Advanced CSS: Layout

Chapter 5

Objectives

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

Normal Flow

Section 1 of 7

Normal Flow

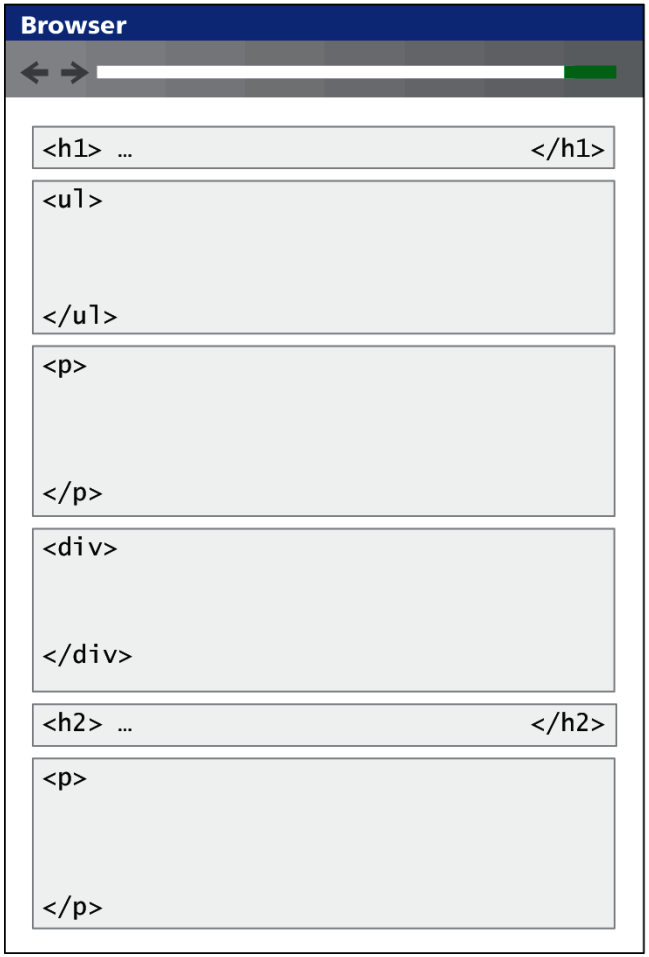
If it's so normal why have I never heard of it?

Normal flow refers here to how the browser will normally display block-level elements and inline elements from left to right and fro m top to bottom

- **Block-level elements** are each contained on their own line
  - <p>, , <h2>, <ul>, and <table>
- **Inline elements** do not form their own blocks but instead are di splayed within lines
  - as , <a>, <img>, and

Normal Flow

Example



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

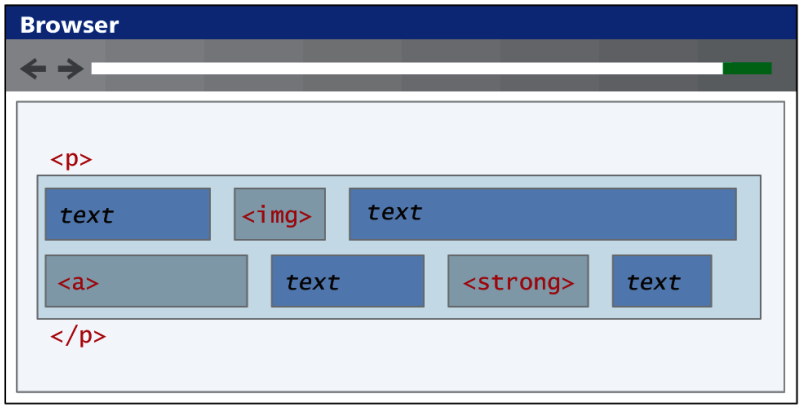
By default each block-level element fills up the entire width of its parent (in this case, it is the <body>, which is equivalent to the width of the browser window).

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

Normal Flow

Example

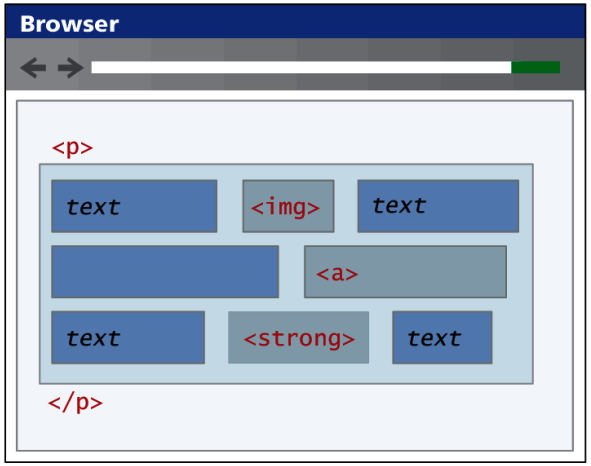
```
<p>
This photo <img src="photo-con.png" alt="" /> of Conservatory Pond in
<a href="http://www.centralpark.com/">Central Park</a> New York City
was taken on October 22, 2015 with a <strong>Canon EOS 30D</strong>
camera.
</p>
```



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 "reflowed" based on the new width.

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

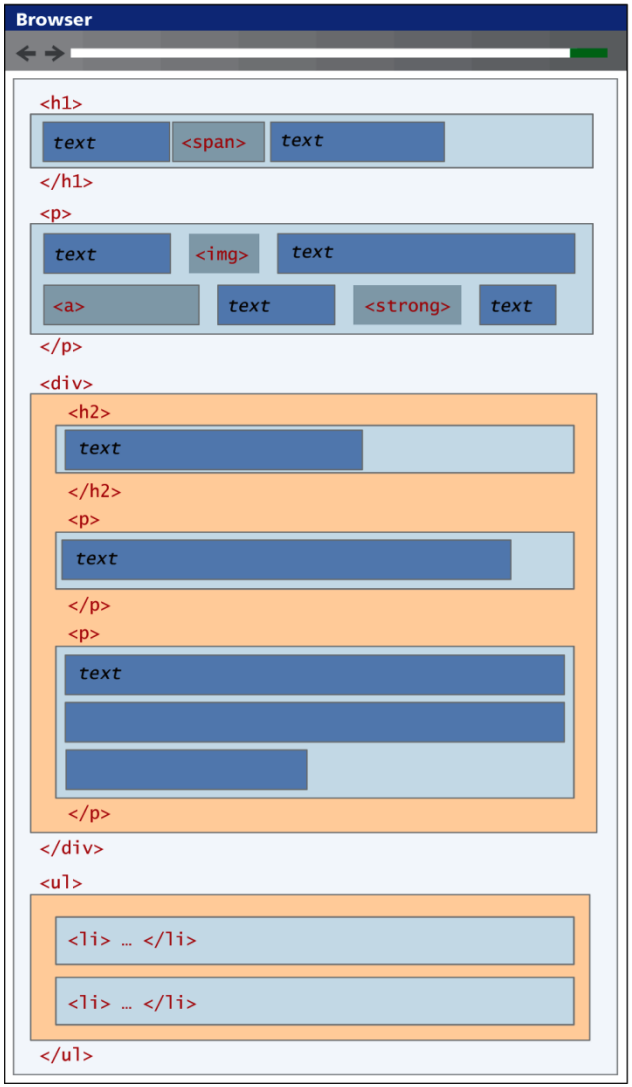
Inline Elements

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

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

- **Nonreplaced inline elements** are those elements whose c ontent is defined within the document, which includes all t he other inline elements.

Working Together



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

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

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

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

Take control

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).

Positioning Elements

Section 2 of 7

Positioning Elements

Move it a little left…

It is possible to move an item from its regular position in the normal flow, and

- move an item outside of the browser viewport so it is not visible
- position it so it is always visible in a fixed position while the rest o f the content scrolls.

The **position** property is used to specify the type of positioning

Positioning

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

Relative Positioning

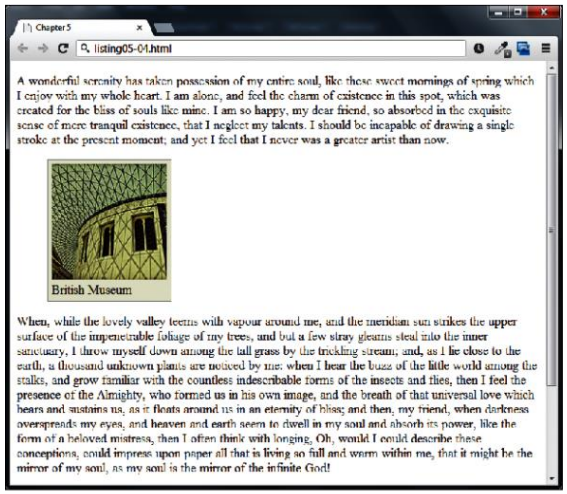
Relative to what

In **relative positioning** an element is displaced out of its normal flow position and moved relative to where it would have been pl aced

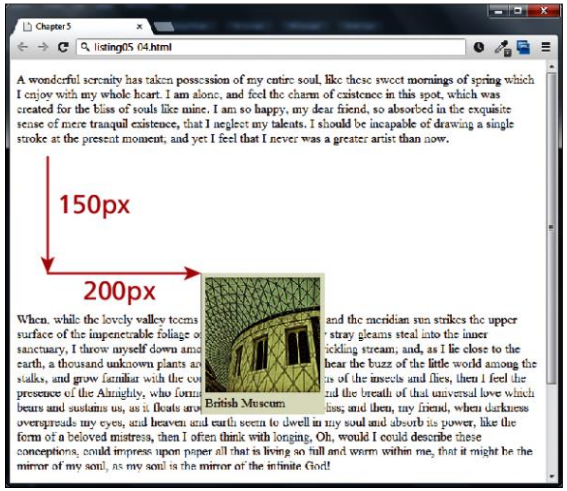
The other content around the relatively positioned element “re members” the element’s old position in the flow; thus the space the element would have occupied is preserved

Relative Positioning

Relatively simple Example




```
<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;
}
```

Absolute Positioning

Absolutely!

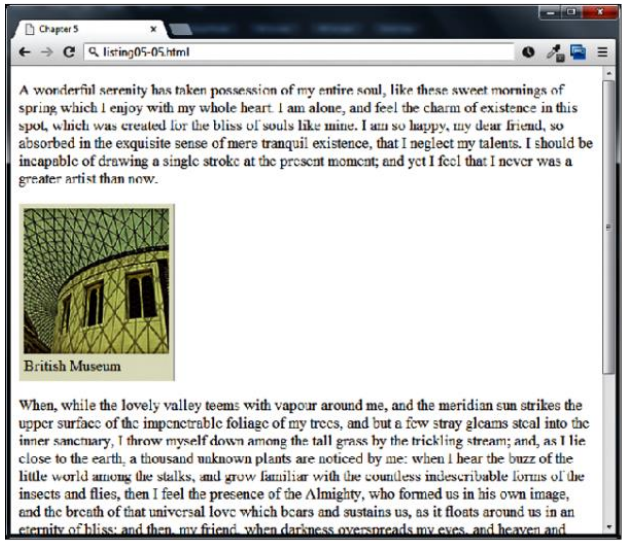
When an element is positioned absolutely, it is removed completely from normal flow.

Unlike with relative positioning, space is not left for the moved element, as it is no longer in the normal flow.

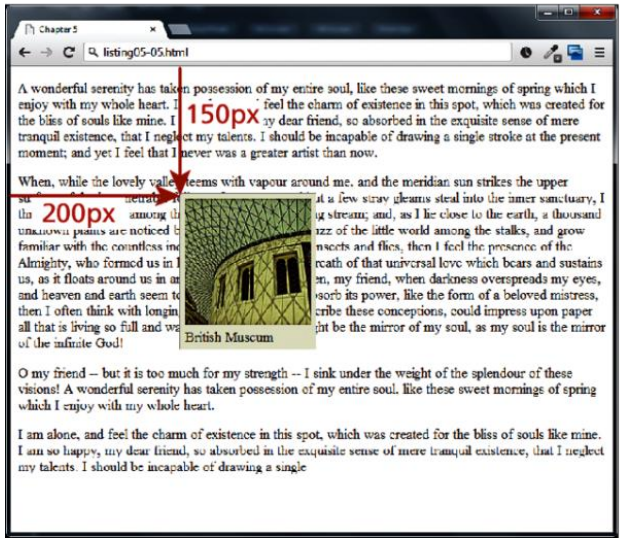
absolute positioning can get confusing

Absolute Positioning

Absolutely great example




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

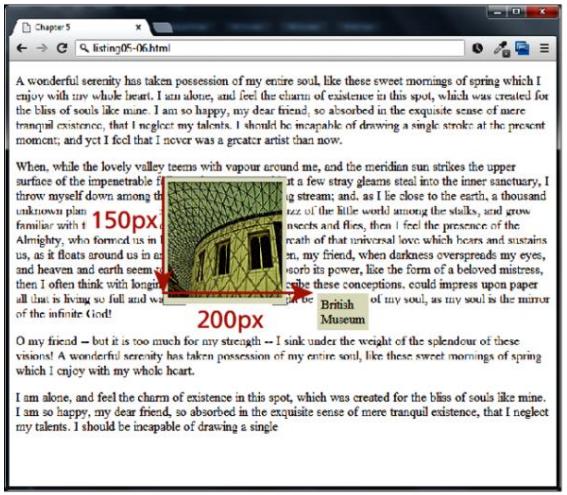
Absolute Positioning

Absolutely relative?

A moved element via absolute position is actually positioned relative to its nearest **positioned** ancestor container

Absolute Positioning


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

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

Z-Index

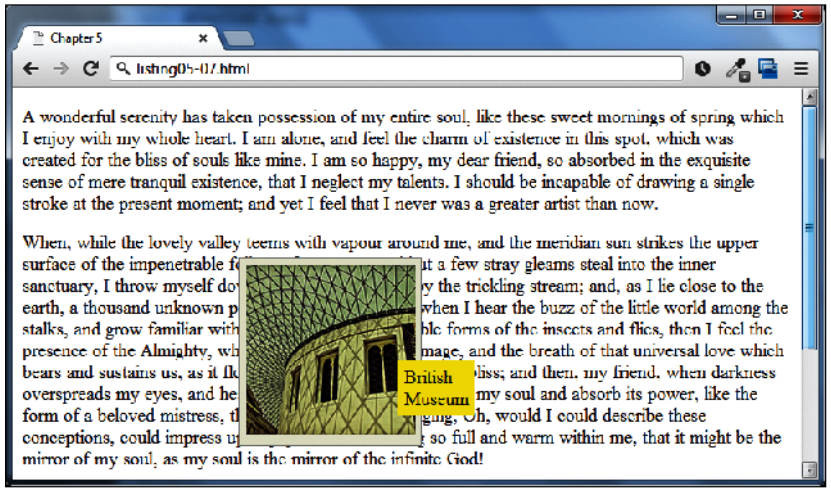
So, like the last topic?

When overlapping elements items closest to the viewer (and thus on the top) have a larger **z-index**

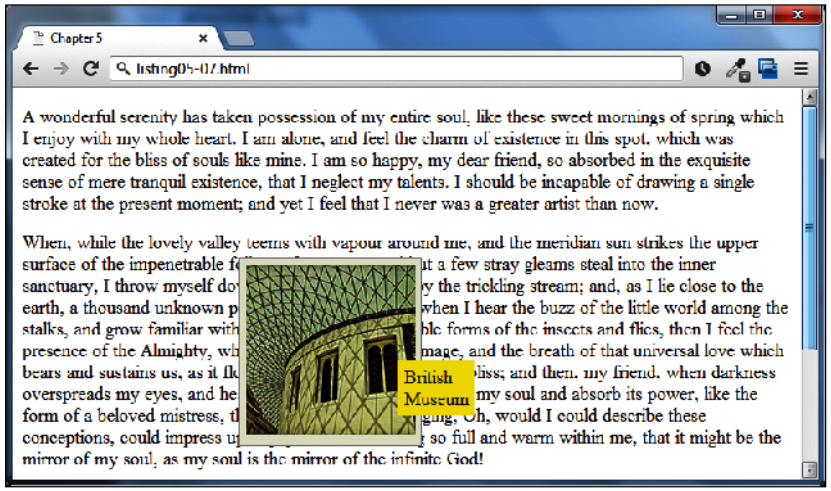
- only positioned elements will make use of their z-index
- simply setting the z-index value of elements will not necessarily move them on top or behind other items.

Z-index

Example




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

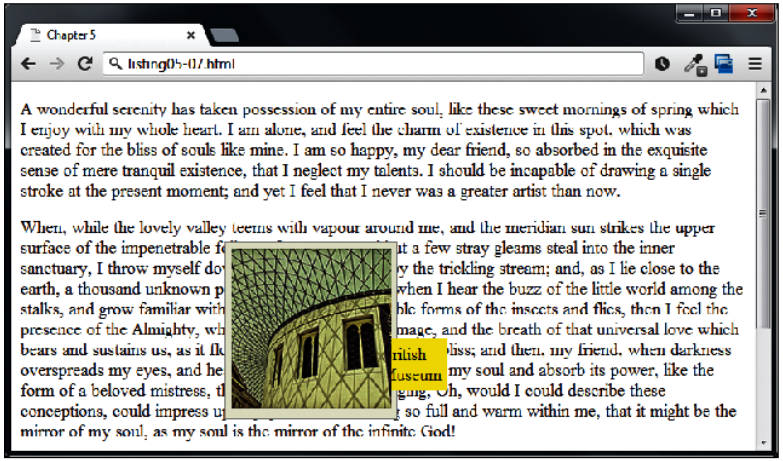



```
figure {
    …
    z-index: 5;
}
figcaption {
    …
    z-index: 1;
}
```

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.

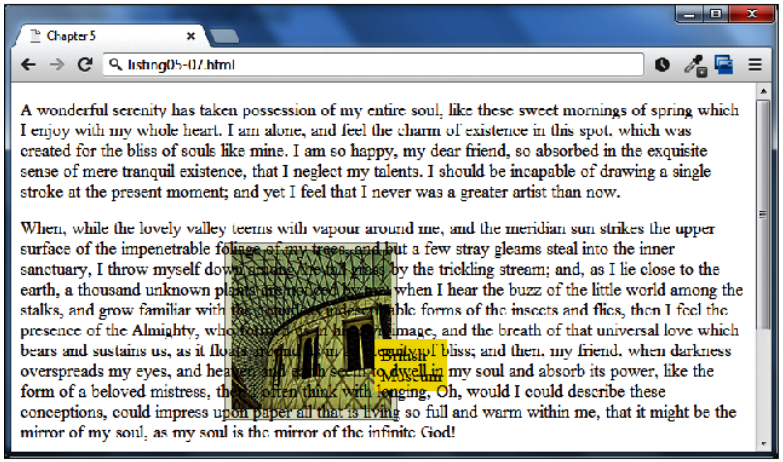
Z-index

Example




```
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.

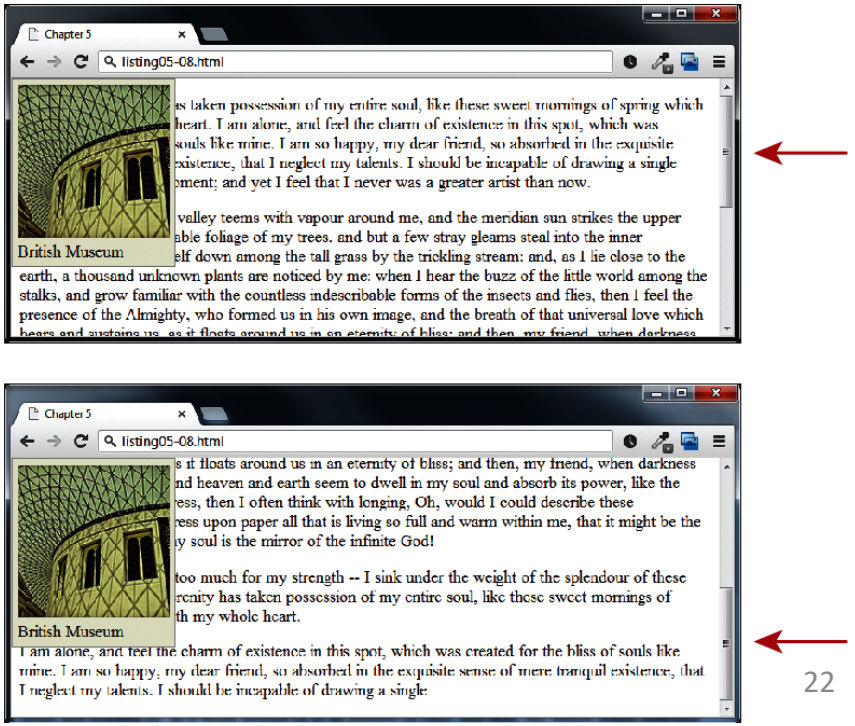
Fixed Position

Place your annoying ads here

Elements with **fixed positioning** do not move when the user scrolls up or down the page

```
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

Section 3 of 7

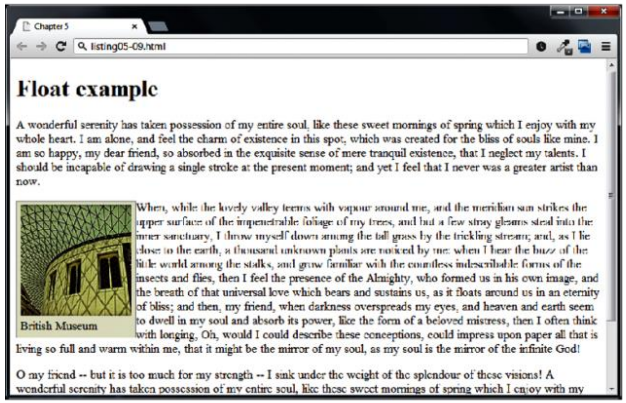
Floating Elements

- 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

Floating Elements

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.

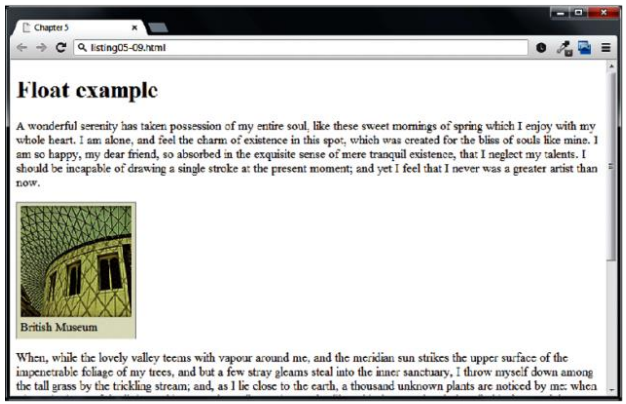
Floating Elements



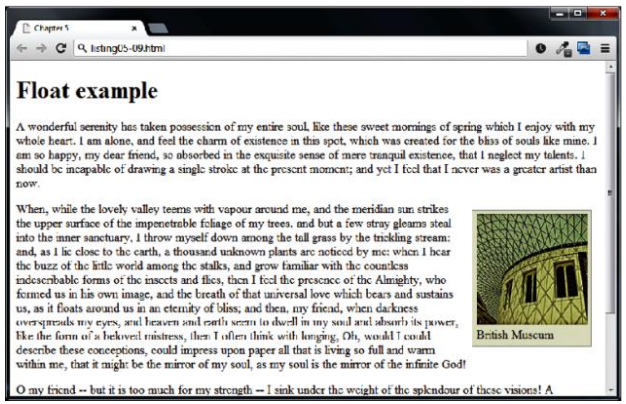

```
<h1>Float example</h1>
<p>A wonderful serenity has taken ...</p>
<figure>
   <img src="images/828.jpg" alt="" />
   <figcaption>British Museum</figcaption>
</figure>
<p>When, while the lovely valley …</p>
```

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

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




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




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

Floating Elements

Details

Notice that a floated block-level element must have a width specified, if you do not, then the width will be set to auto, which will mean it im plicitly fills the entire width of the containing block, and there thus wil l 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**).

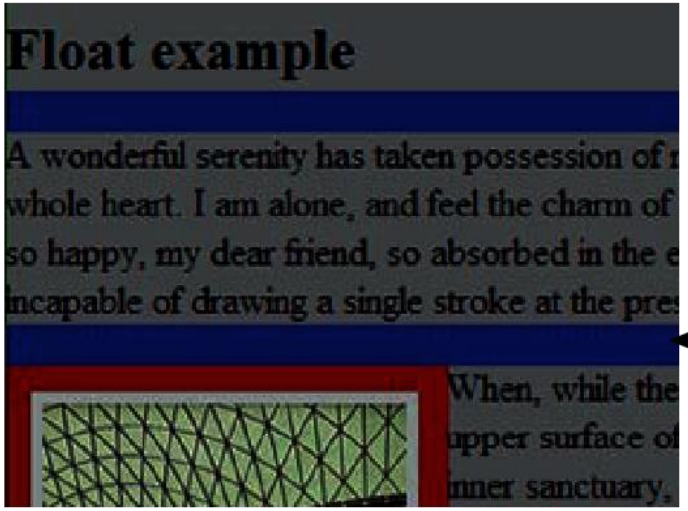
Floating within a container

```
<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>
```


```
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;
}
```

Floating within a container

Zoom



<p> margin-bottom: 16px;

Notice these margins collapse (normal behavior).

<p> margin-top: 16px;

But these margins did **not** collapse.

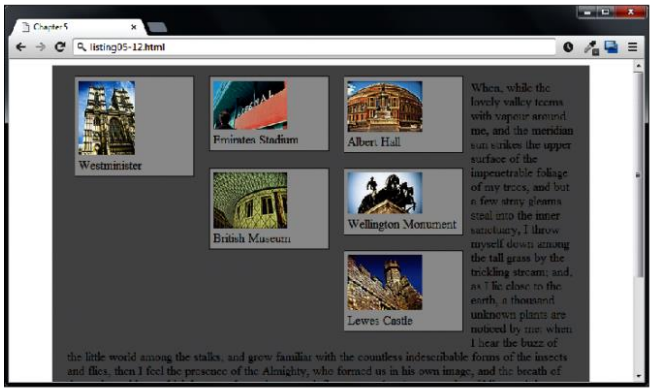
<figure> margin-top: 10px;

Floating Multiple Items side by side

When you float multiple items that are in proximity, each floated item in the container will be nestled up beside the previously floa ted item

All other content in the containing block (including other floated elements) will flow around all the floated elements

Floating side by side



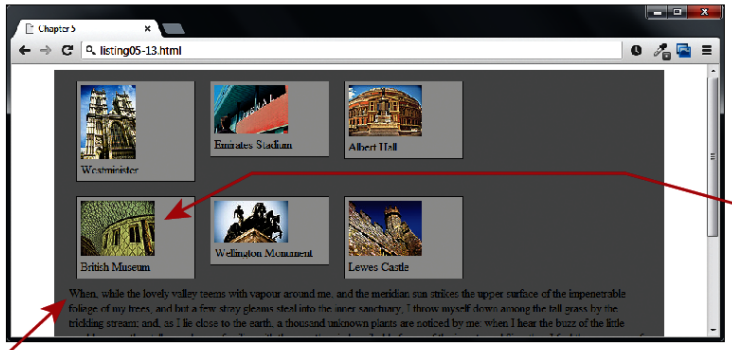

```
<article>
  <figure>
     <img src="images/tiny/275.jpg" alt="" />
     <figcaption>Westminister</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/700.jpg" alt="" />
     <figcaption>Emirates Stadium</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/537.jpg" alt="" />
     <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>

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

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 floated elements.

The Clear Property

You can stop elements from flowing around a floated element by using the clear **property**

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



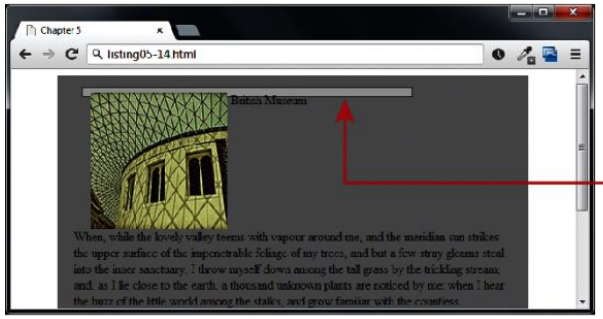

```
<article>
  <figure>
     <img src="images/tiny/275.jpg" alt="" />
     <figcaption>Westminister</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/700.jpg" alt="" />
     <figcaption>Emirates Stadium</figcaption>
  </figure>
  <figure>
     <img src="images/tiny/537.jpg" alt="" />
     <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

Problem

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

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



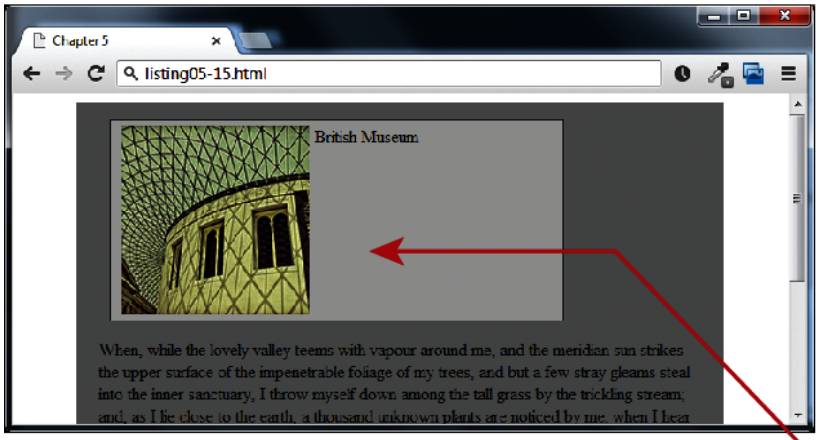
Notice that the <figure> element's content area has shrunk down to zero (it now just has padding space and borders).

```
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; }
```

Containing Floats

Solution

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



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

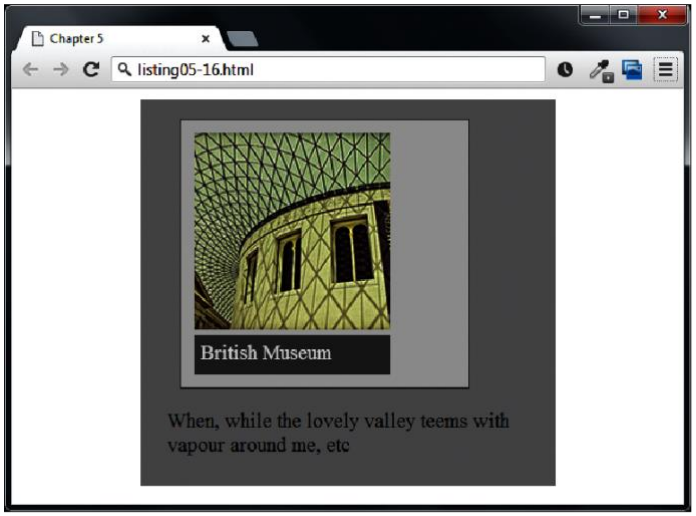
One of the more common design tasks with CSS is to place two elements on top of each other, or to selectively hide and display elements. Positioning is important to both of these tasks.

Positioning is often used for smaller design changes, such as mo ving items relative to other elements within a container

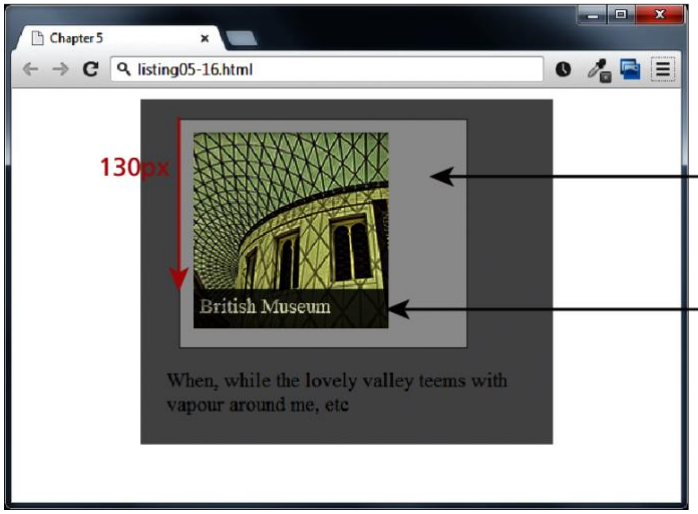
In such a case, relative positioning is used to create the **position ing context** for a subsequent absolute positioning move

Positioning Context

A Helpful thing to learn




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

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




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

This creates the positioning context.

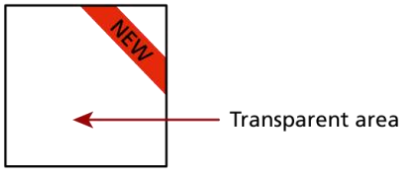
This does the actual move.

Positioning Context

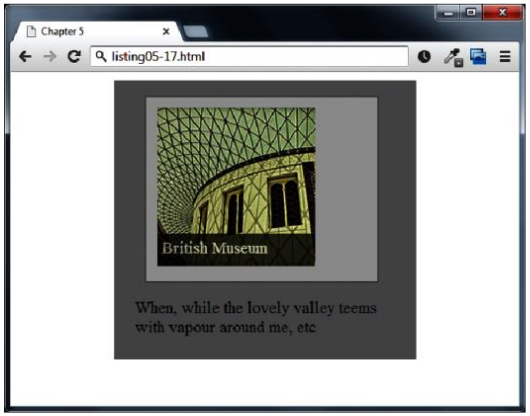
A realistic example

```
<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;
}
```



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"/>
```

Hiding elements

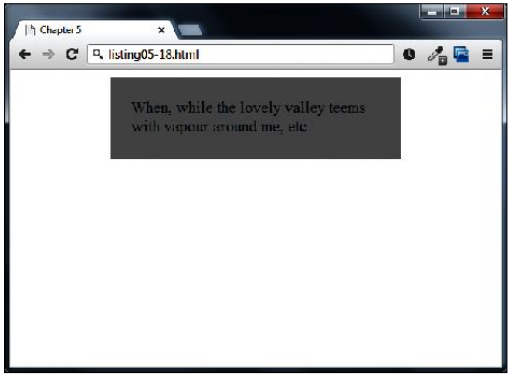
Two different ways to hide elements in CSS:

1. using the display property
    - The display property takes an item out of the flow: it is as if the ele ment no longer exists
2. using the visibility property
    - The visibility property just hides the element, but the space for that element remains.

Hiding elements

Two ways to hide: display and visibility


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




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

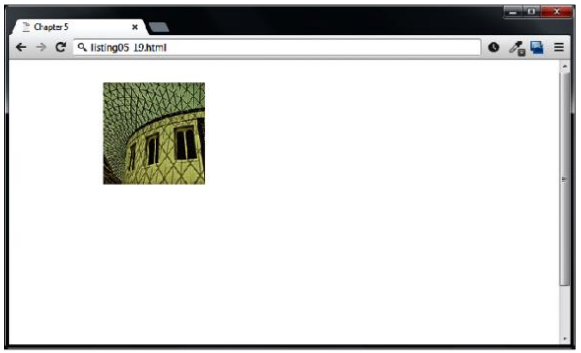

```
figure {
    ...
    visibility: hidden;
}
```

Using :hover to make thumbnails

Next slide shows how the combination of absolute positioning, the :hover pseudo-class, and the visibility property can be used to displa y a larger version of an image

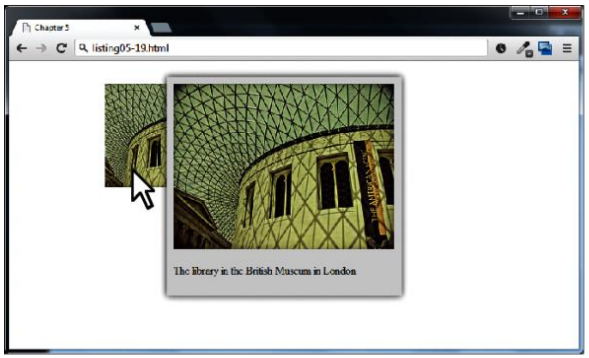
Using :hover to make thumbnails

```
<figure class="thumbnail">
  <img src="images/828.jpg" alt="" />
  <figcaption class="popup">
     <img src="images/828-bigger.jpg" alt="" />
     <p>The library in the British Museum in London</p>
  </figcaption>
</figure>
```



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

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

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

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



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

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

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

}
```

Constructing Multicolumn Layouts

Section 4 of 7

Constructing Multicolumn Layouts

There is unfortunately no simple and easy way to create robust multicolumn page layouts. There are tradeoffs with each approa ch:

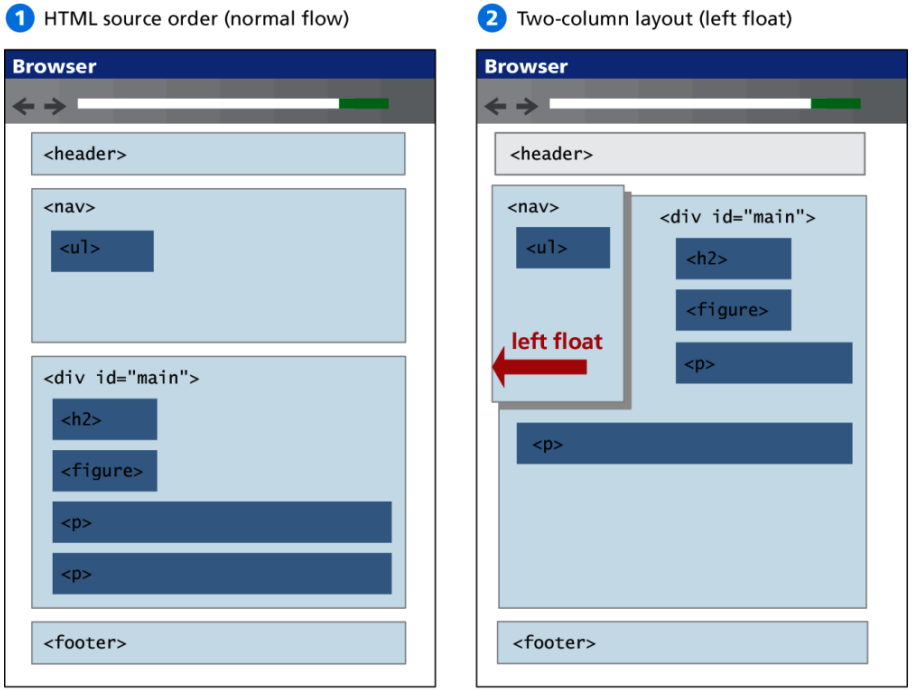
- Using Floats to Create Columns
- Using Positioning to Create Columns

Using Floats to create Columns

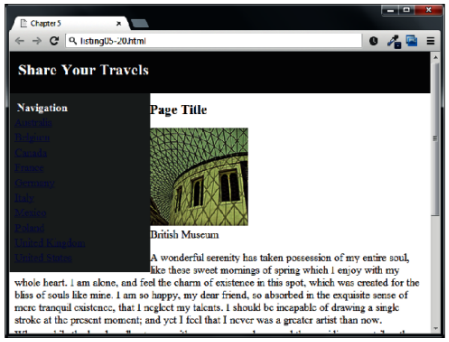
The most common way to create columns of content is using floats.

- The first step is to float the content container that will be on the l eft-hand side. Remember that the floated container needs to hav e a width specified.

Using Floats to create Columns




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

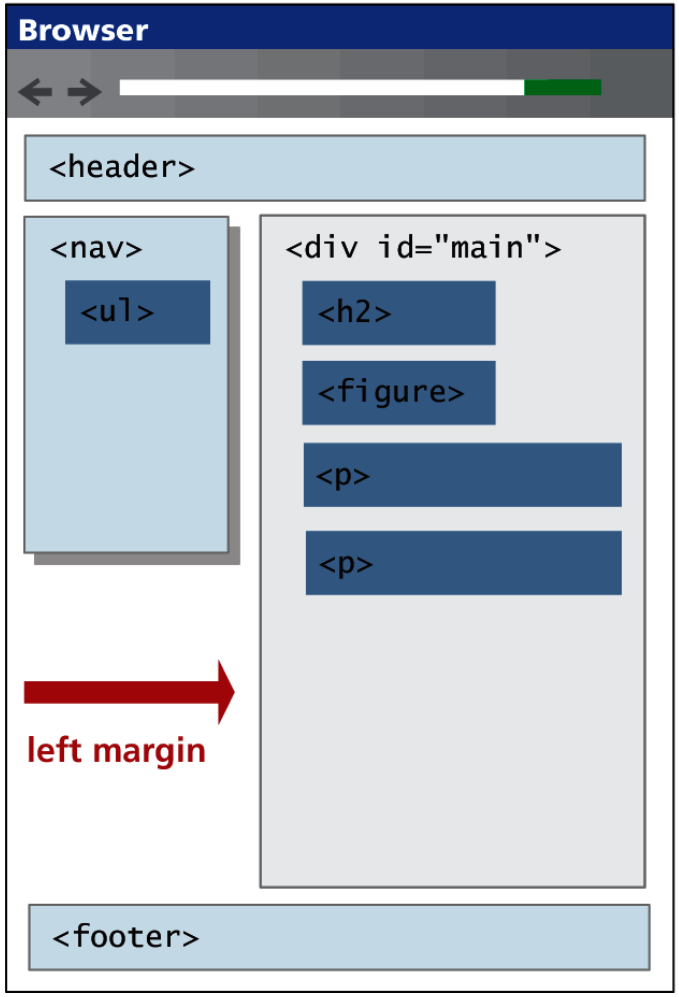


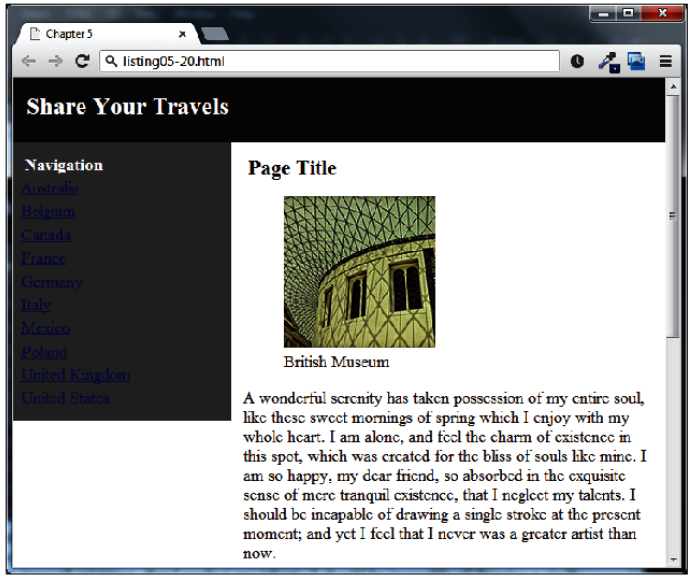
Using Floats to create Columns

The other key step is changing the left-margin so that it no longer flows back under the floated content.

Using Floats to create Columns

3 Set the left margin of non-floated content

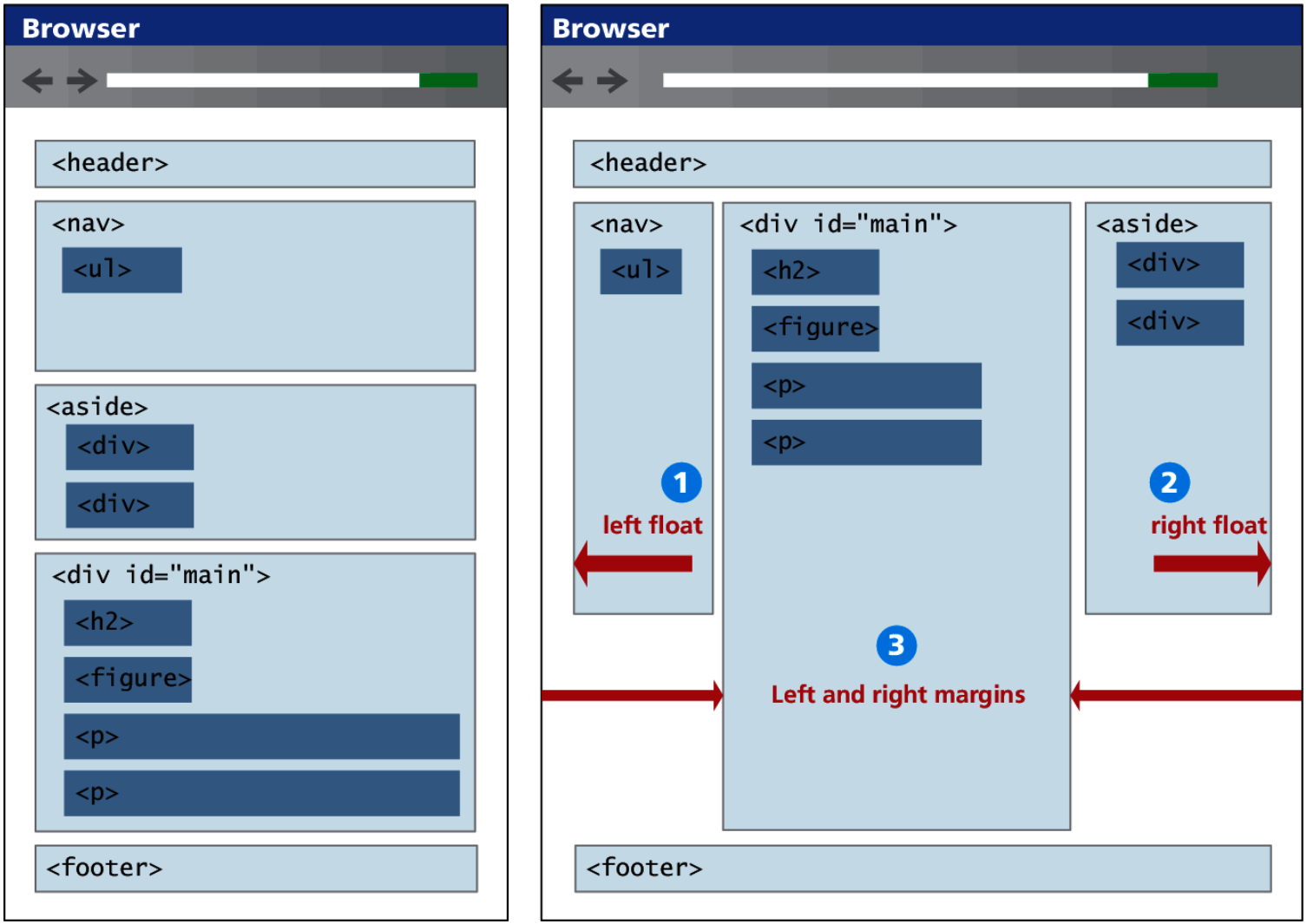





```
div#main {
  …
  margin-left: 220px;
}
```

Using Floats for Columns

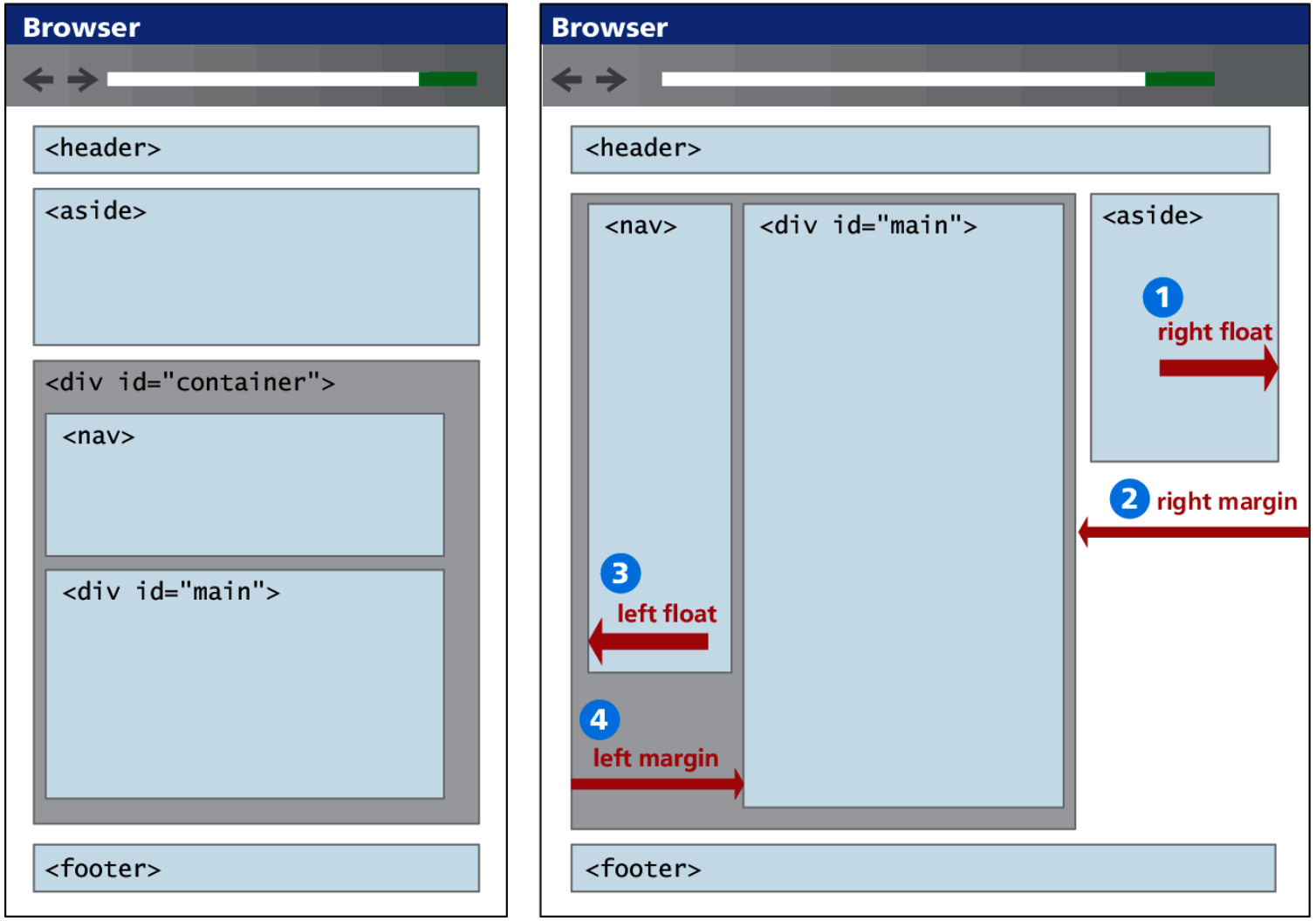
3 Column Example



Using Positioning to Create Columns

Positioning can also be used to create a multicolumn layout. Typically, the approach will be to absolute position the elements that were floa ted in the examples from the previous section

Using Positioning to Create Columns



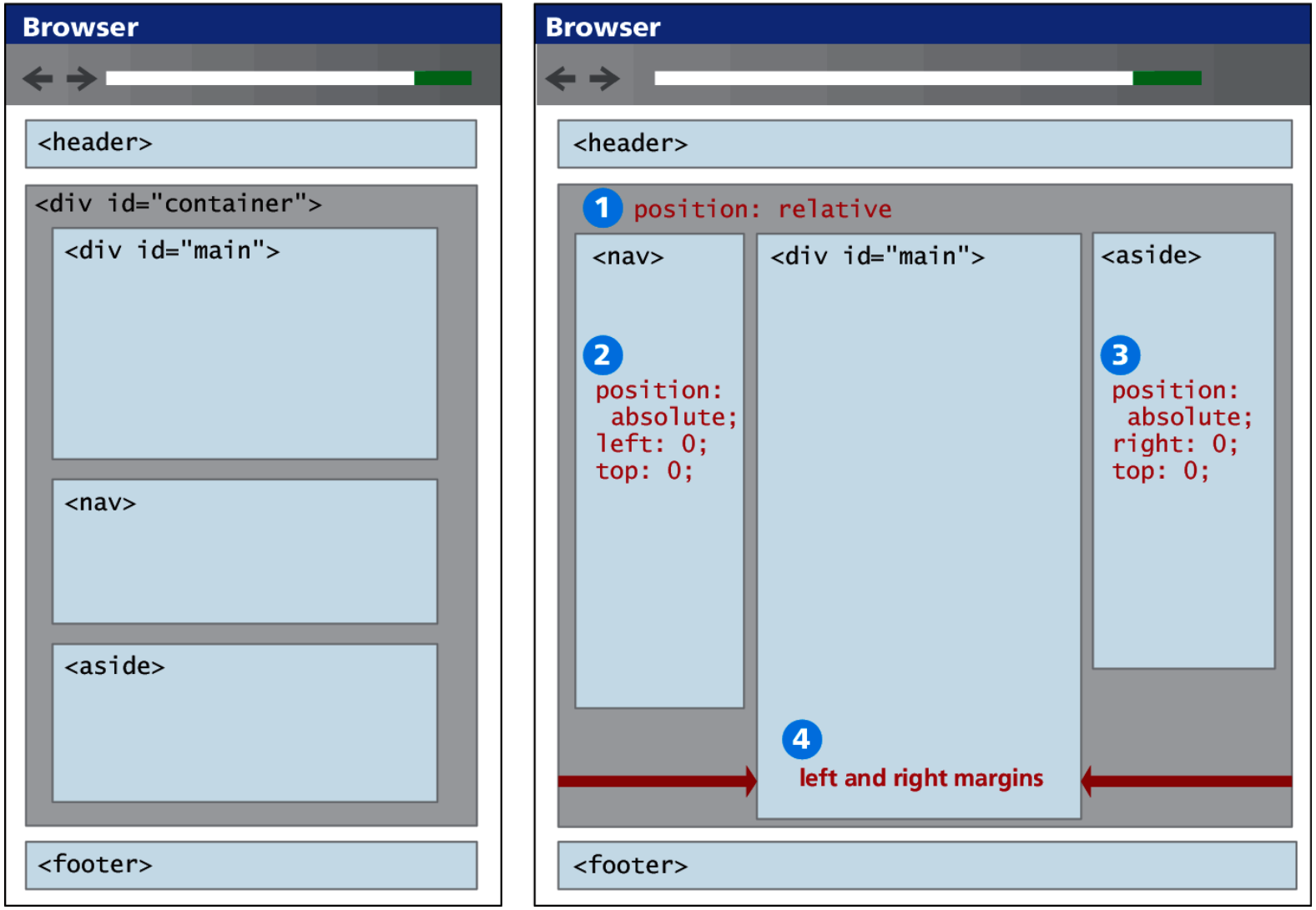
Positioning discussion

Positives

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

Positioning discussion

Positives



Positioning discussion

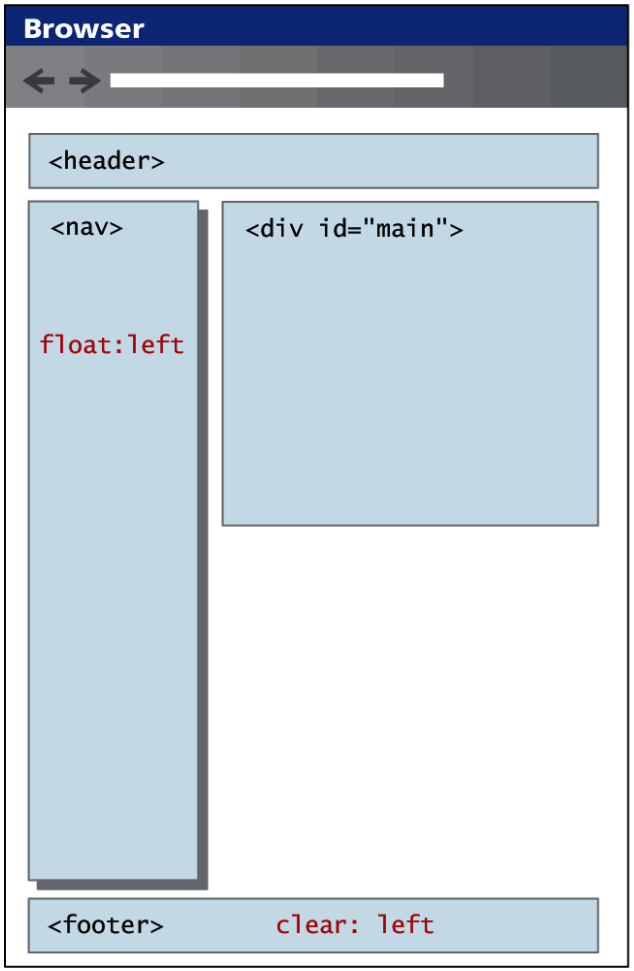
Challenges

What would happen if one of the sidebars had a lot of content and was thus quite long?

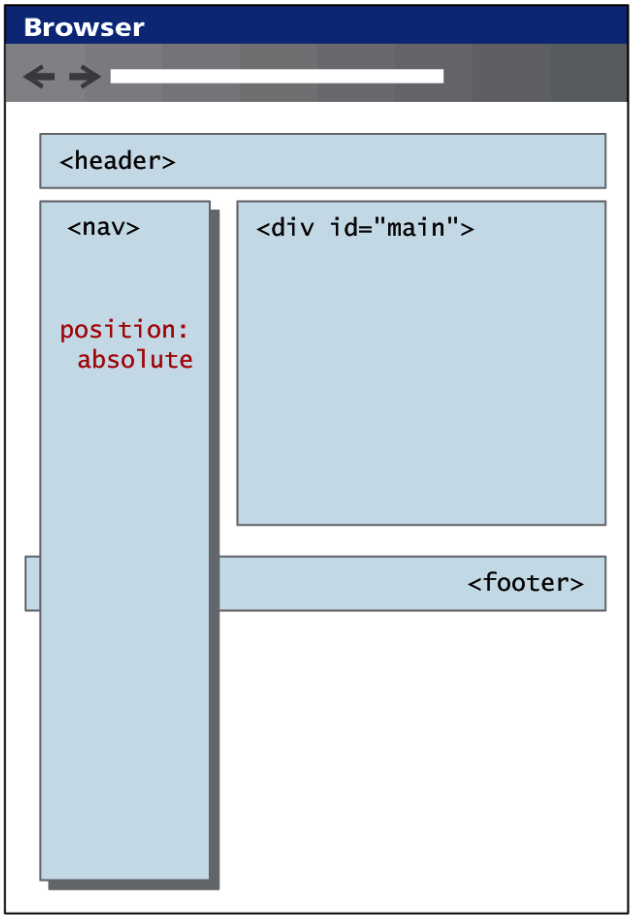
- In the floated layout, this would not be a problem at all, becaus e when an item is floated, blank space is left behind.
- But when an item is positioned, it is removed entirely from nor mal flow, so subsequent items will have no “knowledge” of the positioned item.

Positioning Discussion

Problem illustration



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.

Approaches to CSS Layout

Section 5 of 7

Approaches to CSS Layout

One of the main problems faced by web designers is that the size of t he screen used to view the page can vary quite a bit.

- 21-inch wide screen monitor that can display 1920x1080 pixels
- older iPhone with a 3.5 screen and a resolution of 320x480 px

Satisfying both users can be difficult; the approach to take for one typ e of site content might not work as well with another site with differe nt content.

Approaches to CSS Layout

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

- Fixed Layout
- Liquid Layout
- Hybrid Layout

Fixed Layout

It isn't even broken

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” moni tor resolution

The advantage of a fixed layout is that it is easier to produce and gen erally has a predictable visual result.

Fixed layouts have drawbacks.

- For larger screens, there may be an excessive amount of blank sp ace to the left and/or right of the content.
- It is also optimized for typical desktop monitors; however, as mor e and more user visits are happening via smaller mobile devices

Fixed Layout

Notice the fixed size


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

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


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

Problem with fixed layouts

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

Liquid Layout

Liquidy goodness

In a **liquid layout** (also called a **fluid layout**) widths are not specified using pixels, but percentage values.

Advantage:

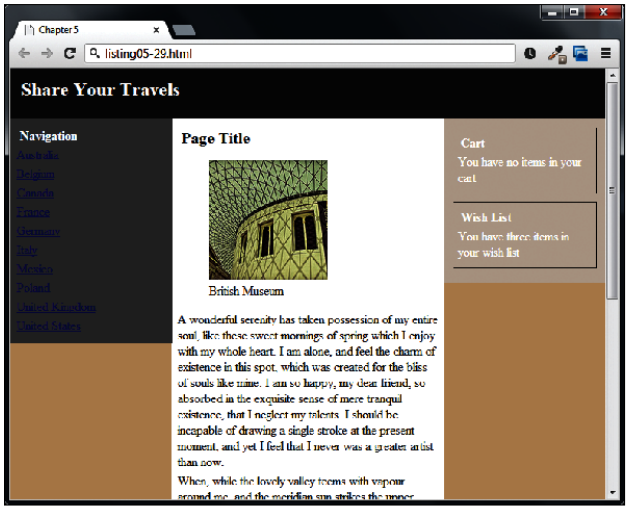
- adapts to different browser sizes,

Disadvantages:

- Liquid layouts can be more difficult to create because some elem ents, such as images, have fixed pixel sizes
- the line length (which is an important contributing factor to read ability) may become too long or too short

Liquid Layout

Liquidy goodness



Fluid layouts are based on the browser window.

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

Hybrid Layout

Such a smug feeling

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 nav igation areas, with perhaps min and max size limits in pixels set f or the navigation areas

Responsive Design

Section 6 of 7

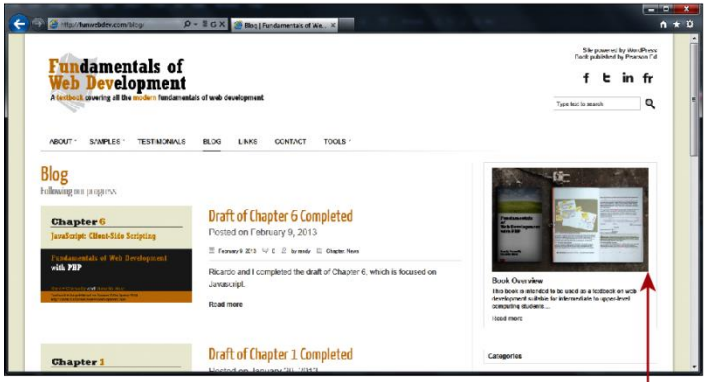
Responsive Design

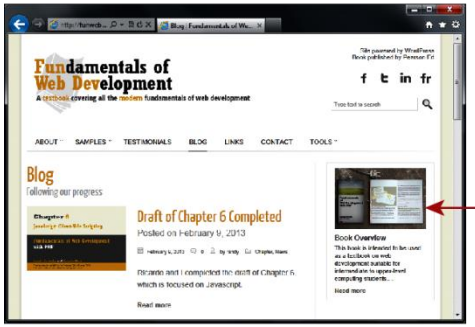
In a **responsive design**, the page “responds” to changes in the brows er size that go beyond the width scaling of a liquid layout.

One of the problems of a liquid layout is that images and horizontal navigation elements tend to take up a fixed size, and when the brow ser window shrinks to the size of a mobile browser, liquid layouts can become unusable. In a responsive layout, images will be scaled dow n and navigation elements will be replaced as the browser shrinks,

Responsive Design

Example





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.

Responsive Design

Example

There are four key components that make responsive design work.

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 Design

Liquid Layout

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 si mply need to specify the following rule:

```
img {

        max-width: 100%;

}
```

Responsive Design

Setting Viewports

A key technique in creating responsive layouts makes use of the abili ty of current mobile browsers to shrink or grow the web page to fit t he width of the screen.

Responsive Design

Setting Viewports

If the developer has created a responsive site that will scale to fit a smaller screen, she may not want the mobile browser to render it o n the full-size viewport. The web page can tell the mobile browser t he viewport size to use via the viewport <meta> element

<html>

<head>

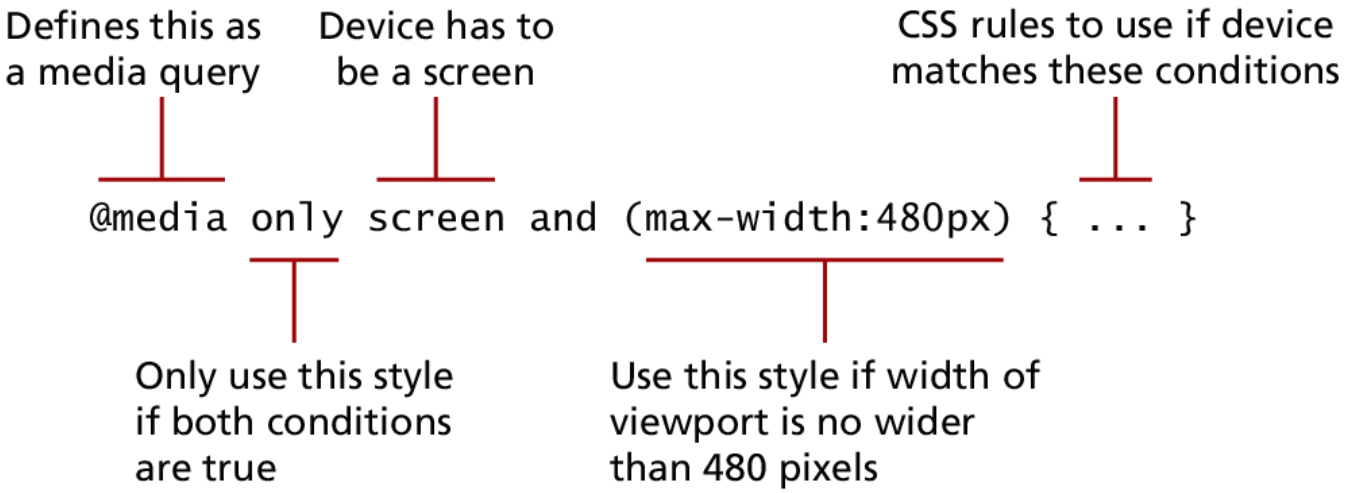
<meta name="viewport" content="width=device-width" />

Responsive Design

Setting Viewports

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 capabiliti es of the device, and then define CSS rules to target that device.



Media Queries

Contemporary responsive sites will typically provide CSS rules for phone displays first, then tablets, then desktop monitors, a n approach called **progressive enhancement**, in which a desig n is adapted to progressively more advanced devices

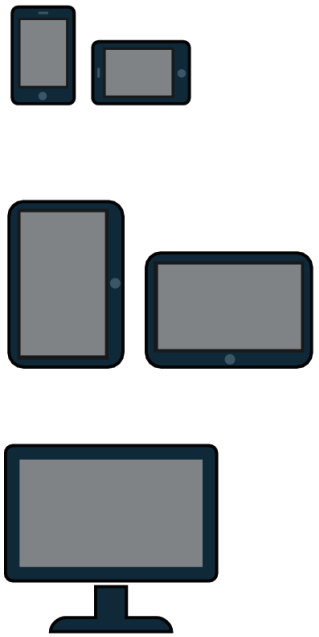
Media Queries

Browser features you can Examine 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

Media Queries

Media queries in action



styles.css

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


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



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

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

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

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

Handles Internet Explorer 8 and earlier using IE conditional comments.

CSS FramEworks

Section 7 of 7

CSS Frameworks

This is hard!

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

They are two main types of CSS framework:

- Grid systems
- CSS preprocessors.

Grid Systems

Frameworks do the heavy lifting

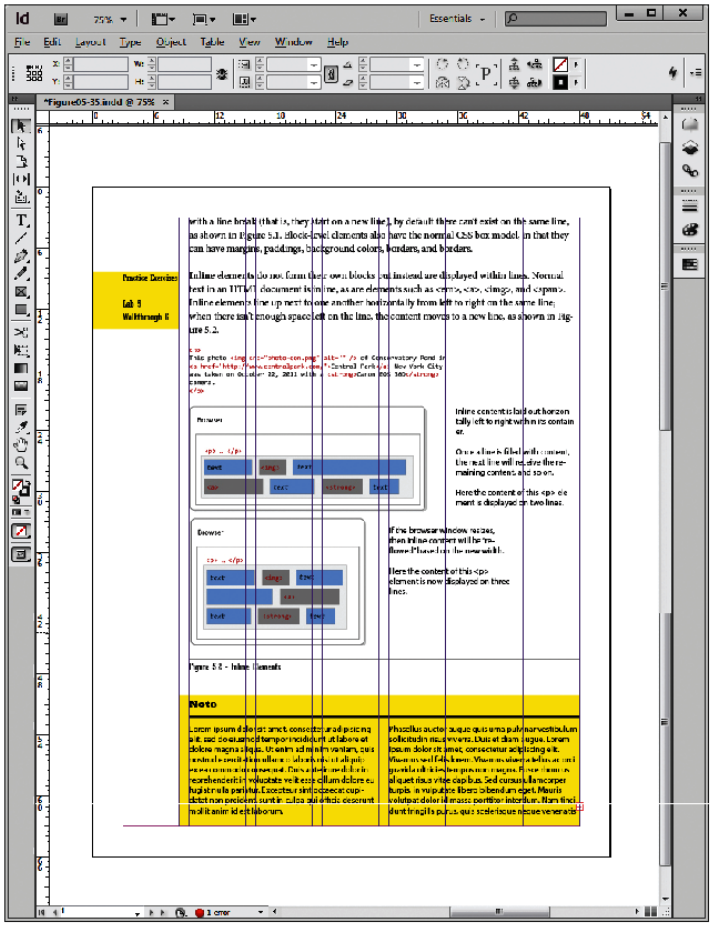
Grid systems make it easier to create multicolumn layouts. Some of the most popular are

- Bootstrap (**twitter.github.com/bootstrap**)
- Blueprint (**www.blueprintcss.org**)
- 960 (**960.gs**)

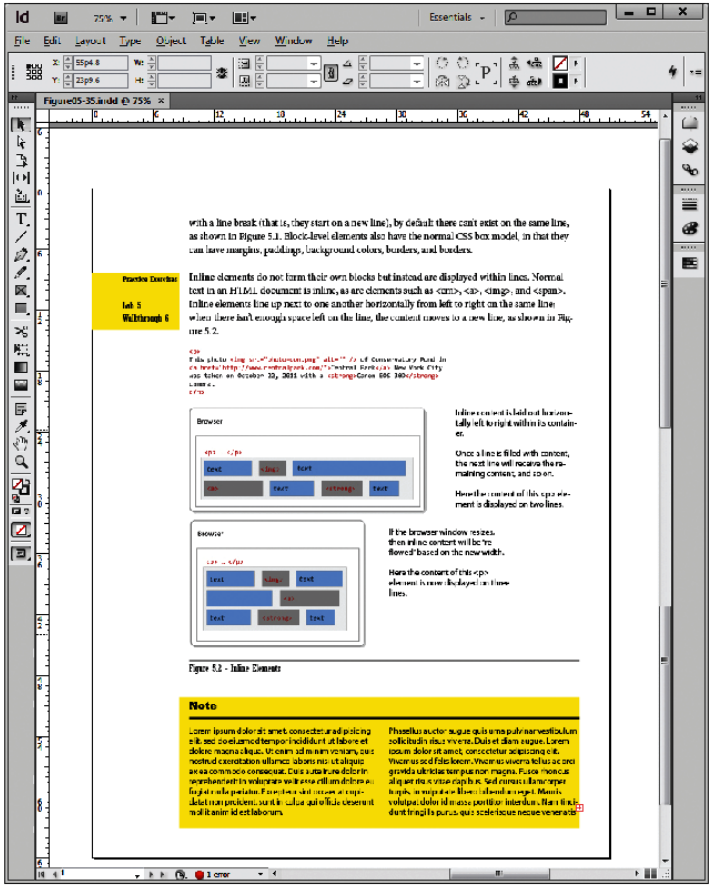
Print designers typically use grids as a way to achieve visual unifo rmity in a design. In print design, the very first thing a designer m ay do is to construct, for instance, a 5- or 7- or 12-column grid. T he rest of the document, whether it be text or graphics, will be al igned and sized according to the grid

Grid Systems

Frameworks do the heavy lifting



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.

Grid Systems

960 code

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

LISTING 5.2 Using the 960 grid

Grid Systems

Bootstrap code

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

LISTING 5.3 Using the Bootstrap grid

Example Pure Bootstrap Layouts

CSS Preprocessors

CSS preprocessors are tools that allow the developer to write CSS that takes advantage of programming ideas such as variables, inh eritance, calculations, and functions.

Most sites make use of some type of color scheme, perhaps four or five colors. Many items will have the same color.

CSS Preprocessors

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


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


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

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

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

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

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

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

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

A mixin can be referenced/called and passed parameters.

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

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

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

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

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

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

What you Learned

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