8 Designing a checkout cart

This chapter covers

- Using responsive tables
- Autopositioning using Grid
- Formatting numbers
- Conditionally setting CSS based on viewport size via media queries
- Using the `nth-of-type()` pseudo-class

Many of us regularly go online to buy items ranging from food to books to entertainment and everything in between. Common to this experience is the checkout cart. We make our selections by adding them to a virtual cart or basket in which we can review our chosen items before making our final purchase. In this chapter, we'll look at how to style a checkout cart so that it works on both narrow and wide screens. We'll also look at how to handle tables for narrow and wide screens. Tables

are incredibly useful for displaying data, but they can be a bit difficult to style for mobile devices, so we'll look at a CSS solution for narrow screens.

First, though, we'll handle some theming. Regardless of the width of our screen, elements such as our input fields, links, and buttons will look the same, so we'll style them first. Defining a theme early in the process of putting together a user interface can significantly reduce redundant code. It also increases our ability to keep our styles consistent, so that whether we're creating a checkout cart or any other page or application, we can apply this process to any number of designs.

Next, we'll focus on the layout, moving from narrow to wide. On devices with narrow screens, such as phones, we tend to stack things. As screens grow larger, we add rules to make use of the full width available to us. Often, it's easier to start with the mobile layout and add to our styles as the screen gets wider than to start with wide-screen layout and have to override

previously set layout elements as screens become smaller.

.1 Getting started

We'll create styles to accommodate three sizes of screens:

- *Narrow* (most phones)—Maximum width of 549 pixels
- *Medium* (tablets and small screens)—Between 500 and 955 pixels
- *Wide* (desktop computers and high-resolution tablets)—Wider than 955 pixels

Figure 8.1 shows our starting point and the final output for each screen size.

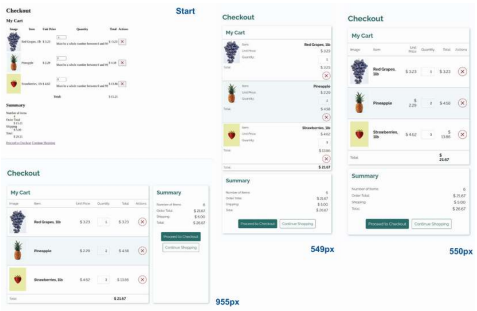

Figure 8.1 Start and end outputs for small, medium, and large screens

Regardless of the screen size, we're going to use the same HTML. We'll have one stylesheet and use media queries to adjust how our elements look depending on

screen size. Our starting HTML is on GitHub at **http://mng.bz/GRpJ** and on CodePen at **https://codepen.io/michaelgearon/pen/ExmLNxL**. The code consists of two sections, one for the cart and one for the summary, which are wrapped in a container that we'll use on wide screens to place the sections side by side. The cart section includes a heading and a table that contains each of the items in the cart. The summary section contains a heading, a description list, and two links. Figure 8.2 diagrams the HTML elements.

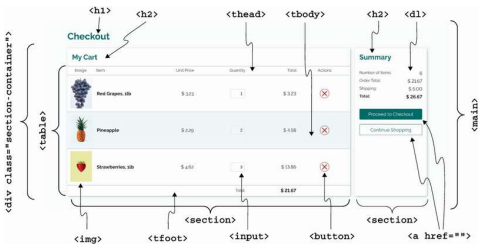

Figure 8.2 Diagram of HTML elements

The following listing is an abridged version of the HTML we're starting with.

Listing 8.1 Starting HTML

```
<body>
  <main>
    <h1>Checkout</h1>
    <div class="section-container">
```

```
<section class="my-cart">
  <h2>My Cart</h2>
  <table>
    <thead>
      <tr>
        <th>Image</th>
        <th>Item</th>
        <th>Unit Price</th>
        <th>Quantity</th>
        <th>Total</th>
        <th>Actions</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>
          <img src="./img/grapes.jpg" width="75" height="105"
            loading="lazy" alt="Red grapes">
        </td>
        <td data-name="Item">Red Grapes, 1lb</td>
        <td data-name="Unit Price">$ 3.23</td>
        <td data-name="Quantity">
          <input name="grapes" type="number"
             aria-label="Pounds of grape baskets"
             min="0" max="99"
             value="1">

        </td>
        <td data-name="Total">
          <!-- value calculated & inserted by JS -->
        </td>
        <td>
          <button type="button" class="destructive">
            <img width="24" height="24"
             src="./img/icons/remove.svg" alt="remove grapes">
          </button>
        </td>
      </tr>
      ...
    <tfoot>
      <tr>
        <th colspan="4" scope="row">Total:</th>
        <td id="total">
```

```
                <!-- value calculated & inserted by JS -->
              </td>
            </tr>
          </tfoot>
        </table>
      </section>

      <section class="summary">
        <h2>Summary</h2>
        <dl>
          <dt>Number of Items</dt>
          <dd id="itemQty">
            <!-- value calculated & inserted by JS -->
          </dd>
      ...

        </dl>
        <div class="actions">
          <a href="#" class="button primary">
            Proceed to Checkout
          </a>
          <a href="#" class="button secondary">
            Continue Shopping
          </a>
        </div>
      </section>
    </div>
  </main>
  <script src="./script.js"></script>
</body>
```

In addition to the starting HTML, we'll use a JavaScript file (`script.js`). We won't be editing or interacting with the file; it's there simply to update the totals for the summary sections.

.2 Theming

Although our layout has two clearly defined sections (the cart and the summary) and needs to work across screen sizes, some styles aren't going to change regardless of where they are or what size the screen is. These styles include

- Fonts
- Buttons and link styles
- Input and error-message styles
- Header size and color

These styles can be referred to as our *theme,* and to keep them consistent across our page, we generally want to write them once and apply them everywhere. Let's start with our fonts.

8.2.1 Typography

Currently, our `font-family` is our browser's default. For this project, we're going to import Raleway from Google Fonts and apply it to the body. We'll import both regular and bold, as we'll need both throughout this project. We'll also set a default text color of `#171717`, which looks almost black, for

our text. We aren't using black with this design because it's a soft design, and pure black can be quite harsh.

Next, we're going to handle our numbers. A font family by default has either old-style or modern numbers. The difference is how the numbers are aligned compared with the meanline and baseline, as shown in figure 8.3.

Figure 8.3 Old-style versus modern figures

Numbers in old style have portions that peek above and below the baseline; modern ones don't. Because we're creating a shopping cart, and we want to stack numbers to show them being added to create a total, we want to use modern figures so that they line up nicely. However, Raleway (the font family we've chosen for the page) uses old-style figures by default. To make our typeface use modern figures, we can use the `font-variant-numeric` property, which lets us set how we want our numbers to display. This lesser-known

property is handy for handling numbers because it allows us to control multiple facets of their display, including

- Whether zeros are displayed with a slash in them
- How the numbers are aligned
- How fractions are displayed

We're going to use a `font-variant-numeric: lining-nums` property that will change our numbers from old-style to modern. Figure 8.4 shows the summary section before and after we apply `font-variant-numeric` to our body rule. In the before version, the numerals are different sizes; in the after version, they're aligned and uniformly sized.

Figure 8.4 Before and after applying the `font-variant-numeric` property

Last, we'll change the color of our headers to teal. After that change, we'll have set the base

typography for our page. We applied it directly in the `<body>` element so that other child elements within our page will inherit the values. Listing 8.2 shows the rules we've constructed up to this point.

Listing 8.2 Typography-related styles applied to the `<body>` element

```
@import url('https:/ /fonts.googleapis.com/css2?
➥ family=Raleway:wght@400;700&display=swap');

body {
  font-family: 'Raleway', sans-serif;
  color: #171717;
  font-variant-numeric: lining-nums;
}

h1, h2 {
  color: #2c6c69;
}
```

Figure 8.5 shows our updated output.

Figure 8.5 Applied typography

Let's turn our attention to links and buttons.

8.2.2 Links and buttons

Our page has several links and buttons, but stylistically, all these elements look like buttons. They can be categorized by purpose:

- *Primary call to action*—Proceed to Checkout link
- *Secondary call to action*—Continue Shopping link
- *Destructive*—Button with an *X* to remove items from the cart

We're going to use these categories to name our classes so

that our rules will be easy to reuse.

Links versus buttons

We have both links and buttons in our project. The decision to use one or the other isn't a matter of preference; it's based on the intended functionality or purpose. For navigating, we should use a link. For performing an action, such as removing an item from our cart, we should use a button. We can style these elements however we please, but the underlying element should match the intended use case.

The reason for the distinction is that links and buttons have information and behaviors tied to them automatically by the browser. These behaviors include their capability to focus and, more important, their roles. The role of the element is used by assistive technologies to help users interact with the page.

A specific example of a difference in the behavior of a link and a button is the user's ability to right-click it to open the link in a new tab or window. If

the link is created with a button and JavaScript, this functionality isn't available to the user.

In this chapter, we're dealing with a single page, but this situation is an anomaly. In a full application, we have multiple pages or components that will reuse the same styles, so instead of naming a class something like `proceed-to-checkout`, we're going to use `primary` so that the class can easily be reused in a different context.

Before we address the differences among button types, let's consider the similarities and write a baseline for all of our links that look like buttons (which were given a class of `button`) and for buttons. After we've written a baseline, we'll make rules for each of the button types.

To create our baseline, we'll start by removing the default gray background set by the browser, which we'll do by using `background: none`. We'll also update the `padding`, `border`, and `border-radius` values.

Finally, because we're applying this rule to the links and buttons and because links are underlined by default, we're going to remove the underline from the links by setting the `text-decoration` property to `none`. The following listing shows our base rule for our buttons and links with a class of `button`.

Listing 8.3 Base styles for buttons

```
button, .button {        ①
  background: none;
  border-radius: 4px;
  padding: 10px;
  border: solid 1px #ddd;
  text-decoration: none;
}
```

① This rule will be applied to all button elements and to all elements with a class of button.

With the default state of the buttons taken care of, we'll add style changes to apply when a user hovers over a button with their mouse or focuses on it via their keyboard. To achieve this goal, we'll use the `:hover` and `:focus` pseudo-classes.

NOTE A pseudo-class is added to a selector to target a specific state. Adding style changes on `hover` and `focus` is important for accessibility, as it provides visual feedback, letting the user know that they can interact with the item. For keyboard navigation, style changes on `focus` let the user know which element they're about to interact with. Without these visual cues, it's difficult to know what we can click and where we're focused (**http://mng.bz/zmdA**).

On `hover`, we're going to add a dotted teal outline around our buttons, and to give the outline some breathing room, we're going to offset it by 2 pixels. We'll use two properties: `outline` and `outline-offset`. `outline` works similarly to `border`, taking the same three properties, which are `style`, `width`, and `color`. `outline-offset` takes a `length` value (which can be negative) that dictates the amount of space we want between the outline and the edge of the element.

For `focus`, we have the same styles as those for `hover`, but

instead of having a dotted-line outline, we'll make the line solid. The following listing shows our final CSS for the `hover` and `focus` states.

Listing 8.4 Button `hover` and `focus` states

```
button:hover,
.button:hover {
  outline: dotted 1px #2c6c69;
  outline-offset: 2px;
}

button:focus,
.button:focus {
  outline: solid 1px #2c6c69;
  outline-offset: 2px;
}
```

Figure 8.6 shows the styled links and buttons for the `hover` and `focus` states.

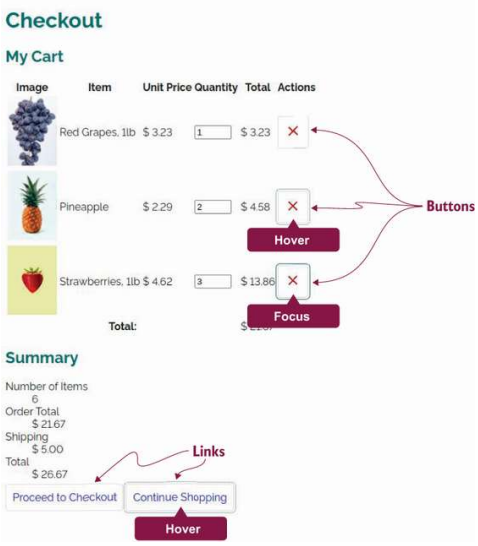

Figure 8.6 Button base styles, including `hover` and `focus`

Now that we have a baseline, we can start focusing on each individual use case. We'll start with our calls to action (the Proceed to Checkout and Continue Shopping links). Because we already have a baseline set, all we need to do is edit the colors for these use cases, as shown in listing 8.5. We differentiate between the two actions based on which we prefer (or expect) the user to select to highlight the primary choice. Being consistent about styling action types throughout the application helps us guide our users through the choices they'll have to make.

Listing 8.5 Call-to-action styles

```
button.primary,                    ①
.button.primary {                  ①
  border-color: #2c6c69;           ①
  background: #2c6c69;             ①
  color: #ffffff;                  ①
}                                  ①

button.secondary,                  ②
.button.secondary {                ②
  border-color: #2c6c69;           ②
  color: #2c6c69;                  ②
}                                  ②
```

① Applies to the Proceed to Checkout link

② Applies to the Continue Shopping link

Left to style is the Remove button in the table for the items in the cart. This button has been given a class of `destructive`. As for the previous two button types, we'll want to change the border, text, and outline colors, this time to red instead of teal to emphasize that this action is destructive. We make the button look circular by giving it a `border-radius` of `50%`. We also decrease the `padding` value; otherwise, the Remove button becomes the most prominent element in our table, which is undesirable. Finally, we center the image in the middle of the button via the `vertical-align` property. This property, which can be applied to both inline- and inline-block-level elements, dictates how the element is aligned vertically based on the inline and inline-block elements around it. We want to center the image vertically inside the button, so we'll use a property value of `middle`.

Listing 8.6 shows the CSS for the Remove button. Figure 8.7

shows the output for each state.

Listing 8.6 Remove button

```
button.destructive {
  border-color: #9d1616;
  color: #9d1616;
  border-radius: 50%;
  padding: 5px;
}

button.destructive img {      ①
  vertical-align: middle;     ①
}                             ①

button.destructive:hover,
button.destructive:focus {
  outline-color: #9d1616;
}
```

① Centers the image inside the button

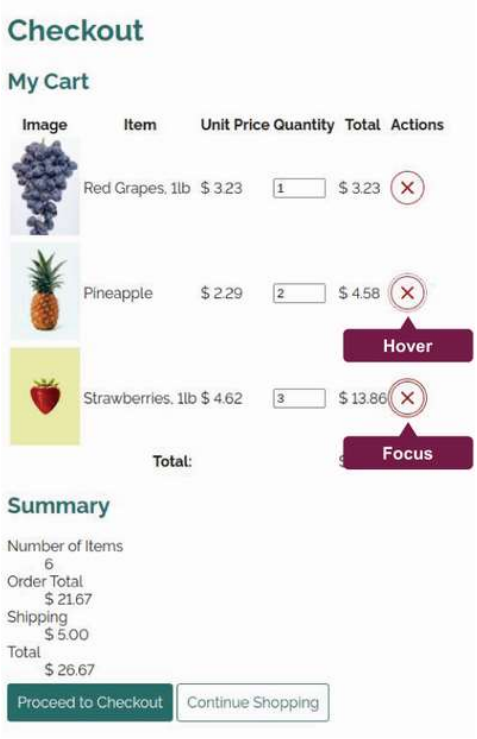

Figure 8.7 Link and button styles

8.2.3 Input fields

We're going to do some minimal styling of the input fields. We won't handle styles for invalid inputs or error messages here because the focus of this chapter is creating a responsive layout that contains a table. For a detailed look at styling forms, see chapter 10.

For this layout, we're going to give our input fields the same base styles we gave the buttons and links. Instead of writing a new rule, however, we'll add the `input` selector to the existing ruleset, as shown in the following listing.

Listing 8.7 Adding `input` to base button styles

```
button,
.button,
input {
  background: none;
  border-radius: 4px;
  padding: 10px;
  border: solid 1px #ddd;
  text-decoration: none;
}
```

Figure 8.8 shows the styled input fields.

Figure 8.8 Formatted fields

8.2.4 Table

Next, we're going to style the table. We're going to concern ourselves only with styles that relate to the theme, such as colors and borders. We'll handle layout and responsiveness in sections 8.3 through 8.5.

Our table is divided into three sections, which we'll address

in order:

- *Header*— `<thead>`
- *Body*— `<tbody>`
- *Footer*— `<tfoot>`

Styling the table headers

We're going to start by styling our table headers. Because the headers aren't as important as the content of the table itself, we'll give them a slightly smaller font size and lighter color than the rest of the text. We'll also decrease their default `font-weight` of `bold` to `normal`. By subduing the headers a bit, we're creating a visual hierarchy in the table and emphasizing what the user cares most about (the items in their shopping cart). The rule is shown in the following listing.

Listing 8.8 Styling the cells' contents

```
th {
  color: #3a3a3a;
  font-weight: normal;
  font-size: .875em;
}
```

At this point, our table headers look like figure 8.9.

Figure 8.9 Styled header cells

Boldfacing items in the second cell

In the table body (`<tbody>`), we're going to emphasize the item name (in the second column) by making the text bold. To add the `font-weight` property with a value of `bold` to the item, we're going to use the pseudo class `:nth-of-type ()` , which allows us to select an element based on its position among its siblings of the same tag. To target the second cell—the second `<td>` element—of each row in the table's body, we use `tbody td:nth-of-type(2)` . Listing 8.9 shows our rule.

Listing 8.9 Boldfacing the second cell of each row in the table's body

```
tbody td:nth-of-type(2) {
  font-weight: bold;
}
```

Figure 8.10 shows our updated table with the item names in bold.

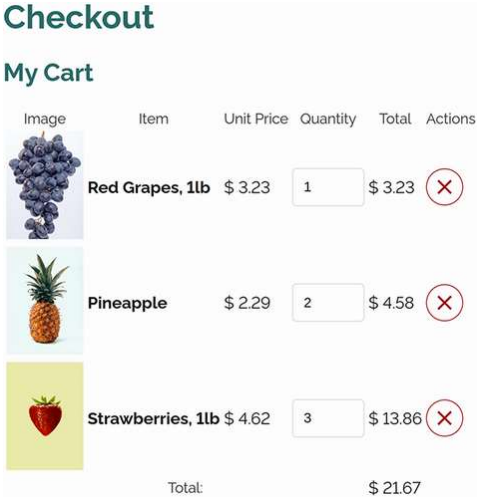

Figure 8.10 Item name in bold

Striping the rows

Next, we'll stripe the table rows. We'll use `:nth-of-type()` again, but instead of passing in a number, we'll use the keyword `even`. The rule in the following listing selects the even-numbered rows in the table body (`<tbody>`), to which we give a light-teal background color.

Listing 8.10 Striping the table body's rows

```
tbody tr:nth-of-type(even) {
  background: #f2fcfc;
}
```

Figure 8.11 shows our updated rows.

Figure 8.11 Striped rows

Boldfacing the grand total in the footer

We want to bold the grand total, which appears in the table's footer cell. Because we already have a rule that boldfaces text—the one we created to boldface item names—we can add the `tfoot td` selector to that existing rule, as shown in the following listing.

Listing 8.11 Boldfacing the footer

```
tbody td:nth-of-type(2),
tfoot td {
  font-weight: bold;
}
```

Our updated footer looks like figure 8.12.

Figure 8.12 Grand total in bold

We'll add a top border to all the rows, regardless of where they are in the table. We also want to remove the protruding white lines that appear between cells. If we darken the background color of the row, it becomes particularly visible (figure 8.13).

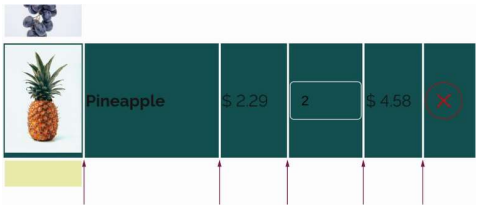

Figure 8.13 White lines between cells

Let's start by removing the gaps between our cells. But first, why are these white lines present? If we decided to give each cell in our table a border, our table would look like figure 8.14.

Image	Item	Unit Price	Quantity	Total	Actions
	Red Grapes, 1lb	$ 3.23	1	$ 3.23	
	Pineapple	$ 2.29	2	$ 4.58	
	Strawberries, 1lb	$ 4.62	3	$ 13.86	
Total:				$ 21.67	

Figure 8.14 Borders on individual cells

Notice that each cell has a square around it. The gap we see in our row is the gap between the individual cells. If we collapse the borders so that only one line appears between the cells, the gap disappears (figure 8.15).

Figure 8.15 Table with collapsed borders

The CSS property we use to remove the gap and combine the borders is `border-collapse` with a value of `collapse`. With this property added, we can also give our rows a border. Before we collapsed the borders, only the individual cells could take a border. In our project, therefore, we collapse the borders on the table and apply a border to the top of each row, as shown in the following listing.

Listing 8.12 Handling the table's borders

```
table { border-collapse: collapse; }
tr { border-top: solid 1px #aeb7b7; }
```

Figure 8.16 shows our updated table.

Figure 8.16 Styled table borders

Next, let's move on to the description list inside the summary section of the project.

8.2.5 Description list

Still to be themed is the description list (`<dl>`) in the summary section. Commonly used for creating glossaries or displaying metadata, a description list is perfect for our summary, which contains items and their values. We're going to style the description term (`<dt>`) the same way we did our table headers. We want to deemphasize them from the descriptions themselves (`<dd>`), which contains the dollar value of each ele-

ment. Because we want to style them the same way as the table headers, we'll add `dt` as a selector to the existing rule, similar to what we did to add input to the button rule in section 8.2.3.

After that, we'll add two colons after each `<dt>` by using the pseudo-element `::after` with a property of `content` to insert the character. The CSS and output are shown in listing 8.13 and figure 8.17.

Listing 8.13 Styling the description list

```
th, dt {                ①
  color: #3a3a3a;
  font-weight: normal;
  font-size: .875em;
}

dt::after {             ②
  content: ": ";        ②
}                       ②
```

① Adds a description term to our existing header styles

② Adds a colon after each description term

Figure 8.17 Description list theme

8.2.6 Cards

To give the layout some depth and achieve separation between sections, we're going to style our sections' containers as cards. *Cards* are a design pattern commonly used to separate content by encapsulating it in a box or container reminiscent of a playing card. This concept is the same as the one we used to create a profile card in chapter 6.

To pull off our card design, we'll add a pale teal background to the `<body>` and outline the sections with a shadow that looks like it's hovering slightly above the `<body>`. To create the shadow, we'll use the `box-shadow` property, which allows us to control the amount of shadow to add on the x- and y-axes, as well as the amount of blur (fuzziness), the distance it should spread, and the color the shadow should be. Figure 8.18 details how the property values are applied.

Figure 8.18 `box-shadow` property values

Optionally, we can also set a value of `inset` to indicate that the shadow should be turned inward within the element rather than around the outside. To finish the appearance of our card, we'll curve the corners with a `border-radius` value of `4px` —the same value we used for our links, buttons, and inputs. The following listing shows our section rule.

Listing 8.14 Styling the sections

```
body {
  font-family: 'Raleway', sans-serif;
  color: #171717;
  font-variant-numeric: lining-nums;
  background: #fbffff;                    ①
}

section {                                 ②
  background: #ffffff;                    ②
  border-radius: 4px;                     ②
  box-shadow: 2px 2px 7px #aeb7b7;        ②
}                                         ②
```

① Adds a background color to the page

② Makes our sections look like cards

Figure 8.19 shows our styled sections. Notice, however, that

at the bottom of the summary card, the links extend out beyond the card. This effect happens because links have a `display` value of `inline` by default.

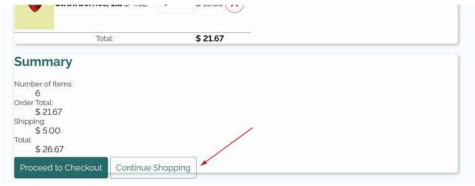

Figure 8.19 Themed sections with overflowing links

When vertical padding is added to an inline element—the links, in this case—the height of the element doesn't increase inside the flow of the page. Thus, it takes up only as much room as its content (the text), which is why it isn't increasing the height of the card. To fix this problem, we'll change their `display` value from `inline` to `inline-block`. The following listing shows the updated rule.

Listing 8.15 Section styles

```
button, .button, input {
  background: none;
  border-radius: 4px;
  padding: 10px;
  border: solid 1px #ddd;
  text-decoration: none;
```

```
    display: inline-block;
  }
```

With the fix in place, our layout looks like figure 8.20.

Figure 8.20 Styled cards

With our theme taken care of, we can start focusing on the layout.

.3 Mobile layout

We'll start with the mobile layout. The first thing we're going to do is make our table responsive.

8.3.1 Table mobile view

A traditional table layout doesn't work well on mobile devices because tables need a lot of width, which phone screens don't offer. To accommodate mobile phones, we'll make the table rows and cells act more like cards on narrow screens.

Using a media query

We'll start by using a media query to apply a set of rules to the table when the viewport is

less than or equal to `549px`. The query will be `@media(max-width: 549px) { }`. Notice that we use `max-width` here. In previous chapters, we used `min-width` because we wanted the styles to be applied only when the screen reached a certain size. In this case, we're doing the opposite: we want the styles to be applied *until* the screen reaches a certain width.

Inside this media query, we'll define what we want the table to look like on narrow screens. Figure 8.21 shows what our table currently looks like and what we're trying to achieve.

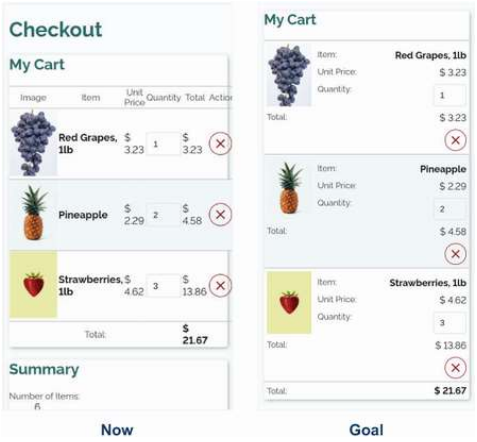

Figure 8.21 Before and after tables for a mobile device

To view the narrow-screen or mobile version, most browsers' developer tools allow us to make the browser simulate the screen of a partic-

ular device. In Google Chrome, to select a particular device, we toggle the device toolbar by clicking the icon with the phone on it at the top of the DevTools bar and then choosing the device we want to use, as shown in figure 8.22. It's worth noting, however, that this simulation is limited and shouldn't replace testing on the physical device itself.

Figure 8.22 Device simulation in Chrome DevTools

Changing the table's display structure

First, we'll stack everything vertically rather than have the elements of each row represented horizontally. We accomplish this task by giving our rows and cells a `display` value of `block`. By default, table cells have a `display` value of `table-cell`, whereas rows have a `display` value of `table-row`.

Next, we float the image to the left (chapter 7) so that the rest of the contents of the row

wrap around it. We also include some margin around the image to create some white-space between the image and the rest of the row content. The following listing shows the start of our media query and our updated cell styles.

Listing 8.16 Mobile cell and row layout

```
@media(max-width: 549px) {
  td, tr { display: block; }
  table td > img {              ①
    float: left;
    margin-right: 10px;
  }
}
```

① Specifically targets images that are immediate children of the cell to avoid floating the image in the button (the red X)

We're getting closer to our goal, but our header information isn't where we need it to be. As we see in figure 8.23, header information is at the top of the table rather than before each piece of information in the table-body rows.

Displaying content from a data attribute

To place the header information before each piece of content, we aren't going to use the header. Instead, we'll add some data attributes to the cells in our HTML: `<td data-name="Item">Red Grapes, 1lb</td>`. This data will drive labeling each row rather than the header contents in the table head.

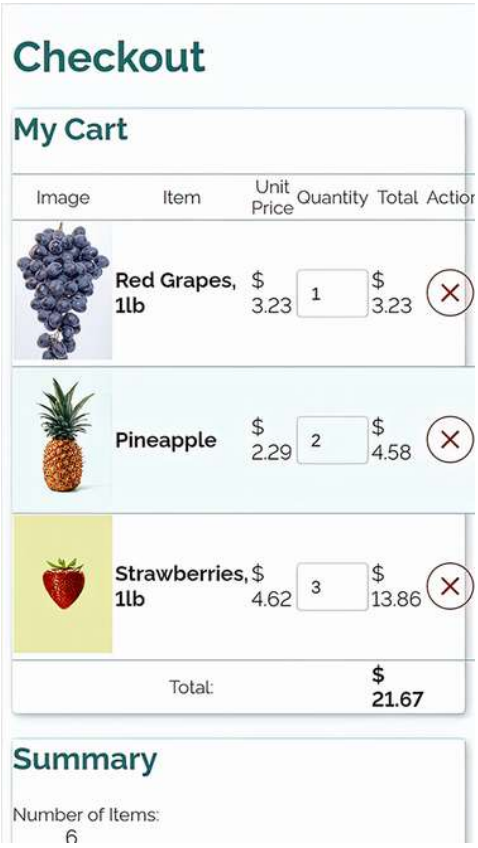

Figure 8.23 The table header is at the top of the mobile table.

We move the table header offscreen by using absolute positioning, as shown in listing

8.17. We don't want to use `display:none`, as the information available in the header is still needed by assistive technologies. By absolutely positioning it offscreen (using a large negative value), we hide it visually but not programmatically.

Listing 8.17 Hiding the table headers

```
@media(max-width: 549px) {
  ...
  thead {
    position: absolute;
    left: -9999rem;
  }
}
```

With our table head out of the way (figure 8.24), we can focus on extracting the data from the `data-name` attribute and displaying it to the user. We notice that our content shifted a bit after we removed the header because our table currently isn't taking up the full width of the screen. We'll remedy that problem later in this section. For now, let's finish handling our header information.

Figure 8.24 Removing the table head from view

To display the attribute value, we use the `attr()` function, which takes an attribute name and returns a value. For our use case, our `content` property will be `td[data-name]: before { content: attr(data-name) ":"; }`. Figure 8.25 breaks it down in detail.

Figure 8.25 Adding the header information before the cell

To align our labels and content, we use a combination of `text-align` and `float`. We use `text-align: right` in the cell to right-justify the cell contents—the item name, unit price, input field, total, and button—and then float the label (the content we get from the `data-name` attribute) to the left to create a gap between the two elements, as shown in figure 8.26. We also give the cell some padding for added whitespace between the lines of content. Listing 8.18 shows the CSS used to align the contents of the table cells.

Listing 8.18 Displaying the contents of the `data-name` attribute

```
@media(max-width: 549px) {
  ...
  td {
    text-align: right;
    padding: 5px;
  }
  td[data-name]::before {
    content: attr(data-name) ":";
    float: left;
  }
}
```

Figure 8.26 Aligning the labels and the content

Now that the data in the `data-name` attribute is being displayed, let's style it to match the definition titles. Rather than copy the styles, we can append the selector to the existing rule as shown in the following listing.

Listing 8.19 Finishing touches

```
@media(max-width: 549px) {
  ...
  th, dt, td[data-name]::before {
    color: #3a3a3a;
    font-weight: normal;
    font-size: .875em;
```

```
    }
  }
```

Full width

With the labels styled, let's turn our attention back to the fact that our table isn't taking up the full width that's available to it. We can fix this problem by giving it a width of `100%` by using the rule `table { width: 100%; }`. Because we'll want the table to take up the full width available to it regardless of screen size, we add this rule *outside* the media query.

We're almost done with the mobile styles of the table (figure 8.27). The only thing left to do is handle the table foot.

Figure 8.27 Full-width table

Table footer

In the table footer (`<tfoot>`), we want to align the text on a single line. For this task, we'll use Flexbox with a `justify-content` property value of `space-between` and an `align-items` value of `baseline` to align the label and to-

tal at opposite ends of the row on the same line. (To see how the CSS Flexbox Layout Module works, check out chapter 6.)

Looking at our table-footer HTML (listing 8.20), we notice that our first cell is a table header (`<th>`), not a table data cell (`<td>`), which makes sense because it describes the contents of that row.

Listing 8.20 Table-footer HTML

```
@media(max-width: 549px) {
  <tfoot>
    <tr>
      <th colspan="4" scope="row">Total:</th>
      <td id="total">
        <!-- value calculated & inserted by JS -->
      </td>
    </tr>
  </tfoot>
}
```

If we look closely at figure 8.27, we notice that the footer content doesn't have any padding; it goes right up against the edge of the card and row border. Earlier, we added padding to all the table data cells, not the headers, so now we'll add padding to the footer. The following listing shows a recap of the styles we edited and cre-

ated to create our mobile table layout along with our changes for the table footer.

Listing 8.21 Mobile table CSS

```
th, td, td[data-name]::before {
  color: #3a3a3a;
  font-weight: normal;
  font-size: .875em;
}
@media(max-width: 549px) {
  td, tr { display: block }
  table td > img {
    float: left;
    margin-right: 10px;
  }
  thead {
    position: absolute;
    left: -9999rem;
  }
  td {
    text-align: right;
    padding: 5px;
    vertical-align: baseline;
  }
  td[data-name]::before {
    content: attr(data-name) ":";
    float: left;
  }

  tfoot tr {
    display: flex;
    justify-content: space-between;
    align-items: baseline;
  }
  tfoot th { padding: 5px }
}
  table { width: 100% }
```

Figure 8.28 shows the finished table.

Figure 8.28 Table displayed as cards for narrow screens

Now that the table looks good on mobile devices, we'll turn our attention to the description list and the overall layout. Unlike the rules that created styles specific to small screens, this next set of rules will apply regardless of the width of the screen, so they won't be inside a media query. We'll start by addressing the description list (`<dl>`).

8.3.2 Description list

Unlike the table, which looks completely different on mobile and desktop screens, the description list will look the same regardless of screen width. Its position will change on wider screens, but the list itself will not. Because the description list is the same regardless of screen size, we won't put the layout styles inside a media query.

To display the description list, we'll use `grid` (chapter 2).

We'll define two columns and let items autoposition themselves within the two columns. When not given specific placement instructions, child elements of a grid container place themselves in the first available space, which is exactly the behavior we're going to exploit. We'll also define a gap and add some padding to the container to space the elements within the grid and card. Finally, we'll left-justify numbers. Listing 8.22 shows the CSS, and figure 8.29 shows the before and after versions of the description list.

Listing 8.22 Description list styles

```
dl {
  display: grid;
  gap: .5rem;
  grid-template-columns: auto max-content;     ①
  padding: 0 1rem;
}
dd { text-align: right; }
```

① We use max-content for our second column because we don't want the numbers to wrap, which would make them difficult to read.

Figure 8.29 Before and after layouts for the description list

8.3.3 Call-to-action links

Our description list looks a lot better, but the call-to-action links still need some help. As we did for the description list in section 8.3.2, we want our call-to-action links to be laid out the same regardless of screen size, so styles will go outside our media query.

First, we'll give the links' containers some padding and use the `text-align` property to center them. When there isn't enough room for links to be side by side and they end up stacked, we'll give them some margin to prevent them from running right up against one another. Listing 8.23 displays the code. Figure 8.30 shows before and after versions of the output.

Listing 8.23 Action links

```
.actions {
  padding: 1rem;
  text-align: center;
}
.actions a {
```

```
    margin: 0 .25rem .5rem;
  }
```

Figure 8.30 Before and after layout of call-to-action links

8.3.4 Padding, margin, and margin collapse

All the content within our sections except the headers is laid out for mobile devices. The browser gives headers a margin by default, but that setting isn't accomplishing what we want it to; instead of creating vertical space between the edge of the card and the header, it's pushing the card down. A margin pushes content but doesn't affect how much room an element or its content occupies, which is why the top margin (header) is bleeding out of the card.

If we remove the header's margin and give it padding instead, the card will expand, but the gap between the two cards will disappear. Therefore, we need to give the `section` itself some margin to add space between the two cards. If we give the sections a `margin` with a value of `1rem`

`0` (1-rem top and bottom, but not left and right), we'll still have a 1-rem gap between the two cards—a direct result of margin collapse. Unless the positioning of the elements has been altered via float or flex, two margins that run up against each other will collapse to equal the greater of the two margins. Figure 8.31 diagrams this effect.

Figure 8.31 Effects of margins and margin collapse

To add space between the edges of the card and the header, we'll replace the card header's margin with padding. Then we'll add section margins to the card to regain the lost vertical space. Finally, we'll add padding to the body so that the cards aren't stuck against the left and right edges of the screen. The following listing shows how.

Listing 8.24 Section margin and header padding

```
body {
  font-family: 'Raleway', sans-serif;
  color: #171717;
  font-variant-numeric: lining-nums;
```

```
    background: #fbffff;
    padding: 1rem;
  }

  section { margin: 1rem 0 }

  section h2 {
    padding: 1rem;
    margin: 0;
  }
```

With the mobile layout finished (figure 8.32), let's increase the width of the screen for tablets and laptops.

Figure 8.32 Before and after card headers

.4 Medium screen layout

Most of what we did for mobile devices will look great on medium-size screens. Because we used a media query to restrict table-layout changes to screens less than or equal to 549 pixels wide, the styles we wrote to edit the table won't apply to any screen that's 550 pixels wide or wider. Figure 8.33 shows the table when the viewport is 549 pixels wide and when it's 550 pixels wide. At 550 pixels of width, we're back to a standard table layout.

Figure 8.33 Break point for table

8.4.1 Right-justified numbers

Next, we're going to update the alignments of the values in the table. Because it makes computation at a glance easier, it's customary to right-justify numbers, especially if they're being totaled in a column. We'll update both the header and the cells of the unit price, quantity, and total to be right-justified.

To select the headers and cells, we could use the `:nth-of-type(n)` selector. To select the header and cells of the Unit Price column (third column), we'd use `th:nth-of-type(3), td:nth-of-type(3) { ... }` and repeat the same process for all the other columns (Quantity, Total, and Actions).

We could also think about the process a little differently. We want to right-align all columns after the first two. Inside `:nth-of-type()`, we can pass not only numbers, but also patterns. In section 8.2.4, we used this trick when we set

our background colors on our rows by passing a parameter of `even`. In this case, we're going to pass a custom pattern, using the parameter `n+3`. This pattern indicates that we want to select all matching elements starting with the third instance where `n` is the iterator and `3` is the starting point. Figure 8.34 illustrates the pattern.

Figure 8.34 `nth-of-type(n+3)` explained

Using this technique, we can select the third, fourth, fifth, and sixth cells for each row and right-align their contents, as shown in listing 8.25. Notice that we put our rule inside a media query with a `min-width` of `550px`. We don't want to apply these changes to smaller screens (defined by our previous media query as any screen smaller than or equal to 549 pixels), so we use a second media query to apply these styles only to screens that are 550 pixels wide or wider.

Listing 8.25 Right-aligning contents

```
@media (min-width: 550px) {
  th:nth-of-type(n+3),
  td:nth-of-type(n+3) {
    text-align: right;
  }
}
```

After our styles are applied (figure 8.35), we notice a few things:

- Our first two columns need their titles to be left-justified to match their content.
- The numbers inside the fields didn't right-justify themselves.
- The Remove button is up against the edge of the card.

Figure 8.35 Right-aligned number and action columns

Let's address these problems in order.

8.4.2 Left-justifying the first two columns

We'll use specificity to our advantage to handle the headers. Because, as a selector, `th` is less specific than `th:nth-of-type(n+3)`, we can make a `th` rule that aligns the text to the

left and keeps our previous rule for the other columns. The `th` rule will left-justify the header content for all columns. Then we'll override the `text-align` property value for our number and button columns in our `th:nth-of-type(n+3)` rule. The following listing shows the changes.

Listing 8.26 Updating the table header rules

```
@media (min-width: 550px) {

  th { text-align: left }

  th:nth-of-type(n+3),
  td:nth-of-type(n+3) {
    text-align: right;
  }
}
```

Now our first two table headers are left-justified instead of centered (the browser's default setting), and our other columns kept their right justification (figure 8.36).

Figure 8.36 Styled headers

8.4.3 Right-justifying numbers in the input fields

We can choose to right-justify the text inside the input field only in this table view or all the time regardless of screen size, and we do that outside the media queries. Because we right-aligned our numbers and totals in the mobile view as well, it seems logical to update the input-field style for all display sizes and include the update in our theme.

To select inputs of a type number, we can use an attribute selector: `input[type= "number"] { ... }`. We'll add `input[type="number"] { text-align: right }` to our stylesheet *outside* our media queries, as we want to apply it regardless of screen size.

With the text inside the input fields aligned (figure 8.37), the last piece we need to address is padding in all our table data cells and table headers.

Figure 8.37 Right-aligned text in input field

8.4.4 Cell padding and margin

To complete our table (medium-size screen) view, we'll add padding and margin to our cells in the table header, body, and footer. To achieve this effect, we add `td, th { padding: 10px }` to our medium-size screen (`min-width: 550px`) media query. The following listing shows the full set of changes we make to achieve the table layout.

Listing 8.27 Medium-size screens

```
input[type="number"] { text-align: right }

@media (min-width: 550px) {

  th { text-align: left }

  th:nth-of-type(n+3),
  td:nth-of-type(n+3) {
    text-align: right;
  }

  td, th { padding: 10px }
}
```

Now that we have both small and medium-size screens styled (figure 8.38), let's go a bit further and handle wide screens.

Figure 8.38 Finished mobile and tablet layouts

.5 Wide screens

As we continue to increase the width of the screen, the summary section becomes harder to read because of the increasing distance between the definition titles and descriptions (figure 8.39).

Figure 8.39 Desktop view of summary (viewport width 955 pixels)

Because we have more horizontal real estate to play with as the screen gets wider, we'll bring the summary section up beside the cart section when the viewport reaches 995 pixels wide or larger, as shown in the wireframe in figure 8.40.

Figure 8.40 Layout wireframes

To change our layout conditionally based on the screen being 955 pixels wide or larger, we'll create the media query `@media (min-width: 995px) { }`. In the HTML

shown in the following listing, we have a container `<div>` around our two sections with a class of `section-container`.

Listing 8.28 Page HTML

```
<main>
   <h1>Checkout</h1>

   <div class="section-container">          ①

     <section class="my-cart">              ②
       <h2>My Cart</h2>                     ②
       <table> ... </table>                 ②
     </section>                             ②

     <section class="summary">              ③
       <h2>Summary</h2>                     ③
       <dl> ... </dl>                       ③
       <div class="actions"> ... </div>     ③
     </section>                             ③

   </div>                                   ④

 </main>
```

① Container for the two cards (My Cart and Summary)

② My Cart card

③ Summary card

④ End of container

Inside our new media query, we'll give the container a `display` property value of `flex`. This value allows the two

items to come side by side and align themselves on the x-axis. Then we'll add a gap of `20px` between the two sections.

Flexbox will autocalculate the amount of space to give each section. We can influence how the browser assigns dimensions via the properties `flex-grow`, `flex-shrink`, and `flex-basis`. We're going to give the summary section a `flex-basis` value of `250px` and the cart section a `flex-grow` value of `1`.

Applied to the summary card, `flex-basis` will set the initial size of the section when the browser starts calculating how much room to assign each section. If the content to which flex is being applied can accommodate the section's being 250 pixels wide, the browser won't alter the section's dimensions; otherwise, the browser will adjust the section as necessary. The `flex-grow` property tells the browser that if space is left over after `flex` has been applied to the content, this element should be made wider to use the extra space. Figure 8.41 shows our sections with and without

these two properties influencing how the elements are sized.

Figure 8.41 Influencing the size of elements to which `flex` has been applied

With `flex-grow` and `flex-basis` , we can control the width of the table relative to the summary card. Therefore, we use the media query in the following listing for our project.

Listing 8.29 Placing the two cards side by side on wide screens

```
@media (min-width: 955px) {
  .section-container {
    display: flex;
    gap: 20px;
  }
  section.my-cart { flex-grow: 1; }
  section.summary { flex-basis: 250px }
}
```

Figure 8.41 shows our layout when the screen is 955 pixels wide. But if we make the screen even wider, such as for the extra-wide curved displays, we eventually get to a point where the content once again becomes unreadable

(figure 8.42). Because we set a `flex-basis` value on the summary card, it stays readable, but because the table is made to keep growing (via the `flex-grow` property), it becomes unwieldy.

Figure 8.42 Layout on a screen 2,000 pixels wide

To prevent this growth, we can limit the width of the `<main>` element (inside which our main header and cards are contained). This change ensures that no matter how wide the user's display is or how the user chooses to extend the window, the content remains usable. We can center the body by giving the left and right margins a value of `auto`, as shown in the following listing.

Listing 8.30 Maximum width of the `main` element

```
main {
  max-width: 1280px;
  margin: 0 auto;
}
```

If we look at our layout again on an extremely wide screen

with these last styles applied (figure 8.43), we see that we've constrained and centered our content.

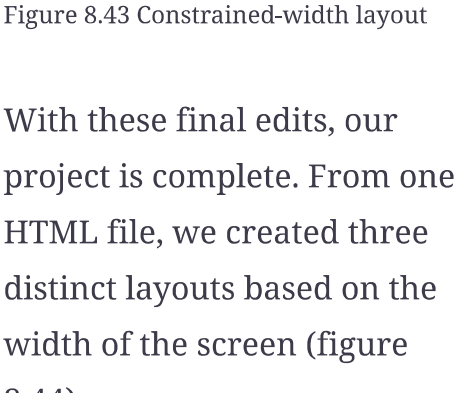

Figure 8.43 Constrained-width layout

With these final edits, our project is complete. From one HTML file, we created three distinct layouts based on the width of the screen (figure 8.44).

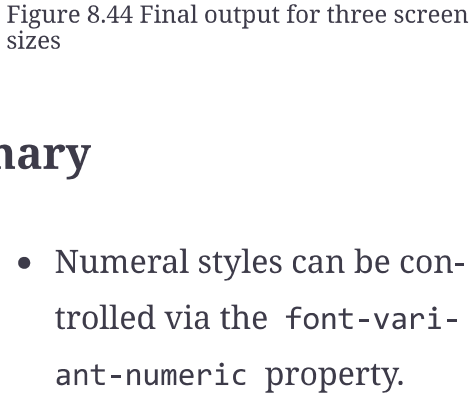

Figure 8.44 Final output for three screen sizes

ummary

- Numeral styles can be controlled via the `font-variant-numeric` property.
- Media queries allow us to apply styles conditionally based on screen size.
- Naming CSS classes based on the content they're styling or the content they represent can be helpful for creating names that are easy to understand and maintain.
- HTML attribute values can be used to select elements.

- HTML attributes can be displayed via CSS using pseudo-elements, the `content` property, and the `attr()` function.
- Margins can collapse.
- Elements set to `display:flex` can be controlled via `flex-grow`, `flex-shrink`, and `flex-basis`.
- `:nth-of-type` can take numbers, keywords, or custom patterns to target elements based on their position relative to others of the same type of elements inside a container.