HTML and CSS

Training manual

ver.1.2

Copyright

Copyright © 2016 Mitrais

All rights reserved.

Confidentiality

This document contains confidential information of Mitrais.

This document can be used and copied within Mitrais for use in relation to this project. However, this document, including copies of it, must not be disclosed to any person who is not a director, officer, employee, consultant or agent of Mitrais without the prior written approval of Mitrais and such person or agent has signed a non-disclosure agreement satisfactory to Mitrais.

Document control

This document is deemed to be controlled only as long as the project to which it relates is active. Controlled copies will be automatically updated. Once the project is completed or terminated, this document will revert to an uncontrolled document status. No further advice will be provided, and each recipient may either destroy the document or mark it as obsolete and retain it for future personal reference. All copies of this document will be issued electronically.

Distribution

Copies of this document will be made available to Mitrais.

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 6 Nov 2002 | 0.1 | Created | Kok Sang |
| 19 Sep 2006 | 0.2 | Revision | Kok Sang |
| 09 Jun 2009 | 0.3 | Fix dead links | Eko Pramono |
| 29 Oct 2009 | 0.4 | Add Exercises | Eko Pramono |
| 16 Sep 2010 | 1.0 | Rewrite and remove Javascript topic | Andrew Candraditya D |
| 21 Jan 2012 | 1.1 | Additional for HTML 5 section and CSS3 | Yusuf Widiyatmono |
| 29 Feb 2016 | 1.2 | Add HTML5 Form  Add API - Geolocation and drag-drop  Add Detail Video/Audio in Section New Element  Add HTML5 Graphics – Canvas and SVG  Add MicroData | Dedy Hartono  Andry Baka  Agni Pradharma |

Table of Contents

[Copyright 2](#_Toc444507757)

[Revision History 3](#_Toc444507758)

[Table of Contents 4](#_Toc444507759)

[1. Introduction 6](#_Toc444507760)

[2. HTML 6](#_Toc444507761)

[2.1. Getting Started 6](#_Toc444507762)

[2.2. HTML Markup 7](#_Toc444507763)

[2.2.1. Syntax 7](#_Toc444507764)

[2.2.2. Presentation and Behaviour 8](#_Toc444507765)

[2.2.3. Document head Elements 8](#_Toc444507766)

[2.2.4. Document body Elements 10](#_Toc444507767)

[2.2.5. Comments 13](#_Toc444507768)

[2.3. Document Type Definition (DTD) 14](#_Toc444507769)

[2.4. Introduction to XHTML 14](#_Toc444507770)

[2.4.1. Syntax Differences 14](#_Toc444507776)

[2.4.2. Behavioural Differences 15](#_Toc444507777)

[2.4.3. XHTML 1.0 Document Type 15](#_Toc444507778)

[2.5. HTML Semantics 16](#_Toc444507779)

[2.6. HTML Validation 17](#_Toc444507780)

[2.7. HTML Today - HTML 5 18](#_Toc444507781)

[2.7.1. Simple DTD 18](#_Toc444507784)

[2.7.2. HTML and XHTML 18](#_Toc444507785)

[2.7.3. New way to build your structure 18](#_Toc444507786)

[2.7.4. New Elements 19](#_Toc444507787)

[2.7.5. HTML5 Graphics 21](#_Toc444507788)

[2.7.5.1. Canvas 21](#_Toc444507789)

[2.7.5.2. SVG 22](#_Toc444507790)

[2.7.5.3. Comparison between Canvas and SVG 23](#_Toc444507791)

[2.7.6. API 23](#_Toc444507792)

[2.7.6.1. API for Geolocation 24](#_Toc444507793)

[2.7.6.1.1. Getting the Current Position 24](#_Toc444507794)

[2.7.6.1.2. Handling Geolocation Errors 26](#_Toc444507795)

[2.7.6.1.3. Specifying Geolocation Options 28](#_Toc444507796)

[2.7.6.1.4. Monitoring the Position 29](#_Toc444507797)

[2.7.7. Working with Forms in HTML5 30](#_Toc444507798)

[2.7.7.1. New HTML5 Form Element and Input Type 31](#_Toc444507799)

[2.7.7.2. Additional Attributes for Input Type 31](#_Toc444507800)

[2.7.7.3. Form Input Validation 32](#_Toc444507801)

[2.7.8 Browser Support 34](#_Toc444507802)

[3. CSS 35](#_Toc444507803)

[3.1. Getting your styles to hit the target 35](#_Toc444507805)

[3.1.1. Type Selectors 36](#_Toc444507806)

[3.1.2. ID and Class Selectors 36](#_Toc444507807)

[3.1.3. Universal Selectors 36](#_Toc444507808)

[3.1.4. Advanced Selectors 36](#_Toc444507809)

[3.2. The Cascade and Specificity 39](#_Toc444507810)

[3.2.1. Specificity Hierarchy 39](#_Toc444507813)

[3.2.2. How to measure specificity? 40](#_Toc444507814)

[3.2.3. Using specificity in your stylesheets 40](#_Toc444507815)

[3.3. Inheritance 40](#_Toc444507816)

[3.4. Applying the style 41](#_Toc444507817)

[3.5. Commenting your Code 41](#_Toc444507818)

[3.6. CSS 2.1 Properties 43](#_Toc444507819)

[3.6.1. Box Model 43](#_Toc444507820)

[3.6.2. Visual Effects 43](#_Toc444507821)

[3.6.3. Colors and Backgrounds 44](#_Toc444507822)

[3.6.4. Fonts 44](#_Toc444507823)

[3.6.5. Text 45](#_Toc444507824)

[3.6.6. User Interface 45](#_Toc444507825)

[3.7. CSS 3.0 Properties 45](#_Toc444507826)

[3.7.1. Borders 45](#_Toc444507827)

[3.7.2. Background 46](#_Toc444507828)

[3.7.3. Color 46](#_Toc444507829)

[3.7.4. Text Effects 46](#_Toc444507830)

[3.7.5. User Interface 46](#_Toc444507831)

[3.7.6. Web Fonts 47](#_Toc444507832)

[3.8. Visual Formatting 47](#_Toc444507833)

[3.8.1. Box Model Recap 47](#_Toc444507834)

[3.8.2. Positioning 51](#_Toc444507835)

[3.8.3. Layout 58](#_Toc444507836)

[3.9. Font Formatting 70](#_Toc444507837)

[3.10. CSS Reset 72](#_Toc444507838)

[3.11. Hacks and Filters 72](#_Toc444507839)

[3.11.1. Conditional CSS 73](#_Toc444507840)

[3.11.2. Conditional Comment 75](#_Toc444507841)

[3.11.3. Child Selector Hack 75](#_Toc444507842)

[3.11.4. Attribute Selector Hack 76](#_Toc444507843)

[3.11.5. Star HTML Hack 76](#_Toc444507844)

[3.11.6. !important and underscore Hack 76](#_Toc444507845)

[3.11.7. Owen Hack 78](#_Toc444507846)

[3.12. Bug Fixing 78](#_Toc444507847)

[3.12.1. Having Layout 78](#_Toc444507848)

[4. References 80](#_Toc444507849)

1. Introduction

A **web page** or **webpage** is a document or resource of information that is suitable for the World Wide Web and can be accessed through a web browser and displayed on a monitor or mobile device. This information is usually in HTML or XHTML format, and may provide navigation to other webpages via hypertext links.

Webpages may be retrieved from a local computer or from a remote web server. The web server may restrict access only to a private network, e.g. a corporate intranet, or it may publish pages on the World Wide Web. Webpages are requested and served from web servers using Hypertext Transfer Protocol (HTTP).

1. HTML

HTML stands for the **Hypertext Mark-up Language**. Is written in the form of HTML elements consisting of "tags" surrounded by angle brackets within the web page content.

It allows images and objects to be embedded and can be used to create interactive forms. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items.

It is important to understand that HTML is a standard, composed of recommendations published by an international consortium the [World Wide Web Consortium](http://www.w3.org/MarkUp/) (**W3C**).

* 1. Getting Started

HTML markup consists of several key components, including *elements* (and their *attributes*), character-based *data types*, *character references* and *entity references*.

A HTML document is composed of three parts:

1. a line containing HTML version information,
2. a declarative header section (delimited by the **HEAD** element),
3. a body, which contains the document's actual content. The body may be implemented by the **BODY** element or the **FRAMESET** element.

White space (spaces, newlines, tabs, and comments) may appear before or after each section. Sections 2 and 3 should be delimited by the **HTML** element.

The Example of HTML code and tree structure:

|  |  |
| --- | --- |
| <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  "http://www.w3.org/TR/html4/strict.dtd">  <HTML>  <HEAD>  <TITLE>My first HTML document</TITLE>  </HEAD>  <BODY>  <P>  <strong>Hello world!</strong>  </p>  </BODY>  </HTML> |  |

* 1. HTML Markup

An HTML page is a basic text file containing **tags** for specifying text format, images, etc. The use of these tags is called *markup*.

* + 1. Syntax

A tag is a text element (a name) flanked by a less-than sign (<) and greater-than sign (>). For example, "*<H1>*". It usually work in pairs, and affect whichever elements they are surrounding. The first one is called the *opening tag*, and the second one is called the *closing tag*. The closing tag begins with a backslash ( */* ):

<p> This is paragraph </p>

The other example for the HTML Tags that used alone rather than in pairs is the <br> tag, for example, represents a line break.

**Nesting Tags**

HTML tags can be nested within one another, to allow multiple properties to be applied to the same text. However, overlapping tags are not tolerated by the HTML standard.

**Incorrect:**

~~<strong> This is a wrong <em>Nesting Structure </strong></em>~~

**Correct:**

<strong> This is a wrong <em>Nesting Structure</em> </strong>

**Element / Tags Attributes**

**Attributes** define the behavior or indicate additional element properties. Most attributes require a *value*. In HTML, the value can be left unquoted if it doesn't include spaces (name=value), or it can be quoted with single or double quotes (name='value' or name="value"). But, in XML those quotes are required. Boolean attributes, on the other hand, don't require a value to be specified.

An example is the checked for checkboxes:

<input type=checkbox checked>

In the XML syntax, though, the name should be repeated as the value:

<input type="checkbox" checked="checked"/>

* + 1. Presentation and Behaviour

In keeping with the principle of [separation of concerns](http://en.wikipedia.org/wiki/Separation_of_concerns), the function of HTML is primarily to add structural and semantic information to the raw text of a document. Presentation and behaviour are separate functions, which can be added as desired, ideally through links to external documents such as stylesheets, graphics files, and scripts.

This allows the document to be presented by different user agents according to their purposes and abilities; for example, a user agent can select an appropriate stylesheet to present a document by displaying on a monitor, printing on paper, or to determine speech characteristics in an aural user agent. The structural and semantic functions of the markup remain identical in each case.

An HTML document can also be extended through the use of scripts to provide additional behaviours beyond the abilities of HTML hyperlinks and forms. The elements style and script, with related attributes, provide reference points in HTML markup for links to stylesheets and scripts. They can also contain instructions directly.

* In the document head, script and style may either link to shared external documents, or contain embedded instructions. (The link element can also be used to link stylesheets.)
* The style attribute is valid in most document body elements for inclusion of inline style instructions.
* Event-handling attributes, which provide links to scripts, are optional in most elements.
* script can occur at any point in the document body.
* For user agents which do not operate scripts, the noscript element provides alternative content where appropriate; however, it can only be used as a block-level element.
  + 1. Document head Elements

<base>

Specifies a **base URL** for all relative href and other links in the document. Must appear before any element that refers to an external resource. HTML permits only one base element for each document. The base element has attributes, but no contents.

<title>…</title>

Define a document title. Any text between the opening and closing TITLE tags displays in the browser title bar and in the Microsoft Windows taskbar. In Web pages, "Internet Explorer" is appended to the title.

<title>Hello</title>

<style>…</style>

Specifies a style sheet for the page. This element is not rendered and requires a closing tag.

<style type=”text/css”>

H1 { font: 8pt Arial bold; }

<style>

Can either act as a container for style instructions or link to external stylesheets – for example,

<style> @import *url*; </style>

<script>…</script>

Specifies a script for the page that is interpreted by a script engine. Code within the SCRIPT block that is not contained within a function is executed immediately as the page is loaded. To keep scripts from being displayed on down-level browsers, nest the SCRIPT block within a COMMENT block.

<script language=”javascript”>

alert(“Hello Word”);

</script>

Can either act as a container for link to an external script with the optional src attribute

<script type="text/javascript" src=”somepath.js”></script>

<link>

Specifies links to other documents, such as *previous* and *next* links, or alternate versions. A common use is to link to external stylesheets, using the form:

<link rel=stylesheet href="styles.css" type="text/css">

A less-common, but important, usage is to supply navigation hints consistently through use of microformats. Several common relationships are defined, that may be exposed to users through the browser interface rather than directly in the web page.

<link rel="next" href="*url*">

<meta>

Can be used to specify additional metadata about a document, such as its author, publication date, expiration date, page description, keywords, or other information not provided through the other header elements and attributes. Because of their generic nature, meta elements specify associative [key-value pairs](http://en.wikipedia.org/wiki/Associative_array).

In one form, meta elements can specify HTTP headers which should be sent by a web server before the actual content, for example:

<meta http-equiv="foo" content="bar">

In the general form, a meta element specifies name and associated content attributes describing aspects of the HTML page. To prevent possible ambiguity, an optional third attribute, scheme, may be supplied to specify a semantic framework that defines the meaning of the key and its value: for example:

<meta name="foo" content="bar" scheme="DC">

* + 1. Document body Elements

In visual browsers, displayable elements can be rendered as either *block* or *inline*. While all elements are part of the document sequence, block elements appear within their parent elements:

* as rectangular objects which do not break across lines;
* with block margins, width and height properties which can be set independently of the surrounding elements.

Conversely, inline elements are treated as part of the flow of document text; they cannot have margins, width or height set, and do break across lines.

**Block Elements**

<p>…</p>

Creates a paragraph

<p> This is a paragraph </p>

<h1>…</h1> … <h6>…</h6>

Section headings at different levels. <h1> delimits the highest-level heading, <h2> the next level down (sub-section), <h3> for a level below that, and so on to <h6>

<h1> This is heading </h1>

<ul>…</ul>

An unordered (bulleted) list. Used together with <li> element.

<ul>

<li> Menu 1 </li>

<li> Menu 2 </li>

</ul>

<li>…</li>

A list item in ordered <ol> or unordered <ul> lists.

**Inline Elements**

**Anchor**

<a>

An anchor element is called an anchor because web designers can use it to anchor a URL to some text on a web page. When users view the web page in a browser, they can click the text to activate the link and visit the page whose URL is in the link.

<a href="#anchor">anchor</a>

Both text and images can be included within an anchor. The anchor becomes a hyperlink to either another part of the document or another resource (e.g. a webpage) using an external URL.

<a href=”http://www.w3.org”>World Wide Web Consortium</a>

<a href=”http://www.w3.org”><img src=”link to images” /></a>

**Phrase Elements**

<strong>

**strong emphasis** (conventionally displayed bold).

<strong>This text would appear as bold</strong>

<em>…</em>

Emphasis (conventionally displayed in italics)

<strong>This text would appear as italic</strong>

<span>…</span>

An inline logical division. A generic element with no semantic meaning used to distinguish a document section, usually for purposes such as presentation or behaviour controlled by stylesheets or DOM calls.

<br>

A forced line-break.

<p>This is line 1<br>This is line 2</p>

**Forms**

These elements can be combined into a form or in some instances used separately as user-interface controls; in the document, they can be simple HTML or used in conjunction with Scripts. HTML markup specifies the elements that make up a form, and the method by which it will be submitted.

<form action="url">…</form>

Creates a form. The **form** element specifies and operates the overall action of a form area, using the required **action** attribute.

<button>…</button>

A generic form button which can contain a range of other elements to create complex buttons.

<fieldset>…</fieldset>

A container for adding structure to forms. For example, a series of related controls can be grouped within a **fieldset**, which can then have a **legend** added in order to identify their function.

<input>

**input** elements allow a variety of standard form controls to be implemented.

**Tables**

<table>…</table>

Identifies a table. Several attributes are possible in HTML Transitional, but most of these are invalid in HTML Strict and can be replaced with stylesheets.

<tr>…</tr>

Contains a row of cells in a table.

<th>…</th>

A table header cell; contents are conventionally displayed bold and centered. An aural user agent may use a louder voice for these items.

<td>…</td>

A table data cell.

<colgroup>…</colgroup>

Specifies a column group in a table.

<col> or <col/>

Specifies a column in a table.

<caption>…</caption>

Specifies a caption for a table.

<thead>…</thead>

Specifies the header part of a table. This section may be repeated by the user agent if the table is split across pages (in printing or other paged media).

<tbody>…</tbody>

Specifies a body of data for the table.

<tfoot>…</tfoot>

Specifies the footer part of a table. Like <thead>, this section may be repeated by the user agent if the table is split across pages (in printing or other paged media).

**Frames**

Frames allow a visual HTML Browser window to be split into segments, each of which can show a different document. This allows for lower bandwidth use, as repeating parts of a layout can be used in one frame, while variable content is displayed in another. This comes at a significant usability cost, especially in non-visual user agents. Because of this cost, frames (excluding the iframe element) are only allowed in HTML and XHTML Frameset Document Type.

<frameset>…</frameset>

Contains the frameset. The frames layout is given by comma separated lists in the rows and cols attributes.

<frame> or <frame/>

Delimits a single frame, or region, within the frameset. A separate document linked with the src attribute appears inside.

<noframes>…</noframes>

Contains normal HTML content for user agents that don't support frames.

<iframe>…</iframe>

An inline frame places another HTML document in a frame. Unlike an object element, an inline frame can be the "target" frame for links defined by other elements and it can be selected by the user agent as the focus for printing, viewing its source etc.

* + 1. Comments

A comment can appear anywhere in a document, even before the doctype, but not in other tags. (However, placing comments – or indeed any characters except for whitespace – before the doctype will cause Internet Explorer 6 to use quirks mode for the document).

<!-- A Comment -->

Comments do not nest.

For complete reference to HTML elements, please refer to the W3C.

<http://www.w3.org/TR/REC-html40/index/elements.html>

* 1. Document Type Definition (DTD)

Each HTML document must begin with a *document type declaration* that declares which version of HTML the document adheres to. The DTD

HTML 4 comes in three flavors, each with a different **DOCTYPE**.

**HTML 4 Strict**

HTML 4 Strict is a trimmed down version of HTML 4 that emphasizes structure over presentation. Deprecated elements and attributes (including most presentational attributes), frames, and link targets are not allowed in HTML 4 Strict.

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">

**HTML 4 Transitional**

HTML 4 Transitional includes all elements and attributes of HTML 4 Strict but adds presentational attributes, deprecated elements, and link targets.

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
"http://www.w3.org/TR/html4/loose.dtd">

**HTML 4 Frameset**

HTML 4 Frameset is a variant of HTML 4 Transitional for documents that use frames.

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"  
"http://www.w3.org/TR/html4/frameset.dtd">

* 1. Introduction to XHTML

The XHTML (Extensible Hypertext Markup Language) is a stricter and cleaner version of HTML. The HTML is a combination of HTML and XML which consists of all the elements in HTML 4.01, combined with the strict syntax of XML.

1. 4. 1. Syntax Differences

* The XML rules require that all elements be closed, either by a separate closing tag or using self closing syntax (e.g. <br/>), while HTML syntax permits some elements to be unclosed because either they are always empty (e.g. <input>) or their end can be determined implicitly ("omissibility", e.g. <p>).
* XML is case-sensitive for element and attribute names, while HTML is not.
* Some shorthand features in HTML are omitted in XML, such as :
* *attribute minimization*, where attribute values or their quotes may be omitted (e.g. <option selected> or <option selected=selected>, while XML this must be expressed as <option selected="selected">);
* *element minimization* may be used to remove elements entirely (such as <tbody> inferred in a table if not given)
* There are numerous other technical requirements surrounding namespaces and precise parsing of whitespace and certain characters and elements. The exact parsing of HTML in practice has been undefined until recently.
  + 1. Behavioural Differences
* JavaScript processing is a little different. document.write() method is not working in XHTML pages so the scripts fail on pages served as XHTML (that is, using an XML MIME type)
* CSS is also applied slightly differently too, with case-sensitivity and some differences in handling of backgrounds on <html> and <body> in XHTML and HTML.
  + 1. XHTML 1.0 Document Type

DOCTYPE declaration is mandatory in XHTML document. Which will make the valid XHTML document consists of DOCTYPE declaration, <html> section (as document root), <head> section, and <body> section. There are three XHTML DTDs:

**XHTML 1.0 Strict**

XHTML Strict is structure strict and use separation style document to handle the presentation.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

**XHTML 1.0 Transitional**

XHTML Transitional is follow XML syntax rules but still allow the HTML’s presentation features.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

**XHTML 1.0 Frameset**

XHTML 1.0 Frameset is a variant of XHTML 1.0 Transitional for documents that use frames.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

* 1. HTML Semantics

Semantics simply means trying to make sure your documents mean something, even if CSS is not available, even if your document is being displayed on a device that does not use the same default styles as you expected, or if it is being interpretted through a non-visual medium, such as braille or speech.

The idea is simple. Use the right elements for the right tasks. Browsers understand what those elements mean, and they can use many different techniques to convey that meaning to the user. But that only works if you use the elements the way they were meant to be used.

**Headings**

Use heading elements for headings: H1, H2, H3, H4, H5, H6. Don’t use CSS, the STRONG element or other markup to fake your headings.

**Paragraphs: the P Element**

Use the P element for paragraphs. Don’t use the <br /> element to instead provide paragraph-like breaks.

**Line Breaks: the BR Element**

The BR element is one of the more abused HTML elements around. When other HTML elements and CSS are understood and used appropriately, forced line breaks with the BR element can be minimized. For example:

* Use the P element for paragraphs, not the BR element.
* Use list item elements for lists (UL, OL, LI) rather than marking up a series of items with the BR element, and remember that you can use CSS to hide the HTML bullets if you wish.
* Use CSS to set widths, margins, and padding for text rather than using a multitude of <br /> tags for text line breaks.

**Emphasis: the EM Element**

Use the EM element for emphasis. If you’re after italicized text aside from emphasized text or citations use CSS (font-style:italic) rather than EM or I. Use the CITE element for citing a source.

**List Elements: UL, OL, LI**

When you have a list of something, use the list element tags. While there are a few choices of list style types, these can be replaced with images using CSS. It’s also possible to hide bullets completely, change indenting, use lists inline or block, and more.

**Definition List Elements: DL, DT, DD**

Use definition list elements for terms and their corresponding descriptions. Another possible use is for marking up dialogues, with each DT element naming a speaker followed by the DD element containing the speaker’s words.

By using the right elements, you have the benefit that without CSS, the browser will display it emphasised in some way. Some text based browsers may use bold or underline, speech browsers may say that part with a little more stress or volume, but in all cases, the browser can use that information to tell the user that the text is emphasised.

So you can see how using the right elements in a sentence is a useful approach. But it does not end there. It is equally impotant to use the right levels of headings in the right places, and not to use other elements to replace headings. If you use proper headings, some browsers will even allow users to jump from heading to heading. This is only possible if you actually use proper headings.

There is no single rule for how to make a semantic document, but just remember that HTML has a lot of element types available, and whenever you think that you would like to make a part of the document look or behave in a certain way, take a look at the [list of available elements](http://www.w3.org/TR/html401/index/elements.html), and use the one that suits the purpose for what that part of the page represents. If you want to make it look a specific way, style it with CSS, and leave the HTML free to denote what the parts of the page represent, instead of how they should look.

* 1. HTML Validation

As well as being semantically marked up, an (X)HTML document needs to be written using valid code and follow the standard. If the code is invalid, browsers will try to interpret the markup themselves, sometimes getting it wrong. Worse still, if an XHTML document is being sent with the correct MIME type, browsers that understand XML simply won’t display an invalid page.

Because browsers need to know which DTD to use in order to process the page correctly, a DOCTYPE declaration is required for the page to validate. You can check to see if your (X)HTML is valid by using the W3C validator, a validator bookmarklet, or a plug-in like the Firefox Developer Extension. Many (X)HTML editors now have validators built in, and you can even install a copy of the W3C validator locally on your computer. The validator will tell you if your page validates, and if not, why not.

Validation is important because it can help you track down bugs in your code. As such, it is a good idea to get into the habit of validating early and often. However, validation isn’t an end unto itself, and many otherwise good pages fail to validate due to small errors such as unencoded ampersands, or because of legacy content.

So although validation is important, in the real world, a degree of common sense is required.

<http://validator.w3.org/>

* 1. HTML Today - HTML 5

Today the new HTML version was born. HTML5 is a cooperation between the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG). WHATWG was working with web forms and applications, and W3C was working with XHTML 2.0. In 2006, they decided to cooperate and create a new version of HTML.

So what is HTML5, really? Basically, it’s about extending HTML/XHTML with new more semantically rich elements, deprecating attributes, introducing new attributes and altering how some element and attributes are allowed to be used.

* 2. 1. Simple DTD

HTML5 offer simple Document Type. No versioning, no redundant information. All progressive enhancement on a feature-level, as opposed to complete releases that need to be implemented. And, to be a bit more pragmatic than before, HTML5 will allow both quick-closing of empty tags (such as input, link etc), but you can use those elements just as well without quick-closing them. Quotes for attributes are also optional and you can even use upper-case letters for your element names.

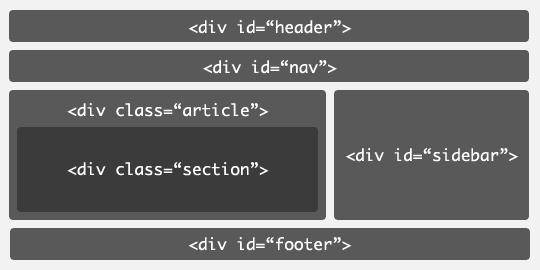
<!DOCTYPE html>

* + 1. HTML and XHTML

Two options to serve HTML5 content: as HTML or XHTML. The somewhat confusingly named XHTML5 differs a bit from HTML5, though:

* No doctype is needed, just an XML prolog.
* It must have a namespace: <html xmlns="http://www.w3.org/1999/xhtml">
* It *must* be served with either of these MIME types: application/xhtml+xml or application/xml.
* The noscript element can not be used.
  + 1. New way to build your structure

HTML 5 introduces a whole set of new elements that make it much easier to structure pages. Most HTML 4 pages include a variety of common structures, such as headers, footers and columns and today, it is fairly common to mark them up using div elements, giving each a descriptive id or class.



The use of div elements is largely because current versions of HTML 4 lack the necessary semantics for describing these parts more specifically. HTML 5 addresses this issue by introducing new elements for representing each of these different sections.

|  |  |
| --- | --- |
|  | <body>  <header>...</header>  <nav>...</nav>  <article>  <section>  ...  </section>  </article>  <aside>...</aside>  <footer>...</footer>  </body> |

* + 1. New Elements

**New Markup Elements**

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <article> | For external content, like text from a news-article, blog, forum, or any other content from an external source |
| <aside> | For content aside from the content it is placed in. The aside content should be related to the surrounding content |
| <command> | A button, or a radiobutton, or a checkbox |
| <details> | For describing details about a document, or parts of a document |
| <summary> | A caption, or summary, inside the details element |
| <figure> | For grouping a section of stand-alone content, could be a video |
| <figcaption> | The caption of the figure section |
| <footer> | For a footer of a document or section, could include the name of the author, the date of the document, contact information, or copyright information |
| <header> | For an introduction of a document or section, could include navigation |
| <hgroup> | For a section of headings, using <h1> to <h6>, where the largest is the main heading of the section, and the others are sub-headings |
| <mark> | For text that should be highlighted |
| <meter> | For a measurement, used only if the maximum and minimum values are known |
| <nav> | For a section of navigation |
| <progress> | The state of a work in progress |
| <ruby> | For ruby annotation (Chinese notes or characters) |
| <rt> | For explanation of the ruby annotation |
| <rp> | What to show browsers that do not support the ruby element |
| <section> | For a section in a document. Such as chapters, headers, footers, or any other sections of the document |
| <time> | For defining a time or a date, or both |

**New Media Elements**

HTML5 includes support for playing back audio and video files in the browser without the use of plugins such as Adobe Flash:

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <audio> | For multimedia content, sounds, music or other audio streams |
| <video> | For video content, such as a movie clip or other video streams |
| <source> | For media resources for media elements, defined inside video/audio elements.  Multiple ***source*** elements can link to different video/audio files. The browser will use the first recognized format. |
| <embed> | For embedded content, such as a plug-in |

There are some attributes for video/audio as below:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| autoplay | If present, this attribute causes the browser to start playing the video as soon as it is able to do so |
| preload | Tells the browser whether or not to load the video in advance. Three possible values are:   * None: The video will not be loaded until the user starts playback. * Metadata: Only the metadata for the video (width, height, first frame, duration, and other such information) should be loaded before the user starts playback. * Auto: Requests that the browser download the video in its entirety as soon as possible (this is the default behavior). |
| controls | The browser will not display controls unless this attribute is present |
| loop | If present, this attribute tells the browser to repeat the video |
| poster | Specifies an image to display when the video data is being loaded |
| height | Specifies the height of the video |
| width | Specifies the width of the video |
| muted | If this attribute is present, the video will be muted initially |
| src | Specifies the video resource to display |

Example script to load video:

<video width="360" height="240" src="myvideo.mp4"

autoplay controls preload="none" muted>

Video cannot be displayed (this is a fallback content)

</video>

Note: If the browser doesn’t support the video element or cannot play the video, the fallback content will be displayed instead.

**New Canvas and SVG Elements**

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <canvas> | For making graphics with a script |
| <svg> | Container for Scalable Vector Graphics (SVG) graphics |

**New Form Elements and Input Type Attribute Values**

See section 2.7.6.1: New HTML5 Form and Input Type

* + 1. Working with Forms in HTML5

Forms are the HTML mechanism for gathering input from the user. Forms are incredibly important to web applications, but for many years the functionality defined in HTML has lagged behind the way forms are used. In HTML5, the entire form system has been overhauled and aligning the standard with the way forms have evolved in use.

In this section, we will cover some important form features in HTML5 changes that have attracted the most attention — the new ways of gathering specific data types from the user, and the ability to validate the data in the browser.

* + - 1. New HTML5 Form Element and Input Type

HTML5 introduces 3 form element as belows:

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <datalist> | A list of options for input values |
| <keygen> | Generate keys to authenticate users |
| <output> | For different types of output, such as output written by a script |

Also HTML5 introduces some new input types/values for the input element’s type attribute that let you be more specific about the kind of data that you want from the user as below:

|  |  |
| --- | --- |
| **Type** | **Description** |
| tel | The input value is of type telephone number |
| search | The input field is a search field |
| url | The input value is a URL |
| email | The input value is one or more email addresses |
| datetime | The input value is a date and/or time |
| date | The input value is a date |
| month | The input value is a month |
| week | The input value is a week |
| time | The input value is of type time |
| datetime-local | The input value is a local date/time |
| number | The input value is a number |
| range | The input value is a number in a given range |
| color | The input value is a hexadecimal color, like #FF8800 |

|  |
| --- |
| **Note:** Notall browser can support the HTML5 input type. Please check the browser support information in <http://caniuse.com/> |

* + - 1. Additional Attributes for Input Type

Here is a list of some common additional attributes (some are new in HTML5):

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **New in HTML5** |
| autocomplete | Specifies whether an element should have autocomplete enabled | Yes |
| autofocus | Specifies that an element should automatically get focus when the page loads | Yes |
| disabled | Specifies that the element should be disabled (i.e. un-clickable). Disabled means that no data from that form element will be submitted when the form is submitted | No |
| dirname | Specifies a value for the name of the directionality of an element | No |
| list | Specifies the id of a datalist element that provides values for this element | Yes |
| max | Specifies the maximum value for an input field | Yes |
| maxlength | Specifies the maximum number of characters that the user can enter into an element | No |
| min | Specifies the minimum value for an input field | Yes |
| multiple | When present, it specifies that the user is allowed to enter more than one value in an element. This attribute applies when the type attribute is set to *email* or *file*; otherwise it is ignored. | Yes |
| pattern | Specifies a regular expression pattern for the purposes of input validation | Yes |
| placeholder | Specifies a hint to the user as to the kind of input that you expect. This attribute applies when the value of the type attribute is *text*, *search*, *tel*, *url*, *email* or *password*; otherwise it is ignored | Yes |
| readonly | If present, this attribute makes user cannot edit the content. The difference between disabled is, Readonly means any data from within the element will be submitted, but it cannot be changed by the user. | No |
| required | Specifies that the user must enter a value for the purposes of input validation | Yes |
| size | Specifies the width of the element, expressed as the number of characters that are visible in an element | No |
| Value | Specifies the initial value for an element. For the *email* type, this can be a single address, or multiple addresses  separated by commas. | No |

|  |
| --- |
| **Note:** Notall browser can support the HTML5 attributes. Please check the browser support information in <http://caniuse.com/> |

* + - 1. Form Input Validation

As you can see in previous section, HTML5 introduces many new attributes to support input validation, which is where you provide the browser with some basic information about the kind of data you require. The browser uses this information to check that the user has entered usable data before the form is submitted. If the data is problematic, the user is prompted to correct the problem and can’t submit the form until the issue is resolved.

The benefit of input validation in the browser is that the user gets immediate feedback about problems. Without this feature, the user has to submit the form, wait for the server to respond, and then deal with any problems that are reported. On a low-performing network and an over-utilized server, this can be a slow and frustrating process.

|  |
| --- |
| Caution: Input validation in the browser complements, rather than replaces, validation at the server. You cannot rely on users to employ browsers that properly support input validation, and it is a small matter for a malicious user to craft a script that will send input directly to your server without any form of validation at all. |

In next section we will see some validation examples.

**The *required* Attribute**

The simplest kind of input validation is to ensure that the user provides a value. You do this with the *required* attribute. The user cannot submit the form until a value has been provided. The required attribute works with the following input types: *text, search, url, tel, email, password, date pickers, number, checkbox, radio,* and *file.*

Example:

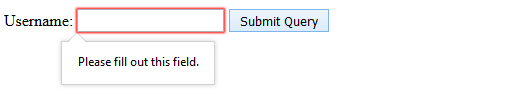
<form action="action\_page.php">

Username: <input type="text" name="usrname" required>

<input type="submit">

</form>

Each browser that supports input validation does so in a slightly different way, but the effect is much the same: when the user clicks the button to submit the form, the first element that has the required attribute and that does not have a value is flagged for the user’s attention. The user can then correct the omission and submit the form again. If there are other omissions, then the next problem element is flagged. The process continues until the user has provided a value for all of the elements with the required attribute as below sreenshot:



**Numeric example**

You create an upper/lower/step limit for the input as below example:

<form>  
  Quantity:  
  <input type="number" name="points" **min="0" max="100" step="10" value="30"**>  
</form>

The screenshot for above numeric example:



**Email example**

You can create hint and pattern for email as below example:

<form>  
  E-mail:  
 <input type="email" name="email" id="email" **placeholder="user@mydomain.com" pattern=".\*@mydomain.com$"** required>  
</form>

Above example used three of the validation features:

* The *email* type of the input element ensures that use enters a valid e-mail address.
* The *required* attribute ensures that the user provides a value.
* The *pattern* attribute ensures that the user enters an e-mail address that belongs to a specific domain (*mydomain.com* in this example).

The use of the *email* input type and the *pattern* attribute might seem redundant, but the input element is still responsible for ensuring that everything before the @ character is valid as an e-mail address.

Let’s see the user action screenshot below for the detail:

|  |  |  |
| --- | --- | --- |
| **No** | **Action** | **Result and screenshot** |
| 1 | Initial screen |  |
| 2 | User do not enter any value and just click Submit |  |
| 3 | User enter some characters and and click Submit |  |
| 4 | User enter email address but not match the pattern. |  |
| 5 | User enter valid email and match the pattern.  In this case, no error and form is submitted successfully. |  |

* + 1. HTML5 Graphics

HTML5 provides a feature to draw graphic elements, produce animations or visual effects. Previously, this kind of task is handled by the external plugins such as swift or flash that embedded to the document.

HTML5 supports drawing graphic elements via Canvas or SVG. Canvas draws 2D graphics on the fly with help from JavaScript, and SVG draws 2D graphics using XML language.

* + - 1. Canvas

The HTML <canvas> element is used to draw graphics, on the fly, via scripting (usually JavaScript).The <canvas> element is only a container for graphics. You must use a script to actually draw the graphics. Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

A canvas is a rectangular area on an HTML page. By default, a canvas has no border and no content. The example of the basic canvas tag:

<canvas id="myCanvas"

width="200"

height="100"

style="border:1px solid #000000;">

</canvas>

**Note:** Always specify an id attribute (to be referred to in a script), and a width and height attribute to define the size of the canvas.

The code above will display a rectangle canvas. To draw something inside the canvas, it requires some javascript code. For example, here is the code to draw a diagonal line inside the myCanvas:

var c = document.getElementById("myCanvas");

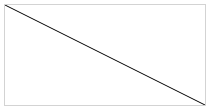
var ctx = c.getContext("2d");

ctx.moveTo(0,0);

ctx.lineTo(200,100);

ctx.stroke();

The code above will produce a canvas and a diagonal line like below:



Here is the explanation of each line in the javascript code:

var c = document.getElementById("myCanvas");

This line is to get the canvas element by using the HTML DOM method with id as the selector.

var ctx = c.getContext("2d");

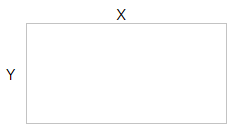
This line is to define the drawing object for the canvas, the getContext() is a built-in HTML function to get the canvas content.

ctx.moveTo(0,0);

ctx.lineTo(200,100);

For these two lines, it defines two coordinates that required to draw the line.

The HTML canvas is a two dimensional grid (*x*, *y*) where *x* is the horizontal dimension and *y* is the vertical dimension. The upper-left corner of the canvas has the coordinates (0, 0).



To draw a straight line on a canvas on the example, it uses the following methods:

* moveTo(x,y) - defines the starting point of the line.
* lineTo(x,y) - defines the ending point of the line .

ctx.stroke();

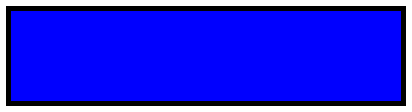
This line is to render a line element through the coordinate that specified in the previous lines.

**Note:** The full HTML Canvas syntax list is available in [MSDN](https://msdn.microsoft.com/en-us/library/hh771733(v=vs.85).aspx) website.   
Another reference is in [W3Schools](http://www.w3schools.com/tags/ref_canvas.asp) website.

* + - 1. SVG

SVG is a language for describing 2D graphics in XML. Therefore, SVG is XML based, which means that every element is available within the SVG DOM. JavaScript event handlers can be attached for an element. In SVG, each drawn shape is remembered as an object. If attributes of an SVG object are changed, the browser can automatically re-render the shape.

SVG has several methods for drawing paths, boxes, circles, text, and graphic images. Below is the example of drawing a rectangle using SVG tag in HTML5:



<svg width="400" height="100">

<rect width="400"

height="100"

style="fill:rgb(0,0,255);

stroke-width:10;

stroke:rgb(0,0,0)" />

</svg>

Different with the canvas that requires JavaScript to draw objects inside the canvas, in SVG it only needs the object specified in the HTML file.

**Note:** Comperehensive SVG examples available in the [W3Schools](http://www.w3schools.com/svg/svg_examples.asp) website along with the [SVG Reference](http://www.w3schools.com/svg/svg_reference.asp).

* + - 1. Comparison between Canvas and SVG

The table below shows some important differences between Canvas and SVG:

|  |  |
| --- | --- |
| **Canvas** | **SVG** |
| * Resolution dependent. * No support for event handlers. * Poor text rendering capabilities. * Can save the resulting image as .png or .jpg. * Well suited for graphic-intensive games. | * Resolution independent. * Support for event handlers. * Best suited for applications with large rendering areas (Google Maps). * Slow rendering if complex (anything that uses the DOM a lot will be slow). * Not suited for game applications. |

* + 1. API

It also introduces a number of [APIs for making it easier to create web applications](http://dev.w3.org/html5/html4-differences/#apis):

* 2D drawing API with the canvas element.
* API for playing of video and audio with the video and audio elements.
* API that enables offline Web applications.
* API that allows a Web application to register itself for certain protocols or media types.
* Editing API in combination with a new global contenteditable attribute.
* Drag & drop API in combination with a draggable attribute.
* API that exposes the history and allows pages to add to it to prevent breaking the back button.
* Cross-document messaging with postMessage.

Other things that initially was in the specification, but broken out into separate specifications are:

* API for Geolocation
* Web Storage.
* Web Workers.
* querySelectorAll.
  + - 1. Geolocation API

The Geolocation API allows us to obtain information about the current geographic position of the user (or at least the position of the system on which the browser is running). This isn’t part of the HTML5 specification, but it is usually grouped up as part of the new features associated with HTML5.

We access the geolocation feature through the global navigator.geolocation property, which returns a Geolocation object. Below is the methods of Geolocation object:

|  |  |  |
| --- | --- | --- |
| Name | Description | Returns |
| **GetCurrentPosition(callback, errorCallback, options)** | Get the current position | void |
| **watchPosition(callback, error, options)** | Start monitoring the current poition | number |
| **ClearWatch(id)** | Stop monitoring the current position | void |

* + - * 1. Getting the Current Position

GetCurrentPosition method isn’t returned as the result of the method itself. Instead, we supply a success callback function which is invoked when the position information is available – there can be a delay between requesting the position.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Example</title>  <style>  table {  border-collapse: collapse;  }  th, td {  padding: 4px;  }  th {  text-align: right;  }  </style>  </head>  <body>  <table border="1">  <tr>  <th>Longitude:</th>  <td id="longitude">-</td>  <th>Latitude:</th>  <td id="latitude">-</td>  </tr>  <tr>  <th>Altitude:</th>  <td id="altitude">-</td>  <th>Accuracy:</th>  <td id="accuracy">-</td>  </tr>  <tr>  <th>Altitude Accuracy:</th>  <td id="altitudeAccuracy">-</td>  <th>Heading:</th>  <td id="heading">-</td>  </tr>  <tr>  <th>Speed:</th>  <td id="speed">-</td>  <th>Time Stamp:</th>  <td id="timestamp">-</td>  </tr>  </table>  <script>  navigator.geolocation.getCurrentPosition(displayPosition);  function displayPosition(pos) {  var properties = ["longitude", "latitude", "altitude", "accuracy", "altitudeAccuracy", "heading", "speed"];  for (var i = 0; i < properties.length; i++) {  var value = pos.coords[properties[i]];  document.getElementById(properties[i]).innerHTML = value;  }  document.getElementById("timestamp").innerHTML = pos.timestamp;  }  </script>  </body> </html> |

When the position information is available, the nominated function is invoked and the browser passes in a Position object which gives the details of the position.

The table below describes the Position object property:

|  |  |  |
| --- | --- | --- |
| Name | Description | Returns |
| **Cords** | Returns the coordinates for the current position | Coordinates |
| **Timestamp** | Returns the time that the coordinate information was obtained | string |

The Coordinates object has the following property:

|  |  |  |
| --- | --- | --- |
| Name | Description | Returns |
| **latitude** | Returns the latitude in decimal degrees | Number |
| **longitude** | Returns the longitude in decimal degrees | Number |
| **altitude** | Returns the height in meters | Number |
| **accuracy** | Returns the accuracy of the coordinates in meters | Number |
| **altitudeAccuracy** | Returns the accuracy of the altitude in meters | Number |
| **heading** | Returns the direction of travel in degrees | Number |
| **speed** | Returns the speed of travel in meters/second | Number |

|  |
| --- |
| **Note**: API geolocation only works if the computer has internet access or it will produce error. The first thing that all of the browsers do when there is a geolocation feature is ask the user to grant permission. Deny the permission will produce error. Try to run the html markup above in WC3 testing code machine <http://www.w3schools.com/css/tryit.asp?filename=trycss_default>.  If the computer doesn’t have GPS, compass, altimeter or accelerometer, the only data that is avaialbe is latitude, longitude and accuracy. |

* + - * 1. Handling Geolocation Errors

We can provide a second argument to the getCurrentPosition method, which allows us to supply a function that will be invoked if there is an error obtaining the location. The function is passed a PositionError object, which defines the properties as described below:

|  |  |  |
| --- | --- | --- |
| Name | Description | Returns |
| **code** | Returns a code indicating the type of error | Number |
| **message** | Returns a string that describes the error | string |

There are three possible values for the code property:

|  |  |
| --- | --- |
| Value | Description |
| **1** | The user did not grant permission to user the geolocation feature |
| **2** | The position could not be determined |
| **3** | The attempt to request the location timed out |

|  |
| --- |
| <html>  <head>  <title>Example</title>  <style>  table {  border-collapse: collapse;  }  th, td {  padding: 4px;  }  th {  text-align: right;  }  </style>  </head>  <body>  <table border="1">  <tr>  <th>Longitude:</th>  <td id="longitude">-</td>  <th>Latitude:</th>  <td id="latitude">-</td>  </tr>  <tr>  <th>Altitude:</th>  <td id="altitude">-</td>  <th>Accuracy:</th>  <td id="accuracy">-</td>  </tr>  <tr>  <th>Altitude Accuracy:</th>  <td id="altitudeAccuracy">-</td>  <th>Heading:</th>  <td id="heading">-</td>  </tr>  <tr>  <th>Speed:</th>  <td id="speed">-</td>  <th>Time Stamp:</th>  <td id="timestamp">-</td>  </tr>  **<tr>**  **<th>Error Code:</th>**  **<td id="errcode">-</td>**  **<th>Error Message:</th>**  **<td id="errmessage">-</td>**  **</tr>**  </table>  <script>  navigator.geolocation.getCurrentPosition(displayPosition, **handleError**);  function displayPosition(pos) {  var properties = ["longitude", "latitude", "altitude", "accuracy",  "altitudeAccuracy", "heading", "speed"];  for (var i = 0; i < properties.length; i++) {  var value = pos.coords[properties[i]];  document.getElementById(properties[i]).innerHTML = value;  }  document.getElementById("timestamp").innerHTML = pos.timestamp;  }  **function handleError(err) {**  **document.getElementById("errcode").innerHTML = err.code;**  **document.getElementById("errmessage").innerHTML = err.message;**  **}**  </script>  </body> </html> |

* + - * 1. Specifying Geolocation Options

The third argument we can provide to the getCurrentPosition method is a PositionOptions object. This feature allows us to exert some control over the way that locations are obtained.

The table below shows the properties of PositionOPtions object:

|  |  |  |
| --- | --- | --- |
| Name | Description | Returns |
| **enableHighAccuracy** | Tells the browser that we would like the best possible result - there are no guarantees that it will lead to a more accurate location | Boolean |
| **timeout** | Returns a string that describes the error | Number |
| **maximumAge** | Tells the browser that we are willing to accept a cached location, as long as it is no older than the specified number of milliseconds | Number |

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Example</title>  <style>  table {  border-collapse: collapse;  }  th, td {  padding: 4px;  }  th {  text-align: right;  }  </style>  </head>  <body>  <table border="1">  <tr>  <th>Longitude:</th>  <td id="longitude">-</td>  <th>Latitude:</th>  <td id="latitude">-</td>  </tr>  <tr>  <th>Altitude:</th>  <td id="altitude">-</td>  <th>Accuracy:</th>  <td id="accuracy">-</td>  </tr>  <tr>  <th>Altitude Accuracy:</th>  <td id="altitudeAccuracy">-</td>  <th>Heading:</th>  <td id="heading">-</td>  </tr>  <tr>  <th>Speed:</th>  <td id="speed">-</td>  <th>Time Stamp:</th>  <td id="timestamp">-</td>  </tr>  <tr>  <th>Error Code:</th>  <td id="errcode">-</td>  <th>Error Message:</th>  <td id="errmessage">-</td>  </tr>  </table>  <script>  **var options = {**  **enableHighAccuracy: false,**  **timeout: 2000,**  **maximumAge: 30000**  **};**  navigator.geolocation.getCurrentPosition(displayPosition,  handleError, **options**);  function displayPosition(pos) {  var properties = ["longitude", "latitude", "altitude", "accuracy", "altitudeAccuracy", "heading", "speed"];  for (var i = 0; i < properties.length; i++) {  var value = pos.coords[properties[i]];  document.getElementById(properties[i]).innerHTML = value;  }  document.getElementById("timestamp").innerHTML = pos.timestamp;  }  function handleError(err) {  document.getElementById("errcode").innerHTML = err.code;  document.getElementById("errmessage").innerHTML = err.message;  }  </script>  </body>  </html> |

* + - * 1. Monitoring the Position

This method takes the same arguments as the getCurrentPosition method and works in the same way – the difference is that the callback functions in this method will be repeatedly called when the position changes.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Example</title>  <style>  table {  border-collapse: collapse;  }  th, td {  padding: 4px;  }  th {  text-align: right;  }  </style>  </head>  <body>  <table border="1">  <tr>  <th>Longitude:</th>  <td id="longitude">-</td>  <th>Latitude:</th>  <td id="latitude">-</td>  </tr>  <tr>  <th>Altitude:</th>  <td id="altitude">-</td>  <th>Accuracy:</th>  <td id="accuracy">-</td>  </tr>  <tr>  <th>Altitude Accuracy:</th>  <td id="altitudeAccuracy">-</td>  <th>Heading:</th>  <td id="heading">-</td>  </tr>  <tr>  <th>Speed:</th>  <td id="speed">-</td>  <th>Time Stamp:</th>  <td id="timestamp">-</td>  </tr>  <tr>  <th>Error Code:</th>  <td id="errcode">-</td>  <th>Error Message:</th>  <td id="errmessage">-</td>  </tr>  </table>  **<button id="pressme">Cancel Watch</button>**  <script>  var options = {  enableHighAccuracy: false,  timeout: 2000,  maximumAge: 30000  };  **var watchID = navigator.geolocation.watchPosition(displayPosition,**  **handleError,**  **options);**  **document.getElementById("pressme").onclick = function (e) {**  **navigator.geolocation.clearWatch(watchID);**  **};**  function displayPosition(pos) {  var properties = ["longitude", "latitude", "altitude", "accuracy",  "altitudeAccuracy", "heading", "speed"];  for (var i = 0; i < properties.length; i++) {  var value = pos.coords[properties[i]];  document.getElementById(properties[i]).innerHTML = value;  }  document.getElementById("timestamp").innerHTML = pos.timestamp;  }  function handleError(err) {  document.getElementById("errcode").innerHTML = err.code;  document.getElementById("errmessage").innerHTML = err.message;  }  </script>  </body>  </html> |

From the above code, the script uses the watchPosition method to monitor the location. This method returns an ID value which we can pass to the clearWatch method when we want to stop monitoring. By the clearWatch, when the position is updated, it will not repeatly call the function fallback.

* + - 1. Drag & Drop API

HTML5 adds support for drag and drop. We tell the browser which elements in the document can be dragged through the **draggable** attribute. There are three permitted values for this attribute, which are described below:

|  |  |
| --- | --- |
| Value | Description |
| **true** | Element can be dragged |
| **False** | Element can’t be dragged |
| **Auto** | Browser may decide if an element can be dragged. This is the default value. |

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Example</title>  <style>  #src > \* {  float: left;  }  #target, #src > img {  border: thin solid black;  padding: 2px;  margin: 4px;  }  #target {  height: 81px;  width: 81px;  text-align: center;  display: table;  }  #target > p {  display: table-cell;  vertical-align: middle;  }  #target > img {  margin: 1px;  }  </style>  </head>  <body>  <div id="src">  <img draggable="true" id="banana" src="banana100.png" alt="banana" />  <img draggable="true" id="apple" src="apple100.png" alt="apple" />  <img draggable="false" id="cherries" src="cherries100.png" alt="cherry" />  <div id="target">  <p>Drop Here</p>  </div>  </div>  <script>  var src = document.getElementById("src");  var target = document.getElementById("target");  </script>  </body>  </html> |

From the testing code above, user will not be able to drag the banana image. We can drag the fruit images without doing any further work, but the browser will indicate that we can’t drop them anywhere. This is usually done by showing a no-entry sign as the cursor.

* + - * 1. Handling the Drag Events

These are events that are targeted at the dragged element and events that are targeted at potential drop zones, as described below:

|  |  |
| --- | --- |
| Value | Description |
| **dragstart** | Triggered when the element is first dragged |
| **drag** | Triggered repeatedly as the element is being dragged |
| **dragend** | Triggered when the drag operation is completed |

|  |
| --- |
| …  #target > img {  margin: 1px;  }  **img.dragged {**  **background-color: lightgrey;**  **}**  </style>  </head>  …  <div id="target">  **<p id="msg">Drop Here</p>**  </div>  </div>  <script>  var src = document.getElementById("src");  var target = document.getElementById("target");  **var msg = document.getElementById("msg");**  **src.ondragstart = function(e) {**  **e.target.classList.add("dragged");**  **}**  **src.ondragend = function(e) {**  **e.target.classList.remove("dragged");**  **msg.innerHTML = "Drop Here";**  **}**  **src.ondrag = function (e) {**  **msg.innerHTML = e.target.id;**  **}**  </script>  </body>  </html> |

From the above testing code, the image is added a new style when image has been dragged in response to the dragstart event and remove the style in response to the dragend event. In response to the drag event, the text displayed in the drop zone. The drag event is called every few milliseconds during the drag operation.

* + - * 1. Creating the Drop Zone

To make an element a drop zone, we need to handle the dragenter and dragover events. These are the events which are targeted at the drop zone:

|  |  |
| --- | --- |
| Value | Description |
| **dragenter** | Triggered when a dragged element enters the screen space occupied by the drop zone |
| **dragover** | Triggered when a dragged element moves within the drop zone |
| **dragleave** | Triggered when a dragged element leveas the drop zone without being dropped |
| **drop** | Triggered when a dragged element is dropped in the drop zone |

The default action for the dragenter and dragover events is to refuse to accept any dragged items, so the most important thing we must do is prevent the default action from being performed.

|  |
| --- |
| …  <script>  var src = document.getElementById("src");  var target = document.getElementById("target");  var msg = document.getElementById("msg");  **target.ondragenter = handleDrag;**  **target.ondragover = handleDrag;**  **function handleDrag(e) {**  **e.preventDefault();**  **}**  src.ondragstart = function(e) {  e.target.classList.add("dragged");  }  … |

With these additions, we have an active drop zone. When we drag an item over the drop zone element, the browser will indicate that it will be accepted if we drop it, but the drop zone still can’t accept it before we handle the drop event.

* + - * 1. Receiving the Drop

We receive the dropped element by handling the drop event, which is triggered when an item is dropped.

|  |
| --- |
| …  <script>  var src = document.getElementById("src");  var target = document.getElementById("target");  var msg = document.getElementById("msg");  **var draggedID;**  target.ondragenter = handleDrag;  target.ondragover = handleDrag;  function handleDrag(e) {  e.preventDefault();  }  **target.ondrop = function (e) {**  **var newElem = document.getElementById(draggedID).cloneNode(false);**  **target.innerHTML = "";**  **target.appendChild(newElem);**  **e.preventDefault();**  **}**  src.ondragstart = function (e) {  **draggedID = e.target.id;**  e.target.classList.add("dragged");  }  … |

The above code is showing that the drop image will be cloned based on the id. The Image id has been kept when the image starts to be dragged.

**Note**: The code above prevented the default action for the drop event. Without this, the browser can do some unexpected things.

* + - * 1. Working with the DataTransfer Object

The object dispatched along with the events triggered for drag and drop is DragEvent, which is derived from MouseEvent. The DragEvent object defines all of the functionality of the Event and MouseEvent objects with the additional property shown below:

|  |  |  |
| --- | --- | --- |
| Value | Description | Returns |
| **dataTransfer** | Returns the object used to transfer data to the drop zone | DataTransfer |

We use the DataTransfer object to transfer arbitrary data from the dragged element to the drop zone element. The properties and methods that the DataTransfer object defines are described below:

|  |  |  |
| --- | --- | --- |
| Value | Description | Returns |
| **types** | Returns the formats for the data | String[] |
| **getData(<format>)** | Returns the data for a specific format | string |
| **setData (<format>, <data>)** | Sets the data for a given format | void |
| **clearData(<format>)** | Removes the data for a given format | void |
| **files** | Returns a list of the files that have been dragged | FileList |

In the previous example, I cloned the element itself; however, the DataTransfer object allows us a

more sophisticated approach. The first thing we can do is to use the DataTransfer object to transfer data from the dragged element to the drop zone, as demonstrated by code below:

|  |
| --- |
| …  <script>  var src = document.getElementById("src");  var target = document.getElementById("target");  var msg = document.getElementById("msg");  target.ondragenter = handleDrag;  target.ondragover = handleDrag;  function handleDrag(e) {  e.preventDefault();  }  target.ondrop = function (e) {  **var droppedID = e.dataTransfer.getData("Text");**  **var newElem = document.getElementById(droppedID).cloneNode(false);**  target.innerHTML = "";  target.appendChild(newElem);  e.preventDefault();  }  src.ondragstart = function (e) {  **e.dataTransfer.setData("Text", e.target.id);**  e.target.classList.add("dragged");  }  … |

Use the setData method when responding to the dragstart event to set the data that we want to transfer. There are only two supported values for the first argument which specifies the type of data—Text or Url (and only Text is reliably supported by the browsers). The second argument is the data we want to transfer.

To retrieve the value, use the getData method, using the data type as the argument.

This is a better approach than using a global variable. Using DataTransfer can transfer element across tabs in the same browsers and even across different types of browser. Example: Open the html file in Chrome and FF browsers. We can drag the image from Chrome to the drop zone area in FF.

* + - * 1. Filtering Dragged Items by Data

We can use the data stored in the DataTransfer object to be selective about the kinds of elements that we are willing to accept in the drop zone.

|  |
| --- |
| …  function handleDrag(e) {  **if (e.dataTransfer.getData("Text") == "banana") {**  **e.preventDefault();**  **}**  }  … |

In this example, the dropdown accept the dragged element only if the data value is banana.

* + - * 1. Dragging and Dropping Files

File API is one of HTML5 feature which allow us to work with files. WE don’t usually interact with the file API directly. Instead, it is exposed through other features.

When user drops files on the drop zone, the files property of the DataTransfer object return a FileList object. We can treat this as an array of File objects, each of which represents a file that the user has dropped (the user can select multiple files and drop them in one go). Below is the properties of the file object:

|  |  |  |
| --- | --- | --- |
| Value | Description | Returns |
| **Name** | Gets the name of the file | String[] |
| **type** | Gets the type of file, expressed as a MIME type | String |
| **size** | Gets the size (in bytes) of the file | number |

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Example</title>  <style>  body > \* {  float: left;  }  #target {  border: medium double black;  margin: 4px;  height: 75px;  width: 200px;  text-align: center;  display: table;  }  #target > p {  display: table-cell;  vertical-align: middle;  }  table {  margin: 4px;  border-collapse: collapse;  }  th, td {  padding: 4px;  }  ;  </style>  </head>  <body>  <div id="target">  <p id="msg">Drop Files Here</p>  </div>  <table id="data" border="1"></table>  <script>  var target = document.getElementById("target");  target.ondragenter = handleDrag;  target.ondragover = handleDrag;  function handleDrag(e) {  e.preventDefault();  }  target.ondrop = function (e) {  **var files = e.dataTransfer.files;**  var tableElem = document.getElementById("data");  tableElem.innerHTML = "<tr><th>Name</th><th>Type</th><th>Size</th></tr>";  **for (var i = 0; i < files.length; i++) {**  **var row = "<tr><td>" + files[i].name + "</td><td>" +**  **files[i].type + "</td><td>" +**  **files[i].size + "</td></tr>";**  **tableElem.innerHTML += row;**  **}**  e.preventDefault();  }  </script>  </body>  </html> |

* + - * 1. Uploading Dropped Files in a Form

We can combine the drag and drop feature, the File API and uploading data using an Ajax request to allow users to drag the files that want included in a form submission.

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>Example</title>  <style>  .table {  display: table;  }  .row {  display: table-row;  }  .cell {  display: table-cell;  padding: 5px;  }  .label {  text-align: right;  }  #target {  border: medium double black;  margin: 4px;  height: 50px;  width: 200px;  text-align: center;  display: table;  }  #target > p {  display: table-cell;  vertical-align: middle;  }  </style>  </head>  <body>  <form id="fruitform" method="post" action="http://titan:8080/form">  <div class="table">  <div class="row">  <div class="cell label">Bananas:</div>  <div class="cell"><input name="bananas" value="2" /></div>  </div>  <div class="row">  <div class="cell label">Apples:</div>  <div class="cell"><input name="apples" value="5" /></div>  </div>  <div class="row">  <div class="cell label">Cherries:</div>  <div class="cell"><input name="cherries" value="20" /></div>  </div>  <div class="row">  <div class="cell label">File:</div>  <div class="cell"><input type="file" name="file" /></div>  </div>  <div class="row">  <div class="cell label">Total:</div>  <div id="results" class="cell">0 items</div>  </div>  </div>  <div id="target">  <p id="msg">Drop Files Here</p>  </div>  <button id="submit" type="submit">Submit Form</button>  </form>  <script>  var target = document.getElementById("target");  var httpRequest;  var fileList;  document.getElementById("submit").onclick = handleButtonPress;  target.ondragenter = handleDrag;  target.ondragover = handleDrag;  function handleDrag(e) {  e.preventDefault();  }  **target.ondrop = function (e) {**  **fileList = e.dataTransfer.files;**  **e.preventDefault();**  **}**  function handleButtonPress(e) {  e.preventDefault();  var form = document.getElementById("fruitform");  var formData = new FormData(form);  **if (fileList || true) {**  **for (var i = 0; i < fileList.length; i++) {**  **formData.append("file" + i, fileList[i]);**  **}**  **}**  httpRequest = new XMLHttpRequest();  httpRequest.onreadystatechange = handleResponse;  httpRequest.open("POST", form.action);  httpRequest.send(formData);  }  function handleResponse() {  if (httpRequest.readyState == 4 && httpRequest.status == 200) {  var data = JSON.parse(httpRequest.responseText);  document.getElementById("results").innerHTML = "You ordered "  + data.total + " items";  }  }  </script>  </body>  </html> |

* + 1. HTML5 Microdata

The HTML5 draft specification includes Microdata. The Microdata spec provides a standardized syntax for additional semantic markup to the web pages to enhance the machine readability of the web pages. The purpose is not to make a new widget appear on the web page but to help automated programs, like Google, understand and handle the content of the web pages better.

* + - 1. Overview

Microdata defines five HTML attributes that can be applied to any HTML5 tag. Most developers will only ever use itemscope, itemtype and itemprop. Itemref and itemid aren’t necessary to get up and running with microdata and aren’t needed by the most common formats.

* Itemscope - Indicates the element is a microdata element and its child elements are part of its microdata format.
* Itemtype - Defines the vocabulary to be used by the microdata format.
* Itemid - The unique identifier of the item, if defined by the microdata vocabulary.
* Itemprop - An individual data element.
* Itemref - Allows a microdata element to reference another element on the page to define it by either HTML id or by itemid.
  + - 1. Usage

At a high level, microdata consists of a group of name-value pairs. The groups are called items, and each name-value pair is a property. Items and properties are represented by regular elements. Below is the example of HTML that uses microdata to help the search engines understand the objects:

<html>

<body>

<div itemscope itemtype="http://www.data-vocabulary.org/Person">

<p>My name is <span itemprop="name">Zara</span>.</p>

</div>

<div itemscope itemtype="http://www.data-vocabulary.org/Person">

<p>My name is <span itemprop="name">Nuha</span>.</p>

</div>

</body>

</html>

In the example above, popular search engine, like Google, will easily recognize two Person objects, both has name property with value Zara and Nuha respectively.

* + - 1. Testing Microdata

Google supports microdata as part of their Rich Snippets program. When Google's web crawler parses the page and finds microdata properties that conform to the *http://data-vocabulary.org/Person* vocabulary, it parses out those properties and stores them alongside the rest of the page data.

Google provides a way to test the microdata using this (<https://developers.google.com/structured-data/testing-tool>) website. Try to test the macrodata provided in the previous section using the Google structured data tool and see how google reads the person objects on the HTML syntax.

* + 1. Browser Support

Essentially using certain elements of those elements; they will look nothing but intended to most of the view of the pages. So while HTML5 and its applications are definitely going to offer looks for cooler websites it is just not practical yet to use for web designs.

HTML5 was supported on Internet Explorer. 9.0. Firefox. 4.0. Safari. 4.0. Chrome. 3.0. Opera. 10.5. Opera Mobile. 11.0. iOS. 4.0. Android. 2.1

1. CSS

In the early, web was little more than a series of interlinked research document using HTML for the formatting and structure. However, as the WWW increased in popularity, HTML started being used for presentational purposes. Instead of using heading element for page headlines, people would use a combination of font and bold tags to create the visual effect they wanted. As web pages became more and more presentational, the code became almost impossible to understand and complicated WYSIWYG tools were needed to handle the codes.

Then along came ***Cascading Style Sheets***. With CSS it became possible to control how a page looked externally and to separate the presentational aspect of a document from its content. Presentational tags like the font tag could be ditched, and layout could be controlled using CSS instead of tables.

1. 1. Getting your styles to hit the target

A valid and well-structured document provides the framework to which your styles are applied. To be able to style a particular (X)HTML element using CSS, you need to have some way of targeting that element. In CSS the part of a style rule that does this is called the *selector*.

The following table summarizes CSS 2.1 selector syntax:

|  |  |
| --- | --- |
| **Pattern** | **Meaning** |
| \* | Matches any element. |
| E | Matches any E element (i.e., an element of type E). |
| E F | Matches any F element that is a descendant of an E element. |
| E > F | Matches any F element that is a child of an element E. |
| E:first-child | Matches element E when E is the first child of its parent. |
| E:link E:visited | Matches element E if E is the source anchor of a hyperlink of which the target is not yet visited (:link) or already visited (:visited). |
| E:active E:hover E:focus | Matches E during certain user actions. |
| E:lang(c) | Matches element of type E if it is in (human) language c (the document language specifies how language is determined). |
| E + F | Matches any F element immediately preceded by a sibling element E. |
| E[foo] | Matches any E element with the "foo" attribute set (whatever the value). |
| E[foo="warning"] | Matches any E element whose "foo" attribute value is exactly equal to "warning". |
| E[foo~="warning"] | Matches any E element whose "foo" attribute value is a list of space-separated values, one of which is exactly equal to "warning". |
| E[lang|="en"] | Matches any E element whose "lang" attribute has a hyphen-separated list of values beginning (from the left) with "en". |
| DIV.warning | Language specific. (In HTML, the same as DIV[class~="warning"].) |
| E#myid | Matches any E element with ID equal to "myid". |

* + 1. Type Selectors

The most common kinds of selectors are type and descendant selectors. Type selectors are used to target a particular type of element, such as a paragraph, an anchor, or a heading element. You do this by simply specifying the name of the element you wish to style. Type selectors are sometimes also referred to as element or simple selectors.

p {color: black;}

a {text-decoration: underline;}

h1 {font-weight: bold;}

Descendant selectors allow you to target the descendants of a particular element or group of elements. A descendant selector is indicated by a space between two other selectors. In this example, only anchor elements that are descendants of a list item will be styled, so anchors within a paragraph will be unaffected.

li a {text-decoration: none;}

* + 1. ID and Class Selectors

To be more specific and target selected elements, you can use ID and class selectors. As the names suggest, these selectors will target elements with the corresponding ID or class name. ID selectors are identified using a hash character; class selectors are identified with a period. The first rule in this example will make the text in the introductory paragraph bold, and the second rule will make the date green:

#intro {font-weight: bold;}

.datePosted {color: green;}

<p id="intro">Some Text</p>

<p class="datePosted">Sep 19, 2010</p>

* + 1. Universal Selectors

The universal selector is possibly one of the most powerful and least used of all the selectors. The universal selector acts like a wildcard, matching all the available elements. Like wildcards in other languages, the universal selector is denoted by an asterisk. The universal selector is normally used to style every element on a page. For instance, you can remove the default browser padding and margin on every element using the following rule:

\* {

padding: 0;

margin: 0;

}

* + 1. Advanced Selectors

CSS2 has a number of other useful selectors. Unfortunately, while modern browsers such as Firefox and Safari support these advanced selectors, IE 6 and below do not. Luckily, CSS was created with backward compatibility in mind. If a browser doesn’t understand a selector, it ignores the whole rule. That way, you can apply stylistic and usability embellishments in more modern browsers, and not worry about it causing problems in older browsers. Just remember to avoid using these more advanced selectors for anything critical to the functioning of your site.

**Child and adjacent sibling selectors**

The first of these advanced selectors is the child selector. Whereas a descendant selector will select all the descendants of an element, a child selector only targets the element’s immediate descendants, or “children.” In the following example, the list items in the outer list will be bold while list items in the nested list will remain unaffected:

#nav > li {font-weight: bold;}

<ul id="nav">

<li>Home</li>

<li>Services

<ul>

<li>Design</li>

<li>Development</li>

<li>Consultancy</li>

</ul>

</li>

<li>Contact Us </li>

</ul>

It is possible to “fake” a child selector that works in IE 6 and below by using the universal selector. To do this you first apply to all of the descendants the style you want the children to have. You then use the universal selector to override these styles on the children’s descendants. So to fake the previous child selector example you would do this:

#nav li {font-weight: bold;}

#nav li \* {font-weight: normal;}

You may also want to style an element based on its proximity to another element. The adjacent sibling selector allows you to target an element that is preceded by another element that shares the same parent. Using the sibling selector, you could make the first paragraph following a top-level heading bold, while leaving other paragraphs unaffected:

h1 + p {font-weight: bold;}

<h1>Main Heading</h1>

<p>First Paragraph</p>

<p>Second Paragraph</p>

**Attribute selectors**

As the name suggests, the attribute selector allows you to target an element based on the existence of an attribute or the attribute’s value. This allows you to do some very interesting and powerful things. For example, when you hover over an element with a title attribute, most browsers will display a tooltip. You can use this behavior to expand the meaning of things such as abbreviations:

<abbr title="Cascading Style Sheets">CSS</abbr>

However, there is no way to tell that this extra information exists without hovering over the element. To get around this problem, you can use the attribute selector to style abbr elements with titles differently from other elements—in this case, by giving them a dotted bottom border. You can provide more contextual information by changing the cursor from a pointer to a question mark when the cursor hovers over the element, indicating that this element is different from most.

abbr[title] {border-bottom: 1px dotted #999;}

abbr[title]:hover {cursor: help;}

In addition to styling an element based on the existence of an attribute, you can apply styles based on a particular value. For instance, sites that are linked to using a rel attribute of nofollow gain no added ranking benefit from Google. The following rule displays an image next to such links, possibly as a way of showing disapproval of the target site:

a[rel="nofollow"] {

background-image: url(nofollow.gif);

padding-right: 20px;

}

**Pseudo-classes**

There are instances where you may want to style an element based on something other than the structure of the document—for instance, the state of a form element or link. This can be done using a pseudo-class selector.

/\* makes all unvisited links blue \*/

a:link {color:blue;}

/\* makes all visited links green \*/

a:visited {color:green;}

/\* makes links red when hovered or activated \*/

a:hover, a:active {color:red;}

/\* makes table rows red when hovered over \*/

tr:hover {background-color: red;}

/\* makes input elements yellow when focus is applied \*/

input:focus {background-color:yellow;}

:link and :visited are known as link pseudo-classes and can only be applied to anchor elements. :hover, :active, and :focus are known as dynamic pseudo-classes and can theoretically be applied to any element.

* 1. The Cascade and Specificity

With even a moderately complicated stylesheet, it is likely that two or more rules will target the same element. CSS handles such conflicts through a process known as the cascade. The cascade works by assigning an importance to each rule. Author stylesheets are considered the most important, followed by user stylesheets, and finally the default

stylesheets used by your browser or user agent. To give users more control, they can override any rule by specifying it as !important—even a rule flagged as !important by the author.

So the cascade works in the following order of importance:

* User styles flagged as !important
* Author styles flagged as !important
* Author styles
* User styles
* Styles applied by the browser/user agent

Rules are then ordered by how specific the selector is. **Rules with more specific selectors override those with less specific ones.** If two rules are equally specific, the last one defined takes precedence.

To calculate how specific a rule is, each type of selector is assigned a numeric value. The

specificity of a rule is then calculated by adding up the value of each of its selectors. Unfortunately, specificity is not calculated in base 10 but a high, unspecified, base number. This is to ensure that a highly specific selector, such as an ID selector, is never overridden by lots of less specific selectors, such as type selectors. However, if you have fewer than 10 selectors in a specific selector, you can calculate specificity in base 10 for simplicity’s sake.

1. 1. 1. Specificity Hierarchy

Every selector has its place in the specificity hierarchy. There are four distinct categories which define the specificity level of a given selector:

**Inline styles (Presence of style in document).**

An inline style lives within your XHTML document. It is attached directly to the element to be styled. E.g. <h1 style="color: #fff;">

**IDs (# of ID selectors)**

ID is an identifier for your page elements, such as #div.

**Classes, attributes and pseudo-classes (# of class selectors).**

This group includes .classes, [attributes] and pseudo-classes such as :hover, :focus etc.

**Elements and pseudo-elements (# of Element (type) selectors).**

Including for instance :before and :after.

* + 1. How to measure specificity?
* count 1 if the declaration is **from a 'style‘** (= a)
* count the number of **ID attributes** in the selector (= b)
* count the number of other **attributes** and **pseudo-classes** in the selector (= c)
* count the **number of element names** and **pseudo-elements** in the selector (= d)

**Examples:**

\* {} /\* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0 \*/

li {} /\* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 \*/

li.red.level {} /\* a=0 b=0 c=2 d=1 -> specificity = 0,0,2,1 \*/

#x34y {} /\* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 \*/

* + 1. Using specificity in your stylesheets

Specificity is very useful when writing CSS as it allows you to set general styles for common elements and then override them for more specific elements. For instance, say you want most of the forms on your site to be 30em wide but your search form needs to be only 15em wide:

form {width: 30em;}

form#search {width: 15em;}

Whenever you want to create a new form you do not have to worry about changing anything in the CSS, as you know it will be styled correctly. However, on larger sites you will find more and more exceptions will start to creep in. Maybe you will have a login form that you want to be 20em wide or a larger application form that needs to be 40em wide. Each time you create a more specific style, you will probably need to override some of the general rules. This can lead to quite a bit of extra code. It can also start to get very complicated as one element may be picking up styles from a variety of places.

* 1. Inheritance

People often confuse inheritance with the cascade. Although they seem related at first glance, the two concepts are actually quite different. Luckily, inheritance is a much easier concept to grasp. Certain properties, such as color or font size, are inherited by the descendants of the elements those styles are applied to. For instance, if you were to give the body element a text color of black, all the descendants of the body element would also have black text. The same would be true of font sizes. If you gave the body a font size of 14 pixels, everything on the page should inherit that font size. I say should because IE for Windows and Netscape have problems inheriting font sizes in tables. To get around this, you will either have to specify that tables should inherit font sizes or set the font size on tables separately. If you set the font size on the body, you will notice that this style is not picked up by any headings on the page. You may assume that headings do not inherit text size. But it is actually the browser default stylesheet setting the heading size. Any style applied directly to an element will always override an inherited style. This is because inherited styles have a null specificity. Inheritance is very useful as it lets you avoid having to add the same style to every descendant of an element. If the property you are trying to set is an inherited property, you may as well apply it to the parent element. After all, what is the point of writing this:

p, div, h1, h2, h3, ul, ol, dl, li {color: black;}

when you can just write this:

body {color: black;}

Just as sensible use of the cascade can help simplify your CSS, good use of inheritance can help to reduce the number and complexity of the selectors in your code. It you have lots of elements inheriting various styles, though, determining where the styles originate can become confusing.

* 1. Applying the style

You can add styles directly to the head of a document by placing them between style tags. However, this is not a very sensible way to apply styles to a document. If you want to create another page using the same styles, you would have to duplicate the CSS on the new page. If you then wanted to change a style, you would have to do it in two places rather than one. Luckily, CSS allows us to keep all our styles in one or more external stylesheets. There are two ways to attach external stylesheets to a web page. You can link to them or you can import them:

<link href="/css/basic.css" rel="stylesheet" type="text/css" />

<style type="text/css">

<!--

@import url("/css/advanced.css");

-->

</style>

* 1. Commenting your Code

Adding comments in CSS is very simple. A CSS comment starts with /\* and ends with \*/. This type of commenting is known as C style commenting as it is the type of comment used in the C programming language. Comments can be single or multiline and can appear anywhere within the code.

/\* Body Styles \*/

body {

font-size: 67.5%; /\* Set the font size \*/

}

**Adding structural comments**

/\*–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Basic Style Sheet (for version 4 browsers)

version: 1.1

author: andy budd

email: info@andybudd.com

website: http://www.andybudd.com/

–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––\*/

* 1. CSS 2.1 Properties
     1. Box Model

**Margin: 'margin-top', 'margin-right', 'margin-bottom', 'margin-left', and 'margin'**

Margin properties specify the width of the margin area of a box. The 'margin' shorthand property sets the margin for all four sides while the other margin properties only set their respective side.

**Padding properties: 'padding-top', 'padding-right', 'padding-bottom', 'padding-left', and 'padding'**

The padding properties specify the width of the padding area of a box. The 'padding' shorthand property sets the padding for all four sides while the other padding properties only set their respective side.

**Border properties**

The border properties specify the width, color, and style of the border area of a box. These properties apply to all elements.

Border width: 'border-top-width', 'border-right-width', 'border-bottom-width', 'border-left-width', and 'border-width'

Border color: 'border-top-color', 'border-right-color', 'border-bottom-color', 'border-left-color', and 'border-color'

Border style: 'border-top-style', 'border-right-style', 'border-bottom-style', 'border-left-style', and 'border-style'

* + 1. Visual Effects

**Overflow Properties**

Generally, the content of a block box is confined to the content edges of the box. In certain cases, a box may overflow, meaning its content lies partly or entirely outside of the box, e.g.:

* A line cannot be broken, causing the line box to be wider than the block box.
* A block-level box is too wide for the containing block. This may happen when an element's 'width' property has a value that causes the generated block box to spill over sides of the containing block.
* An element's height exceeds an explicit height assigned to the containing block (i.e., the containing block's height is determined by the 'height' property, not by content height).
* A descendant box is positioned absolutely, partly outside the box. Such boxes are not always clipped by the overflow property on their ancestors.

**Visibility Properties**

The 'visibility' property specifies whether the boxes generated by an element are rendered. Invisible boxes still affect layout (set the 'display' property to 'none' to suppress box generation altogether).

* + 1. Colors and Backgrounds

**Color Properties**

This property describes the foreground color of an element's text content.

**Background Properties**

"background" refers to the background of the content, padding and border areas. Border colors and styles are set with the border properties. Margins are always transparent.

* + 1. Fonts

**Font Family Properties**

The value is a prioritized list of font family names and/or [generic family names.](http://www.w3.org/TR/CSS2/fonts.html#generic-font-families) Unlike most other CSS properties, values are separated by a comma to indicate that they are alternatives:

body { font-family: Gill, Helvetica, sans-serif }

**Font Style Properties**

The 'font-style' property selects between normal (sometimes referred to as "roman" or "upright"), italic and oblique faces within a font family.

**Font Weight Properties**

The 'font-weight' property selects the weight of the font. The values '100' to '900' form an ordered sequence, where each number indicates a weight that is at least as dark as its predecessor. The keyword 'normal' is synonymous with '400', and 'bold' is synonymous with '700'.

**Font Size Properties**

The font size corresponds to the em square, a concept used in typography. Note that certain glyphs may bleed outside their em squares.

**Font Properties**

The 'font' property is a shorthand property for setting 'font-style', 'font-variant', 'font-weight', 'font-size', 'line-height' and 'font-family' at the same place in the style sheet. The syntax of this property is based on a traditional typographical shorthand notation to set multiple properties related to fonts.

* + 1. Text

**Text Indent Properties**

This property specifies the indentation of the first line of text in a block. More precisely, it specifies the indentation of the first box that flows into the block's first [line box](http://www.w3.org/TR/CSS2/visuren.html#line-box). The box is indented with respect to the left (or right, for right-to-left layout) edge of the line box.

**Text Align Properties**

This property describes how inline content of a block is aligned.

**Text Decoration Properties**

This property describes decorations that are added to the text of an element using the element's color. When specified on an inline element, it affects all the boxes generated by that element;

* + 1. User Interface

**Cursors: the 'cursor' property**

This property specifies the type of cursor to be displayed for the pointing device.

For complete reference to CSS 2.1 specification, please refer to the W3C. [Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification](http://www.w3.org/TR/CSS2/)

* 1. CSS 3.0 Properties
     1. Borders

Border Color. Next to rounded borders, border-color is also very interesting. Mozila/Firefox has implemented this function, which allows you to create cool colored borders

Border Image. With this feature you can define an image to be used instead of the normal border of an element.

* + **border-image**: border-top-image border-right-image border-bottom- image border-left-image
  + **border-corner-image**: border-top-left-image border-top-right-image border-bottom-left-image border-bottom-right-image

Border Radius. It allows web developers to easily utilise rounder corners in their design elements, without the need for corner images or the use of multiple div tags, and is perhaps one of the most talked about aspects of CSS3.

* **border-radius:** border-bottom-left-radius, border-bottom-right-radius, border-top-left-radius, border-top-right-radius

Box Shadow. **box-shadow** allows designers to easily implement multiple drop shadows (outer or inner) on box elements, specifying values for color, size, blur and offset.

* + 1. Background

Background origin. The background-origin property is used to determine how the background-position of a background in a certain box is calculated.

Background size. This property adds new functionality to CSS allowing designers to specify the size of background images using either lengths, percentages, or by using one of two keywords; contain or cover.

Multiple Backgrounds. It allows web designers to specify multiple background images for box elements, using nothing more than a simple comma-separated list.

* + 1. Color

There is a huge variety of colors available with the new CSS edition. HSL, HSLA and RGBA are the new color schemes. The new ‘A’ value stands for the opacity. With this value, the divs can make transparent and give a much attractive look to the appearance. In early editions, the transparency was obtained by adding a .png or .gif image as the background.

* + 1. Text Effects

Text Shadow. CSS3 finally eliminates the need for Photoshop when all you want to do is a simple shadow with text-shadow.

Text Overflow. text-overflow helps you when you need text to overflow an element’s box and you want a visual “hint” to the user that text has been clipped.

Word Wrap. The word-wrap property was invented by Microsoft and added to CSS3. It allows long words to be able to be broken and wrap onto the next line. It takes in two values; normal or break-word.

* + 1. User Interface

Box model is a new property which was introduced with the CSS3 edition. With this, it can be easily assigned with customized div appearance. There should be two inner divs and a single outer div with the width of each inner div being 50%, there is also overflow-x, overflow-y as additional for basic box model.

Box Sizing. It let’s you change the behavior of the browser in calculating the width of an element. By default, box-sizing is set to content-box. With that set, it calculates width and height as specified by CSS 2.1, adding the border-width and border-height and the padding to the size of the box. By setting it to border-box, you can force the browser to instead render the box with the specified width and height, and add the border and padding *inside* the box.

Resize. It allows you to specify if a box is resizable.

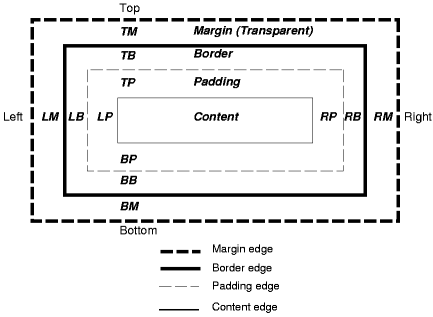
Outline. This allows the offset to be rendered away from the edge of the specified element.

* + 1. Web Fonts

A new collection of web fonts was introduced with the CSS3 edition. The multi column layout view also can be applied to the content as it can be used to provide multi column paragraphs. Media boxes are also a new implementation that could help to give a good view to the end user.

* 1. Visual Formatting
     1. Box Model Recap

The box model is one of the cornerstones of CSS and dictates how elements are displayed and, to a certain extent, how they interact with each other. Every element on the page is considered to be a rectangular box made up of the element’s content, padding, border, and margin.



The margin, border, and padding can be broken down into top, right, bottom, and left segments (e.g., in the diagram, "LM" for left margin, "RP" for right padding, "TB" for top border, etc.).

The perimeter of each of the four areas (content, padding, border, and margin) is called an "edge", so each box has four edges:

**content edge or** **inner edge**

The content edge surrounds the rectangle given by the width and height of the box, which often depend on the element's rendered content. The four content edges define the box's content box.

**padding edge**

The padding edge surrounds the box padding. If the padding has 0 width, the padding edge is the same as the content edge. The four padding edges define the box's padding box.

**border edge**

The border edge surrounds the box's border. If the border has 0 width, the border edge is the same as the padding edge. The four border edges define the box's border box.

**margin edge or** **outer edge**

The margin edge surrounds the box margin. If the margin has 0 width, the margin edge is the same as the border edge. The four margin edges define the box's margin box.

In CSS, width and height refer to the width and height of the content area. Adding padding, borders, and margins will not affect the size of the content area but will increase the overall size of an element’s box. If you wanted a box with a 10-pixel margin and a 5-pixel padding on each side to be 100 pixels wide, you would need to set the width of the content to be 70 pixels

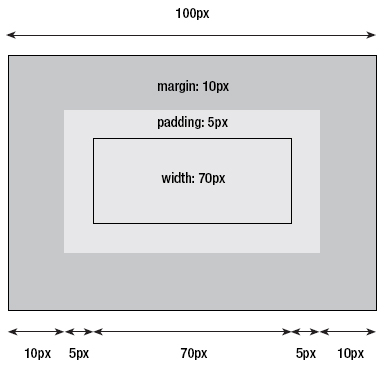
#myBox {

margin: 10px;

padding: 5px;

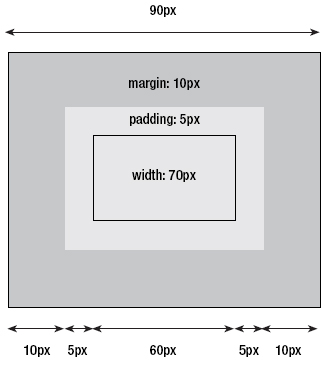
width: 70px;

}



**IE/Win and the box model**

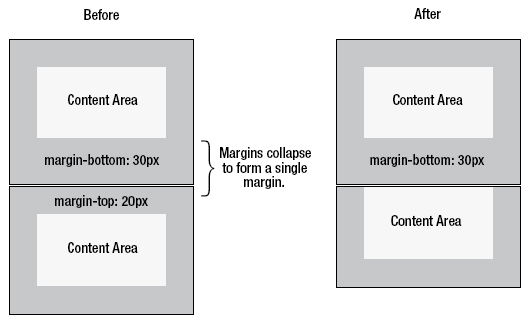
Unfortunately, IE 5.x and IE 6 in quirks mode use their own, nonstandard box model. Instead of measuring just the width of the content, these browsers take the width property as the sum of the width of the content, padding, and borders. This actually makes a lot of sense because in the real world boxes have a fixed size and the padding goes on the inside. The more padding you add, the less room there would be for the content. However, despite the logic, the fact that these versions of IE disregard the specification can cause significant problems. For instance, in the previous example the total width of the box would only be 90 pixels in IE 5.x. This is because IE 5.x will consider the 5 pixels of padding on each side as part of the 70-pixel width, rather than in addition to it.



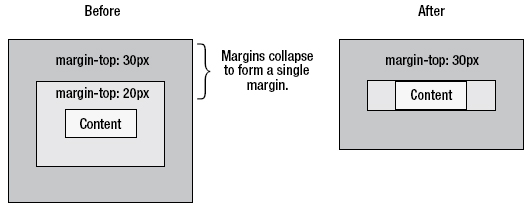
However, by far the best solution is to avoid the problem altogether. You can do this by never adding padding to an element with a defined width. Instead, try adding padding or margins to the element’s parent or children.

**Margin Collapsing**

Margin collapsing is a relatively simple concept. In practice, however, it can cause a lot of confusion when you’re laying out a web page. Put simply, when two or more vertical margins meet, they will collapse to form a single margin. The height of this margin will equal the height of the larger of the two collapsed margins. When two elements are above one another, the bottom margin of the first element will collapse with the top margin of the second element.



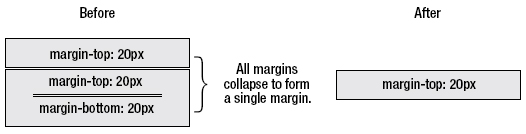
When one element is contained within another element, assuming there is no padding or border separating margins, their top and/or bottom margins will also collapse together



It may seem strange at first, but margins can even collapse on themselves. Say you have an empty element with a margin, but no border or padding. In this situation the top margin is touching the bottom margin and they collapse together

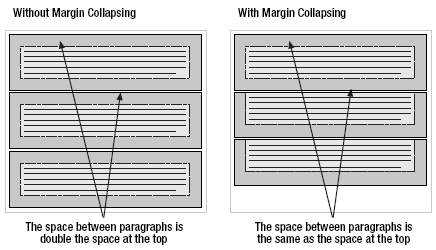


If this margin is touching the margin of another element, it will itself collapse



This is why a series of empty paragraph elements take up very little space, as all their margins collapse together to form a single small margin.

Margin collapsing may seem strange at first, but it actually makes a lot of sense. Take a typical page of text made up of several paragraphs. The space above the first paragraph will equal the paragraph’s top margin. Without margin collapsing, the space between all subsequent paragraphs will be the sum of their two adjoining top and bottom margins. This means that the space between paragraphs will be double the space at the top of the page. With margin collapsing, the top and bottom margins between each paragraph collapse, leaving the spacing the same everywhere.



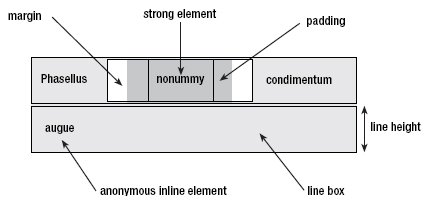
* + 1. Positioning

People often refer to elements such as p, h1, or div as block-level elements. This means they are elements that are visually displayed as blocks of content, or “block boxes.” Conversely, elements such as strong and span are described as inline elements because their content is displayed within lines as “inline boxes.”

It is possible to change the type of box generated by using the display property. This means you can make an inline element such as an anchor behave like a block-level element by setting its display property to block. It is also possible to cause an element to generate no box at all by setting its display property to none. The box, and thus all of its content, is no longer displayed and takes up no space in the document.

There are three basic positioning schemes in CSS: normal flow, floats, and absolute positioning. Unless specified, all boxes start life being positioned in the normal flow. As the name suggests, the position of an element’s box in the normal flow will be dictated by that element’s position in the (X)HTML.

Block-level boxes will appear vertically one after the other; the vertical distance between boxes is calculated by the boxes’ vertical margins. Inline boxes are laid out in a line horizontally. Their horizontal spacing can be adjusted using horizontal padding, borders, and margins. However, vertical padding, borders, and margins will have no effect on the height of an inline box. The horizontal box formed by a line is called a line box, and a line box will always be tall enough for all the line boxes it contains. There is another caveat, though—setting the line height can increase the height of this box.



**Relative Positioning**

Relative positioning is a fairly easy concept to grasp. If you relatively position an element, it will stay exactly where it is. You can then shift the element “relative” to its starting point by setting a vertical or horizontal position. If you set the top position to be 20 pixels, the box will appear 20 pixels below the top of its original position. Setting the left position to 20 pixels will create a 20-pixel space on the left of the element, moving the element to the right.

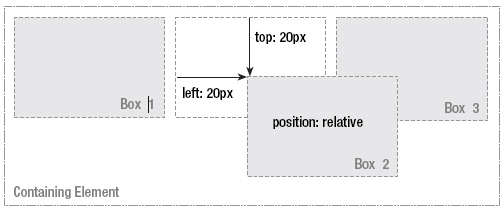
#myBox {

position: relative;

left: 20px;

top: 20px;

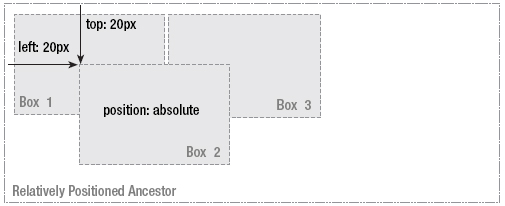
}



**Absolute Positioning**

Relative positioning is actually considered part of the normal flow positioning model, as the position of the element is relative to its position in the normal flow. By contrast, absolute positioning takes the element out of the flow of the document, thus taking up no space. Other elements in the normal flow of the document will act as though the

absolutely positioned element was never there.



An absolutely positioned element is positioned in relation to its nearest positioned ancestor. If the element has no positioned ancestors, it will be positioned in relation to the initial containing block. Depending on the user agent, this will either be the canvas or the HTML element.

As with relatively positioned boxes, an absolutely positioned box can be offset from the top, bottom, left, or right of its containing block. This gives you a great deal of flexibility. You can literally position an element anywhere on the page.

Because absolutely positioned boxes are taken out of the flow of the document, they can overlap other elements on the page. You can control the stacking order of these boxes by setting a property called the z-index. The higher the z-index, the higher up the box appears in the stack.

Absolutely positioning a box in relation to a relatively positioned ancestor works well in most modern browsers. However, there is a bug in IE 5.5 and IE 6 on Windows. If you try to set the position of the absolutely positioned box relative to the right or bottom of the relatively positioned box, you need to make sure the relatively positioned box has some dimensions set. If not, IE will incorrectly position the box in relation to the canvas instead.

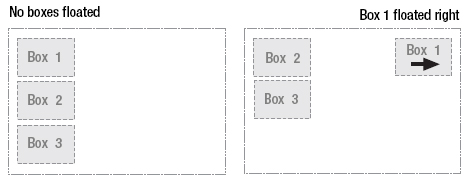
**Fixed Positioning**

Fixed positioning is a subcategory of absolute positioning. The difference is that a fixed element’s containing block is the viewport. This allows you to create floating elements that always stay at the same position in the window.

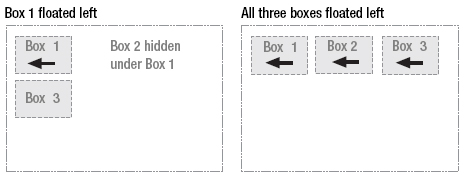
**Floating**

The last positioning model is the float model. A floated box can either be shifted to the left or the right until its outer edge touches the edge of its containing box, or another floated box. Because floated boxes aren’t in the normal flow of the document, block boxes in the regular flow of the document behave as if the floated box wasn’t there.

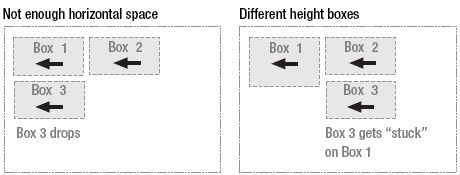
when you float Box 1 to the right, it’s taken out of the flow of the document and moved to the right until its right edge touches the right edge of the containing block.



when you float Box 1 to the left, it is taken out of the flow of the document and moved left until its left edge touches the left edge of the containing block. Because it is no longer in the flow, it takes up no space and actually sits on top of Box 2, obscuring it from view. If you float all three boxes to the left, Box 1 is shifted left until it touches its containing box, and the other two boxes are shifted left until they touch the preceding floated box.

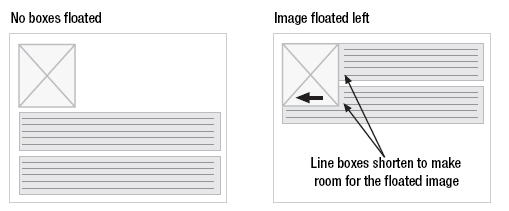


If the containing block is too narrow for all of the floated elements to fit horizontally, the remaining floats will drop down until there is sufficient. If the floated elements have different heights, it is possible for floats to get “stuck” on other floats when they drop down.

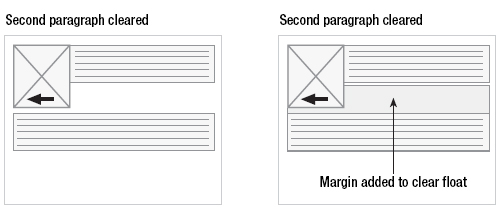


**Line Box and Clearing**

Line boxes next to a floated box are shortened to make room for the floated box, and flow around the float. In fact, floats were created to allow text to flow around images.



To stop line boxes flowing around the outside of a floated box, you need to apply a clear to that box. The clear property can be left, right, both, or none, and indicates which side of the box should not be next to a floated box. To accomplish this, enough space is added above the cleared element’s top margin to push the element’s top border edge vertically down, past the float.



As you’ve seen, floated elements are taken out of the flow of the document and have no effect on surrounding elements. However, clearing an element essentially clears a vertical space for all the preceding floated elements. This can be a useful layout tool as it allows surrounding elements to make space for floated elements. This solves the problem we saw earlier with absolute positioning where changes in vertical height do not affect surrounding elements and can break your design. Let’s have a look at floating and clearing in a little more detail. Say you have a picture that you want to float to the left of a block of text. You want this picture and text to be contained in another element with a background color and border. You would probably try something like this:

.news {

background-color: gray;

border: solid 1px black;

}

.news img {

float: left;

}

.news p {

float: right;

}

<div class="news">

<img src="news-pic.jpg" />

<p>Some text</p>

</div>

However, because the floated elements are taken out of the flow of the document, the wrapper div takes up no space. How do you visually get the wrapper to enclose the floated element? You need to apply a clear somewhere inside that element. Unfortunately, no existing element is available that we can clear so you need to add an empty element and clear that.

.news {

background-color: gray;

border: solid 1px black;

}

.news img {

float: left;

}

.news p {

float: right;

}

.clear {

clear: both;

}

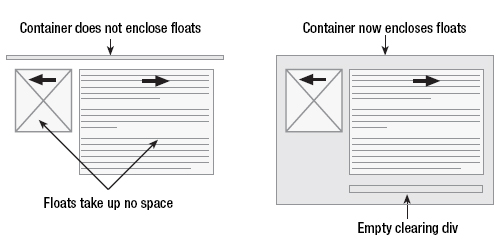
<div class="news">

<img src="news-pic.jpg" />

<p>Some text</p>

<div class="clear"></div>

</div>



This gets the result we want, but at the expense of adding extraneous code to our markup. Often there will be an existing element you can apply the clear to, but sometimes you may have to bite the bullet and add meaningless markup for the purpose of layout.

Instead of clearing the floated text and image, you could choose to float the container div as well:

.news {

background-color: gray;

border: solid 1px black;

float: left;

}

.news img {

float: left;

}

.news p {

float: right;

}

<div class="news">

<img src="news-pic.jpg" />

<p>Some text</p>

</div>

This creates the desired result. Unfortunately, the next element is now going to be affected by the float. To solve this problem, some people choose to float nearly everything in a layout and then clear those floats using an appropriate meaningful element, often the site footer. This helps reduce or eliminate the need for extraneous markup. However, floating can be complicated and some older browsers may choke on heavily floated layouts. As such, many people prefer to add that extra bit of markup. Applying an overflow property of hidden or auto will automatically clear any contained floats without the addition of extra markup. This method is not appropriate in all situations, since setting the box’s overflow property will affect how it behaves.

Lastly, some people have taken to clearing floats using CSS-generated content or JavaScript. The basic concept for both methods is the same. Rather than add a clearing element directly to the markup, you add it to the page dynamically. For both methods you need to indicate where the clearing element goes, and this is usually done with the addition of a class name:

<div class="news clear">

<img src="news-pic.jpg" />

<p Some text</p>

</div>

Using the CSS method, you use the :after pseudo-class in combination with the content declaration to add new content at the end of the specified existing content. In this case I’m adding a full stop as it is a fairly small and unobtrusive character. You don’t want the new content to take up any vertical space or be dzisplayed on the page, so you need to set height to 0 and visibility to hidden. Because cleared elements have space added to their top margin, the generated content needs to have its display property set to block. Once this is done, you can then clear your generated content:

.clear:after {

content: ".";

height: 0;

visibility: hidden;

display: block;

clear: both;

}

* + 1. Layout

One of the major benefits of CSS is the ability to control page layout without needing to use presentational markup. However, CSS layout has gained a rather undeserved reputation of being difficult, particularly among those new to the language. This is partly due to browser inconsistencies, but mostly due to a proliferation of different layout techniques available on the Web. It seems that every CSS author has their own technique for creating multicolumn layouts, and new CSS developers will often use a technique without really understanding how it works. This “black box” approach to CSS layout may get quick results, but ultimately stunts the developer’s understanding of the language.

All these CSS layout techniques rely on three basic concepts: positioning, floating, and margin manipulation. The different techniques really aren’t that different, and if you understand the core concepts, it is relatively easy to create your own layouts with little or no hassle.

**Centering a design**

There are two basic methods for centering a design: one uses auto margins and the other uses positioning and negative margins.

**Centering a design using auto margins**

Say you have a typical layout where you wish to center a wrapper div horizontally on the screen:

<body>

<div id="wrapper">

</div>

</body>

To do this you simply define the width of your wrapper div and then set the horizontal margins to auto:

#wrapper {

width: 720px;

margin: 0 auto;

}

In this example I have decided to fix the width of my wrapper div in pixels, so that it fits nicely on an 800✕600 resolution screen. However, you could just as easily set the width as a percentage of the body or relative to the size of the text using ems.

This works on all modern browsers. However, IE 5.x and IE 6 in quirks mode doesn’t honor auto margins. Luckily, IE misunderstands text-align: center, centering everything instead of just the text. You can use this to your advantage by centering everything in the body tag, including the wrapper div, and then realigning the contents of the wrapper back to the left:

body {

text-align: center;

}

#wrapper {

width: 720px;

margin: 0 auto;

text-align: left;

}

Using the text-align property in this way is a hack—but a fairly innocuous hack that has no adverse effect on your code. The wrapper now appears centered in IE as well as more standards-compliant browsers

There is one final thing that needs to be done in order for this method to work smoothly in all browsers. In Netscape 6, when the width of the browser window is reduced below the width of the wrapper, the left side of the wrapper spills off the side of the page and cannot be accessed. To keep this from happening, you need to give the body element a minimum width equal to or slightly wider than the width of the wrapper element:

body {

text-align: center;

min-width: 760px;

}

#wrapper {

width: 720px;

margin: 0 auto;

text-align: left;

}

Now if you try to reduce the width of the window below the width of the wrapper div, scroll bars will appear, allowing you to access all of the content.

**Centering a design using positioning and negative margins**

The auto margin method of centering is by far the most common approach, but it does involve using a hack to satisfy IE 5.x. It also requires you to style two elements rather than just the one. Because of this, some people prefer to use positioning and negative margins instead.

You start as you did before, by defining the width of the wrapper. You then set the position property of the wrapper to relative and set the left property to 50%. This will position the left edge of the wrapper in the middle of the page.

#wrapper {

width: 720px;

position: relative;

left: 50%;

}

However, you don’t want the left edge of the wrapper centered—you want the middle of the wrapper centered. You can do this by applying a negative margin to the left side of the wrapper, equal to half the width of the wrapper. This will move the wrapper half its width to the left, centering it on screen:

#wrapper {

width: 720px;

position: relative;

left: 50%;

margin-left: -360px;

}

Your choice of centering technique comes down to personal taste. However, it is always useful to have several techniques up your sleeve, as you never know when one may come in handy.

**Float-based layouts**

There are a few different ways of doing CSS-based layout, including absolute positioning and using negative margins. I find float-based layouts the easiest method to use. As the name suggests, in a float-based layout you simply set the width of the elements you want to position, and then float them left or right.

Because floated elements no longer take up any space in the flow of the document, they no longer appear to exert any influence on the surrounding block boxes. To get around this, you will need to clear the floats at various points throughout the layout. Rather than continuously floating and clearing elements, it is quite common to float nearly everything, and then clear once or twice at strategic points throughout the document, such as the page footer.

**Two-column floated layout**

To create a simple two-column layout using floats, you need to start off with a basic (X)HTML framework. In this example the (X)HTML consists of a branding area, a content area, an area for the main navigation, and finally a page footer. The whole design is enclosed in a wrapper div, which will be horizontally centered using one of the preceding methods:

<div id="wrapper">

<div id="branding">

...

</div>

<div id="content">

...

</div>

<div id="mainNav">

...

</div>

<div id="footer">

...

</div>

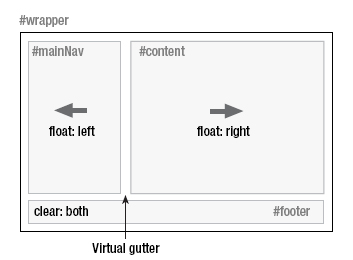
</div>

The main navigation for this design will be on the left side of the page and the content will be on the right. However, I have chosen to put the content area above the navigation in the source order for usability and accessibility reasons. First, the main content is the most important thing on the page and so should come first in the document. Second, there is no point forcing screenreader users to trawl through a potentially long list of links before they get to the content if they don’t have to.

Normally when people create float-based layouts, they float both columns left, and then create a gutter between the columns using margin or padding. When using this approach, the columns are packed tightly into the available space with no room to breathe. Although this wouldn’t be a problem if browsers behaved themselves, buggy browsers can cause tightly packed layouts to break, forcing columns to drop below each other.

This can happen on IE because IE/Win honors the size of an element’s content, rather than the size of the element itself. In standards-compliant browsers, if the content of an element gets too large, it will simply flow out of the box. However, on IE/Win, if the content of an element becomes too big, the whole element expands. If this happens in very tightly packed layouts, there is no longer enough room for the elements to sit next to each other, and one of the floats will drop.

To prevent this from happening, you need to avoid cramming floated layouts into their containing elements. Rather than using horizontal margin or padding to create gutters, you can create a virtual gutter by floating one element left and one element right. If one element inadvertently increases in size by a few pixels, rather than immediately running out of horizontal space and dropping down, it will simply grow into the virtual gutter.



The CSS for achieving this layout is very straightforward. You simply set the desired width of each column, then float the navigation left and the content right:

#content {

width: 520px;

float: right;

}

#mainNav {

width: 180px;

float: left;

}

Then, to ensure that the footer is positioned correctly below the two floats, the footer needs to be cleared:

#footer {

clear: both;

}

The basic layout is now complete. Just a few small tweaks are required to tidy things up. First, the content in the navigation area is flush to the edges of the container and needs some breathing room. You could add horizontal padding directly to the navigation element, but this will invoke IE 5.x’s proprietary box model. To avoid this, add the horizontal padding to the navigation area’s content instead:

#mainNav {

padding-top: 20px;

padding-bottom: 20px;

}

#mainNav li {

padding-left: 20px;

padding-right: 20px;

}

The right side of the content area is also flush to the right edge of its container and needs some breathing room. Again, rather than apply padding directly to the element, you can apply padding to the content and avoid having to deal with IE’s box model problems:

#content h1, #content h2, #content p {

padding-right: 20px;

}

**Three-column floated layout**

The HTML needed to create a three-column layout is very similar to that used by the twocolumn layout, the only difference being the addition of two new divs inside the content div: one for the main content and one for the secondary content.

<div id="content">

<div id="mainContent">

…

</div>

<div id="secondaryContent">

…

</div>

</div>

Using the same CSS as the two-column technique, you can float the main content left and the secondary content right, inside the already floated content div. This essentially divides the second content column in two, creating your three-column effect.

As before, the CSS for this is very simple. You just set your desired widths and then float the main content left and the secondary content right:

#mainContent {

width: 320px;

float: left;

}

#secondaryContent {

width: 180px;

float: right;

}

You can tidy up the layout slightly by removing the padding from the content element and applying it to the content of the secondary content instead:

#secondaryContent h1, #secondaryContent h2, #secondaryContent p {

padding-left: 20px;

padding-right: 20px;

}

**Fixed-width, liquid, and elastic layout**

So far, all the examples have used widths defined in pixels. This type of layout is known as fixed-width layout, or sometimes “ice layout” due to its rigid nature. Fixed-width layouts are very common as they give the developer more control over layout and positioning. If you set the width of your design to be 720 pixels wide, it will always be 720 pixels. If you then want a branding image spanning the top of your design, you know it needs to be 720 pixels wide to fit. Knowing the exact width of each element allows you to lay them out precisely and know where everything will be. This predictability makes fixed-width layout by far the most common layout method around.

However, fixed-width designs have their downsides. First, because they are fixed, they are always the same size no matter what your window size. As such, they don’t make good use of the available space. On large screen resolutions, designs created for 800✕600 can appear tiny and lost in the middle of the screen. Conversely, a design created for a 1024✕760 screen will cause horizontal scrolling on smaller screen resolutions. With an increasingly diverse range of screen sizes to contend with, fixed-width design is starting to feel like a poor compromise.

To work around these issues, you could choose to use liquid or elastic layout instead of fixed-width layout.

**Liquid layouts**

With liquid layouts, dimensions are set using percentages instead of pixels. This allows liquid layouts to scale in relation to the browser window. As the browser window gets bigger, the columns get wider. Conversely, as the window gets smaller, the columns will reduce in width. Liquid layouts make for very efficient use of space, and the best liquid layouts aren’t even noticeable.

However, liquid layouts are not without their own problems. At small window widths, line lengths can get incredibly narrow and difficult to read. This is especially true in multicolumn layouts. As such, it may be worth adding a min-width in pixels or ems to prevent the layout from becoming too narrow.

Conversely, if the design spans the entire width of the browser window, line lengths can become long and difficult to read. There are a couple of things you can do to help avoid this problem. First, rather than spanning the whole width, you could make the wrapper span just a percentage—say, 85 percent. You could also consider setting the padding and margin as percentages as well. That way, the padding and margin will increase in width in relation to the window size, stopping the columns from getting too wide, too quickly. Lastly, for very severe cases, you could also choose to set the maximum width of the wrapper in pixels to prevent the content from getting ridiculously wide on oversized monitors.

You can use these techniques to turn the previous fixed-width, three-column layout into a fluid, three-column layout. Start by setting the width of the wrapper as a percentage of the overall width of the window. In this example I have chosen 85 percent as it produces good results on a range of screen sizes. Next, set the width of the navigation and content areas as a percentage of the wrapper width. After a bit of trial and error, setting the navigation area to be 23 percent and the content area to 75 percent produced nice results. This leaves a 2-percent virtual gutter between the navigation and the wrapper to deal with any rounding errors and width irregularities that may occur:

#wrapper {

width: 85%;

}

#mainNav {

width: 23%;

float: left;

}

#content {

width: 75%;

float: right;

}

You then need to set the widths of the columns in the content area. This gets a bit trickier as the widths of the content divs are based on the width of the content element and not the overall wrapper. If you want the secondaryContent to be the same width as the main navigation, you need to work out what 23 percent of the wrapper is in terms of the width of the content area. This is 23 (width of the nav) divided by 75 (width of the content area), multiplied by 100—which works out at around 31 percent. You will want the gutter between the content columns to be the same width as the gutter between the navigation and content areas. Using the same method, this works out to be around 3 percent, meaning that the width of the main content area should be 66 percent:

#mainContent {

width: 66%;

float: left;

}

#secondaryContent {

width: 31%;

float: right;

}

Because this layout scales so nicely, there isn’t any need to add a max-width property. However, the content does start to get squashed at smaller sizes, so you could set a minimum width of 720px on the wrapper element if you liked.

**Elastic layouts**

While liquid layouts are useful for making the most of the available space, line lengths can still get uncomfortably long on large resolution monitors. Conversely, lines can become very short and fragmented in narrow windows or when the text size is increased a couple of steps. If this is a concern, then elastic layouts may be a possible solution. Elastic layouts work by setting the width of elements relative to the font size instead of the browser width. By setting widths in ems, you ensure that when the font size is increased the whole layout scales. This allows you to keep line lengths to a readable size and is particularly useful for people with reduced vision or cognitive disorders.

Like other layout techniques, elastic layouts are not without their problems. Elastic layouts share some of their problems with fixed-width layouts, such as not making the most use of the available space. Also, because the whole layout increases when the text size is increased, elastic layouts can become much wider than the browser window, forcing the appearance of horizontal scroll bars. To combat this, it may be worth adding a max-width of 100% to the body tag. max-width isn’t currently supported by IE 6 and below, but it is supported by standards-compliant browsers such as Safari and Firefox.

Elastic layouts are much easier to create than liquid layouts as all of the HTML elements essentially stay in the same place relative to each other; they just all increase in size. Turning a fixed-width layout into an elastic layout is a relatively simple task. The trick is to set the base font size so that 1em roughly equals 10 pixels. The default font size on most browsers is 16 pixels. Ten pixels works out at 62.5 percent of 16 pixels, so setting the font size on the body to 62.5% does the trick:

body {

font-size: 62.5%;

}

Because 1em now equals 10 pixels at the default font size, we can convert our fixed-width layout into an elastic layout by converting all the pixel widths to em widths:

#wrapper {

width: 72em;

margin: 0 auto;

text-align: left;

}

#mainNav {

width: 18em;

float: left;

}

#content {

width: 52em;

float: right;

}

#mainContent {

width: 32em;

float: left;

}

#secondaryContent {

width: 18em;

float: right;

}

**Elastic-liquid hybrid**

Lastly, you could choose to create a hybrid layout that combines both elastic and liquid techniques. This hybrid approach works by setting the widths in ems, then setting the maximum widths as percentages:

#wrapper {

width: 72em;

max-width: 100%;

margin: 0 auto;

text-align: left;

}

#mainNav {

width: 18em;

max-width: 23%;

float: left;

}

#content {

width: 52em;

max-width: 75%;

float: right;

}

#mainContent {

width: 32em;

max-width: 66%;

float: left;

}

#secondaryContent {

width: 18em;

max-width: 31%;

float: right;

}

**Liquid and elastic images**

If you choose to use a liquid or an elastic layout, fixed-width images can have a drastic effect on your design. When the width of the layout is reduced, images will shift and may interact negatively with each other. Images will create natural minimum widths, preventing some elements from reducing in size. Other images will break out of their containing elements, wreaking havoc on finely tuned designs. Increasing the width of the layout can also have dramatic consequences, creating unwanted gaps and unbalancing designs. But never fear—there are a few ways to avoid such problems. For images that need to span a wide area, such as those found in the site header or branding areas, consider using a background image rather than an image element. As the branding element scales, more or less of the background image will be revealed:

#branding {

height: 171px;

background: url(images/branding.png) no-repeat left top;

}

<div id="branding"></div>

If the image needs to be on the page as an image element, try setting the width of the container element to 100% and the overflow property to hidden. The image will be truncated so that it fits inside the branding element but will scale as the layout scales:

#branding {

width: 100%;

overflow: hidden;

}

<div id="branding">

<img src="images/branding.png" width="1600" height="171" />

</div>

For regular content images, you will probably want them to scale vertically as well as horizontally to avoid clipping. You can do this by adding an image element to the page without any stated dimensions. You then set the percentage width of the image, and add a max-width the same size as the image to prevent pixelization.

For example, say you wanted to create a news story style with a narrow image column on the left and a larger text column on the right. The image needs to be roughly a quarter of the width of the containing box, with the text taking up the rest of the space. You can do this by simply setting the width of the image to 25% and then setting the max-width to be the size of the image—in this case 200 pixels wide:

.news img {

width: 25%;

max-width: 200px;

float: left;

padding: 2%;

}

.news p {

width: 68%;

float: right;

padding: 2% 2% 2% 0;

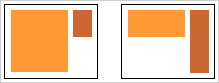
}

As the news element expands or contracts, the image and paragraphs will also expand or contract, maintaining their visual balance. However, on standards compliant browsers, the image will never get larger than its actual size.

**Faux columns**

One of the somewhat frustrating properties of CSS is the fact that elements only stretch vertically as far as they need to. Meaning, if a 200-pixel tall image is contained within a <div>, the <div> will only expand down the page 200 pixels.

This becomes an interesting dilemma when you use <div>s to section your markup, then apply CSS to create a columnar layout. One column may be longer than the other (see Figure 1). Depending on the amount of content contained, it becomes difficult to create a layout with two equally tall columns when a unique background color is desired for each column.



The embarassingly simple secret is to use a vertically tiled background image to create the illusion of colored columns. For SimpleBits, my background image looks something like:



with a decorative stripey thing on the left, a wide white section for the content column, a 1 pixel border, and a light brown section for the right column’s background followed by the reverse of the left side’s decorative border.

The whole image is no more than a few pixels tall, but when vertically tiled, it creates the colored columns that will flow all the way down to the bottom of the page — regardless of the length of content in the columns.

This elementary CSS rule is added to the body element:

background: #ccc url(../images/bg\_800.gif) repeat-y 50% 0;

Essentially, we’re making the entire page’s background color grey and tiling it vertically only (repeat-y). The 50% 0 bit refers to the positioning of the background image — in this case, 50% from the left side of the browser window (resulting in a centered image) and 0 pixels from the top.

With the background image in place, my positioned layout sits on top, with padding and margins set for the left and right columns, ensuring that they will line up in the right place — within the faux columns created by the background image.



* 1. Font Formatting

One of the most confusing aspects of CSS styling is the application of the **font-size** attribute for text scaling. In CSS, you’re given four different units by which you can measure the size of text as it’s displayed in the web browser.

**“Ems” (em)**

The “em” is a scalable unit that is used in web document media. An em is equal to the current font-size, for instance, if the font-size of the document is 12pt, 1em is equal to 12pt. Ems are scalable in nature, so 2em would equal 24pt, .5em would equal 6pt, etc. Ems are becoming increasingly popular in web documents due to scalability and their mobile-device-friendly nature.

**Pixels (px)**

Pixels are fixed-size units that are used in screen media (i.e. to be read on the computer screen). One pixel is equal to one dot on the computer screen (the smallest division of your screen’s resolution). Many web designers use pixel units in web documents in order to produce a pixel-perfect representation of their site as it is rendered in the browser. One problem with the pixel unit is that it does not scale upward for visually-impaired readers or downward to fit mobile devices.

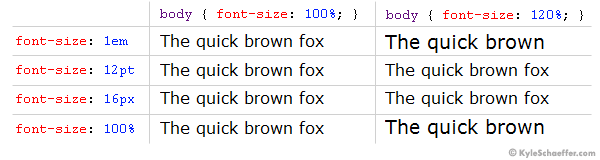
**Points (pt)**

Points are traditionally used in print media (anything that is to be printed on paper, etc.). One point is equal to 1/72 of an inch. Points are much like pixels, in that they are fixed-size units and cannot scale in size.

**Percent (%)**

The percent unit is much like the “em” unit, save for a few fundamental differences. First and foremost, the current font-size is equal to 100% (i.e. 12pt = 100%). While using the percent unit, your text remains fully scalable for mobile devices and for accessibility.

It’s easy to understand the difference between font-size units when you see them in action. Generally, **1em = 12pt = 16px = 100%**. When using these font-sizes, let’s see what happens when you increase the base font size (using the body CSS selector) from 100% to 120%.



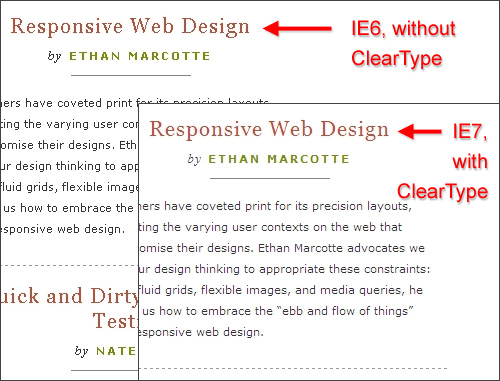
As you can see, both the em and percent units get larger as the base font-size increases, but pixels and points do not. It can be easy to set an absolute size for your text, but it’s much easier on your visitors to use scalable text that can display on any device or any machine.

**Typography Will Always Look Different**

Another area in which we can’t expect pixel-perfect designs is with regards to fonts, particularly fonts on body copy. Different methods have sprung up to help with custom fonts in headers, and the recently launched Google Font API will contribute to this. But body copy will probably always look different in different browsers.

With typography, we not only face the problem of font availability on different machines, but in some cases even when the font is available on two different machines, the type will look different. [Windows ClearType](http://www.microsoft.com/typography/cleartype/tuner/step1.aspx), for example, is available on IE7, but not on IE6, causing the same font to look different on two different versions of IE.

The graphic below displays screenshots from [A List Apart](http://www.alistapart.com/) on IE6 and IE7. The grainy text in IE6 is more evident on the heading than in the body copy, but all text displays a marked difference between the two browsers (unless of course the text is an image):



* 1. CSS Reset

The goal of a reset stylesheet is to reduce browser inconsistencies in things like default line heights, margins and font sizes of headings, and so on.

The basic reason is that all browsers have presentation defaults, but no browsers have the same defaults. (Okay, no two browser families—most Gecko-based browsers do have the same defaults.) For example, some browsers indent unordered and ordered lists with left margins, whereas others use left padding. In past years, we tackled these inconsistencies on a case-by-case basis; for example, making sure to always [set both left padding and left margin on lists](http://developer.mozilla.org/en/docs/Consistent_List_Indentation).

But there are all kinds of inconsistencies, some more subtle than others. Headings have slightly different top and bottom margins, indentation distances are different, and so on. Even something as basic as the default line height varies from one browser to another—which can have profound effects on element heights, vertical alignments, and overall feel.

This is not something we consider very often. We think of our CSS as modifying the default look of a document—but with a “reset” style sheet, we can make that default look more consistent across browsers, and thus spend less time fighting with browser defaults.

Some list of CSS reset:

* [Tripoli](http://devkick.com/lab/tripoli/)
* [Yahoo! UI Library: Reset CSS](http://developer.yahoo.com/yui/reset/)
* [CSS Reset Reloaded](http://meyerweb.com/eric/thoughts/2007/05/01/reset-reloaded/) by Eric Meyer
  1. Hacks and Filters

In an ideal world, properly coded CSS would work in every browser with CSS support. Unfortunately, we do not live in an ideal world, and browsers are littered with bugs and inconsistencies. To create pages that displayed the same across a variety of browsers, CSS developers had to get creative. By using bugs and unimplemented CSS, developers were able to selectively apply different rules to different browsers. Hacks and filters are a powerful weapon in a CSS developer’s arsenal. However, with great power comes great responsibility. It is important to know about the various common hacks and how they work, but it is equally important to know when and when not to use them.

* + 1. Conditional CSS

Conditional CSS allows you to write maintainable CSS with conditional logic to target specific CSS statements at both individual browsers and groups of browsers. Undoubtedly you will have found many a situation where different web-browsers require different style statements. Conditional CSS addresses this by letting you add Internet Explorer style conditions inline with your CSS statements. For example:

/\* Conditional-CSS example \*/

a.button\_active, a.button\_unactive {

display: inline-block;

   [if lte Gecko 1.8] display: -moz-inline-stack;

   [if lte Konq 3.1] float: left;

   height: 30px;

   [if IE 5.0] margin-top: -1px;

   text-decoration: none;

   outline: none;

   [if IE] text-decoration: expression(hideFocus='true');

}

**Conditional Statement**

* [if {!} browser]
* [if {!} browser version]
* [if {!} condition browser version]

Browser Names:

* IE : Internet Explorer
* Gecko : Gecko based browsers (Firefox, Camino etc)
* Webkit : Webkit based browsers (Safari, Shiira etc)
* 'SafMob' : Mobile Safari (iPhone / iPod Touch)
* Opera : Opera's browser
* IEMac : Internet Explorer for the Mac
* Konq : Konqueror
* IEmob : IE mobile
* PSP : Playstation Portable
* NetF : Net Front

**Conditional Operators**

* lt - Less than
* lte - Less than or equal to
* eq - Equal to
* gte - Greater than or equal to
* gt - Greater then
  + 1. Conditional Comment

Conditional comments only work in Explorer on Windows, and are thus excellently suited to give special instructions meant only for Explorer on Windows. They are supported from Explorer 5 onwards, and it is even possible to distinguish between 5.0, 5.5 and 6.0.

<!--[if IE]>

According to the conditional comment this is Internet Explorer

<![endif]-->

<!--[if IE 6]>

According to the conditional comment this is Internet Explorer 6

<![endif]-->

<!--[if IE 7]>

According to the conditional comment this is Internet Explorer 7

<![endif]-->

Conditional comments work as follows:

* Their basic structure is the same as an HTML comment (<!-- -->). Therefore all other browsers will see them as normal comments and will ignore them entirely.
* Explorer Windows, though, has been programmed to recognize the special <!--[if IE]> syntax, resolves the if and parses the content of the conditional comment as if it were normal page content.
* Since conditional comments use the **HTML** comment structure, they can only be included in HTML files, and not in CSS files. It is also possible to put an entire new <link> tag in the conditional comment referring to an extra style sheet.
  + 1. Child Selector Hack

The safest filters rely on unimplemented CSS rather than browser bugs. As these filters use valid CSS selectors to apply valid declarations, they are not, strictly speaking, filters at all. They are simply valid CSS rules that certain browsers fail to understand. The first of these filters is known as the child selector hack. IE 6 and below on Windows does not support the child selector, so you can use it to hide rules from those browsers. For this filter to work, you must make sure that there is no whitespace before or after the child selector.

In this example, the child selector hack is being used to hide a transparent background PNG image from IE 5-6/Win:

html>body {

background-image: url(bg.png);

}

IE 7 is expected to support the child selector. It is also expected to support native PNG transparency. By using the child selector filter in this way, you are building in forward compatibility by allowing newer versions of IE to view the transparent background without needing to revisit the code.

* + 1. Attribute Selector Hack

Another interesting way to filter rules is by using the attribute selector. Many modern browsers such as Safari and Firefox support the attribute selector, but it is not supported by IE 6 and below. As such, you can use the attribute selector as a way of styling classes and IDs for more advanced browsers. In this example, the attribute selector is being used to apply a background PNG to the content div on more compliant browsers:

div[id="content"] {

background-image: url(bg.png);

}

* + 1. Star HTML Hack

One of the best-known and possibly most useful CSS filters is known as the star HTML hack. This filter is incredibly easy to remember and targets IE 6 and below. As you are aware, the HTML element is supposed to be the first, or root, element on a web page.

However, all current versions of IE have an anonymous root element wrapping around the HTML element. By using the universal selector, you can target an HTML element enclosed inside another element. Because this only happens in IE 6 and below, you can apply specific rules to these browsers:

\* html {

font-size: small;

}

* + 1. !important and underscore Hack

There may be some instances where you wish to apply one declaration to IE 6 and below on Windows and another to all other browsers, within the same rule. To do this, you could use the commented property hack, or you could use the !important or the underscore hack.

The !important hack works because IE 6 and below on Windows has problems dealing with multiple properties in a single rule:

#nav {

position: fixed !important;

position: static;

}

IE 4-6/Win will ignore the first declaration and apply the second. All other browsers will apply the first declaration because it is using the !important keyword, which increases the rule’s priority within the cascade.

Similar to the !important hack is the underscore hack. By placing an underscore in front of a property, compliant browsers will no longer recognize that property and the declaration will be ignored. However, IE 6 and below on Windows ignores the underscore and thus applies the rule. So in this example, all modern browsers will apply a position of fixed, skipping the unknown second rule. IE 4-6/Win will ignore the underscore and will override the first rule, setting the position to static.

#nav {

position: fixed;

\_position: static;

}

* + 1. Owen Hack

All of the filters so far have been aimed at various versions of IE. This is partly because IE has more bugs than most current browsers. However, it is also because IE is by far the most prevalent browser, so more bugs get found and documented. But there are other buggy browsers out there, including Opera 6 and below.

The Owen hack allows authors to hide styles from Opera 6 and below, as well as from IE 6 and below on Windows. This filter works because these browsers do not implement the first-child selector. Because there can only ever be one head element, it is always a first child.

The body tag always comes after the head tag, and so can be targeted using an adjacent sibling selector. The resulting selector is understood by more compliant browsers, while being ignored by version 6 and below of Opera and IE on Windows.

head:first-child+body {

background-image: url("bg.png");

}

If you only want to target Opera 6 and below, you need to combine the Owen hack with the child selector hack. Say you wanted to display an upgrade notice to Opera 6 users. You would first use the child selector hack to show your upgrade message to every browser except IE 6 and below on Windows. You could then use the Owen hack to hide the message from more modern browsers:

html>body #getFirefox {

display: static;

}

head:first-child+body #getFirefox {

display: none;

}

* 1. Bug Fixing
     1. Having Layout

We all know that browsers can be buggy, and IE on Windows seems buggier than most. One of the reasons IE/Win behaves differently from other browsers is because the rendering engine uses an internal concept called “layout.”

Internet Explorer on Windows uses the layout concept to control the size and positioning of elements. Elements that are said to “have layout” are responsible for sizing and positioning themselves and their children. If an element does not “have layout,” its size and position are controlled by the nearest ancestor with layout. The layout concept is a hack used by IE’s rendering engine to reduce its processing overhead. Ideally all elements would be in control of their own size and positioning. However, this causes huge performance problems in IE. As such, the IE/Win development team decided that by applying layout only to those elements that actually needed it, they could reduce the performance overhead substantially.

Elements that have layout by default include:

* body
* html in standards mode
* table
* tr, td
* img
* hr
* input, select, textarea, button
* iframe, embed, object, applet
* marquee

The concept of layout is specific to IE on Windows, and is not a CSS property. Layout cannot be explicitly set in the CSS, although setting certain CSS properties will give an element layout. It is possible to see if an element has layout by using the JavaScript function, hasLayout. This will return true if the element has layout and false if it doesn’t. hasLayout is a read-only property and so cannot be set using JavaScript.

Setting the following CSS properties will automatically give that element layout:

* position: absolute
* float: left or right
* display: inline-block
* width: any value
* height: any value
* zoom: any value (Microsoft property—doesn’t validate)
* writing-mode: tb-rl (Microsoft property—doesn’t validate)

**Layout Effect**

Layout is the cause of many IE/Win rendering bugs. For instance, if you have a paragraph of text next to a floated element, the text is supposed to flow around the element. However, in IE 6 and below on Windows, if the paragraph has layout—by setting the height, for example—it is constrained to a rectangular shape, stopping the text from flowing around the float

This can cause all kinds of problems with floated layouts. Worse still, many people who use IE as their main browser mistakenly assume this is the correct behavior and get confused when other browsers treat floats differently.

Another problem revolves around how elements with layout size themselves. If the content of an element becomes larger than the element itself, the content is supposed to flow out of the element. However, in IE 6 and below on Windows, elements with layout incorrectly grow to fit the size of their contents

This means that width in IE/Win actually acts more like a min-width. This behavior is also the cause of many broken floated layouts in IE/Win. When the content of a floated box incorrectly forces the width of the box to grow, the box becomes too big for the available space and drops below the other floated elements.

Other problems include:

* Elements with layout not shrinking to fit
* Floats being auto-cleared by layout elements
* Relatively positioned elements not gaining layout
* Margins not collapsing between elements with layout
* The hit area of block-level links without layout only covering the text

1. References

* [HTML - Wikipedia, the free encyclopedia](http://www.google.com/url?sa=t&source=web&cd=1&ved=0CBYQFjAA&url=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FHTML&rct=j&q=wiki%20html&ei=2-OWTMfqNIzksQPxraXlCQ&usg=AFQjCNHz5BTKb46OwND1wWTg2oGRj1X2Qg&sig2=Sl3S-zfDPimj0Tzjgiitnw&cad=rja)
* [HTML element - Wikipedia, the free encyclopedia](http://en.wikipedia.org/wiki/HTML_element)
* [HTML Tutorial](http://www.w3schools.com/html/default.asp)
* [Index of the HTML 4 Elements](http://www.w3.org/TR/REC-html40/index/elements.html)
* [A List Apart: Articles: Semantics in HTML 5](http://www.alistapart.com/articles/semanticsinhtml5)
* [Dive Into HTML5](http://diveintohtml5.org/)
* [Cascading Style Sheets - Wikipedia, the free encyclopedia](http://en.wikipedia.org/wiki/Cascading_Style_Sheets)
* [Cascading Style Sheets](http://www.w3.org/Style/CSS/)
* [Eric Meyer: CSS](http://meyerweb.com/eric/css/)
* [Conditional-CSS](http://www.conditional-css.com/)
* [CSS - Conditional comments](http://www.quirksmode.org/css/condcom.html)
* [Targeting IE Using Conditional Comments](http://www.positioniseverything.net/articles/cc-plus.html)
* [CSS Font-Size: em vs. px vs. pt vs. percent | Kyle Schaeffer](http://kyleschaeffer.com/best-practices/css-font-size-em-vs-px-vs-pt-vs/)
* [CSS Mastery: Advanced Web Standards Solutions Second Edition](http://www.google.com/url?sa=t&source=web&cd=1&sqi=2&ved=0CCgQFjAA&url=http%3A%2F%2Fwww.cssmastery.com%2F&rct=j&q=css%20mastery&ei=ZOSWTKb6GZSosQOo3JDACg&usg=AFQjCNFGyTMbINamTGKOAclprtS19MsiHw&sig2=MzRV7mmnja4G4e0lzSNUMQ&cad=rja)
* Freeman, Adam. The Definitive Guide to HTML5. Apress. 2011