WEB TECHNOLOGY AND ITS APPLICATIONS (17CS71)

Module 2

HTML Tables and Forms, Introducing Tables, Styling Tables, Introducing Forms, Form Control Elements, Table and Form Accessibility, Microformats.

Advanced CSS: Layout, Normal Flow, Positioning Elements, Floating Elements, Constructing Multicolumn Layouts, Approaches to CSS Layout, Responsive Design, CSS Frameworks.

- **Text Book** - Randy Connolly, Ricardo Hoar, **"Fundamentals of Web Development",**

1st Edition

Chapter 4 and Chapter 5

➢ Introducing Tables

- **<table> element** is used to **create table** in HTML and can be used to represent information that exists in a **two-dimensional form**.

- Tables can be used to display **calendars, financial data, pricing tables**, and many other types of data.

➢ Basic Table Structure

- <table> contains any number of **rows (<tr>);** each row contains any number of **table data cells (Columns-<td>).**

- Some browsers do not display borders by default.

- Many tables will contain some type of headings in the first row. In HTML, you can indicate **header data** by using the **<th> element.**

- Browsers will automatically make the content within a **<th> element bold.**

- Basic table structure

<table>

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

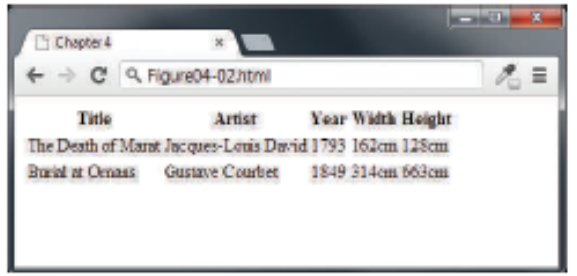
```
<table>
    <tr>
        <td>The Death of Marat</td>
        <td>Jacques-Louis David</td>
        <td>1793</td>
        <td>162cm</td>
        <td>128cm</td>
    </tr>
    <tr>
        <td>Burial at Ornans</td>
        <td>Gustave Courbet</td>
        <td>1849</td>
        <td>314cm</td>
       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

- Adding table headings

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

<table> <tr> <tr>

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



- Spanning Rows and Columns

- Each row must have the same number of <td> or <th> containers. There is a way to change this behaviour.

- If you want a given cell to **cover several columns or rows**, (to merge rows and columns) then you can do that by using the **colspan** or **rowspan** attributes.

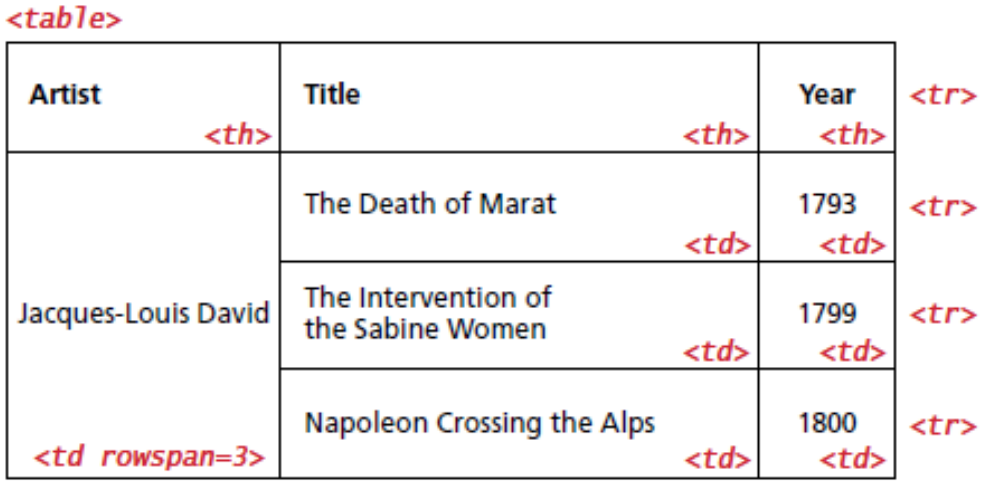
- Spanning columns

<table>
<tr><th>Title</th><th>Artist</th><th>Year</th><th colspan="2">Size (width x 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>

Notice that this row now only has four cell elements.

```
<table>
   <tr>
      <th>Title</th>
      <th>Artist</th>
      <th>Year</th>
      <th colspan="2">Size (width x 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>
   ...
</table>
```

- Spanning rows




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

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

➢ Additional Table Elements

1. The **<caption> element** is used to provide a brief **title or description** of the table, which improves the accessibility of the table.

- **caption-side** CSS property can be used to change the position of the caption below the table.

2. The **<thead>, <tfoot>, and <tbody>** elements are used for table header, table footer and table body.

3. The **<col> and <colgroup>** elements are mainly used for styling all columns within a <colgroup> with just a single style.

- Unfortunately, the only properties you can set via these two elements are borders, backgrounds, width, and visibility, and only if they are not overridden in a <td>, <th>, or <tr> element.

- Additional table element

A title for the table is good for accessibility.

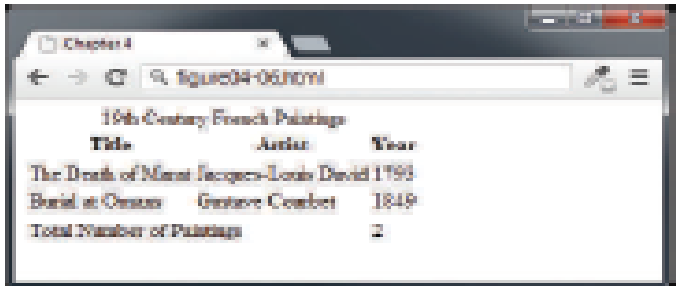
These describe our columns, and can be used to aid in styling.

Table header could potentially also include other <tr> elements.

Yes, the table footer comes *before* the body.

Potentially, with styling the browser can scroll this information, while keeping the header and footer fixed in place.

```
<table>
    <caption>19th Century French Paintings</caption>
    <col class="artistName" />
    <colgroup id="paintingColumns">
        <col />
        <col />
    </colgroup>

    <thead>
        <tr>
            <th>Title</th>
            <th>Artist</th>
            <th>Year</th>
        </tr>
    </thead>

    <tfoot>
        <tr>
            <td colspan="2">Total Number of Paintings</td>
            <td>2</td>
        </tr>
    </tfoot>

    <tbody>
        <tr>
            <td>The Death of Marat</td>
            <td>Jacques-Louis David</td>
            <td>1793</td>
        </tr>
        <tr>
            <td>Burial at Ornans</td>
            <td>Gustave Courbet</td>
            <td>1849</td>
        </tr>
    </tbody>

</table>
```



➢ Using Tables for Layout

- HTML tables were frequently used to create page layouts. Since HTML block-level elements exist on their own line.

The practice of using tables for layout had some problems.

1. This approach will **increase the size of the HTML document**. the large number of extra tags required for <table> elements can significantly bloat the HTML document.

- These larger files take **longer to download**, more **difficult to maintain** because of the extra markup.

2. A second problem with using tables is that the resulting markup is **not semantic**. Tables are meant to indicate tabular data;

3. The other key problem is that using tables for layout results in a page that is **not accessible.** It is much better to use CSS for layout.

➢ Styling Tables

- There is certainly no limit to style a table , Table can be style by using CSS properties.

1. Table Borders

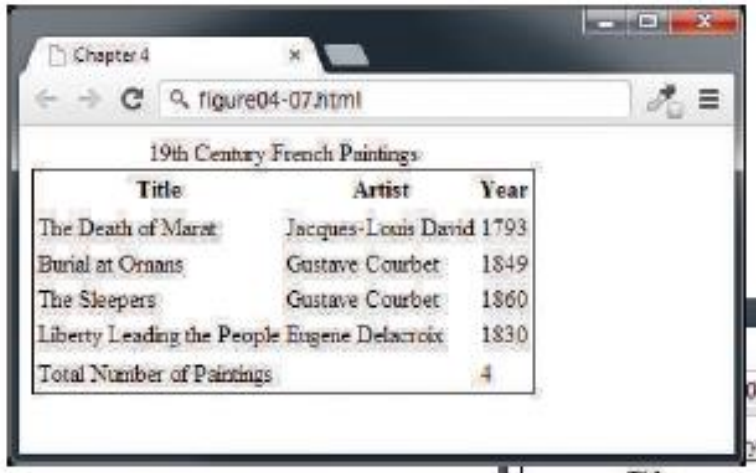
- Borders can be assigned to both the <table> and the <td> element.

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

- border-collapse property This property selects the table's border & removes the cell border.

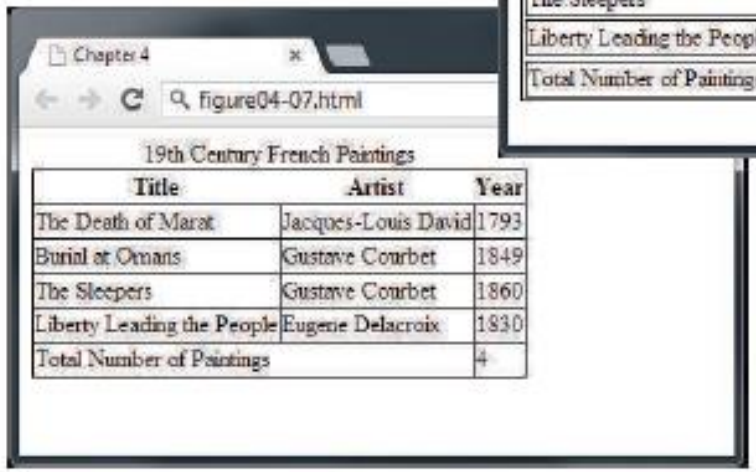
- **width, height—**for setting the width and height of cells
- **cellspacing—**for adding space between every cell in the table
- **cellpadding**—for adding space between the content of the cell and its border
- **bgcolor**—for changing the background color of any table element
- **background**—for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

- Styling table borders



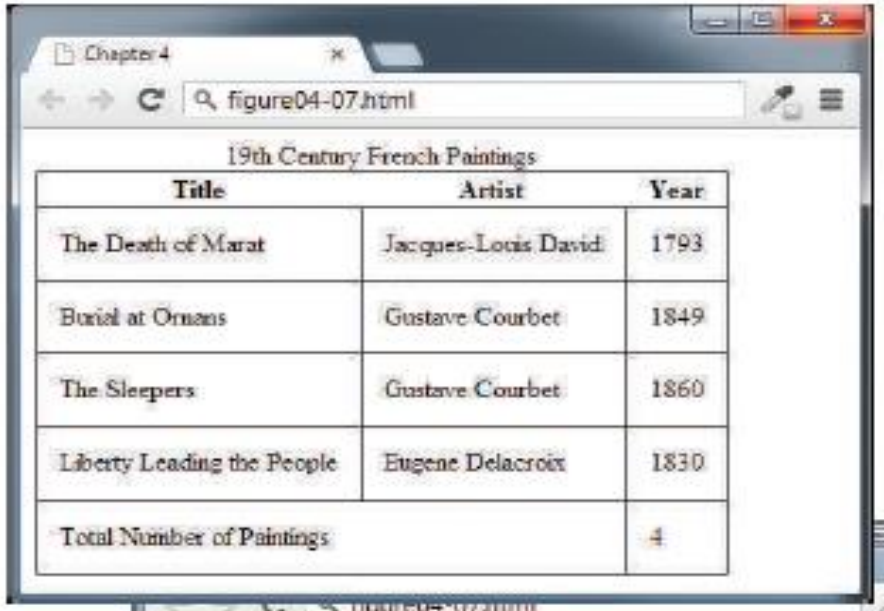

```
table {
   border: solid 1pt black;
}
```


```
table {
   border: solid 1pt black;
}
td {
   border: solid 1pt black;
}
```




```
table {
   border: solid 1pt black;
   border-collapse: collapse;
}
td {
   border: solid 1pt black;
}
```

- Styling table borders cont..




```
table {
    border: solid 1pt black;
    border-collapse: collapse;
}
td {
    border: solid 1pt black;
    padding: 10pt;
}
```

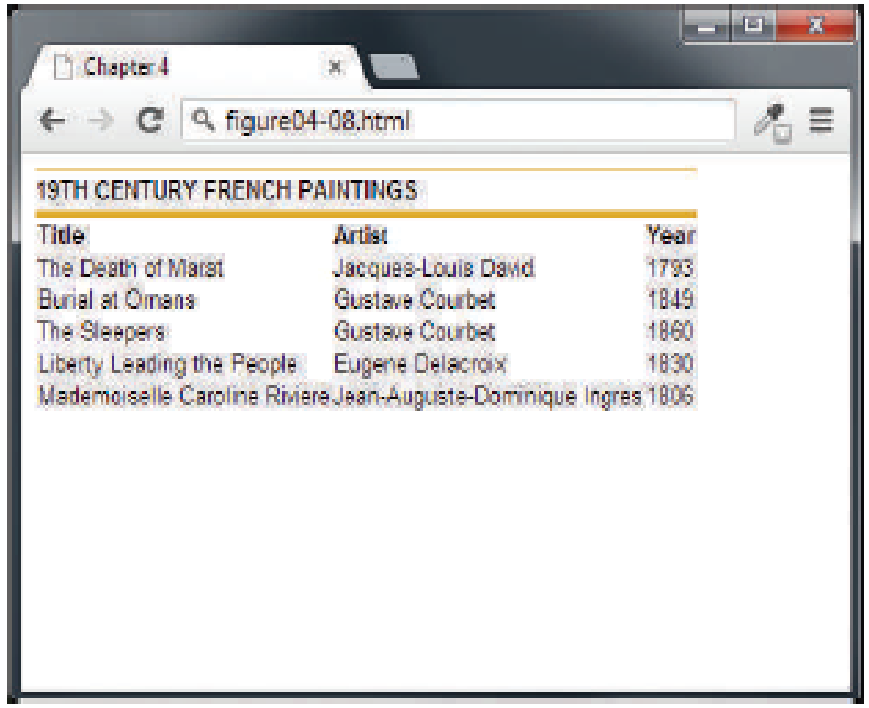

```
table {
    border: solid 1pt black;
    border-spacing: 10pt;
}
td {
    border: solid 1pt black;
}
```

2. Boxes and Zebras

- There are different ways one can style a table let discuss two of them here.

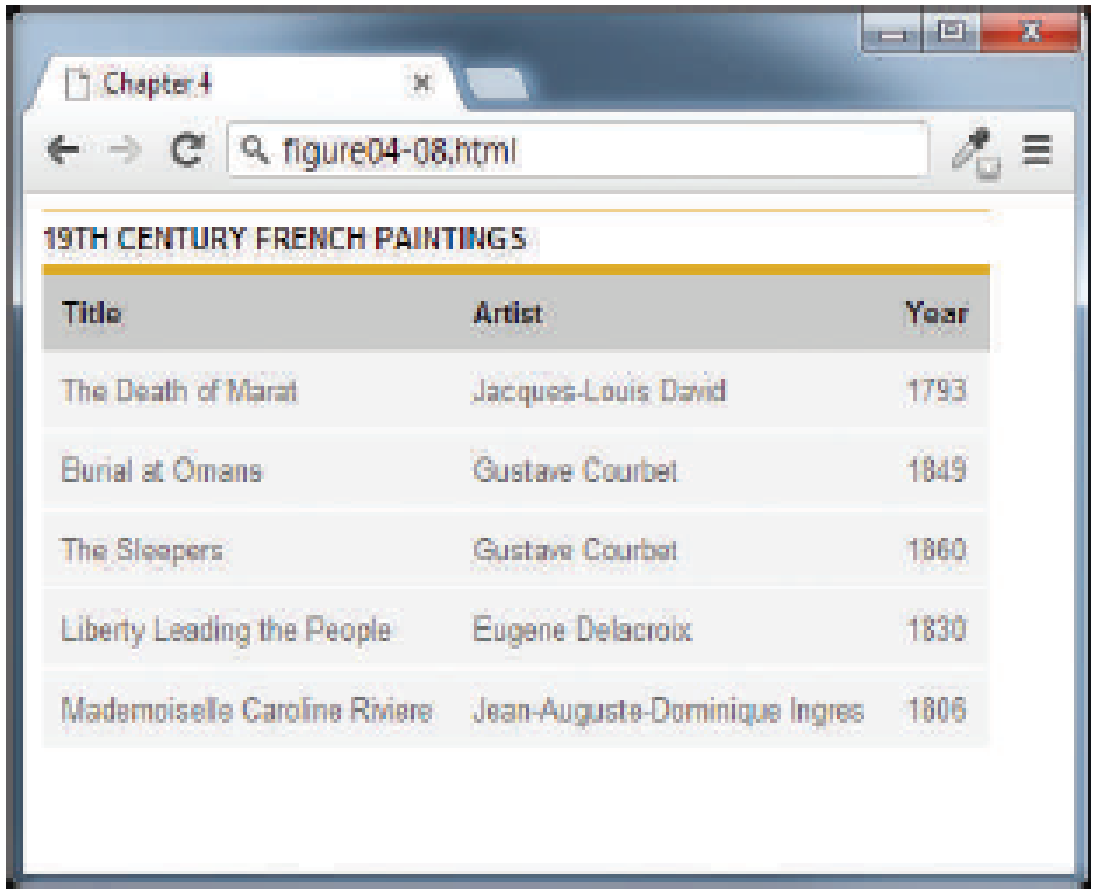
i). The first of these is a **box format**, in which we simply apply **background colors and borders** in various ways.

An example boxed table




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

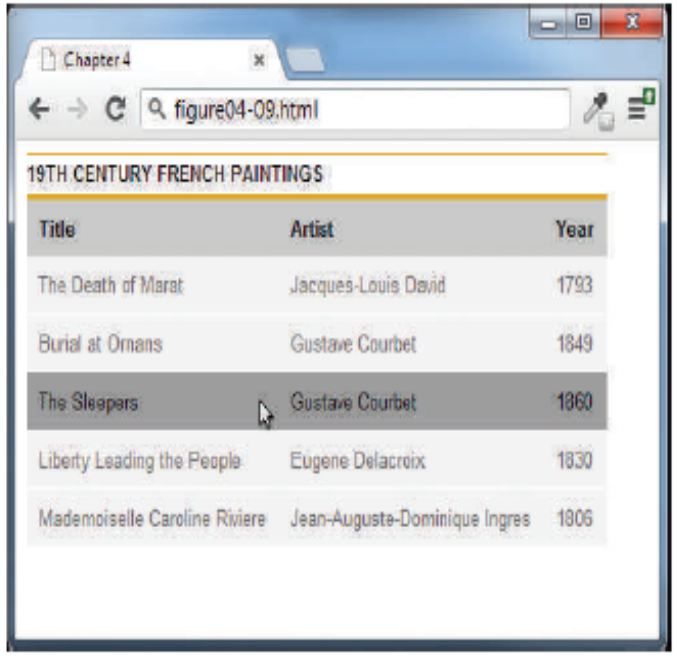
Cont..



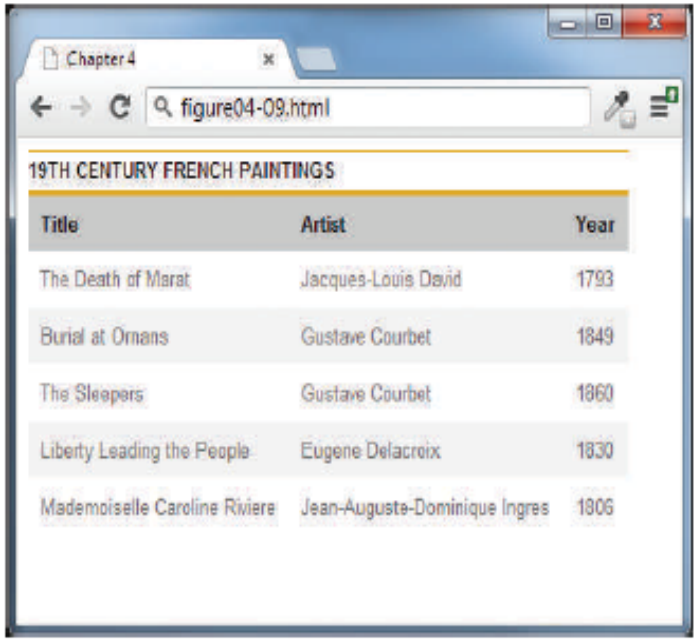

```
tbody tr {
    background-color: #F1F1F1;
    border-bottom: 1px solid white;
    color: #6E6E6E;
}
tbody td {
    padding: 0.75em;
}
```

ii). Zebras – Can add special styling to the <tr> element, to highlight a row when the mouse cursor hovers over a cell with **:hover pseudo-class.**

- **pseudo-element nth-child** is used to alternate the format of every second row.




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




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

- Hover effect and zebra stripes

➢ Introducing Forms

- Forms provide the user with an alternative way to **interact with a web server**.

- Up to now, clicking hyperlinks was the only mechanism available to the user for communicating with the server. Forms provide a much richer mechanism.

- Using a form, the user can **enter text, choose items from lists, and click buttons**.

- Typically, programs running on the server will take the input from HTML forms and do something with it, such as **save it in a database**, **interact with an external web service**, or **customize subsequent HTML based on that input**.

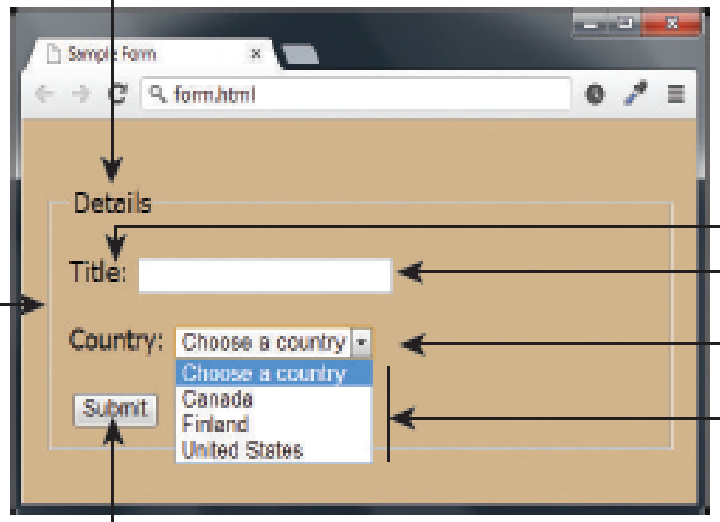
➢ Form Structure

- A form is defined by a **<form> element**.

- Which is a **container** for other elements that represent the various input elements within the form as well as plain text and almost any other HTML element.

- But a form **cannot contain another <form>** element.

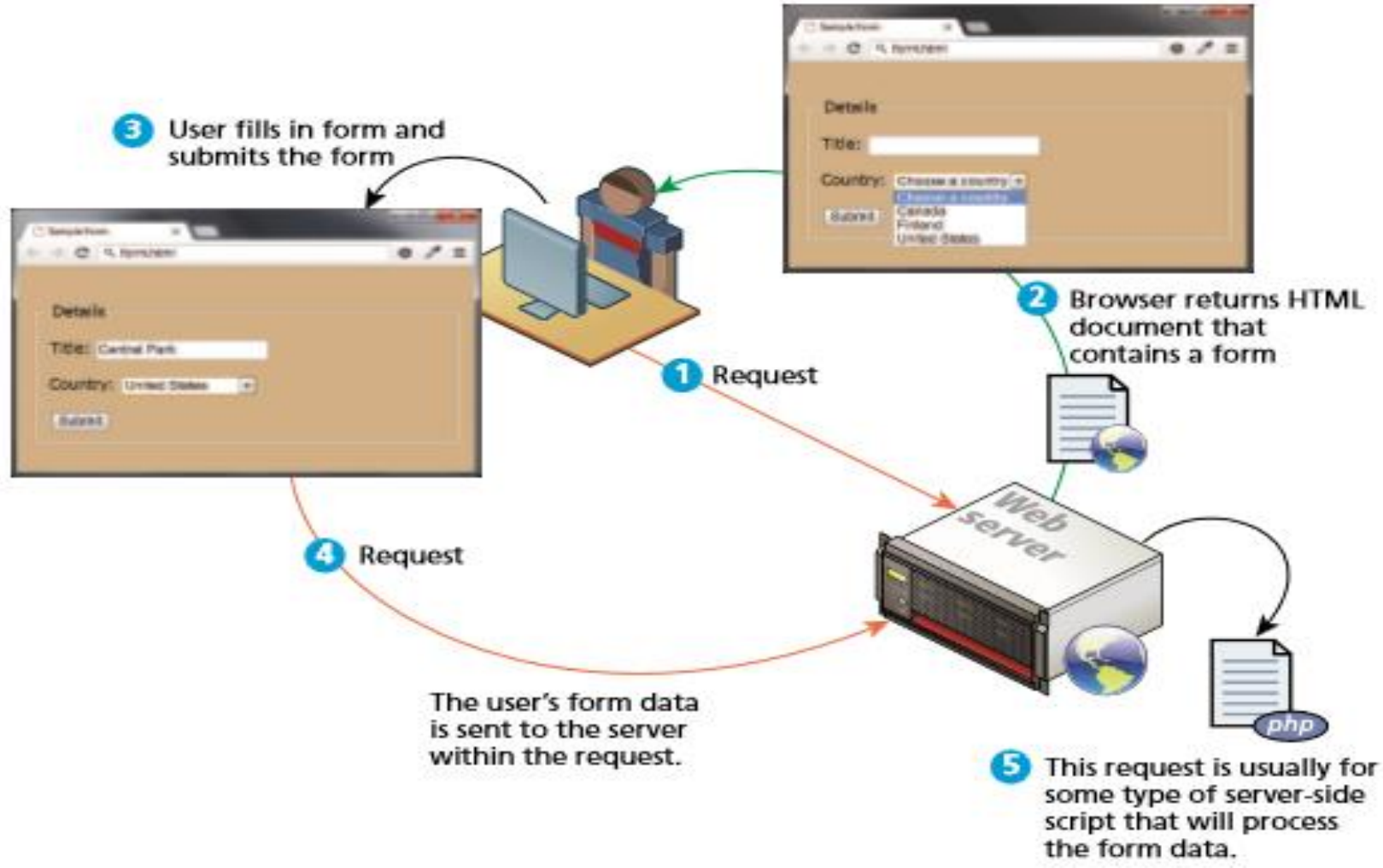
- **Sample HTML form**




```
<form method="get" action="process.php">
 <fieldset>
  <legend>Details</legend>
  <p>
   <label>Title: </label>
   <input type="text" name="title" />
  </p>
  <p>
   <label>Country: </label>
   <select name="where">
    <option>Choose a country</option>
    <option>Canada</option>
    <option>Finland</option>
    <option>United States</option>
   </select>
  </p>
  <input type="submit" />
 </fieldset>
</form>
```

➢ How Forms Work

- While forms are constructed with HTML elements, but it also requires some type of server-side resource that processes the user's form input.



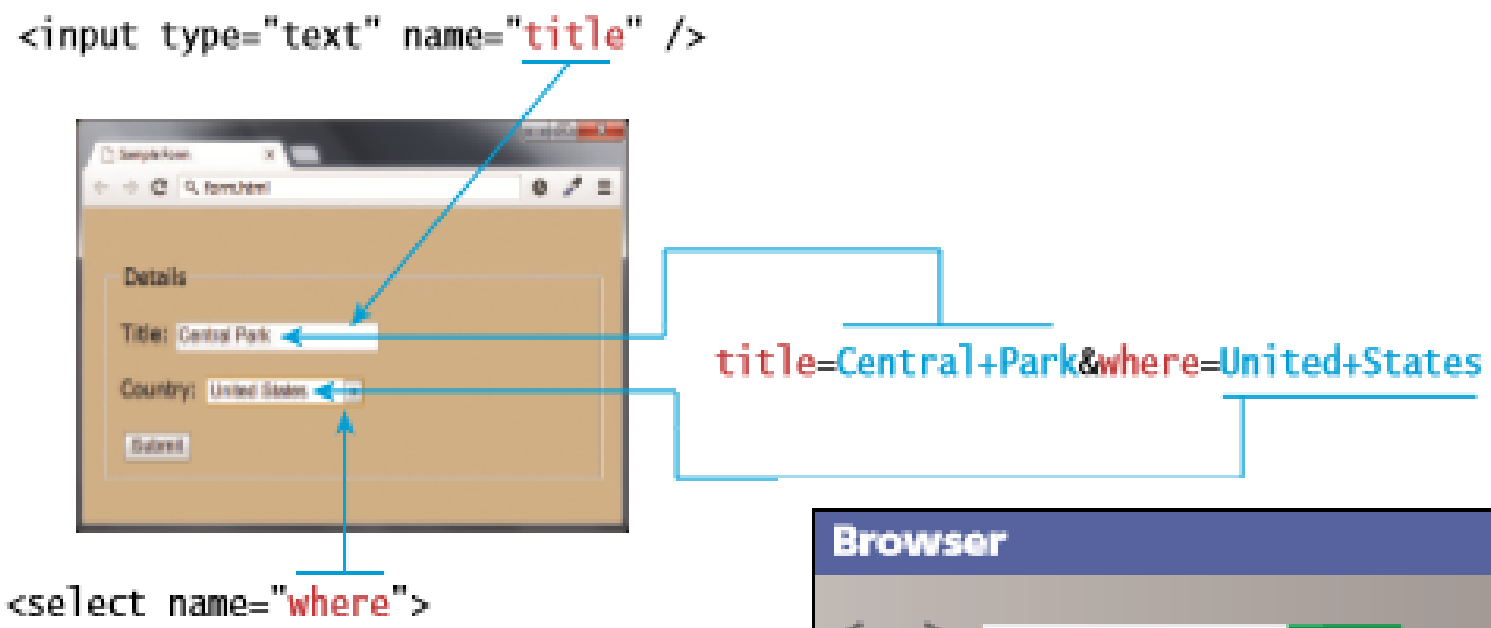
➢ Query strings

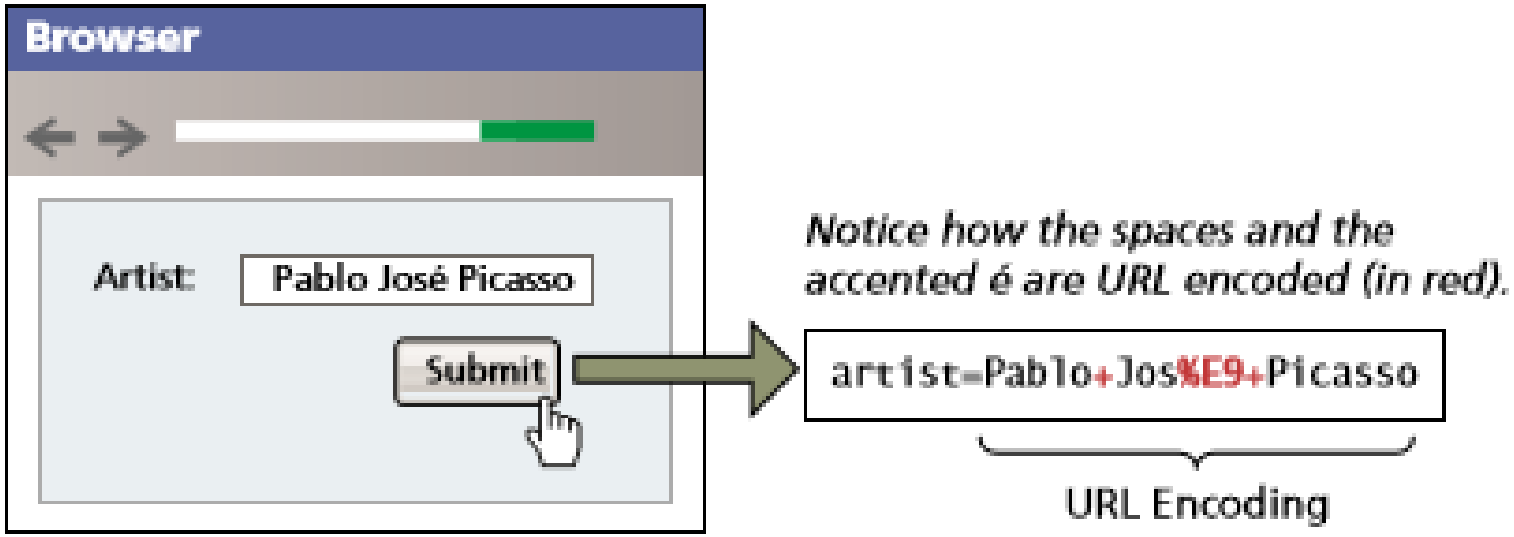
- Interaction between the **browser** and **the web server** is governed by the **HTTP protocol**.

- The form data must be sent to the server via a standard **HTTP request**. This request is typically some type of server-side program that will process the form data.

- **But how is the data packaged in a request?**
- The browser packages the user's data input into something called a **query string.**
- A query string is a series of **name=value pairs** separated by **ampersands**.
- Each form 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.**
- 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**)

- Figure : Query string data





- Figure : URL encoding

➢ The <form> element

- There are **two important attributes** those are essential for any form, namely the **action** and the **method** attributes

```
<form method="get" action="process.php">
```

- 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.

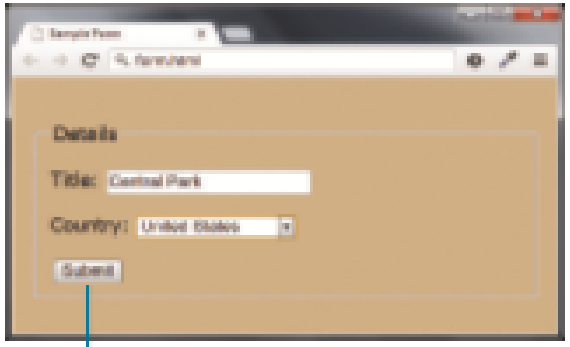
- Example of server side programing PHP pages, ASP.NET (.aspx), ASP (.asp), and Java Server Pages (.jsp).

- 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 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.




```
<form method="get" action="process.php">
```

```
GET /process.php?title=Central+Park&where=United+States http/1.1
```

querystring

```
<form method="post" action="process.php">
```

```
POST /process.php http/1.1
Date: Sun, 20 May 2012 23:59:59 GMT
Host: www.mysite.com
User-Agent: Mozilla/4.0
Content-Length: 47

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

HTTP Header

querystring

- **Which of these two methods should one use? (GET or POST)**

- Generally, form data is sent using the POST method. However, the GET method is useful when you are **testing or developing a system**, since you can examine the query string directly in the browser's address bar.

- Since the GET method uses the URL to transmit the query string, form data will be saved when the user **bookmarks a page**, which may be desirable, but is generally a potential **security risk** for shared use computers.

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.

- Form-Related HTML Elements

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.

➢ Form Control elements

1. Text input Controls
2. Choice Controls
3. Button Controls
4. Specialized Controls
5. Date and Time Controls

➢ 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 for that.

- List of Text Input Controls

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


```
<textarea>
  enter some text
</textarea>
```

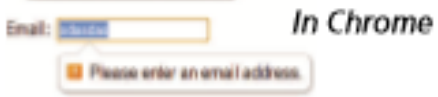
```
<textarea placeholder="enter some text">
</textarea>
```


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


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


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

In Opera



In Chrome

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


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

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" ... />`

➢ Choice Controls

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

- **select Lists**

- The **<select>** element is used to **create a multiline box** for selecting one or more items.

- The options (defined using the **<option> element**) can be hidden in a 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.

- 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.

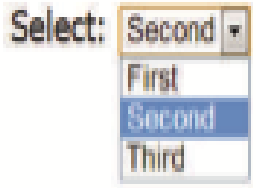
- The value attribute is optional; if it is not specified, then the text within the container is sent to the server.

- Using the <select> element

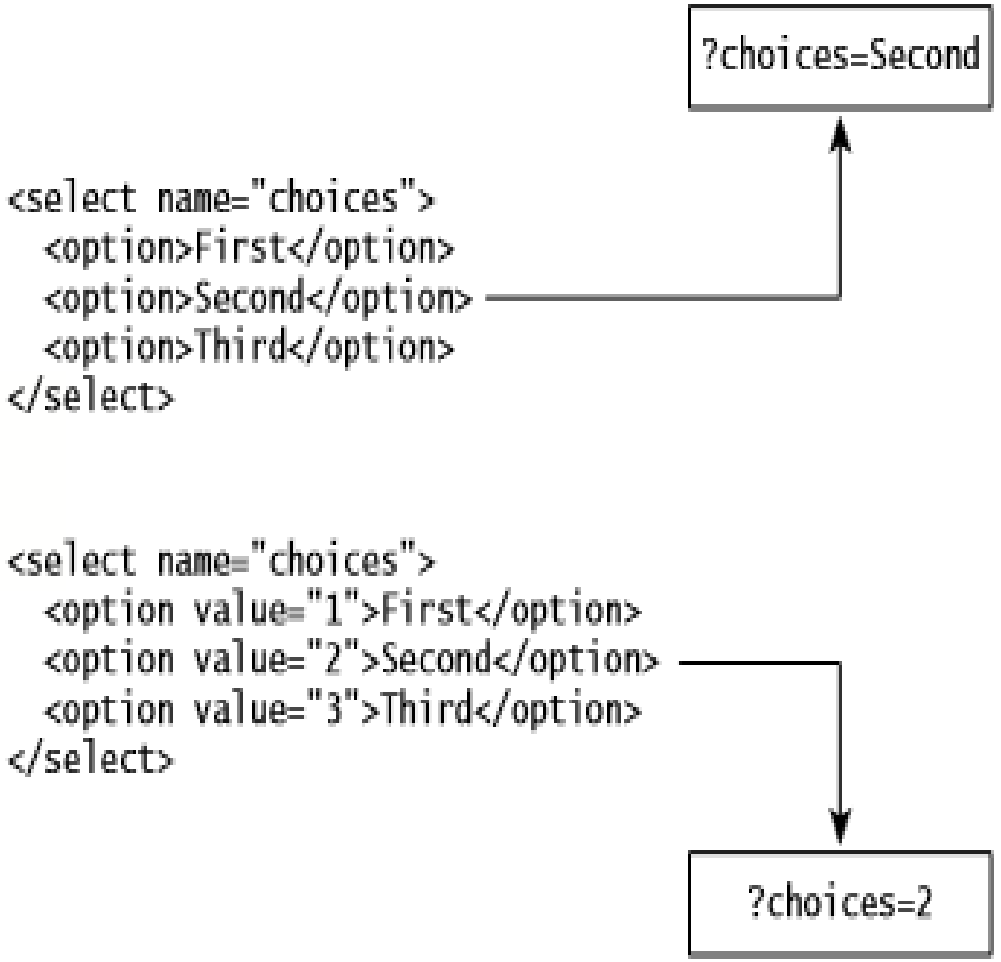
```
<select name="choices">
  <option>First</option>
  <option selected>Second</option>
  <option>Third</option>
</select>
```

```
<select size="3" ... >
```

```
<select … >
 <optgroup label="North America">
   <option>Calgary</option>
   <option>Los Angeles</option>
 </optgroup>
 <optgroup label="Europe">
   <option>London</option>
   <option>Paris</option>
   <option>Prague</option>
 </optgroup>
</select>
```



- The value attribute



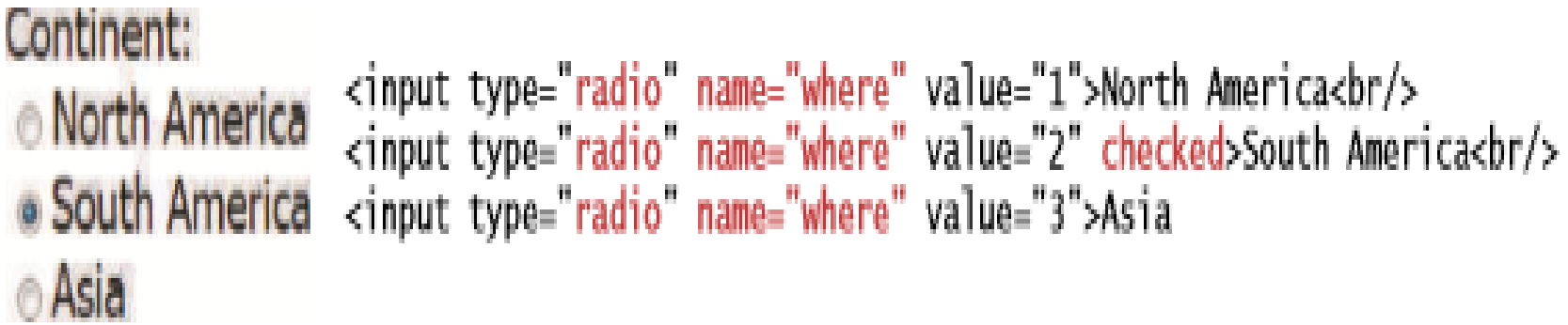
➢ Radio buttons

-Radio buttons are useful when you want the user to **select a single item** from a small list of choices and you want all the choices to be visible.

- Radio buttons are added via the **<input type="radio">** element.

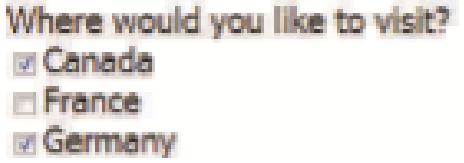
- The buttons are made mutually exclusive (i.e., only one can be chosen) by sharing the **same name attribute**.

- The **checked attribute** is used to indicate the default choice.



➢ Checkboxes

- Checkboxes are used for getting **yes/no or on/off responses** from the user
- Checkboxes are added via the **<input type="checkbox">** element.
- You can also group checkboxes together by having the same name attribute.
- **checked attribute** can be used to set the default value of a checkbox.



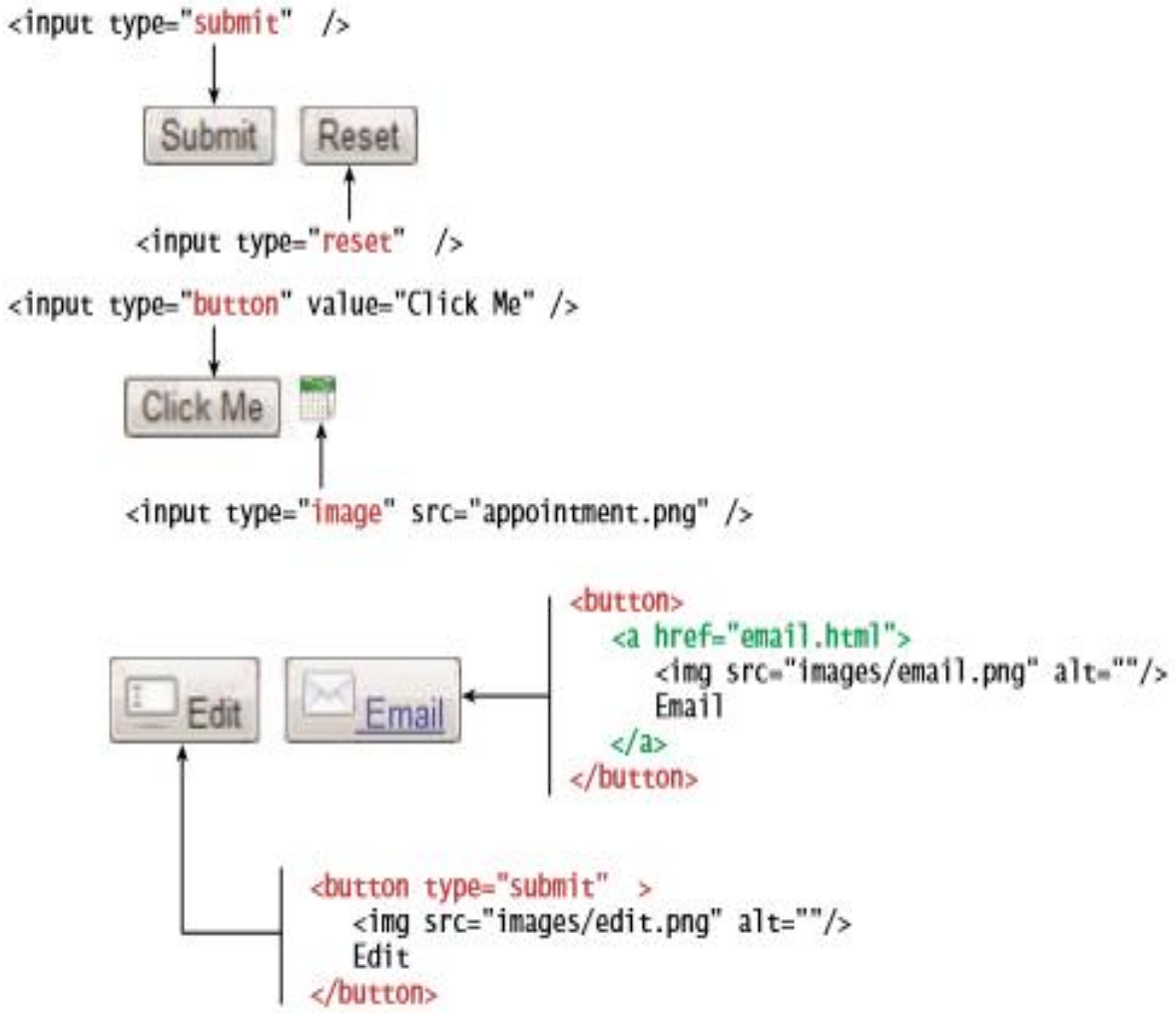

```
<label>Where would you like to visit? </label><br/>
<input type="checkbox" name="visit" value="canada">Canada<br/>
<input type="checkbox" name="visit" value="france">France<br/>
<input type="checkbox" name="visit" value="germany">Germany
```

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

➢ button Controls

- HTML defines several different types of buttons

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.



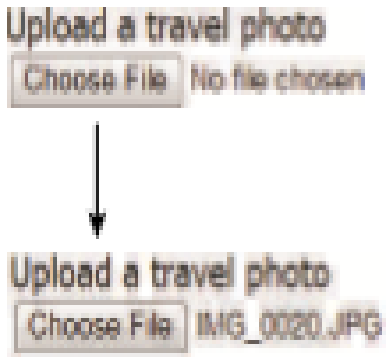
➢ Specialized Controls

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

i. **<input type="hidden">** element – allows the developers to include data that cannot be seen or modified by users when a form is submitted.

ii. **<input type="file">** element - which is used to **upload a file** from the client to the server.

- **Notice** that the <form> element must use the **post method** and it must include the **enctype="multipart/form-data**" attribute as well.




```
<form method="post" enctype="multipart/form-data" ... >
    ...
    <label>Upload a travel photo</label>
    <input type="file" name="photo" />
    ...
</form>
```

➢ Number and Range

- HTML5 introduced two new controls for the **input of numeric values**.

- When input via a standard text control, numbers typically require validation to ensure that the user has entered an actual number

and, because the range of numbers is infinite, the entered number has to be checked to ensure it is not too small or too large.

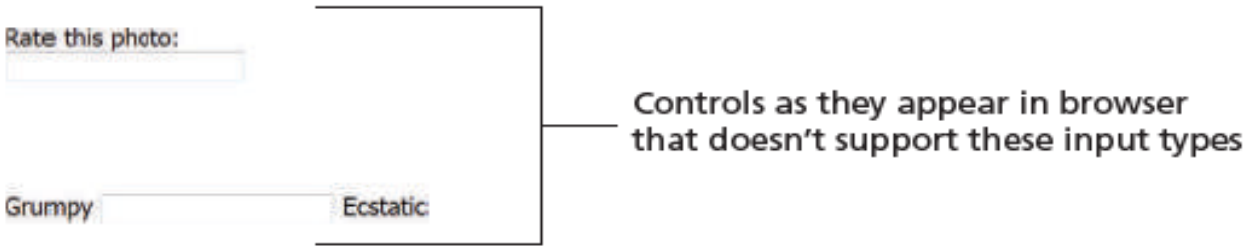
- The **number and range controls** provide a way to input numeric values that eliminate the need for client-side numeric validation.

Rate this photo:
2

```
<label>Rate this photo: <br/>
<input type="number" min="1" max="5" name="rate" />
```

Grumpy Ecstatic

```
Grumpy
<input type="range" min="0" max="10" step="1" name="happiness" />
Ecstatic
```

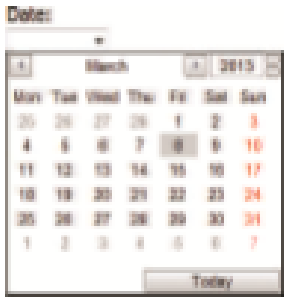


➢ Date and time Controls

- 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: which format should follow when entering a date into a web form.

-The **new date and time controls** in HTML try to make it easier for users to input these tricky date and time values




```
<label>Date: <br/>
<input type="date" ... />
```


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


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


```
<input type="datetime-local" ... />
```


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


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

➢ Table and Form accessibility

- Not all web users are able to view the content on web pages in the same manner.

- Users with **sight disabilities**, experience the web using voice reading software.

- Color blind users might have trouble in differentiating certain colors.

- 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

➢ Web Content Accessibility Guidelines

1. 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.

2. Create content that can be presented in different ways (for example simpler layout) without losing information or structure.

3. Make all functionality available from a keyboard.

4. Provide ways to help users navigate, find content, and determine where they are.

➢ Accessible tables

- Users who rely on visual readers can find difficult in pages with many tables .

- One vital way to improve the situation is to use tables only 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.

2. Connect the cells with a **textual description** in the header.

- **scope attribute** is used to connect cells with their headers.

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

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

Connecting cells with headers

➢ Accessible Forms

- Describe the form content using <fieldset>, <legend>, and <label> elements, which provide a connection between the input elements in the form and their actual meaning.

- While the browser does provide some unique formatting to the <fieldset> and <legend> elements, their main purpose is to logically group related form input elements together, with the <legend> providing a type of caption for those elements.

- The <label> element has no special formatting, Each <label> element should be associated with a single input element.

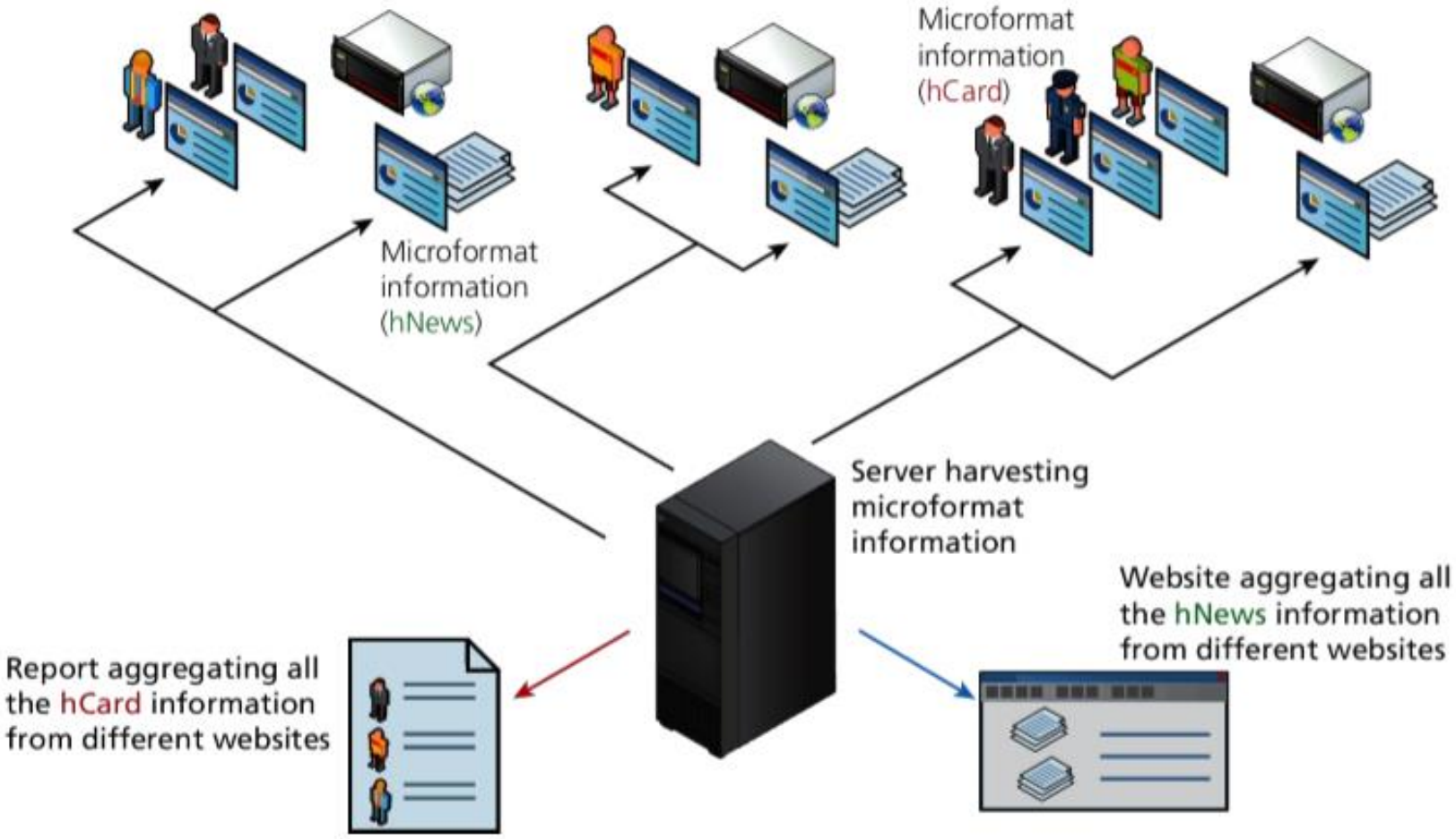
- But can make this association **explicit** by using the **for attribute**, meaning that if the user clicks on the <label> text, that control will receive the form's focus.

```
<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>
```

➢ Microformats

- The web has millions of pages in it. Yet despite the incredible variety, there is a surprising amount of **similar information from site to site**.
- Most sites have some type of Contact us pages. Similarly, many sites contain 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.**
- 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 Microformats



- One of the most common microformat is **hcard**, which is used to semantically mark up contact information of a person.

```
<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>
```

- Example of an hCard

Advanced CSS: Layout

➢ Normal Flow

- Normal flow, which refers how the browser will normally display **block-level elements** and **inline elements** from left to right and from top to bottom.

- **Block-level elements** such as <p>, , <h2>, <ul>, and <table> are each contained on their own line. Because block-level elements begin with a line break .

- Without styling, two block-level elements can't exist on the same line.

```
<h1> …                    </h1>
<ul>


</ul>
<p>


</p>
<div>


</div>
```

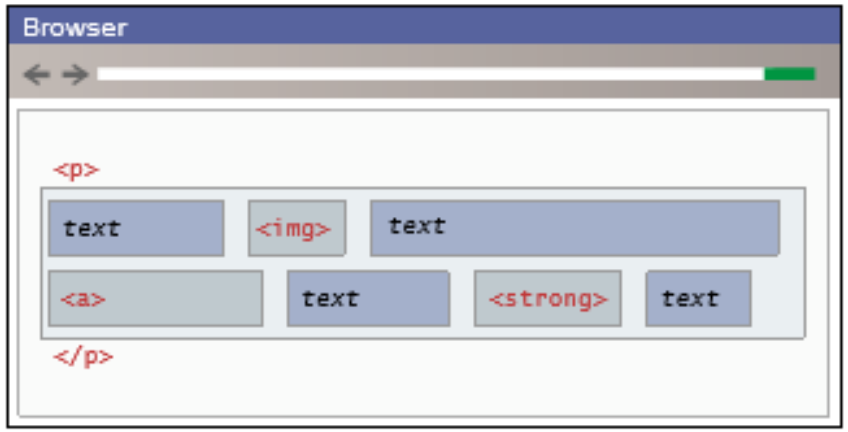
Each block exists on its own line and is displayed in normal fow from the browser window's top to its bottom.

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

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

- **Inline elements** do not form their own blocks but instead are displayed within lines. Normal text in an HTML document is inline

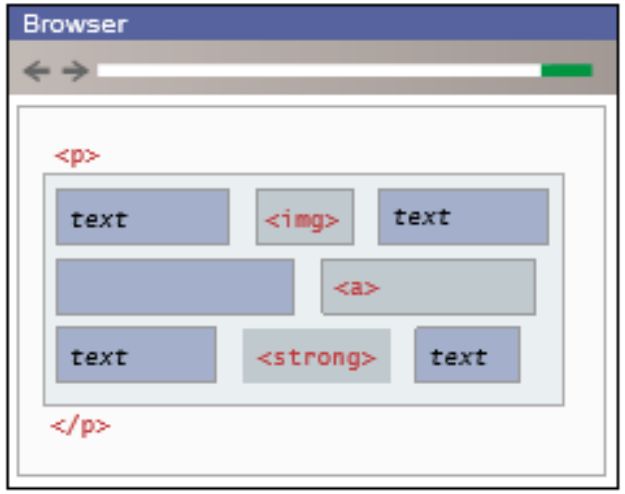
- Inline elements line up next to one another horizontally from left to right on the same line; when there isn't enough space left on the line, the content moves to a new line.



Inline content is laid out horizontally left to right within its container.

Once a line is filled with content, the next line will receive the remaining content, and so on.

Here the content of this <p> element is displayed on two lines.



If the browser window resizes, then inline content will be "refowed" based on the new width.

Here the content of this <p> element is now displayed on three lines.

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

- **Replaced inline elements** are elements whose content and thus appearance is defined by some external resource, such as <img> and the various form elements.

- **Nonreplaced inline elements** are those elements whose content is defined within the document, which includes all the other inline elements.

- 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; }
```

➢ Positioning elements

- It is possible to move an item from its regular position in the normal flow by using position property.

- The possible values of position property are.

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.

➢ 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.

```
<!DOCTYPE html>
<html>
<head>

<style>
figure {
  position: absolute;
top: 150px;
left: 200px;
}
</style>

</head>
<body>

<p>Lorem Ipsum is simply dummy text of the printing and typesetting
industry.<p>
<figure>
<img src="../tree.png" alt="" />
<figcaption>British Museum</figcaption>
</figure>

<p>Lorem Ipsum is simply dummy text of the printing and typesetting
industry.<p>
</body>
</html>
```

➢ 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.

```
<!DOCTYPE html>
<html>
<head>

<style>
figure {
  position: relative;
top: 150px;
left: 200px;
}
</style>

</head>
<body>

<p>Lorem Ipsum is simply dummy text of the printing and typesetting
industry.<p>
<figure>
<img src="../tree.png" alt="" />
<figcaption>British Museum</figcaption>
</figure>

<p>Lorem Ipsum is simply dummy text of the printing and typesetting
industry.<p>
</body>
</html>
```

➢ Z-Index

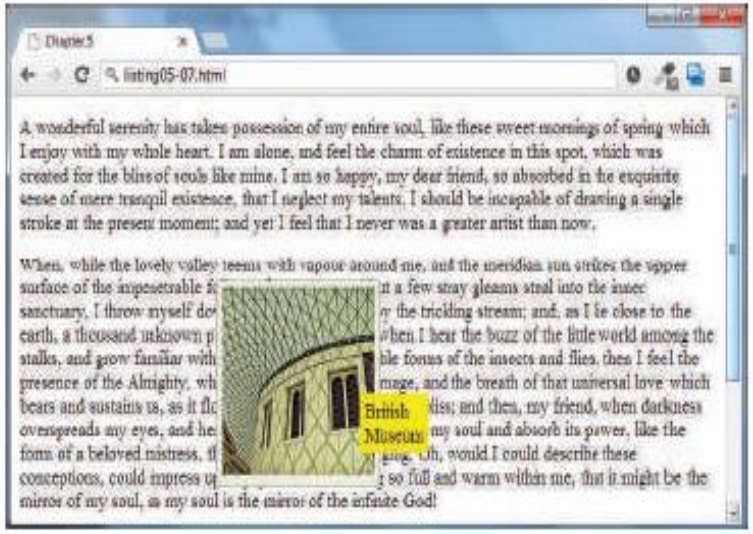
- Each positioned element has a stacking order defined by the **z-index property** (named for the z-axis).

- Items closest to the viewer have a larger z-index.

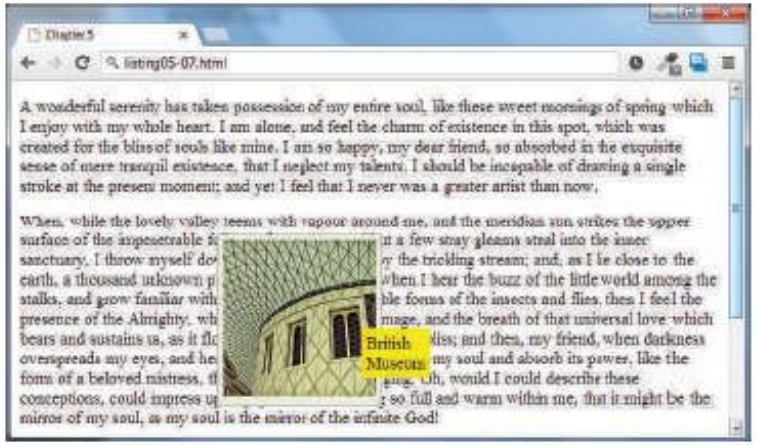
- working with z-index can be tricky

- First, only positioned elements will make use of their z-index.

- Second, as in below fig, simply setting the z-index value of elements will not necessarily move them on top or behind other items.

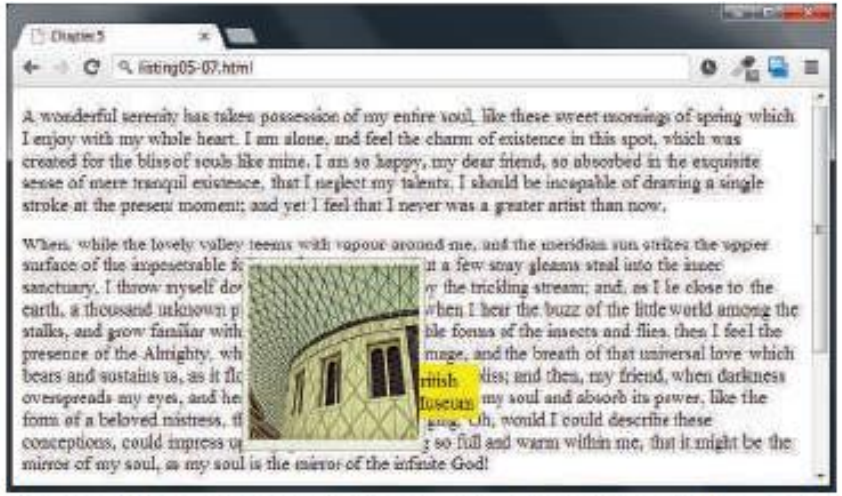


```
figure {
   position: absolute;
   top: 150px;
   left: 200px;
}
figcaption {
   position: absolute;
   top: 90px;
   left: 140px;
}
```



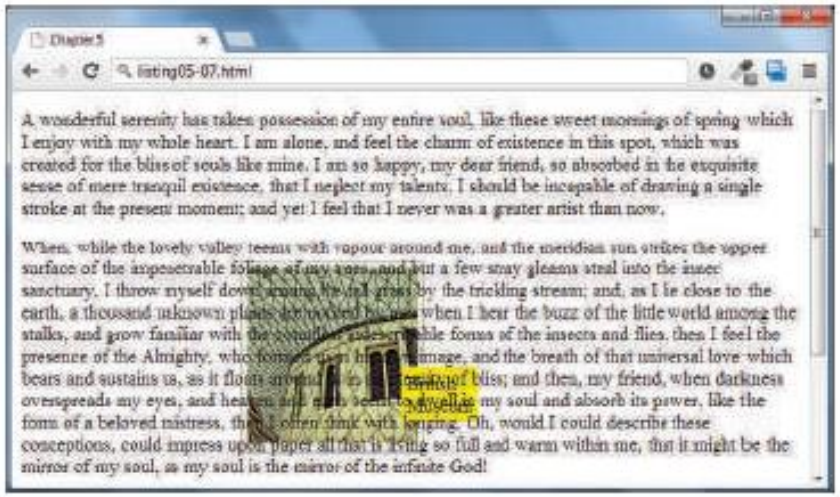
```
figure {
   …
   z-index: 5;
}
figcaption {
   …
   z-index: 1;
}
```

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




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

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




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

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

➢ Fixed Position

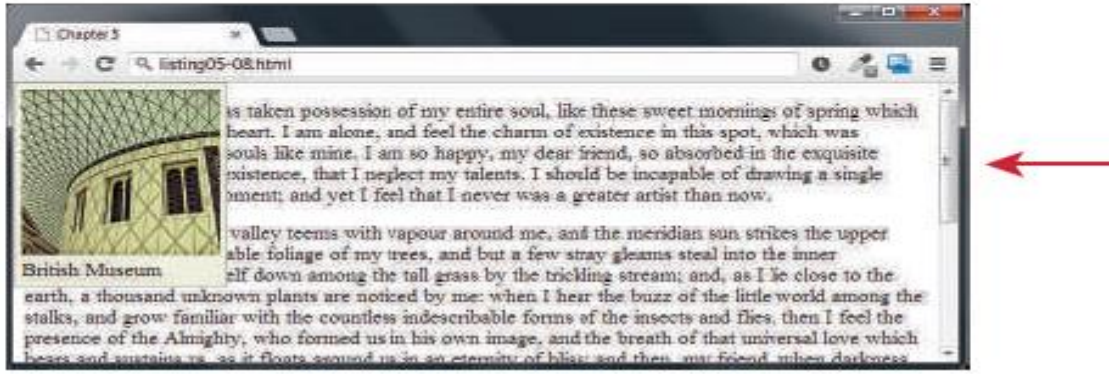
- The fixed position value is used relatively infrequently.

- It’s a type of absolute positioning, except that the positioning values are in relation to the viewport.

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

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



➢ Floating elements

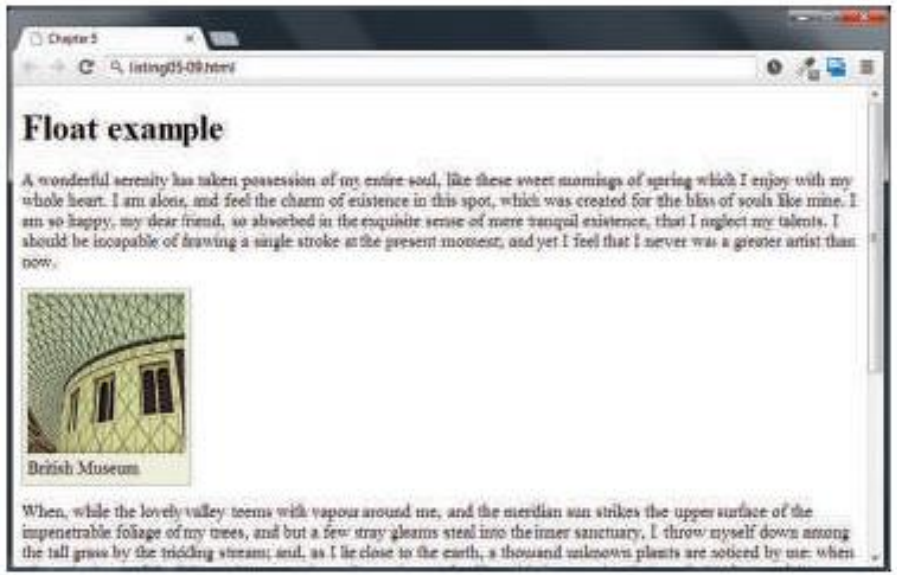
- It is possible to **displace an element** out of its position in the normal flow via the css **float property.**

- 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.

- Note - A floated block-level element must have a **width specified**; if not, then the width will be set to auto, which means it fills the entire width of the containing block, and there thus will be no room available to flow content around the floated item.

- Floating elements example




```
<h1>Float example</h1>
<p>A wonderful serenity has taken ...</p>
<figure>
   <img src="images/828.jpg" alt="" />
   <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley …</p>

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


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

➢ Floating within a Container

- Float will help to move left or right of its container also called as container block

- In above figure container is HTML document itself, so figures moves left or right of browser window.

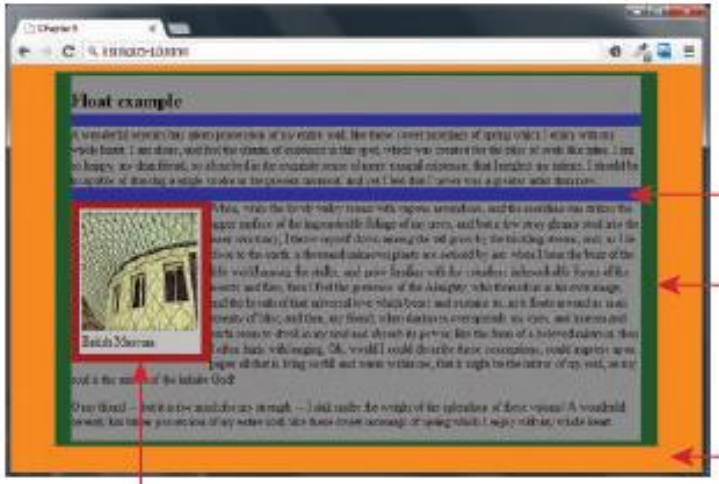
- In below example the floated figure is contained within an <article> element that is indented from the browser's edge.

- There is an important change happening which can be seen only by zooming the below figure.

- The overlapping margins for the adjacent <p> elements behave normally and collapse.

- But notice that the top margin for the floated <figure> and the bottom margin for the <p> element above it do not collapse.

Eg - Floating to the containing block




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

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

  <figure>
    <img src="images/828.jpg" alt="" />
    <figcaption>British Museum</figcaption>
  </figure>

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

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

➢ Floating Multiple Items Side by Side

- One of the most common usages of floats is to **place multiple items side by side** on the same line.

- When you float multiple items side by side, All other content in the containing block (including other floated elements) will flow around all the floated elements.

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

```
<article>
  <figure>
     <img src="images/tiny/275.jpg" alt="" />
     <figcaption>Westminister</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/700.jpg" alt="" />
     <figcaption>Emirates Stadium</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/537.jpg" alt="" />
     <figcaption>Albert Hall</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/828.jpg" alt="" />
     <figcaption>British Museum</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/464.jpg" alt="" />
     <figcaption>Wellington Monument</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/224.jpg" alt="" />
     <figcaption>Lewes Castle</figcaption>
  </figure>
  <p>When, while the lovely valley teems ..
</article>
```

- Problems with multiple floats

- As the window resizes, the content in the containing block (the <article> element), will try to fill the space that is available to the right of the foated elements.

- you can stop elements from flowing around a floated element by using the **clear property**.

```
.first { clear: left; }
```

```
<article>
  <figure>
     <img src="images/tiny/275.jpg" alt="" />
     <figcaption>Westminister</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/700.jpg" alt="" />
     <figcaption>Emirates Stadium</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/537.jpg" alt="" />
     <figcaption>Albert Hall</figcaption>
  </figure>
  <figure class="first">
     <img src="images/tiny/828.jpg" alt="" />
     <figcaption>British Museum</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/464.jpg" alt="" />
     <figcaption>Wellington Monument</figcaption>
  </figure>
  <figure>

     <img src="images/tiny/224.jpg" alt="" />
     <figcaption>Lewes Castle</figcaption>
  </figure>
  <p class="first">When, while the lovely valley ..
</article>
```

➢ 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 that case, the containing block essentially **disappears**.
- In the below figure, the <figure> containing block contains only an <img> and a figcaption element and both of these elements are floated to the left.
- That means both elements have been removed from the normal flow, from the browser's perspective, since the <figure> contains no normal flow content, it essentially has nothing in it, hence it has a content height of zero.
- Best solution for this problem is to use **overflow property.**

- Disappearing parent containers

```
<article>
 <figure>
   <img src="images/828.jpg" alt="" />
   <figcaption>British Museum</figcaption>
 </figure>
 <p class="first">When, while the lovely valley …
</article>
```

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

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

- Using the overflow property

Setting the overflow property to auto solves the problem.

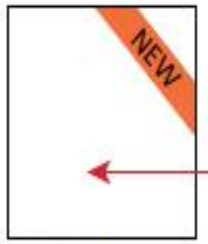
```
figure img {
   width: 170px;
   float: left;
   margin: 0 5px;
}
figure figcaption {
   width: 100px;
   float: left;
}
figure {
   border: 1pt solid #262626;
   background-color: #c1c1c1;
   padding: 5px;
   width: 400px;
   margin: 10px;
   overflow: auto;
}
```

➢ Overlaying and Hiding elements

- One of the most common design tasks with CSS is to place **two elements on top of each other or to selectively hide and display elements**. Positioning is important for both tasks.

```
<figure>
   <img src="images/828.jpg" alt="" />
   <figcaption>British Museum</figcaption>
   <img src="images/new-banner.png" alt="" class="overlayed"/>
</figure>
```


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



Transparent area

new-banner.png


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

This hides the overlayed image.

```
.hide {
   display: none;
}
```

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

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

There are two different ways to hide elements in css:

1. **Display property**

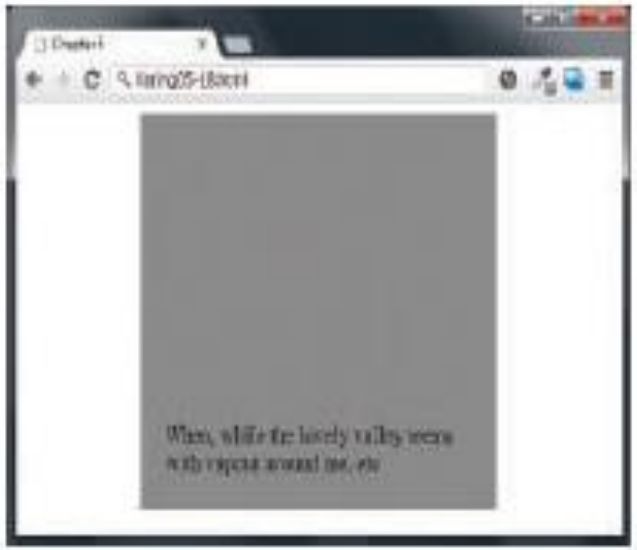
- It takes an item out of the flow – element no longer exists

2. Visibility property

- Hides the element, but the space for that element remains.

```
figure {
    ...
    display: auto;
}
```

```
figure {
    ...
    display: none;
}
```



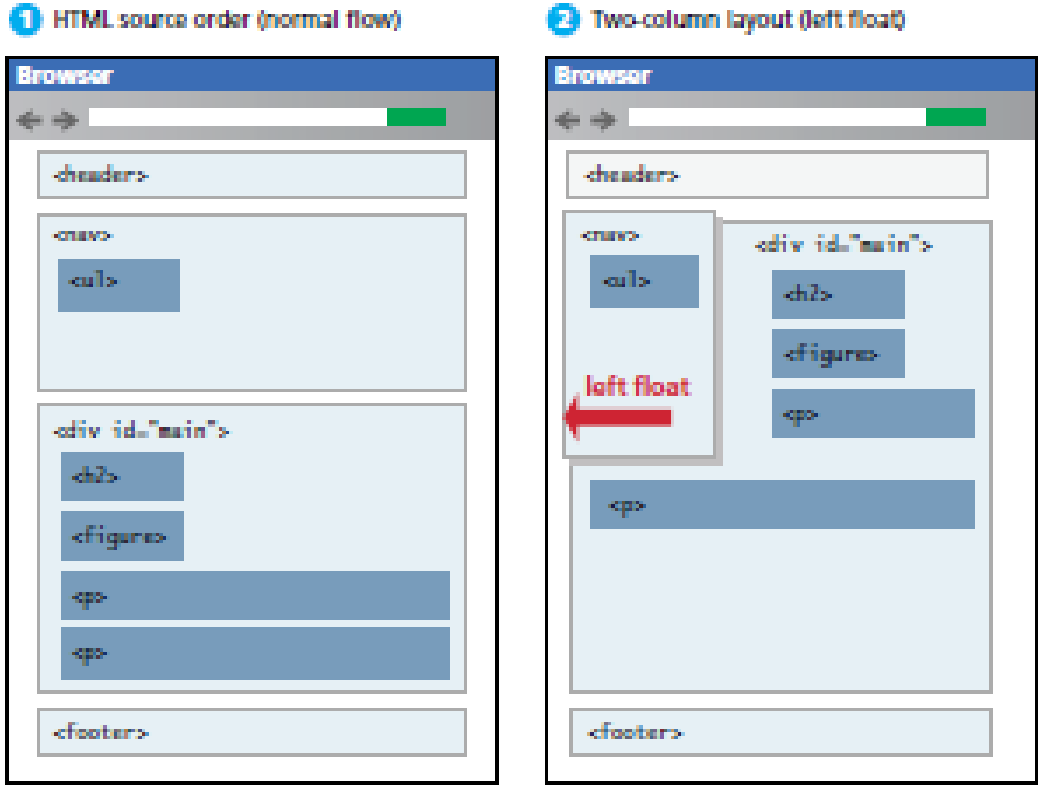
```
figure {
    ...
    visibility: hidden;
}
```

➢ Constructing Multicolumn Layouts

- The previous sections showed two different ways to move items out of the normal top-down flow, namely, by using positioning and by using floats.

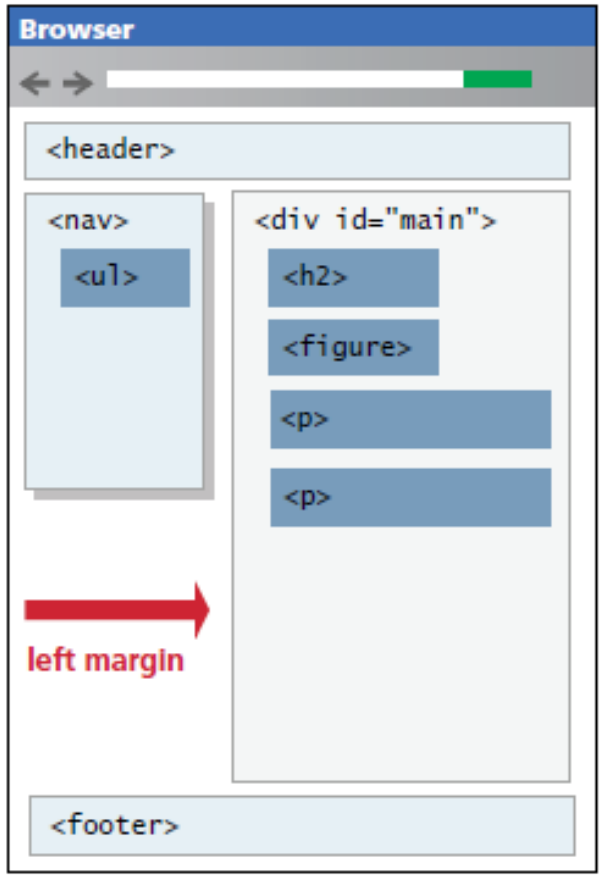
➢ Using Floats to Create Columns (Creating two-column layout)

- The first step is to float the content container that will be on the left-hand side.



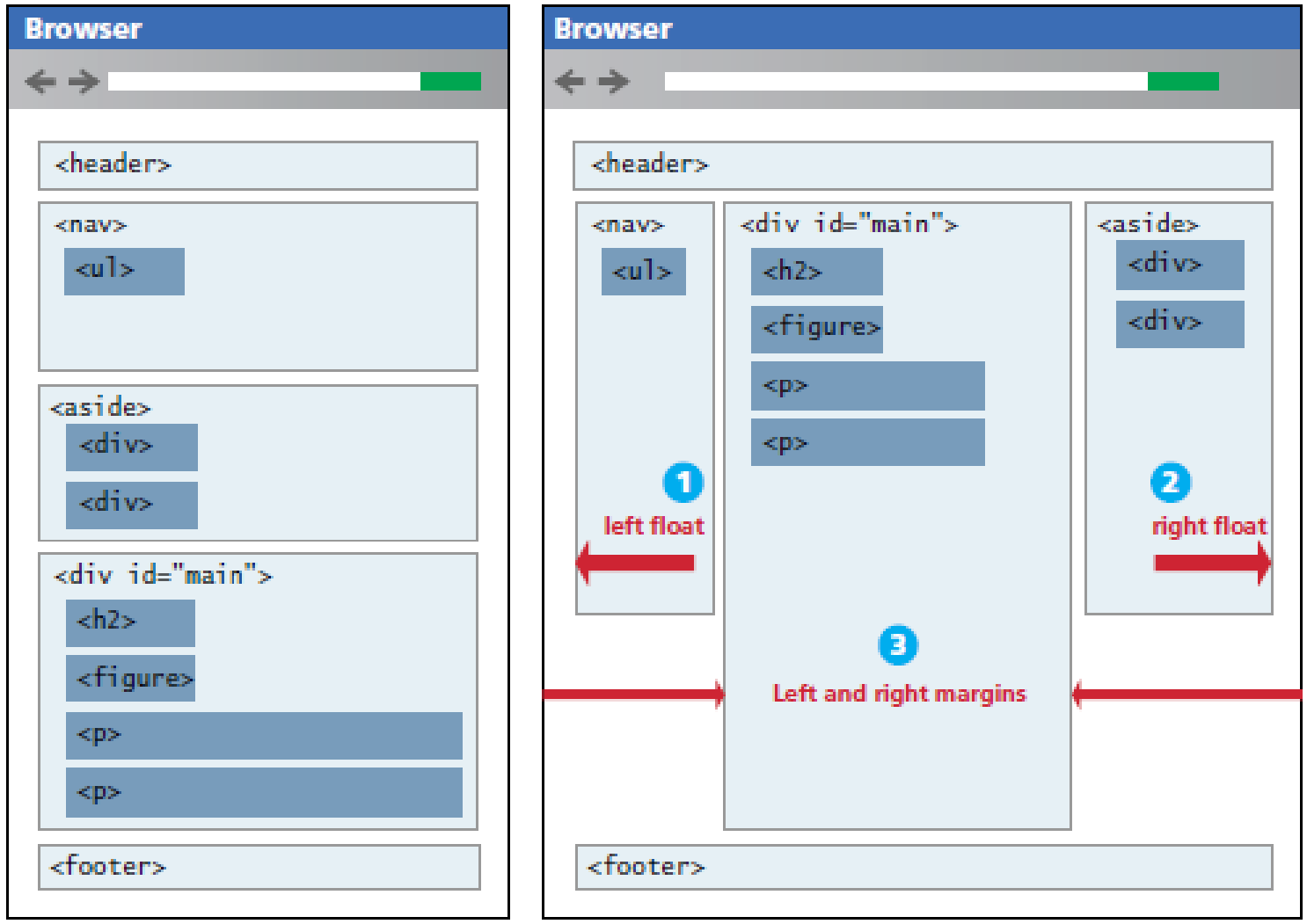

```
nav {
  …
  width: 12em;
  float: left;
}
```

3 Set the left margin of non-floated content




```
div#main {
  …
  margin-left: 220px;
}
```

- Fig : Creating a three-column layout

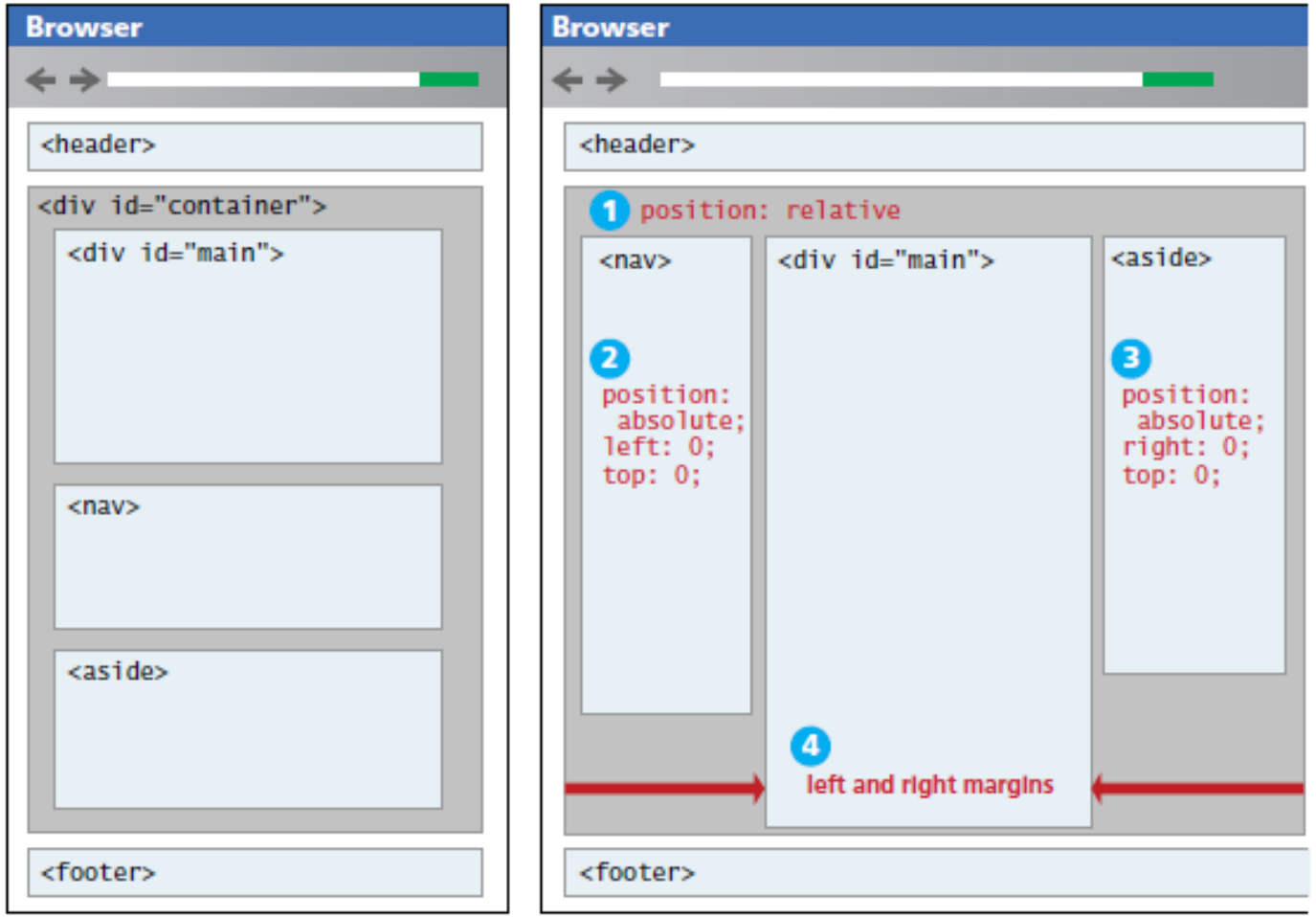


➢ Using Positioning to Create Columns

- Positioning can also be used to create a multicolumn layout. Typically, the approach will be to absolute position the elements.

(absolute positioning is related to the nearest positioned ancestor)

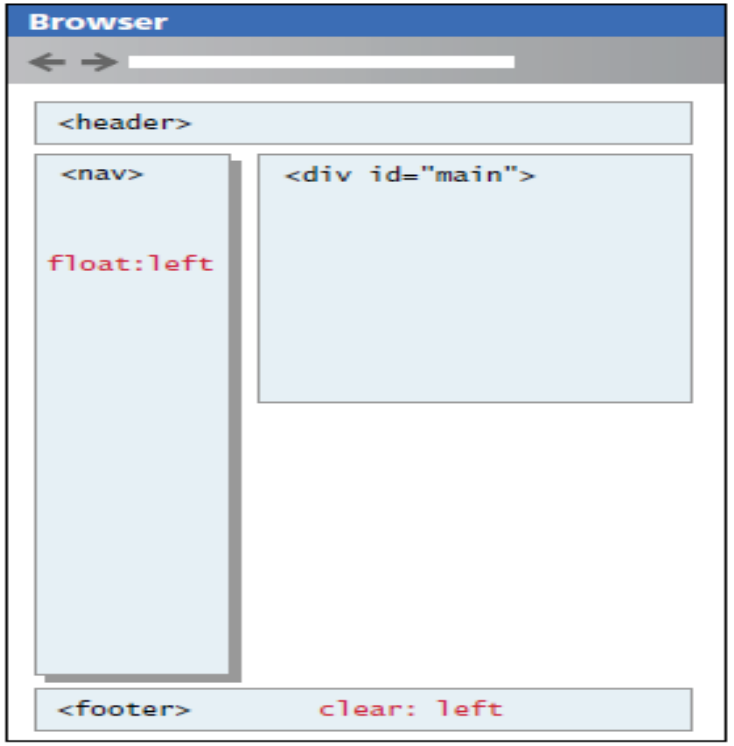
- Three-column layout with positioning



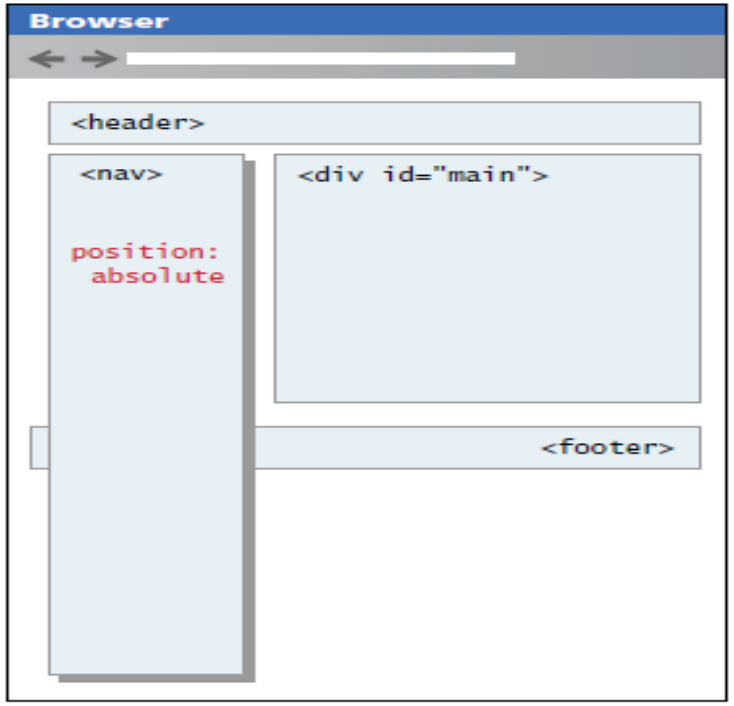
However, absolute positioning has its own problems. What would happen if one of the sidebars had a lot of content and was thus quite long?

- In the floated layout, this would not be a problem at all, because when an item is floated, blank space is left behind.

- But when an item is positioned, it is removed entirely from normal flow, so subsequent items will have no “knowledge” of the positioned item.



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



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

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

- Creating layouts with floats and positioning has certain strengths and weaknesses.

While there are other ways to construct multicolumn layouts with CSS.

➢ Approaches to CSS Layout

- One of the main problems faced by web designers is the size of the screen used to view the page can vary.

- Some users will visit a site on a 21-inch wide screen monitor that can display 1920 × 1080 pixels.

- 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.

- Most designers take one of two basic approaches to dealing with the problems of screen size.

1. Fixed Layout

2. Liquid Layout

3. Hybrid Layout

➢ Fixed Layout

- In a **fixed layout**, the basic width of the design is set by the designer, typically corresponding to an “ideal” width based on a “typical” monitor resolution.

- A common width used is something in the 960 to 1000 pixel range, which fits nicely in the common desktop monitor resolution (1024 × 768).

- This content can then be positioned on the left or the center of the monitor.

- Fixed layouts are created using **pixel units**, typically with the entire content within a container (often named "container", "main", or "wrapper") whose width property has been set to some width.

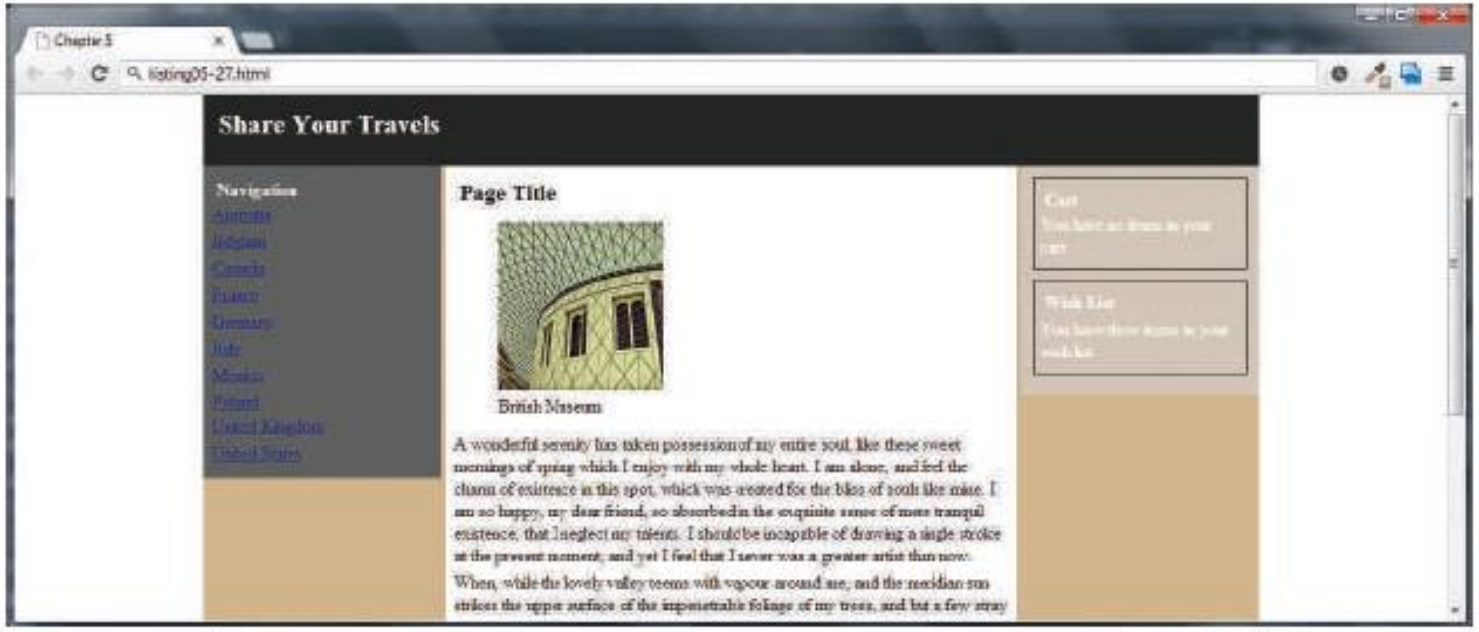
- The **advantage** of a fixed layout is that it is easier to produce and generally has a predictable visual result.

- It is also optimized for typical desktop monitors.

- **Fixed layouts have drawbacks.**

- For larger screens, there may be an excessive amount of blank space to the left and/or right of the content.

- when the browser window shrinks below the fixed width; the user will have to horizontally scroll to see all the content.




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

➢ 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.

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

- There are several **disadvantages**, Liquid layouts can be more difficult to create because some elements, such as images, have fixed pixel sizes.

- The line length (which is an important contributing factor to readability) may become too long or too short

Fluid layouts are based on the browser window.

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

➢ Hybrid layout

- 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.

- A **hybrid layout** combines pixel and percentage measurements.

- Fixed pixel measurements might make sense for a sidebar column containing mainly graphic advertising images that must always be displayed and which always are the same width.

- Percentages would make more sense for the main content or navigation areas.

➢ Responsive Design

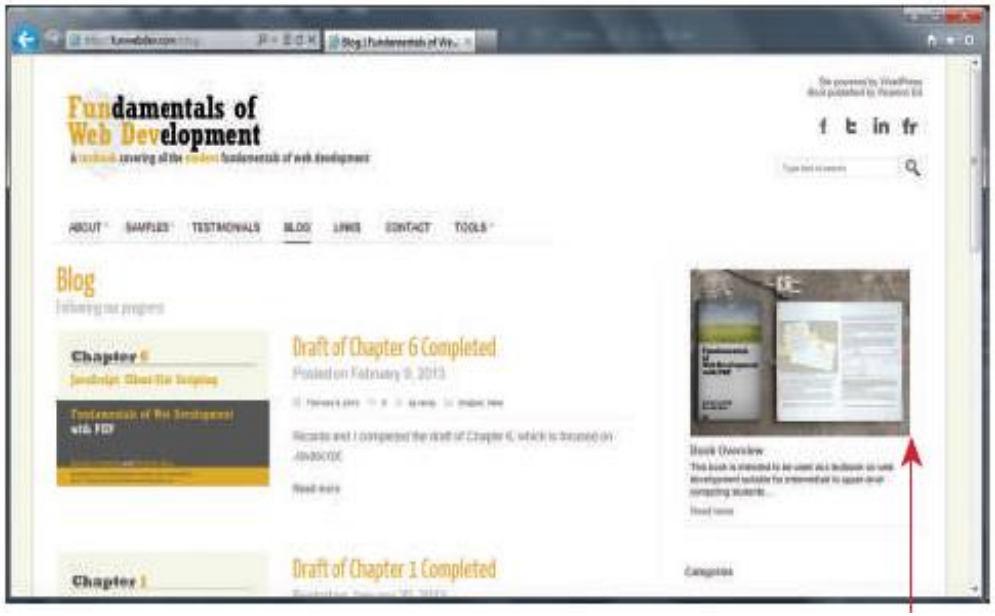
- In the past several years, a lot of attention has been given to so-called responsive layout designs.

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

- We had problems with liquid layout for images.

- In a responsive design layout, **images will be scaled down** and **navigation elements will be replaced as the browser shrinks**.

- Responsive layouts



Notice how some elements are scaled to shrink as browser window reduces in size.

When browser shrinks below a certain threshold, then layout and navigation elements change as well.

In this case, the `<ul>` list of hyperlinks changes to a `<select>` and the two-column design changes to one column.

There are 4 important key components to make responsive design work

1. Liquid layouts.
2. Scaling images to 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.

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

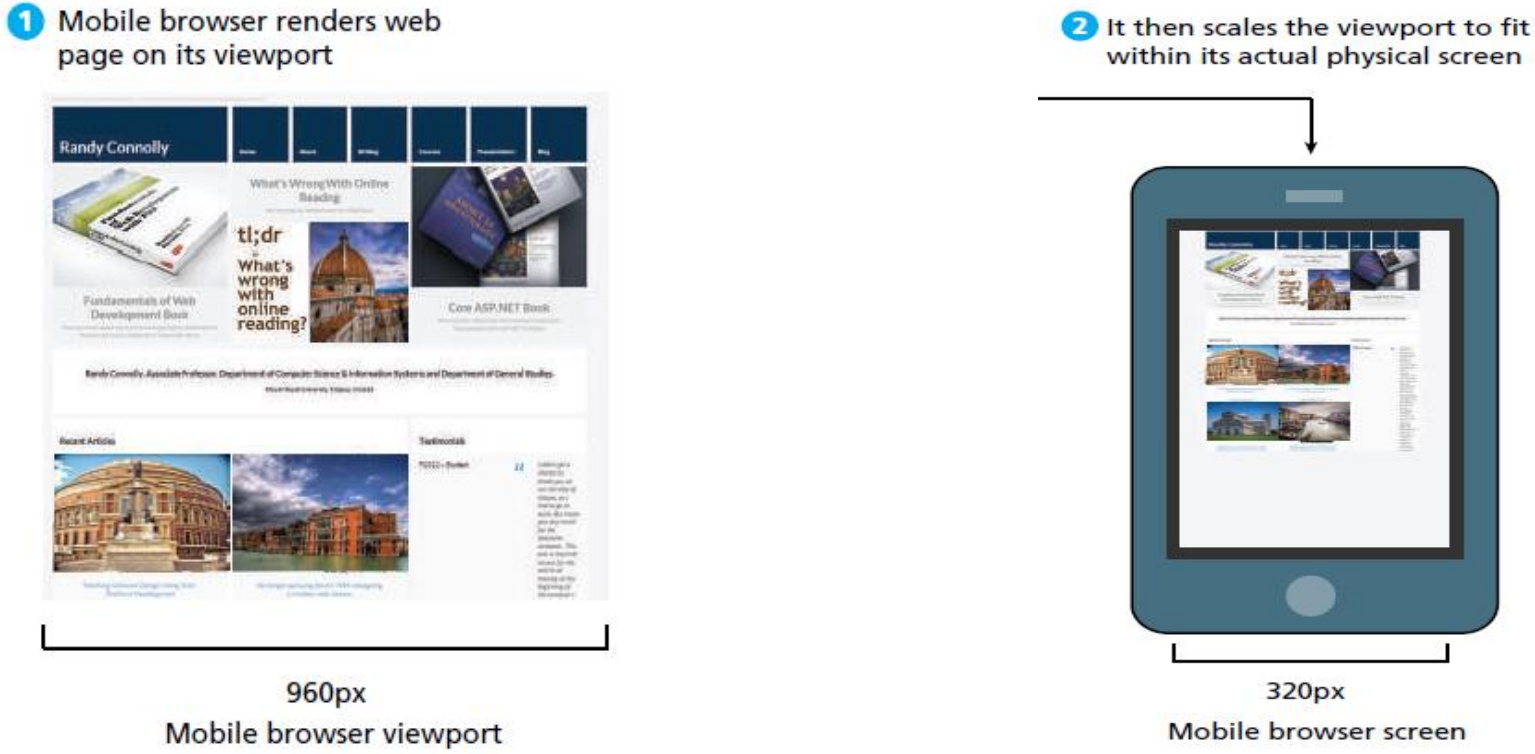
- But it does not change the downloaded size of the image; it only shrinks or expands its visual display to fit the size of the browser window.

➢ Setting Viewports

- If you have ever used a modern mobile browser, you may have been surprised to see how the web page was scaled to fit into the small screen of the browser.

- The way this works is the mobile browser renders the page on a canvas called the **viewport.**

- On iPhones, for instance, the viewport width is 980 px, and then that viewport is scaled to fit the current width of any device.



- <meta> tag is used by developers to **control the size of the initial viewport.**

- The web page can tell the mobile browser the viewport size to use via the **viewport <meta> element.**

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

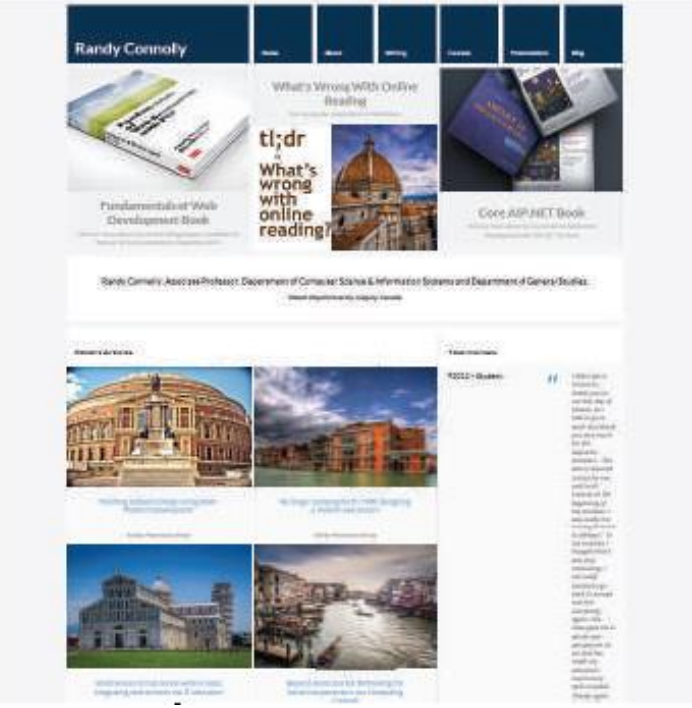
- Setting the viewport

- By setting the viewport, the page is telling the browser there is no need of any scaling and it makes the viewport as many pixels wide as device screen width.

- Ex: if device has a screen that is 320 px wide, then the viewport width will be 320 px,

- if its 480px wide, then viewport will be 480.

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



Mobile browser renders web page on its viewport and because of the <meta> setting, makes the viewport the same size as the pixel size of screen.

- However, since *only* setting the viewport as shown above, shrank but still cropped the content, setting the viewport is only one step in creating a responsive design.
- There needs to be 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

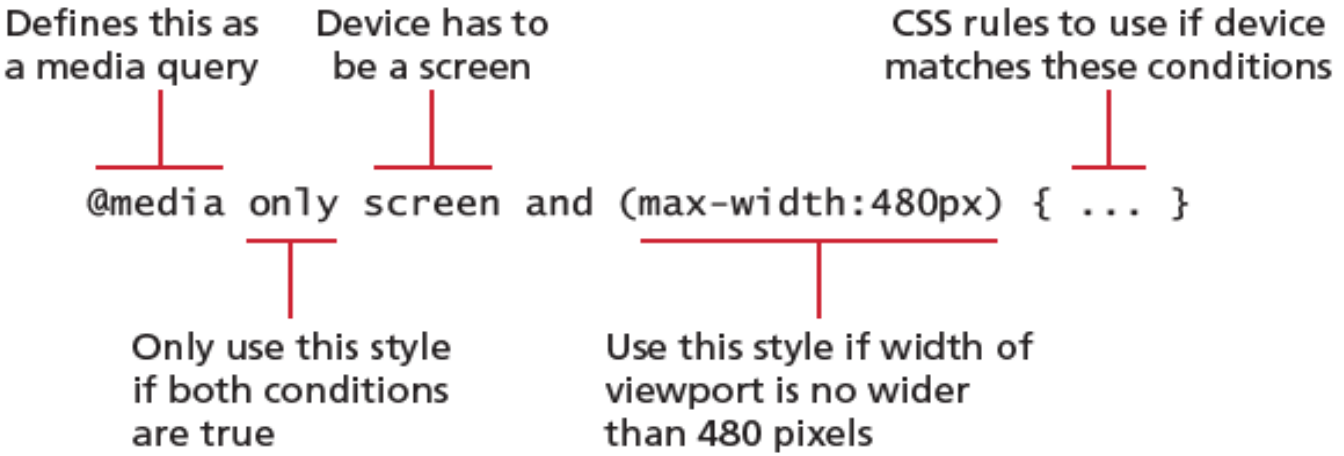
- The other key component of responsive designs is **CSS media queries**.

- A **media query** is a way to apply style rules **based on the medium** that is displaying the file.

- You can use these queries to look at the capabilities of the device, and then define CSS rules to target that device.

- Media queries are not supported by Internet Explorer 8 and earlier.

- syntax of a media query

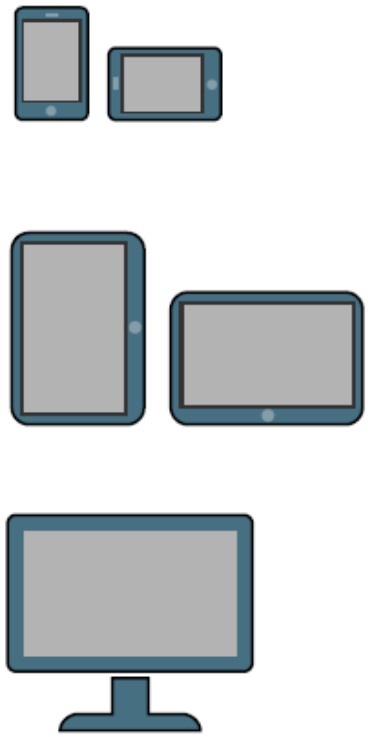


- These queries are Boolean expressions and can be added to your CSS files or to the <link> element to conditionally use a different external CSS file based on the capabilities of the device.

- Below table shows the list of features you can use with media queries.

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

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



styles.css

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


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



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

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

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

Handles Internet Explorer 8 and earlier using IE conditional comments.

➢ CSS Frameworks

- CSS layouts are quite complicated and difficult to create, many have struggled to create complex layouts with CSS.

- Larger web development companies often have several dedicated CSS experts who handle this part of the web development workflow.

- A **CSS framework** is a precreated set of CSS classes or other software tools that make it easier to use and work with CSS.

They are two main types of CSS framework:

1. Grid systems

2. CSS pre-processors.

➢ 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**).

- CSS frameworks provide similar grid features.
- **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.

- Below examples illustrates a three column layout within the grid system of the 960 framework, and the same thing in the Bootstrap framework.

```
<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>
```

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>
```

Using the Bootstrap grid

- In both systems, elements are laid out in rows; elements in a row will span from 1 to 12 columns.

- In 960 system, a row is terminated with .

- In Bootstrap, content must be placed within the row container.

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

- These frameworks are also responsive as well.

- Bootstrap framework is widely used grid system, which is an open-source system, was originally created by the designers at Twitter.

- It has a wide variety of very useful **additional styling classes**, such as **classes for drop-down menus**, **fancy buttons** and **form elements**, and **integration with a variety of jQuery plug-ins.**

➢ 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.

- Example in a site many items will have the same color.
- For instance, consider the below example, the background color of the .box class, the text color in the <footer> element, the border color of the <fieldset>, and the text color for placeholder text within the <textarea> element, might all be set to #796d6d.

- The trouble with regular CSS is that when a change needs to be made (perhaps the client likes #e8cfcf more than #796d6d), then some type of **copy and replace is necessary**. The change might be made to the wrong elements.

- Similarly, for different site elements to have similar CSS formatting, for instance, different boxes to have the same padding. Again, in normal CSS, one has to use **copy and paste to create that uniformity.**

- In a programming language, a developer can use variables, nesting, functions, or inheritance to handle duplication and avoid copy-and-pasting and search-and-replacing.

- CSS preprocessors such as **LESS, SASS, and Stylus** provide this type of functionality.

- Below example illustrates how a CSS preprocessor (in this case SASS) is used to handle some of the just-mentioned duplication and change problems.

- SASS source file, e.g. source.scss

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


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


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

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

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

SASS source file, e.g. source.scss

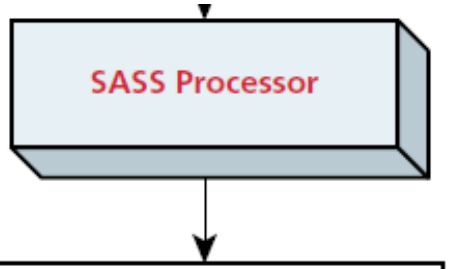
This example uses SASS (Syntactically Awesome Stylesheets). Here three variables are defined.

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

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

A mixin can be referenced/called and passed parameters.

- Generated CSS file, e.g., styles.css from Sass file using CSS preprocessor (SASS)



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

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

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

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

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