Module II-Chapter -2
Advanced CSS

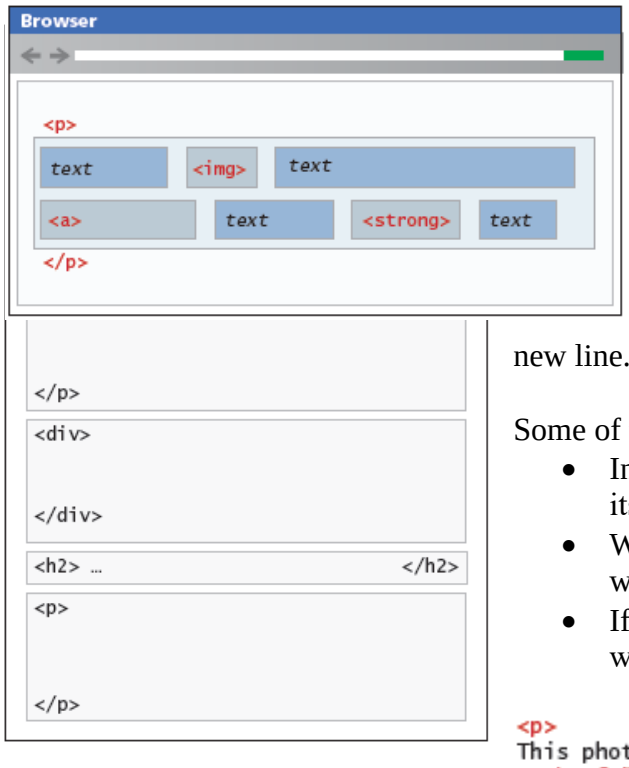
2.7 Normal Flow

The **normal flow** in html refers to how the browser will normally display block-level elements and inline elements from left to right and from top to bottom.

Block-level elements such as <p>, , <h2>, <ul>, and <table> are elements that are contained on their own line, because block-level elements begin with a line break (new line). Two block-level elements can't exist on the same line, without styling.

Some of the properties of block-level elements -

- Each block exists on its own line.
- It 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 (browser window).
- CSS box model properties can be used to customize, for instance, the width of the box and the margin space between other block level elements



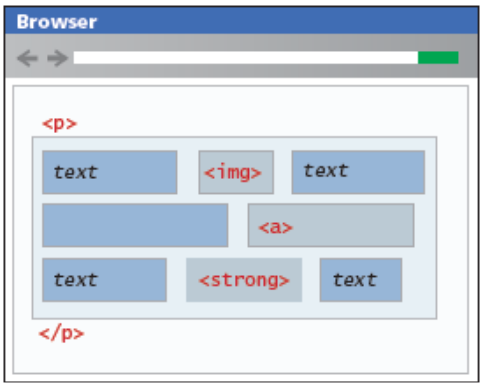
Inline elements do not form their own blocks but instead are displayed within lines. Normal text in an HTML document is inline, and also elements such as , <a>, <img>, and are inline. Inline elements line up next to one another horizontally from left to right on the same line, when there is no enough space left on the line, the content moves to a new line.

Some of the properties of inline elements are –

- Inline element is displayed in normal flow from its container's left to right.
- When a line is filled with content, the next line will receive the remaining content, and so on.
- If the browser window resizes, then inline content will be "re-flowed" based on the new width.

```
<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>
```

If the <p> tag contains many tags as shown, the inline tags are placed with-in the <p> tag as shown. If the window is re-sized the elements are re-flowed and now occupies three rows.



There are two types of inline elements: replaced and nonreplaced. **Replaced inline elements** are elements whose content and 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. Eg: <a>,<b>,<i>,.
Replaced inline elements have a width and height that are defined by the external resource, eg. the size of image is defined externally.

In a document with normal flow, block-level elements and inline elements are placed together. Block-level elements will flow from top to bottom, and inline elements flow from left to right within a block. A block element can contain another block.

It is possible to change whether an element is block-level or inline using 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).

2.8 Positioning Elements

It is possible to
1) move an item from its regular position in the normal flow
2) move an item outside of the browser viewport so that it is not visible
3) move to position so that it is always visible in a fixed position while the rest of the content scrolls.

The position property is used to specify the type of positioning, and the possible values are

Type	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**.

The left, right, top, and bottom properties are used to indicate the distance the element will move.

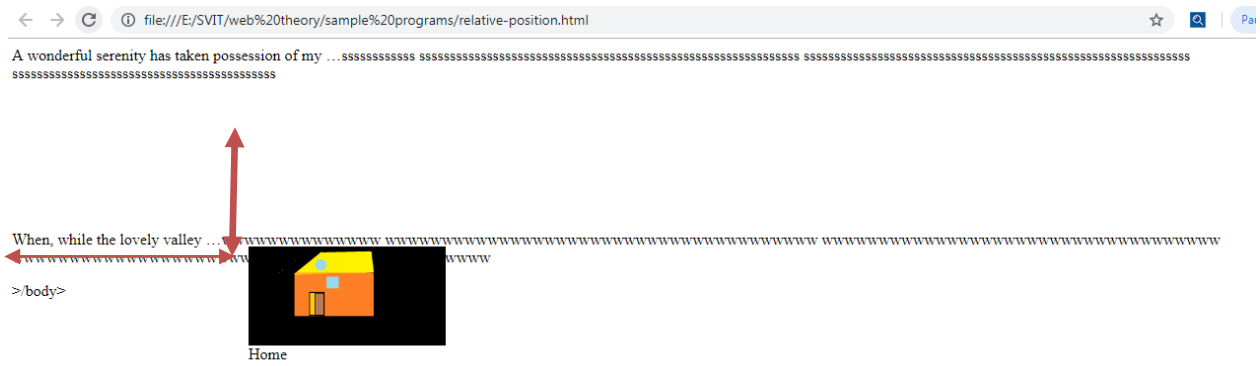
Relative Positioning

In **relative positioning** an element is displaced out of its normal flow position and moved relative to where it would have been placed normally. The other contents around the relatively positioned element remain in its old position in the flow; thus the space the element would have occupied is preserved as shown in the example below.

Eg:

```
<html>
<head>
	<style>
		figure {
		position: relative;
		top: 150px;
		left: 200px;
		}
	</style>
</head>
<body>
<p>A wonderful serenity has taken possession of my …sssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss</p>
<figure>
<img src="HOUSE.png" width="200" height="100" />
<figcaption>Home</figcaption>
</figure>
<p>When, while the lovely valley …wwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww</p>

</body>
</html>
```



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

Absolute Positioning

When an element is positioned absolutely, it is removed completely from normal flow. Here, space is not left for the moved element, as it is no longer in the normal flow. Its position is moved in relation to its container block. In the below example, <figure> block's container is body block.

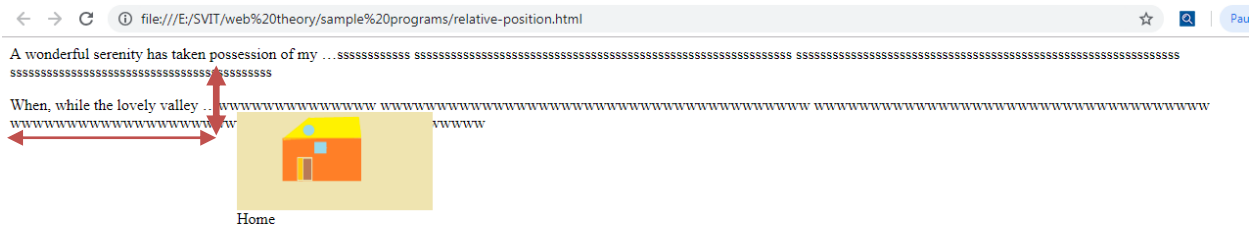
The moved block can overlap the content in the underlying normal flow.

Eg:

```
<html>
<head>
        <style>
        figure {
        position: absolute;
        top: 60px;
        left: 200px;
        }
        </style>
</head>
<body>
<p>A wonderful serenity has taken possession of my …sssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssss</p>
<figure>
<img src="HOUSE.png" width="200" height="100" />
<figcaption>Home</figcaption>
</figure>
<p>When, while the lovely valley …wwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww</p>

</body>
```

</html>



Z-Index

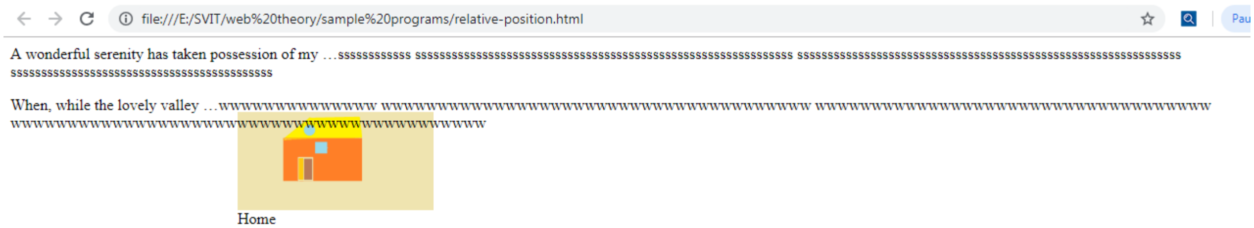
Each positioned element has a stacking order defined by the z-index property (named for the z-axis). Items closest to the viewer (and thus on the top) have a larger **z-index** value, as shown in the example below.

Eg:

```
<html>
<head>
    <style>
        figure {
        position: absolute;
        top: 60px;
        left: 200px;
        z-index:-1;
        }

        body{
        z-index:1:
        }
     </style>
</head>
<body>
<p>A wonderful serenity has taken possession of my …sssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss</p>
<figure>
<img src="HOUSE.bmp" width="200" height="100" />
<figcaption>Home</figcaption>
</figure>
<p>When, while the lovely valley …wwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww</p>

</body>
</html>
```



Observe the <p> tag content has appeared over the image.

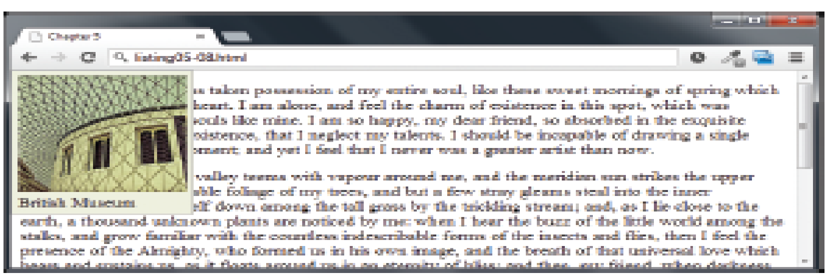
Fixed Position

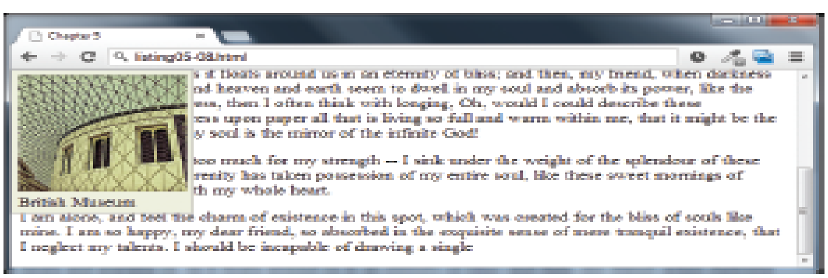
The fixed position value is used relatively infrequently. The element is positioned in relation to the viewport (i.e., to the browser window). Elements with **fixed positioning** do not move when the user scrolls up or down the page.

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

```
<head>
      <style>
          figure {
          position: fixed;
          top: 60px;
          left: 200px;
          }
       </style>
</head>
```

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





2.9 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.

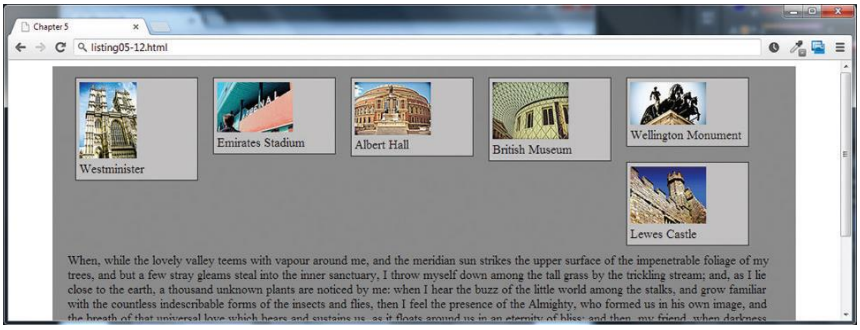
Notice that a floated block-level element must have a width specified; otherwise, the width will be set to auto, which will mean it implicitly fills the entire width of the containing block, and there will be no room available to flow content around the floated item.

Floating within a Container

It should be reiterated that a floated item moves to the left or right of its container.
The floated figure contained within an <article> element that is indented from the browser's edge. The relevant margins and padding areas are color coded to help make it clearer how the float interacts with its container.

Floating Multiple Items Side by Side

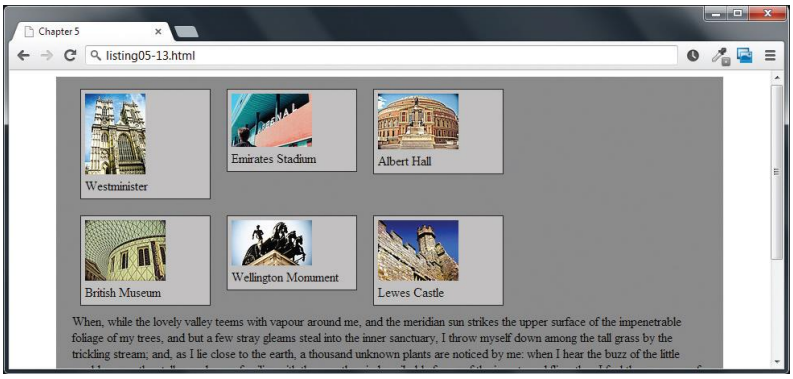
A common use of float property is to place multiple items side by side on the same line. When multiple items are floated, each element will be nestled up beside the previously floated item. All other content in the containing block will flow around all the floated elements.




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

This arrangement of images floated changes as the browser window size changes. If suppose any element has to be stopped from flowing around a floated element, it can be done by using the **clear** CSS **property**.

By setting the clear property of third image to left, it means that there should be no elements to its left. { clear : left;}



The other values for clear property are described below –

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	the left-hand and right-hand edges of the element cannot be adjacent
none	The element can be adjacent to other elements.

Overlaying and Hiding Elements

One of the more common design tasks with CSS is to place two elements on top of each other, or to selectively hide and display elements. Positioning is important to both of these tasks. Positioning is often used for smaller design changes, such as moving items relative to other elements within a container.

An image that is the same size as the underlying one is placed on top of the other image using absolute positioning.

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

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

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

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

2.10 Constructing Multicolumn Layouts

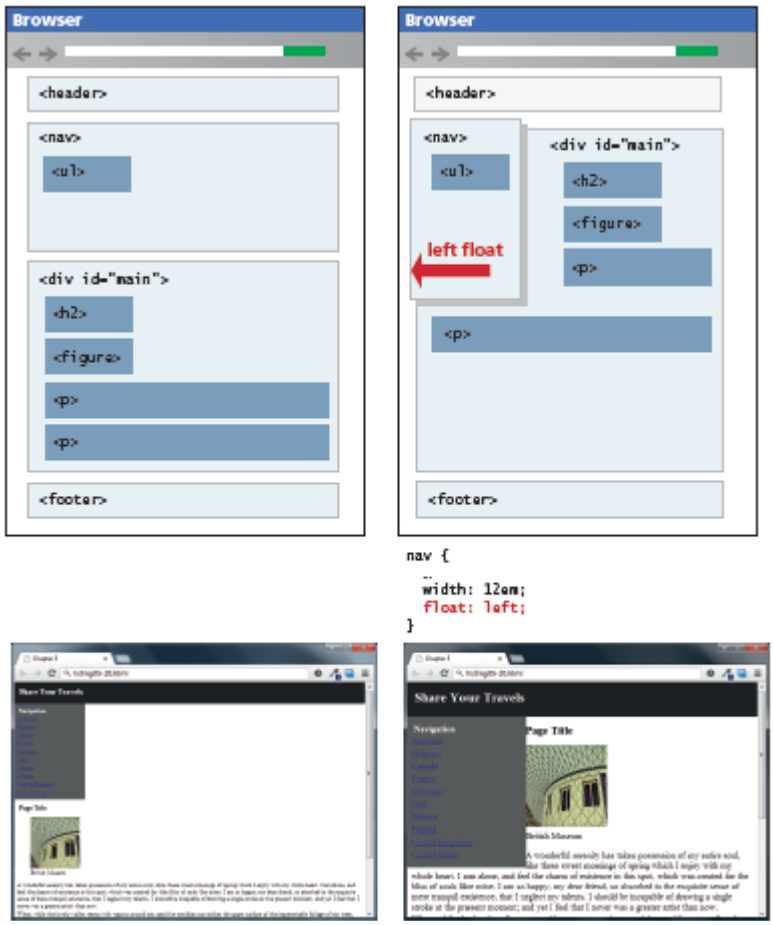
The previous sections showed two different ways to move items out of the normal top-down flow, by using positioning (relative, absolute or fixed) and by using floats. They are the techniques that can be used to create more complex layouts. The below topics are about the creation of layout using float and positioning property of CSS.

Using Floats to Create Columns

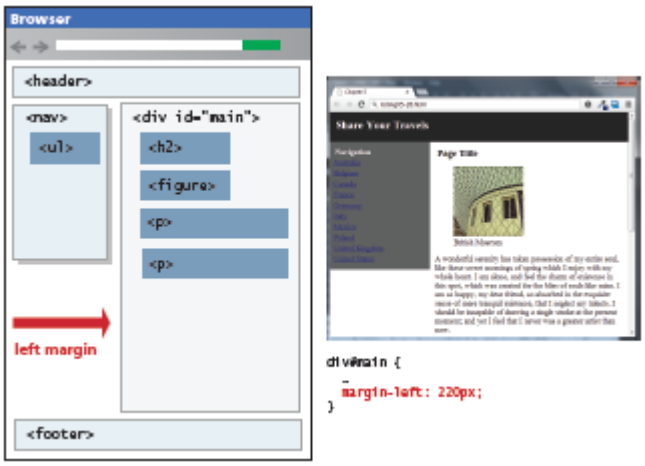
Using floats is the most common way to create columns of content. The steps for this approach are as follows –

1. float the content container that will be on the left-hand side. (the floated container needs to have a width specified).
2. The other content will flow around the floated element.
3. Set the left – hand side margin for the non-floated element.

The layout without float Property	The layout after using the float property for left side element



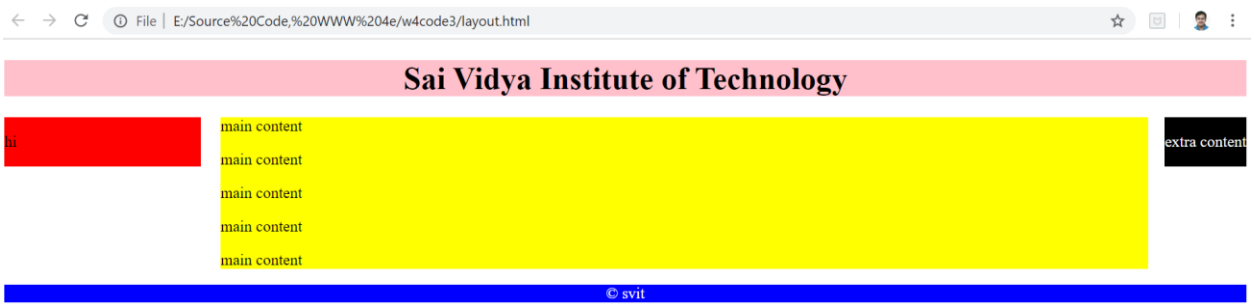
Step 3: setting the left margin -




```
<!DOCTYPE html>
<html>
        <head>
                <title> Simple Form </title>
                <style type = "text/css">
                        nav{
```

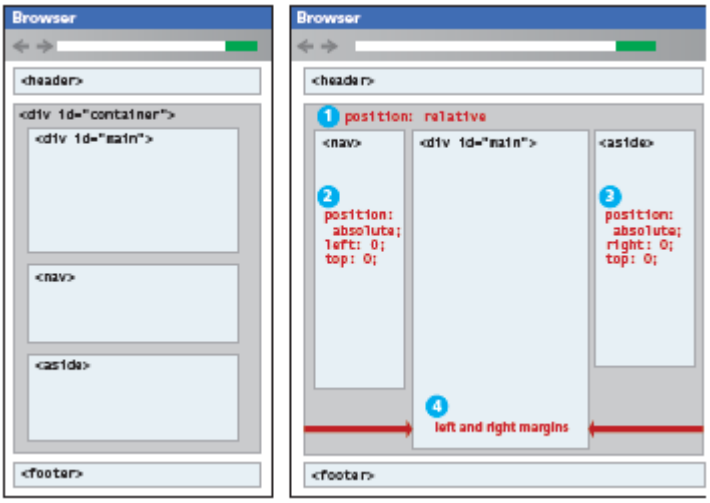
```
                        width:200px;
                        float:left;
                        background-color:red;
                        }
                div{
                        margin-left: 220px;
                        margin-right: 100px;
                        background-color:yellow;
                        }
        </style>
    </head>
    <body>
        <header style="background-color:pink;align:center;">
            <h1 style="text-align:center;">Sai Vidya Institute of Technology</h1>
        </header>
        <aside style="background-color:black;color:white;float:right;">
            <p>extra content</p>
        </aside>
        <nav>
            <p>hi</p>
        </nav>
        <div>
            <p> main content </p>
            <p> main content </p>
            <p> main content </p>
            <p> main content </p>
            <p> main content </p>
        </div>

        <footer style="background-color:blue;color:white;">
            <p style="text-align:center;">&copy svit</p>
        </footer>
    </body>
</html>
```



Using Positioning to Create Columns

Positioning can also be used to create a multicolumn layout. Typically, the approach is to absolute position the elements.
This approach uses some type of container, in which the elements are positioned.



The following steps are followed –

1. Position the container element (into which all other elements are positioned) with respect to the browser window.
2. Position the other elements with respect to the container element, created in step 1.

```
<html>
<head>
<style>

#container{
        position:relative;              //main container
        top:0px;
        left:0px;
}
nav
{
        position:absolute;              // positioned with respect to main container
        top: 0px; left:0px;
        width:150px;
        height:300px;
        background-color: #cc0055
}

#side{
        margin-left: 150px;                     // column in the middle
        margin-right:150px;
        height: 300px;
        background-color:#CCCCCC}
h1{   background-color:red; }
```

```
</style>
</head>
<body>

<h1 align="center" > HHHHH</h1>

<div id="container">

<nav>
<ul>
<li>abc</li>
<li>def</li>
<li>abc</li>
<li>def</li>
</ul>
</nav>

<article id="side">
<figure>
<img src="HOUSE.bmp" width="200" height="100" />
<figcaption>Home</figcaption>
</figure>

<p>A wonderful serenity has taken possession of my …sssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssss</p>

</article>

<p style = "clear:left" >When, while the lovely valley …wwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww</p>
</div>
</body>
</html>
```

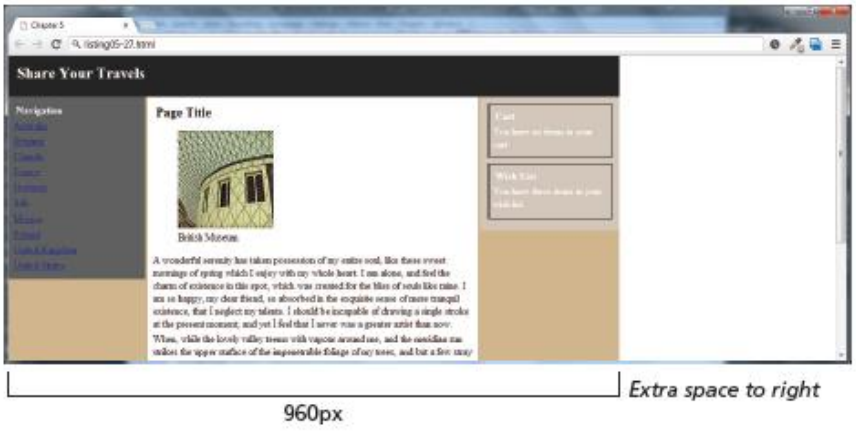
2.11 Approaches to CSS Layout

One of the main problems faced by web designers is that the size of the screen used to view the page can vary. Some users will visit a site on a 21-inch wide screen monitor that can display 1920 × 1080 pixels (px); others will visit it on an older iPhone with a 3.5 screen and a resolution of 320 × 480 px. Users with the large monitor might expect a site to take advantage of the extra size; users with the small monitor will expect the site to scale to the smaller size and still be usable.

Most designers take one of two basic approaches to dealing with the problems of screen size - **Fixed Layout and Liquid 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 may be positioned on the left or the center of the monitor.



Fixed layouts are created using pixel units, typically with the entire content within a container whose width property has been set to some width.

```
<style>
div#wrapper {
width: 960px;
background_color: tan;
}
</style>

<body>
<div id="wrapper">
……
</div>
……..
</body>
```

The advantage of a fixed layout –

- easy to produce
- predictable visual result
- optimized for typical desktop monitors

The disadvantage of a fixed layout –

- 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 is less than the fixed width; the user will have to horizontally scroll to see all the content.
- If smaller mobile devices are used, more horizontal scrolling has to be done

Liquid Layout

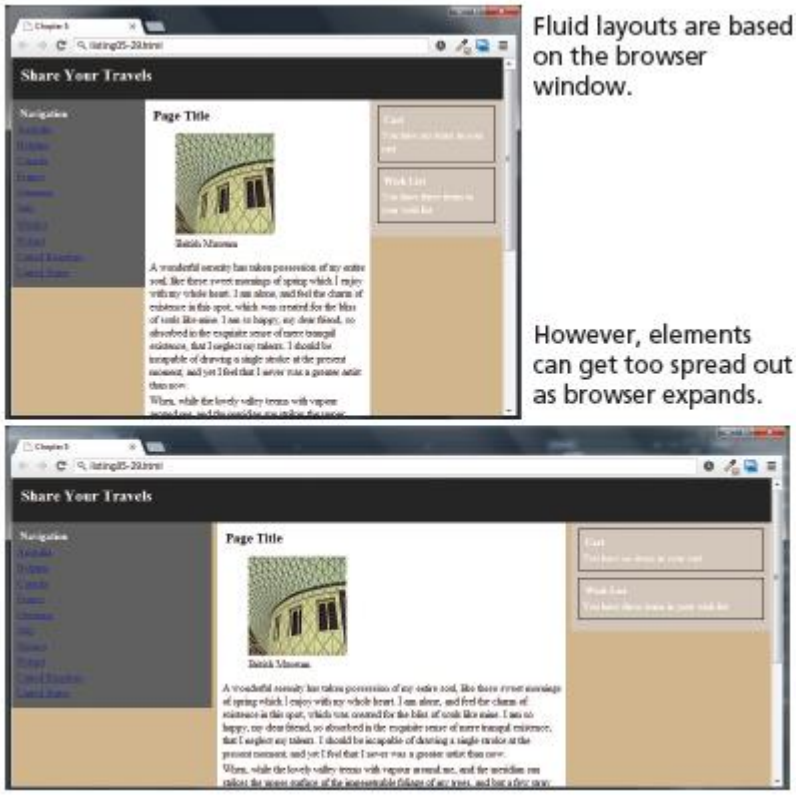
In Liquid Layout, the widths are not specified using pixels, but percentage values. Percentage values in CSS are a percentage of the current browser width, so a layout in which all widths expressed as percentages should adapt to any browser size.

The advantage of a liquid layout –

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

The disadvantage of a liquid layout –

- more difficult to create because some elements, such as images, have fixed pixel sizes.
- The screen may grow or shrink dramatically.



Other Layout Approaches

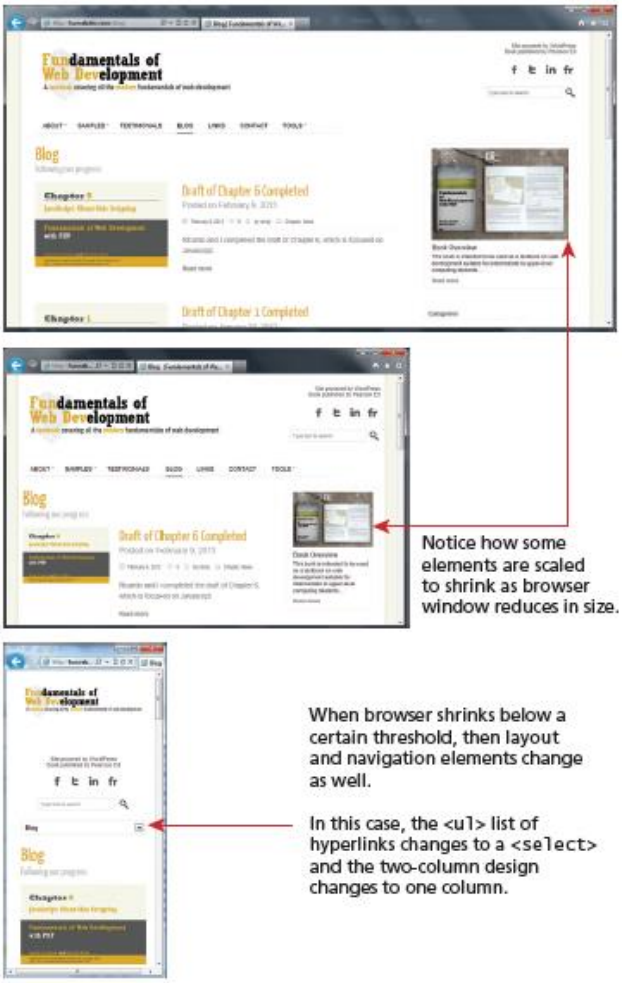
While the fixed and liquid layouts are the two basic paradigms for page layout, there are some other approaches that combine the two layout styles. Most of the other approaches are **hybrid layout**, were there is a fixed layout and liquid layout.

Fixed layout is commonly used for a sidebar column containing graphic advertising images that must always be displayed and which always are the same width. But liquid layout are used for the main content or navigation areas, with perhaps min and max size limits in pixels set for the navigation areas.

2.12 Responsive Design

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

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



There are four key components that make responsive design work. They are:

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

Responsive designs begin with a liquid layout, in which most elements have their widths specified as percentages. Making images scale in size is done as follows:

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

But this does not change the downloaded size of the image; it only shrinks or expands its visual display to fit the size of the browser window, never expanding beyond its actual dimensions.

Setting Viewports

A key technique in creating responsive layouts is the ability of current mobile browsers to shrink or grow the web page to fit the width of the screen. The mobile browser renders the page on a canvas called the **viewport**. On iPhones, for instance, the viewport width is 980 px, and then that viewport is scaled to fit the current width of the device. The mobile Safari browser introduced the viewport <meta> tag as a way for developers to control the size of that initial viewport.

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

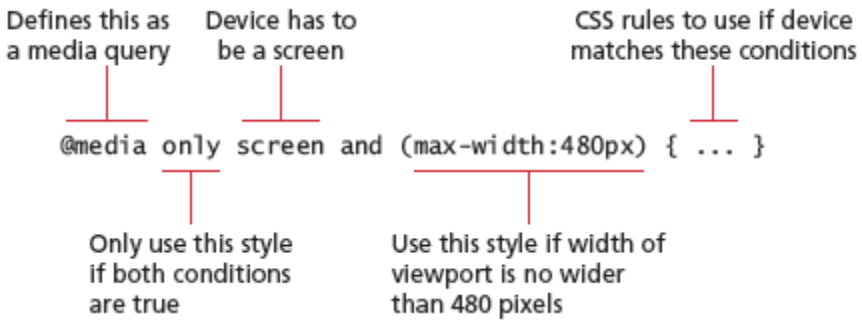
By setting the viewport as above, the page is telling the browser that no scaling is needed, and to make the viewport as many pixels wide as the device screen width. This means that if the device has a screen that is 320 px wide, the viewport width will be 320 px; if the screen is 480 px, then the viewport width will be 480 px.

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. Use these queries to look at the capabilities of the device, and then define CSS rules to target that device.

Example of media query

@media only screen and (max-width: 480px) {……}



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.

Few elements of the browser features that can be examined with media queries are –

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

Contemporary responsive sites will typically provide CSS rules for phone displays first, then tablets, then desktop monitors, an approach called **progressive enhancement**. The media queries can be within your CSS file or within the <link> element; the later requires more HTTP requests but results in more manageable CSS files.

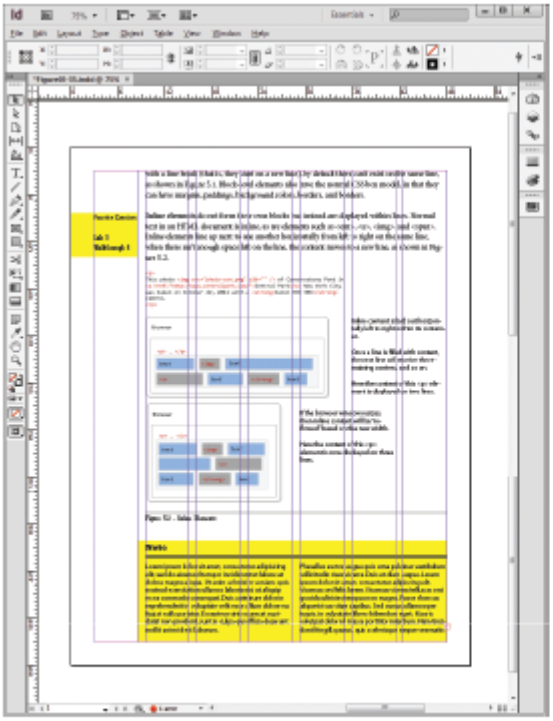
2.13 CSS Frameworks

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

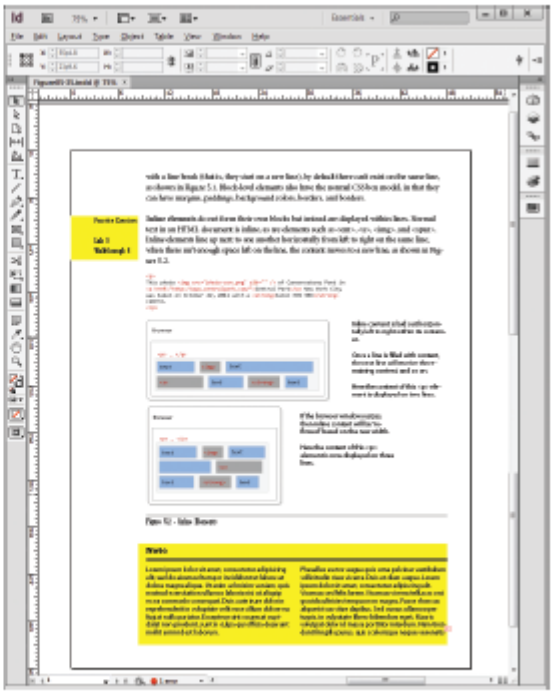
Grid Systems

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**).

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



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.

CSS frameworks provide similar grid features. The 960 framework uses either a 12- or 16-column grid. Bootstrap uses a 12-column grid. Blueprint uses a 24-column grid. The grid is constructed using elements with classes defined by the framework. The HTML elements for the rest of the site are then placed within these elements.

Eg –

```
<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>
```

The above code creates a three column layout similar as shown in the above figure. The frameworks allow columns to be nested, making it quite easy to construct the most complex of layouts.

CSS Preprocessors

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

The advantage of a CSS preprocessor is that it can provide additional functionalities that are not available in CSS. One of the best ways to see the power of a CSS preprocessor is with colors. Most sites make use of some type of color scheme, perhaps four or five colors. Many items will have the same color.

As shown in the below figure, the background color of the .box class, the text color in the <footer> element, the border color of the <fieldset>, and the text color for placeholder text within the <textarea> element, might all be set to #796d6d. The trouble with regular CSS is that when a change needs to be made, then some type of copy and replace is necessary, which always leaves the possibility that a change might be made to the wrong elements. Similarly, it is common for different site elements to have similar CSS formatting, for instance, different boxes to have the same padding.

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.

```
$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.