

FUTURE VISION BIE

One Stop for All Study Materials
& Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : <https://hemanthrajhemu.github.io>

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

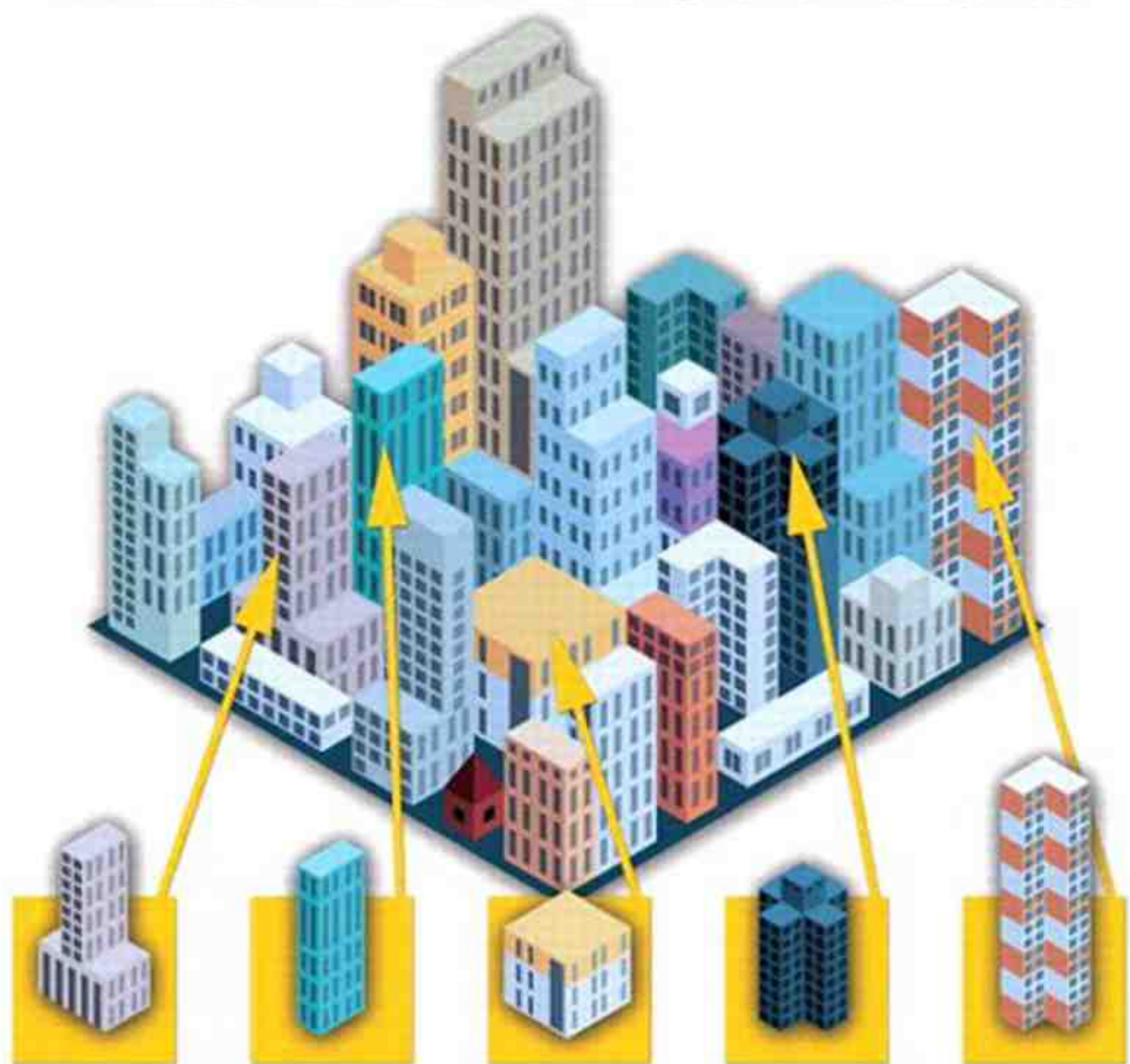
Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: <https://bit.ly/FVBIESHARE>

FUNDAMENTALS of WEB DEVELOPMENT



RANDY CONNOLLY • RICARDO HOAR

Review Questions	141
Hands-On Practice	142
References	147

Chapter 4 **HTML Tables and Forms** 148

4.1	Introducing Tables	149
	Basic Table Structure	149
	Spanning Rows and Columns	150
	Additional Table Elements	151
	Using Tables for Layout	152
4.2	Styling Tables	155
	Table Borders	155
	Boxes and Zebras	156
4.3	Introducing Forms	158
	Form Structure	159
	How Forms Work	160
	Query Strings	161
	The <form> Element	162
4.4	Form Control Elements	163
	Text Input Controls	165
	Choice Controls	167
	Button Controls	169
	Specialized Controls	171
	Date and Time Controls	172
4.5	Table and Form Accessibility	174
	Accessible Tables	175
	Accessible Forms	176
4.6	Microformats	177
4.7	Chapter Summary	178
	Key Terms	179

Review Questions	179
Hands-On Practice	180

Chapter 5 **Advanced CSS: Layout** 184

5.1	Normal Flow	185
5.2	Positioning Elements	188
	Relative Positioning	188
	Absolute Positioning	189
	Z-Index	190
	Fixed Position	191
5.3	Floating Elements	193
	Floating within a Container	193
	Floating Multiple Items Side by Side	195
	Containing Floats	198
	Overlaying and Hiding Elements	199
5.4	Constructing Multicolumn Layouts	203
	Using Floats to Create Columns	204
	Using Positioning to Create Columns	207
5.5	Approaches to CSS Layout	209
	Fixed Layout	210
	Liquid Layout	211
	Other Layout Approaches	213
5.6	Responsive Design	214
	Setting Viewports	215
	Media Queries	218
5.7	CSS Frameworks	220
	Grid Systems	220
	CSS Preprocessors	222
5.8	Chapter Summary	225
	Key Terms	225

4 HTML Tables and Forms

CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What HTML tables are and how to create them
- How to use CSS to style tables
- What forms are and how they work
- What the different form controls are and how to use them
- How to improve the accessibility of your websites
- What microformats are and how we use them

This chapter covers the key remaining HTML topics. The first of these topics is HTML tables; the second topic is HTML forms. Tables and forms often have a variety of accessibility issues, so this chapter also covers accessibility in more detail. Finally, the chapter covers microformats and microdata, which are a way to add semantic information to web pages.

4.1 Introducing Tables

A **table** in HTML is created using the `<table>` element and can be used to represent information that exists in a two-dimensional grid. Tables can be used to display calendars, financial data, pricing tables, and many other types of data. Just like a real-world table, an HTML table can contain any type of data: not just numbers, but text, images, forms, even other tables, as shown in Figure 4.1.

4.1.1 Basic Table Structure

To begin we will examine the HTML needed to implement the following table.

The Death of Marat	Jacques-Louis David	1793	162 cm	128 cm
Burial at Ornans	Gustave Courbet	1849	314 cm	663 cm

As can be seen in Figure 4.2, an HTML `<table>` contains any number of rows (`<tr>`); each row contains any number of table data cells (`<td>`). The indenting shown in Figure 4.2 is purely a convention to make the markup more readable by humans.

As can be seen in Figure 4.2, some browsers do not by default display borders for the table; however, we can do so via CSS.



HANDS-ON EXERCISES

LAB 4 EXERCISE

Create a Basic Table
Complex Content in Tables

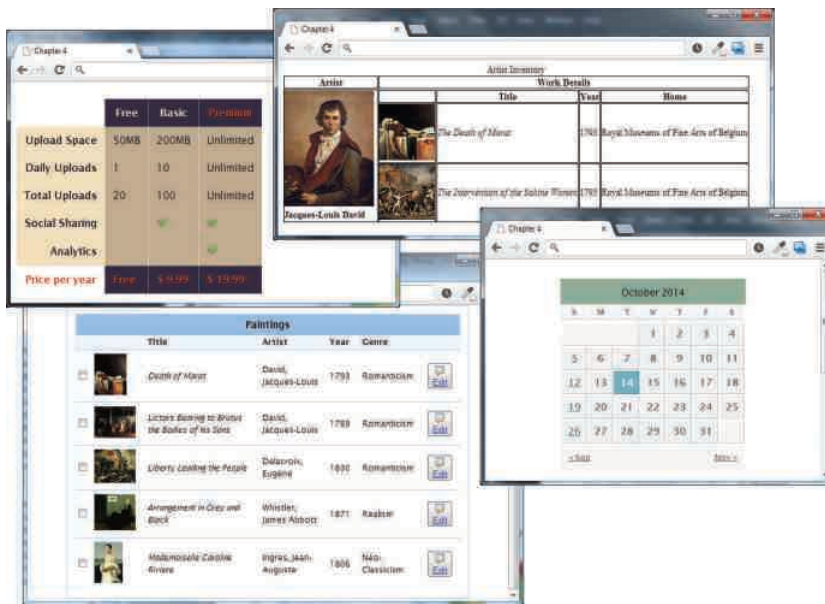


FIGURE 4.1 Examples of tables

<code><tr></code>	<code><td></code>	<code><td></code>	<code><td></code>	<code><td></code>	<code><td></code>
	The Death of Marat	Jacques-Louis David	1793	162cm	128cm
<code><tr></code>	<code><td></code>	<code><td></code>	<code><td></code>	<code><td></code>	<code><td></code>
	Burial at Ornans	Gustave Courbet	1849	314cm	663cm
	<code><td></code>	<code><td></code>	<code><td></code>	<code><td></code>	<code><td></code>

```

<table>
<tr>
  <td>The Death of Marat</td>
  <td>Jacques-Louis David</td>
  <td>1793</td>
  <td>162cm</td>
  <td>128cm</td>
</tr>
<tr>
  <td>Burial at Ornans</td>
  <td>Gustave Courbet</td>
  <td>1849</td>
  <td>314cm</td>
  <td>663cm</td>
</tr>
</table>

```

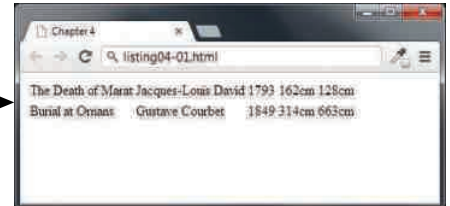


FIGURE 4.2 Basic table structure

Many tables will contain some type of headings in the first row. In HTML, you indicate header data by using the `<th>` instead of the `<td>` element, as shown in Figure 4.3. Browsers tend to make the content within a `<th>` element bold, but you could style it anyway you would like via CSS.

The main reason you should use the `<th>` element is not, however, due to presentation reasons. Rather, you should also use the `<th>` element for accessibility reasons (it helps those using screen readers, which we will cover in more detail later in this chapter) and for search engine optimization reasons.

4.1.2 Spanning Rows and Columns

So far, you have learned two key things about tables. The first is that all content must appear within the `<td>` or `<th>` container. The second is that each row must have the same number of `<td>` or `<th>` containers. There is a way to change this second behavior. If you want a given cell to cover several columns or rows, then you can do so by using the `colspan` or `rowspan` attributes (Figure 4.4).

Spanning rows is a little less common and perhaps a little more complicated because the `rowspan` affects the cell content in multiple rows, as can be seen in Figure 4.5.



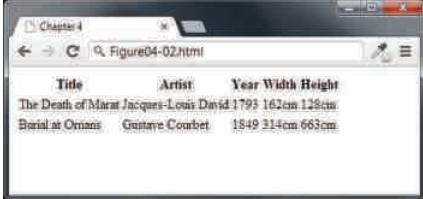
HANDS-ON EXERCISES

LAB 4 EXERCISE Spanning Rows and Columns


```
<table>
```

Title	Artist	Year	Width	Height
The Death of Marat	Jacques-Louis David	1793	162cm	128cm
Burial at Ornans	Gustave Courbet	1849	314cm	663cm

```
<table>
  <tr>
    <th>Title</th>
    <th>Artist</th>
    <th>Year</th>
    <th>Width</th>
    <th>Height</th>
  </tr>
  <tr>
    <td>The Death of Marat</td>
    <td>Jacques-Louis David</td>
    <td>1793</td>
    <td>162cm</td>
    <td>128cm</td>
  </tr>
  <tr>
    <td>Burial at Ornans</td>
    <td>Gustave Courbet</td>
    <td>1849</td>
    <td>314cm</td>
    <td>663cm</td>
  </tr>
</table>
```



Title	Artist	Year	Width	Height
The Death of Marat	Jacques-Louis David	1793	162cm	128cm
Burial at Ornans	Gustave Courbet	1849	314cm	663cm

FIGURE 4.3 Adding table headings

4.1.3 Additional Table Elements

While the previous sections cover the basic elements and attributes for most simple tables, there are some additional table elements that can add additional meaning and accessibility to one's tables.

Figure 4.6 illustrates these additional (and optional) table elements.

The `<caption>` element is used to provide a brief title or description of the table, which improves the accessibility of the table, and is strongly recommended. You can use the `caption-side` CSS property to change the position of the caption below the table.

The `<thead>`, `<tfoot>`, and `<tbody>` elements tend in practice to be used quite infrequently. However, they do make some sense for tables with a large number of rows. With CSS, one could set the `height` and `overflow` properties of the `<tbody>`



HANDS-ON EXERCISES

LAB 4 EXERCISE

Alternate Table Structure Elements

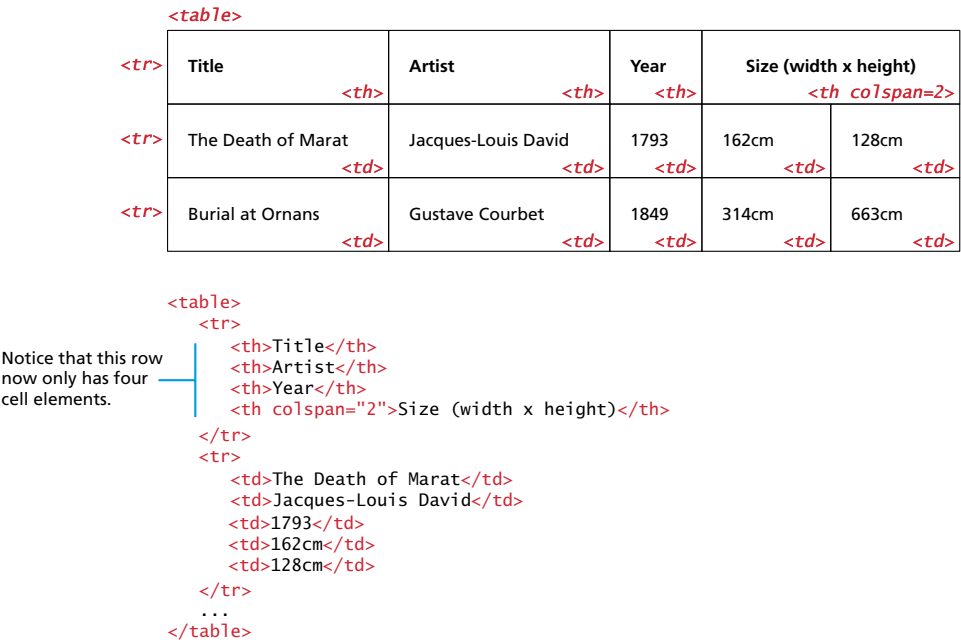


FIGURE 4.4 Spanning columns

element so that its content scrolls, while the header and footer of the table remain always on screen.

The `<col>` and `<colgroup>` elements are also mainly used to aid in the eventual styling of the table. Rather than styling each column, you can style all columns within a `<colgroup>` with just a single style. Unfortunately, the only properties you can set via these two elements are borders, backgrounds, width, and visibility, and only if they are not overridden in a `<td>`, `<th>`, or `<tr>` element (which, because they are more specific, will override any style settings for `<col>` or `<colgroup>`). As a consequence, they tend to not be used very often.

4.1.4 Using Tables for Layout

Prior to the broad support for CSS in browsers, HTML tables were frequently used to create page layouts. Since HTML block-level elements exist on their own line, tables were embraced by developers in the 1990s as a way to get block-level HTML elements to sit side by side on the same line. Figure 4.7 illustrates a typical example of how tables were used for layout. The first image shows the layout as the user would see it; the second has borders turned on so that you can see the embedded table within the first table. It was not uncommon for a complex layout to have dozens of embedded tables.

`<table>`

Artist	Title	Year
Jacques-Louis David	The Death of Marat	1793
	The Intervention of the Sabine Women	1799
	Napoleon Crossing the Alps	1800

```

<table>
  <tr>
    <th>Artist</th>
    <th>Title</th>
    <th>Year</th>
  </tr>
  <tr>
    <td rowspan="3">Jacques-Louis David</td>
    <td>The Death of Marat</td>
    <td>1793</td>
  </tr>
  <tr>
    <td>The Intervention of the Sabine Women</td>
    <td>1799</td>
  </tr>
  <tr>
    <td>Napoleon Crossing the Alps</td>
    <td>1800</td>
  </tr>
  ...
</table>

```

Notice that these two rows now only have two cell elements.

FIGURE 4.5 Spanning rows

Unfortunately, this practice of using tables for layout had some problems. The first of these problems is that this approach tended to dramatically increase the size of the HTML document. As you can see in Figure 4.7, the large number of extra tags required for `<table>` elements can significantly bloat the HTML document. These larger files take longer to download, but more importantly, were often more difficult to maintain because of the extra markup.

A second problem with using tables for markup is that the resulting markup is not semantic. Tables are meant to indicate tabular data; using `<table>` elements simply to get two block-elements side by side is an example of using markup simply for presentation reasons.

A title for the table is good for accessibility.	<code><table></code>
	<code><caption>19th Century French Paintings</caption></code>
	<code><col class="artistName" /></code>
	<code><colgroup id="paintingColumns"></code>
	<code><col /></code>
	<code><col /></code>
These describe our columns, and can be used to aid in styling.	<code></colgroup></code>
	<code><thead></code>
	<code><tr></code>
	<code><th>Title</th></code>
	<code><th>Artist</th></code>
	<code><th>Year</th></code>
Table header could potentially also include other <code><tr></code> elements.	<code></tr></code>
	<code></thead></code>
	<code><tfoot></code>
	<code><tr></code>
	<code><td colspan="2">Total Number of Paintings</td></code>
	<code><td>2</td></code>
Yes, the table footer comes <i>before</i> the body.	<code></tr></code>
	<code></tfoot></code>
	<code><tbody></code>
	<code><tr></code>
	<code><td>The Death of Marat</td></code>
	<code><td>Jacques-Louis David</td></code>
	<code><td>1793</td></code>
Potentially, with styling the browser can scroll this information, while keeping the header and footer fixed in place.	<code></tr></code>
	<code><tr></code>
	<code><td>Burial at Ornans</td></code>
	<code><td>Gustave Courbet</td></code>
	<code><td>1849</td></code>
	<code></tr></code>
	<code></tbody></code>
	<code></table></code>



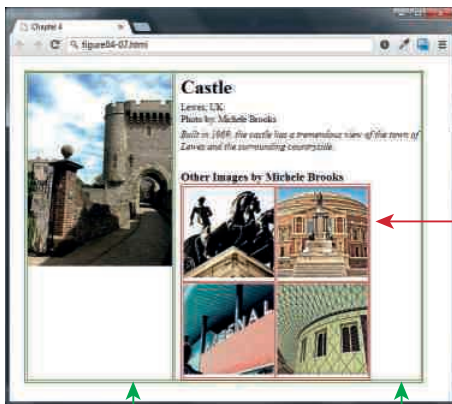
FIGURE 4.6 Additional table elements

The other key problem is that using tables for layout results in a page that is not accessible, meaning that for users who rely on software to voice the content, table-based layouts can be extremely uncomfortable and confusing to understand.

It is much better to use CSS for layout. The next chapter will examine how to use CSS for layout purposes. Unfortunately, as we will discover, the CSS required to create complicated (and even relatively simple) layouts is not exactly easy and intuitive. For this reason, many developers still continue to use tables for layout, though it is a practice that this book strongly discourages.



```
<table>
<tr>
  <td>
    
  </td>
</tr>
<tr>
  <td>
    <h2>Castle</h2>
    <p>Lewes, UK</p>
    <p>Photo by: Michele Brooks</p>
    <p>Built in 1069, the castle has a tremendous
      view of the town of Lewes and the
      surrounding countryside.
    </p>
  </td>
</tr>
<tr>
  <td>
    <h3>Other Images by Michele Brooks</h3>
    <table>
      <tr>
        <td></td>
        <td></td>
      </tr>
      <tr>
        <td></td>
        <td></td>
      </tr>
    </table>
  </td>
</tr>
</table>
```



```
<table>
  <tr>
    <td></td>
    <td></td>
  </tr>
  <tr>
    <td></td>
    <td></td>
  </tr>
</table>
</td>
</tr>
</table>
```

FIGURE 4.7 Example of using tables for layout

4.2 Styling Tables

There is certainly no limit to the way one can style a table. While most of the styling that one can do within a table is simply a matter of using the CSS properties from Chapter 3, there are a few properties unique to styling tables that you have not yet seen.

4.2.1 Table Borders

As can be seen in Figure 4.8, borders can be assigned to both the `<table>` and the `<td>` element (they can also be assigned to the `<th>` element as well).

**NOTE**

While now officially deprecated in HTML5, there are a number of table attributes that are still supported by the browsers and which you may find in legacy markup. These include the following attributes:

- `width`, `height`—for setting the width and height of cells
- `cellspacing`—for adding space between every cell in the table
- `cellpadding`—for adding space between the content of the cell and its border
- `bgcolor`—for changing the background color of any table element
- `background`—for adding a background image to any table element
- `align`—for indicating the alignment of a table in relation to the surrounding container

You should avoid using these attributes for new markup and instead use the appropriate CSS properties instead.

Interestingly, borders cannot be assigned to the `<tr>`, `<thead>`, `<tfoot>`, and `<tbody>` elements.

Notice as well the `border-collapse` property. This property selects the table's border model. The default, shown in the second screen capture in Figure 4.8, is the separated model or value. In this approach, each cell has its own unique borders. You can adjust the space between these adjacent borders via the `border-spacing` property, as shown in the final screen capture in Figure 4.8. In the third screen capture, the collapsed border model is being used; in this model adjacent cells share a single border.

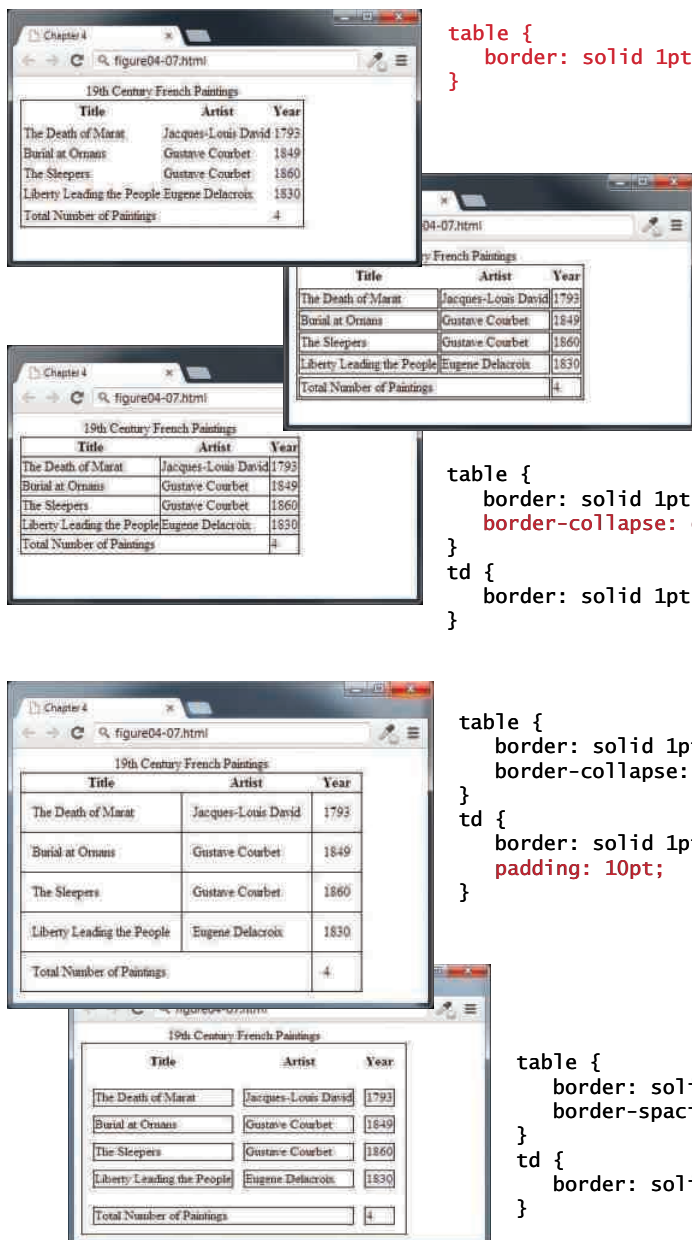
4.2.2 Boxes and Zebras

While there is almost no end to the different ways one can style a table, there are a number of pretty common approaches. We will look at two of them here. The first of these is a box format, in which we simply apply background colors and borders in various ways, as shown in Figure 4.9.

We can then add special styling to the `:hover` pseudo-class of the `<tr>` element, to highlight a row when the mouse cursor hovers over a cell, as shown in Figure 4.10. That figure also illustrates how the pseudo-element `nth-child` (covered in Chapter 3) can be used to alternate the format of every second row.

**HANDS-ON EXERCISES**

LAB 4 EXERCISE
Simple Table Styling
CSS Table Styling



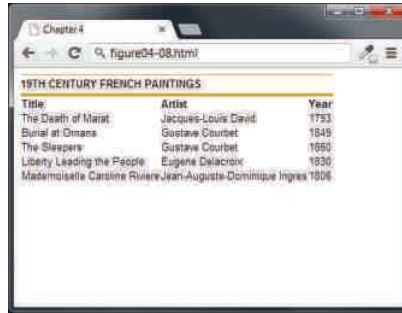
The figure shows five browser windows displaying a table titled "19th Century French Paintings". The table has three columns: Title, Artist, and Year. The data rows are:

Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Total Number of Paintings	4	

The styling evolution is as follows:

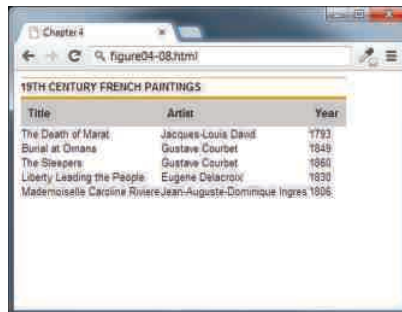
- First screenshot:** Basic table with no borders.
- Second screenshot:** Table with `border: solid 1pt black;` applied to the `table` tag.
- Third screenshot:** Table with `border: solid 1pt black;` and `border-collapse: collapse;` applied to the `table` tag, and `border: solid 1pt black;` applied to the `td` tags.
- Fourth screenshot:** Table with `border: solid 1pt black;` and `border-collapse: collapse;` applied to the `table` tag, and `border: solid 1pt black;` and `padding: 10pt;` applied to the `td` tags.
- Fifth screenshot:** Table with `border: solid 1pt black;` and `border-collapse: collapse;` applied to the `table` tag, and `border: solid 1pt black;` and `border-spacing: 10pt;` applied to the `td` tags.

FIGURE 4.8 Styling table borders



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
table {
    font-size: 0.8em;
    font-family: Arial, Helvetica, sans-serif;
    border-collapse: collapse;
    border-top: 4px solid #DCA806;
    border-bottom: 1px solid white;
    text-align: left;
}
caption {
    font-weight: bold;
    padding: 0.25em 0 0.25em 0;
    text-align: left;
    text-transform: uppercase;
    border-top: 1px solid #DCA806;
}
```



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
thead tr {
    background-color: #CACACA;
}
th {
    padding: 0.75em;
}
```



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sleepers	Gustave Courbet	1860
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
tbody tr {
    background-color: #F1F1F1;
    border-bottom: 1px solid white;
    color: #6E6E6E;
}
tbody td {
    padding: 0.75em;
}
```

FIGURE 4.9 An example boxed table

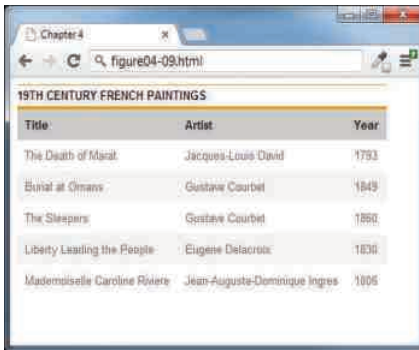
4.3 Introducing Forms

Forms provide the user with an alternative way to interact with a web server. Up to now, clicking hyperlinks was the only mechanism available to the user for communicating with the server. Forms provide a much richer mechanism. Using a form, the user can enter text, choose items from lists, and click buttons. Typically, programs running on the server will take the input from HTML forms and do something with



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sheepers	Gustave Courbet	1866
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
tbody tr:hover {
    background-color: #9e9e9e;
    color: black;
}
```



Title	Artist	Year
The Death of Marat	Jacques-Louis David	1793
Burial at Ornans	Gustave Courbet	1849
The Sheepers	Gustave Courbet	1866
Liberty Leading the People	Eugene Delacroix	1830
Mademoiselle Caroline Riviere	Jean-Auguste-Dominique Ingres	1806

```
tbody tr:nth-child(odd) {
    background-color: white;
}
```

FIGURE 4.10 Hover effect and zebra stripes

it, such as save it in a database, interact with an external web service, or customize subsequent HTML based on that input.

Prior to HTML5, there were a limited number of data-entry controls available in HTML forms. There were controls for entering text, controls for choosing from a list, buttons, checkboxes, and radio buttons. HTML5 has added a number of new controls as well as more customization options for the existing controls.

4.3.1 Form Structure

A form is constructed in HTML in the same manner as tables or lists: that is, using special HTML elements. Figure 4.11 illustrates a typical HTML form.

Notice that a form is defined by a `<form>` element, which is a container for other elements that represent the various input elements within the form as well as plain text and almost any other HTML element. The meaning of the various attributes shown in Figure 4.11 is described below.



**HANDS-ON
EXERCISES**

LAB 4 EXERCISE
Creating a Form

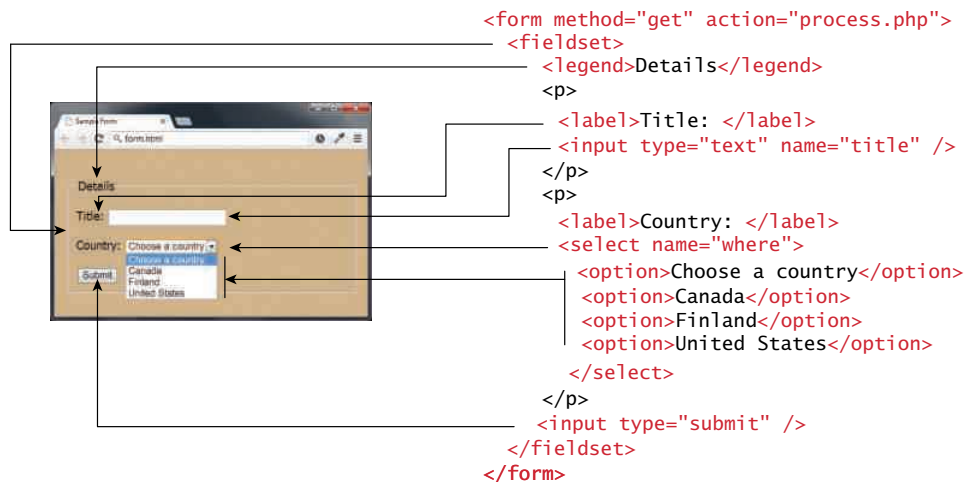


FIGURE 4.11 Sample HTML form

**NOTE**

While a form can contain most other HTML elements, a form **cannot** contain another `<form>` element.

4.3.2 How Forms Work

While forms are constructed with HTML elements, a form also requires some type of server-side resource that processes the user's form input as shown in Figure 4.12.

The process begins with a request for an HTML page that contains some type of form on it. This could be something as complex as a user registration form or as simple as a search box. After the user fills out the form, there needs to be some mechanism for submitting the form data back to the server. This is typically achieved via a submit button, but through JavaScript, it is possible to submit form data using some other type of mechanism.

Because interaction between the browser and the web server is governed by the HTTP protocol, the form data must be sent to the server via a standard HTTP request. This request is typically some type of server-side program that will process the form data in some way; this could include checking it for validity, storing it in a database, or sending it in an email. In Chapters 8 and 9, you will learn how to write PHP scripts to process form input. In the remainder of this chapter, you will learn only how to construct the user interface of forms through HTML.

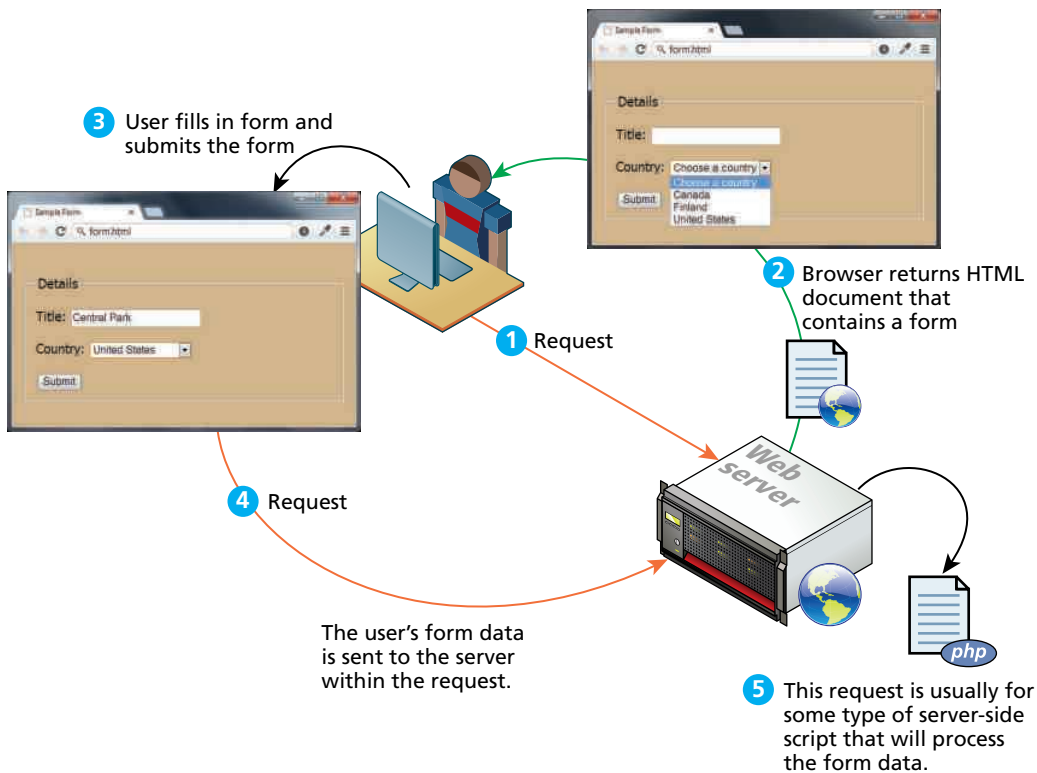


FIGURE 4.12 How forms work

4.3.3 Query Strings

You may be wondering how the browser “sends” the data to the server. As mentioned already, this occurs via an HTTP request. But how is the data packaged in a request?

The browser packages the user’s data input into something called a query string. A **query string** is a series of name=value pairs separated by ampersands (the & character). In the example shown in Figure 4.12, the names in the query string were defined by the HTML form (see Figure 4.11); each form element (i.e., the first <input> elements and the <select> element) contains a name attribute, which is used to define the name for the form data in the query string. The values in the query string are the data entered by the user.

Figure 4.13 illustrates how the form data (and its connection to form elements) is packaged into a query string.

Query strings have certain rules defined by the HTTP protocol. Certain characters such as spaces, punctuation symbols, and foreign characters cannot be part of

```
<input type="text" name="title" />
```



```
title=Central+Park&where=United+States
```

```
<select name="where">
```

FIGURE 4.13 Query string data and its connection to the form elements

a query string. Instead, such special symbols must be **URL encoded** (also called **percent encoded**), as shown in Figure 4.14.



HANDS-ON EXERCISES

LAB 4 EXERCISE Testing a Form

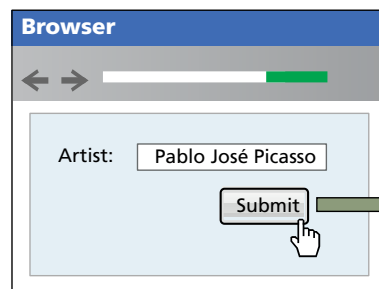
4.3.4 The `<form>` Element

The example HTML form shown in Figure 4.11 contains two important attributes that are essential features of any form, namely the `action` and the `method` attributes.

The `action` attribute specifies the URL of the server-side resource that will process the form data. This could be a resource on the same server as the form or a completely different server. In this example (and of course in this book as well), we will be using PHP pages to process the form data. There are other server technologies, each with their own extensions, such as ASP.NET (`.aspx`), ASP (`.asp`), and Java Server Pages (`.jsp`). Some server setups, it should be noted, hide the extension of their server-side programs.

The `method` attribute specifies how the query string data will be transmitted from the browser to the server. There are two possibilities: GET and POST.

What is the difference between GET and POST? The difference resides in where the browser locates the user's form input in the subsequent HTTP request. With



Notice how the spaces and the accented é are URL encoded (in red).

```
artist=Pablo+Jos%E9+Picasso
```

URL Encoding

FIGURE 4.14 URL encoding

GET, the browser locates the data in the URL of the request; with **POST**, the form data is located in the HTTP header after the HTTP variables. Figure 4.15 illustrates how the two methods differ.

Which of these two methods should one use? Table 4.1 lists the key advantages and disadvantages of each method.

Generally, form data is sent using the POST method. However, the GET method is useful when you are testing or developing a system, since you can examine the query string directly in the browser's address bar. Since the GET method uses the URL to transmit the query string, form data will be saved when the user bookmarks a page, which may be desirable, but is generally a potential security risk for shared use computers. And needless to say, any time passwords are being transmitted, they should be transmitted via the POST method.

4.4 Form Control Elements

Despite the wide range of different form input types in HTML5, there are only a relatively small number of form-related HTML elements, as shown in Table 4.2. This section will examine how these elements are typically used.

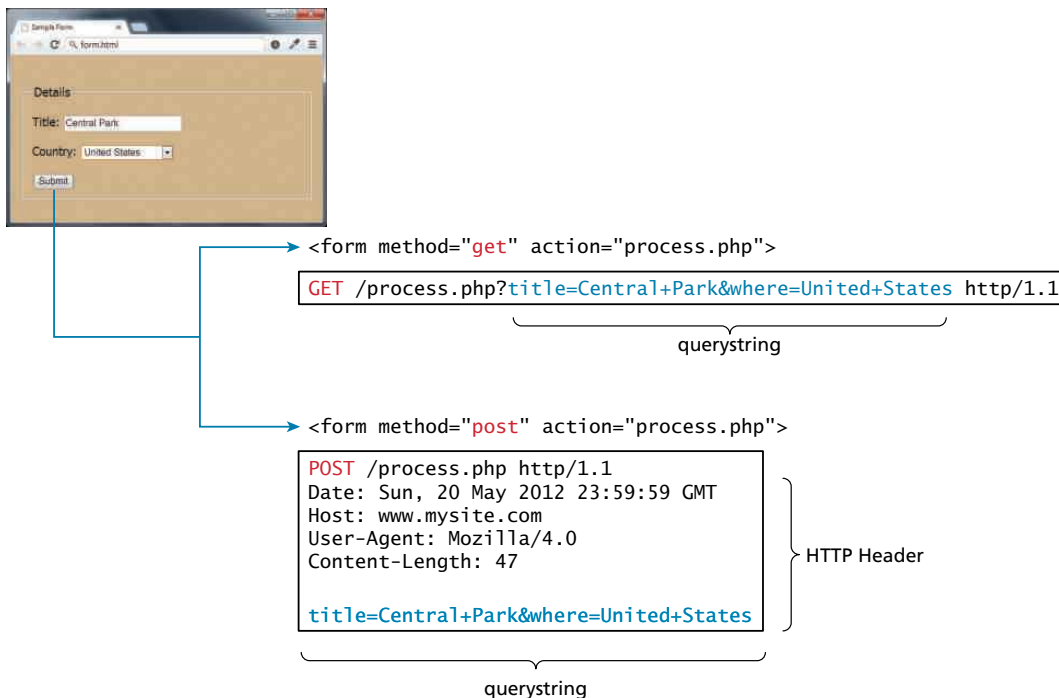


FIGURE 4.15 GET versus POST

Type	Advantages and Disadvantages
GET	<p>Data can be clearly seen in the address bar. This may be an advantage during development but a disadvantage in production.</p> <p>Data remains in browser history and cache. Again this may be beneficial to some users, but a security risk on public computers.</p> <p>Data can be bookmarked (also an advantage and a disadvantage).</p> <p>Limit on the number of characters in the form data returned.</p>
POST	<p>Data can contain binary data.</p> <p>Data is hidden from user.</p> <p>Submitted data is not stored in cache, history, or bookmarks.</p>

TABLE 4.1 GET versus POST

**PRO TIP**

Query strings can make a URL quite long. While the HTTP protocol does not specify a limit to the size of a query string, browsers and servers do impose practical limitations. For instance, the maximum length of a URL for Internet Explorer is 2083 characters, while the Apache web server limits URL lengths to 4000 characters.

Type	Description
<button>	Defines a clickable button.
<datalist>	An HTML5 element that defines lists of pre-defined values to use with input fields.
<fieldset>	Groups related elements in a form together.
<form>	Defines the form container.
<input>	Defines an input field. HTML5 defines over 20 different types of input.
<label>	Defines a label for a form input element.
<legend>	Defines the label for a fieldset group.
<option>	Defines an option in a multi-item list.
<optgroup>	Defines a group of related options in a multi-item list.
<select>	Defines a multi-item list.
<textarea>	Defines a multiline text entry box.

TABLE 4.2 Form-Related HTML Elements

4.4.1 Text Input Controls

Most forms need to gather text information from the user. Whether it is a search box, or a login form, or a user registration form, some type of text input is usually necessary. Table 4.3 lists the different text input controls.

While some of the HTML5 text elements are not uniformly supported by all browsers, they still work as regular text boxes in older browsers. Figure 4.16

Type	Description
text	Creates a single-line text entry box. <code><input type="text" name="title" /></code>
textarea	Creates a multiline text entry box. You can add content text or if using an HTML5 browser, placeholder text (hint text that disappears once user begins typing into the field). <code><textarea rows="3" ... /></code>
password	Creates a single-line text entry box for a password (which masks the user entry as bullets or some other character) <code><input type="password" ... /></code>
search	Creates a single-line text entry box suitable for a search string. This is an HTML5 element. Some browsers on some platforms will style search elements differently or will provide a clear field icon within the text box. <code><input type="search" ... /></code>
email	Creates a single-line text entry box suitable for entering an email address. This is an HTML5 element. Some devices (such as the iPhone) will provide a specialized keyboard for this element. Some browsers will perform validation when form is submitted. <code><input type="email" ... /></code>
tel	Creates a single-line text entry box suitable for entering a telephone. This is an HTML5 element. Since telephone numbers have different formats in different parts of the world, current browsers do not perform any special formatting or validation. Some devices may, however, provide a specialized keyboard for this element. <code><input type="tel" ... /></code>
url	Creates a single-line text entry box suitable for entering a URL. This is an HTML5 element. Some devices may provide a specialized keyboard for this element. Some browsers also perform validation on submission. <code><input type="url" ... /></code>

TABLE 4.3 Text Input Controls


```
<input type="text" ... />
```

Text:

```
<textarea>enter some text</textarea>
<textarea placeholder="enter some text">
</textarea>
```

TextArea: TextArea:

```
<input type="password" ... />
```

Password: Password:

```
<input type="search" placeholder="enter search text" ... />
```

Search: Search:

```
<input type="email" ... />
```

Email: *In Opera*
 Please enter a valid email address
 Email: *In Chrome*
 Please enter an email address.

```
<input type="url" ... />
```

url:
 Please enter a URL.

```
<input type="tel" ... />
```

Tel:

FIGURE 4.16 Text input controls



PRO TIP

HTML5 added some helpful additions to the form designer's repertoire. The first of these is the `pattern` attribute for text controls. This attribute allows you to specify a regular expression pattern that the user input must match. You can use the `placeholder` attribute to provide guidance to the user about the expected format of the input. Figure 4.17 illustrates a sample pattern for a Canadian postal code. You will learn more about regular expressions in Chapter 12.

Another addition is the `required` attribute, which allows you to tell the browser that the user cannot leave the field blank, but must enter something into it. If the user leaves the field empty, then the browser will display a message.

The `<autocomplete>` attribute is also a new addition to HTML5. It tells the browser whether the control (or the entire form if placed within the `<form>` element) should have autocomplete enabled, which allows the browser to display predictive options for the element based on previously entered values.

```
<input type="text" ... placeholder="L#L #L#" pattern="[a-z][0-9][a-z] [0-9][a-z][0-9]" />
```



FIGURE 4.17 Using the pattern attribute

The new `<datalist>` element is another new addition to HTML5. This element allows you to define a list of elements that can appear in a drop-down autocomplete style list for a text element. This can be helpful for situations in which the user must have the ability to enter anything, but are often entering one of a handful of common elements. In such a case, the `<datalist>` can be helpful. Figure 4.18 illustrates a sample usage.

It should be noted that there are a variety of JavaScript-based autocomplete solutions that are often better choices than the HTML5 `<datalist>` since they work on multiple browsers (the `<datalist>` is not supported by all browsers) and provide better customization.

illustrates the various text element controls and some examples of how they look in selected browsers.

4.4.2 Choice Controls

Forms often need the user to select an option from a group of choices. HTML provides several ways to do this.

Select Lists

The `<select>` element is used to create a multiline box for selecting one or more items. The options (defined using the `<option>` element) can be hidden in a drop-down list or multiple rows of the list can be visible. Option items can be grouped together via the `<optgroup>` element. The selected attribute in the `<option>` makes it a default value. These options can be seen in Figure 4.19.



**HANDS-ON
EXERCISES**

LAB 4 EXERCISE
Choice Controls

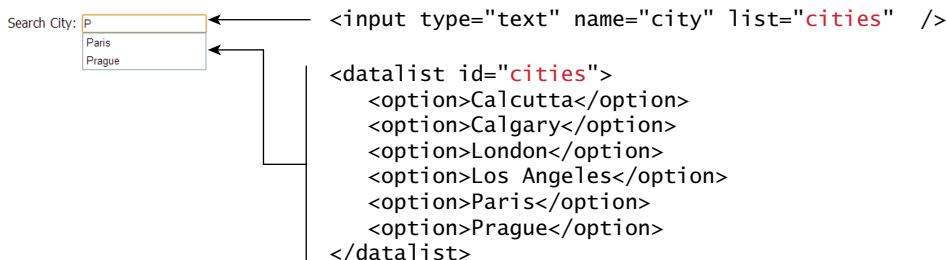
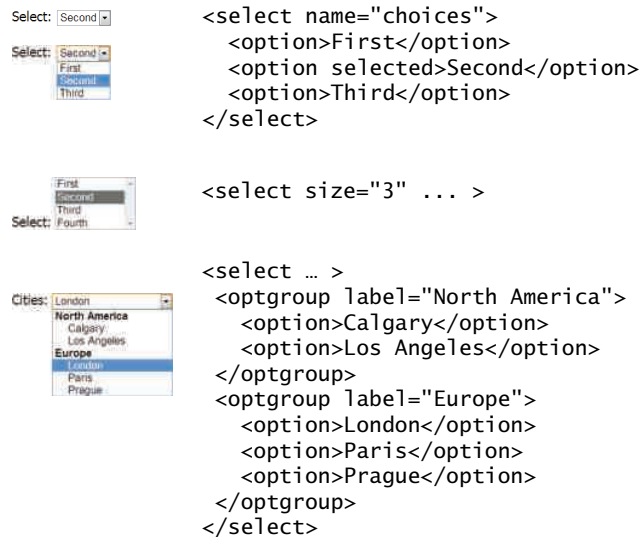
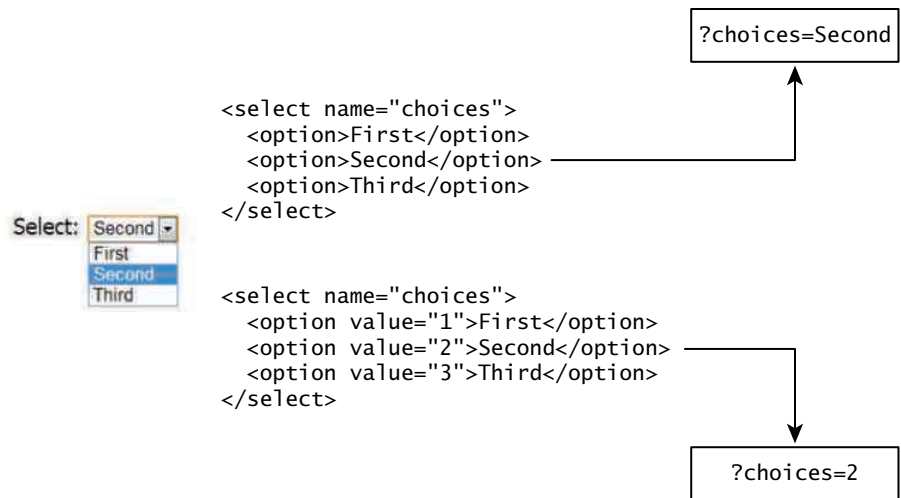


FIGURE 4.18 Using the `<datalist>` element

FIGURE 4.19 Using the `<select>` element

The `value` attribute of the `<option>` element is used to specify what value will be sent back to the server in the query string when that option is selected. The `value` attribute is optional; if it is not specified, then the text within the container is sent instead, as can be seen in Figure 4.20.

FIGURE 4.20 The `value` attribute

Continent:

☐ North America

☒ South America

☐ Asia

```
<input type="radio" name="where" value="1">North America<br/>
<input type="radio" name="where" value="2" checked="">South America<br/>
<input type="radio" name="where" value="3">Asia
```

FIGURE 4.21 Radio buttons

Radio Buttons

Radio buttons are useful when you want the user to select a single item from a small list of choices and you want all the choices to be visible. As can be seen in Figure 4.21, radio buttons are added via the `<input type="radio">` element. The buttons are made mutually exclusive (i.e., only one can be chosen) by sharing the same name attribute. The `checked` attribute is used to indicate the default choice, while the `value` attribute works in the same manner as with the `<option>` element.

Checkboxes

Checkboxes are used for getting yes/no or on/off responses from the user. As can be seen in Figure 4.22, checkboxes are added via the `<input type="checkbox">` element. You can also group checkboxes together by having them share the same name attribute. Each checked checkbox will have its value sent to the server.

Like with radio buttons, the `checked` attribute can be used to set the default value of a checkbox.

4.4.3 Button Controls

HTML defines several different types of buttons, which are shown in Table 4.4. As can be seen in that table, there is some overlap between two of the button types. Figure 4.23 demonstrates some sample button elements.



HANDS-ON
EXERCISES

LAB 4 EXERCISE
Button Controls

I accept the software license ☒

Where would you like to visit?

☒ Canada

☐ France

☒ Germany

```
<label>I accept the software license</label>
<input type="checkbox" name="accept" checked="">
```

```
<label>Where would you like to visit? </label><br/>
<input type="checkbox" name="visit" value="canada">Canada<br/>
<input type="checkbox" name="visit" value="france">France<br/>
<input type="checkbox" name="visit" value="germany">Germany
```

?accept=on&visit=canada&visit=germany

FIGURE 4.22 Checkbox buttons

Type	Description
<code><input type="submit"></code>	Creates a button that submits the form data to the server.
<code><input type="reset"></code>	Creates a button that clears any of the user's already entered form data.
<code><input type="button"></code>	Creates a custom button. This button may require JavaScript for it to actually perform any action.
<code><input type="image"></code>	Creates a custom submit button that uses an image for its display.
<code><button></code>	<p>Creates a custom button. The <code><button></code> element differs from <code><input type="button"></code> in that you can completely customize what appears in the button; using it, you can, for instance, include both images and text, or skip server-side processing entirely by using hyperlinks.</p> <p>You can turn the button into a submit button by using the <code>type="submit"</code> attribute.</p>

TABLE 4.4 Button Elements

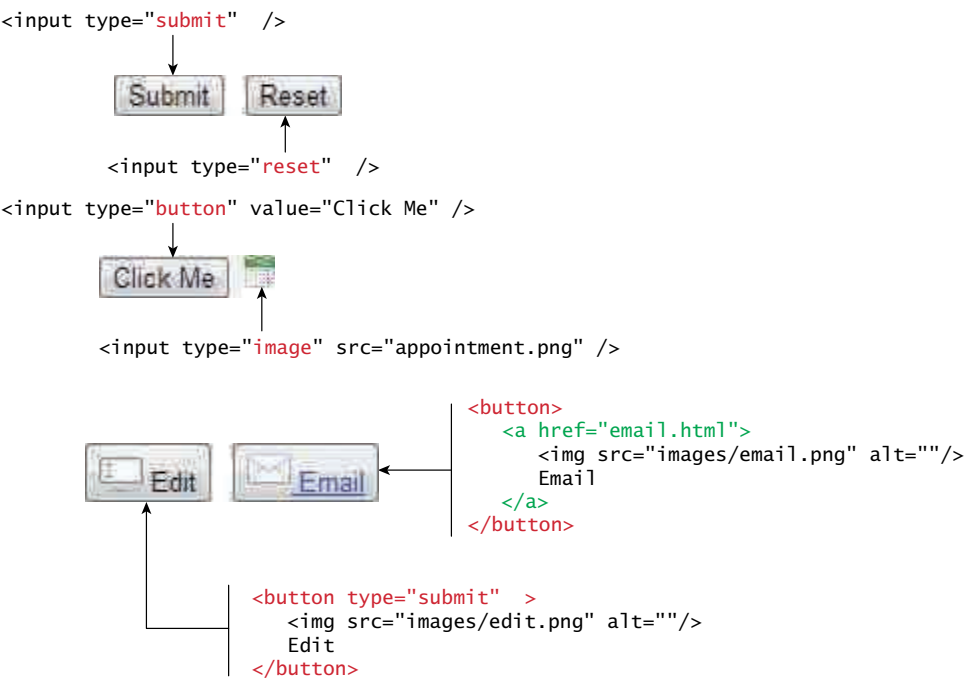


FIGURE 4.23 Example button elements

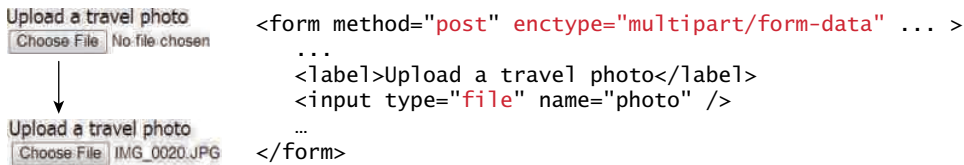


FIGURE 4.24 File upload control (in Chrome)

4.4.4 Specialized Controls

There are two important additional special-purpose form controls that are available in all browsers. The first of these is the `<input type="hidden">` element, which will be covered in more detail in Chapter 13 on State Management. The other specialized form control is the `<input type="file">` element, which is used to upload a file from the client to the server. The usage and user interface for this control are shown in Figure 4.24. The precise look for this control can vary from browser to browser, and platform to platform.

Notice that the `<form>` element must use the `post` method and that it must include the `enctype="multipart/form-data"` attribute as well. As we have seen in the section on query strings, form data is URL encoded (i.e., `enctype="application/x-www-form-urlencoded"`). However, files cannot be transferred to the server using normal URL encoding, hence the need for the alternative `enctype` attribute.

Number and Range

HTML5 introduced two new controls for the input of numeric values. When input via a standard text control, numbers typically require validation to ensure that the user has entered an actual number and, because the range of numbers is infinite, the entered number has to be checked to ensure it is not too small or too large.

The number and range controls provide a way to input numeric values that eliminate the need for client-side numeric validation (for security reasons you would still check the numbers for validity on the server). Figure 4.25 illustrates the usage and appearance of these numeric controls.

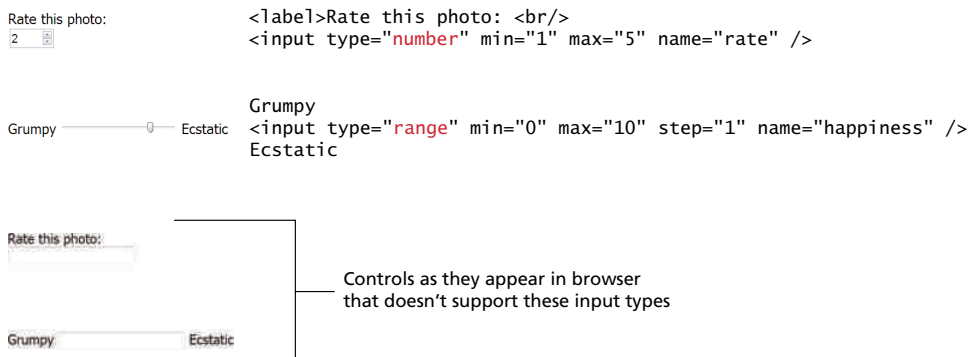


FIGURE 4.25 Number and range input controls



HANDS-ON
EXERCISES

LAB 4 EXERCISE

Specialized Controls

Background Color:



```
<label>Background Color: <br/>
<input type="color" name="back" />
```



Background Color:



Control as it appears in browser that
doesn't support this input type

FIGURE 4.26 Color input control

Color

Not every web page needs the ability to get color data from the user, but when it is necessary, the HTML5 color control provides a convenient interface for the user, as shown in Figure 4.26. At the time of writing, only the latest versions of Chrome and Opera support this control.

4.4.5 Date and Time Controls



HANDS-ON EXERCISES

LAB 4 EXERCISE

Date and Time
Controls

Asking the user to enter a date or time is a relatively common web development task. Like with numbers, dates and times often need validation when gathering this information from a regular text input control. From a user's perspective, entering dates can be tricky as well: you probably have wondered at some point in time when entering a date into a web form, what format to enter it in, whether the day comes before the month, whether the month should be entered as an abbreviation or a number, and so on. The new date and time controls in HTML try to make it easier for users to input these tricky date and time values.

Table 4.5 lists the various HTML5 date and time controls. Their usage and appearance in the browser are shown in Figure 4.27.

Type	Description
date	Creates a general date input control. The format for the date is “yyyy-mm-dd.”
time	Creates a time input control. The format for the time is “HH:MM:SS,” for hours:minutes:seconds.
datetime	Creates a control in which the user can enter a date and time.
datetime-local	Creates a control in which the user can enter a date and time without specifying a time zone.
month	Creates a control in which the user can enter a month in a year. The format is “yyyy-mm.”
week	Creates a control in which the user can specify a week in a year. The format is “yyyy-W##.”

TABLE 4.5 HTML5 Date and Time Controls

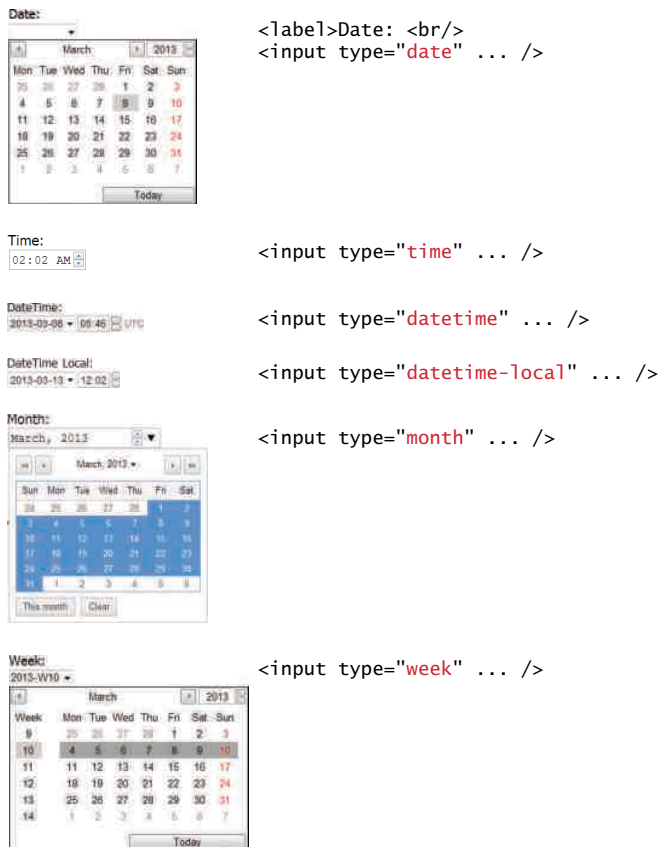


FIGURE 4.27 Date and time controls



NOTE

There are four additional form elements that we have not covered here. The `<progress>` and `<meter>` elements can be used to provide feedback to users, but require JavaScript to function dynamically. The `<output>` element can be used to hold the output from a calculation. This could be used in a form as a way, for instance, to semantically mark up a subtotal or a count of the number of items in a shopping cart. Finally, the `<keygen>` element can be used to hold a private key for public-key encryption.

4.5 Table and Form Accessibility

In Chapter 2, you were reminded that not all web users are able to view the content on web pages in the same manner. Users with sight disabilities, for instance, experience the web using voice reading software. Color blind users might have trouble differentiating certain colors in proximity; users with muscle control problems may have difficulty using a mouse, while older users may have trouble with small text and image sizes. The term **web accessibility** refers to the assistive technologies, various features of HTML that work with those technologies, and different coding and design practices that can make a site more usable for people with visual, mobility, auditory, and cognitive disabilities.

In order to improve the accessibility of websites, the W3C created the **Web Accessibility Initiative (WAI)** in 1997. The WAI produces guidelines and recommendations, as well as organizing different working groups on different accessibility issues. One of its most helpful documents is the Web Content Accessibility Guidelines, which is available at <http://www.w3.org/WAI/intro/wcag.php>.

Perhaps the most important guidelines in that document are:

- *Provide text alternatives for any nontext content so that it can be changed into other forms people need, such as large print, braille, speech, symbols, or simpler language.*
- *Create content that can be presented in different ways (for example simpler layout) without losing information or structure.*
- *Make all functionality available from a keyboard.*
- *Provide ways to help users navigate, find content, and determine where they are.*

The guidelines provide detailed recommendations on how to achieve this advice. This section will look at how one can improve the accessibility of tables and forms, two HTML structures that are often plagued by a variety of accessibility issues.

4.5.1 Accessible Tables

HTML tables can be quite frustrating from an accessibility standpoint. Users who rely on visual readers can find pages with many tables especially difficult to use. One vital way to improve the situation is to only use tables for tabular data, not for layout. Using the following accessibility features for tables in HTML can also improve the experience for those users:

1. Describe the table's content using the `<caption>` element (see Figure 4.6).
This provides the user with the ability to discover what the table is about before having to listen to the content of each and every cell in the table. If you have an especially long description for the table, consider putting the table within a `<figure>` element and use the `<figcaption>` element to provide the description.
2. Connect the cells with a textual description in the header. While it is easy for a sighted user to quickly see what row or column a given data cell is in, for users relying on visual readers, this is not an easy task.

It is quite revealing to listen to reader software recite the contents of a table that has not made these connections. It sounds like this: “row 3, cell 4: 45.56; row 3, cell 5: Canada; row 3, cell 6: 25,000; etc.” However, if these connections have been made, it sounds instead like this: “row 3, Average: 45.56; row 3, Country: Canada; row 3, City Count: 25,000; etc.,” which is a significant improvement.

Listing 4. 1 illustrates how to use the `scope` attribute to connect cells with their headers.

```
<table>
  <caption>Famous Paintings</caption>
  <tr>
    <th scope="col">Title</th>
    <th scope="col">Artist</th>
    <th scope="col">Year</th>
    <th scope="col">Width</th>
    <th scope="col">Height</th>
  </tr>
  <tr>
    <td>The Death of Marat</td>
    <td>Jacques-Louis David</td>
    <td>1793</td>
    <td>162cm</td>
    <td>128cm</td>
  </tr>
  <tr>
    <td>Burial at ornans</td>
```

(continued)

```

        <td>Gustave Courbet</td>
        <td>1849</td>
        <td>314cm</td>
        <td>663cm</td>
    </tr>
</table>

```

LISTING 4.1 Connecting cells with headers

4.5.2 Accessible Forms

HTML forms are also potentially problematic from an accessibility standpoint. If you remember the advice from the WAI about providing keyboard alternatives and text alternatives, your forms should be much less of a problem.

The forms in this chapter already made use of the `<fieldset>`, `<legend>`, and `<label>` elements, which provide a connection between the input elements in the form and their actual meaning. In other words, these controls add semantic content to the form.

While the browser does provide some unique formatting to the `<fieldset>` and `<legend>` elements, their main purpose is to logically group related form input elements together with the `<legend>` providing a type of caption for those elements. You can of course use CSS to style (or even remove the default styling) these elements.

The `<label>` element has no special formatting (though we can use CSS to do so). Each `<label>` element should be associated with a single input element. You can make this association explicit by using the `for` attribute, as shown in Figure 4.28. Doing so means that if the user clicks on or taps the `<label>` text, that control will

```

<label for="f-title">Title: </label>

<input type="text" name="title" id="f-title"/>

<label for="f-country">Country: </label>

<select name="where" id="f-country">
  <option>Choose a country</option>
  <option>Canada</option>
  <option>Finland</option>
  <option>United States</option>
</select>

```

FIGURE 4.28 Associating labels and input elements



BACKGROUND

In the middle 2000s, websites became much more complicated as new JavaScript techniques allowed developers to create richer user experiences almost equivalent to what was possible in dedicated desktop applications. These richer Internet applications were (and are) a real problem for the accessibility guidelines that had developed around a much simpler web page paradigm. The W3C's Website Accessibility Initiative (WAI) developed a new set of guidelines for Accessible Rich Internet Applications (ARIA).

The specifications and guidance in the WAI-ARIA site are beyond the scope of this book. Much of its approach is based on assigning standardized roles via the `role` attribute to different elements in order to make clear just what navigational or user interface role some HTML element has on the page. Some of the ARIA roles include: navigation, link, tree, dialog, menu, and toolbar.

receive the form's focus (i.e., it becomes the current input element and any keyboard input will affect that control).

4.6 Microformats

The web has millions of pages in it. Yet, despite the incredible variety, there is a surprising amount of similar information from site to site. Most sites have some type of Contact Us page, in which addresses and other information are displayed; similarly, many sites contain a calendar of upcoming events or information about products or news. The idea behind microformats is that if this type of common information were tagged in similar ways, then automated tools would be able to gather and transform it.

Thus, a **microformat** is a small pattern of HTML markup and attributes to represent common blocks of information such as people, events, and news stories so that the information in them can be extracted and indexed by software agents. Figure 4.29 illustrates this process.

One of the most common microformats is **hCard**, which is used to semantically mark up contact information for a person. Google Map search results now make use of the hCard microformat so that if you used the appropriate browser extension, you could save the information to your computer or phone's contact list.

Listing 4.2 illustrates the example markup for a person's contact information that uses the hCard microformat. To learn more about the hCard format, visit <http://microformats.org/wiki/hcard>.

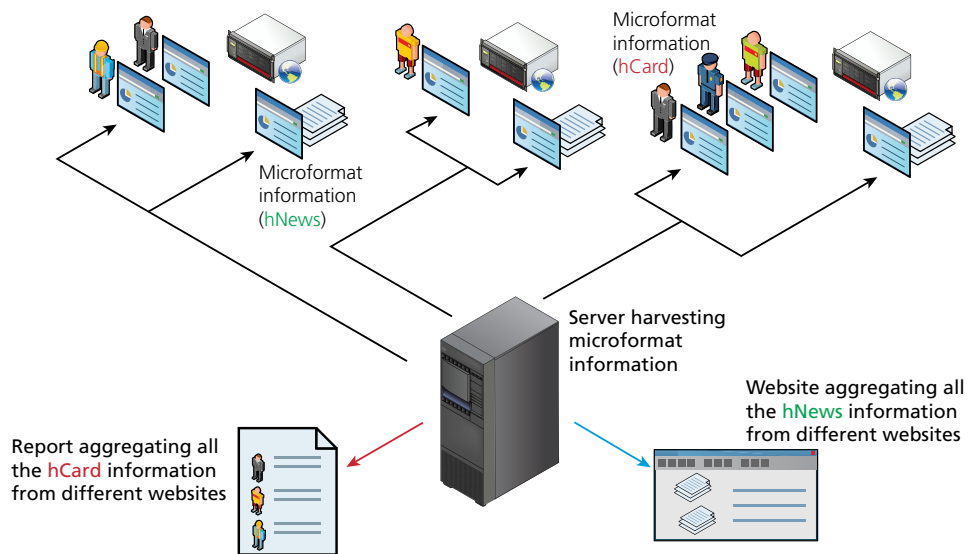


FIGURE 4.29 Microformats

```
<div class="vcard">
  <span class="fn">Randy Connolly</span>
  <div class="org">Mount Royal University</div>
  <div class="adr">
    <div class="street-address">4825 Mount Royal Gate SW</div>
    <div>
      <span class="locality">Calgary</span>,
      <abbr class="region" title="Alberta">AB</abbr>
      <span class="postal-code">T3E 6K6</span>
    </div>
    <div class="country-name">Canada</div>
  </div>
  <div>Phone: <span class="tel">+1-403-440-6111</span></div>
</div>
```

LISTING 4.2 Example of an hCard

4.7 Chapter Summary

This chapter has examined the remaining essential HTML topics: tables and forms. Tables are properly used for presenting tabular data, though in the past, tables were also used for page layout. Forms provide a way to send information to the server,

and are thus an essential part of almost any real website. Both forms and tables have accessibility issues, and this chapter also examined how the accessibility of websites can be improved through the correct construction of tables and forms. Finally, this chapter covered microformats, which can be used to provide additional semantic information about snippets of information within a page.

4.7.1 Key Terms

checkbox	POST	URL encoded
colspan	query string	web accessibility
form	radio buttons	Web Accessibility
GET	rowspan	Initiative (WAI)
hCard	table	
microformat		

4.7.2 Review Questions

1. What are the elements used to define the structure of an HTML table?
2. Describe the purpose of a table caption and the table heading elements.
3. How are the rowspan and colspan attributes used?
4. Create a table that correctly uses the caption, thead, tfoot, and tbody elements. Briefly discuss the role of each of these elements.
5. What are the drawbacks of using tables for layout?
6. What is the difference between HTTP GET and POST?
7. What is a query string?
8. What is URL encoding?
9. What are the two different ways of passing information via the URL?
10. What is the purpose of the action attribute?
11. In what situations would you use a radio button? A checkbox?
12. What are some of the main additions to form construction in HTML5?
13. What is web accessibility?
14. How can one make an HTML table more accessible? Create an example accessible table with three columns and three rows in which the first row contains table headings.
15. What are microformats? What is their purpose?

4.7.3 Hands-On Practice

PROJECT 1: Book Rep Customer Relations Management

DIFFICULTY LEVEL: Beginner



**HANDS-ON
EXERCISES**

PROJECT 4.1

Overview

Edit `Chapter04-project01.html` and `Chapter04-project01.css` so the page looks similar to that shown in Figure 4.30.

Instructions

1. You will need to create the calendar month using tables and provide the styling.
2. The month and date of the calendar should be within a `<caption>`.
3. Be sure to use the `<fieldset>` and `<legend>` elements for the form. As well, be sure to use the appropriate accessibility features in the form.
4. Set up the form's method attribute to GET and its action attribute to <http://www.randyconnolly.com/tests/process.php>.

Test

1. Test the form in the browser. Verify that the output from `process.php` matches that shown in Figure 4.30.
2. Change the form method to POST and retest.

PROJECT 2: Art Store

DIFFICULTY LEVEL: Intermediate



**HANDS-ON
EXERCISES**

PROJECT 4.2

Overview

Edit `Chapter04-project02.html` and `Chapter04-project02.css` so the page looks similar to that shown in Figure 4.31.

Instructions

1. The form at the top of this page consists of a text box, and two drop-down lists. For the Genre list, make the other choices “Baroque,” “Renaissance,” and “Realism.” The drop-down list items should have numeric values starting with 0. Notice the placeholder text in the search box.
2. Create a table of paintings that looks similar to that shown in Figure 4.31. Be sure to make the table properly accessible.
3. The checkboxes in the table should be an array of elements, e.g., `<input type="checkbox" name="index[]" value="10" />`. The name and values are arbitrary, but each checkbox needs to have a unique value.
4. The button in each row is a `<button>` element with a dummy link.
5. Set the form's method attribute to POST and its action attribute to <http://www.randyconnolly.com/tests/process.php>.

Test

1. Test the form in the browser. Verify that the output from `process.php` matches that shown in Figure 4.31.

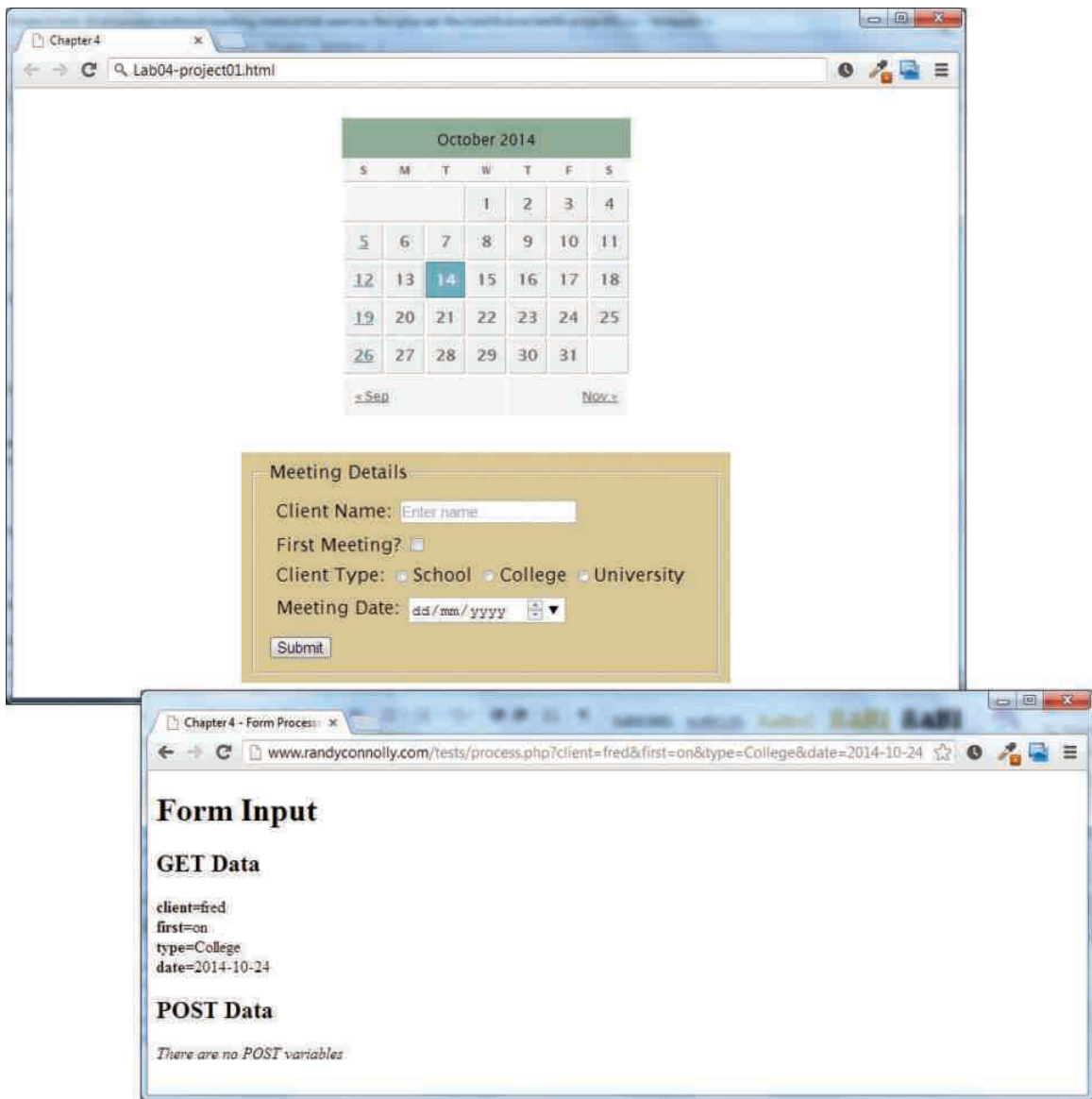


FIGURE 4.30 Completed Project 1

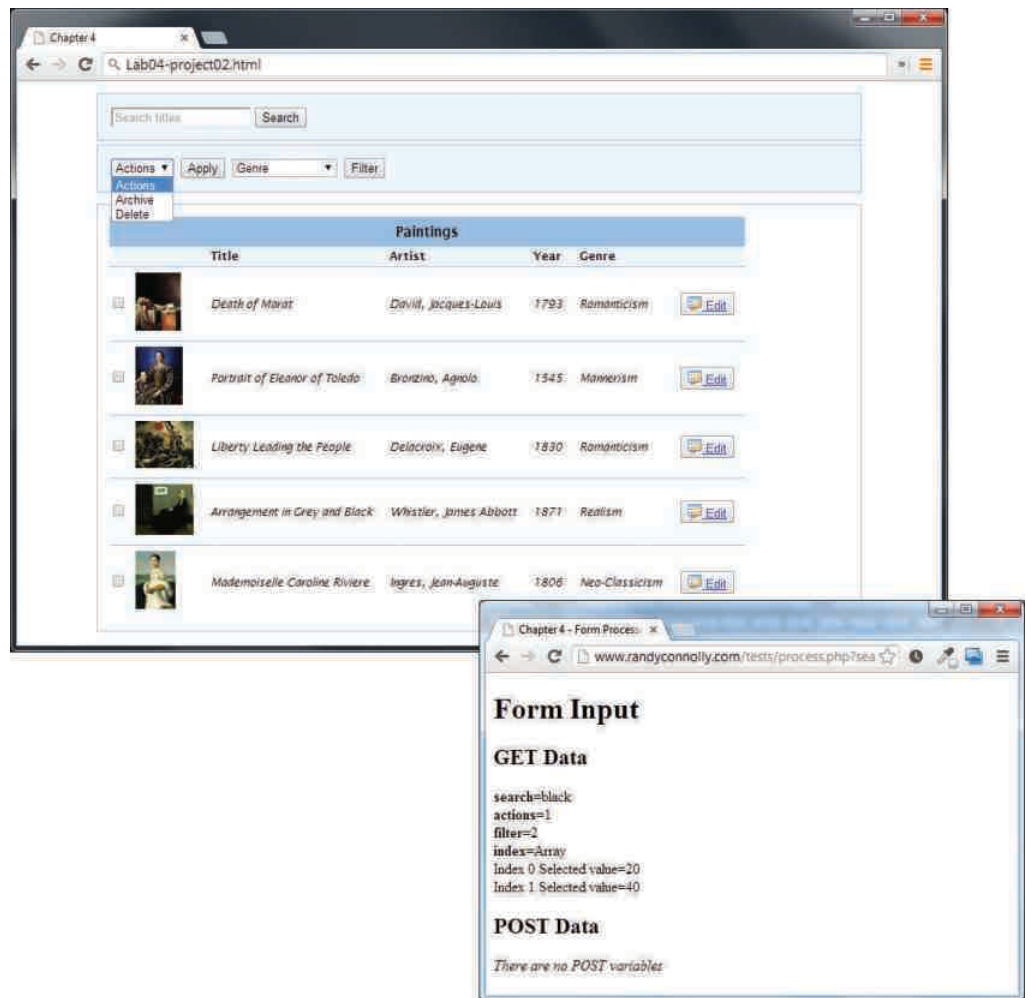


FIGURE 4.31 Completed Project 2

PROJECT 3: Share Your Travel Photos**DIFFICULTY LEVEL:** Advanced**HANDS-ON
EXERCISES
PROJECT 4.3****Overview**

Edit `Chapter04-project03.html` and `Chapter04-project03.css` so the page looks similar to that shown in Figure 4.32.

1. Create the form and position the elements by placing them within a table. While we do not believe that this is best practice, legacy sites often use tables for layout so it may be sensible to get some experience with this approach. In the next chapter, you will learn how to use CSS for layout as a better alternative.

- For the drop-down lists, add a few sensible items to each list. For the checkbox list, they should be an array of elements. Notice also that this form makes use of a number of HTML5 form elements.

Test

- Test the form in the browser. Verify that the output from `process.php` matches that shown in Figure 4.32. Because this form uses HTML5 input elements that are not supported by all browsers, be sure to test in more than one browser.

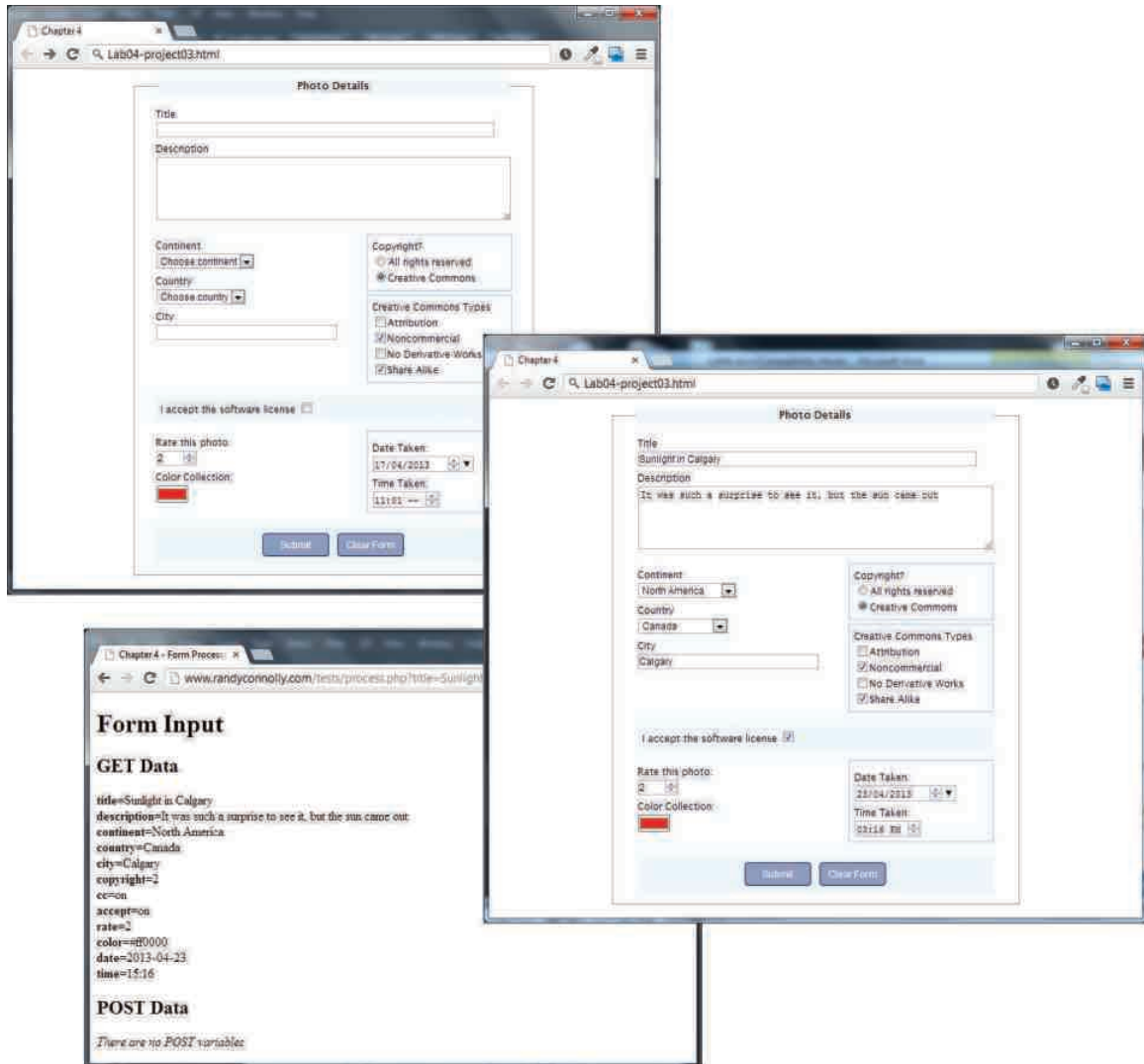


FIGURE 4.32 Completed Project 3

5 Advanced CSS: Layout

CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What normal page flow is
- How to position and float elements outside of the normal page flow
- How to construct multicolumn layouts using positioning and floating
- Different approaches to page layout in CSS
- What responsive web design is and how to construct responsive designs
- How to use CSS frameworks to simplify complex CSS tasks

This chapter covers the other half of CSS. It builds on your knowledge of the basic principles of CSS, including the box model and the most common appearance properties. This chapter examines additional CSS properties that take items out of the normal flow and move them up, down, left, and right, all of which are essential for creating complex layouts. The chapter will examine different approaches to creating page layouts, approaches that can be tricky and complicated to learn and implement. To aid in that process, the chapter will also look at the alternative of using a CSS framework to simplify the process of creating layouts.

5.1 Normal Flow

In Chapter 3, there were occasional references to block-level elements and to inline elements. To understand CSS positioning and layout, it is essential that we understand this distinction as well as the idea of **normal flow**, which refers here to how the browser will normally display block-level elements and inline elements from left to right and from top to bottom.

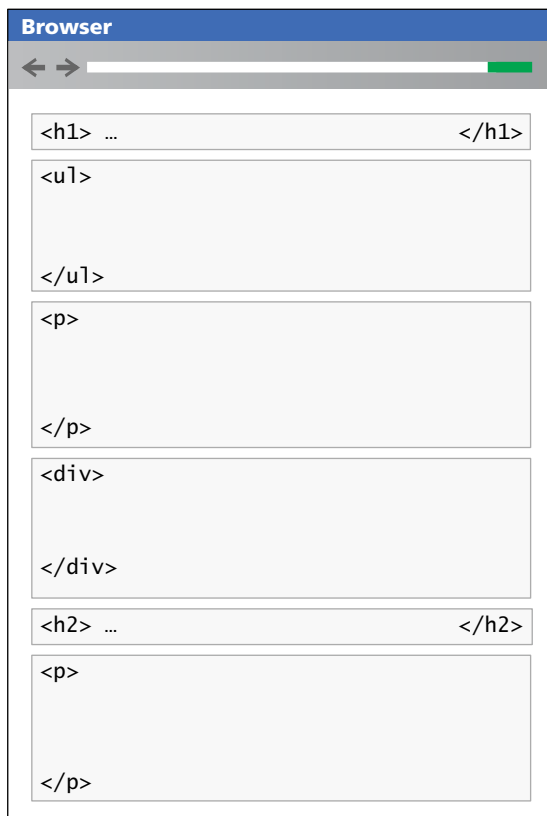
Block-level elements such as `<p>`, `<div>`, `<h2>`, ``, and `<table>` are each contained on their own line. Because block-level elements begin with a line break (that is, they start on a new line), without styling, two block-level elements can't exist on the same line, as shown in Figure 5.1. Block-level elements use the normal CSS box model, in that they have margins, paddings, background colors, and borders.

Inline elements do not form their own blocks but instead are displayed within lines. Normal text in an HTML document is inline, as are elements such as ``, `<a>`, ``, and ``. Inline elements line up next to one another horizontally



**HANDS-ON
EXERCISES**

LAB 5 EXERCISE
Document Flow



Each block exists on its own line and is displayed in normal flow from the browser window's top to its bottom.

By default each block-level element fills up the entire width of its parent (in this case, it is the `<body>`, which is equivalent to the width of the browser window).

You can use CSS box model properties to customize, for instance, the width of the box and the margin space between other block-level elements.

FIGURE 5.1 Block-level elements

from left to right on the same line; when there isn't enough space left on the line, the content moves to a new line, as shown in Figure 5.2.

There are actually two types of inline elements: replaced and nonreplaced. **Replaced inline elements** are elements whose content and thus appearance is defined by some external resource, such as `` and the various form elements. **Nonreplaced inline elements** are those elements whose content is defined within the document, which includes all the other inline elements.

Replaced inline elements have a width and height that are defined by the external resource and thus have the regular CSS box model discussed in Chapter 3. Nonreplaced inline elements, in contrast, have a constrained box model. For instance, because their width is defined by their content (and by other properties such as `font-size` and `letter-spacing`), the `width` property is ignored, as are the `margin-top`, `margin-bottom`, and the `height`.

```
<p>
This photo  of Conservatory Pond in
<a href="http://www.centralpark.com/">Central Park</a> New York City
was taken on October 22, 2015 with a <strong>Canon EOS 30D</strong>
camera.
</p>
```

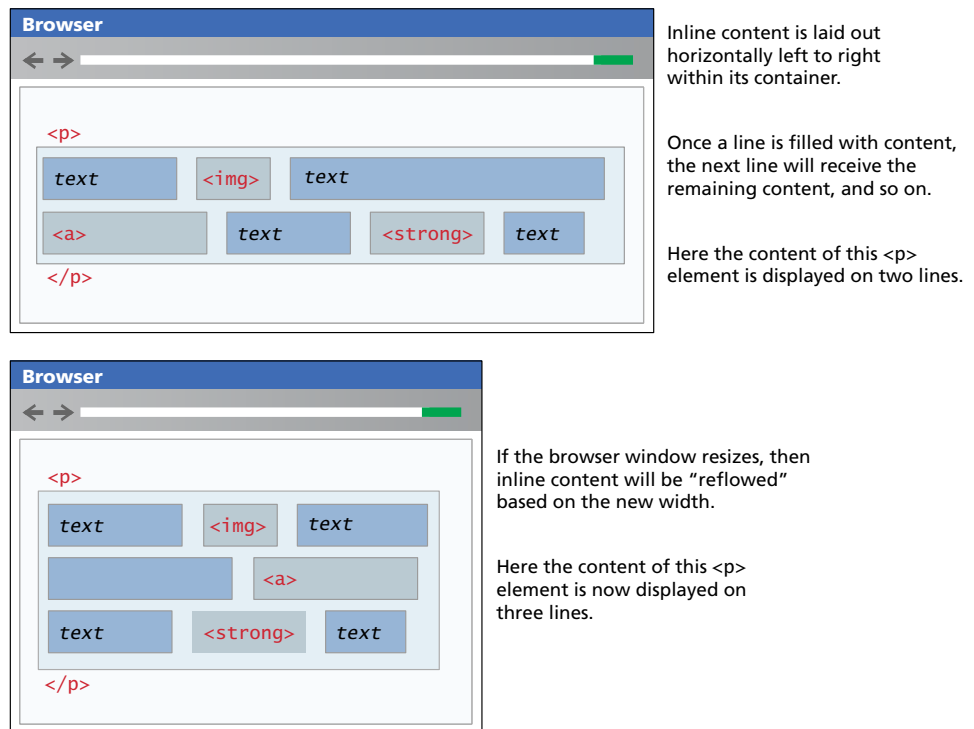
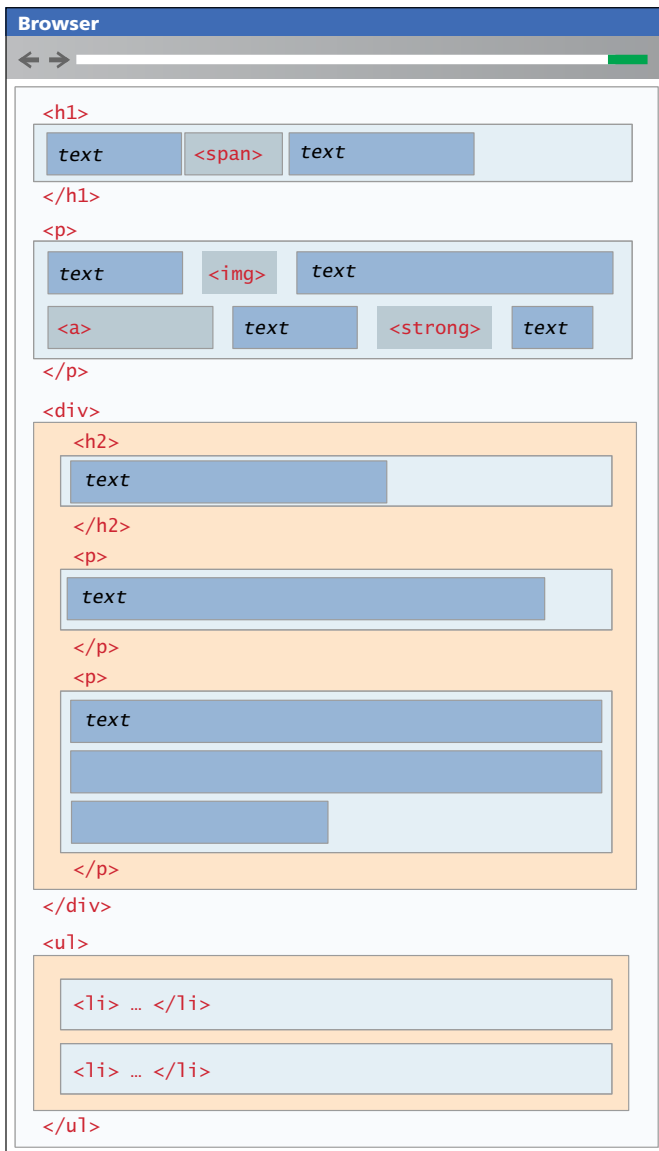


FIGURE 5.2 Inline elements

In a document with normal flow, block-level elements and inline elements work together as shown in Figure 5.3. Block-level elements will flow from top to bottom, while inline elements flow from left to right within a block. If a block contains other blocks, the same behavior happens: the blocks flow from the top to the bottom of the block.



A document consists of block-level elements stacked from top to bottom.

Within a block, inline content is horizontally placed left to right.

Some block-level elements can contain other block-level elements (in this example, a `<div>` can contain other blocks).

In such a case, the block-level content inside the parent is stacked from top to bottom within the container (`<div>`).

FIGURE 5.3 Block and inline elements together

It is possible to change whether an element is block-level or inline via the CSS `display` property. Consider the following two CSS rules:

```
span { display: block; }
li { display: inline; }
```

These two rules will make all `` elements behave like block-level elements and all `` elements like inline (that is, each list item will be displayed on the same line).

5.2 Positioning Elements

It is possible to move an item from its regular position in the normal flow, and even move an item outside of the browser viewport so it is not visible or to position it so it is always visible in a fixed position while the rest of the content scrolls.

The `position` property is used to specify the type of positioning, and the possible values are shown in Table 5.1. The `left`, `right`, `top`, and `bottom` properties are used to indicate the distance the element will move; the effect of these properties varies depending upon the `position` property.

The next several sections will provide examples of how to use `absolute`, `fixed`, and `relative` positioning. While `fixed` position is used relatively infrequently, `absolute` and `relative` positioning are absolutely essential to many of the most common layout techniques in CSS.

5.2.1 Relative Positioning

In **relative positioning** an element is displaced out of its normal flow position and moved relative to where it would have been placed. When an element is positioned relatively, it is displaced out of its normal flow position and moved relative to where it would have been placed. The other content around the relatively positioned element “remembers” the element’s old position in the flow; thus the space the element would have occupied is preserved as shown in Figure 5.4.



HANDS-ON EXERCISES

LAB 5 EXERCISE

Relative Positioning

Value	Description
absolute	The element is removed from normal flow and positioned in relation to its nearest positioned ancestor.
fixed	The element is fixed in a specific position in the window even when the document is scrolled.
relative	The element is moved relative to where it would be in the normal flow.
static	The element is positioned according to the normal flow. This is the default.

TABLE 5.1 Position Values



```
<p>A wonderful serenity has taken possession of my ...
```

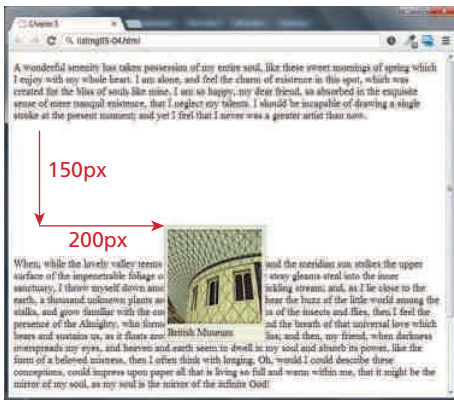
```
<figure>
```

```
  
```

```
  <figcaption>British Museum</figcaption>
```

```
</figure>
```

```
<p>When, while the lovely valley ...
```



```
figure {
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: relative;
  top: 150px;
  left: 200px;
}
```

FIGURE 5.4 Relative positioning

As you can see in Figure 5.4, the original space for the positioned `<figure>` element is preserved, as is the rest of the document's flow. As a consequence, the repositioned element now overlaps other content: that is, the `<p>` element following the `<figure>` element does not change to accommodate the moved `<figure>`.

5.2.2 Absolute Positioning

When an element is positioned absolutely, it is removed completely from normal flow. Thus, unlike with relative positioning, space is not left for the moved element, as it is no longer in the normal flow. Its position is moved in relation to its container block. In the example shown in Figure 5.5, the container block is the `<body>` element. Like with the relative positioning example, the moved block can now overlap content in the underlying normal flow.



HANDS-ON
EXERCISES

LAB 5 EXERCISE
Absolute Positioning



FIGURE 5.5 Absolute positioning

While this example is fairly clear, **absolute positioning** can get confusing. A moved element via absolute position is actually positioned relative to its nearest **positioned** ancestor container (that is, a block-level element whose position is fixed, relative, or absolute). In the example shown in Figure 5.6, the `<figcaption>` is absolutely positioned; it is moved 150 px down and 200 px to the left of its nearest positioned ancestor, which happens to be its parent (the `<figure>` element).

5.2.3 Z-Index

Looking at Figure 5.6, you may wonder what would have happened if the `<figcaption>` had been moved so that it overlapped the `<figure>`. Each positioned element has a stacking order defined by the `z-index` property (named for the z-axis). Items closest to the viewer (and thus on the top) have a larger **z-index** value, which can be seen in the first example in Figure 5.7.

Unfortunately, working with `z-index` can be tricky and seemingly counter-intuitive. First, only positioned elements will make use of their `z-index`. Second, as

```
<p>A wonderful serenity has taken possession of my ...
```

```
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
```

```
<p>When, while the lovely valley ...
```

```
figure {
  margin: 0;
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: absolute;
  top: 150px;
  left: 200px;
}
```



**HANDS-ON
EXERCISES**

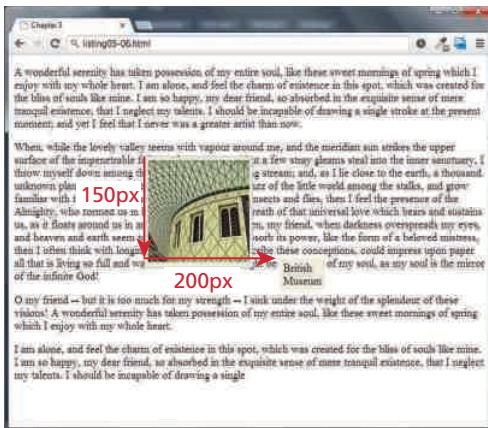
LAB 5 EXERCISE
Stacking Using Z-Index



<p>A wonderful serenity has taken possession of my entire soul, like these sweet mornings of spring which I enjoy with my whole heart. I am alone, and feel the charm of existence in this spot, which was created for the bliss of souls like mine. I am so happy, my dear friend, so absorbed in the exquisite sense of mere tranquil existence, that I neglect my talents. I should be incapable of drawing a single stroke at the present moment; and yet I feel that I never was a greater artist than now.

```
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
```

<p>When, while the lovely valley teems with vapour around me, and the meridian sun strikes the upper surface of the impenetrable foliage of my trees, and but a few stray gleams steal into the inner sanctuary, I throw myself down among the tall grass by the trickling stream; and, as I lie close to the earth, a thousand unknown plants are noticed by me; when I hear the buzz of the little world among the stalks, and grow familiar with the countless indescribable forms of the insects and flies, then I feel the presence of the Almighty, who formed us in his own image, and the breath of that universal love which bears and sustains us, as it floats around us in an eternity of bliss; and then, my friend, when darkness overcasts my eyes, and heaven and



```
figure {
  margin: 0;
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  padding: 5px;
  width: 150px;
  position: absolute;
  top: 150px;
  left: 200px;
}

figcaption {
  background-color: #EDEDDE;
  padding: 5px;
  position: absolute;
  top: 150px;
  left: 200px;
}
```

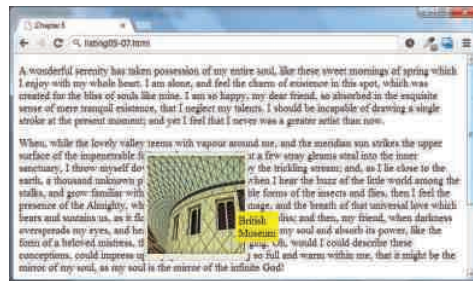
FIGURE 5.6 Absolute position is relative to nearest positioned ancestor container.

can be seen in Figure 5.7, simply setting the z-index value of elements will not necessarily move them on top or behind other items.

5.2.4 Fixed Position

The fixed position value is used relatively infrequently. It is a type of absolute positioning, except that the positioning values are in relation to the viewport (i.e., to the browser window). Elements with **fixed positioning** do not move when the user scrolls up or down the page, as can be seen in Figure 5.8.

The fixed position is most commonly used to ensure that navigation elements or advertisements are always visible.



```
figure {
    position: absolute;
    top: 150px;
    left: 200px;
}
figcaption {
    position: absolute;
    top: 90px;
    left: 140px;
}
```



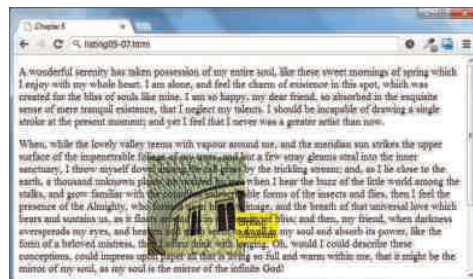
```
figure {
    ...
    z-index: 5;
}
figcaption {
    ...
    z-index: 1;
}
```

Note that this did not move the `<figure>` on top of the `<figcaption>` as one might expect. This is due to the nesting of the caption within the figure.



```
figure {
    ...
    z-index: 1;
}
figcaption {
    ...
    z-index: -1;
}
```

Instead the `<figcaption>` `z-index` must be set below 0. The `<figure>` `z-index` could be any value equal to or above 0.



```
figure {
    ...
    z-index: -1;
}
figcaption {
    ...
    z-index: 1;
}
```

If the `<figure>` `z-index` is given a value less than 0, then any of its positioned descendants change as well. Thus both the `<figure>` and `<figcaption>` move underneath the body text.

FIGURE 5.7 Z-index


```
figure {
  ...
  position: fixed;
  top: 0;
  left: 0;
}
```

Notice that figure is fixed in its position regardless of what part of the page is being viewed.

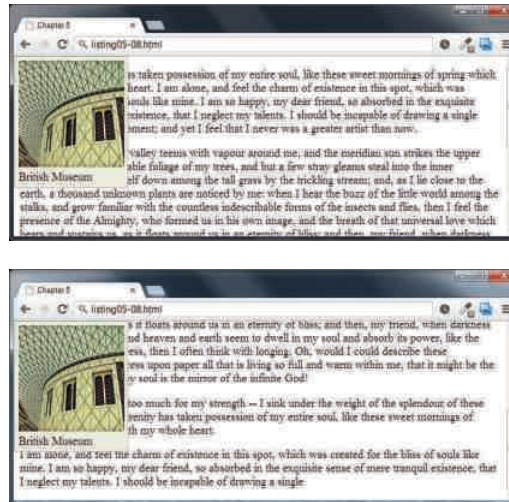


FIGURE 5.8 Fixed position

5.3 Floating Elements

It is possible to displace an element out of its position in the normal flow via the CSS `float` property. An element can be floated to the left or floated to the right. When an item is floated, it is moved all the way to the far left or far right of its containing block and the rest of the content is “re-flowed” around the floated element, as can be seen in Figure 5.9.

Notice that a floated block-level element must have a width specified; if you do not, then the width will be set to auto, which will mean it implicitly fills the entire width of the containing block, and there thus will be no room available to flow content around the floated item. Also note in the final example in Figure 5.9 that the margins on the floated element are respected by the content that surrounds the floated element.

5.3.1 Floating within a Container

It should be reiterated that a floated item moves to the left or right of its container (also called its `containing block`). In Figure 5.9, the containing block is the HTML document itself so the figure moves to the left or right of the browser window. But



**HANDS-ON
EXERCISES**

LAB 5 EXERCISE
Floating Elements



**HANDS-ON
EXERCISES**

LAB 5 EXERCISE
Floating In a Container



FIGURE 5.9 Floating an element

```
<h1>Float example</h1>
<p>A wonderful serenity has taken ...</p>
<figure>
  
  <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley
```

```
figure {
  border: 1pt solid #A8A8A8;
  background-color: #EDEDDE;
  margin: 0;
  padding: 5px;
  width: 150px;
}
```

Notice that a floated block-level element **must** have a width specified.

```
figure {
  ...
  width: 150px;
  float: left;
}
```

```
figure {
  ...
  width: 150px;
  float: right;
  margin: 10px;
}
```

in Figure 5.10, the floated figure is contained within an `<article>` element that is indented from the browser's edge. The relevant margins and padding areas are color coded to help make it clearer how the float interacts with its container.

There is an important change happening in this example, which might not be apparent unless one zooms in to see better, as is shown in Figure 5.11. In this illustration, you can see that the overlapping margins for the adjacent `<p>` elements behave normally and collapse. But notice that the top margin for the floated `<figure>` and the bottom margin for the `<p>` element above it do *not* collapse.

```

<article>
  <h1>Float example</h1>
  <p>A wonderful serenity has taken possession of ... </p>

  <figure>
    
    <figcaption>British Museum</figcaption>
  </figure>

  <p>When, while the lovely valley teems with ...</p>

  <p>O my friend -- but it is too much for my ...</p>
</article>

```



```

article {
  background-color: #898989;
  margin: 5px 50px;
  padding: 5px 20px;
}
p { margin: 16px 0; }
figure {
  border: 1pt solid #262626;
  background-color: #c1c1c1;
  padding: 5px;
  width: 150px;
  float: left;
  margin: 10px;
}

```

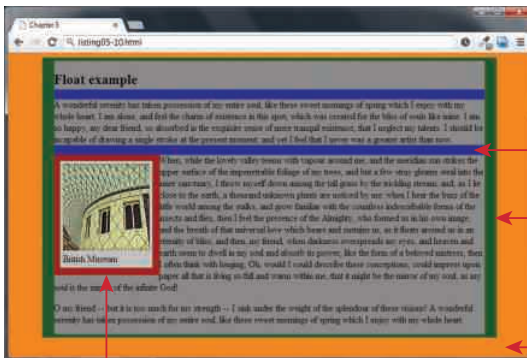


FIGURE 5.10 Floating to the containing block

5.3.2 Floating Multiple Items Side by Side

One of the more common usages of floats is to place multiple items side by side on the same line. When you float multiple items that are in proximity, each floated item in the container will be nestled up beside the previously floated item. All other content in the containing block (including other floated elements) will flow around all the floated elements, as shown in Figure 5.12.

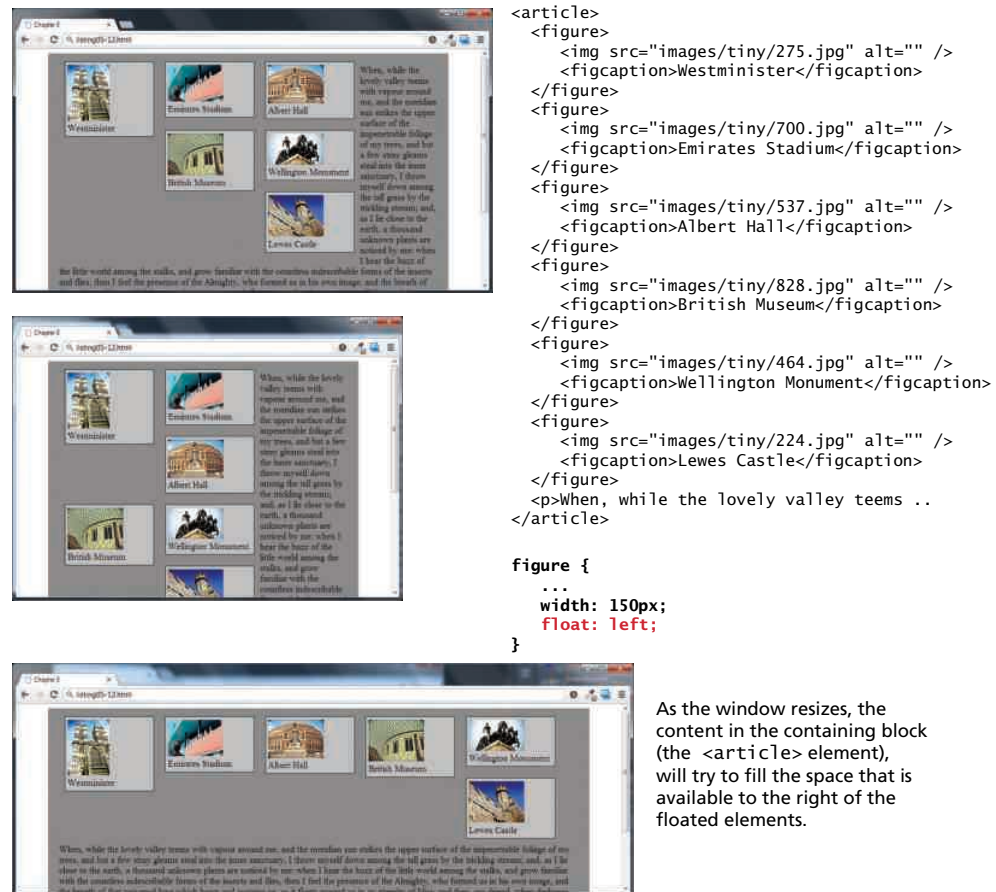
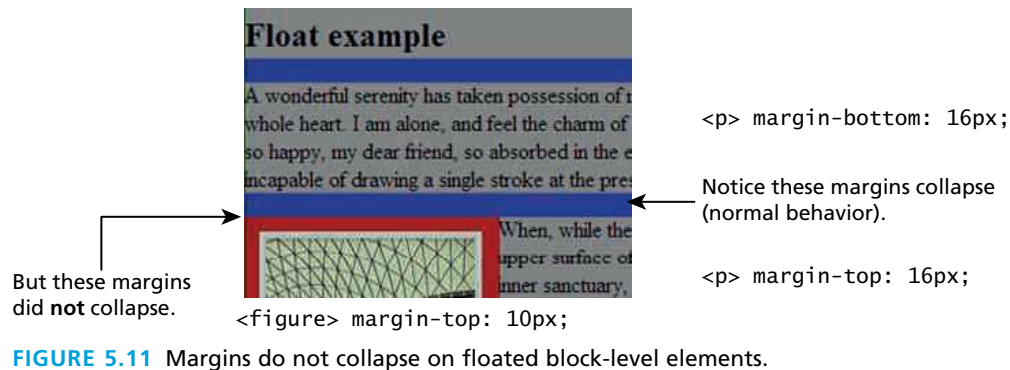


**HANDS-ON
EXERCISES**

LAB 5 EXERCISE

Floating and Clearing

<https://hemanthrajhemu.github.io>



Value	Description
left	The left-hand edge of the element cannot be adjacent to another element.
right	The right-hand edge of the element cannot be adjacent to another element.
both	Both the left-hand and right-hand edges of the element cannot be adjacent to another element.
none	The element can be adjacent to other elements.

TABLE 5.2 Clear Property

As can be seen in Figure 5.12, this can create some pretty messy layouts as the browser window increases or decreases in size (that is, as the containing block resizes). Thankfully, you can stop elements from flowing around a floated element by using the **clear property**. The values for this property are shown in Table 5.2.

Figure 5.13 demonstrates how the use of the `clear` property can solve some of our layout problems. In it, a new CSS class has been created that sets the `clear` property to `left`. The class is then assigned to the elements that need to start on a



FIGURE 5.13 Using the clear property

new line, in this case to one of the `<figure>` elements and to the `<p>` element after the figures.

Unfortunately, the layout in Figure 5.13 will still fall apart if the browser width shrinks so that there is only enough room for one or two of the figures to be displayed. This is not a trivial problem, and this chapter will examine some potential solutions in the section on Responsive Design.

5.3.3 Containing Floats

Another problem that can occur with floats is when an element is floated within a containing block that contains *only* floated content. In such a case, the containing block essentially disappears, as shown in Figure 5.14.

In Figure 5.14, the `<figure>` containing block contains only an `` and a `<figcaption>` element, and both of these elements are floated to the left. That means both elements have been removed from the normal flow; from the browser's perspective, since the `<figure>` contains no normal flow content, it essentially has nothing in it, hence it has a content height of zero.

One solution would be to float the container as well, but depending on the layout this might not be possible. A better solution would be to use the `overflow` property as shown in Figure 5.15.

```
<article>
  <figure>
    
    <figcaption>British Museum</figcaption>
  </figure>
  <p class="first">When, while the lovely valley ...
</article>
```



Notice that the `<figure>` element's content area has shrunk down to zero (it now just has padding space and borders).

```
figure img {
  width: 170px;
  float: left;
  margin: 0 5px;
}
figure figcaption {
  width: 100px;
  float: left;
}
figure {
  border: 1pt solid #262626;
  background-color: #c1c1c1;
  padding: 5px;
  width: 400px;
  margin: 10px;
}
.first { clear: left; }
```

FIGURE 5.14 Disappearing parent containers



Setting the overflow property to auto solves the problem.

```
figure img {
  width: 170px;
  float: left;
  margin: 0 5px;
}
figure figcaption {
  width: 100px;
  float: left;
}
figure {
  border: 1pt solid #262626;
  background-color: #c1c1c1;
  padding: 5px;
  width: 400px;
  margin: 10px;
  overflow: auto;
}
```

FIGURE 5.15 Using the overflow property



PRO TIP

There are a number of different solutions to some of the layout problems with floats. Perhaps the most common of these is the so-called clearfix solution, in which a class named `clearfix` is defined (see below) and assigned to a floated element:

```
.clearfix:after {
  content: "\00A0";
  display: block;
  height: 0;
  clear: both;
  visibility: hidden;
  zoom: 1
}
```

In the example shown in Figure 5.14, it could also be assigned to the `<figure>` element to solve the disappearing parent container. It works by inserting a blank space that is hidden but has the `block` display mode.

5.3.4 Overlaying and Hiding Elements

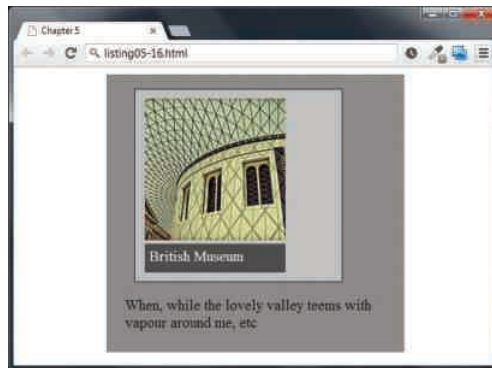
One of the more common design tasks with CSS is to place two elements on top of each other, or to selectively hide and display elements. Positioning is important to both of these tasks.

Positioning is often used for smaller design changes, such as moving items relative to other elements within a container. In such a case, relative positioning is used to create the **positioning context** for a subsequent absolute positioning move. Recall



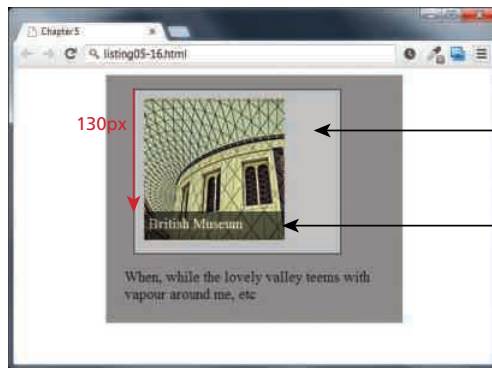
HANDS-ON
EXERCISES

LAB 5 EXERCISE
Using Positioning



```
figure {
  border: 1pt solid #262626;
  background-color: #c1c1c1;
  padding: 10px;
  width: 200px;
  margin: 10px;
}

figcaption {
  background-color: black;
  color: white;
  opacity: 0.6;
  width: 140px;
  height: 20px;
  padding: 5px;
}
```



```
figure {
  ...
  position: relative;
}

figcaption {
  ...
  position: absolute;
  top: 130px;
  left: 10px;
}
```

This creates the positioning context.

This does the actual move.

FIGURE 5.16 Using relative and absolute positioning

that absolute positioning is positioning in relation to the closest positioned ancestor. This doesn't mean that you actually have to move the ancestor; you just set its position to relative. In Figure 5.16, the caption is positioned on top of the image; it doesn't matter where the image appears on the page, its position over the image will always be the same.

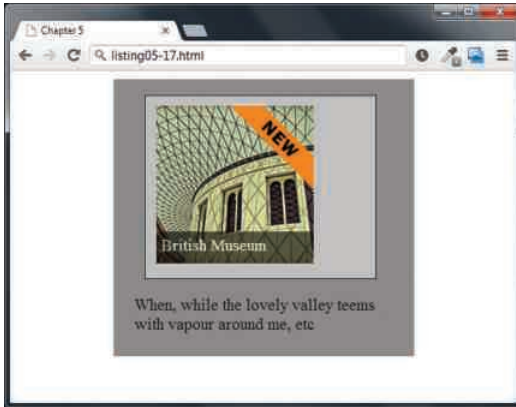
This technique can be used in many different ways. Figure 5.17 illustrates another example of this technique. An image that is the same size as the underlying one is placed on top of the other image using absolute positioning. Since most of this new image contains transparent pixels (something we will learn more about in Chapter 7), it only covers part of the underlying image.

But imagine that this new banner is only to be displayed some of the time. You can hide this image using the `display` property, as shown in Figure 5.17. You might think that it makes no sense to set the `display` property of an element to `none`, but this property is often set programmatically in JavaScript, perhaps in response to user actions or some other logic.

```

<figure>
  
  <figcaption>British Museum</figcaption>
  
</figure>

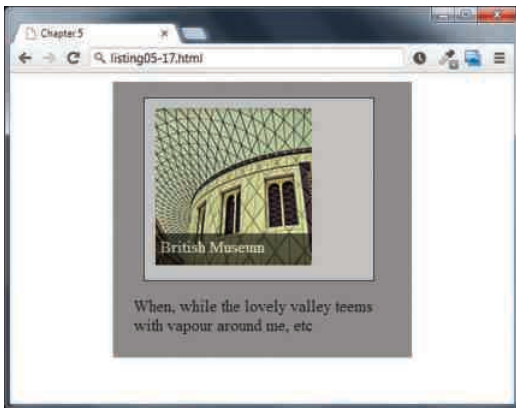
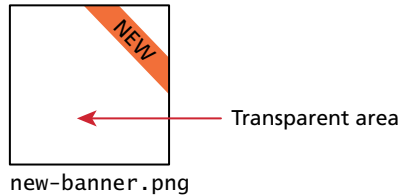
```



```

.overlaid {
  position: absolute;
  top: 10px;
  left: 10px;
}

```



```

.overlaid {
  position: absolute;
  top: 10px;
  left: 10px;
  display: none;
}

```

↑ This hides the overlaid image.

```

.hide {
  display: none;
}

```

↑ This is the preferred way to hide: by adding this class to another element. This makes it clear in the markup that an element is not visible.

```
<img ... class="overlaid hide"/>
```

FIGURE 5.17 Using the display property

There are in fact two different ways to hide elements in CSS: using the `display` property and using the `visibility` property. The `display` property takes an item out of the flow: it is as if the element no longer exists. The `visibility` property just hides the element, but the space for that element remains. Figure 5.18 illustrates the difference between the two properties.

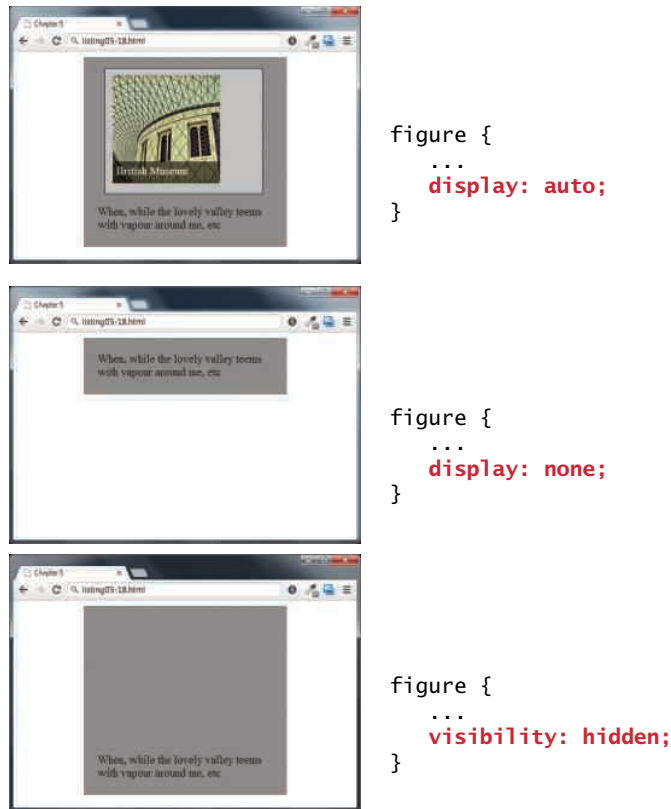


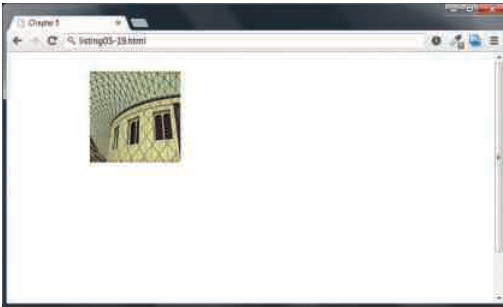
FIGURE 5.18 Comparing display to visibility

While these two properties are often set programmatically via JavaScript, it is also possible to make use of these properties without programming using the `:hover` pseudo-class. Figure 5.19 demonstrates how the combination of absolute positioning, the `:hover` pseudo-class, and the `visibility` property can be used to display a larger version of an image (as well as other markup) when the mouse hovers over the thumbnail version of the image. This technique is also commonly used to create sophisticated tool tips for elements.

**NOTE**

Using the `display:none` and `visibility:hidden` properties on a content element also makes it invisible to screen readers as well (i.e., the content will not be spoken by the screen reader software). If the hidden content is meant to be accessible to screen readers, then another hiding mechanism (such as large negative margins) will be needed.

```
<figure class="thumbnail">
  
  <figcaption class="popup">
    
    <p>The library in the British Museum in London</p>
  </figcaption>
</figure>
```

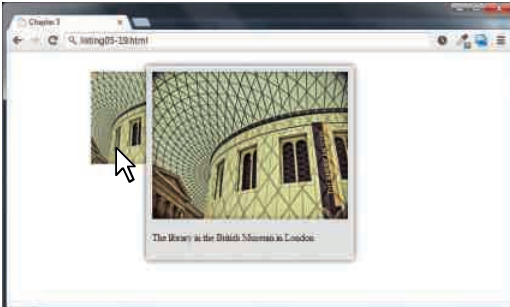


When the page is displayed, the larger version of the image, which is within the `<figcaption>` element, is hidden.

```
figcaption.popup {
  padding: 10px;
  background: #e1e1e1;
  position: absolute;

  /* add a drop shadow to the frame */
  -webkit-box-shadow: 0 0 15px #A9A9A9;
  -moz-box-shadow: 0 0 15px #A9A9A9;
  box-shadow: 0 0 15px #A9A9A9;

  /* hide it until there is a hover */
  visibility: hidden;
}
```



When the user moves/hovers the mouse over the thumbnail image, the `visibility` property of the `<figcaption>` element is set to `visible`.

```
figure.thumbnail:hover figcaption.popup {
  position: absolute;
  top: 0;
  left: 100px;

  /* display image upon hover */
  visibility: visible;
}
```

FIGURE 5.19 Using hover with display

5.4 Constructing Multicolumn Layouts

The previous sections showed two different ways to move items out of the normal top-down flow, namely, by using positioning and by using floats. They are the raw techniques that you can use to create more complex layouts. A typical layout may very well use both positioning and floats.

There is unfortunately no simple and easy way to create robust multicolumn page layouts. There are tradeoffs with each approach, and while this chapter cannot examine the details of every technique for creating multicolumn layouts, it will provide some guidance as to the general issues and provide some illustrations of typical approaches.

**NOTE**

As a reminder from the previous chapter, prior to the broad support for CSS in browsers, HTML tables were frequently used to create page layouts. Unfortunately, this practice of using tables for layout has a variety of problems: larger HTML files, unsemantic markup, and inaccessible.

**HANDS-ON
EXERCISES****LAB 5 EXERCISE**
Two-Column Layout

5.4.1 Using Floats to Create Columns

Using floats is perhaps the most common way to create columns of content. The approach is shown in Figures 5.20 and 5.21. The first step is to float the content container that will be on the left-hand side. Remember that the floated container needs to have a width specified.

As can be seen in the second screen capture in Figure 5.20, the other content will flow around the floated element. Figure 5.21 shows the other key step: changing the left-margin so that it no longer flows back under the floated content.

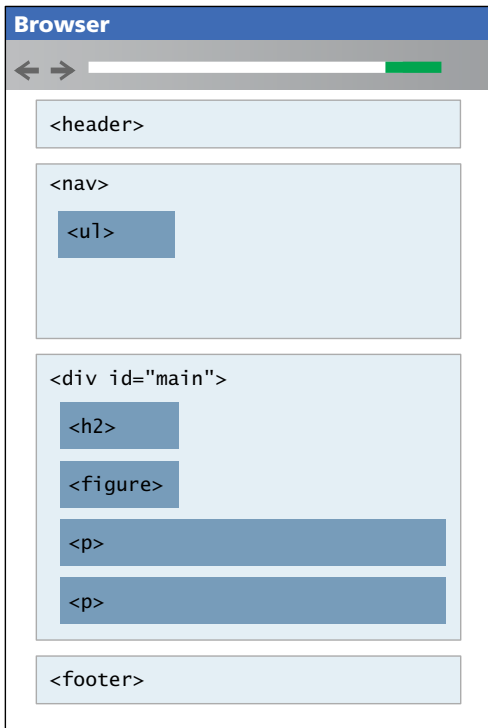
As you can see in Figure 5.21, there are still some potential issues. The background of the floated element stops when its content ends. If we wanted the background color to descend down to the footer, then it is difficult (but not impossible) to achieve this visual effect with floats. The solution (one of which is to use background images) to this type of problem can be found in any dedicated CSS book.

A three-column layout could be created in much the same manner, as shown in Figure 5.22.

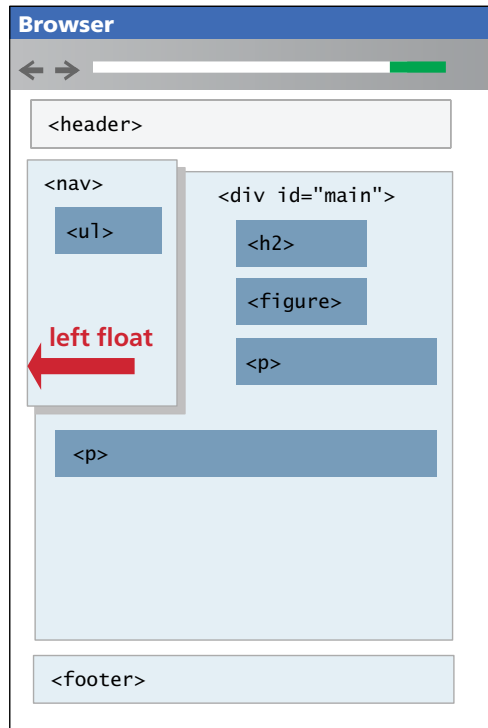
**NOTE**

There is a very important point to be made about the source order of the content in Figure 5.22. Notice that the left and right floated content must be in the source *before* the main nonfloated <div>. If the <aside> element had been after the main <div>, then it would have been floated below the main <div>. As well, screen readers will read the content in the order it is in the HTML.

1 HTML source order (normal flow)



2 Two-column layout (left float)



```
nav {
  ...
  width: 12em;
  float: left;
}
```

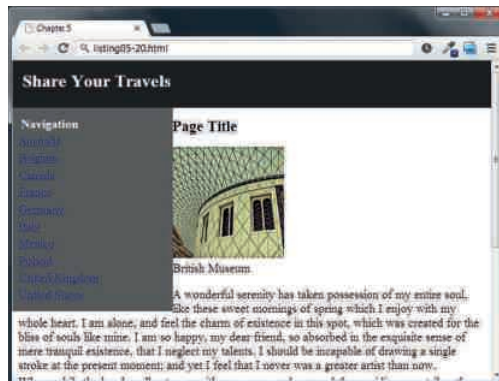


FIGURE 5.20 Creating two-column layout, step one

3 Set the left margin of non-floated content

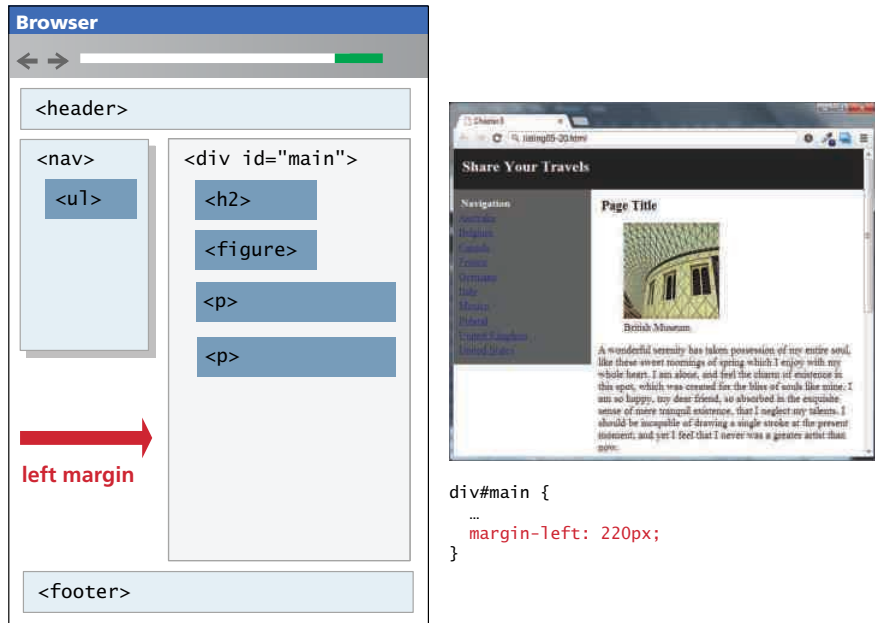


FIGURE 5.21 Creating two-column layout, step two

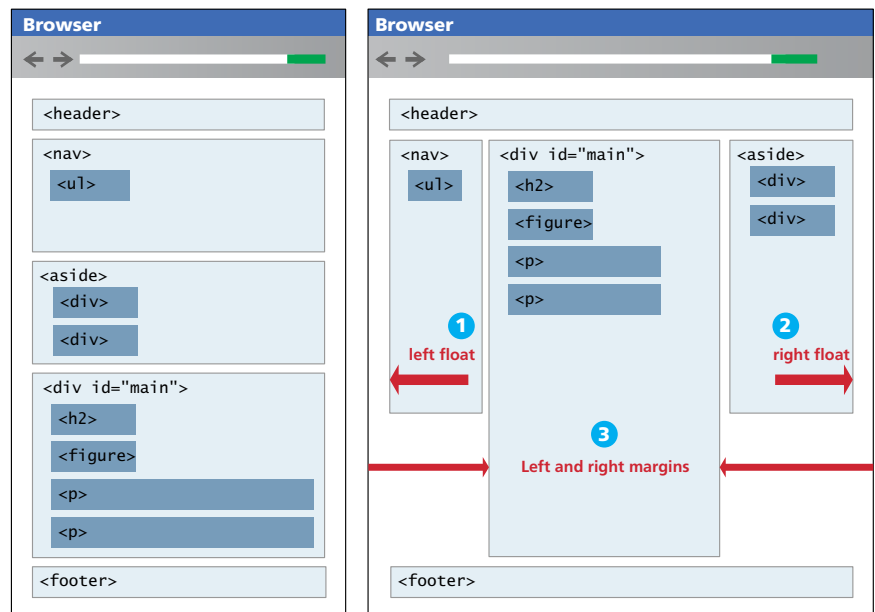


FIGURE 5.22 Creating a three-column layout

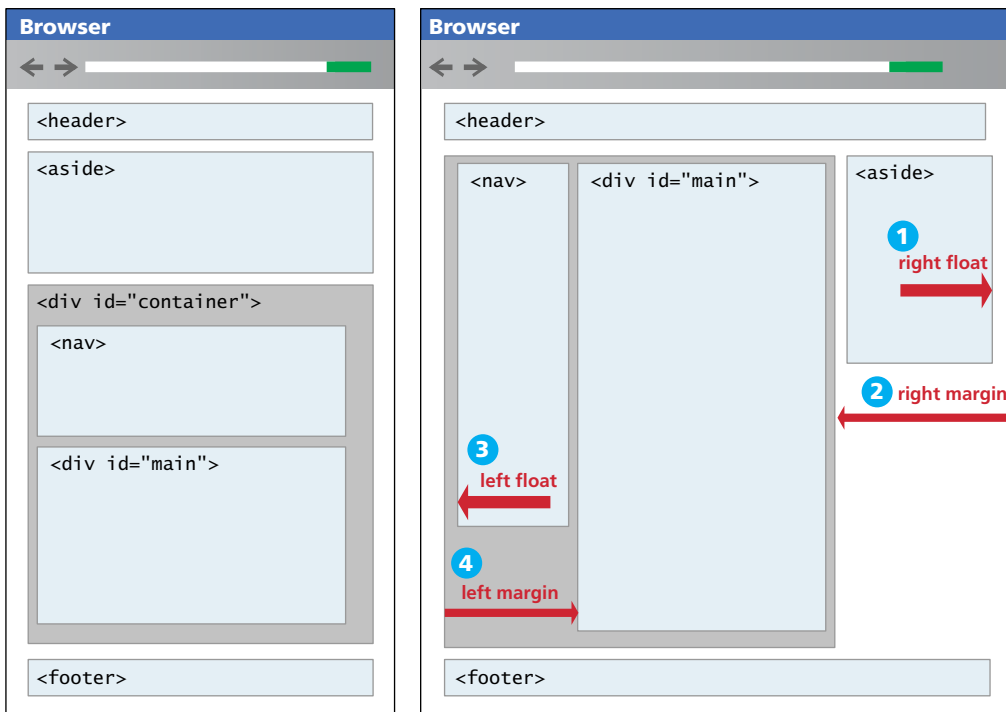


FIGURE 5.23 Creating a three-column layout with nested floats

Another approach for creating a three-column layout is to float elements *within* a container element. This approach is actually a little less brittle because the floated elements within a container are independent of elements outside the container. Figure 5.23 illustrates this approach.

Notice again that the floated content must appear in the source *before* the non-floated content. This is the main problem with the floated approach: that we can't necessarily put the source in an SEO-optimized order (which would be to put the main page content *before* the navigation and the aside). There are in fact ways to put the content in an SEO-optimized order with floats, but typically this requires making use of certain tricks such as giving the main content negative margins.

5.4.2 Using Positioning to Create Columns

Positioning can also be used to create a multicolumn layout. Typically, the approach will be to absolute position the elements that were floated in the examples from the previous section. Recall that absolute positioning is related to the nearest positioned ancestor, so this approach typically uses some type of container that establishes the positioning context. Figure 5.24 illustrates a typical three-column layout implemented via positioning.



**HANDS-ON
EXERCISES**

LAB 5 EXERCISE
Three-Column Layout

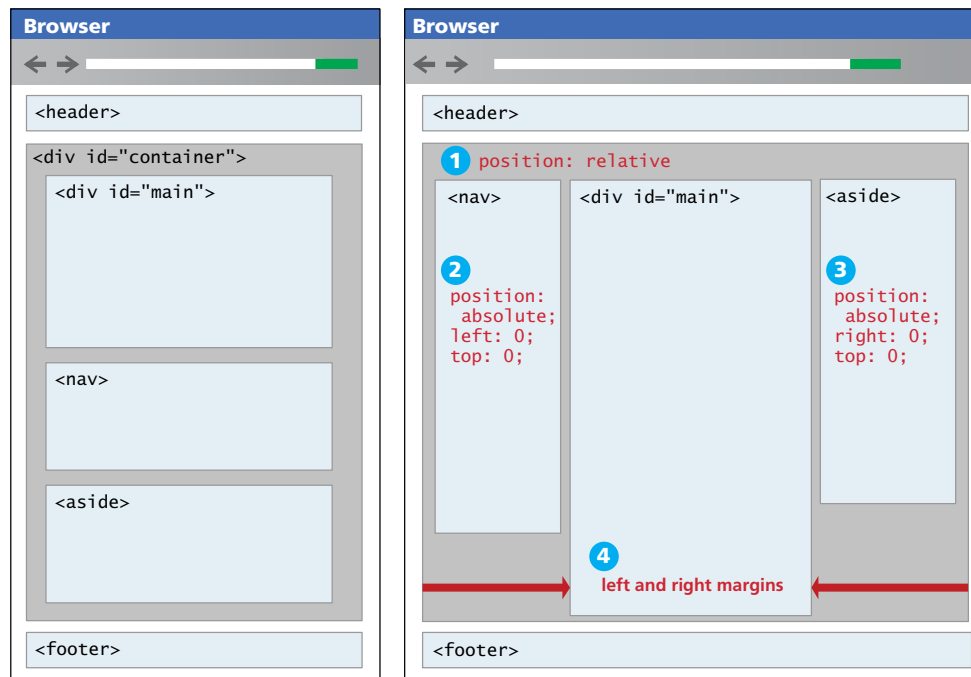


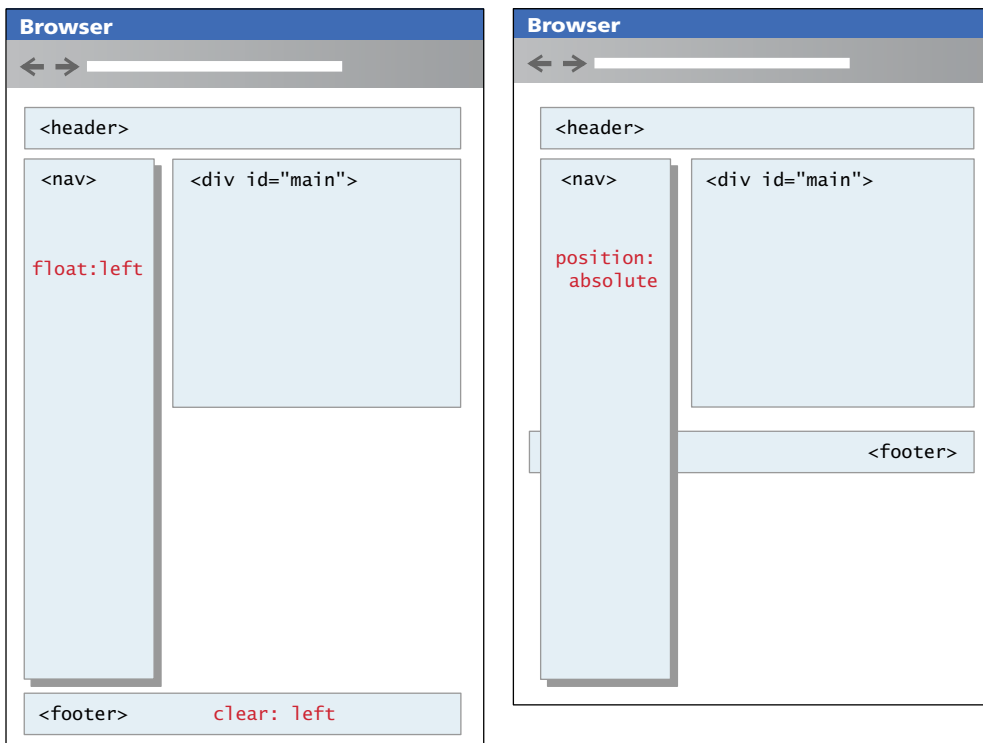
FIGURE 5.24 Three-column layout with positioning

Notice that with positioning it is easier to construct our source document with content in a more SEO-friendly manner; in this case, the main `<div>` can be placed first.

However, absolute positioning has its own problems. What would happen if one of the sidebars had a lot of content and was thus quite long? In the floated layout, this would not be a problem at all, because when an item is floated, blank space is left behind. But when an item is positioned, it is removed entirely from normal flow, so subsequent items will have no “knowledge” of the positioned item. This problem is illustrated in Figure 5.25.

One solution to this type of problem is to place the footer within the main container, as shown in Figure 5.26. However, this has the problem of a footer that is not at the bottom of the page.

As you can see, creating layouts with floats and positioning has certain strengths and weaknesses. While there are other ways to construct multicolumn layouts with CSS3 (table design mode, grid layout), there is still quite uneven support for these layout modes, even in newer browsers.



Elements that are floated leave behind space for them in the normal flow. We can also use the `clear` property to ensure later elements are below the floated element.

Absolute positioned elements are taken completely out of normal flow, meaning that the positioned element may overlap subsequent content. The `clear` property will have no effect since it only responds to floated elements.

FIGURE 5.25 Problems with absolute positioning

5.5 Approaches to CSS Layout

One of the main problems faced by web designers is that the size of the screen used to view the page can vary quite a bit. Some users will visit a site on a 21-inch wide screen monitor that can display 1920×1080 pixels (px); others will visit it on an older iPhone with a 3.5 screen and a resolution of 320×480 px. Users with the large monitor might expect a site to take advantage of the extra size; users with the small monitor will expect the site to scale to the smaller size and still be usable. Satisfying both users can be difficult; the approach to take for one type of site content might not work as well with another site with different content. Most designers take one of two basic approaches to dealing with the problems of screen size. While there are other approaches than these two, the others are really just enhancements to these two basic models.



FIGURE 5.26 Solution to footer problem

5.5.1 Fixed Layout

The first approach is to use a fixed layout. In a **fixed layout**, the basic width of the design is set by the designer, typically corresponding to an “ideal” width based on a “typical” monitor resolution. A common width used is something in the 960 to 1000 pixel range, which fits nicely in the common desktop monitor resolution (1024 × 768). This content can then be positioned on the left or the center of the monitor.

Fixed layouts are created using pixel units, typically with the entire content within a `<div>` container (often named “container”, “main”, or “wrapper”) whose width property has been set to some width, as shown in Figure 5.27.

The advantage of a fixed layout is that it is easier to produce and generally has a predictable visual result. It is also optimized for typical desktop monitors; however, as more and more user visits are happening via smaller mobile devices, this advantage might now seem to some as a disadvantage. Fixed layouts have other drawbacks. For larger screens, there may be an excessive amount of blank space to the left and/or right of the content. Much worse is when the browser window

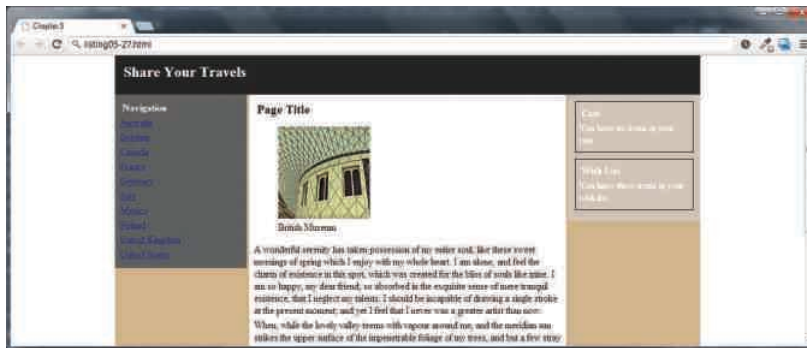


960px

Extra space to right

```
div#wrapper {
  width: 960px;
  background-color: tan;
}
```

```
<body>
  <div id="wrapper">
    <header>
      ...
    </header>
    <div id="main">
      ...
    </div>
    <footer>
      ...
    </footer>
  </div>
</body>
```



960px

Equal space to the left and to right

```
div#wrapper {
  width: 960px;
  margin-left: auto;
  margin-right: auto;
  background-color: tan;
}
```

FIGURE 5.27 Fixed layouts

shrinks below the fixed width; the user will have to horizontally scroll to see all the content, as shown in Figure 5.28.

5.5.2 Liquid Layout

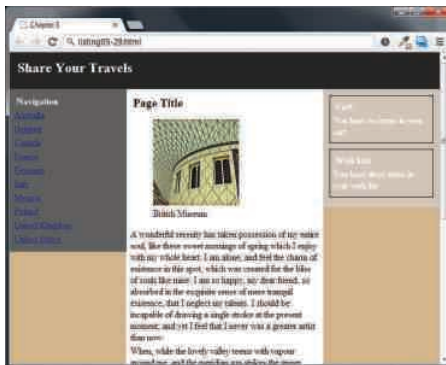
The second approach to dealing with the problem of multiple screen sizes is to use a **liquid layout** (also called a **fluid layout**). In this approach, widths are not specified using pixels, but percentage values. Recall from Chapter 3 that percentage values in CSS are a percentage of the current browser width, so a layout in which all widths are expressed as percentages should adapt to any browser size, as shown in Figure 5.29.

The obvious advantage of a liquid layout is that it adapts to different browser sizes, so there is neither wasted white space nor any need for horizontal scrolling.



The problem with fixed layouts is that they don't adapt to smaller viewports.

FIGURE 5.28 Problems with fixed layouts



Fluid layouts are based on the browser window.

However, elements can get too spread out as browser expands.

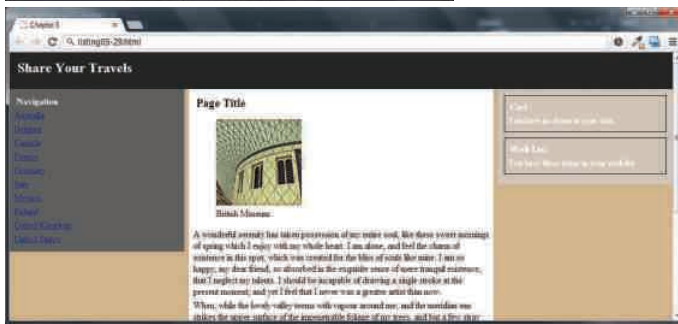


FIGURE 5.29 Liquid layouts

There are several disadvantages however. Liquid layouts can be more difficult to create because some elements, such as images, have fixed pixel sizes. Another problem will be noticeable as the screen grows or shrinks dramatically, in that the line length (which is an important contributing factor to readability) may become too long or too short. Thus, creating a usable liquid layout is generally more difficult than creating a fixed layout.

5.5.3 Other Layout Approaches

While the fixed and liquid layouts are the two basic paradigms for page layout, there are some other approaches that combine the two layout styles. You can find out more about them in most dedicated CSS books. Most of these other approaches are a type of **hybrid layout**, in that they combine pixel and percentage measurements. Fixed pixel measurements might make sense for a sidebar column containing mainly graphic advertising images that must always be displayed and which always are the same width. But percentages would make more sense for the main content or navigation areas, with perhaps min and max size limits in pixels set for the navigation areas. Unfortunately, this mixing of percentages, em units, and pixels can be quite complicated, so it is beyond the scope of this book to cover in detail.

5.6 Responsive Design

In the past several years, a lot of attention has been given to so-called responsive layout designs. In a **responsive design**, the page “responds” to changes in the browser size that go beyond the width scaling of a liquid layout. One of the problems of a liquid layout is that images and horizontal navigation elements tend to take up a fixed size, and when the browser window shrinks to the size of a mobile browser, liquid layouts can become unusable. In a responsive layout, images will be scaled down and navigation elements will be replaced as the browser shrinks, as can be seen in Figure 5.30.

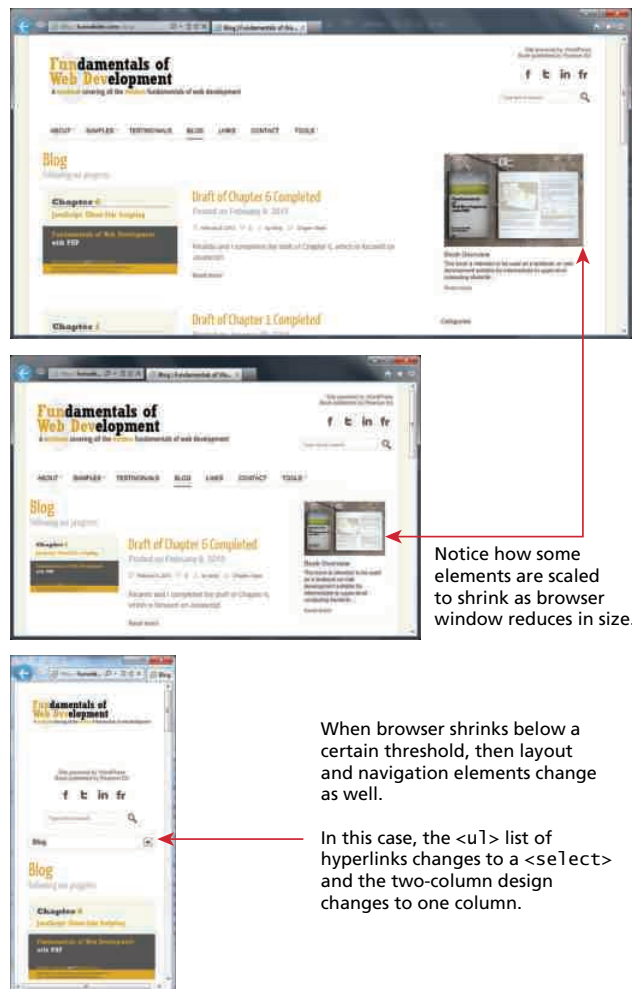


FIGURE 5.30 Responsive layouts



NOTE

One of the most influential recent approaches to web design is sometimes referred to as **mobile first design**. As the name suggests, the main principle in this approach is that the first step in the design and implementation of a new web site should be the design and development of its mobile version (rather than as an afterthought as is often the case).

The rationale for the mobile-first approach lies not only in the increasingly larger audience whose principal technology for accessing websites is a smaller device such as a phone or a tablet. Focusing first on the mobile platform also forces the designers and site architects to focus on the most important component of any site: the content. Due to the constrained sizes of these devices, the key content must be highlighted over the many extraneous elements that often litter the page for sites designed for larger screens.

There are now several books devoted to responsive design, so this chapter can only provide a very brief overview of how it works. There are four key components that make responsive design work. They are:

1. Liquid layouts
2. Scaling images to the viewport size
3. Setting viewports via the <meta> tag
4. Customizing the CSS for different viewports using media queries

Responsive designs begin with a liquid layout, that is, one in which most elements have their widths specified as percentages. Making images scale in size is actually quite straightforward, in that you simply need to specify the following rule:

```
img {  
    max-width: 100%;  
}
```

Of course this does not change the downloaded size of the image; it only shrinks or expands its visual display to fit the size of the browser window, never expanding beyond its actual dimensions. More sophisticated responsive designs will serve different sized images based on the viewport size.

5.6.1 Setting Viewports

A key technique in creating responsive layouts makes use of the ability of current mobile browsers to shrink or grow the web page to fit the width of the screen. If you



HANDS-ON
EXERCISES

LAB 5 EXERCISE

Setting the Viewport

- 1 Mobile browser renders web page on its viewport



960px
Mobile browser viewport

- 2 It then scales the viewport to fit within its actual physical screen



320px
Mobile browser screen

FIGURE 5.31 Viewports

have ever used a modern mobile browser, you may have been surprised to see how the web page was scaled to fit into the small screen of the browser. The way this works is the mobile browser renders the page on a canvas called the **viewport**. On iPhones, for instance, the viewport width is 980 px, and then that viewport is scaled to fit the current width of the device (which can change with orientation and with newer versions that have more physical pixels in the screen), as shown in Figure 5.31.

The mobile Safari browser introduced the viewport `<meta>` tag as a way for developers to control the size of that initial viewport. If the developer has created a responsive site similar to that shown in Figure 5.30, one that will scale to fit a smaller screen, she may not want the mobile browser to render it on the full-size viewport. The web page can tell the mobile browser the viewport size to use via the viewport `<meta>` element, as shown in Listing 5.1.

```
<html>
<head>
  <meta name="viewport" content="width=device-width" />
```

LISTING 5.1 Setting the viewport

By setting the viewport as in this listing, the page is telling the browser that no scaling is needed, and to make the viewport as many pixels wide as the device screen width. This means that if the device has a screen that is 320 px wide, the viewport width will be 320 px; if the screen is 480 px (for instance, in landscape mode), then the viewport width will be 480 px. The result will be similar to that shown in Figure 5.32.

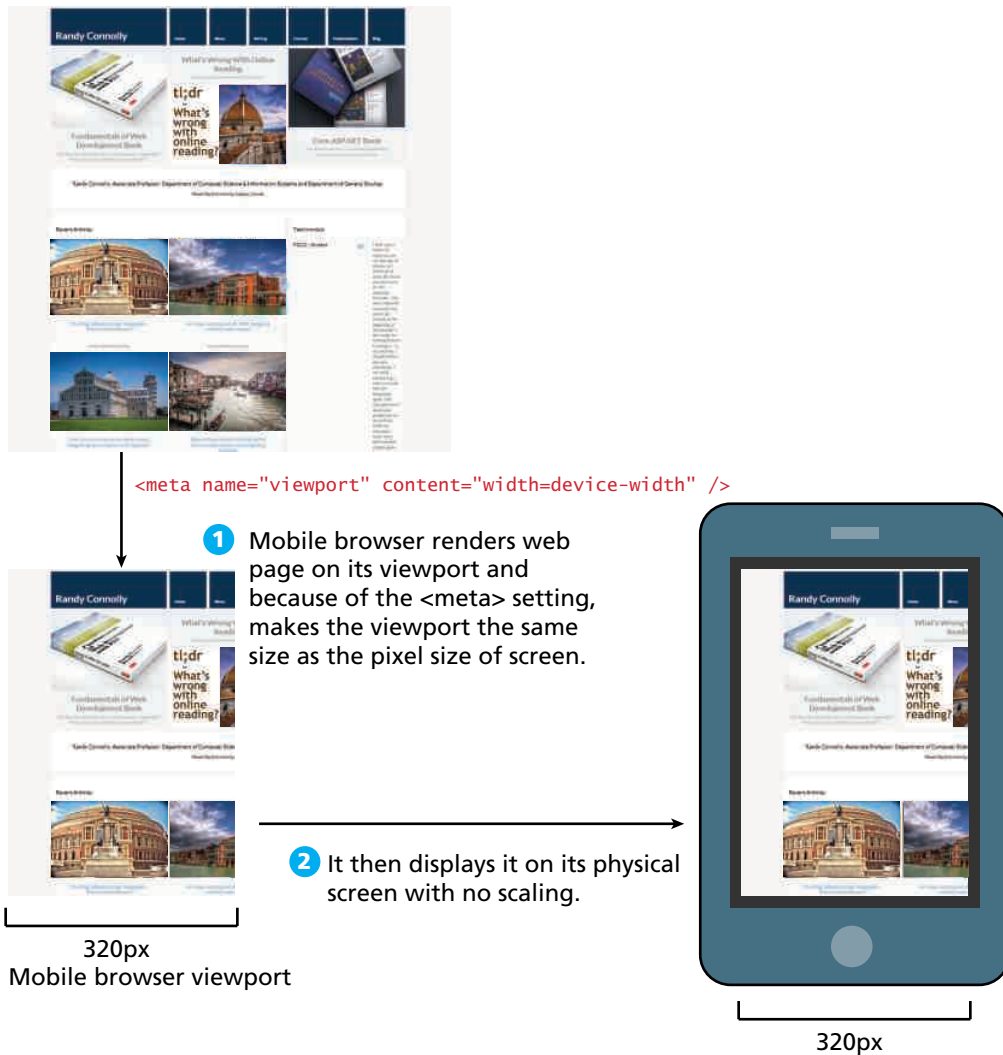


FIGURE 5.32 Setting the viewport

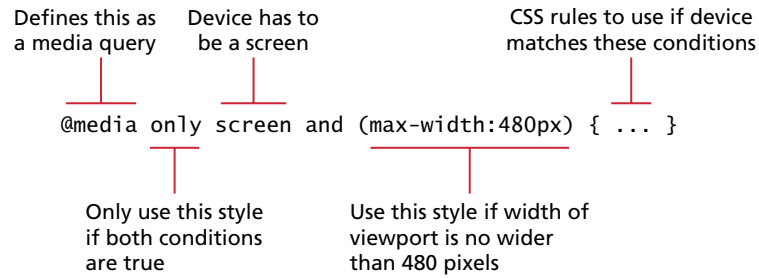


FIGURE 5.33 Sample media query

However, since *only* setting the viewport as in Figure 5.32 shrank but still cropped the content, setting the viewport is only one step in creating a responsive design. There needs to be a way to transform the look of the site for the smaller screen of the mobile device, which is the job of the next key component of responsive design, media queries.



NOTE

It is worth emphasizing that what Figure 5.31 illustrates is that if an alternate viewport is not specified via the `<meta>` element, then the mobile browser will try to render a shrunk version of the full desktop site.

5.6.2 Media Queries

The other key component of responsive designs is **CSS media queries**. A media query is a way to apply style rules based on the medium that is displaying the file. You can use these queries to look at the capabilities of the device, and then define CSS rules to target that device. Unfortunately, media queries are not supported by Internet Explorer 8 and earlier.

Figure 5.33 illustrates the syntax of a typical media query. These queries are Boolean expressions and can be added to your CSS files or to the `<link>` element to conditionally use a different external CSS file based on the capabilities of the device.

Table 5.3 is a partial list of the browser features you can examine with media queries. Many of these features have `min-` and `max-` versions.

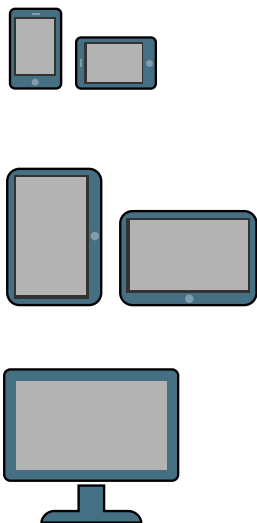
Contemporary responsive sites will typically provide CSS rules for phone displays first, then tablets, then desktop monitors, an approach called **progressive enhancement**, in which a design is adapted to progressively more advanced devices, an approach you will also see in the JavaScript chapter. Figure 5.34 illustrates how a responsive site might use media queries to provide progressive enhancement.

Notice that the smallest device is described first, while the largest device is described last. Since later (in the source code) rules override earlier rules, this

HANDS-ON
EXERCISESLAB 5 EXERCISE
Media Queries

Feature	Description
width	Width of the viewport
height	Height of the viewport
device-width	Width of the device
device-height	Height of the device
orientation	Whether the device is portrait or landscape
color	The number of bits per color

TABLE 5.3 Browser Features You Can Examine with Media Queries



styles.css

```
/* rules for phones */
@media only screen and (max-width:480px)
{
  #slider-image { max-width: 100%; }
  #flash-ad { display: none; }
  ...
}

/* CSS rules for tablets */
@media only screen and (min-width: 481px)
and (max-width: 768px)
{
  ...
}

/* CSS rules for desktops */
@media only screen and (min-width: 769px)
{
  ...
}
```

Instead of having all the rules in a single file, we can put them in separate files and add media queries to `<link>` elements.

```
<link rel="stylesheet" href="mobile.css" media="screen and (max-width:480px)" />
<link rel="stylesheet" href="tablet.css" media="screen and (min-width:481px)
and (max-width:768px)" />
<link rel="stylesheet" href="desktop.css" media="screen and (min-width:769px)" />
```

```
<!--[if lt IE 9]>
<link rel="stylesheet" media="all" href="style-ie.css"/>
<![endif]-->
```

Handles Internet Explorer 8 and earlier using IE conditional comments.

FIGURE 5.34 Media queries in action

provides progressive enhancement, meaning that as the device grows you can have CSS rules that take advantage of the larger space. Notice as well that these media queries can be within your CSS file or within the `<link>` element; the later requires more HTTP requests but results in more manageable CSS files.

5.7 CSS Frameworks

At this point in your CSS education you may be thinking that CSS layouts are quite complicated and difficult. You are not completely wrong; many others have struggled to create complex (and even not so complex) layouts with CSS. Larger web development companies often have several dedicated CSS experts who handle this part of the web development workflow. Smaller web development companies do not have this option, so as an alternative to mastering the many complexities of CSS layout, they instead use an already developed CSS framework.

A **CSS framework** is a precreated set of CSS classes or other software tools that make it easier to use and work with CSS. They are two main types of CSS framework: grid systems and CSS preprocessors.

5.7.1 Grid Systems

Grid systems make it easier to create multicolumn layouts. There are many CSS grid systems; some of the most popular are Bootstrap (twitter.github.com/bootstrap), Blueprint (www.blueprintcss.org), and 960 (960.gs). Most provide somewhat similar capabilities. The most important of these capabilities is a grid system.

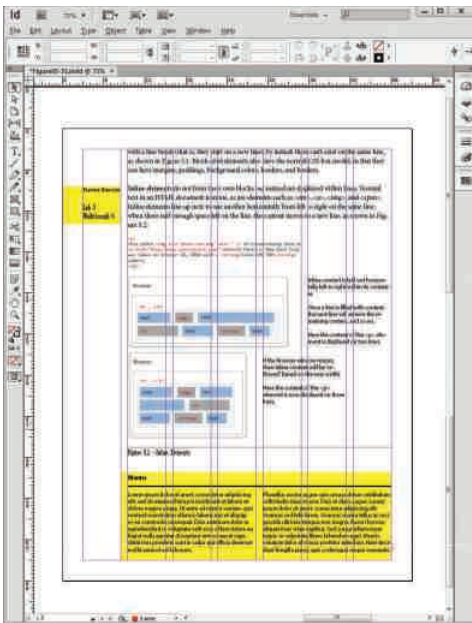
Print designers typically use grids as a way to achieve visual uniformity in a design. In print design, the very first thing a designer may do is to construct, for instance, a 5- or 7- or 12-column grid in a page layout program like InDesign or Quark Xpress. The rest of the document, whether it be text or graphics, will be aligned and sized according to the grid, as shown in Figure 5.35.

CSS frameworks provide similar grid features. The 960 framework uses either a 12- or 16-column grid. Bootstrap uses a 12-column grid. Blueprint uses a 24-column grid. The grid is constructed using `<div>` elements with classes defined by the framework. The HTML elements for the rest of your site are then placed within these `<div>` elements. For instance, Listing 5.2 illustrates a three-column layout similar to Figure 5.22 within the grid system of the 960 framework, while Listing 5.3 shows the same thing in the Bootstrap framework. In both systems, elements are laid out in rows; elements in a row will span from 1 to 12 columns. In the 960 system, a row is terminated with `<div class="clear"></div>`. In Bootstrap, content must be placed within the `<div class="row">` row container.

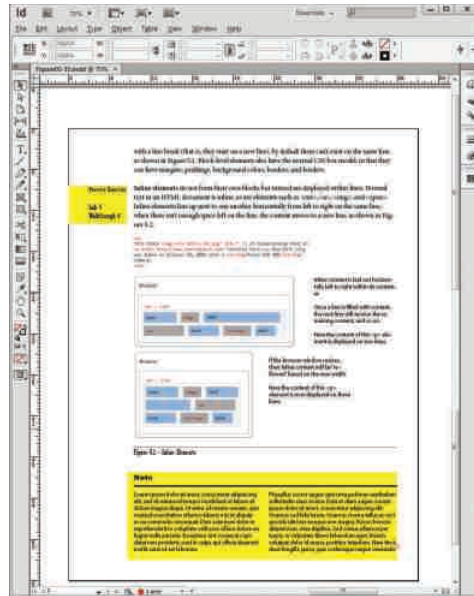


HANDS-ON EXERCISES

LAB 5 EXERCISE Using Bootstrap



Most page design begins with a grid. In this case, a seven-column grid is being used to layout page elements in Adobe InDesign.



Without the gridlines visible, the elements on the page do not look random, but planned and harmonious.

FIGURE 5.35 Using a grid in print design

```
<head>
  <link rel="stylesheet" href="reset.css" />
  <link rel="stylesheet" href="text.css" />
  <link rel="stylesheet" href="960.css" />
</head>
<body>
  <div class="container_12">
    <div class="grid_2">
      left column
    </div>
    <div class="grid_7">
      main content
    </div>
    <div class="grid_3">
      right column
    </div>
    <div class="clear"></div>
  </div>
</body>
```

LISTING 5.2 Using the 960 grid

```
<head>
  <link href="bootstrap.css" rel="stylesheet">
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-md-2">
        left column
      </div>
      <div class="col-md-7">
        main content
      </div>
      <div class="col-md-3">
        right column
      </div>
    </div>
  </div>
</body>
```

LISTING 5.3 Using the Bootstrap grid

Both of these frameworks allow columns to be nested, making it quite easy to construct the most complex of layouts. As well, modern CSS frameworks are also responsive, meaning that some of the hard work needed to create a response site has been done for you. Because of this ease of construction, this book's examples will often make use of a grid framework. However, CSS frameworks may reduce your ability to closely control the styling on your page, and conflicts may occur when multiple CSS frameworks are used together.

We will be using the Bootstrap framework, which is an open-source system, but was originally created by the designers at Twitter. Bootstrap provides more than just a grid system. It also has a wide variety of very useful additional styling classes, such as classes for drop-down menus, fancy buttons and form elements, and integration with a variety of jQuery plug-ins. Figure 5.36 illustrates two example pages created using nothing but the built-in classes that come with the Bootstrap framework (that is, no additional CSS was defined).

5.7.2 CSS Preprocessors

CSS preprocessors are tools that allow the developer to write CSS that takes advantage of programming ideas such as variables, inheritance, calculations, and functions. A CSS preprocessor is a tool that takes code written in some type of preprocessed language and then converts that code into normal CSS.

The advantage of a CSS preprocessor is that it can provide additional functionalities that are not available in CSS. One of the best ways to see the power

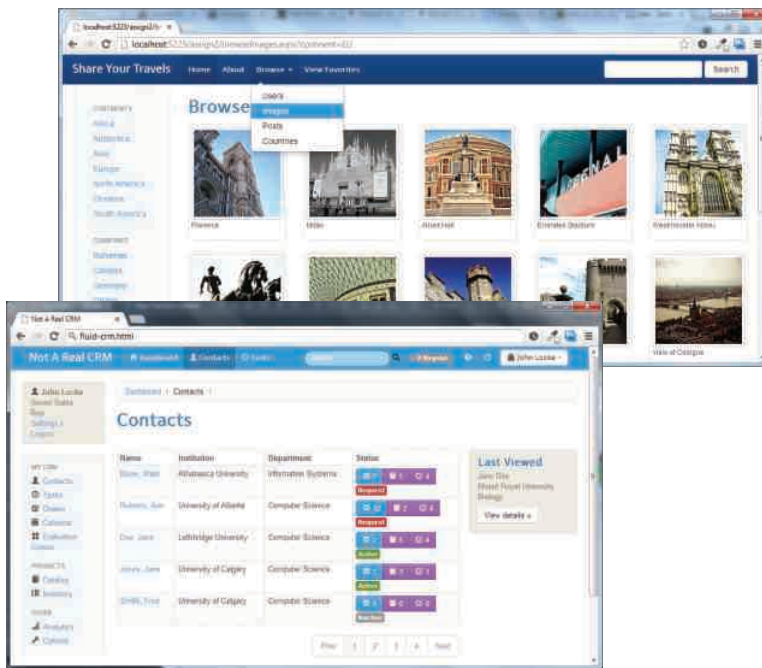


FIGURE 5.36 Examples using just built-in Bootstrap classes

of a CSS preprocessor is with colors. Most sites make use of some type of color scheme, perhaps four or five colors. Many items will have the same color. For instance, in Figure 5.37, the background color of the `.box` class, the text color in the `<footer>` element, the border color of the `<fieldset>`, and the text color for placeholder text within the `<textarea>` element, might all be set to `#796d6d`. The trouble with regular CSS is that when a change needs to be made (perhaps the client likes `#e8cfcf` more than `#796d6d`), then some type of copy and replace is necessary, which always leaves the possibility that a change might be made to the wrong elements. Similarly, it is common for different site elements to have similar CSS formatting, for instance, different boxes to have the same padding. Again, in normal CSS, one has to use copy and paste to create that uniformity.

In a programming language, a developer can use variables, nesting, functions, or inheritance to handle duplication and avoid copy-and-pasting and search-and-replacing. CSS preprocessors such as LESS, SASS, and Stylus provide this type of functionality. Figure 5.37 illustrates how a CSS preprocessor (in this case SASS) is used to handle some of the just-mentioned duplication and change problems.

```

$colorSchemeA: #796d6d;
$colorSchemeB: #9c9c9c;
$paddingCommon: 0.25em;

footer {
  background-color: $colorSchemeA;
  padding: $paddingCommon * 2;
}

@mixin rectangle($colorBack, $colorBorder) {
  border: solid 1pt $colorBorder;
  margin: 3px;
  background-color: $colorBack;
}

fieldset {
  @include rectangle($colorSchemeB, $colorSchemeA);
}

.box {
  @include rectangle($colorSchemeA, $colorSchemeB);
  padding: $paddingCommon;
}

```

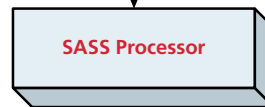
SASS source file, e.g. source.scss

This example uses SASS (Syntactically Awesome Stylesheets). Here three variables are defined.

You can reference variables elsewhere. SASS also supports math operators on its variables.

A mixin is like a function and can take parameters. You can use mixins to encapsulate common styling.

A mixin can be referenced/called and passed parameters.



The processor is some type of tool that the developer would run.

```

footer {
  padding: 0.50em;
  background-color: #796d6d;
}

fieldset {
  border: solid 1pt #796d6d;
  margin: 3px;
  background-color: #9c9c9c;
}

.box {
  border: solid 1pt #9c9c9c;
  margin: 3px;
  background-color: #796d6d;
  padding: 0.25em;
}

```

Generated CSS file, e.g., styles.css

The output from the processor is a normal CSS file that would then be referenced in the HTML source file.

FIGURE 5.37 Using a CSS preprocessor

In Chapter 8, you will learn about server-side languages such as PHP and ASP.NET. One way to think of these server-side environments is that they are a type of preprocessor for HTML. In reality, most real-world sites are not created as static HTML pages, but use programs running on the server that output HTML. CSS preprocessors are analogous: they are programs that generate CSS, and perhaps in a few years, it will be much more common for developers to use them, just as today, most developers use an HTML preprocessor like PHP or ASP.NET.

5.8 Chapter Summary

This chapter has covered the sometimes complicated topics of CSS layout. It began with the building blocks of layout in CSS: positioning and floating elements. The chapter also examined different approaches to creating page layouts as well as the recent and vital topic of responsive design. The chapter ended by looking at different types of CSS frameworks that can simplify the process of creating complex CSS designs.

5.8.1 Key Terms

absolute positioning	float property	positioning context
block-level elements	fluid layout	progressive enhancement
clear property	grid systems	relative positioning
containing block	hybrid layout	replaced inline elements
CSS framework	inline elements	responsive design
CSS media queries	liquid layout	viewport
CSS preprocessors	nonreplaced inline elements	z-index
fixed layout		
fixed positioning	normal flow	

5.8.2 Review Questions

1. Describe the differences between relative and absolute positioning.
2. What is normal flow in the context of CSS?
3. Describe how block-level elements are different from inline elements. Be sure to describe the two different types of inline elements.
4. In CSS, what does floating an element do? How do you float an element?
5. In CSS positioning, the concept of a positioning context is important. What is it and how does it affect positioning? Provide an example of how positioning context might affect the positioning of an element.
6. Briefly describe the two ways to construct multicolumn layouts in CSS.

7. Write the CSS and HTML to create a two-column layout using positioning and floating.
8. Briefly describe the differences between fixed, liquid, and hybrid layout strategies.
9. What is responsive design? Why is it important?
10. What are the advantages and disadvantages of using a CSS framework.
11. Explain the role of CSS preprocessors in the web development work flow.

5.8.3 Hands-On Practice

PROJECT 1: Share Your Travel Photos

DIFFICULTY LEVEL: Basic



**HANDS-ON
EXERCISES**

PROJECT 5.1

Overview

Demonstrate your proficiency with CSS floats along with margins and padding by modifying `chapter05-project01.css` so that `chapter05-project01.html` looks similar to that shown in Figure 5.38.

Instructions

1. Examine `chapter05-project01.html` in the browser. The HTML does not need to be modified for this project.
2. Change the margins and padding of the `<article>` element. For most of the margins, paddings, widths, and heights, you should use `em` units.
3. Use the techniques from Sections 5.3.3 and 5.3.4 to display the content within the `<article>`. The `<footer>` uses the same float techniques. It is okay to use pixel units for the overlapping elements.
4. The colors used in this example are: `#F5F5F5`, `#FF8800`, and `#474747`. You are also free to use whatever colors you like.

Testing

1. Be sure to test in more than one browser and also try increasing/decreasing the browser zoom level. If you have used `em` units for font sizes and most margin and padding values, it should scale to the different zoom levels.

PROJECT 2: Book Rep Customer Relations Management

DIFFICULTY LEVEL: Intermediate



**HANDS-ON
EXERCISES**

PROJECT 5.2

Overview

Demonstrate your proficiency with absolute positioning and floats by modifying `chapter05-project02.css` so that `chapter05-project02.html` looks similar to that shown in Figure 5.39.

Instructions

1. Examine `chapter05-project01.html` in the browser. The HTML does not need to be modified for this project.

2. You will create the three-column layout using absolute positioning and margin settings as described in Section 5.4.1.
3. Within the main column, the company and client addresses will use floats rather than positioning.
4. Within the boxes for recent messages, weekly changes, and top sellers, you will need to use floats, block display, and padding values.

Testing

1. Be sure to test by increasing/decreasing the browser zoom level. If you have used `em` units for font sizes and most margin and padding values, it should scale to the different zoom levels.

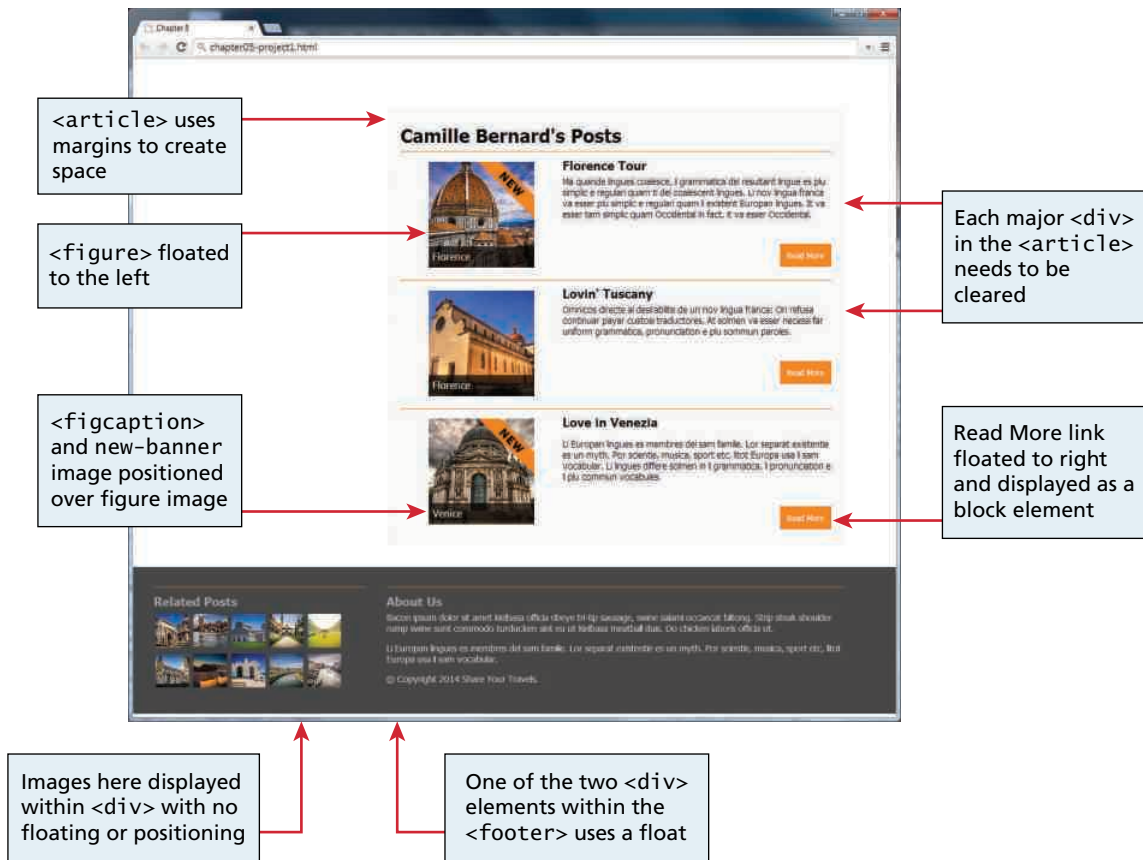


FIGURE 5.38 Completed Project 1

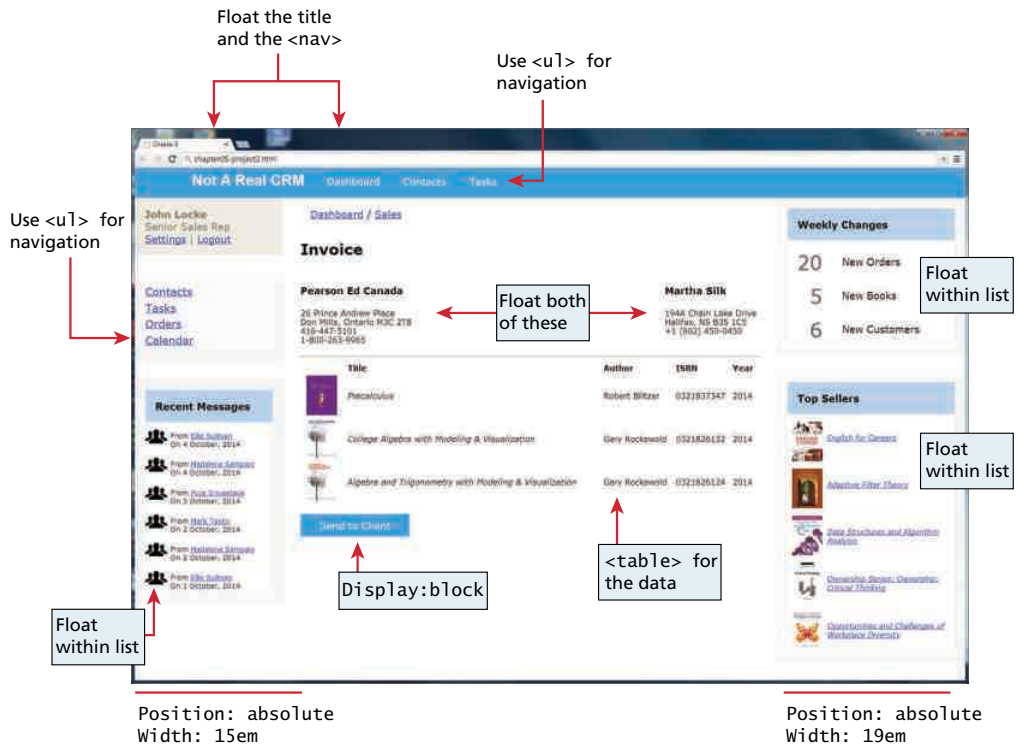


FIGURE 5.39 Completed Project 2

PROJECT 3: Art Store

DIFFICULTY LEVEL: Advanced

Overview

Use the Bootstrap CSS framework (included and available from the web) as well as modify `chapter05-project03.css` and `chapter05-project03.html` so it looks similar to that shown in Figure 5.40.

Instructions

1. Examine `chapter05-project03.html` in the browser. You will need to add a fair bit of HTML in accordance with the Bootstrap documentation. Since you can use the various Bootstrap classes, you will need to write very little CSS (the solution shown in Figure 5.40 has fewer than ten rules defined).
2. The first step will be defining the basic structure. Figure 5.40 shows that most of the content is contained within a main row (i.e., below the navbars and above the footer) that is composed of two columns (one 10 wide, the other 2 wide). The Bootstrap grid classes (e.g., `col-md-10`) are shown at the top of the figure. One of the columns has a nested row within it that contains the painting image and the data on the painting.

**HANDS-ON EXERCISES****PROJECT 5.3**

3. Figure 5.40 identifies the other Bootstrap components that are used in this project. You will need to use the online Bootstrap documentation for more information on how to use these components

Testing

1. Be sure to test by increasing/decreasing the size of the browser window. If you shrink the browser window sufficiently it should use the built-in Bootstrap media queries to adapt nicely to the smaller window size. This will require you to construct the navbars with the appropriate collapse classes.

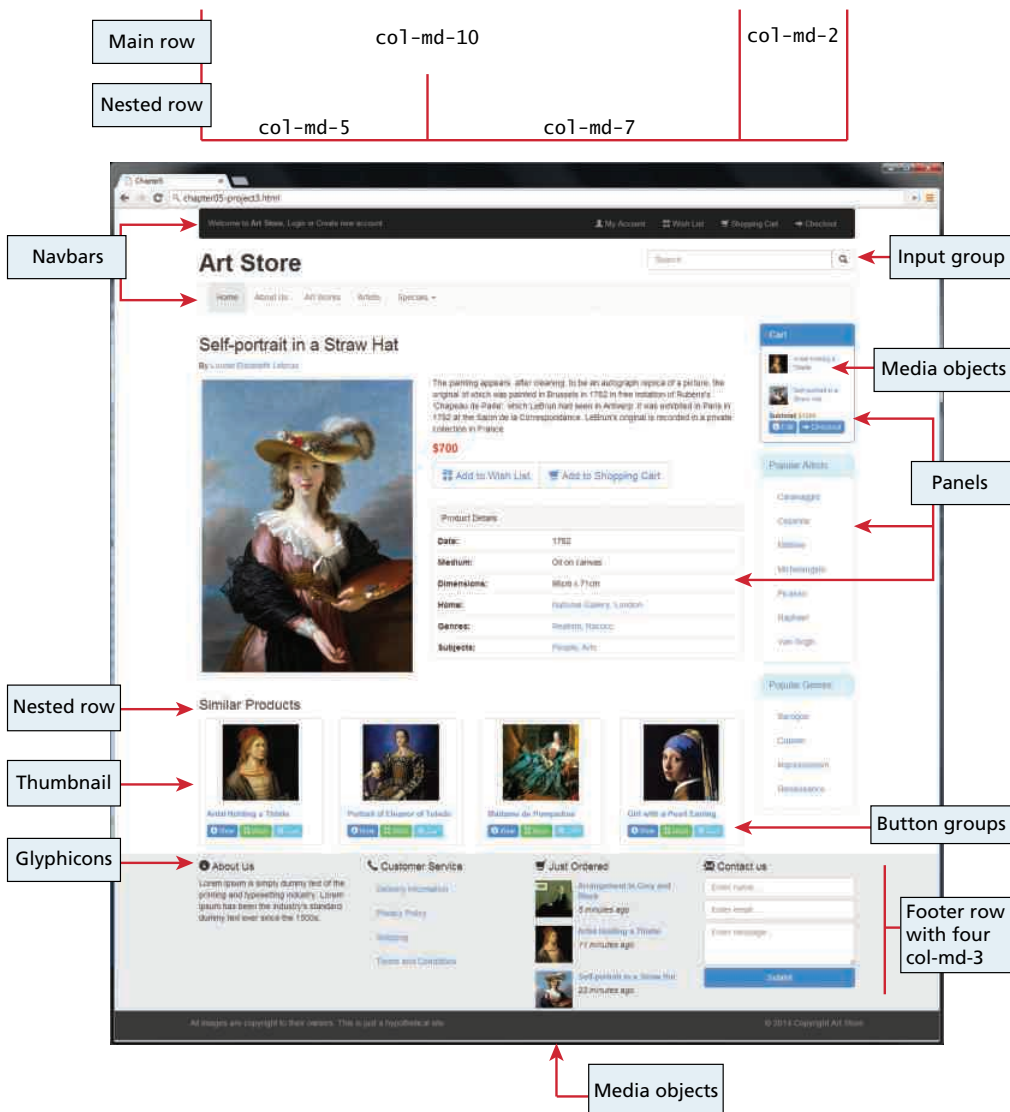


FIGURE 5.40 Completed Project 3

<https://hemanthrajhemu.github.io>