Chapter 17

More about HTML and CSS

Intended Learning Outcomes

After completing this chapter, you should be able to:

- Incorporate additional elements into your HTML forms (hidden fields, radio buttons, single and multiple checkboxes, text areas, and reset buttons).
- Distinguish between data validation and data sanitizing.
- Recognize and implement ordered, unordered (bulleted), and definition lists.
- Add horizontal rules of varying lengths and heights to a Web page.
- Utilize the HTML <blockquote> and <cite> elements.
- Apply background images to your Web pages.
- Use clickable images to provide links to other Web pages.
- Use and containers to apply styles to specific sections of a Web page.
- Include HTML entities in your page content to render special characters.

HTML and CSS were introduced in Chapter 4, with the goal of providing sufficient coverage of these languages to support the overall purpose of the book (to teach fundamental programming algorithms, syntax and control structures). Chapter 17 is designed to supplement Chapter 4 by introducing other HTML elements: additional form elements (**hidden fields**, **radio buttons**, **check boxes**, **text areas**, and **reset buttons**); different kinds of lists (**ordered**, **unordered**, and **definition** lists); **horizontal rules**; the <**block-quote**> and <**cite**> elements (used to identify quotes from other sources, and titles of creative works); the <**div**> and <**span**> elements (used to assign CSS styles to specific sections of a Web page), and **html entities** (used to render special characters). In your **samples** folder you will find a **more-html** folder which contains the examples that are used in this chapter.

The chapter ends with a brief discussion of deprecated HTML tags, and a list of useful references

More About HTML Forms

In Chapter 4 you learned to create forms that contained **input text boxes**, **drop down lists**, and **submit buttons**. HTML forms can also include **hidden fields**, **radio buttons**, **check boxes**, **text areas**, and **reset buttons**. These are explained here, and the **more-html** folder (under your **samples** folder) includes an example that demonstrates the use of these elements in an application (**pizza-order.php** and **confirm-order.php**).

Hidden Fields

You have learned that drop down lists and input text boxes should include a **name** attribute. When the form is submitted to Web server each **value** is assigned to the **$_POST** array to an element indexed with the same **name**. Consider this code in an HTML form:

```
<label>Your first name:
<input type="text" name="custName" size = "10"></label>
```

The PHP program that processes the form can receive the value submitted from this input box as follows:

```
$customerName = $_POST["custName"];
```

In addition to input provided by the user, it is often useful to code a form to also submit other values that might be needed to process the form input. For example, a form designed to allow a user to order a pizza for pickup might also need to submit the **price** of the pizza when the form is submitted, which is not a value that the user should provide!

Forms can include **hidden fields**. These fields are defined with names and values that will be included when the form data is submitted to the $_POST array, although the values were not provided by the user. Hidden fields are created using an <**input**> tag, but with the **type** attribute set to **"hidden"**, and the **value** attribute set to the required value, for example:

```
<input type="hidden" name="price" value="12.75">
```

Although the user does not see this field, the **name** and **value** from the field is included when the form is submitted. In this example, the value **12.75** is stored in the

$_POST[**"price"**] element, and can be retrieved in the same way as any other form input, for example:

```
$pizzaPrice = $_POST["price"];
```

This example shows a **literal** value (**12.75**) being sent in a hidden field. A value stored in a **variable** can also be sent. For example it is probably important to send the **time** that the order was submitted. Chapter 16 demonstrated how to use the PHP **date**() function to obtain the current time in any format; the following assignment will assign the current time to a variable named **$time**, in the format "**2:30pm**" (or whatever the time is):

```
$time = date("g:ia");
```

The form could include a hidden field that submits the value stored in **$time** as follows

```
<input type="hidden" name="timeOfOrder" value=$time>
```

And once again this value can be retrieved from the $_POST array as follows:

```
$timeOfOrder = $_POST["timeOfOrder"];
```

Note that the **value** stored in $time was sent to the $_POST array, associated with the name "**timeOfOrder**"; the actual variable **$time** is **not** sent.

A single HTML form can combine any number of hidden fields, along with fields designed to receive input from the user.

Radio Buttons

Radio buttons provide the user with a list of options, only one of which can be chosen. If the user clicks a button it is **selected**, and any previously selected button is **deselected**. The form in our pizza order example asks the user to choose a **crust** from three options (**thin**, **thick**, or **stuffed**). These could be displayed as a drop down list, but radio buttons are more visually intuitive. Radio buttons are created using an <**input**> element with the type set to "**radio**". Radio buttons are associated with one another by using the **same name for each button** (this is important because you may have more than one group of radio buttons in your form: each group of radio buttons would use a different name). Although each button has the same name, they must each have a different **value**: the value of the button that is selected by the user will be sent to the Web server along with the name. Here is the example from **pizza-order.php**:

```
<p>Which type of crust would you like?</p>

<label>Thin Crust
  <input type="radio" name="crust" value="thin"></label>
<label>Thick Crust
  <input type="radio" name="crust" value="thick"></label>
<label>Stuffed Crust
  <input type="radio" name="crust" value="stuffed"></label>
```

Each button's **label** text describes each option to the user, and the **value** attribute indicates the actual **value** that is to be sent to the **$_POST** array, with the **name** "crust" (if that button is selected). The program that receives the form input (in this case **confirm-order.php**) will retrieve the user's selection as follows:

```
$crust= $_POST["crust"];
```

In this example, **$_POST**[**"crust2"**] will contain either "thin", or "thick", or "stuffed".

Single and Multiple Check Boxes

Just like radio buttons, **check boxes also** provide the user with a list of options, but in this case with the option to select as many options as he or she wants, just by clicking boxes on and off. In the pizza example, an excellent use of check boxes will be to allow the user to choose any number of pizza toppings (is this making you hungry?). Sometimes a form component only requires a single check box, usually to obtain a **yes** or **no** response. For example our pizza form may also ask the user to check a single box to indicate that he or she has a coupon to get a free topping.

Check boxes are created using an <**input**> element with the type set to **"checkbox"**. If just a **single** check box is needed for a form component, then the element should have a **unique name**, and the **value** attribute can be used to send a value to the $_POST array if the check box is checked. It is important to note that **no** value will be sent if the check box is **not** checked. Here is the pizza order example, where the check box name is **"freeTopping"**, and the value **"yes"** is sent if the box is checked:

```
<label>Do you have a "Free Topping" Coupon?
  <input type="checkbox" name="freeTopping"
    value="yes"></label>
```

However the $_POST array will **only** include a "freeTopping" element if the freeTopping checkbox was **checked**. If **confirm-order.php** attempts to retrieve a value from **$_POST** [**"freeTopping"**], the processor may issue a warning if **$_POST** [**"freeTopping"**] doesn't exist. One way to avoid this is to use the **isset**() function to test if this array element has been set, and assign the result (**true** or **false**) of this test to a variable, for example:

```
  if (isset($_POST["freeTopping"]))
  {
    // code if freeTopping was set (the box was checked)
  }
  else
  {
    // code if freeTopping was not set (the box was NOT
checked)
  }
```

You may be wondering, if we are using the **isset**() function to test if **$_POST**[**"freeTopping"**] exists, why do we need to send a value at all? After all, if isset() returns **true** that in itself indicates that the check box **was** checked, and, if **false** then the checkbox was **not** checked. While there may be some instances where, when working with a single checkbox, it is useful to send a value, in most cases it is not really needed, so the input element could be coded with no **value** attribute:

```
  <label>Do you have a "Free Topping" Coupon?
    <input type="checkbox" name="freeTopping" </label>
```

Now let's look at a form requirement that requires a **group** of check boxes. The pizza order form must allow the user to select any number of pizza toppings. In this case, the check box for each topping must contain a unique value that will be passed to the $_POST array if that box is checked (for example "mushroom", or "olives"). Since the user can check any number of these boxes, how can you code this so that a list of the names of the selected toppings will be sent to the $_POST array? The answer is to submit an **array** of the checked toppings: the check boxes share the same name (just as with radio buttons), but the name now includes square brackets to indicate that the value of the check box will be sent as one element in an array:

```
  <label> Extra Cheese Topping
    <input type="checkbox" name="toppings[]"
value="cheese"></label>
  <label> Green Pepper Topping
    <input type="checkbox" name="toppings[]"
value="pepper"></label>
  <label> Mushroom Topping
    <input type="checkbox" name="toppings[]"
value="mushroom"></label>
  <label> Onion Topping
    <input type="checkbox" name="toppings[]"
value="onion"></label>
  <label> Olive Topping
    <input type="checkbox" name="toppings[]"
value="olives"></label>
```

In the example the five **toppings** check boxes all use the same name: "**toppings**[]". The inclusion of square brackets means that a numerically indexed array containing the values of every checked check box will be sent to the $_POST array with the name "**toppings**". However if **no** check boxes are checked the $_POST array will not receive this array, so once again you should use the **isset**() function to test that the array exists as an element in the $_POST array:

```
if (isset($_POST["toppings"]))
{
  $toppings = $_POST["toppings"]; //assign array to $toppings
  // code to execute if there are toppings
}
else
{
  // code to execute if there are no toppings
}
```

In this example, if $_POST["toppings"] is set, the array is assigned to a variable named **$toppings**, and this variable can then used to refer to the array. While it not required to copy **$_POST**["**toppings**"] to **$toppings**, it is less cumbersome, and more readable, to use **$toppings**, rather than **$_POST**["**toppings**"], in every program statement that works with the array.

If an array of toppings has been received, you can determine how many items are in the array using either the **count**() or **sizeof**() function (these two functions are the same), for example **count**(**$toppings**). You can also reference a specific array element using its index position (for example **$toppings**[2]), would refer to the **third** element of the array), but this would be dangerous since the array may only contain 1 or 2 elements (if the user only chose 1 or 2 toppings. A better approach to work with individual toppings is to use the **in_array**() function, which takes **two** arguments, a **search** value, and the **array** to be searched. So for example, if the user checked "**mushroom**" but not "**olives**", **in_array**("**mushroom**", **$toppings**) would return true, but in_array("olives", $toppings) would return false.

In many cases, you will be more interested in using a loop to work with **all** the values in the array. A **FOR** loop can be used for this purpose (using **count**(**$toppings**) to control the number of repetitions), but a **FOREACH** loop is designed for this purpose. The **confirm-pizza.php** program uses a **FOREACH** loop to simply display a list of the toppings that were ordered. Here is a simplified version of this code (the complete version will be shown shortly):

```
print ("<p>You ordered ");
foreach ($toppings as $topping) {
     print ("$topping and ");
}
     print ("hey that sounds truly delicious!</p>");
```

This code segment first begins a paragraph with the words "You ordered", and then, each time the loop repeats, the program prints the next topping in the **$toppings** array, followed the word " and ". Once the loop has completed, the paragraph ends with the phrase "hey that sounds truly delicious!". If the user had only checked olives this would produce " **You ordered olives and hey that sounds truly delicious!** ". If the user had selected cheese, mushroom, and olives, this would produce " **You ordered cheese and mushroom and olives and hey that sounds truly delicious!** "

Text Areas

Unlike the HTML **input text box** that displays as single line, a **text area** displays as a box containing multiple rows, and can be used to invite the user to enter a message of some kind, for example feedback, or a comment, or instructions. The pizza program will use a text area to allow the user to provide delivery instructions:

```
<label>Special delivery instructions?
  <textarea name="delivery" rows="2" cols="38"
   maxlength = "80"></textarea></label>
```

As always the **name** attribute is used to associate a name with the user input. The **rows** and **cols** attributes are used to set the displayed size of the text area (the size of the text area can also be defined in CSS). Note that the user can expand or diminish the size (also the text that the user types in the text area can be scrolled up and down). The number of characters that can be typed can be controlled with the optional **maxlength** attribute. The program that receives the form input (in this case **confirm-order.php**) will retrieve the user's message as follows:

```
$instructions= $_POST["delivery"];
```

Processing the Form

Let's see how all of these different form elements can work together in a simple pizza order application. Open **pizza-order.php** and try submitting some orders. **Figure 17-1** shows a sample interaction.

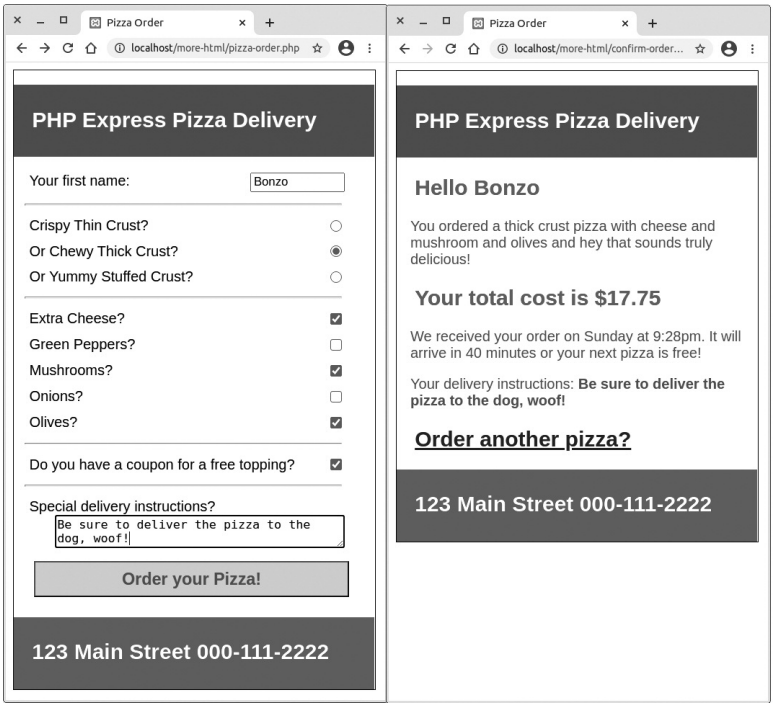

Figure 17-1: pizza-order.php and confirm-order.php screenshots

Here is **pizza-order.php** which displays the order form:

```
<form action= "confirm-order.php" method = "post">
<?php
   $time = date("l")." at ".date("g:ia");
   print ("<input type=\"hidden\"
      name=\"timeOfOrder\" value=\"$time\">");
 ?>

<input class="hidden" type="hidden" name="pizzaCost"
value="12.75">
<input class="hidden" type="hidden" name="toppingCost"
value="2.50">

<label>Your first name:
   <input type="text" name="custName" size = "10"></label>
```

```
<hr>
<label>Crispy Thin Crust?
  <input type="radio" name="crust" value="thin"></label>
<label>Or Chewy Thick Crust?
  <input type="radio" name="crust" value="thick"></label>
<label>Or Yummy Stuffed Crust?
  <input type="radio" name="crust" value="stuffed"></label>
<hr>
<label>Extra Cheese?
  <input type="checkbox" name="toppings[]"
value="cheese"></label>
<label> Green Peppers?
  <input type="checkbox" name="toppings[]"
value="pepper"></label>
<label>Mushrooms?
  <input type="checkbox" name="toppings[]"
value="mushroom"></label>
<label>Onions?
  <input type="checkbox" name="toppings[]" value="onion"></label>
<label>Olives?
  <input type="checkbox" name="toppings[]"
value="olives"></label>
<hr>
<label>Do you have a coupon for a free topping?
  <input type="checkbox" name="freeTopping"></label>
<hr>
<label>Special delivery instructions?
  <textarea name="delivery" rows="2" cols="38"
  maxlength = "80"></textarea></label>
<input type="submit" class="submit" value="Order your Pizza!" />
</form>
```

Look through this code and identify the various form elements that have been discussed above. The code for pizza-order.php also includes <hr> (hard rule) tags, which generate a line across the page, and tags (not shown here) which are used to assign different styles to different code sections. These and other HTML elements will be discussed later in this chapter.

The **confirm-pizza.php** program receives and processes the form input from this form. The $_POST array will contain: the user's name (text input box) , the time of the order (hidden field), the pizza cost (hidden field), the toppings cost (hidden field), the type of crust (radio button input), the list of toppings (array of checked check box values, but only if at least one topping was checked), the user's response when asked if he or she has a coupon (single check box input, but only if this box was checked), and special delivery instructions, if any (text area). The program then displays a Web page in response to the user's order that calculate and displays the cost of the pizza, which is

based on the number of toppings (allowing for no toppings) and includes a deduction if the user had a coupon. The program also tells the user what time the order was received, guarantees the pizza will be ready within 40 minutes of that time, and confirms the delivery instructions if any were provided.

Here is the code for confirm-pizza.php:

```
$timeOfOrder = $_POST["timeOfOrder"];
$pizzaCost = $_POST["pizzaCost"];
$toppingCost = $_POST["toppingCost"];
$customerName = $_POST["custName"];
$crust= $_POST["crust"];
$instructions = $_POST["delivery"];

print ("<h2>Hello ".$customerName."</h2>");
print ("<p>You ordered a $crust crust pizza with ");

if (isset($_POST["toppings"]))
{
  $toppings = $_POST["toppings"];

  foreach ($toppings as $topping) {
    print ("$topping and ");
  }
  print ("hey that sounds truly delicious!</p>");
  if (isset($_POST["freeTopping"]))
    $totalToppingCost = (count($toppings)- 1) *$toppingCost;
  else
    $totalToppingCost = count($toppings) *$toppingCost;
}
else
{
  print ("no toppings</p>");
  $totalToppingCost = 0;
}

$totalCost = $pizzaCost + $totalToppingCost;

print ("<h2>Your total cost is <strong>$".number_format($total-
Cost, 2)."</strong></h2>");

print ("<p>We received your order on ".$timeOfOrder.". It will
arrive in 40 minutes or your next pizza is free!</p>");
```

```
if (!empty($instructions))
  print ("<p>Your delivery instructions: <strong>".
    $instructions."</strong></p>");

print ("<h2><a href=\"pizza-order.php\">Order another
pizza?</a></h2>");
```

Run the program a few times, using different inputs, while you examine the logic in this code.

HTML Form Reset Buttons

Web forms can also include **reset** buttons, which can be provided in addition to a **submit** button. A reset button is useful in the case of longer forms, or forms with extensive text entry: if the user clicks the reset button the form is reset to its original state, and any previous user input is cleared. A reset button is coded just like a submit button except the type is changed to "reset", for example:

```
<input type="reset" value="Clear the form">
```

Validating and Sanitizing Form Input

As you can see, a number of HTML form elements allow the user to type text directly from the keyboard (for example, text input boxes, text areas, and password input boxes). This input data should **always** be thoroughly **validated** and **sanitized** before it is "cleared for use" by your application. "Validation" refers to procedures that **test** procedures that ensure that the submitted data corresponds to a required format (for example numeric values, dates, zip codes, email addresses). "Sanitizing" refers to procedures that **remove** content that might contain malicious code (for example certain characters, such as '<' and '>', that might indicate the presence of a script).

The pizza order example does not include any validation of the user name, choice of crust, or delivery instructions in order to keep the focus on the handling the various form elements. Before you move your applications to a "live" online server, where they will be vulnerable to misuse, it is critical that you make them secure. **Appendix E** covers this topic in more detail, including explicit steps to accomplish this.

HTML List Elements

Web pages often need to display **lists** of different kinds. HTML distinguishes between **three** types of list: **ordered** lists, **bulleted** (unordered) lists, and **definition** lists. Let's consider these in turn.

Ordered Lists

Ordered lists use **numbers** or **letters** to display an ordered sequence of list items. An entire ordered list is enclosed inside <**ol**> and </**ol**> tags. Nested inside these tags, each **list item** is enclosed inside <**li**> and </**li**> tags. The **type** attribute can be used in the <**ol**> tag to specify a preferred ordering system. For example **type** = "**1**" indicates a **numerically** ordered list (1, 2, 3, ..), whereas **type** = "**a**" or **type** = "**A**" indicates a list ordered by **letters** (a, b, c, ..) or (A, B, C, ..), and **type** = "**i**" indicates a list ordered by **Roman numerals** (i, ii, iii, ..). If no type attribute is included the list will be ordered numerically (1, 2, 3, ..). Here is an example using Roman numerals to explain why programming is a good career choice:

```
<ol type = "i">
<li>Coding is fun</li>
<li>There are many different careers</li>
<li>Employment outlook is good</li>
</ol>
```

Bulleted Lists

Bulleted lists are lists that use **bullets** to identify each list item. An entire bulleted list is enclosed inside <**ul**> and </**ul**> (unordered list) tags. Nested inside these tags, each **list item** is enclosed inside <**li**> and </**li**> tags. The **type** attribute can be used to specify the type of bullet. For example "**disc**" indicates the use of filled circle bullets, "**square**" indicates square bullets, and "**circle**" indicates unfilled circles. If no type attribute is included the list will be use discs. For example, here is a list using square bullets:

```
<ul type = "square">
<li>Coding is fun</li>
<li>There are many different careers</li>
<li>Employment outlook is good</li>
</ul>
```

Definition Lists

Definition lists contain terms followed by indented definitions of these terms. An entire definition list is enclosed inside <**dl**> and </**dl**> tags. Nested inside these tags, each definition term is enclosed enclosed inside <**dt**> and </**dt**> tags, and each data definition is enclosed enclosed inside <**dd**> and </**dd**> tags. For example:

```
<dl>
  <dt>VCS</dt>
  <dd>Stands for Version Control System.</dd>
  <dt>git</dt>
  <dd>A freely available open source version control
  software,available for Windows, macOS, and Linux.</dd>
  <dt>GitHub</dt>
  <dd>A cloud-based repository, founded on git, that facilitates
  distribution, collaboration, and project development.</dd>
</dl>
```

Terms and definitions can be mixed as needed, for example a single term might be followed by multiple definitions. Note that you will need to style the **dt** selector in CSS if you want the terms to appear differently from the definitions (for example you may want your terms to display as bold).

The **lists.php** file, in the **more-html** folder, generates and displays examples of ordered, unordered, and definition lists, and **Figure 17-2** provides a screen shot.

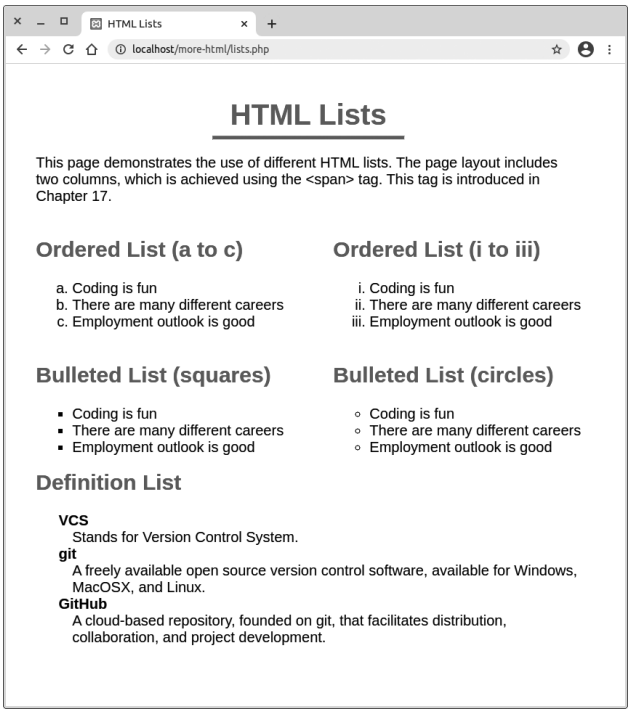

Figure 17-2: lists.php screenshot

Horizontal Rules

The horizontal rule (<**hr**>) element simply draws a line across the page, and is used to visually separate content on a page. A horizontal rule can be defined to cross the entire page, or just a percentage of the page width, or a specified distance. Other settings include height (thickness) and color. The screenshot in **Figure 17-1** shows how **pizza-order.php** uses green horizontal rules to separate different components of the order form. **Figure 17-2** shows a rule below the page heading in **lists.php** that is centered across **35%** of the page, **3 pixels** high, with a color and background color of **red** (note that the color property will apply to the **border** of a rule and the background-color will apply to the area **within** the border). Here is the CSS that was used in lists.php:

```
hr {   margin:auto; width:35%; height:3px;
  color:red; background-color:red }
```

Background Images for Web Pages

Use your style sheet ir you want to apply a background image to your Web page: add a **background** property to the body selector, with a **url** value that specifies the address of the image. By default the image will **repeat**. If you don't want it to repeat, add a **background-repeat** property with the value **no-repeat**. For example here is the CSS to apply a **repeating** image stored in a file named "**test.jpg**" to your entire page:

```
body { background:url(test.jpg);}
```

To apply the same image as a **non-repeating** image:

```
body {background:url(test.jpg); background-repeat:no-repeat;}
```

Creating a Clickable Image

You already know how to create a text-based link to a Web page, for example:

```
<a href="example.html">Click here for an example</a>
```

In this case the page displays a link text ("Click here for an example") that the user can click to open the **example.html** page. Instead of link text, you can provide an **image** that the user can click to go to the linked page: simply locate an image element **inside** your anchor <a> element instead of text:

```
<a href="example.html"><img src="SomeImage.jpg"></a>
```

Block Quotes and Citations

HTML provides a <**blockquote**> element to use when you need to indicate a quote from another source, and also a <**cite**> element to identify a title of a creative work of some kind. These can be used independently of one another, but here is a simple example using them together:

```
<blockquote>"It was the best of times, it was the worst of
times, it was the age of wisdom, it was the age of foolishness,
it was the epoch of belief, it was the epoch of incredulity, it
was the season of Light, it was the season of Darkness, it was
the spring of hope, it was the winter of despair."</blockquote>
<cite>A Tale of Two Cities</cite>
```

DIV and SPAN Containers

The HTML <**div**> and <**span**> tags are used as "containers", to mark the beginning and end of sections of your Web page content, in order to apply specific styles to different sections. The <**div**> tag is a **block-level** container, meaning that the content within a section will begin on a new line and extend across the entire page. The <**span**> tag is an **inline** container, meaning that the content within this section will **continue** across the page from the previous content without beginning a new line (unless the previous content ended with a closing block element, such as a or), and content **after** this section will continue across the page without starting on a new line (unless the closing is followed by an opening block element, such as a or).

These tags are used extensively to layout and design Web pages, allowing custom CSS to be applied to different areas of the page. The three code examples in the **more_html** folder all provide simple applications of these tags.

Both pages of the pizza order application (**pizza-order.php** and **confirm-order.php** use <**div**> tags to provide a header and footer for each page. Here is the HTML code for the header section:

```
<div style ="height:80px; background:green;">
<h1>PHP Express Pizza Delivery</h1>
</div>
```

and here is the HTML code for the footer section:

```
<div style ="height:80px; background:red;">
<h1>123 Main Street 000-111-2222</h1>
</div>
```

In both cases, inline styles are applied to create a container that is **80px** high, with a different background color as the rest of the page (the header has a **green** background and the footer has a **red** background, to replicate the colors of the Italian flag). Each section only contains a single <**h1**> element (the color for the **h1** selector was specified to be **white** in **pizza.css**).

Open **pizza-order.php** in your Web browser (to see the page in color), or review the screenshot in **Figure 17-1, earlier in this chapter**.

The **lists.php** page demonstrates the use of different lists. This page uses four <**span**> containers to locate the four list examples. The first and third of these spans are identified with the class name "**leftCol**":

```
<span class="leftCol">
```

and the second and fourth spans are identified with class name "**rightCol**":

```
<span class="leftCol">
```

The only CSS property applied to these two classes is **float**, which is assigned the value **left** in the **leftCol** class and **right** in the **rightCol** class:

```
span.leftCol {float:left; }
span.rightCol {float:right;}
```

Each span element contains a heading and an ordered or unordered (bulleted) list. The first and third containers are floated to the left of the page, while the second and fourth are floated to the right, in effect creating two columns and two rows of lists. Open **lists.php** in your Web browser (to see the page in color), or review the screenshot in **Figure 17-2**, earlier in this chapter.

The **containers.php** page provides a more complete example, including both <**div**> and <**span**> containers, and applying a number of styles to each container. **Figure 17-3** provides a screen shot of this page but you will probably want to open the page in a Web browser to see the colors.

The page is divided into **three** sections, using class names ("**header**", "**main**", and "**footer**") to identify each <**div**> container. The **header** section contains a heading and a sub-heading; the **main** section contains a heading and a paragraph; and the **footer** section also contains a heading and a paragraph. Within the paragraph in the main section, two <**span**> containers are used to mark text that will receive special styling. The first has the class name "**red**" and the second has the class name "**green**". The internal style sheet specifies the CSS for these five classes.

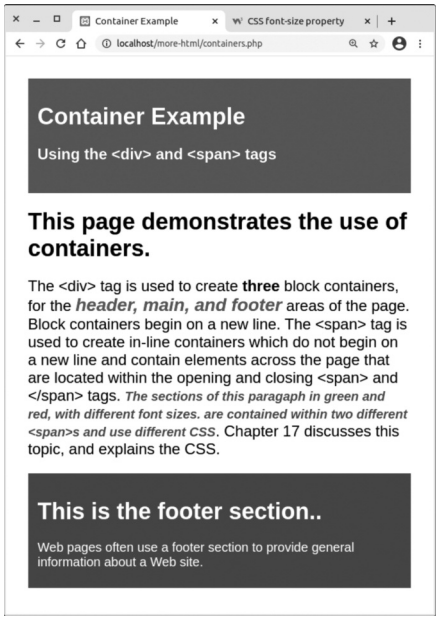

Figure 17-3: containers.php screenshot

Here is the code for **containers.php**:

```
<html>
<head>
<title>Container Example</title>
<style>
  body {width:400px;padding:15px;
    font-family: Arial,Helvetica,Sans-serif;}
  h1 {font-size:18pt}
  h2 {font-size:12pt}
  div.heading {height:100px; padding:10px;
    background: red; color:white;}
    div.main {background:white; color:black;font-size:12pt}
  div.footer {height:100px; padding:10px; background:green;
    font-size:10pt; color:white }
  span.green {font-weight:bold; color:green;
    font-style:italic; font-size:10pt}
  span.red {font-weight:bold; color:red;
    font-style:italic; font-size:14pt}
</style>
</head>

<body>
```

```
<div class="heading">
<h1>Container Example</h1>
<h2>Using the &lt;div&gt; and &lt;span&gt; tags
</div>

<div class="main">
<h1>This page demonstrates the use of containers.</h1>
<p>The &lt;div&gt; tag is used to create <strong>three</strong>
block containers, for the <span class="red">header, main, and
footer</span> areas of the page. Block containers begin on a new
line. The &lt;span&gt; tag is used to create in-line containers
which do not begin
on a new line and contain elements across the page that are lo-
cated within the opening and closing &lt;span&gt; and
&lt;/span&gt; tags. <span class="green">The sections of this
paragraph in green and red, with different font sizes. are con-
tained within two different &lt;span&gt;s and use different
CSS</span>. Chapter 17 discusses this topic, and explains the
CSS.</p>

<div class="footer">
<h1 style="color:white">This is the footer section..</h1>
<p>Web pages often use a footer section to provide general in-
formation about a Web site.</p>
</div>
</body>
</html>
```

Refer to the internal CSS in the <head> section to see what styles are applied to the content of each container. This example demonstrates the difference between **block-level** <**div**> elements that begin and end on a new line and extend across the page, and **inline** <**span**> elements, that are rendered within the current line.

HTML Entities

You may have noticed some unusual code in **containers.php**, for example

```
<h2>Using the &lt;div&gt; and &lt;span&gt; tags </h2>
```

and

```
<p>The &lt;div&gt; tag is used ..
```

Since the Web browser treats the '<' (less than) and '>' (greater than) characters as indicators of HTML tags, how do you include these characters in the actual text of your Web page?

HTML includes **entities**: special characters that can be specified in text using a special code to reference each character, surrounded by the **&** (ampersand) and **;** (semicolon) characters. For example the '<' (**less than**) character can be included in your text using **&lt**; and the '>' (**greater than**) character can be specified using **>**. In containers.php, the text **<div&gt**; will display as "<**div**>", and **<span>** will display as "<**span**>".

A large number of entities are available, including special symbols such as the copyright and trademark symbols, some currency symbols, mathematical symbols, Greek characters, and much more. Here are just a few examples: **©** (copyright), **®** (registered trademark), **¥** and **€** (Japanese Yen and Euro currency symbols), **°** (degree symbol), **Δ** and **δ** (upper and lower case versions of the Greek delta character).

Another HTML entity, the non-breaking space character (** **), can be very useful when formatting text. Use this character to tell the browser not to break two words across a line (for example, with normal spaces, "**20 mph**" might end up with "**20**" at the end of one line and "**mph**" at the start of the next, whereas "**20 mph**" will ensure that the entire text "**20 mph**" will be moved to the start of the next line). This entity can also be used to ensure that the browser will correctly render multiple adjacent spaces in a text string, since usually the browser will collapse multiple spaces into a single space. For example, the text string "**Testing   1, 2, 3**" will display five spaces after the word "Testing".

For a comprehensive list Google "HTML entities", or see:

```
https://dev.w3.org/html5/html-author/charref
```

Deprecated HTML Tags

The specifications for HTML have evolved steadily. Many tags and attributes that were developed early on have since been **deprecated**, which means they have been replaced by more efficient solutions. As you explore HTML you will often find references to tags that have been deprecated. Although the most widely used browsers all continue to recognize deprecated tags, at some point in the future, deprecated tags may no longer be recognized, so you should avoid using these tags. Examples of deprecated tags are <**u**> (underline), <**center**>, and <**font**>. Examples of deprecated attributes are **align**, **bgcolor**, and **width**.

Useful References

For complete information about HTML and CSS you will want to refer to the World Wide Web Consortium's (WC3's) current standards:

https://www.w3.org/standards/
https://www.w3.org/Style/CSS/

Here are two **excellent** online resources to teach you more about HTML and CSS, including tutorials and best practices:

https://developer.mozilla.org/en-US/docs/Learn
https://www.w3schools.com/

For a good reference to color names that are recognized by most browsers, go to:

https://www.w3schools.com/colors/colors_names.asp

To learn more about using **Web fonts**, delivered from a Web server, as an alternative to desktop fonts:

https://developer.mozilla.org/en-US/docs/Learn/CSS/Styling_text/Web_fonts