MODULE 2

Chapter 1: HTML Tables and Forms

1. Introducing Tables
2. Styling Tables
3. Introducing Forms
4. Form Control Elements
5. Table and form Accessibility
6. Microformats

1.1 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.
- HTML table can contain any type of data: like numbers, text, images, forms, even other tables.

Basic Table Structure

- HTML <table> contains any number of rows(<tr>); each row contains any number of table data cells (<td>) as shown in Figure 1.1.
- Some browsers do not display borders for the table by default; however, we can do so via CSS. Many tables will contain some type of headings in the first row.
- In HTML, we indicate header data by using the <th> instead of the <td> element as shown in Figure 1.2, because browsers tend to make the content within a <th> element bold, but this can also be done by CSS.
- The main reason we shouldn't use the <th> element is, due to presentation reasons. Rather, we should also use the <th> element for accessibility reasons and for search engine optimization reasons.

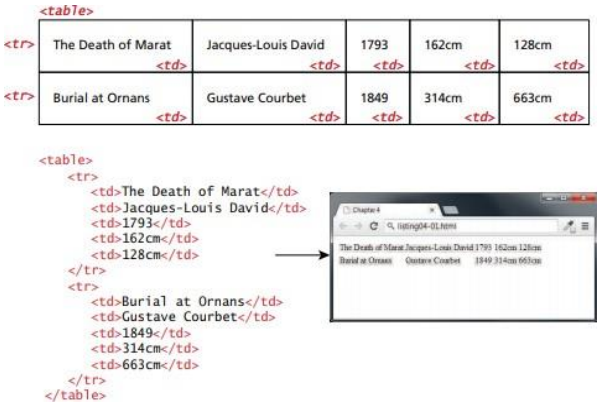


Figure 1.1: Basic Table Structure

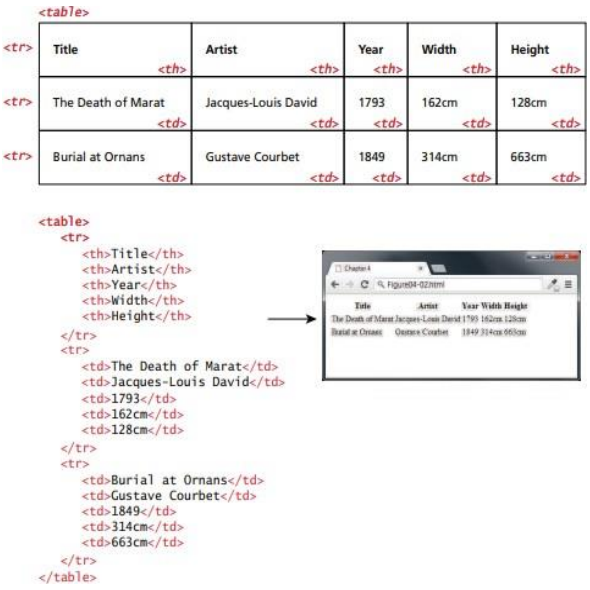


Figure 1.2: Adding table heading

Spanning Rows and Columns

- Two key features about tables are:
  1. The first is that all content must appear within the <td> or <th> container.
  2. 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/feature.
  - i.e., If we want a given cell to cover several columns or rows, then we can do so by using the **colspan** (Figure 1.3) or **rowspan** attributes (Figure 1.4).

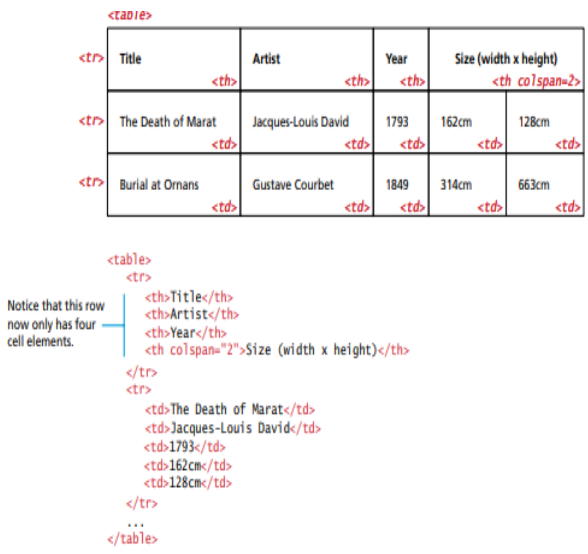


Figure 1.3: Spanning Columns

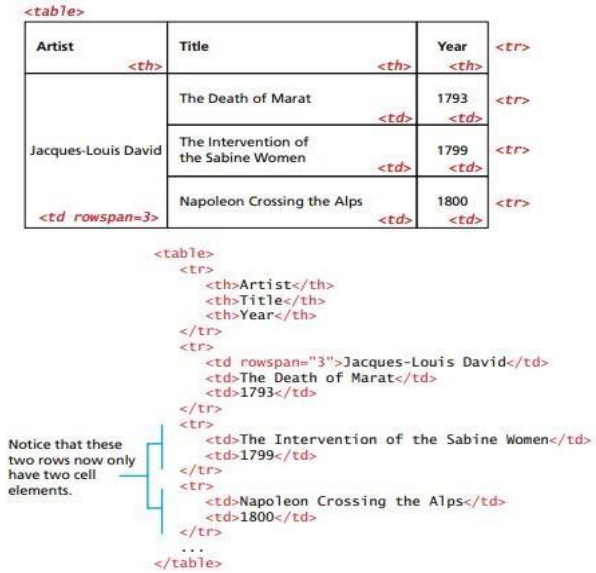


Figure 1.4: Spanning rows

Additional 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.
- We 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 but make s
- ome sense for tables with a large number of rows.
- With CSS, one could set the height and overflow properties of the <tbody>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, we can style all columns within a <colgroup> with just a single style.
- The only properties we 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.

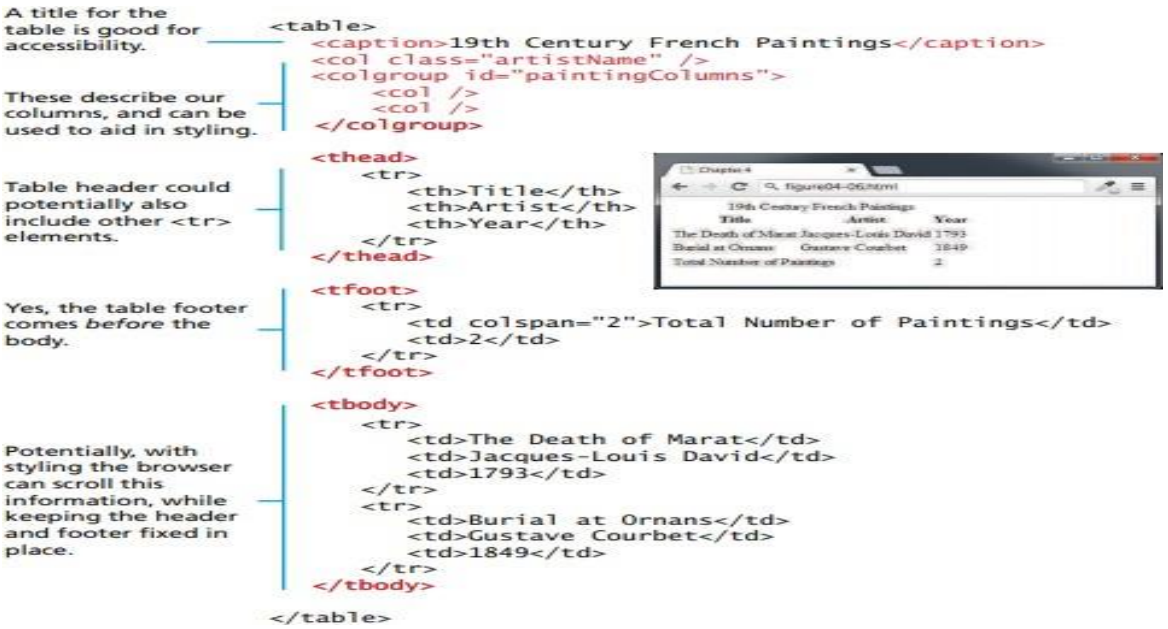


Figure 1.5: Additional table elements

Using Tables for Layout

- 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 as shown in

Figure 1.6.

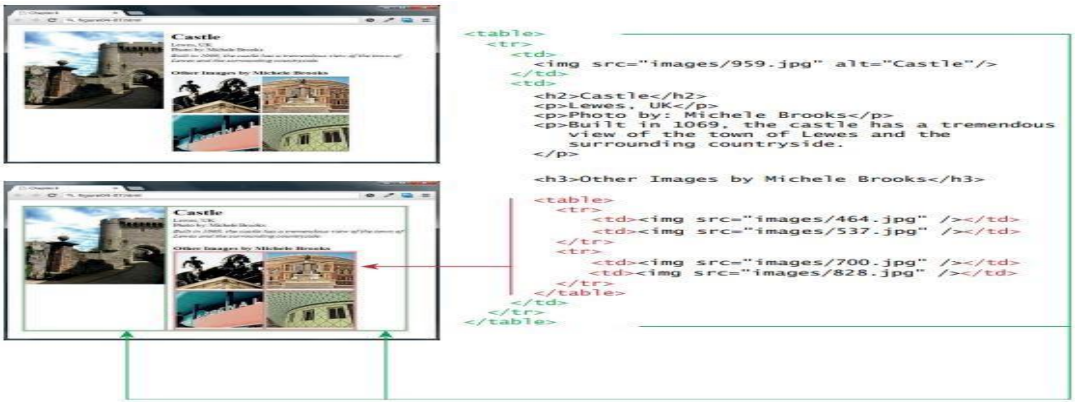


Figure 1.6: Example of using tables for layout

Problems:

1. It dramatically increase the size of the HTML document, so takes longer time to download and maintainability is difficult as it has many table elements.
2. The resulting markup is not semantic because tables are meant to indicate the tabular data but if we use table elements to align the block-elements side by sidemeans we are giving presentation rather than semantic.
3. Using tables for layout results in a page that is not accessible, meaning that forusers who rely on software to voice the content, table-based layouts can be extremely uncomfortable and confusing to understand. It is much better to useCSS for layout.

1.2 Styling Tables

- There is certainly no limit to the way one can style a table.

Table Borders

- Borders can be assigned to the <table>, <th> and the <td> element. Interestingly, borders cannot be assigned to the <tr>, <thead>, <tfoot>, and<tbody> elements.
- This property selects the table's border model. The default, shown in the second screen capture in Figure 1.7.
- In this approach, each cell has its own unique borders. We can adjust the space between these adjacent borders via the border-spacing property, as shown in the final screen capture in Figure 1.7.
- In the third screen capture, the collapsed border model is being used; in this model adjacent cells share a single border.

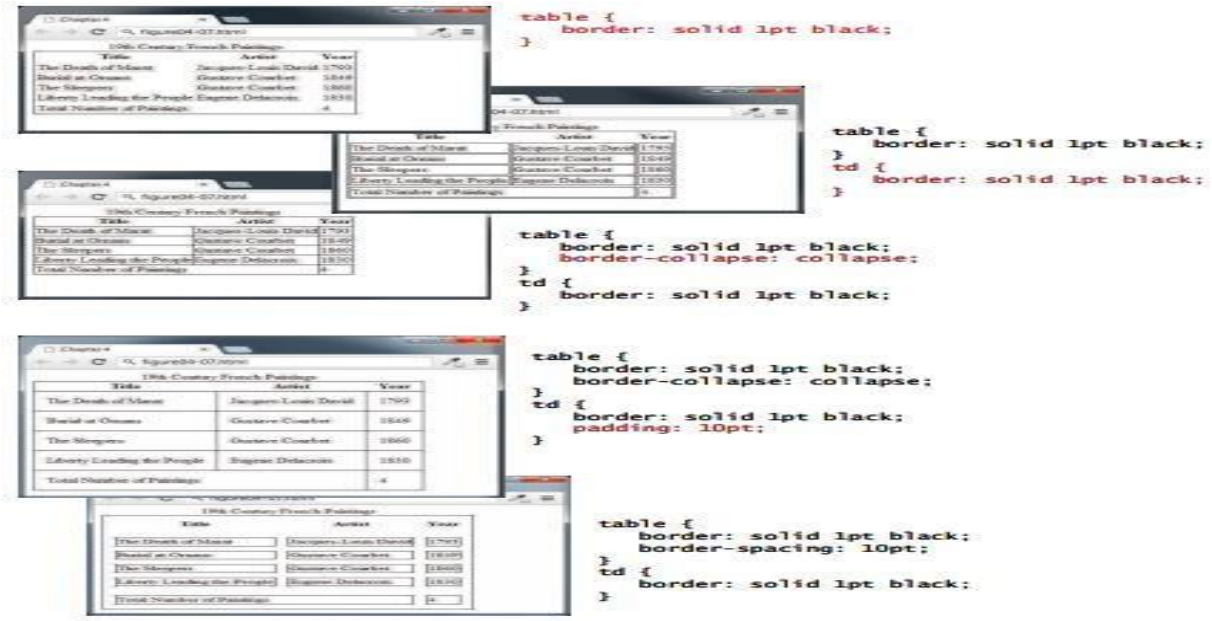


Figure 1.7: Styling table borders

Boxes and Zebras

- There are different ways to style a table.
- The first of these is a box format, in which we simply apply background colors and borders in various ways, as shown in Figure 1.8.
- 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 1.9.

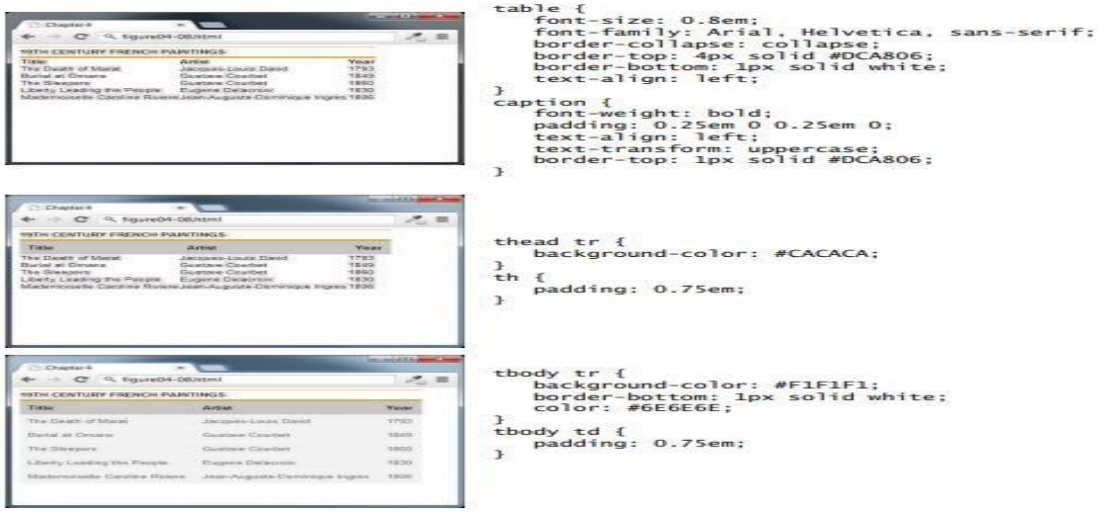


Figure 1.8: An example boxed table

- Figure 1.9 also illustrates how the pseudo-element nth-child can be used to alternate the format of every second row.

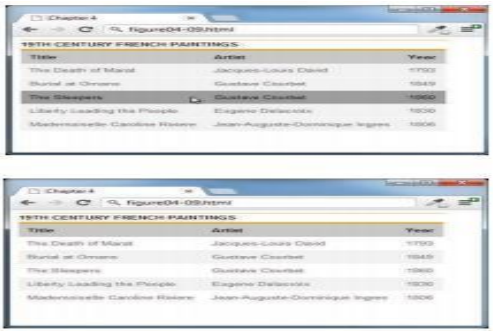


Figure 1.9: Hover effect and zebra stripes

1.3 Introducing Forms

- **Forms** provide an alternative way to interact with a web 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 subsequent HTML based on that input.
- 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.

Form Structure

- A form is constructed in HTML in the same manner as tables or lists: i.e., using special HTML elements (<form>).
- Figure 1.10 illustrates a typical HTML form. The 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.

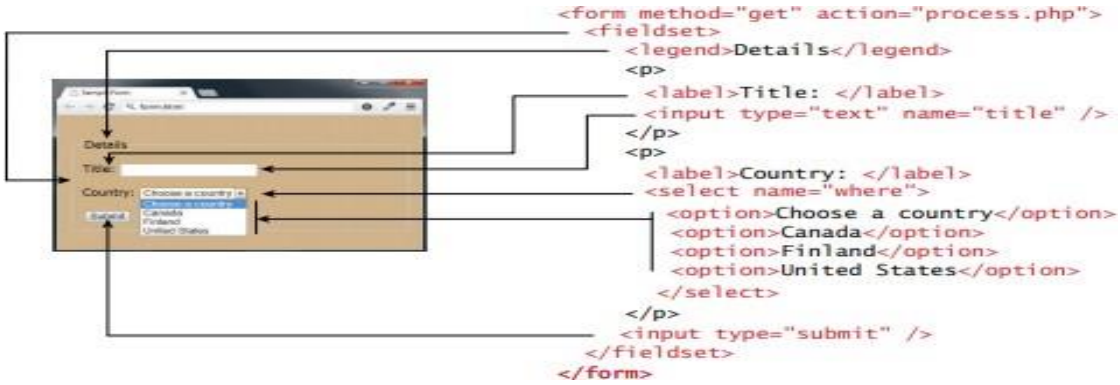


Figure 1.10: Simple HTML form

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 1.11.
- 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 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 HTTPrequest.

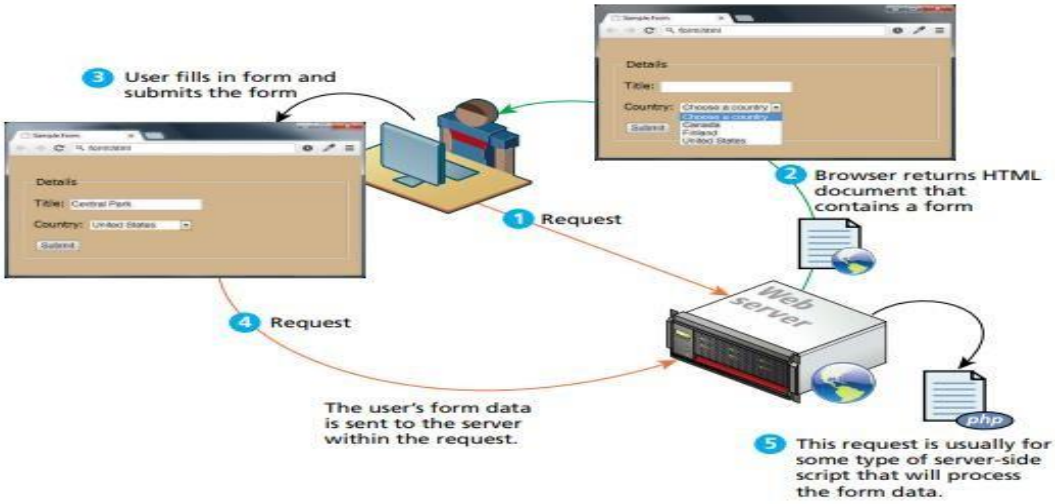


Figure 1.11: how forms work

Query Strings

- A **query string** is a series of name=value pairs separated by ampersands (the **&**character).
- 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 1.12 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 a query string.

- Instead, such special symbols must be **URL encoded** (also called **percent encoded**), as shown in Figure 1.13.

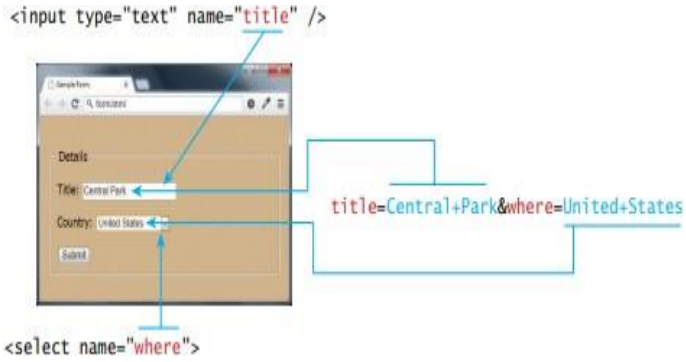


Figure 1.12: Query string data and its connection

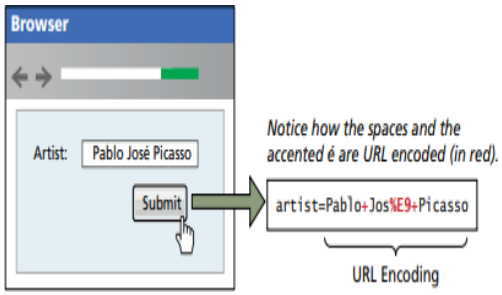


Figure 1.13: URL Encoding

The <form> Element

- 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.
- 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**).
- 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. The difference GET and POST resides in where the browser locates the user's form input in the subsequent HTTP request.
- With **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 1.14 illustrate show the two methods differ. Table 1.1 lists the key advantages and disadvantages of each method. Generally, form data is sent using the POST method.
- The GET method is useful when we 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. And needless to say, any time passwords are being transmitted, they should be transmitted via the POST method.

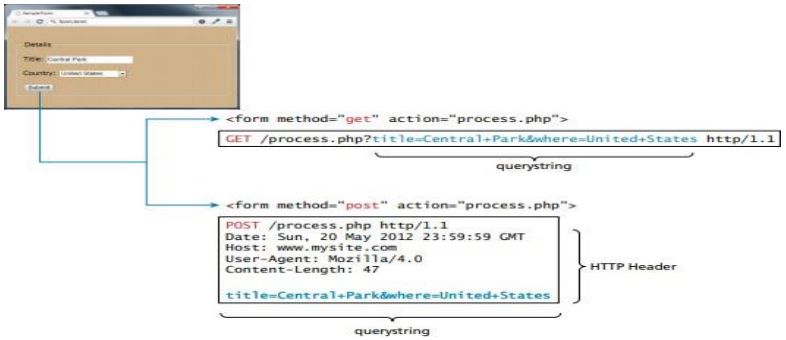


Figure 1.14: GET versus POST

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

Table 1.1: GET versus POST

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 1.2: Form-Related HTML Elements

1.4 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 1.2.

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.

Type	Description
text	Creates a single-line text entry box. <input type="text" name="title" />
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). <textarea rows="3" ... />
password	Creates a single-line text entry box for a password (which masks the user entry as bullets or some other character) <input type="password" ... />
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. <input type="search" ... />
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. <input type="email" ... />
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. <input type="tel" ... />
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. <input type="url" ... />

Table 1.3: Text Input Controls

- Figure 1.15 illustrates the various text element controls and some examples of how they look in selected browsers.

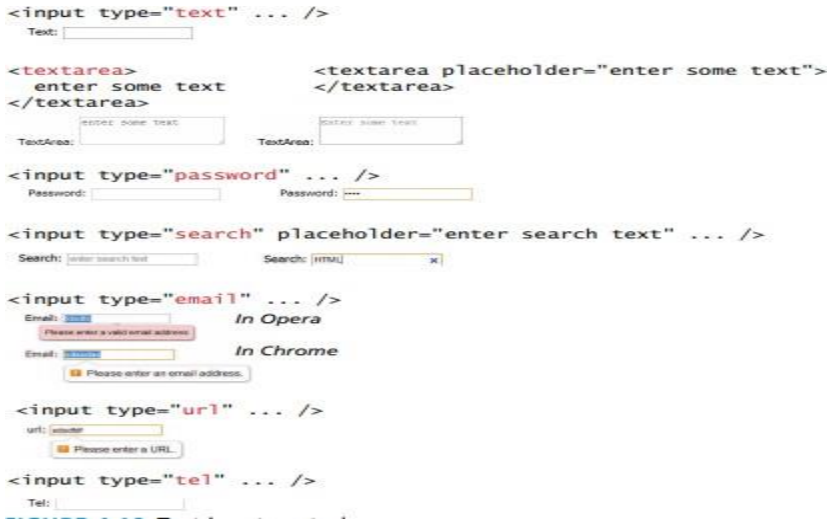


Figure 1.15: Text input controls

Figure 1.16: Using the pattern attribute

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 can be hidden in a dropdown 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 1.17.

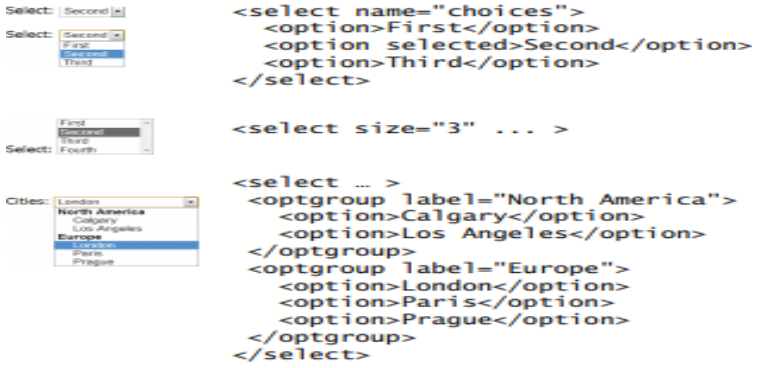


Figure 1.17: Using the <select> element

- 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 1.18.

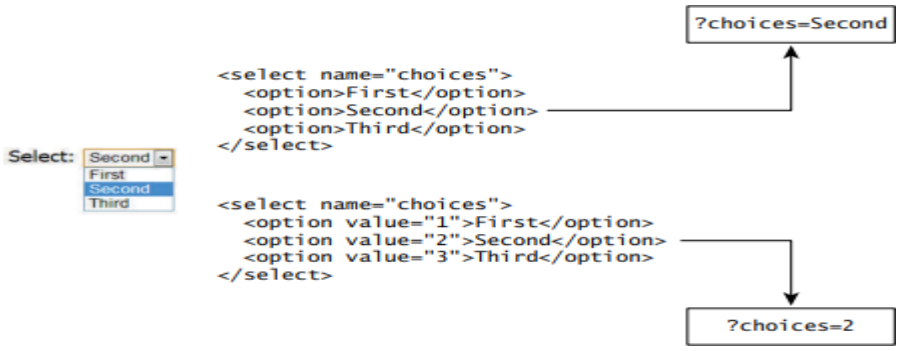


Figure 1.18: The value attribute

Radio Buttons

- **Radio buttons** are useful when we want the user to select a single item from a small list of choices and we want all the choices to be visible.
- As can be seen in Figure 1.19,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.

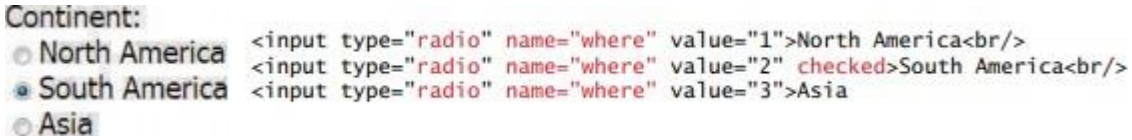


Figure 1.19: Radio buttons

Checkboxes

- **Checkboxes** are used for getting yes/no or on/off responses from the user. As can be seen in Figure 1.20, checkboxes are added via the <input type="checkbox">element.
- We 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.

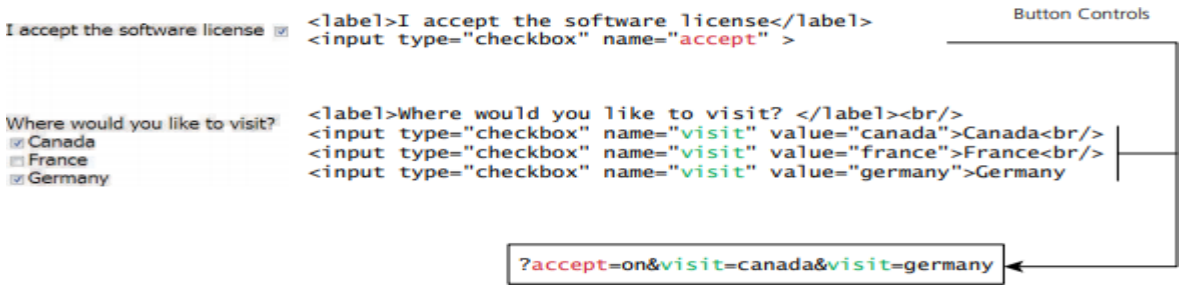


Figure 1.20: Checkbox buttons

Button Controls

- HTML defines several different types of buttons, which are shown in Table 1.4. As can be seen in that table, there is some overlap between two of the button types. Figure 1.21 demonstrates some sample button elements.

Type	Description
`<input type="submit">`	Creates a button that submits the form data to the server.
`<input type="reset">`	Creates a button that clears any of the user's already entered form data.
`<input type="button">`	Creates a custom button. This button may require JavaScript for it to actually perform any action.
`<input type="image">`	Creates a custom submit button that uses an image for its display.
`<button>`	Creates a custom button. The `<button>` element differs from `<input type="button">` 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. You can turn the button into a submit button by using the `type="submit"` attribute.

Table 1.4: Button Elements

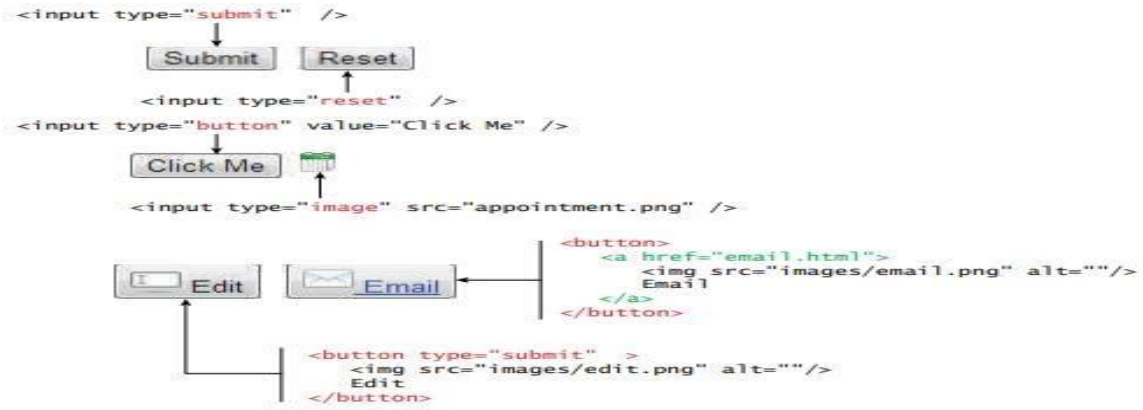


Figure 1.21: Example button elements

Specialized Controls

- There are two important additional special-purpose form controls that are available in all browsers.
- The 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 1.22.
- 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.

Figure 1.22: File upload control (in Chrome)

Number and Range

- 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 1.23 illustrates the usage and appearance of these numeric controls.

Figure 1.23: Number and range input controls

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 1.24.

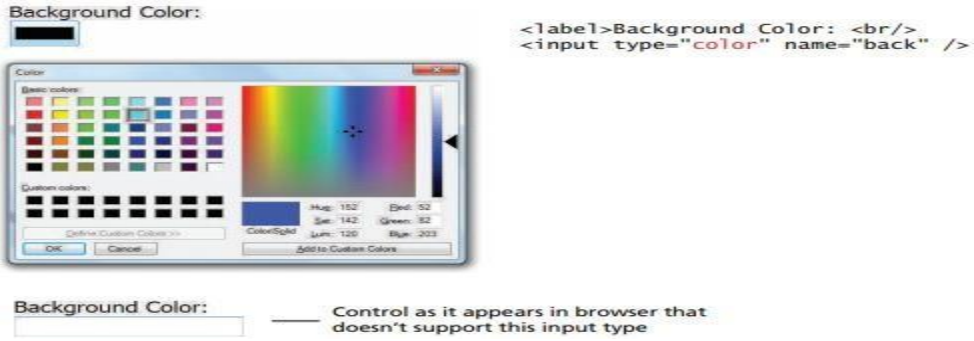


Figure 1.24: Color input control

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.
- Table 1.5 lists the various HTML5 date and time controls. Their usage and appearance in the browser are shown in Figure 1.25.

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 1.5: HTML5 Date and Time Controls

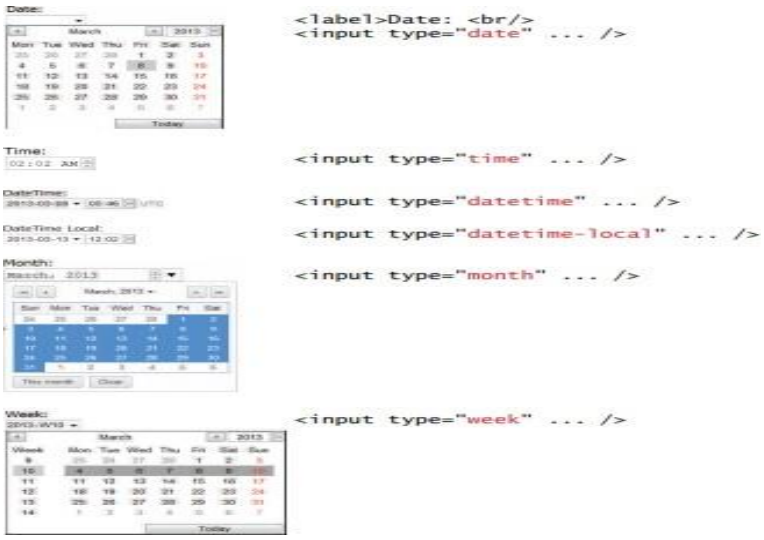


Figure 1.25: Date and time controls

1.5 Table and Form Accessibility

- **Problems:** 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**.
- 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.

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.
    - 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.
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.

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

Listing 1.1: Connecting cells with headers

Accessible Forms

- HTML forms are also potentially problematic from an accessibility standpoint. If we remember the advice from the WAI about providing keyboard alternatives and text alternatives, our forms should be much less of a problem.
- The use of the <fieldset>, <legend>, and<label> elements, which provide a connection between the input elements in the form and their actual meaning.

```
<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 1.26: Associating labels and input elements

- In other words, these controls add semantic content to the form and its logically group related form input elements together with the <legend> providing a type of caption for those elements.
- The <label> element has no special formatting. Each <label> element should be associated with a single input element. i.e., if the user clicks on or taps the <label> text, that control will receive the form's focus.

1.6 Microformats

- 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 1.27 illustrates this process. One of the most common Microformats is **hCard**, which is used to semantically markup contact information for a person. Google Map search results now make use of the hCard microformat.

- Listing 1.2 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**.

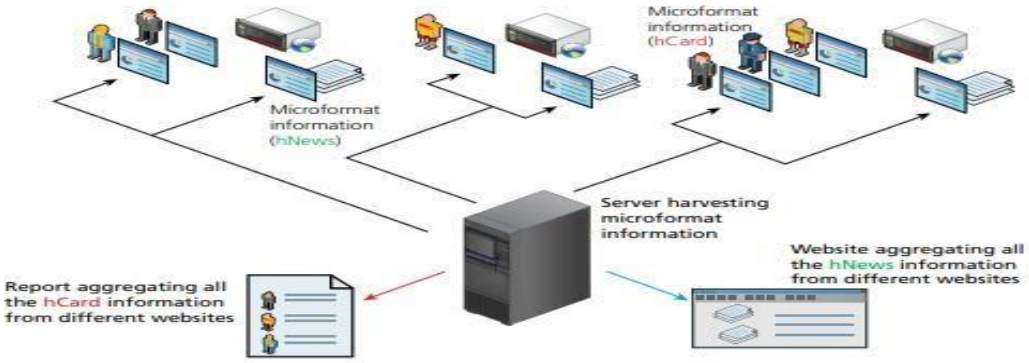


Figure 1.27: Microformats

```
<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 1.2: Example of an hCard

Chapter 2: Advanced CSS-Layout

1. Normal Flow
2. Positioning Elements
3. Floating Elements
4. Constructing Multicolumn Layouts
5. Approaches to CSS Layout
6. Responsive Design
7. CSS Frameworks

2.1 Normal Flow

- **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

- <p>, , <h2>, <ul>, 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.
- Block-level elements use the normal CSS boxmodel, in that they have margins, paddings, background colors, and borders.

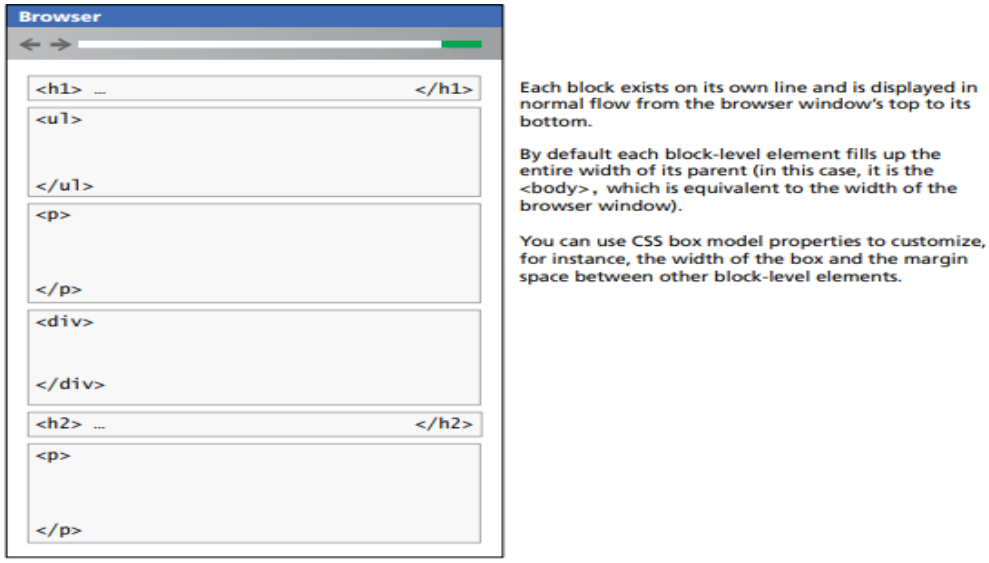


Figure 2.1: Block-level elements

Inline elements

- These 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>, <img>, and .
- Inline elements line up next to one another horizontally 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 2.2.

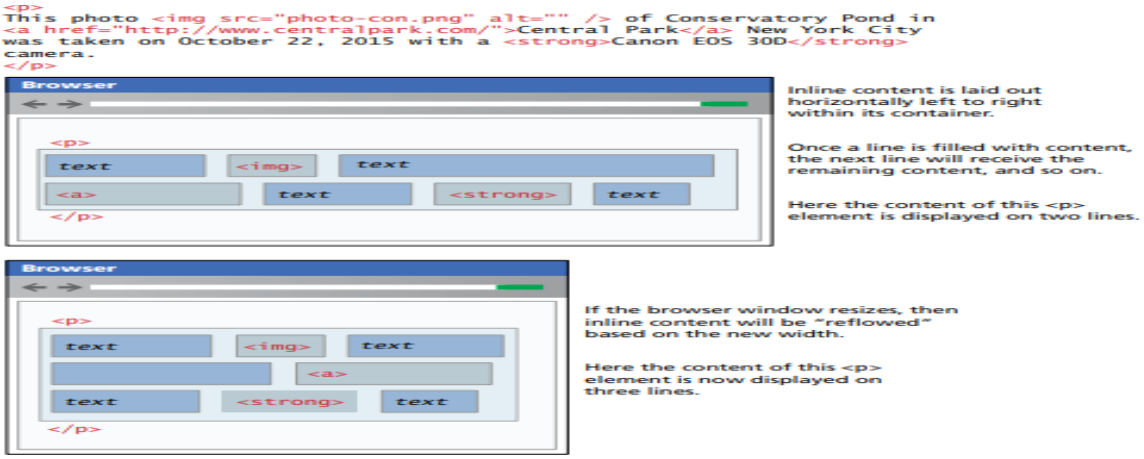


Figure 2.2: Inline elements

- There are actually two types of inline elements: replaced and nonreplaced.

Replaced inline elements

- These are elements whose content is defined by some external resource, such as <img> and the various form elements.
- Replaced inline elements have a width and height that are defined by the external resource

Nonreplaced inline elements

- These are elements whose content is defined within the document, which includes all the other inline elements.
- Width of these elements are 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.

- In a document with normal flow, block-level elements and inline elements work together as shown in Figure 2.3. Block-level elements will flow from top to bottom, while inline elements flow from left to right within a block.
- 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 <li> elements like inline (that is, each list item will be displayed on the same line).

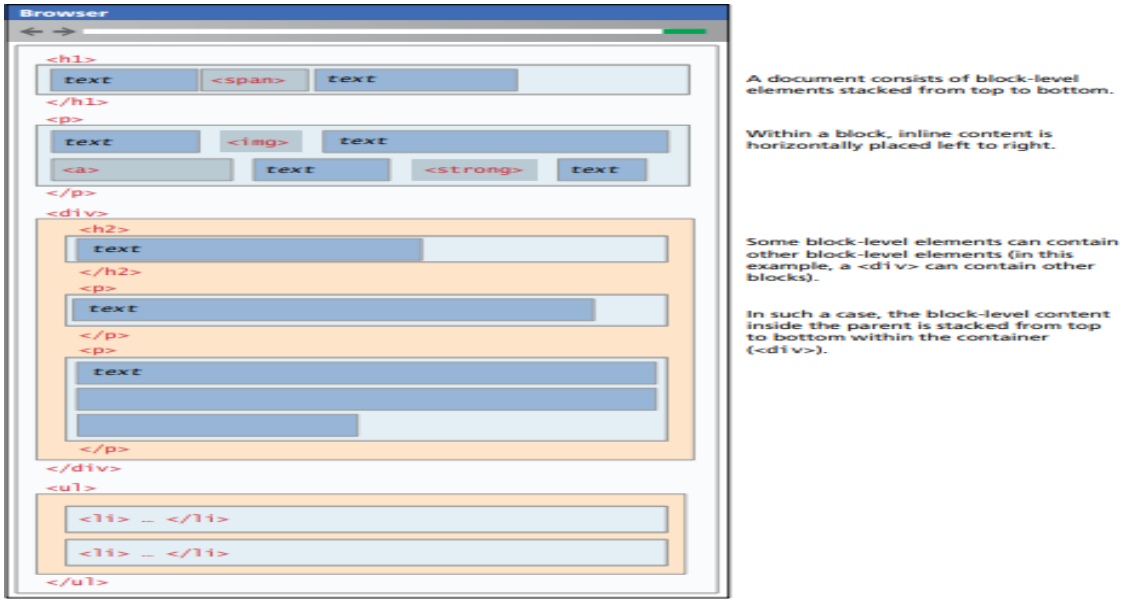


Figure 2.3: Block and inline elements together

2.2 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 2.1.
- 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.

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 2.1: Position Values

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.
- 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 2.4.
- The positioned element now overlaps other content: that is, the <p> element following the <figure> element does not change to accommodate the moved<figure>.

```
<p>A wonderful serenity has taken possession of my …
<figure>
    <img src="images/828.jpg" alt="" />
    <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley …
```

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

Figure 2.4: Relative positioning

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 2.5, 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.

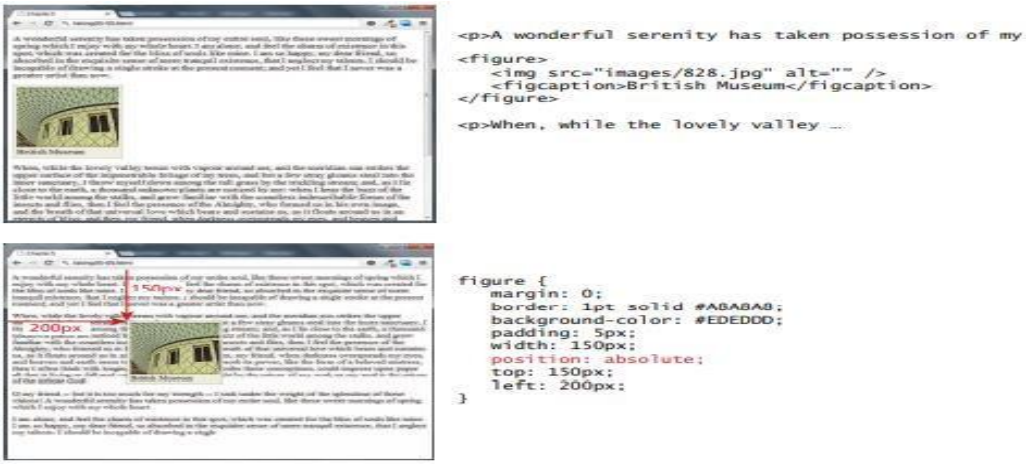


Figure 2.5: Absolute positioning

- 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 2.6, the <figcaption> is absolutely positioned; which happens to be its parent (the <figure> element).

Figure 2.6: Absolute position is relative to nearest positioned ancestor container.

Z-Index

- 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.

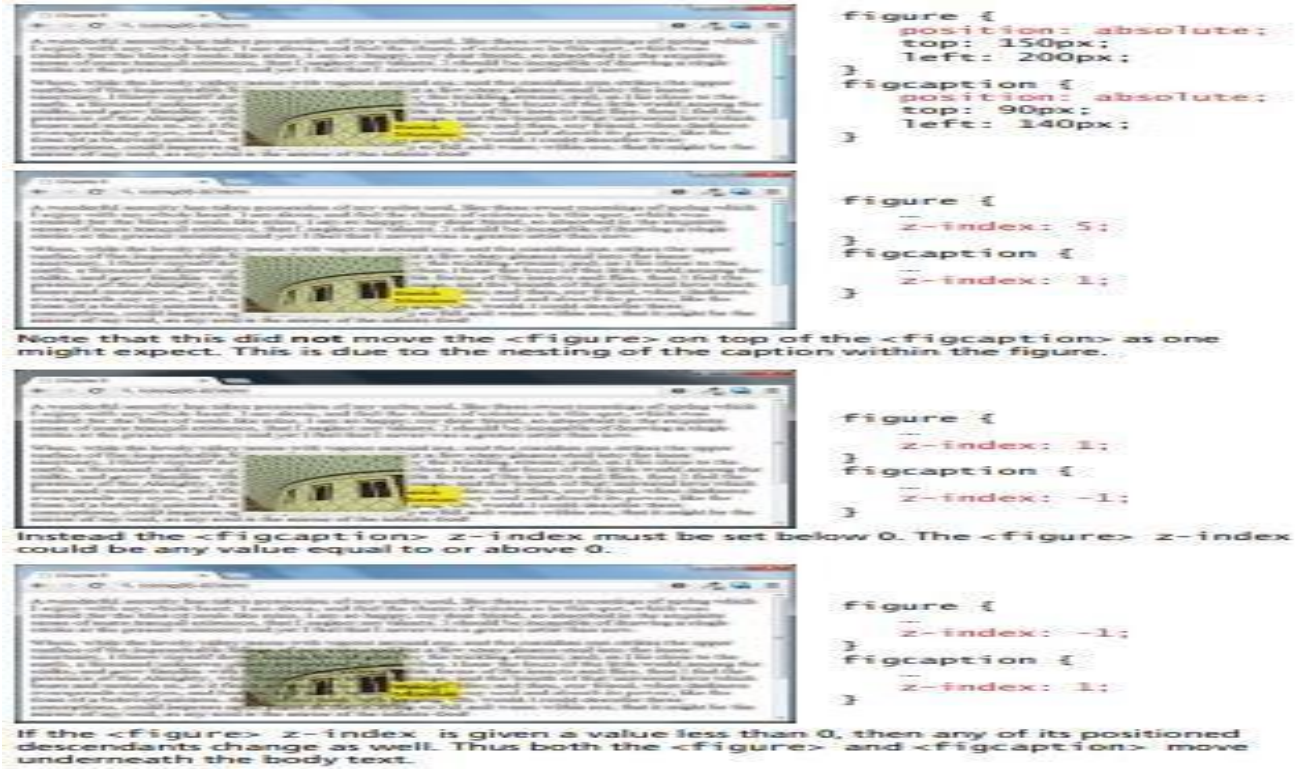


Figure 2.7: Z-index

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

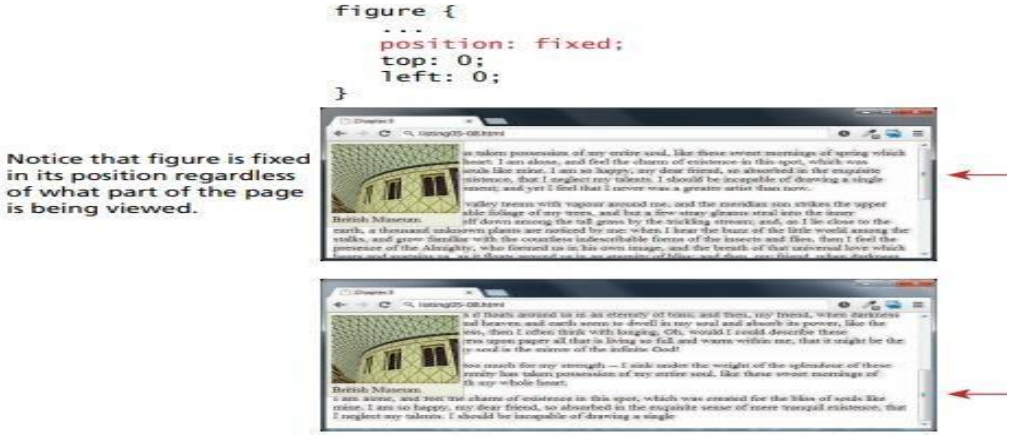


Figure 2.8: Fixed position

2.3 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 2.9.

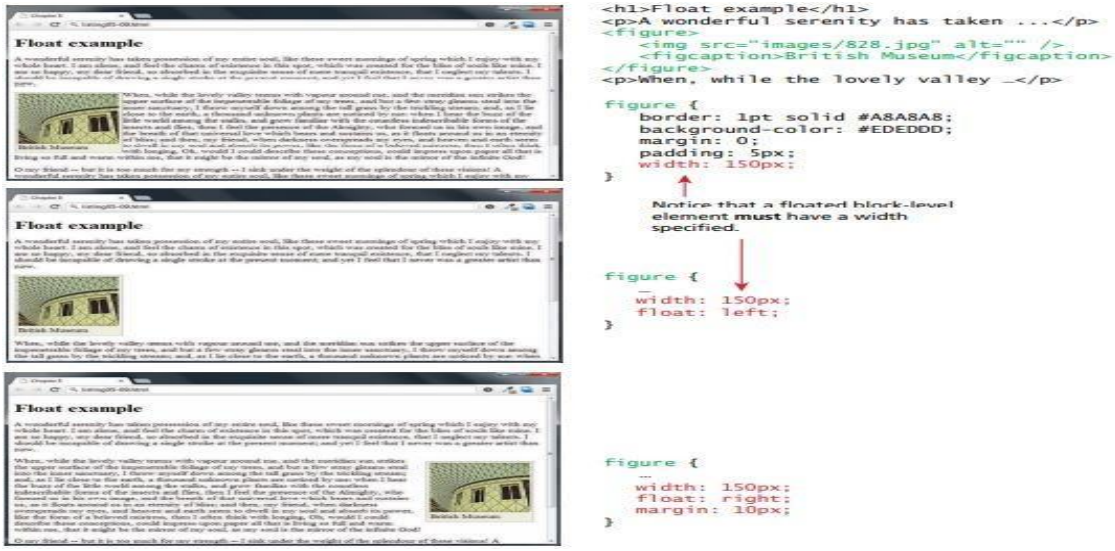


Figure 2.9: Floating an element

- The floated block-level element must have a width specified; if we don't, 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.

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 2.9, the containing block is the HTML document itself so the figure moves to the left or right of the browser window.
- Figure 2.10, 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.
- In this illustration, we can see that the overlapping margins for the adjacent <p> elements behave normally and collapse. But the top margin for the floated<figure> and the bottom margin for the <p> element above it do *not* collapse.

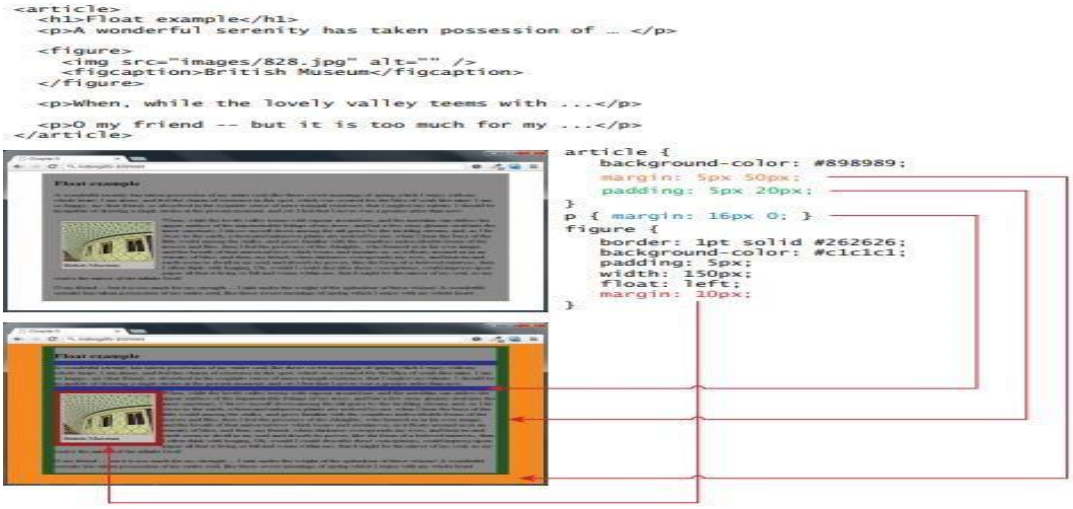


Figure 2.10: Floating to the containing block

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 we 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 2.11.

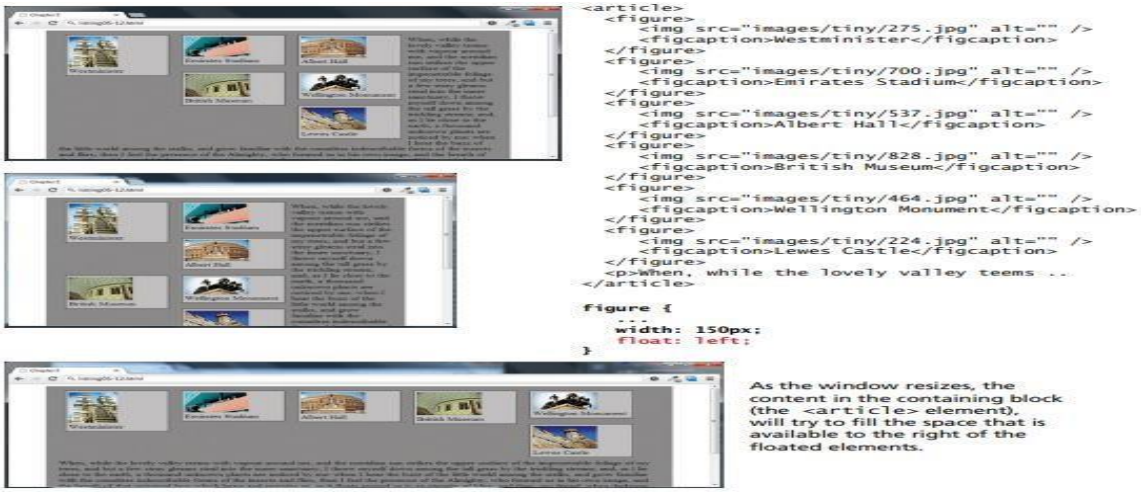


Figure 2.11: Problems with multiple floats

Problems

- As the browser window increases or decreases in size (that is, as the containing block resizes).

Solution:

- We can stop elements from flowing around a floated element by using the clear property.
- The values for this property are shown in Table 2.2. Figure 2.12 demonstrates how the use of the clear property.

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 2.2: clear property

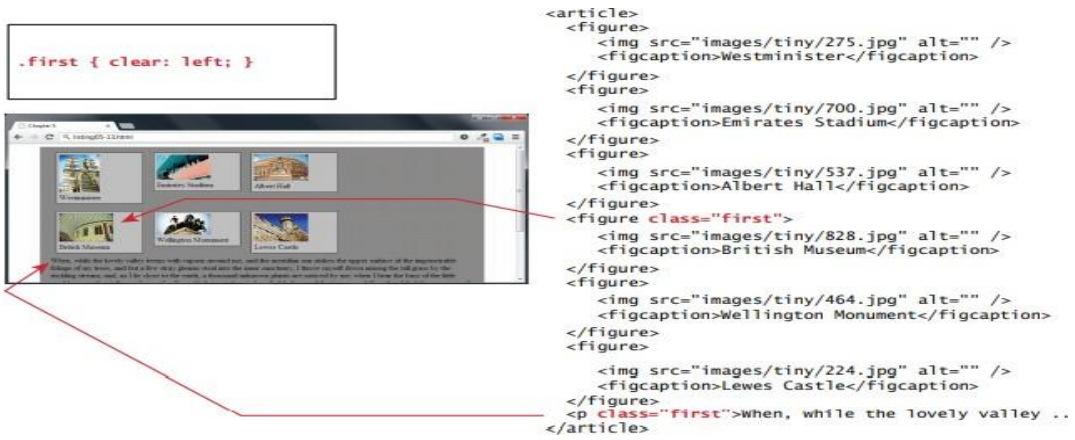


Figure 2.12: Using the clear property

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 2.13.

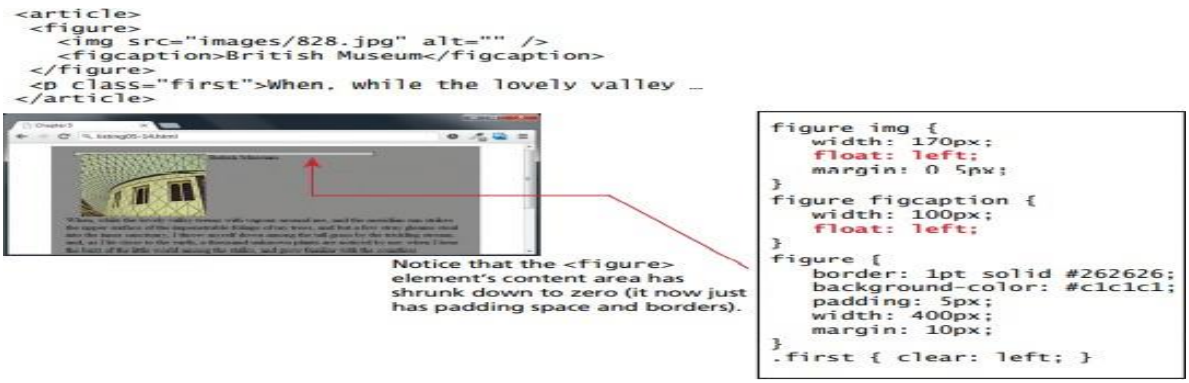


Figure 2.13: Disappearing parent containers

- In Figure 2.13, the <figure> containing block contains only an <img> and <figcaption> element, and both of these elements are floated to the left. i.e., 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. To avoid this a better solution would be to use the **overflow property** as shown in Figure 2.14.

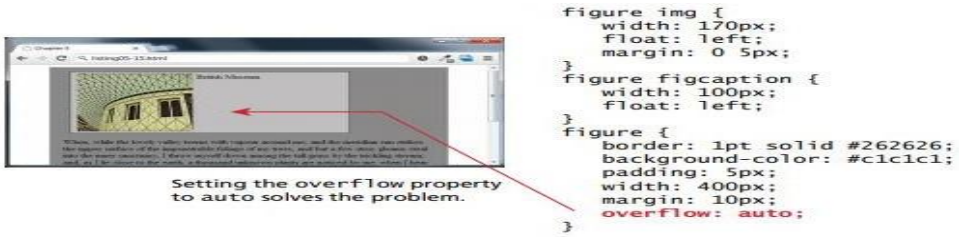


Figure 2.14: Using the overflow property

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 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.

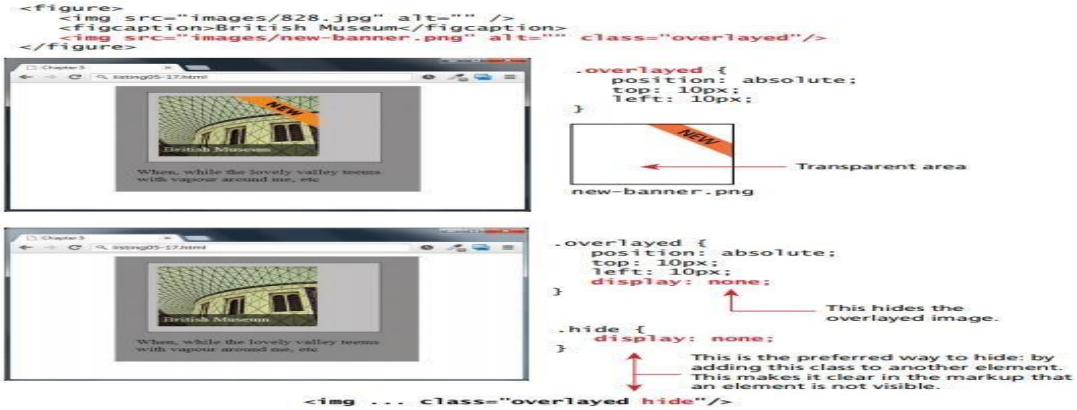


Figure 2.15: Using the display property

- 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, it only covers part of the underlying image.

Figure 2.16: Comparing display to visibility

- 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.
- 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 2.17 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 when the mouse hovers over the thumbnail version of the image.

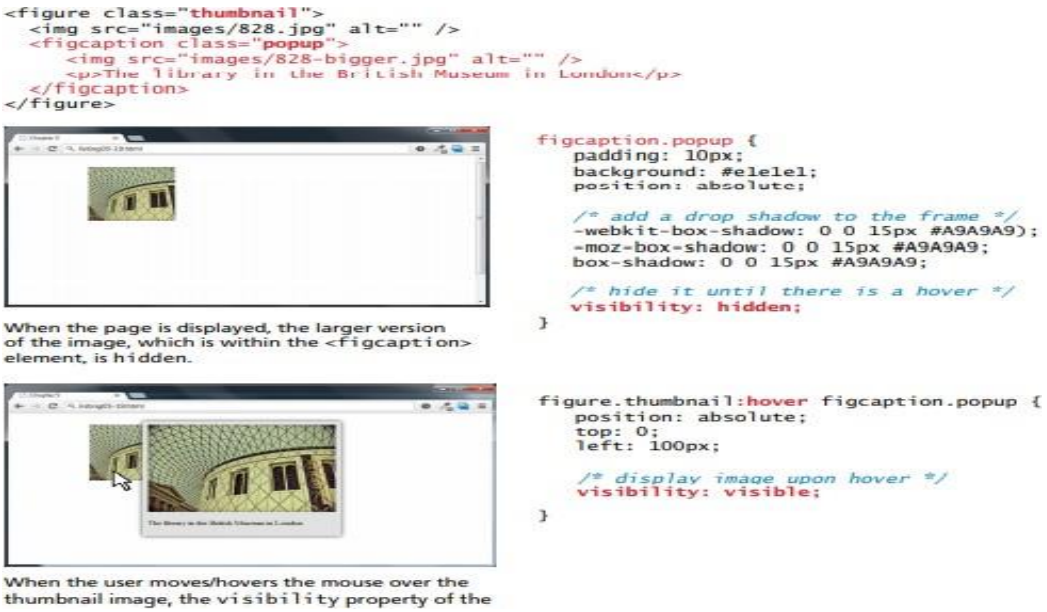


Figure 2.17: Using hover with display

2.4 Constructing Multicolumn Layouts

They are the raw techniques that we can use to create more complex layouts. A typical layout may very well use both positioning and floats.

Using Floats to Create Columns

- Using floats is perhaps the most common way to create columns of content.
- 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 2.18, the other content will flow around the floated element.
- Figure 2.18 shows the other key step: changing the left-margin so that it no longer flows back under the floated content.

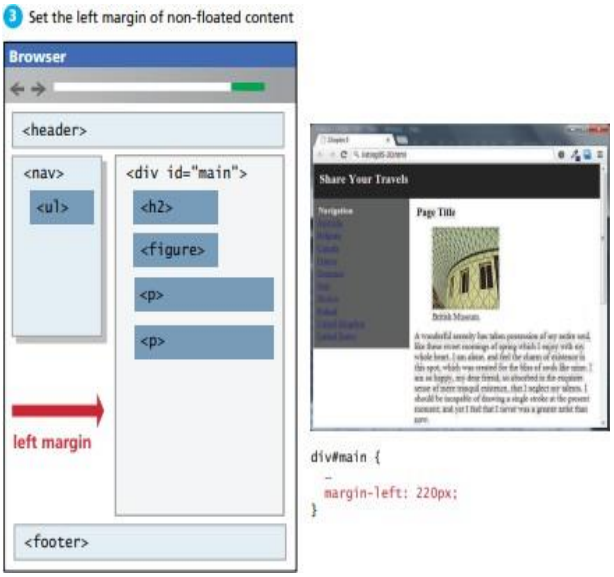


Figure 2.18: Creating two-column layout

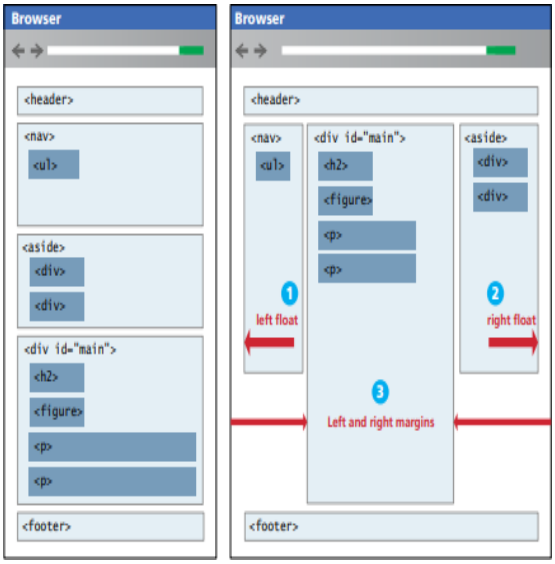


Figure 2.19: Creating a three-column layout

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.

- A three-column layout could be created in much the same manner, as shown in Figure 2.19 page layouts.
- 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.

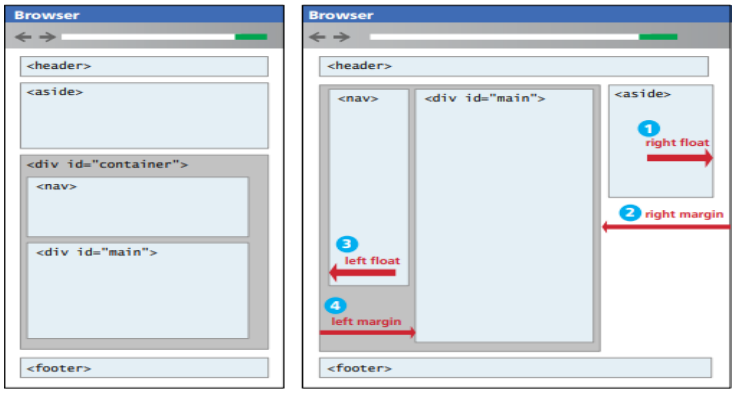


Figure 2.20: Creating a three-column layout with nested floats

- 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.

Using Positioning to Create Columns

- Positioning can also be used to create a multicolumn layout. Typically, the approach will make use of absolute position.
- Figure 2.21 illustrates a typical three-column layout implemented via 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 can be placed first.

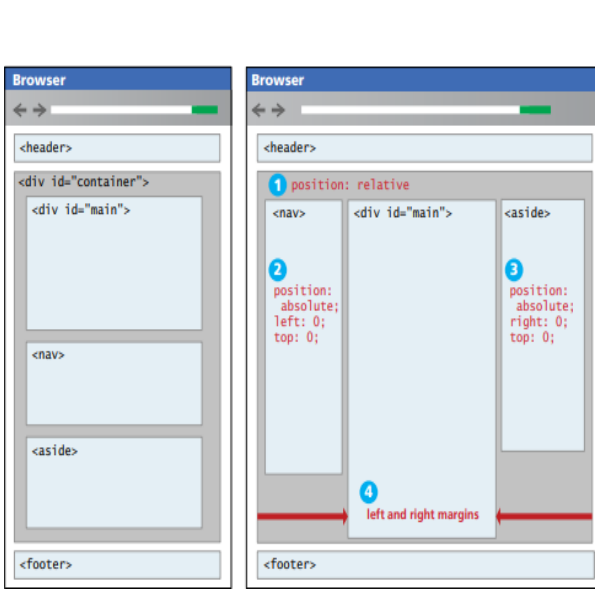


Figure 2.21: Three-column layout with positioning

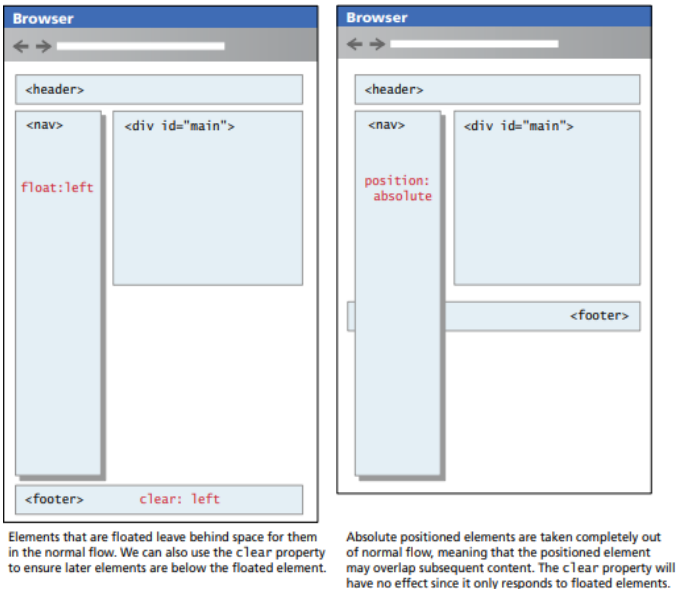


Figure 2.22: Problems with absolute positioning

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 2.22.
- One solution to this type of problem is to place the footer within the main container.

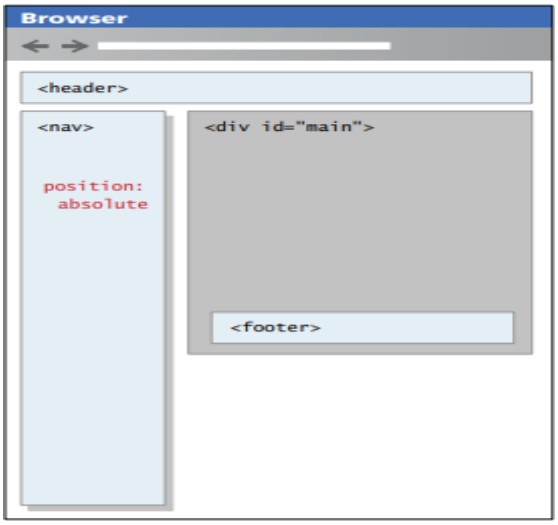


Figure 2.23: Solution to the footer problem

2.5 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.

- The two basic approaches to deal with the problems of screen size are **Fixed** and **Liquid** layouts.

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 to1000 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 container whose width property has been set to some width as shown in figure 2.24.

  Advantage

  - 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.

  Disadvantage

  - Shrinks below the fixed width; the user will have to horizontally scroll to see all the content.

- 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 shrinks below the fixed width; the user will have to horizontally scroll to see all the content, as shown in Figure 2.25.

Figure 2.24: Fixed layouts

Figure 2.25: Problems with fixed layout

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.
- 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 2.26.

Advantage

- It adapts to different browser sizes, so there is neither wasted white space nor any need for horizontal scrolling.

Disadvantages

- Liquid layouts can be more difficult to create because some elements, such as images, have fixed pixel sizes.
- As the screen grows or shrinks dramatically, in that the line length may become too long or too short.

Figure 2.26: Liquid Layouts

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 i.e., **Hybrid layout (Combination pixels and percentages).**
- 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.

2.6 Responsive Design

- **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 design**, the page "responds" to changes in the browser size that go beyond the width scaling of a liquid layout.

- 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 2.27.

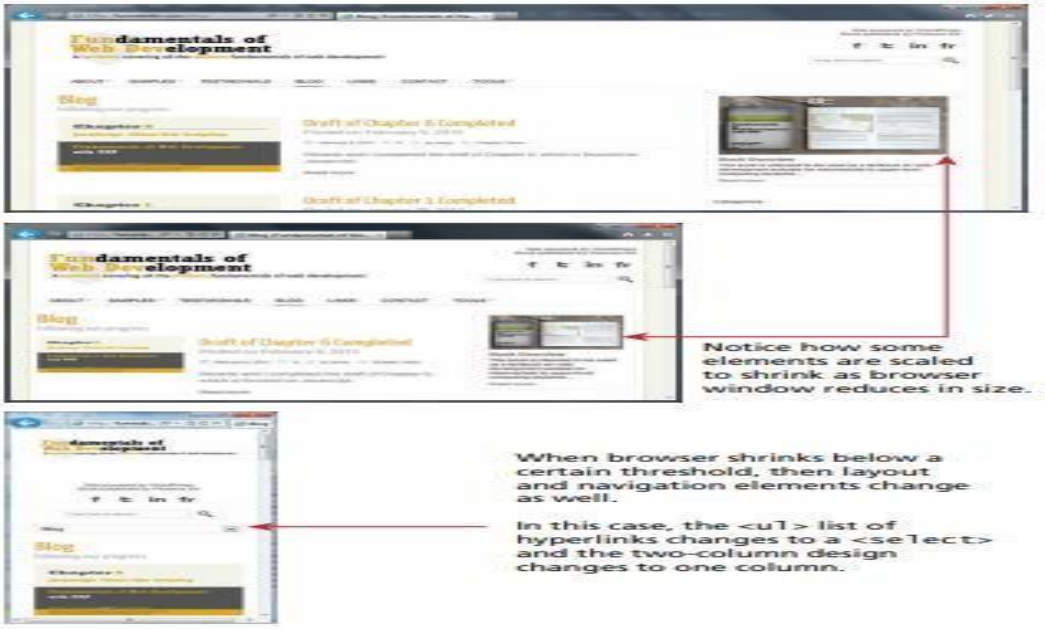


Figure 2.27: Responsive layouts

- 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.

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 we have ever used a modern mobile browser, we 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**.

  Ex: On iPhones, for instance, the viewport width is 980 px, and then that viewport is scaled to fit the current width of the device as shown in Figure 2.28.

Figure 2.28: viewports

- The mobile Safari browser introduced the viewport <meta> tag as a way for developers to control the size of that initial viewport.
- The web page can tell the mobile browser the viewport size to use via the viewport <meta> element, as shown in Listing 2.1.

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

Listing 2.1: Setting the viewports

- 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 then the viewport width will be 480 px. The result will be similar to that shown in Figure 2.29

Figure 2.29: setting the viewports

- 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.

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.
- We 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.

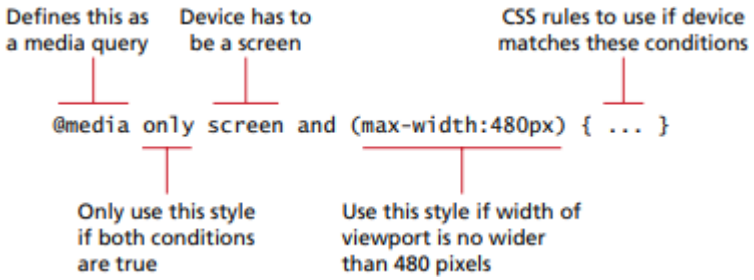


Figure 2.30: Sample media query

- Figure 2.30 illustrates the syntax of a typical media query. These queries are Boolean expressions and can be added to our CSS files or to the <link> element to conditionally use a different external CSS file based on the capabilities of the device.
- Table 2.3 is a partial list of the browser features you can examine with media queries. Many of these features have min- and max- versions.

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 2.3: Browser Features You Can Examine with Media Queries

- Contemporary responsive sites will typically provide CSS rules for phone displays first, then tablets, then desktop monitors, an approach called **progressive enhancement**

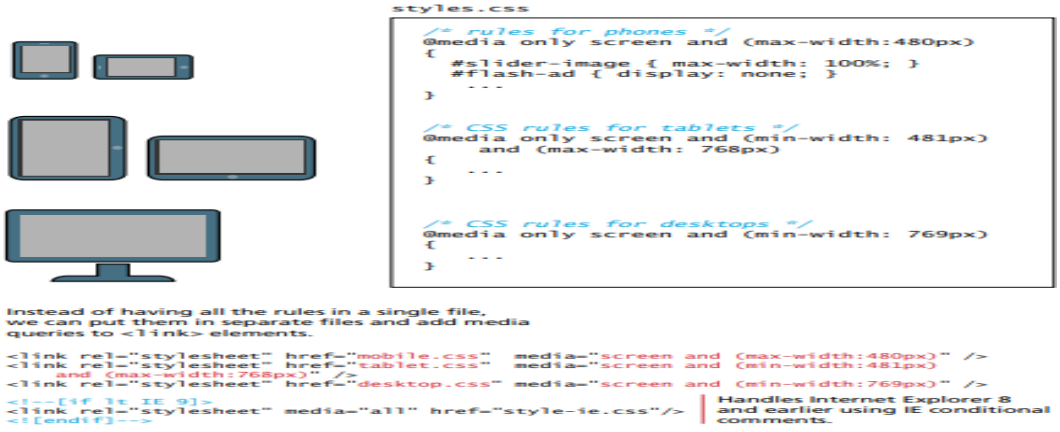


Figure 2.31: Media queries in action

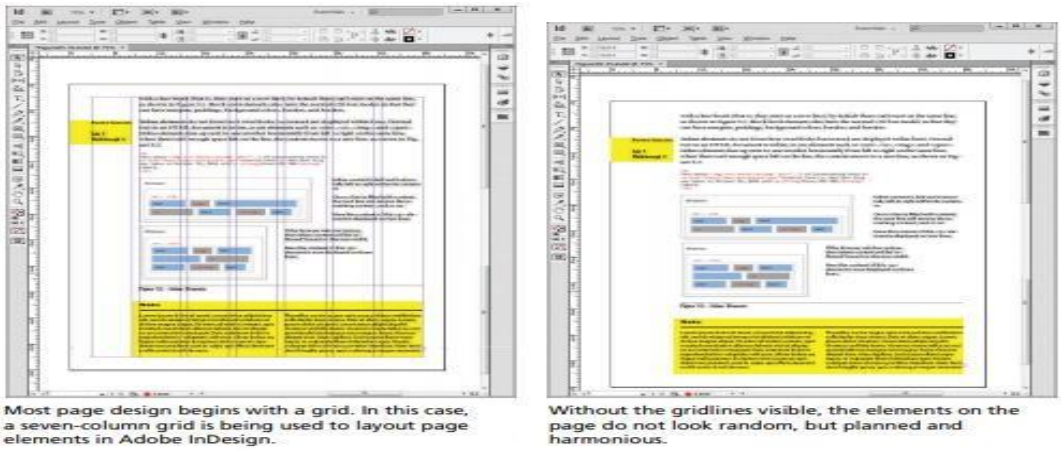
- Since later rules override earlier rules, this. provides progressive enhancement, meaning that as the device grows we can have CSS rules that take advantage of the larger space. Notice as well that these media queries can be within our CSS file or within the <link> element.

2.7 CSS Frameworks

- A **CSS framework** is a pre created 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.

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**).
- 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 2.32.



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 2.32: Using a grid in print design

- 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 elements with classes defined by the framework. The HTML elements for the rest of your site are then placed within these elements.
- Listing 2.2 illustrates a three column layout within the grid system of the 960 framework.
- Listing 2.3 shows the same thing in the Bootstrap framework.
- In both systems, elements are laid out in rows; elements in a row will span from 1to 12 columns.
- In the 960 system, a row is terminated with .
- In Bootstrap, content must be placed within the row container.

```
<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 2.2: 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 2.3: Using the 960 grid

- Both of these frameworks allow columns to be nested, making it quite easy to construct the most complex of layouts.
- 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.

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 of a CSS preprocessor is with colors.
- For instance, in Figure 2.33, 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 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.

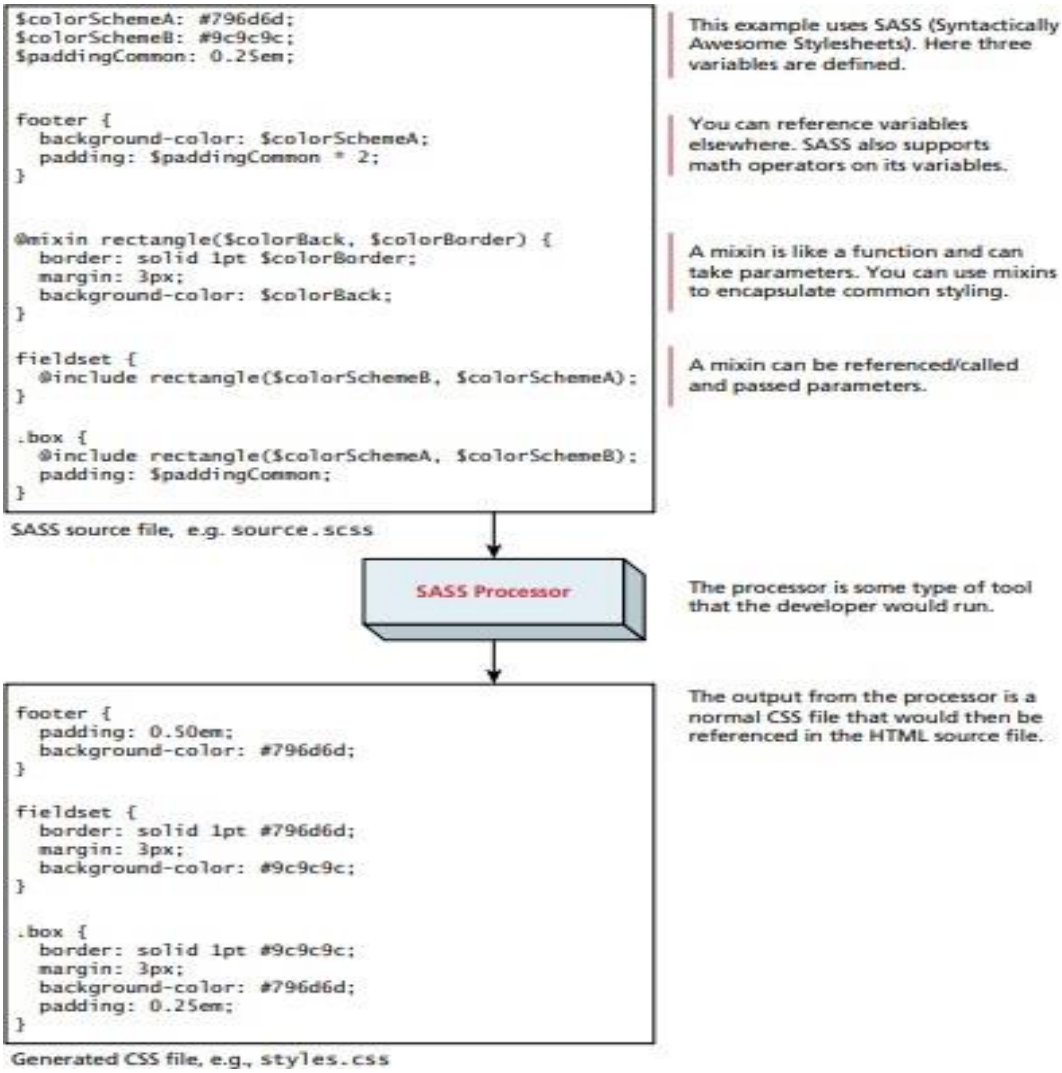


Figure 2.33: Using a CSS preprocessor