4 Creating a responsive web newspaper layout

This chapter covers

- Using the CSS Multi-column Layout Module to create a newspaper layout
- Using the `counter-style` CSS at-rule to create custom list styles
- Styling images using the `filter` property
- Handling broken images
- Formatting captions
- Using the `quotes` property to add quotation marks to HTML elements
- Using media queries to change the layout based on screen size

In chapter 1, we looked at creating a single-column article, which taught us the basic principles of CSS. The design, however, was simple. Let's revisit the concept of formatting articles but make it much more visually interesting. In this chapter, we'll style our content to look like a page out of a newspaper, as shown in figure 4.1.

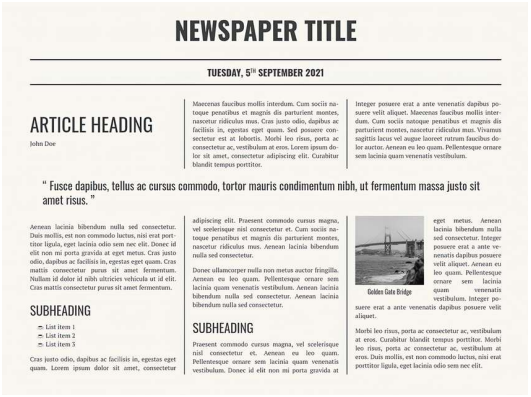

Figure 4.1 The result we want to achieve

To create the content columns, we'll use the CSS Multi-column Layout Module. Along the way, we'll also look at how we can manage the space between the columns, how to span elements across columns, and how to control where the content breaks to a new column.

Part of the newspaper page uses a list of items, which has some default styles provided to us by the user agent (UA) stylesheet. We'll look at how to use the CSS Lists and Counters Module, which allows us to customize how our `list-items` counters (the numbers and bullets) are styled.

Another concept we'll cover in this chapter is how to style images, including the use of the `filter` property in conjunction with functions to alter the image's appearance. We'll also look at solutions for broken images and ways to make them fail gracefully. When we say "fail gracefully" (sometimes known as *graceful degradation*), we're putting in place

fallbacks to employ if the thing we're trying to load is having an problem or a feature we're trying to use isn't compatible with the user's browser.

You can find the code for our project in the `chapter-04` folder of the GitHub repository (**http://mng.bz/OpOa**) or on CodePen at **https://codepen.io/michaelgearon/pen/yLxzbr**. Our starting HTML consists of the elements in listing 4.1. Within the `<body>` element are the title of the newspaper and print date followed by an article. The article has a heading, author name, a quote, two subheadings, a list, some paragraphs, and an image.

Listing 4.1 Starting HTML

```
<body>
  <h1>Newspaper Title</h1>                                          ①
  <time datetime="2021-09-07">                                      ②
    Tuesday, 5<sup>th</sup> September 2021                          ②
  </time>                                                           ②
  <article>                                                         ③
    <h2>Article heading</h2>                                        ④
    <div class="author">John Doe</div>                              ⑤
    <p>Maecenas faucibus mollis interdum. Cum sociis nato...</p>
    <p>Integer posuere erat a ante venenatis dapibus posu...</p>
    <blockquote>                                                    ⑥
      Fusce dapibus, tellus ac cursus commodo, tortor ma...         ⑥
    </blockquote>                                                   ⑥
    <p>Aenean lacinia bibendum nulla sed consectetur. Dui...</p>
    <h3>Subheading</h3>                                             ⑦
    <ul>                                                            ⑧
      <li>List item 1</li>                                          ⑧
      ...                                                           ⑧
    </ul>                                                           ⑧
    <p>Cras justo odio, dapibus ac facilisis in, egestas ...</p>
    <p>Donec ullamcorper nulla non metus auctor fringilla...</p>
```

```
      <h3>Subheading</h3>                                                    ⑨
      <img src="./image.jpg" alt="">                                         ⑩
      <p>Praesent commodo cursus magna, vel scelerisque nisl...</p>
      <p>Morbi leo risus, porta ac consectetur ac, vestibulu...</p>
    </article>                                                               ⑪
  </body>
  </html>
```

① Newspaper title (main heading)

② Print date

③ Start of the article

④ Article heading

⑤ Article author

⑥ Quote

⑦ First subheading

⑧ List

⑨ Second subheading

⑩ Image

⑪ End of the article

Figure 4.2 shows our starting point. The styles applied to the HTML are the defaults provided by the browser. No author styles have been applied to the page yet.

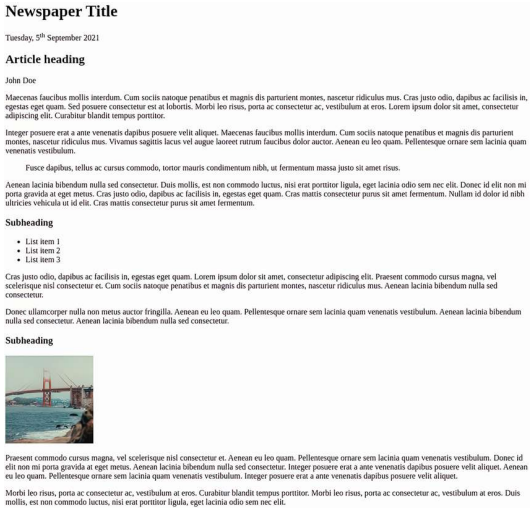

Figure 4.2 Starting point

Before we worry about layout, let's define our theme.

.1 Setting up our theme

The theme sets the tone for the page; it generally consists of colors, fonts, borders, and sometimes padding. Our theme will stay the same regardless of screen size or layout. Often, the theme of a website is tightly coupled to its logo and brand colors.

We'll set some defaults on the `<body>` element that can be inherited by its descendants. As a general rule, styles that revolve around typography (`color`, `font-family`, and so on) can be inherited by most elements. Exceptions are some form elements, which we cover in chapter 10. When we set inheritable properties on the parent, the styles trickle down to the descendents, relieving us of the need to apply them to every element.

4.1.1 Fonts

We apply a background color, font, and text color (listing 4.2). Notice that before the `body` rule, we import our chosen `font-family` from Google Fonts. Google Fonts (**https://fonts.google.com**) is a popular option with developers, as it's freely available, and users don't need to create an account or worry about licensing.

WARNING When loading libraries or assets, including fonts, from a content delivery network (CDN), always check the privacy and data terms, and make sure that they're compliant with local laws such as General Data Protection Regulation (GDPR) and European Union laws. When in doubt, ask your legal team. If CDNs aren't an option for you, check out chapter 9 for details on loading fonts locally.

PT Serif, for example, isn't a font we can expect a user to have already loaded on their computer; therefore, we have to import it for the browser to tell it what the *glyphs* (letters, numbers, and symbols) should look like. We also provide a default of `serif` as a fallback should the import fail.

Web-safe fonts

Only a few web-safe fonts (fonts we can assume that most devices will

have access to) are available. According to W3Schools (**http://mng.bz/Y6Ea**), some safe options are Arial, Verdana, Helvetica, Tahoma, Trebuchet MS, Times New Roman, Georgia, Garamond, Courier New, and Brush Script MT. But no official standard specifies what constitutes a web-safe font or which ones would truly be available on all browsers and devices. Therefore, regardless of the font family we choose, it's good practice always to provide a fallback value (`serif` , `sans-serif` , `monospace` , `cursive` , or `fantasy`).
Although we'll do the bulk of the layout later in the chapter, we'll add some left and right padding on our body now to move our text away from the edge.

Listing 4.2 Defining some theme styles

```
@import url('https:/ /fonts.googleapis.com/css2?family=PT+Serif&display=swap'); ①

body {
  background-color: #f9f7f1;
  font-family: 'PT Serif', serif;                                    ②
  color: #404040;
  padding: 0 24px;
}
```

① Imports PT Serif from Google Fonts

② Applies PT Serif to our content and provides a fallback

Figure 4.3 shows our updated page. Notice that all the elements in the `<body>` have inherited the `color` and `font-family`.

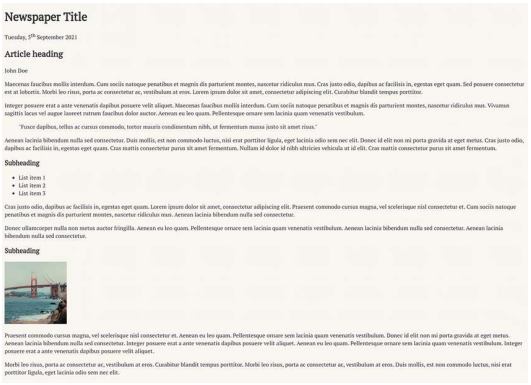

Figure 4.3 Theme styles applied to the body being inherited by descendants

Next, we'll style the main heading and subheadings. Let's start with the newspaper title, which is the `<h1>` in the HTML. We want to change the `font-family` to use a typeface called Oswald, increase the text size, make it bold, transform the font to use all capital letters, set the line height, and center the text. Like PT Serif, Oswald isn't a font that we can expect most users' devices to know about, so we'll import it much as we did PT Serif.

Notice that for the text size, we use unit `rem`, which stands for "root em." An *em* is a relative unit based on the font size of the element's parent. If a container `div` has a font size of `12px`, and we set a child element's size to `.5em`, the child element's size would equal to 12 x .5 or `6px`. The `rem` unit works similarly, but instead of being relative to

the parent's font size, its base value is that of the root element—in our case, `<html>`. We didn't set a font size on the HTML element; therefore, our base will be the browser's default, which in most cases is `16px`. With that in mind, a font size of `4rem`—the size we set on our main heading—would be equivalent to 4 x 16 or `64px`.

To import Oswald from Google Fonts, we can add a second `@import` at the top of our file, or for better performance, we can combine the two imports into one `@import` statement. The ability to combine the two imports is specific to Google Fonts; not all CDNs have this ability.

Notice in listing 4.3 that in our `@import`, after the name of the font, we see `:wght@400;700`. This code indicates which Oswald font weights we want to import.

Listing 4.3 Styling the newspaper title

```
@import url('https:/ /fonts.googleapis.com/css2?
➥ family=Oswald:wght@400;700&family=PT+Serif&display=swap');      ①

h1 {
  font-weight: 700;                                                ②
  font-size: 4rem;
  font-family:'Oswald', sans-serif;
  line-height: 1;
  text-transform: uppercase;
  text-align: center;
}
```

① Updated import that includes both Oswald and PT Serif

② Equivalent to using a value of bold

Figure 4.4 shows our updated title.

Figure 4.4 Styled title

4.1.2 The font-weight property

The `font-weight` property can take either a number value between `100` and `900` or a keyword value (`normal`, `bold`, `lighter`, or `bolder`). `normal` is equivalent to `400`, and `bold` to `700`. `lighter` and `bolder` change the element's font weight based on the font weight of the parent element. Table 4.1 shows the relationships between numeric `font-weight` values and their common name equivalents.

Table 4.1 `font-weight` values and their common weight names

Value	Common weight name
`100`	Thin (Hairline)
`200`	Extra Light (Ultra Light)
`300`	Light
`400`	Normal (Regular)
`500`	Medium
`600`	Semi Bold (Demi Bold)
`700`	Bold
`800`	Extra Bold (Ultra Bold)
`900`	Black (Heavy)
`950`	Extra Black (Ultra Black)

If we don't import the weight that matches the one we set in the rule, the browser will apply the closest weight it has access to. Therefore, had we imported Oswald only with a weight of `400` and applied a `font-weight` value of `bold` to our element, the browser would have displayed our text with a weight of `400` because that value would be the only one it had to work with.

4.1.3 The font shorthand property

Using the `font` shorthand property, we can combine most of the styles in our rule. The `font` property requires us to provide a `font-family` and `size`, optionally followed by `style`, `variant`, `weight`, `stretch`, and `line-height`, using the following syntax: `font: font-style font-variant font-weight font-stretch font-size/line-height font-family`. The next listing shows our updated rule using `font`.

Listing 4.4 Title styles using the `font` shorthand property

```
h1 {
  font: 700 4rem/1 'Oswald', sans-serif;
  text-transform: uppercase;
  text-align: center;
}
```

Let's apply the concepts we've covered regarding importing fonts, `font-weight`, and the `font` shorthand property to style the article's main heading and subheadings.

4.1.4 Visual hierarchy

To create a visual hierarchy on the page, we'll set the article heading `<h2>` to be smaller than our newspaper's main heading `<h1>` but larger than the subheadings within the article `<h3>`. Generally speaking, the larger an element is, the more important it's perceived to be,

so we use size to make our headers stand out. By using a different `font-family` from the one we use for the main body text and making all the heading letters uppercase, we further the distinction.

Creating a visual hierarchy is important, as it allows the user to glance at the screen and immediately recognize elements of interest. It also segments information into groups, making the information easier to process and understand.

Listing 4.5 shows our header rules. We'll keep the same font family, uppercase the lettering, and adjust the sizing. We'll also remove the browser-provided bottom margins of both article headers to keep them closer to the text they precede.

Listing 4.5 Article header rules

```
h2 {                                          ①
  font: 3rem/.95 'Oswald', sans-serif;
  text-transform: uppercase;
  margin-bottom: 16px;
}

h3 {                                          ②
  font: 2rem/.95 'Oswald', sans-serif;
  text-transform: uppercase;
  margin-bottom: 12px;
}
```

① Article heading

② Article subheadings

Now our article's headers look like figure 4.5.

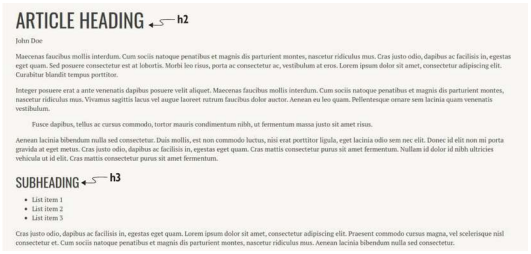

Figure 4.5 Styled article headings

4.1.5 Inline versus block elements

Let's continue to make important elements stand out from the rest of the content, starting with the publication date, which is inside a `<time>` element in our HTML. The `<time>` element semantically denotes a specific period in time; it takes an optional `datetime` attribute that provides the date as a machine-readable format for search engines. Our `<time>` element looks like this: `<time datetime="2021-09-07">Tuesday, 5<sup>th</sup> September 2021</time>`. Figure 4.6 shows the look we want to achieve.

Figure 4.6 Styled publication date

Starting with the typography, we center the text and use the Oswald font family, set the `font-size` to `1.5rem`, and make the text uppercase and bold. Then we change the text size of the *th* found in the super-

script element (`<sup>`) to a slightly smaller font size and normal weight to decrease its prominence.

Next, we add the top and bottom borders to be 3-pixel-thick, solid, dark gray lines. After adding the borders, we add some top and bottom padding so that we have some breathing room between the text and the borders.

The `<time>` element is an inline-level element, meaning that it takes up only the exact amount of space it needs for its content, the same way that a `<span>` or `<a>` element does.

By contrast, block-level elements (such as `<div>` , `<p>` , and `<ul>`) place themselves on a new line and take the full width of their available space unless given a set width. To achieve the design in figure 4.6, we want our `<time>` element to behave as though it were a block-level element so that the text will place itself in the middle of the screen, and the borders will take the full width of the page.

Figure 4.7 The `<time>` element exhibiting inline behavior

To change the element's default behavior, we'll use the `display` property and give it a value of `block` .

Figures 4.7 and 4.8 show the `<time>` element before and after we add the `display` property. In figure 4.7 (before adding the `display` property), the element is exhibiting its default behavior as an inline-level element. In figure 4.8 (after adding the `display` property), the element behaves like a block-level element, taking the full width of the screen.

Figure 4.8 The `<time>` element exhibiting block behavior

Styling the publication date in this manner serves two purposes: the styling makes it stand out, and it creates a visual divide between the newspaper information (the date and newspaper's main heading) and the article itself (everything below the date). The following listing contains the rules we wrote to achieve our design.

Listing 4.6 Styling the publication date

```
time {
  font: 700 1.5rem 'Oswald', sans-serif;     ①
  text-align: center;                        ①
  text-transform: uppercase;                 ①

  border-top: 3px solid #333333;             ②
  border-bottom: 3px solid #333333;          ②
  padding: 12px 0;                           ②
```

```
    display: block;                    ③
  }
  time sup {                           ④
    font-size: .875rem;
    font-weight: normal;
  }
```

① Typography

② Handles the borders and padding

③ Makes the element behave like a block-level element

④ Styles the “th”

4.1.6 Quotes

The last bit of text we want to feature is the `<blockquote>` after the second paragraph in the article. Sticking with our theme, as with all the other elements we want to make stand out, we’ll make the font bigger and bolder. We’ll also adjust the line height and add a margin to the element. Isolating an element from the content around it makes it easier to spot. By adding a top and bottom margin, we add space between the quote and the paragraphs above and below it, creating whitespace around the element. By adding left and right margins, we change its alignment, effectively indenting it. The added whitespace creates isolation.

Let’s also add quotation marks to our `<blockquote>` . To add the quotation marks at the beginning and end of our quote, we could simply go

into the HTML and add them manually, or we can do the job programmatically with CSS.

The `quotes` property allows us to define custom quotation marks. We can pass to this property the symbols we want to use as our double- and single-quote glyphs. Not all languages use the same symbols. American English, for example, uses “ … ” and ‘ … ’, but French uses « … » and ‹ … ›. Using the `quotes` property, we can customize the symbols we want to use. If we don’t provide a value for `quotes`, the browser’s default behavior is to use what is customary for the language set on the document.

The `quotes` property, however, only defines the symbols; it doesn’t add them. To add them, we use the `content` property values `open-quote` and `close-quote` in conjunction with the `::before` and `::after` pseudo-elements, as shown in listing 4.7. The pseudo-elements allow us to insert content via the `content` property before and after the element to which they’re applied, respectively.

Listing 4.7 Styling the `blockquote`

```
blockquote {
  font: 1.8rem/1.25 'Oswald', sans-serif;
  margin: 1.5rem 2rem;
}
blockquote::before { content: open-quote; }
blockquote::after { content: close-quote; }
```

The `open-quote` and `close-quote` keywords represent opening and closing quotation marks as defined by the `quotes` property. Because we didn't add a `quotes` declaration to our `blockquote` rule, the browser will use what is conventional for the document's language, which we set to `en-US` in the language (`lang`) attribute of the `<html>` tag. The value of `en-US` specifies that our document is written in American English; therefore, the symbols that the browser renders are “ and ”, as we see in figure 4.9.

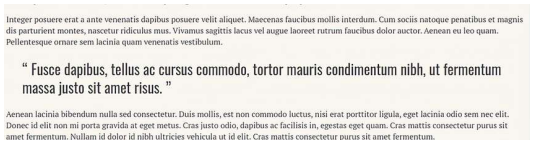

Figure 4.9 Styled title, heading, subheadings, and quote

With our quote styled, let's turn our attention to the bulleted list in the middle of the article.

.2 Using CSS counters

Our article contains an unordered (bulleted) list. Currently, each list item has the default bullet before it. We can alter what our bullet looks like by using the `list-style-type` property. By default, we can choose disc (•), circle (○), square (▪), and numbers or letters in several languages, alphabets, and number formats. But let's say we want our bullet to be an emoji—specifically, the

hot-beverage emoji (☕). We'll have to create a custom list style.

To create our custom list style, we'll use the `@counter-style` at-rule. We used at-rules in chapter 3 when we created keyframes. In this case, instead of defining how an animation will behave, we'll define how a list looks and behaves. The at-rule is called `counter-style` because it specifically addresses the built-in counting mechanism for list items in CSS. Under the covers, regardless of whether the list is ordered or unordered, the browser keeps track of the position of the item in the list—that is, it counts the items.

As with keyframes (which we named so we could reference them inside our `animation` property), we'll name our `@counter-style` so we can reference it with the `list-style` property and apply it to our list. Let's name our list-style `emoji`. Our at-rule, therefore, will be `@counter-style emoji { }`.

Next, we'll define the behavior our `list-style` needs to have inside of our at-rule. We'll use three properties: `symbols`, `system`, and `suffix`.

4.2.1 The symbols descriptor

The `symbols` descriptor defines what will be used to create the bullet style. To define our emoji as the sym-

bol to use, we can use the emoji directly or use its Unicode value.

Unicode is a character-encoding standard that specifies how a 16-bit binary value is represented as a string. In other words, it's the code representation of our emoji. The actual emoji image is determined by the operating system and browser, which is why we see variations in how emojis look between iOS and Android, for example. The Unicode value tells the machine what to render.

We use lookup tables such as the one at **http://mng.bz/GRQJ** to find this value for our emoji. ☕ is listed as having the following code: `U+2615`. To tell our CSS that we're using a Unicode value, we'll replace the `U+` with a backslash (`\`). Using the Unicode value, our declaration value will be `symbols: "\2615"`. If we use the emoji, our declaration value will be `symbols: ☕;`.

Next, we need to define our `system` descriptor.

4.2.2 The system descriptor

Regardless of type of list (ordered or unordered), under the covers the browser keeps track of the list item it's styling based on its position inside the list. The first item's integer value is 1, the second is 2, and so on. The `system` descriptor value defines

the algorithm used to convert that integer value to the visual representation we see on the screen.

We're going to use the `cyclic` value. Earlier, we provided only one emoji in our `symbols` declaration, but we could have included multiple different emojis using a space-delimited list. A `cyclic` value tells the browser to loop through these values and, when it runs out, to start back at the beginning. Because we have only one value, the browser will apply the ☕ to the first list item and then run out of symbols. Having run out before the second list item, the browser starts back at the beginning of the list, applying the ☕ once again but to the second list item this time. Then the browser will run again, moving on to the third list item, and the cycle continues. Finally, we'll set a suffix.

4.2.3 The suffix descriptor

The `suffix` descriptor defines what comes between the bullet (our emoji) and the contents of the list item—by default, a period. We want to replace the period with plain whitespace between our emoji and list-item content. Therefore, we'll set our `suffix` descriptor value to `" "` (a blank space).

4.2.4 Putting everything together

With our `counter-style` defined, we can apply it to our list. Remember that we named the `counter-style` rule `emoji`. We'll apply the name as the `list-style` property value for our list, as shown in the following listing.

Listing 4.8 Styling the list

```
@counter-style emoji {          ①
  symbols: "\2615";             ②
  system: cyclic;
  suffix: " ";
}

article ul {
  list-style: emoji;            ③
}
```

① The at-rule defining the custom list-style's behavior

② ☕

③ Applies the custom list-style to the article's lists

Figure 4.10 shows our newly styled list.

Figure 4.10 List styled using ☕ as counters

4.2.5 @counter versus list-style-image

Another way to change the list item marker being used is to use the `list-style-image` property and assign an image to it, similarly to the way we can set a background image by using the `background-image` property. We didn't use that approach in this project because we used an emoji, which is a Unicode character and not an image. The counter also provides us much more control, such as assigning a suffix or specifying how the counter cycles through the item markers being displayed.

If we're looking only to change the marker to a specific image, `list-style-image` is perfect. But if we want to have more granular control or, as in our case, to use text, we need to use `@counter`. Let's continue going down the page, styling the image next.

.3 Styling images

Historically, newspapers were printed in black and white. Colored ink in newsprint is a fairly new thing when we consider the history of print. To give our design a bit of a retro vibe, therefore, we'll make our image grayscale. First, we'll look at how to alter our image using filters. Unlike in print, on the web we need to worry about resources not load-

ing or links being broken, so we'll also look at how to make the image fail gracefully should it fail to load. Finally, we'll add a caption to accompany the image.

4.3.1 Using the filter property

As in photo editors or on social-media websites like Instagram, we can apply filters to images with CSS. We can alter colors, blur, and add drop shadows, for example. Figure 4.11 shows examples of some of the things we can do to our images by using filters in CSS. Check out this code sample in CodePen to see it in action: **https://codepen.io/michaelgearon/pen/porovxJ** .

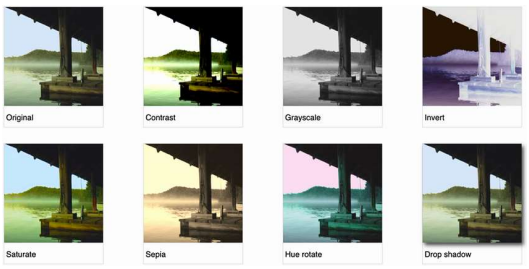

Figure 4.11 Examples of images altered with the `filter` property

If we think about pre-digital-era photography, when we used film and had to go to a shop to have it developed, we applied filters by adding a translucent disk over our lens, which altered the light coming into the camera box and onto the film. By altering the nature of the light, we altered the image being produced. If we used a red filter while taking a picture, for example, only the red-

colored wavelength was allowed through; our picture was tinted red. Polarized sunglasses are another example of a filter that alters the light coming through a lens.

We can still use physical filters with digital cameras. In many cases, however, filters are applied digitally after the picture has been taken.

In CSS, we use the `filter` property to apply a filter to the image; then we use a function that defines the behavior we want the filter to have. You can find a list of the available functions at **http://mng.bz/zmYA**. We'll use the `grayscale()` function to make our picture appear to be a black-and-white photo.

The `grayscale()` function takes a percentage, which represents how much we want to reduce the amount of color in the image. We want to remove all the color, so we'll pass in a value of `100%`. Our rule, then, will be `img { filter: grayscale(100%) }`. Figure 4.12 shows the filter applied to our image.

Figure 4.12 Grayscale Image

One consideration to make before using filters is their impact on website performance. Some of the filter functions, such as `grayscale()`, are relatively simple for the browser to process, but functions such as `drop-shadow ()` and `blur ()` can be resource-intensive. If we find that we're applying many filters to a large number of images, we should consider the impact of the filters on overall page performance and whether we should be preprocessing the image rather than applying the change with CSS.

4.3.2 Handling broken images

Even with the most thorough diligence and best testing practices, broken image links can happen. Let's add some fallbacks to ensure that if our image fails to load (regardless of the reason), we'll maintain a positive experience for our users.

First, let's deliberately break our link. In the HTML, we'll replace the path to the image with an image file that doesn't exist in our project, like

so: `<img src="./my-broken-image.jpg" alt="my broken link" />`. The image will display as broken, as shown in figure 4.13.

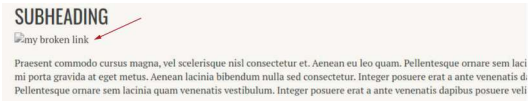

Figure 4.13 Broken link with `alt` text

Notice that the text provided in the `alt` attribute is displayed. The `alt` attribute allows assistive technologies to inform users about the image being displayed. A common use case is a blind user accessing content via a screen reader. In this particular case, because the image is broken, the text replaces the image. Although the situation isn't ideal, in the event of an image failure, users can still be informed of the content that the image was supposed to provide.

In our case, the image is purely decorative and doesn't provide any content value, so if the link is broken, we'll hide the image. Nothing will be there, but "nothing" is less unsightly than a broken-image icon. Because there's no way to detect that an image is broken in CSS, we need to use a little bit of JavaScript to know when to hide the image. We'll use the `onerror` JavaScript event handler to trigger a change in styles as follows: `<img src="..." alt="..." onerror="this.style.display='none'" >`. The bit of code that is of interest

to us here is the `onerror` attribute. When an error occurs, the JavaScript inside the `onerror` attribute triggers and sets the image's `display` property to `none`, hiding the image. We can see that, in figure 4.14, our broken image is missing.

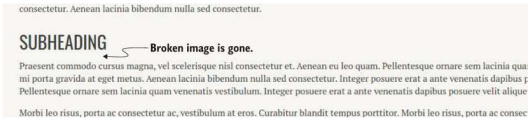

Figure 4.14 The broken image is missing.

The `onerror` code triggers only when the image fails to load, so let's fix our resource path to our image but keep the error handling: `<img src="./image.jpg" alt="" onerror="this.style.display='none'" >`. Now our image is restored (figure 4.15), but we have a safeguard in case it fails.

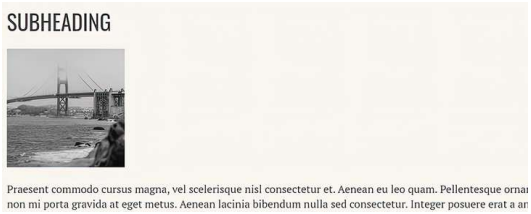

Figure 4.15 Restored image with fallback

Next, let's add a caption to the image.

4.3.3 Formatting captions

The image doesn't have a caption, so we're going to add one by using the `<figure>` and `<figcaption>` HTML elements. Then we'll style it.

These two elements go hand in hand. `<figure>` contains the image and then the optional `<figcaption>`. Often in books and other publishing material, a diagram, chart, or image has text below it that describes it or relates it to the text. Semantically, the benefit of grouping the image and the caption is that grouping programmatically links the image with its caption. From a styling perspective, having the elements together in a parent element allows us to position the element and its caption as a unit. The following listing shows how to change the HTML to add the figure and caption.

Listing 4.9 Adding a `<figure>` and `<figcaption>` to the HTML

```
<figure>                                                                    ①
  <img src="./image.jpg" alt="" onerror="this.style.display='none'" />      ②
  <figcaption>Golden Gate Bridge</figcaption>                               ③
</figure>                                                                   ④
```

① Start of the figure

② Our image

③ Our image caption

④ End of the figure

Let's style the figure and the caption, starting by removing the browser-provided margins (figure 4.16) that are currently being applied to the figure.

Figure 4.16 `<figure>` with browser-provided styles

Next, we'll reinstate a bottom margin so that our caption is kept separate from the paragraph below it. Finally, we'll center the image and caption. We'll style the caption's text to use the Oswald font family (the one we used for all the headers) to differentiate it visually from the article text. The following listing shows the CSS used to style the figure and caption.

Listing 4.10 `figure` and `figcaption` styles

```
figure {
  margin: 0 0 12px 0;      ①
  text-align: center;
}
figcaption {
  font-family: 'Oswald', sans-serif;
}
```

① Padding shorthand property: top, left, and right padding set to 0 and bottom set to 12px

Figure 4.17 shows the progress we've made on our project thus far. At this point, the page looks good on narrow screens, but we still need to display our columns on wide

screens. Next, we'll look at how to create a multicolumn layout using the CSS Multi-column Layout Module.

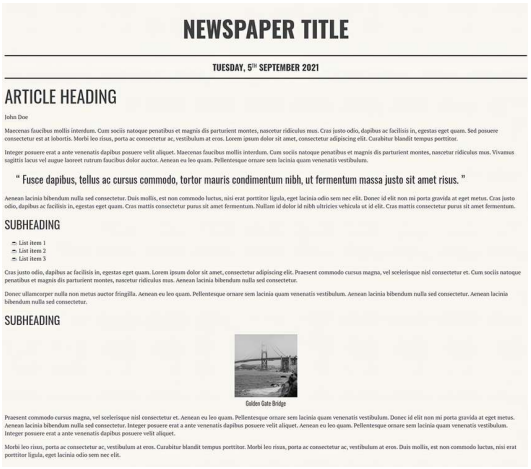

Figure 4.17 Progress thus far, including styled figure and image caption

.4 Using the CSS Multi-column Layout Module

The CSS Multi-column Layout Module is perhaps less known than Grid and Flexbox as a way to present content, but it's no less useful. The purpose of this module is to allow content to flow naturally between multiple columns. It works similarly to the way we create multiple column layouts in a Microsoft Word or Google Docs document. We assign columns to a section of content, and the content naturally flows from one column to another. Because we want our content to be placed in columns only on wider screens, we'll use a media query to apply our columns conditionally only after the window reaches a particular size.

4.4.1 Creating media queries

A *media query* is a type of at-rule; we looked at it briefly in chapter 2 when we changed our grid layout to depend on the width of the screen. Like `@counter-style`, which we used earlier in this chapter, it starts with an at (`@`) symbol followed by the identifier `media`. Then we set the instruction about what to do when the rules inside the media query apply. We want to place the content in columns when our window width is greater than or equal to 955 pixels. Therefore, our media query will be `@media(min-width: 955px) {}`. Figure 4.18 breaks down the individual pieces of the query. Inside the media query, we'll define our columns.

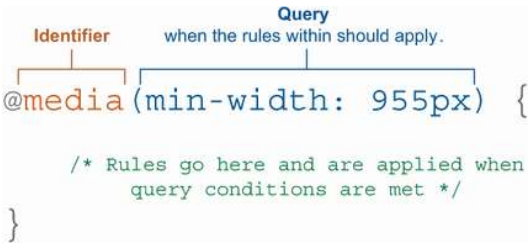

Figure 4.18 Media-query breakdown

4.4.2 Defining and styling columns

There are two ways we can define how the columns are created:

- *Dictate a column width*. The browser will create as many columns of that width as it can in the available space.

- *Dictate how many columns we want.* The browser will fit that number of equal-size columns in the available space.

We'll go with the second option because we already know that we want to create three columns. We specifically target the article, and using the `column-count` property, we set our quantity to `3`, as shown in the following listing.

Listing 4.11 Conditionally breaking an article into three columns based on screen width

```
@media(min-width: 955px) {      ①
  article {
    column-count: 3;            ②
  }
}
```

① Media query

② Sets how many columns we want

Figure 4.19 shows our article laid out in three columns using the CSS from listing 4.11.

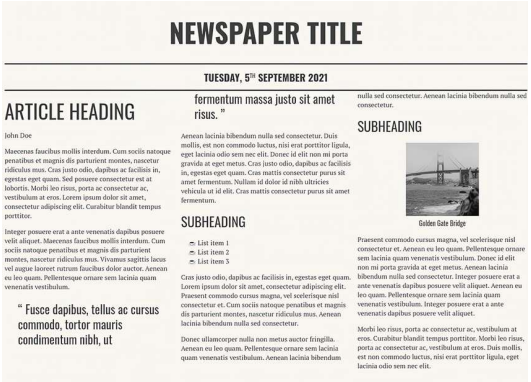

Figure 4.19 Three-column layout

Next, we'll adjust the spacing between columns and add vertical lines between them. Let's start with the vertical lines.

4.4.3 Using the column-rule property

To create a clear separation between our columns, we'll add a vertical line between them, using the `column-rule` property. As with borders and outlines, we need to set a line type, width, and color. To keep our line work consistent, we'll use the same color and style of line that we set for the borders above and below the date at the top of the page. We'll make the lines slightly narrower, however.

The lines at the top of the screen separate content types (title, date, and article). Here, we're within the same content type. We add the lines to make visual separation of the columns easier; we don't want to break up the content. We want the lines to be less prominent, so we'll make them thinner.

To create the lines, we add `column-rule: 2px solid #333333;` to the existing article rule inside the media query. Now our article looks like figure 4.20.

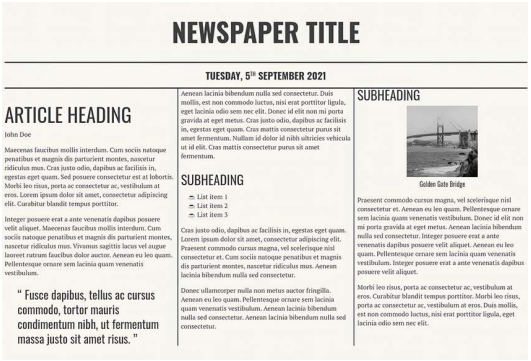

Figure 4.20 Columns with added vertical lines

With our lines in place, we see that we have some crowding between the article itself and the date and that we could use a bit more space between our lines and our text.

4.4.4 Adjusting spacing with the column-gap property

Now we need to do two things: increase the container spacing between the date of the article and the body of the article, and increase the gap between columns within the article. To adjust the spacing between the article and the date, we'll add `36px` of margin to the top of the article. Because working out a value to use isn't an absolute science, sometimes we need a bit of trial and error to determine what will look right on the page. We want to create enough room that each item has its own space and is clear, but not so much room that the items are too far apart and look separated.

Gestalt design principles

The *Gestalt principles* of design are a collection of principles of human perception that describe how humans group similar elements. One of the seven principles is proximity, which talks about how things that are close together appear to be more related than things that are spaced farther apart. For more information about the Gestalt principles, see **http://mng.bz/0yNv**.
With the space between the article and the date handled, let's turn our attention to the space between the columns. To add a gap between our vertical lines and our text, we'll use the `column-gap` property, which defines the amount of whitespace we want to have between our columns. We will set ours to `42px;` .

We continue to add these styles inside the media query as shown in listing 4.12 because we want them to apply only when our layout is columned. We don't want these style changes to apply to narrower screens.

Listing 4.12 Updated media query and article rule

```
@media (min-width: 955px) {
  article {
    column-count: 3;
    column-rule: 2px solid #333333;
    column-gap: 42px;
    margin-top: 36px;
  }
}
```

With these adjustments made (figure 4.21), let's turn our attention to the quote.

Figure 4.21 Layout with adjusted spacing

Earlier in this chapter, we styled the block quote so that it would stand out. But now that we have a multi-column format, it gets a little lost in the other visual elements on the page. Let's make it span multiple columns to make it pop.

4.4.5 Making content span multiple columns

We can make elements span multiple columns by using the `column-span` property. Our choices of values are `all` and `none`. Because we want the quote to go across the entire page, we'll choose `all`. Inside our media query, we'll add the following rule: `blockquote { column-span: all }`. This rule results in the layout shown in figure 4.22.

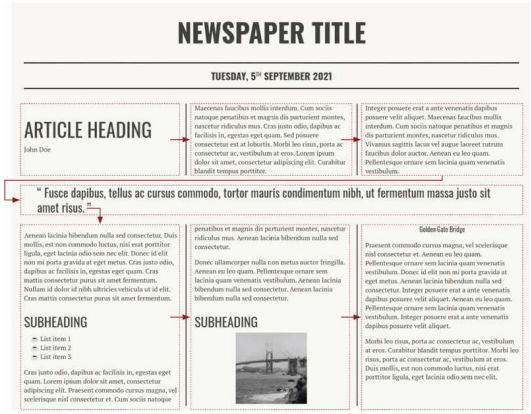

Figure 4.22 Content reflow due to spanning the `blockquote` across the columns

Notice that the flow of the content has changed. We added arrows to show the new flow introduced by making the quote span the screen. Instead of flowing the entire article from top left to bottom right, evenly distributed across the columns, we added `column-span: all` to the quote, so content that's before the quote now flows from top left to top right across the page above the quote. The content after the quote does the same. As a result of spanning content, we changed the flow of the text through our columns.

When we look at the content flow, we notice that the caption and the image have been split across two columns, which isn't ideal. Let's prevent that from happening.

4.4.6 Controlling content breaks

To prevent the image and its caption from ending up in different columns, we can use the `break-inside` property with the keyword value `avoid` , which we set on the

`<figure>` element. With this declaration, we inform the browser that when it's generating the columns, the contents of the element should stay together as a unit, not be split across multiple columns. In other words, the image and figure caption should remain together. The rule we add to the media query is `figure { break-inside: avoid }`. Figure 4.23 shows the resulting output.

Figure 4.23 Keeping the image and caption together

.5 Adding the finishing touches

With our content flowing the way we want it across the columns, let's polish some final details. One of the hallmarks of newspaper layouts is that the text is often justified.

4.5.1 Justifying and hyphenating text

Justification refers to the alignment of the lines inside a body of text, as illustrated in figure 4.24. When text is *justified*, the lines of text start and end at the same spot, forming a box. By contrast, text that is left-aligned has ragged ends.

Figure 4.24 Text justification

Let's justify our paragraph text. To do this, we'll use the `text-align` property and give it a value of `justify`. To make the lines equal in length, we'll distribute extra space across the line. We can tune how the space is redistributed by using the `text-justify` property. If we don't set a `text-justify` value, the browser will choose what it thinks is best for the situation. We have a fluid design; it grows and shrinks with the window size. What is best may be different based on the window size, so we'll let the browser decide what will work best.

We'll add some hyphens, however. By default, browsers don't hyphenate a word at the end of a line; they simply continue to the next line. We can alter this behavior by setting the `hyphens` property to `auto`. Allowing the browser to hyphenate words at the end of lines will help diminish the amount of whitespace that's needed between our words to justify the text.

Listing 4.13 shows our paragraph rule. We continue to include our updates inside our media query, as these changes are relevant only

when we switch to the columns layout.

Listing 4.13 Justifying paragraph text

```
@media (min-width: 955px) {
  ...
  p {
    text-align: justify;
    hyphens: auto;
  }
}
```

Now our paragraphs look like those in figure 4.25.

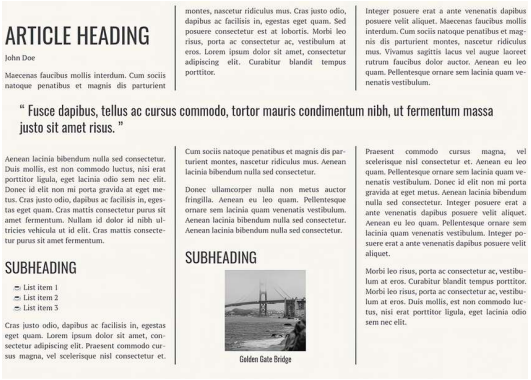

Figure 4.25 Justified and hyphenated paragraph text

As we look at our layout, we notice that the image at the bottom of the second column looks a little odd and out of place. Let's fix that.

4.5.2 Wrapping the text around the image

To reconnect the image with the subsequent text, we'll push the image and its caption to the left and have the text wrap around the image. To create this effect, we'll use the `float` property. Applying the

`float` property to an element pushes it to the left or the right, allowing text and inline elements to wrap around it.

In this situation, having the image and caption as a unit inside a `<figure>` element comes in handy for styling. Because both items are contained in the `<figure>`, we'll apply `float` to the figure, neatly wrapping the text around both the image and the caption.

Listing 4.14 shows how we float the figure. Notice that we added a right margin to the figure. Because we are floating the figure to the left, it places itself on the left side of the column, allowing the text to wrap around it in the leftover space to the right, as shown in figure 4.26. The right margin creates a space between the image and the text so that the text doesn't come right up against the edge of the image.

Listing 4.14 Floating the figure

```
@media (min-width: 955px) {
  ...
  figure {
    float: left;
    margin-right: 24px;
  }
}
```

Figure 4.26 Floated image

As you'll see in chapter 7, we can do a lot more cool things with floating images. For now, though, let's focus on our newspaper page. The last thing we'll address is handling how the page behaves in an extremely wide window.

4.5.3 Using max-width and a margin value of auto

Figure 4.26 shows that our layout starts to degrade as the window gets extremely wide. The wider the window, the worse the problem gets. More and more users have extra-wide screens, so we need to consider what would happen if they have the window maximized, taking up the entire screen. To handle this use case, we'll use the same trick that we used for the loader in chapter 2. We'll set a maximum width for our layout and then set its left and right margins to `auto`, which will center the container horizontally when the window is larger than our maximum width.

For our page, our container is the body, so we'll give our `body` a `max-width` of `1200px` and set our left and right margins to `auto`. We also

need to move the `background-color` from being set on the `body` to being set on the `html` element rule; otherwise, when our screen is wider than 1,200 pixels, we'll end up with a white band to the left and right sides of our page.

These changes won't go inside the media query. We'll edit the styles we set on the `body` at the beginning of this chapter and add an `html` rule to set the background color. The following listing shows our changes.

Listing 4.15 Changes to the `body` and `html` elements

```
html { background-color: #f9f7f1 }

body {
  background-color: #f9f7f1;        ①
  font-family: 'PT Serif', serif;
  color: #404040;
  padding: 0 24px;
  max-width: 1200px;                ②
  margin: 0 auto;                   ③
}
```

① Moves the background color from the body rule to the html rule

② Sets the maximum width our page can become

③ Centers the page

With these final changes, we have a page that works for both mobile and desktop users. Figure 4.27 shows our finished layout.

Figure 4.27 Finished layout

Summary

- A theme is the general look and feel that we maintain throughout an application.
- We may need to import our fonts, as few fonts are universally available. Because no officially defined list of web-safe fonts exists, we should always use a keyword fallback.
- Creating a visual hierarchy will help our users orient themselves to the page and identify important information.
- We can control which symbols the browser uses when it's instructed to display quotation marks.
- We can customize the way our lists display their bullets by using the `counter-style` at-rule.
- Filters allow us to alter the appearance of an image.
- We can create multicolumn layouts by using the CSS Multi-column Layout Module.
- We can make content span all the columns when creating multicolumn layouts.

- We can make the browser use hyphens to break words at the end of lines.
- Floating allows us to wrap text around an element.