name">Your Name</label>
<input type="text" id="name" name="name" maxlength="250" required
    aria-describedby="nameError">      ①
<div class="error" id="nameError">     ②
  <span role="alert">
    Please let us know who you are
  </span>
</div>
```

① Indicates which provides extra information about the input (referenced by id)

② The ID referenced by the aria-describedby attribute

Figure 10.25 shows our styled error messages.

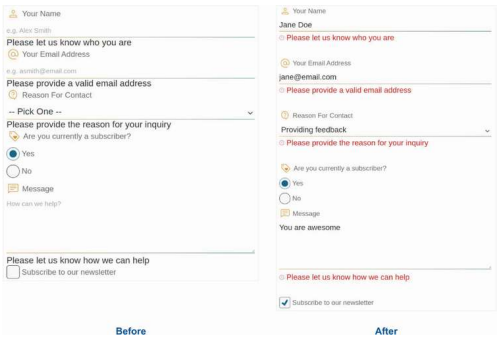

Figure 10.25 Styled error messages

With our error messages styled, we can handle showing them only when appropriate. In figure 10.25, we see that the inputs have valid values, yet the error

messages still appear. To show the error message only when the field is invalid, we'll start by hiding the error message by default. We apply a `display` property value of `none` to the `span` contained in the error `<div>`; then we use the `:invalid` pseudo-class to show it conditionally (only when the field is invalid).

The validity of the field in this case is determined by the properties we set on the field itself. Let's look at the Name input HTML again: `<input type="text" id="name" name="name" maxlength="250" required aria-describedby="nameError">`. We included `required` and `maxlength` attributes; therefore, if there's no value in the field or if the value's length is greater than 250 characters, the field value will be invalid, and styles in the `:invalid` pseudo-class will be applied.

The Email element (`<input type="email" id="email" name="email" maxlength="250" required aria-describedby="emailError">`) also has a `maxlength` and a `required` attribute, so it would be invalid under the same condi-

tions as the Name field. It also has a type of `email`. In HTML, some field types have validation built in, and `email` is one of them. If we were to enter an email address value of `"myE-mail"`, it would be invalid.

Using the `:invalid` pseudo-class helps us prevent errors from being displayed when the field is valid, but it doesn't prevent errors from showing up if the user hasn't interacted with the field yet. We could use the `:user-invalid` pseudo-class instead of `:invalid`, which would trigger one time and only after the user interacted with the field, but at this writing, Mozilla Firefox is the only browser that supports this property. So we turn to JavaScript due to the current lack of cross-browser support. In the future, when the `:user-invalid` property is better supported, we'll no longer need to use JavaScript to show/hide our error messages based on user interaction. The script included in the project listens for blur events, which happen when an element loses focus. When we click or tab away from a field, a blur event occurs. Our script listens for these events and adds a class of `dirty` to the field that we've navigated

away from, letting us know which fields have been interacted with and which haven't. Those with a class of `dirty` have; without a class of `dirty` have not.

Because we have this `dirty` class, in conjunction with the `:invalid` pseudo-class, we'll show the error message only below controls that are invalid and that the user has touched, preventing us from showing error messages before the user has had a chance to fill out the form. We use the selector `.dirty:invalid + .error span`. We select the `span` contained in an element that has a class of `error` located immediately after an element that is both invalid and has a class of `dirty`.

Last, we'll change the border color of the field to our error color when it's both invalid and `dirty`. Because we used a border image to create the gradient effect, we need to remove it. The following listing shows the full rules for showing and hiding the error messages.

Listing 10.21 Error-handling CSS

```
.error span { display: none; }                ①

.dirty:invalid + .error span {                ②
  display: inline;                            ②
}                                             ②

:is(input, textarea).dirty:invalid {          ③
  border-color: var(--error);                 ③
  border-image: none;
}                                             ③
```

① Hides the error message by default

② Shows the error message when the field immediately before it in the HTML is dirty and invalid

③ Changes input and textarea border color to red when invalid and dirty

Figure 10.26 shows fields in their three possible states: invalid and dirty, valid, and invalid but not yet touched.

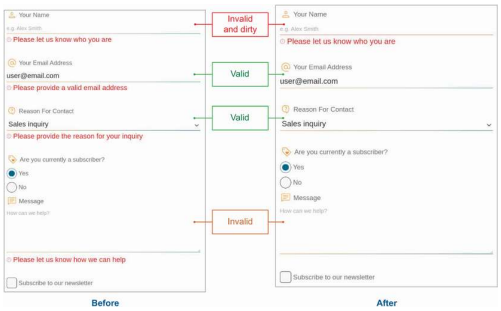

Figure 10.26 Error-handling and field states

On the surface, our form seems to be finished, but we still have

some finishing touches to add.

0.9 Adding hover and focus styles to form elements

Because we want our form to be accessible, we need to make sure to include hover styles and to update the default focus styles to match our theme for our controls and buttons. We've already handled the hover styles for radio buttons and check boxes but not the focus. For the other elements, we haven't considered the hover and focus states.

Let's start with focus because we still need to apply it to everything on our form. Focus is important for users who navigate the web via the keyboard rather than clicking elements with a mouse. It gives the user a visual indicator of which element currently has focus. Therefore, if we don't like the default focus styles, it's fine to restyle them but not remove them.

10.9.1 Using :focus versus :focus-visible

Because showing the focus styles all the time regardless of how the user is navigating the web can be overwhelming depending

on the design, a new property was recently added to the CSS specification to apply focus styles based on the user's modality: keyboard or mouse. The pseudo-class : `focus-visible` allows us to add styles when the user is interacting with the keyboard but won't apply it when the user is using a mouse. By contrast, `:focus` always applies regardless of the user's method of interacting with the element.

For our text and email input fields, drop-down menu, and text area, we'll remove the default outline and change the border's color from the gradient to a solid color. Because (as we mentioned earlier in this chapter) we don't want to rely on color alone for differentiation, we'll also change the border style from solid to dashed, as shown in listing 10.22. We also need to consider what to do with our fields when they're dirty and invalid (show the error message and have a red border). We want to keep the color differentiation between the fields in an error state, so we write a second rule to maintain the red border color.

Listing 10.22 Styling text fields and drop-down menu when focused

```
:is(
  input:not([type="radio"], [type="checkbox"]),
  textarea,
  select
):focus-visible {
  outline: none;                                                    ①
  border-bottom: dashed 1px var(--primary);
  border-image: none;                                               ②
}

:is(                                                                ③
  input:not([type="radio"], [type="checkbox"]).dirty:invalid,       ③
  textarea.dirty:invalid,                                           ③
  select.dirty:invalid                                              ③
):focus-visible {                                                   ③
  border-color: var(--error);                                       ③
}                                                                   ③
```

① Removes the default outline

② Removes the gradient image

③ Maintains the border color when the field has been interacted with and its value is invalid

Figure 10.27 shows our updated fields when in focus.

Figure 10.27 Text fields and drop-down menu when focused

Next, we need to handle the focus state for our radio buttons and check boxes. For those ele-

ments, we'll keep the outline but edit its appearance. As we did for our other fields, we'll use a dashed line and the primary color. We also offset the outline to create separation between the border and the outline, as shown in listing 10.23.

Listing 10.23 Styling radio buttons and check boxes when focused

```
:where(input[type="radio"],  [type="checkbox"]):focus-visible {
  outline: dashed 1px var(--primary);
  outline-offset: 2px;                    ①
}
```

① Moves the outline out 2 pixels so that it isn't right up against the border

Figure 10.28 shows our radio buttons and check box when focused.

Figure 10.28 Focus styles for radio buttons and check box

With focus handled, let's turn our attention to hover.

10.9.2 Adding hover styles

Fields in which the user inputs text, such as inputs with a type of `text` and `email` or `<textarea>` s, already change the cursor type from the default to text on hover. Figure 10.29 shows what each cursor type looks like. Note that cursors may look slightly different depending on the operating system, browser, and user settings.

Figure 10.29 Cursors in Chrome

Although our text and email inputs and text area already have some differentiation on hover, our drop-down menu doesn't. Let's change its cursor to a pointer to emphasize that the field is clickable, as shown in the following listing.

Listing 10.24 Selecting hover styles

```
select:hover { cursor: pointer }
```

With focus and hover handled, the last thing we need to worry about is making sure that our styles work for users who have `forced-colors: active` enabled.

0.10 Handling forced-colors mode

The `forced-colors` mode is a high-contrast setting that allows a user to limit the color palette to a series of colors that they set on their device. Windows' High Contrast mode is an example of this use case. When this mode is enabled, it affects many CSS properties, including some that we've used in this project, most notably `background-color`. We used `background-color` to determine whether the inner portion of the `radio` and `checkbox` inputs were visible for selected versus unselected elements. We also used it to restyle the arrow for the `select` control.

In Chrome, we can use DevTools to emulate enabling `forced-colors` mode on our machine without having to edit our computer settings. In the console of our DevTools, choose the rendering tab. If it isn't already displayed, we can click the ellipsis button to display the possible

tabs and choose it from the drop-down menu. On the tab, we look for the `forced-colors` emulation drop-down menu and set it to `forced-colors: active`. This setting updates the page's styles to act as though we had `forced-colors` set to `active` on our machine. Figure 10.30 shows the Chrome DevTools settings that enable the emulation. (Note: Browsers other than Chrome may not have this functionality, or the technique for enabling it may be different.)

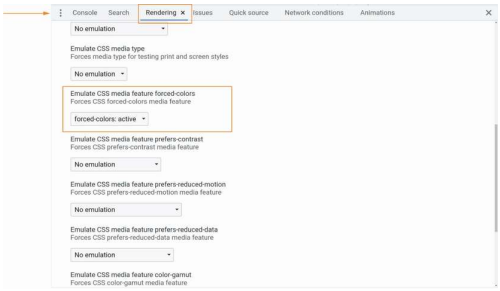

Figure 10.30 A `forced-colors:active` emulation setup in Chrome DevTools

When the emulation is applied, our page styles change (figure 10.31). We can't tell which radio button is selected or whether the check box is checked. This example demonstrates the importance of using more than color to differentiate meaning, because our error message is no longer red.

Figure 10.31 Emulated `forced-colors: active`

We won't try to reinstate our colors in this mode, because we want to respect the user's settings. But we need to make sure that selected inputs are distinguishable from those that aren't selected.

To create rules that apply only when users have `forced-colors` set to `active`, we'll use the media query `@media (forced-colors: active) { }`. Rules created inside the media query will take effect only when users have `forced-colors` enabled.

The reason why our check box and radio buttons are no longer visible is that the system-defined background color (in this case, white) is being applied to them. So we'll change our background to use a system color rather than our accent color. The CSS Color Module Level 4 specification (**http://mng.bz/o1Vy**) lists the colors available to us. We're going to use `CanvasText`, meaning

that the color we'll apply will be the same as the color being used for the text. The following listing shows our full media query.

Listing 10.25 `forced-colors: active` media query

```
@media (forced-colors: active) {
  :where(input[type="radio"], input[type="checkbox"]):checked::before {
    background-color: CanvasText;
  }
}
```

Figure 10.32 shows our page in `forced-colors` mode with our media query applied, fixing the styles that were creating problems for our users.

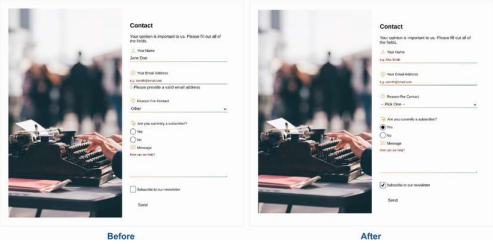

Figure 10.32 `forced-colors: active` styles fixed

When we turn the emulation off, our previously set styles remain as they were; they're not affected by those set inside the media query (figure 10.33).

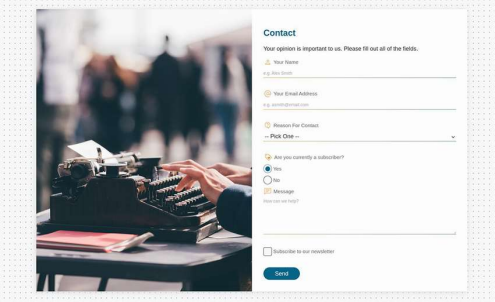

Figure 10.33 Finished product

With this last task complete, we've finished styling our form.

ummary

- Form controls whose functionality is tightly coupled with the operating system, such as drop-down menus, are harder to style than those that lack this coupling.
- We can create shapes by using gradients.
- By using `em`, we can size elements to scale with text size.
- To inherit `font-color` when doing so isn't possible otherwise, we can use the keyword value `currentcolor`.
- The `:where()` and `:is()` pseudo-classes work similarly but have different levels of specificity.
- The `:checked` pseudo-class allows us to target form elements when they're selected.

- The `:invalid` pseudo-class can be used to format fields conditionally when they're invalid.
- The validity of a field's value is determined by the attributes set on the field in the HTML.
- `:focus` styles are necessary to make our designs accessible.
- We can use `:focus-visible` to make focus style show only for keyboard users.
- In some browsers, we can forcibly make the browser apply hover and focus styles.
- It's important to use more than color alone to convey meaning, as demonstrated by the error messages in this project.
- `forced-colors` mode changes how some properties behave and the colors we can apply to the user interface.
- Media queries can be used to apply styles conditionally when `forced-colors` is set to `active`.
- In some browsers, we can emulate `forced-colors` mode to check our designs.

---

[1] *Architecting CSS: The Programmer's Guide to Effective Style Sheets*, by Martine Dowden

and Michael Dowden (2020, Apress).