A Minimal HTML

L. Erawan

A Minimal HTML Document

I am often surprised by just how many professionally-designed sites are delivered in the form of incomplete HTML documents. To be fair, however, the amount of code required for even an *empty* HTML document has grown significantly over the years.

One upon a time, an HTML document only had to contain a <!DOCTYPE> declaration and a <title> tag. From the HTML 3.2 recommendation:

In practice, the HTML, HEAD and BODY start and end tags can be omitted from the markup […]

Every HTML 3.2 document must also include the descriptive title element. A minimal HTML 3.2 document thus looks like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<TITLE>A study of population dynamics</TITLE>
```

At the time HTML 3.2 was the recommended spec, very few web designers bothered with a <!DOCTYPE>, or with valid code at all, so in practice an HTML document could be any text file containing any combination of text and HTML tags.

These days, the needs of accessibility, search engine optimization, document consistency for JavaScript manipulation, and support for international characters all combine to require more of our HTML. The minimal HTML document has gotten a lot bigger.

Here's the very minimum that an HTML 4 document should contain, assuming it has CSS and JavaScript linked to it:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
charset=utf-8">
    <title>title</title>
    <link rel="stylesheet" type="text/css" href="style.css">
    <script type="text/javascript" src="script.js"></script>
  </head>
  <body>

  </body>
</html>
```

If you want to be able to process your document as XML, then this minimal XHTML 1 document should be your starting point instead:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en"
xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
charset=utf-8"/>
    <title>title</title>
    <link rel="stylesheet" type="text/css" href="style.css"/>
    <script type="text/javascript" src="script.js"></script>
  </head>
  <body>

  </body>
</html>
```

Read on below for a description of each line of these minimal documents.

THE BREAKDOWN

Every (X)HTML document should start with a <!DOCTYPE> declaration that tells the browser what version of (X)HTML the document is written in. In practical terms, this tells browsers like Internet Explorer and Firefox to use their most standards-compliant (and therefore cross-browser-compatible) rendering mode. The exact form of the <!DOCTYPE> declaration depends on whether your document is HTML 4 (fine for most purposes) or XHTML 1 (enables the page to be processed as XML):

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

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

Next, we mark the start of the document with the opening <html> tag. This tag should should specify the primary language for the document's content, with the lang attribute:

```
<html lang="en">
```

In an XHTML document, you should also specify the document's default XML namespace (using the xmlns attribute) and re-specify the language using XML's standard xml:lang attribute:

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en"
xml:lang="en">
```

Next comes the <head> tag, which starts the document header:

```
<head>
```

The first thing in the header should be a <meta> tag that specifies the character encoding of the page. Usually, the character encoding is declared by the web server that sends the page to the browser, but many servers are not configured to send this information, and specifying it here ensures the document is displayed correctly even when it is loaded directly from disk, without consulting a server:

```
<meta http-equiv="content-type" content="text/html; charset=utf-
8">
```

In an XHTML document, <meta> tags should end with a slash to indicate they are empty:

```
<meta http-equiv="content-type" content="text/html; charset=utf-
8"/>
```

With the encoding established, we can safely write the first piece of actual content in the page—the page title:

```
<title>title</title>
```

If you want to link a CSS file to the page to control its appearance (which you usually will), a <link> tag at this point will do the trick:

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

Again, the XHTML version of this tag needs a trailing slash to indicate it is empty:

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

If you want to link a JavaScript script to the page, and the script is designed to be invoked from the header, insert a <script> tag at this point. Whether the document is HTML or XHTML, you should include a full </script> closing tag for backwards compatibility:

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

That just about does it. You can end the header, then start the body of the page with a <body> tag. The content of the page is up to you, but since we're talking about a minimal document, there need not be any body content at all:

```
</head>
<body>

</body>
</html>
```

So, how does your most recent work hold up? Have you included all the elements discussed above? Common omissions like the lang attribute and the content-type <meta> tag may seem

unnecessary, but they really put that final layer of polish that ensure your site holds up with the best on the Web.

DOCTYPES

The doctype declaration, which should be the first item to appear in the source markup of any web page, is an instruction to the web browser (or other user agent) that identifies the version of the markup language in which the page is written. It refers to a known Document Type Definition, or DTD for short. The DTD sets out the rules and grammar for that flavor of markup, enabling the browser to render the content accordingly.

The doctype contains a lot of information, none of which you will be likely to find yourself being tested on in a job interview, so don't worry if it all seems too difficult to remember. Besides, most web authoring packages will insert a syntactically correct doctype for you anyway, so there's little chance of you getting it wrong.

The doctype begins with the string `<!DOCTYPE`, which should be written in uppercase:

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

The next part, which reads `html` (for XHTML) or `HTML`, refers to the name of the root element for the document. This information is included for validation purposes, since the DTD itself doesn't say which element is the root element in the document tree:

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

The `PUBLIC` statement informs the browser that the DTD is a publicly available resource. If you had decided that the various flavors of HTML or XHTML were lacking in some way, and you wanted to extend the language beyond the defined specifications, you could go to the effort of creating a custom DTD. This would allow you to define custom elements, and would enable your documents to validate according to that DTD; in this case, you'd change the `PUBLIC` value to `SYSTEM`. That said, I've never actually seen an author do this—most people live within the limitations of the defined HTML/XHTML specifications (or plug the gaps using Microformats). Here's the `PUBLIC` statement:

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

The next section is known as the Public Identifier, and provides information about the owner or guardian of the DTD—in this case, the W3C. The Public Identifier, which is shown here, is ***not*** case sensitive: it also includes the level of the language that the DTD refers to (XHTML 1.0), and identifies the language of the DTD—***not*** the content of the web page, it's important to note. This language is defined as English, or `EN` for short. Authors should not change this `EN` reference, regardless of the language contained in the web page.

Note that if the doctype contains the keyword `SYSTEM`, the Public Identifier section is omitted.

All of this information is highlighted in the short fragment below:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Together, these two parts constitute the Formal Public Identifier (or FPI).

Finally, the doctype includes a URL, known as the Formal System Identifier (FSI), which refers to the location of the DTD. If you want to really geek out, you can copy and paste the address into your web browser's location bar and download a copy of the DTD, but be warned that it doesn't make for light reading! Here's the FSI:

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

Table 1 shows the doctypes available in the WC3 recommendations.

Table 1. Available Doctypes

Doctype	Description
`<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">`	HTML 4.01 Strict allows the inclusion of structural and semantic markup, but not presentational or deprecated elements such as `font`. `frameset`s are not allowed.
`<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">`	HTML 4.01 Transitional allows the use of structural and semantic markup ***as well as*** presentational elements (such as `font`), which are deprecated in Strict. `frameset`s are not allowed. Authors should use the Strict DTD when possible, but may use the Transitional DTD when support for presentation attributes and elements is required.
`<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/html4/frameset.dtd">`	HTML 4.01 Frameset applies the same rules as HTML 4.01 Transitional, but allows the use of `frameset` content.
`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">`	XHTML 1.0 Strict, like HTML4.01 Strict, allows the use of structural and semantic markup, but not presentational or deprecated elements (such as `font`); `frameset`s are not allowed. Unlike HTML 4.01, the markup must be written as well-formed XML.

`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`	XHTML 1.0 Transitional, like HTML4.01 Transitional, allows the use of structural and semantic markup ***as well as*** presentational elements (such as `font`), which are deprecated in Strict; `frameset`s are not allowed. Unlike HTML 4.01, the markup must be written as well-formed XML. Authors should use the Strict DTD when possible, but may use the Transitional DTD when support for presentation attributes and elements is required.
`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">`	XHTML 1.0 Frameset applies the same rules as XHTML 1.0 Transitional, but also allows the use of `frameset` content.
`<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">`	XHTML 1.1 is a reformulation of XHTML 1.0 Strict, thus most of the same rules apply. However, 1.1 allows for modularization, which means that you can add modules (for example, to provide Ruby support for Chinese, Japanese, and Korean characters).
`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">`	HTML 3.2 is an archaic doctype that's no longer recommended for use (it's included here for information only).
`<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.0//EN">`	HTML 3.0 is an archaic doctype that's no longer recommended for use (it's included here for information only).
`<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0 Level 2//EN">`	HTML 2.0 is an archaic doctype that's no longer recommended for use (it's included here for information only). Note that there are

	actually 12 variants of this old doctype, all of which can be found Section 9.6 of RFC1866.

DOCTYPE SWITCHING OR SNIFFING

The way in which a web browser renders a page's content is often affected by the doctype that's defined. Browsers use various modes to determine how to render a web page:

- *Quirks Mode*

  In this mode, browsers violate normal web formatting specifications as a way to avoid the poor rendering (or "breaking," to use the vernacular) of pages that have been written using practices that were commonplace in the late 1990s. The quirks differ from browser to browser. In Internet Explorer 6 and 7, the Quirks Mode displays the document as if it were being viewed in IE version 5.5. In other browsers, Quirks Mode contains a selection of deviations that are taken from Almost Standards mode (explained below).

- *Standards Mode*

  In this mode, browsers attempt to give conforming documents an exact treatment according to the specification (but this is still dependent on the extent to which the standards are implemented in a given browser).

- *Almost Standards Mode*

  Firefox, Safari, and Opera (version 7.5 and above) add a third mode, which is known as Almost Standards Mode. This mode implements the vertical sizing of table cells in a traditional fashion—not rigorously, as defined in the CSS2 specification. (Internet Explorer versions 6 and 7 don't need an Almost Standards Mode, because they don't implement the vertical sizing of table cells rigorously, according to the CSS2 specification, in their respective Standards Modes).

Depending on the doctype that's defined, and the level of detail contained inside the doctype (for example, whether it does or doesn't include a Public Identifier), different browsers trigger different modes from the list above. Doctype switching or sniffing refers to the task of swapping one doctype for another, or changing the level of detail in the doctype, in order to coax a browser to render in one of Quirks, Standards, or Almost Standards Modes.

An example of a situation in which doctype sniffing was put to use most frequently was to address rendering differences between Internet Explorer 6 and earlier versions of the browser, which calculated content widths differently when widths, padding, borders, and margins were applied in CSS. (This topic is not something we'll cover in this HTML reference, but you can find out more in The Ultimate CSS reference.) In Internet Explorer 6, depending on the doctype defined, one of two different rendering modes would be used to calculate these widths.

As an example, imagine that you specify the doctype as HTML 4.01 Strict, like so:

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

In IE6, the doctype above will cause the browser to render in Standards Mode, which includes using the W3C method for box model calculations. However, you see an entirely different result if you use the following doctype:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

In this scenario, IE6 will use its old, incorrect, non-W3C method for making box model calculations (also known as the 'broken box model').

Note that this is not the ***only*** difference between Quirks and Standards Mode—it's ***just one*** example of the differences between the two modes (but one that caused a great deal of problems because of the disastrous effect it could have on page layout).

For a complete reference of how different browsers behave when different doctypes are provided, refer to the chart at the foot of Henri Sivonen's article "Activating Browser Modes with Doctype."

HTML *(HTML ELEMENT)*

Spec		
Depr.	**Empty**	**Version**
No	No	HTML 2

Browser support				
IE5.5+	**FF1+**	**SA1.3+**	**OP9.2+**	**CH2+**
Full	Full	Full	Full	Full

SYNTAX

```
<html xmlns="http://www.w3.org/1999/xhtml">

</html>
```

DESCRIPTION

The `html` element is the outer container for everything that appears in an HTML (or XHTML) document. It can only contain two elements as direct descendants, namely the `head` element and either a `body` or `frameset` element. It can also contain comments. As it is the outermost element in the document, it's also known as the root element.

Note that the `xmlns` is required for XHTML page , but is invalid in HTML pages.

Although the `html` element is the outermost element, it does ***not*** contain the doctype—the doctype must come ***before*** the `html` element.

EXAMPLE

This example shows the `html` element surrounding all elements in a document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Page title goes here</title>
  </head>
  <body>
    <!-- content goes here -->
  </body>
</html>
```

USE THIS FOR ...

This element will have a place on every single web page you ever create! There's no case in which you would ***not*** use this element when crafting a web page.

COMPATIBILITY

Internet Explorer				Firefox					Safari				Opera			Chrome
5.5	6.0	7.0	8.0	1.0	1.5	2.0	3.0	3.5	1.3	2.0	3.1	4.0	9.2	9.5	10.0	2.0
Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full

Every browser listed supports this element type.

IN THIS SECTION

- **xmlns**
  declares the URI of the default namespace for a document. In XHTML documents this will always have the value http://www.w3.org/1999/xhtml

ATRIBUT LANG

DESCRIPTION

The `lang` attribute is used to identify the language of the content on a web page, when it's applied to the `html` element, or within a given section on a web page, when it's applied to a `div`, `span`, `a`, or any other element that may contain content in a language that differs from that used on the rest of the page. Changing the language at a lower level in the document tree overrides the language code that's set higher up, but ***only*** for the nested element to which the different language code is applied.

The intention of the `lang` attribute is to allow browsers (and other user agents) to adjust their displays. For example, if you use the `q` element, a browser should be able to identify the language in use and present the appropriate style of quotation marks. In practice, though, poor support for the `q` element sees few developers use it—instead, many hard-code quotation marks. This is something of a catch-22 situation.

The `lang` attribute is helpful to users of assistive technology such as screen readers that can adjust the pronunciation depending on the language used. For example, the word ***penchant***, meaning "a strong and continued inclination," is French in origin. When the screen reader JAWS encounters the word, it pronounces it similar to "pen-chunt," but when the word is marked up as `<span lang="fr">penchant</span>`, JAWS reads it using the proper French pronunciation, "pon-shont."

It may also be possible that marking up documents or sections of a document in this way benefits search engines that display results to users who have filtered their searches on the basis of language preferences. However, the way in which search engines ***actually*** deal with content marked up in this way is a secret they tend to keep to themselves. As such, the potential for this attribute to make content more search engine friendly should be considered little more than a lucky bonus.

Note that the `lang` attribute cannot be applied to the following elements:

- `applet`
- `base`
- `basefont`
- `br`
- `frame`
- `frameset`
- `iframe`
- `param`
- `script`

EXAMPLE

This markup specifies the language of a page as English:

```
<html lang="en">
```

VALUE

`lang` takes as its value approved International Standards Organisation (ISO) 2 letter language codes only (refer to the language codes reference for details).

COMPATIBILITY

Internet Explorer			Firefox			Safari			Opera	
5.5	6.0	7.0	1.0	1.5	2.0	1.3	2.0	3.0	9.2	9.5
Partial	Partial	Partial	Partial	Partial	Partial	Partial	Partial	Partial	Partial	Partial

As I mentioned in the description above, browsers don't provide great support for the `lang` attribute, but it offers benefits that extend beyond the browser itself (including advantages for search and assistive technology). Language should be indicated for these reasons, as well as for the purpose of forwards compatibility.

HEAD *(HTML ELEMENT)*

Spec		
Depr.	**Empty**	**Version**
No	Yes	HTML 2

Browser support

IE5.5+	FF1+	SA1.3+	OP9.2+	CH2+
Full	Full	Full	Full	Full

SYNTAX

```
<head>

</head>
```

DESCRIPTION

The `head` element is the wrapper for all the head elements that, collectively, instruct the browser where to find style sheets, or define relationships that the document has to others in the web site (`link`); provide essential meta information (`meta`); and point to or include scripts that the document will need to apply later on (`script`).

The `head` element also includes the `title` element, which is the only `head` element that is absolutely required.

EXAMPLE

This example shows the `head`, which contains the required `title` element:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Page title goes here</title>
  </head>
  <body>
    <!-- content goes here -->
 </body>
</html>
```

COMPATIBILITY

Internet Explorer				Firefox					Safari				Opera			Chrome
5.5	6.0	7.0	8.0	1.0	1.5	2.0	3.0	3.5	1.3	2.0	3.1	4.0	9.2	9.5	10.0	2.0
Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full

Every browser listed supports this element type.

IN THIS SECTION

- **profile**
  provides references to additional information that meta elements may require for disambiguation

META *(HTML ELEMENT)*

Spec		
Depr.	**Empty**	**Version**
No	Yes	HTML 2

Browser support				
IE5.5+	**FF1+**	**SA1.3+**	**OP9.2+**	**CH2+**
Full	Full	Full	Full	Full

SYNTAX

```
<meta content="string" {http-equiv="http response header"|name="string"} />
```

DESCRIPTION

The `meta` element provides information about the following document content; that information may be used by the user agent (that is, the browser) to decide how to render content, or it may be meta information that's provided for indexing purposes—for example, to provide keywords that relate to the document for use by search engines or some other form of web service. The `meta` element can also be used to simulate HTTP response headers (the character encoding snippet provided here is an example of this), or it might simply be used for the purposes of causing a document to reload itself after a set interval.

There's not a standard list of `meta` properties (although the Dublin Core initiative aims to correct that), so it would be perfectly valid to define the following `meta` properties:

```
<meta name="department" content="Technology">
<meta name="author" content="John Smith">
```

This definition would also be perfectly valid:

```
<meta name="appearance" content="fluffy">
```

While you can define your own `meta` information, the question you should ask is whether or not it adds any value. Unless you intend to use this element for the purposes of indexing, you're advised to stick to a handful of tried-and-tested meta tags:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<!– sets character encoding –>

<meta http-equiv="Refresh" content="5">
<!– refreshes the page content every five seconds –>
```

The following meta tags are the most relevant in terms of search engine indexing and interpretation:[1]

```
<meta name="robots" content="noindex,follow">
<!— instructs search engines not to index this page, but to follow
links from the page —>

<meta name="description" content="A brief
    summary/description of the content on the page,
    which may appear in search engine results">

<meta name="keywords" content="reference, SitePoint,
    HTML, XHTML, standards">
<!—comma-separated list of keywords that apply to the page —>
```

The search engines' declining interest in the `"description"` and `"keywords"` `meta` elements may mean that it's preferable for us to drop them entirely. As with all `meta` elements, there is a danger that, because they are hidden from normal page view, the content can easily go out of date. So unless you know ***for a fact*** that the content in these types of `meta` elements is going to be properly maintained ***and*** that they are in some way beneficial (and this is more likely to be the case when they're used in an intranet where this information may be properly indexed), not using them at all may be the best course of action.

Note: Optimizing meta Elements? Don't Waste Your Time!

Many search engine optimization (SEO) experts still insist that there's some value in including the `"description"` and `"keywords"` `meta` elements. However, most of these specialists are not party to the ways in which the various search engines actually work. Their advice is largely based on experience, assumption, reverse engineering, common sense, observed patterns—the list goes on. In short, they can't be ***certain*** how the likes of Google, Yahoo, Live, Ask, and others handle this content. Certainly, evidence suggests that your efforts are best spent on other aspects of the document whose search ranking you wish to improve.

The syntax of the examples above is appropriate for HTML documents. For XHTML documents, you must remember to include the trailing `/` character (although it should be noted that there would be little point in adding a Content-Type header as `content="text/html; charset=UTF-8"` if the document is supposed to be XHTML!). This example shows the XHTML syntax:

```
<meta name="robots" content="noindex,follow"/>
<!— the robots meta tag expressed in XHTML syntax —>
```

The `"robots"` meta tag is used to tell robots (that is, search engine crawlers) not to index the content of a page, nor to scan it for links to follow. Different search engines interpret the `"robots"` meta tag differently. For more information about the potential attribute values and their uses, refer to the Search Engine Land article "Meta Robots Tag 101: Blocking Spiders, Cached Pages & More."

EXAMPLE

Here, a `meta` element is used to identify character encoding in an HTML document:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

USE THIS FOR ...

The `meta` element provides information about the document that won't render on the page, but will be machine-parsable.

COMPATIBILITY

Internet Explorer				Firefox					Safari				Opera			Chrome
5.5	6.0	7.0	8.0	1.0	1.5	2.0	3.0	3.5	1.3	2.0	3.1	4.0	9.2	9.5	10.0	2.0
Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full

Full support is provided for the `meta` element—all browsers expose a document's meta information via the Document Object Model.

IN THIS SECTION

- **content**
  specifies the content of the meta information defined by the name attribute
- **http-equiv**
  binds content in meta information to an HTTP header
- **name**
  associates a name with the meta information via the content attribute
- **scheme**
  provides a scheme name for interpreting the meta information

TITLE *(HTML ELEMENT)*

Spec		
Depr.	**Empty**	**Version**
No	No	HTML 2

Browser support				
IE5.5+	**FF1+**	**SA1.3+**	**OP9.2+**	**CH2+**
Full	Full	Full	Full	Full

SYNTAX

```
<title>

</title>
```

DESCRIPTION

The `title` element is arguably one of the most important elements in the whole document—and it's a required element for all flavors of HTML/XHTML. The contents of this element are used for all of the following purposes:

- displaying a title in the browser toolbar or in the task bar (on Windows)
- providing for the document a name that's used by the browser when you add the page as a favorite or bookmark
- displaying a title of the page when it appears in search engine results (this is reason enough to take time to ensure that you create a sensible title; otherwise, you risk not being found in search engines for the appropriate search phrase)

For web sites that are hand-coded, and maintained on a page-by-page basis by the web developer, this element can easily fall foul of copy/paste actions (for example, you may all too easily create a new page about products by copying the Press Information page and forgetting to amend the contents of the `title` element to reflect the new content).

Here's one tip regarding the contents of the title element: it's prudent to include your company or organization name in the `title` for all pages; however, avoid front-loading the title with this phrase. For example, if your company is the XYZ Corp, avoid the following:

- XYZ Corp—Our water treatment products
- XYZ Corp—About the company
- XYZ Corp—Contact us

When pages are bookmarked, they'll appear in one alphabetical block. It would be preferable to use the following title content:

- Water treatment products—XYZ Corp
- About XYZ Corp
- Contact us—XYZ Corp

EXAMPLE

This code shows the `title` in action:

```
<head>
  <title>101 ways to skin a cat - the tutorial!</title>
</head>
```

USE THIS FOR ...

The `title` is used to summarize the content or purpose of the document. This content represents "elevator pitch" for your document, so be succinct and meaningful, and avoid stuffing keywords into the title for the sake of search results alone.

COMPATIBILITY

Internet Explorer				Firefox					Safari				Opera			Chrome
5.5	6.0	7.0	8.0	1.0	1.5	2.0	3.0	3.5	1.3	2.0	3.1	4.0	9.2	9.5	10.0	2.0
Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full

Every browser listed supports this element type.

SCRIPT *(HTML ELEMENT)*

Spec				
Depr.		**Empty**		**Version**
No	No		HTML 4	
Browser support				
IE5.5+	**FF1+**	**SA1.3+**	**OP9.2+**	**CH2+**
Full	Full	Full	Full	Full

SYNTAX

```
<script src="uri" type="MIME type">

</script>
```

DESCRIPTION

The `script` element is used to enclose a series of statements in a scripting language that's processed on the client side (that is, it's processed by the user's computer, rather than being processed on the server before being sent to the user's computer). The language that's used may be JavaScript or the Internet Explorer-specific VBScript. These days, however, it's extremely rare to find examples of VBScript used in client-side scripting, except for web pages or applications that are used on intranets whose client base uses Internet Explorer exclusively (and even then, it's not a good idea to do this!).

As well as enclosing scripting statements within the opening `<script>` and closing `</script>` tags, you can use this element to refer to an external script file through the `src` attribute, usually saved with the .js extension, which allows scripts to be shared across an entire site easily.

It's common practice to place all JavaScript functions within `script` elements inside the `head` element, from where they're available for use by all the markup on the page that follows. In fact, you can place a `script` element anywhere on a page, as shown in the following example (whereby a username is dynamically written into the page using an existing JavaScript variable):

```
<p>Well, hello there, <script>writeUserName();</script>,
    how's your day been so far?</p>
```

Note that there are some important differences in the way that HTML 4 and XHTML 1 deal with the content inside scripts.

In HTML 4, the content type is declared as CDATA, which means entity references won't be parsed. The first occurrence of `</` followed by a name start character actually terminates the `script` element, so something like this should fail: `document.write("<p>Hello!</p>")`. In practice, although it's invalid, browsers recover from that error and don't terminate the script until they reach the `</script>` tag. Actually, using the `document.write()` statement is, in itself, a problem in XHTML: `document.write()` can't be used in XHTML documents that

are served as XML, due to the way XML is parsed and processed (using it modifies the input stream in a way that isn't compatible with XML parsing requirements).

In an XHTML 1 document that's properly served using the MIME type `"application/xhtml+xml"`, the content type is declared as (`#PCDATA`), so entities will be parsed and only a `</script>` tag will terminate the element. This means special characters need to be encoded—for example, ampersands will be encoded as `&`, and greater-than symbols will be encoded as `>`—or all content should be wrapped inside `<![CDATA[ … ]]>` sections. To ***ensure*** that the content inside the opening `<script>` and closing `</script>` tags is parsed correctly when it's included within an XHTML document, use the following comment syntax:

```
<script type="text/javascript"><![CDATA[
  //script goes here
  ⋮
  //]]></script>
```

It's not advisable to use the HTML comment syntax `<!-- -->` to hide script content from older browsers.

EXAMPLE

This script runs after the page has loaded:

```
<script type="text/javascript">
  function doSomethingClever() {
    //clever script goes here
   ⋮
  }
  window.onload = doSomethingClever;
</script>
```

USE THIS FOR ...

The uses for the `script` element are many and varied; in fact, they're limited only by your imagination, your scripting skills, and your computer's capabilities. Typical uses for JavaScript include image swapping and manipulation, making dynamic changes to content, drag-and-drop functionality, form validation, and so on. This topic really deserves a whole book in its own right.

COMPATIBILITY

Internet Explorer				Firefox					Safari				Opera			Chrome
5.5	6.0	7.0	8.0	1.0	1.5	2.0	3.0	3.5	1.3	2.0	3.1	4.0	9.2	9.5	10.0	2.0
Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full	Full

Support for embedding the `script` element on the page is not an issue. However, the way that different browsers handle the content—the scripting language itself—can vary

massively, depending on the content that's contained. This discussion is well beyond the scope of this reference.

IN THIS SECTION

- **charset**
  identifies the character encoding of a linked script
- **defer**
  instructs browser to defer the execution of a script
- **language**
  specifies the scripting language being used
- **src**
  specifies the location of an external script file
- **type**
  specifies the MIME type of the script element's content
- **xml:space**
  specifies whether whitespace in code should be preserved

DO YOU KNOW YOUR CHARACTER ENCODINGS?

Last month, I attended a meeting of the Melbourne chapter of the Web Standards Group, where Richard Ishida, the Internationalization Activity Lead of the W3C gave a remarkably clear presentation of one of the most ignored issues in web development: character encodings.

Have you ever noticed certain characters on your site not displaying the way they should? Perhaps the curly quotation marks look like little boxes, or the long dashes have been replaced with question marks. Problems like these usually arise from an incomplete understanding of character encodings on the part of the developer responsible for the site.

I'd go so far as to guess that, in English speaking circles at least, most web developers that have never learned about character encodings, and just deal with the consequences when issues like the above crop up.

As a site grows to the point where it must address an international audience (or even just an audience that likes curly quotes), however, it's more and more difficult to ignore these issues. Even worse, in these heady times of daily hack attempts, incorrect handling of character encodings can result in severe security vulnerabilities (as Google recently discovered).

So what *is* a character encoding, exactly? Well, let's start with something it's not: a character encoding is not a character set.

CHARACTER SETS

A character set, or more specifically, a **coded character set** is a set of character symbols, each of which has a unique numerical ID, which is called the character's **code point**.

Some examples of character sets include the 128-character ASCII character set, which is mostly made up of the letters, numbers, and punctuation used in the English language, and the 256-character ISO-8859-1, or Latin 1 character set, which includes all the ASCII characters plus accented and other additional characters used in related languages like French. The most expansive character set in common use is the Universal Character Set (UCS), as defined in the Unicode standard, which contains over 1.1 million code points.

The first thing to understand is that **every HTML document uses Unicode's UCS**, or more accurately the ISO 10646 character set, which is a less involved standard describing the same set of characters. Some older browsers, or less powerful devices, may not support (and thus will not display) the complete character set, but the fact remains that any HTML document may contain any character in the UCS.

What *does* vary from document to document is the **character encoding**, which defines how each of the characters in the UCS is to be represented as one or more bytes in the text data of the page.

This figure shows the ASCII, ISO-8859-1, and Unicode code points for three characters (the letter ‘A’, the acute-accented letter ‘e’, and the Hebrew letter ‘alef’), and how those characters map to a series of bytes in five common character encodings:

	Characters:	A	é	א
character sets (code points (hexadecimal))	ASCII	65 (41)	n/a	n/a
	ISO-8859-1 (Latin 1)	65 (41)	233 (E9)	n/a
	Unicode (UCS)	65 (41)	233 (E9)	1488 (05D0)
character encodings (hex byte values)	7-bit ASCII	41	n/a	n/a
	ISO-8859-1	41	E9	n/a
	UTF-8	41	C3 A9	D7 90
	UTF-16	00 41	00 E9	05 D0
	UTF-32	00 00 00 41	00 00 00 E9	00 00 05 D0

Looking first at the character sets, note how the letter ‘A’ is available as a character in all three character sets, but the acute ‘e’ isn’t available in ASCII, and ‘alef’ is only available in Unicode. The fact that characters maintain the same code points across multiple character encodings is due to the fact that ISO-8859-1 was designed as an extension of ASCII, and Unicode in turn was designed as an extension to ISO-8859-1. There are certainly other character sets where the code points of these characters, where they exist, would differ.

As I mentioned above, however, web pages always use the Unicode character set, so these code points are the only ones that matter for the purposes of web development.

CHARACTER ENCODINGS

Now take a look at the character encodings in the figure. The first, 7-bit ASCII, dates back long before the days of MS-DOS, and is commonly used today as a "lowest common denominator" in email systems. If an email message contains only characters from the ASCII character set, and those characters are encoded as per their ASCII code points (e.g. the letter A is code point 65, which in hexadecimal (base-16) is 41, so the byte value used to represent it should be 41), then it should be compatible with any Internet email system, no matter how obsolete. Because ASCII contains only 128 code points, only seven of the eight bits in a byte are needed to represent any ASCII character. The byte values in a 7-bit ASCII document will therefore never exceed 7F (that’s 127 in base-10).

ISO-8859-1 is the default encoding assumed by many browsers and related English-language software. It uses all eight bits of each byte to represent all 256 code points in the ISO-8859-1 character set. Though this provides the characters required for the vast majority of English language documents, as well as documents in many related languages like French, there are plenty of languages that are based on characters not included in this set. Even certain specialized characters in English documents, curly quotes and long dashes for instance, are not a part of ISO-8859-1. This explains why such characters are most often responsible for revealing a character encoding problem.

To serve the needs of other languages, there are an abundance of character encodings like ISO-8859-1 that make use of the possible byte values to represent a set of 256 characters. Additionally, there are a number of character encodings that use *two* bytes per character to allow for 65,536 different characters. Commonly used for Chinese and other languages requiring a large number of characters, these encodings are called **double-byte character sets** (DBCS), even though they are in fact encodings.

But for documents that may contain characters from any language, the best encodings are those that can address Unicode's entire UCS. The simplest of these is UTF-32, which simply uses four bytes to represent each UCS character by its code point. 'A', which is code point 65 (41 hex) is represented by the four byte values 00 00 00 41, the acute 'e' (code point E9 hex) is 00 00 00 E9, and 'alef' (05D0 hex) is 00 00 05 D0.

The problem with UTF-32 is that, because the vast majority of characters in documents occur early in the UCS, almost every character in a given document will begin with two 00 bytes, which is quite a waste. Effectively, most UTF-32 documents will be four times the size of the same documented encoded in a single-byte encoding like ISO-8859-1.

The UTF-8 and UTF-16 encodings address this by using a variable number of bytes per character. In UTF-8, the most common characters use only a single byte, which is equal to that character's UCS code point, while less common characters use two, even rarer characters use three, and only the very rarest of characters use four bytes. UTF-16 accomodates a larger set of "common" characters whose two-byte encodings match their UCS code points, reserving three- and four-byte encodings for rarer characters.

Looking at the figure, you can see that the 'A' character has encodings that match its UCS code point in both UTF-8 and UTF-16. The acute 'e' and 'alef', on the other hand, are less common characters that each have a special two-byte encoding in UTF-8 that differs from its UCS code point. In UTF-16, however, both acute 'e'e and 'alef' are considered common enough to get an encoding that matches their two-byte code points (00 E9 and 05 D0, respectively).

Make sense? If you've followed this far, you've grasped all the concepts you need to work intelligently with character encodings. Keep reading to find out how all this affects your work as a web developer.

CHARACTER ENCODINGS AND THE WEB

Okay, so a character encoding specifies how a set of characters (like Unicode's UCS, which is used on the web) can be written as bytes in a stored document. So what does this mean to web developers?

As a web developer, there are two types of text data that you need to deal with: the text that makes up the pages of your site, and the text that is sent by your users' browsers (usually as a form submission). In each case, you should be aware of the character encoding that is in use, and treat that data accordingly.

It turns out that the encodings of these two bodies of text data are linked: the default encoding that a browser will use when submitting a form is governed by the encoding of the document that contained the form. A page encoded in ISO-8859-1 will submit its form data in ISO-8859-1, while a page encoded in UTF-8 will submit in UTF-8.

So the first thing you need to do is pick an appropriate encoding in whichever editor you use to create your web documents. Depending on your editor, this will involve setting a configuration option (e.g. in Dreamweaver), or simply choosing the right encoding when you first save the file (e.g. in Notepad).

You also need to tell browsers which encoding your documents are using. Browsers cannot guess the character encoding–every document just looks like a series of byte values until an encoding is provided to interpret them. So next you must declare the character encoding of each of your documents. To indicate the encoding of an HTML document, include an appropriate `<meta>` tag. For ISO-8859-1:

```
<meta http-equiv="Content-Type"
    content="text/html; charset=ISO-8859-1" />
```

For UTF-8:

```
<meta http-equiv="Content-Type"
    content="text/html; charset=UTF-8" />
```

Yes, that's right: you specify the character *encoding* with an attribute called `charset`. No wonder people find this stuff confusing!

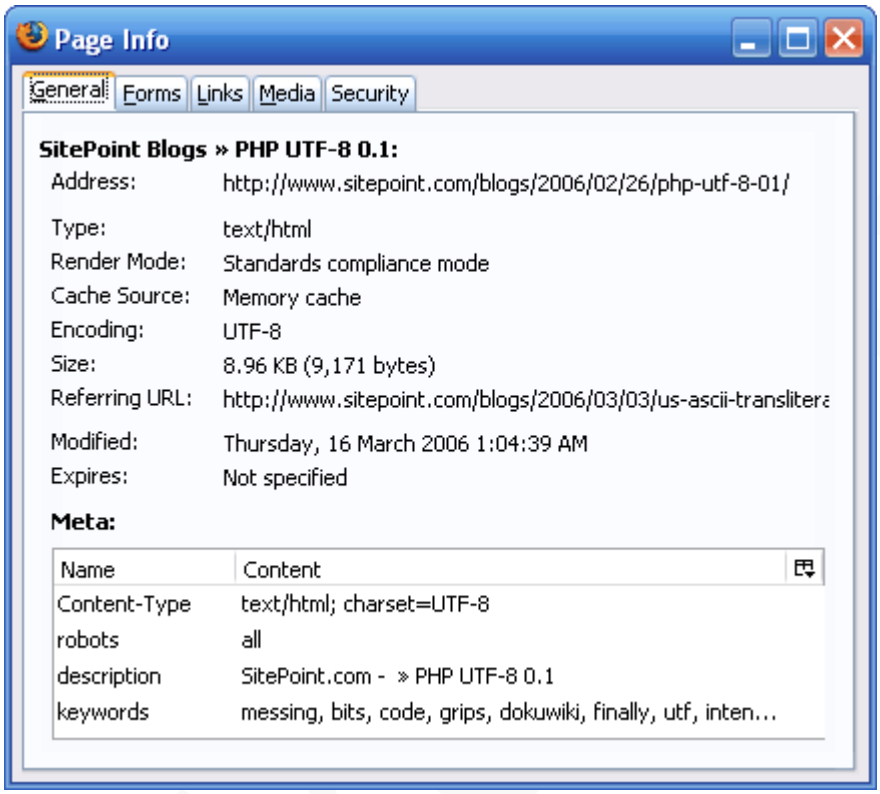
You might wonder how a browser can even read this tag if it doesn't yet know the character encoding, but it turns out that most encodings in popular use have enough characters in common that the simple HTML code leading up to this tag can usually be interpreted by guessing at a simple encoding (say ISO-8859-1), and then starting over if the tag indicates the browser has guessed wrong.

For CSS and JavaScript files, things are trickier. While the standards offer ways to indicate the encodings of these files, support for these is spotty. If you need to use characters outside the relatively safe ASCII character set in these files, you'll need to configure your web server to identify the character encoding in HTTP headers that are sent with these files. For example:

```
Content-Type: text/css; charset=UTF-8
```

You can use the HTTP header approach for HTML documents as well, but you should still include the `<meta>` tag as backup in case the document is loaded without HTTP headers (e.g. it is loaded directly from the file system with a `file://` URL).

Once you've specified an encoding, you can verify that browsers are picking up on it. Open the page in Firefox, right-click the background and choose Page Info. The window that appears will show the character encoding that was used to interpret the document.



So all this begs the question, which character set should you be using? Well, in most cases, the answer is UTF-8. It gives you access to a multitude of characters in your documents without significantly increasing the file size, and it's reasonably backwards-compatible with older browsers and simple devices that do not support Unicode. If, however, you need to use significant quantities of CJK (Chinese, Japanese or Korean) text, which will necessitate a larger character set, then you might find UTF-16 is a more efficient choice.

That is, unless you're using PHP. One of the biggest weaknesses of PHP (up to and including PHP 5.1) is that its built-in string functions handle multi-byte character encodings like UTF-8 and UTF-16 incorrectly. PHP was written with the assumption that one byte equals one character, which simply isn't the case in such encodings. An optional module or library can be used to provide alternative string functions that *do* support multi-byte characters, but many of the PHP scripts in circulation use the built-in functions, and simply can't handle Unicode characters as a result.

This problem will be addressed in PHP 6, where Unicode support will be an integral part of the language, but in the meantime getting PHP to treat Unicode correctly is something of a

black art. It's certainly *possible* to do–high quality PHP scripts like WordPress and phpBB handle Unicode quite well–but you really need to know your PHP to do it.

For this reason, PHP-based web sites are commonly written using the ISO-8859-1 encoding. SitePoint's article and forum pages, for example, are all written using ISO-8859-1.

As you can probably gather, using ISO-8859-1 has a few disadvantages. For one thing, you're limited to using that relatively small character set to write your documents. What happens when you need a curly quote, or some other character not found in the ISO-8859-1 set?

HTML's answer to this problem is the character entity. I'm sure you're familiar with these: codes like `&rdquo;` (right-hand double quotes) and `&mdash;` (em dash) let you include characters not available in your chosen encoding in your document's text. For more exotic characters that do not have an easy-to-remember code in HTML, you can use **numeric character ~~entities~~references** instead. To include the character 'alef' in an ISO-8859-1 document, for example, you would use either `&#1488;` or `&#x05d0;`, the decimal and hexadecimal versions of the character's UCS code point, respectively.

Take a moment to absorb the fact that numeric character entities refer to UCS *code points* for characters, not the byte values for characters in any particular encoding. The numeric character entity for 'alef' is the same no matter what encoding you are using in your document.

So character entities let you deal with characters outside your selected encoding when *writing* documents, but what about the other side of the coin? How do you deal with characters outside a limited encoding like ISO-8859-1 when it comes to form submissions?

Sadly, this is one place where browsers have disagreed for a long time, and even today, after much pulling of hair and gnashing of teeth, the solutions that most browsers now support are less than ideal.

One of the biggest problems is Windows, which on English language systems makes use of a slightly modified version of ISO-8859-1 called Windows-1252. Sam Ruby has documented the differences in his survival guide. Windows-1252 represents certain useful characters like curly quotes as single bytes, taking the places of less commonly used ISO-8859-1 characters. As a result, Internet Explorer browsers will often consider such characters as being *within* the document encoding, and will submit them as such. On the server, these single-byte encodings get interpreted as their ISO-8859-1 equivalents, which is what often leads to ugly boxes and other nonsense characters showing up on web pages in the place of curly quotes and the like, particularly when text entered on a Windows system is displayed on a non-Windows browser like Safari.

That exception aside, most current browsers, when faced with a character that is not in the encoding in which a form is to be submitted, will convert that character to a numeric character entity and submit that instead. This may sound sensible at first, but consider that HTML forms are supposed to submit plain text, not HTML code. Special characters like < and > are not automatically encoded as `<` and `>` for submission by forms, nor should they be. This auto-conversion of out-of-encoding characters means that, in an ISO-8859-1

document, you can't tell from the submitted form data whether the user actually typed the character 'alef', or the series of characters `&#1488;`.

Some browsers have approached this problem differently, replacing certain out-of-encoding characters with in-encoding equivalents (e.g. curly quotes with straight quotes), and replacing other problem characters with a generic substitute (e.g. '?'). While this solution is *technically* superior, you do miss out on the few cases where the more common approach described above manages to preserve the desired characters without any side-effects.

A full discussion of how different browsers tackle the problem of character encoding in form submissions would take too long to go into here, but there are good writeups available for those who go looking. In short, however, your best bet for conquering these problems is to move your site to UTF-8 (or UTF-16 if appropriate) as soon as you can.

FURTHER READING

Much of the information above in this issue is distilled from the second hour of a talk that Richard Ishida gave to the Melbourne Web Standards Group not long ago. If I've piqued your interest but you're still a bit foggy on the details, you can listen to the complete audio of that presentation, and read through his slides, enhanced with complete tutorial notes.

Once you start working with Unicode, you'll find a number of utilities on Ishida's site will come in very handy. There's a tool for browsing the complete UCS, and another for converting between Unicode characters, code points, encodings, and numeric character entities, both of which are definitely worth bookmarking.