



HTML5

[Table of contents](#) – [2 Common infrastructure](#) →

[1 Introduction](#)

[1.1 Background](#)

[1.2 Audience](#)

[1.3 Scope](#)

[1.4 History](#)

[1.5 Design notes](#)

[1.5.1 Serializability of script execution](#)

[1.5.2 Compliance with other specifications](#)

[1.6 HTML vs XHTML](#)

[1.7 Structure of this specification](#)

[1.7.1 How to read this specification](#)

[1.7.2 Typographic conventions](#)

[1.8 A quick introduction to HTML](#)

[1.9 Conformance requirements for authors](#)

[1.9.1 Presentational markup](#)

[1.9.2 Syntax errors](#)

[1.9.3 Restrictions on content models and on attribute values](#)

[1.10 Recommended reading](#)

1 Introduction

1.1 Background

This section is non-normative.

The World Wide Web's markup language has always been HTML. HTML was primarily designed as a language for semantically describing scientific documents, although its general design and adaptations over the years have enabled it to be used to describe a number of other types of documents.

The main area that has not been adequately addressed by HTML is a vague subject referred to as Web Applications. This specification attempts to rectify this, while at the same time updating the HTML specifications to address issues raised in the past few years.

1.2 Audience

This section is non-normative.

This specification is intended for authors of documents and scripts that use the features defined in this specification, implementors of tools that operate on pages that use the features defined in this specification, and individuals wishing to establish the correctness of documents or implementations with respect to the requirements of this specification.

This document is probably not suited to readers who do not already have at least a passing familiarity with Web technologies, as in places it sacrifices clarity for precision, and brevity for completeness. More approachable tutorials and authoring guides can provide a gentler introduction to the topic.

In particular, familiarity with the basics of DOM Core and DOM Events is necessary for a complete understanding of some of the more technical parts of this specification. An understanding of Web IDL, HTTP, XML, Unicode, character encodings, JavaScript, and CSS will also be helpful in places but is not essential.

1.3 Scope

This section is non-normative.

This specification is limited to providing a semantic-level markup language and associated semantic-level scripting APIs for authoring accessible pages on the Web ranging from static documents to dynamic applications.

The scope of this specification does not include providing mechanisms for media-specific customization of presentation (although default rendering rules for Web browsers are included at the end of this specification, and several mechanisms for hooking into CSS are provided as part of the language).

The scope of this specification is not to describe an entire operating system. In particular, hardware configuration software, image manipulation tools, and applications that users would be expected to use with high-end workstations on a daily basis are out of scope. In terms of applications, this specification is targeted specifically at applications that would be expected to be used by users on an occasional basis, or regularly but from disparate locations, with low CPU requirements. For instance online purchasing systems, searching systems, games (especially multiplayer online games), public telephone books or address books, communications software (e-mail clients, instant messaging clients, discussion software), document editing software, etc.

1.4 History

This section is non-normative.

For its first five years (1990-1995), HTML went through a number of revisions and experienced a number of extensions, primarily hosted first at CERN, and then at the IETF.

With the creation of the W3C, HTML's development changed venue again. A first abortive attempt at extending HTML in 1995 known as HTML 3.0 then made way to a more pragmatic approach known as HTML 3.2, which was completed in 1997. HTML4 followed, reaching completion in 1998.

At this time, the W3C membership decided to stop evolving HTML and instead begin work on an XML-based equivalent, called XHTML. This effort started with a reformulation of HTML4 in XML, known as XHTML 1.0, which added no new features except the new serialization, and which was completed in 2000. After XHTML 1.0, the W3C's focus turned to making it easier for other working groups to extend XHTML, under the banner of XHTML Modularization. In parallel with this, the W3C also worked on a new language that was not compatible with the earlier HTML and XHTML languages, calling it XHTML2.

Around the time that HTML's evolution was stopped in 1998, parts of the API for HTML developed by browser vendors were specified and published under the name DOM Level 1 (in 1998) and DOM Level 2 Core and DOM Level 2 HTML (starting in 2000 and culminating in 2003). These efforts then petered out, with some DOM Level 3 specifications published in 2004 but the working group being closed before all the Level 3 drafts were completed.

In 2003, the publication of XForms, a technology which was positioned as the next generation of Web forms, sparked a renewed interest in evolving HTML itself, rather than finding replacements for it. This interest was borne from the realization that XML's deployment as a Web technology was limited to entirely new technologies (like RSS and later Atom), rather than as a replacement for existing deployed technologies (like HTML).

A proof of concept to show that it was possible to extend HTML4's forms to provide many of the features that XForms 1.0 introduced, without requiring browsers to implement rendering engines that were incompatible with existing HTML Web pages, was the first result of this renewed interest. At this early stage, while the draft was already publicly available, and input was already being solicited from all sources, the specification was only under Opera Software's copyright.

The idea that HTML's evolution should be reopened was tested at a W3C workshop in 2004, where some of the principles that underlie the HTML5 work (described below), as well as the aforementioned early draft proposal covering just forms-related features, were presented to the W3C jointly by Mozilla and Opera. The proposal was rejected on the grounds that the

proposal conflicted with the previously chosen direction for the Web's evolution; the W3C staff and membership voted to continue developing XML-based replacements instead.

Shortly thereafter, Apple, Mozilla, and Opera jointly announced their intent to continue working on the effort under the umbrella of a new venue called the WHATWG. A public mailing list was created, and the draft was moved to the WHATWG site. The copyright was subsequently amended to be jointly owned by all three vendors, and to allow reuse of the specification.

The WHATWG was based on several core principles, in particular that technologies need to be backwards compatible, that specifications and implementations need to match even if this means changing the specification rather than the implementations, and that specifications need to be detailed enough that implementations can achieve complete interoperability without reverse-engineering each other.

The latter requirement in particular required that the scope of the HTML5 specification include what had previously been specified in three separate documents: HTML4, XHTML1, and DOM2 HTML. It also meant including significantly more detail than had previously been considered the norm.

In 2006, the W3C indicated an interest to participate in the development of HTML5 after all, and in 2007 formed a working group chartered to work with the WHATWG on the development of the HTML5 specification. Apple, Mozilla, and Opera allowed the W3C to publish the specification under the W3C copyright, while keeping a version with the less restrictive license on the WHATWG site.

Since then, both groups have been working together.

The [HTML specification](#) published by the WHATWG is not identical to this specification. The main differences are that the WHATWG version includes features not included in this W3C version: some features have been omitted as they are considered part of future revisions of HTML, not HTML5; and other features are omitted because at the W3C they are published as separate specifications. There are also some minor differences. For an exact list of differences, please see the WHATWG specification.

A separate document has been published by the W3C HTML working group to document the differences between this specification and the language described in the HTML4 specification. [\[HTMLDIFF\]](#)

1.5 Design notes

This section is non-normative.

It must be admitted that many aspects of HTML appear at first glance to be nonsensical and inconsistent.

HTML, its supporting DOM APIs, as well as many of its supporting technologies, have been developed over a period of several decades by a wide array of people with different priorities who, in many cases, did not know of each other's existence.

Features have thus arisen from many sources, and have not always been designed in especially consistent ways. Furthermore, because of the unique characteristics of the Web,

implementation bugs have often become de-facto, and now de-jure, standards, as content is often unintentionally written in ways that rely on them before they can be fixed.

Despite all this, efforts have been made to adhere to certain design goals. These are described in the next few subsections.

1.5.1 Serializability of script execution

This section is non-normative.

To avoid exposing Web authors to the complexities of multithreading, the HTML and DOM APIs are designed such that no script can ever detect the simultaneous execution of other scripts. Even with workers, the intent is that the behavior of implementations can be thought of as completely serializing the execution of all scripts in all [browsing contexts](#).

Note: The [navigator.yieldForStorageUpdates\(\)](#) method, in this model, is equivalent to allowing other scripts to run while the calling script is blocked.

1.5.2 Compliance with other specifications

This section is non-normative.

This specification interacts with and relies on a wide variety of other specifications. In certain circumstances, unfortunately, conflicting needs have led to this specification violating the requirements of these other specifications. Whenever this has occurred, the transgressions have each been noted as a **"willful violation"**, and the reason for the violation has been noted.

1.6 HTML vs XHTML

This section is non-normative.

This specification defines an abstract language for describing documents and applications, and some APIs for interacting with in-memory representations of resources that use this language.

The in-memory representation is known as "DOM HTML", or "the DOM" for short. This specification defines version 5 of DOM HTML, known as "DOM5 HTML".

There are various concrete syntaxes that can be used to transmit resources that use this abstract language, two of which are defined in this specification.

The first such concrete syntax is the HTML syntax. This is the format suggested for most authors. It is compatible with most legacy Web browsers. If a document is transmitted with an [HTML MIME type](#), such as [text/html](#), then it will be processed as an HTML document by Web browsers. This specification defines version 5 of the HTML syntax, known as "HTML5".

The second concrete syntax is the XHTML syntax, which is an application of XML. When a document is transmitted with an [XML MIME type](#), such as [application/xhtml+xml](#), then it is treated as an XML document by Web browsers, to be parsed by an XML processor. Authors

are reminded that the processing for XML and HTML differs; in particular, even minor syntax errors will prevent a document labeled as XML from being rendered fully, whereas they would be ignored in the HTML syntax. This specification defines version 5 of the XHTML syntax, known as "XHTML5".

The DOM, the HTML syntax, and XML cannot all represent the same content. For example, namespaces cannot be represented using the HTML syntax, but they are supported in the DOM and in XML. Similarly, documents that use the [noscript](#) feature can be represented using the HTML syntax, but cannot be represented with the DOM or in XML. Comments that contain the string "-->" can only be represented in the DOM, not in the HTML and XML syntaxes.

1.7 Structure of this specification

This section is non-normative.

This specification is divided into the following major sections:

[Common infrastructure](#)

The conformance classes, algorithms, definitions, and the common underpinnings of the rest of the specification.

[Semantics, structure, and APIs of HTML documents](#)

Documents are built from elements. These elements form a tree using the DOM. This section defines the features of this DOM, as well as introducing the features common to all elements, and the concepts used in defining elements.

[The elements of HTML](#)

Each element has a predefined meaning, which is explained in this section. Rules for authors on how to use the element, along with user agent requirements for how to handle each element, are also given.

[Loading Web pages](#)

HTML documents do not exist in a vacuum — this section defines many of the features that affect environments that deal with multiple pages.

[Web application APIs](#)

This section introduces basic features for scripting of applications in HTML.

[User interaction](#)

HTML documents can provide a number of mechanisms for users to interact with and modify content, which are described in this section.

[The HTML syntax](#)

[The XHTML syntax](#)

All of these features would be for naught if they couldn't be represented in a serialized form and sent to other people, and so these sections define the syntaxes of HTML, along with rules for how to parse content using those syntaxes.

There are also some appendices, defining [rendering rules](#) for Web browsers and listing [obsolete features](#) and [IANA considerations](#).

1.7.1 How to read this specification

This specification should be read like all other specifications. First, it should be read cover-to-cover, multiple times. Then, it should be read backwards at least once. Then it should be read by picking random sections from the contents list and following all the cross-references.

1.7.2 Typographic conventions

This is a definition, requirement, or explanation.

Note: *This is a note.*

|| This is an example.

This is an open issue.

△Warning! *This is a warning.*

```
interface Example {
  // this is an IDL definition
};
```

This box is non-normative. Implementation requirements are given below this box.
~~**variable = object . method ([optionalArgument])**~~

This is a note to authors describing the usage of an interface.

```
/* this is a CSS fragment */
```

The defining instance of a term is marked up like **this**. Uses of that term are marked up like this or like this.

The defining instance of an element, attribute, or API is marked up like `this`. References to that element, attribute, or API are marked up like this.

Other code fragments are marked up like `this`.

Variables are marked up like *this*.

This is an implementation requirement.

1.8 A quick introduction to HTML

This section is non-normative.

A basic HTML document looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample page</title>
  </head>
  <body>
    <h1>Sample page</h1>
    <p>This is a <a href="demo.html">simple</a> sample.</p>
    <!-- this is a comment -->
```


Tags have to be nested such that elements are all completely within each other, without overlapping:

<p>This is very wrong!</p>

<p>This is correct.</p>

Elements can have attributes, which control how the elements work. In the example below, there is a [hyperlink](#), formed using the `a` element and its `href` attribute:

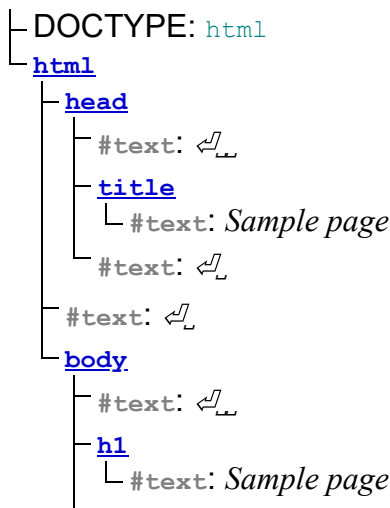
```
<a href="demo.html">simple</a>
```

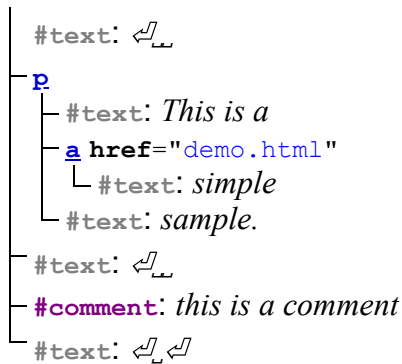
```
<!-- empty attributes -->
<input name=address disabled>
<input name=address disabled="">

<!-- attributes with a value -->
<input name=address maxlength=200>
<input name=address maxlength='200'>
<input name=address maxlength="200">
```

DOM trees contain several kinds of nodes, in particular a DOCTYPE node, elements, text nodes, and comment nodes.

The [markup snippet at the top of this section](#) would be turned into the following DOM tree:





The [root element](#) of this tree is the [html](#) element, which is the element always found at the root of HTML documents. It contains two elements, [head](#) and [body](#), as well as a text node between them.

There are many more text nodes in the DOM tree than one would initially expect, because the source contains a number of spaces (represented here by " ") and line breaks ("␣") that all end up as text nodes in the DOM. However, for historical reasons not all of the spaces and line breaks in the original markup appear in the DOM. In particular, all the whitespace before [head](#) start tag ends up being dropped silently, and all the whitespace after the [body](#) end tag ends up placed at the end of the [body](#).

The [head](#) element contains a [title](#) element, which itself contains a text node with the text "Sample page". Similarly, the [body](#) element contains an [h1](#) element, a [p](#) element, and a comment.

This DOM tree can be manipulated from scripts in the page. Scripts (typically in JavaScript) are small programs that can be embedded using the [script](#) element or using [event handler content attributes](#). For example, here is a form with a script that sets the value of the form's [output](#) element to say "Hello World":

```

<form name="main">
  Result: <output name="result"></output>
  <script>
    document.forms.main.elements.result.value = 'Hello World';
  </script>
</form>

```

Each element in the DOM tree is represented by an object, and these objects have APIs so that they can be manipulated. For instance, a link (e.g. the [a](#) element in the tree above) can have its "[href](#)" attribute changed in several ways:

```

var a = document.links[0]; // obtain the first link in the document
a.href = 'sample.html'; // change the destination URL of the link
a.protocol = 'https'; // change just the scheme part of the URL
a.setAttribute('href', 'http://example.com/'); // change the content attribute directly

```

Since DOM trees are used as the way to represent HTML documents when they are processed and presented by implementations (especially interactive implementations like Web browsers), this specification is mostly phrased in terms of DOM trees, instead of the markup described above.

HTML documents represent a media-independent description of interactive content. HTML documents might be rendered to a screen, or through a speech synthesizer, or on a braille display. To influence exactly how such rendering takes place, authors can use a styling language such as CSS.

In the following example, the page has been made yellow-on-blue using CSS.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample styled page</title>
    <style>
      body { background: navy; color: yellow; }
    </style>
  </head>
  <body>
    <h1>Sample styled page</h1>
    <p>This page is just a demo.</p>
  </body>
</html>
```

For more details on how to use HTML, authors are encouraged to consult tutorials and guides. Some of the examples included in this specification might also be of use, but the novice author is cautioned that this specification, by necessity, defines the language with a level of detail that might be difficult to understand at first.

1.9 Conformance requirements for authors

This section is non-normative.

Unlike previous versions of the HTML specification, this specification defines in some detail the required processing for invalid documents as well as valid documents.

However, even though the processing of invalid content is in most cases well-defined, conformance requirements for documents are still important: in practice, interoperability (the situation in which all implementations process particular content in a reliable and identical or equivalent way) is not the only goal of document conformance requirements. This section details some of the more common reasons for still distinguishing between a conforming document and one with errors.

1.9.1 Presentational markup

This section is non-normative.

The majority of presentational features from previous versions of HTML are no longer allowed. Presentational markup in general has been found to have a number of problems:

The use of presentational elements leads to poorer accessibility

While it is possible to use presentational markup in a way that provides users of assistive technologies (ATs) with an acceptable experience (e.g. using ARIA), doing so is significantly more difficult than doing so when using semantically-appropriate markup. Furthermore, even using such techniques doesn't help make pages accessible for non-AT non-graphical users, such as users of text-mode browsers.

Using media-independent markup, on the other hand, provides an easy way for documents to be authored in such a way that they work for more users (e.g. text browsers).

Higher cost of maintenance

It is significantly easier to maintain a site written in such a way that the markup is style-independent. For example, changing the color of a site that uses `<font color="">` throughout requires changes across the entire site, whereas a similar change to a site based on CSS can be done by changing a single file.

Higher document sizes

Presentational markup tends to be much more redundant, and thus results in larger document sizes.

For those reasons, presentational markup has been removed from HTML in this version. This change should not come as a surprise; HTML4 deprecated presentational markup many years ago and provided a mode (HTML4 Transitional) to help authors move away from presentational markup; later, XHTML 1.1 went further and obsoleted those features altogether.

The only remaining presentational markup features in HTML are the [style](#) attribute and the [style](#) element. Use of the [style](#) attribute is somewhat discouraged in production environments, but it can be useful for rapid prototyping (where its rules can be directly moved into a separate style sheet later) and for providing specific styles in unusual cases where a separate style sheet would be inconvenient. Similarly, the [style](#) element can be useful in syndication or for page-specific styles, but in general an external style sheet is likely to be more convenient when the styles apply to multiple pages.

It is also worth noting that some elements that were previously presentational have been redefined in this specification to be media-independent: [b](#), [i](#), [hr](#), [s](#), and [small](#).

1.9.2 Syntax errors

This section is non-normative.

The syntax of HTML is constrained to avoid a wide variety of problems.

Unintuitive error-handling behavior

Certain invalid syntax constructs, when parsed, result in DOM trees that are highly unintuitive.

For example, the following markup fragment results in a DOM with an [hr](#) element that is an *earlier* sibling of the corresponding [table](#) element:

```
<table><hr>...
```

Errors with optional error recovery

To allow user agents to be used in controlled environments without having to implement the more bizarre and convoluted error handling rules, user agents are permitted to fail whenever encountering a [parse error](#).

Errors where the error-handling behavior is not compatible with streaming user agents

Some error-handling behavior, such as the behavior for the `<table><hr>...` example mentioned above, are incompatible with streaming user agents (user agents that

process HTML files in one pass, without storing state). To avoid interoperability problems with such user agents, any syntax resulting in such behavior is considered invalid.

Errors that can result in infoset coercion

When a user agent based on XML is connected to an HTML parser, it is possible that certain invariants that XML enforces, such as comments never containing two consecutive hyphens, will be violated by an HTML file. Handling this can require that the parser coerce the HTML DOM into an XML-compatible infoset. Most syntax constructs that require such handling are considered invalid.

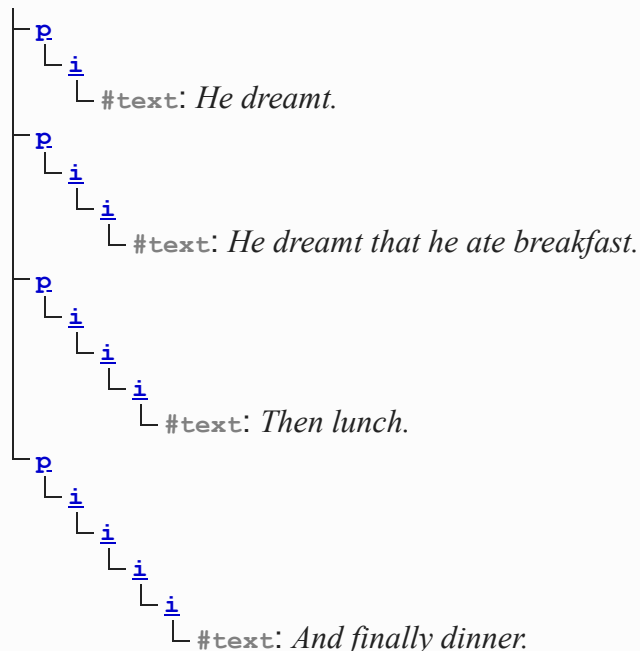
Errors that result in disproportionately poor performance

Certain syntax constructs can result in disproportionately poor performance. To discourage the use of such constructs, they are typically made non-conforming.

For example, the following markup results in poor performance, since all the unclosed `i` elements have to be reconstructed in each paragraph, resulting in progressively more elements in each paragraph:

```
<p><i>He dreamt.  
<p><i>He dreamt that he ate breakfast.  
<p><i>Then lunch.  
<p><i>And finally dinner.
```

The resulting DOM for this fragment would be:



Errors involving fragile syntax constructs

There are syntax constructs that, for historical reasons, are relatively fragile. To help reduce the number of users who accidentally run into such problems, they are made non-conforming.

For example, the parsing of certain named character references in attributes happens even with the closing semicolon being omitted. It is safe to include an ampersand followed by letters that do not form a named character reference, but if the letters are changed to a string that *does* form a named character reference, they will be interpreted as that character instead.

In this fragment, the attribute's value is `"?bill&ted"`:

```
<a href="?bill&ted">Bill and Ted</a>
```

In the following fragment, however, the attribute's value is actually "?art@", *not* the intended "?art©":

```
<a href="?art&copy">Art and Copy</a>
```

To avoid this problem, all named character references are required to end with a semicolon, and uses of named character references without a semicolon are flagged as errors.

Thus, the correct way to express the above cases is as follows:

```
<a href="?bill&ted">Bill and Ted</a> <!-- &ted is ok, since it's not
a named character reference -->
```

```
<a href="?art&amp;copy">Art and Copy</a> <!-- the & has to be
escaped, since &copy is a named character reference -->
```

Errors involving known interoperability problems in legacy user agents

Certain syntax constructs are known to cause especially subtle or serious problems in legacy user agents, and are therefore marked as non-conforming to help authors avoid them.

For example, this is why the U+0060 GRAVE ACCENT character (`) is not allowed in unquoted attributes. In certain legacy user agents, it is sometimes treated as a quote character.

Another example of this is the DOCTYPE, which is required to trigger [no-quirks mode](#), because the behavior of legacy user agents in [quirks mode](#) is often largely undocumented.

Errors that risk exposing authors to security attacks

Certain restrictions exist purely to avoid known security problems.

For example, the restriction on using UTF-7 exists purely to avoid authors falling prey to a known cross-site-scripting attack using UTF-7.

Cases where the author's intent is unclear

Markup where the author's intent is very unclear is often made non-conforming. Correcting these errors early makes later maintenance easier.

For example, it is unclear whether the author intended the following to be an [h1](#) heading or an [h2](#) heading:

```
<h1>Contact details</h2>
```

Cases that are likely to be typos

When a user makes a simple typo, it is helpful if the error can be caught early, as this can save the author a lot of debugging time. This specification therefore usually considers it an error to use element names, attribute names, and so forth, that do not match the names defined in this specification.

For example, if the author typed `<capton>` instead of `<caption>`, this would be flagged as an error and the author could correct the typo immediately.

Errors that could interfere with new syntax in the future

In order to allow the language syntax to be extended in the future, certain otherwise harmless features are disallowed.

For example, "attributes" in end tags are ignored currently, but they are invalid, in case a future change to the language makes use of that syntax feature without conflicting with already-deployed (and valid!) content.

Some authors find it helpful to be in the practice of always quoting all attributes and always including all optional tags, preferring the consistency derived from such custom over the minor benefits of terseness afforded by making use of the flexibility of the HTML syntax. To aid such authors, conformance checkers can provide modes of operation wherein such conventions are enforced.

1.9.3 Restrictions on content models and on attribute values

This section is non-normative.

Beyond the syntax of the language, this specification also places restrictions on how elements and attributes can be specified. These restrictions are present for similar reasons:

Errors involving content with dubious semantics

To avoid misuse of elements with defined meanings, content models are defined that restrict how elements can be nested when such nestings would be of dubious value.

For example, this specification disallows nesting a [section](#) element inside a [kbd](#) element, since it is highly unlikely for an author to indicate that an entire section should be keyed in.

Errors that involve a conflict in expressed semantics

Similarly, to draw the author's attention to mistakes in the use of elements, clear contradictions in the semantics expressed are also considered conformance errors.

In the fragments below, for example, the semantics are nonsensical: a row cannot simultaneously be a cell, nor can a radio button be a progress bar.

```
<tr role="cell">
  <input type=radio role=progressbar>
```

Another example is the restrictions on the content models of the [ul](#) element, which only allows [li](#) element children. Lists by definition consist just of zero or more list items, so if a [ul](#) element contains something other than an [li](#) element, it's not clear what was meant.

Cases where the default styles are likely to lead to confusion

Certain elements have default styles or behaviors that make certain combinations likely to lead to confusion. Where these have equivalent alternatives without this problem, the confusing combinations are disallowed.

For example, [div](#) elements are rendered as block boxes, and [span](#) elements as inline boxes. Putting a block box in an inline box is unnecessarily confusing; since either nesting just [div](#) elements, or nesting just [span](#) elements, or nesting [span](#) elements inside [div](#) elements all serve the same purpose as nesting a [div](#) element in a [span](#) element, but only the latter involves a block box in an inline box, the latter combination is disallowed.

Another example would be the way [interactive content](#) cannot be nested. For example, a [button](#) element cannot contain a [textarea](#) element. This is because

the default behavior of such nesting interactive elements would be highly confusing to users. Instead of nesting these elements, they can be placed side by side.

Errors that indicate a likely misunderstanding of the specification

Sometimes, something is disallowed because allowing it would likely cause author confusion.

For example, setting the [disabled](#) attribute to the value "false" is disallowed, because despite the appearance of meaning that the element is enabled, it in fact means that the element is *disabled* (what matters for implementations is the presence of the attribute, not its value).

Errors involving limits that have been imposed merely to simplify the language

Some conformance errors simplify the language that authors need to learn.

For example, the [area](#) element's [shape](#) attribute, despite accepting both [circ](#) and [circle](#) values in practice as synonyms, disallows the use of the [circ](#) value, so as to simplify tutorials and other learning aids. There would be no benefit to allowing both, but it would cause extra confusion when teaching the language.

Errors that involve peculiarities of the parser

Certain elements are parsed in somewhat eccentric ways (typically for historical reasons), and their content model restrictions are intended to avoid exposing the author to these issues.

For example, a [form](#) element isn't allowed inside [phrasing content](#), because when parsed as HTML, a [form](#) element's start tag will imply a [p](#) element's end tag. Thus, the following markup results in two [paragraphs](#), not one:

```
<p>Welcome. <form><label>Name:</label> <input></form>
```

It is parsed exactly like the following:

```
<p>Welcome. </p><form><label>Name:</label> <input></form>
```

Errors that would likely result in scripts failing in hard-to-debug ways

Some errors are intended to help prevent script problems that would be hard to debug.

This is why, for instance, it is non-conforming to have two [id](#) attributes with the same value. Duplicate IDs lead to the wrong element being selected, with sometimes disastrous effects whose cause is hard to determine.

Errors that waste authoring time

Some constructs are disallowed because historically they have been the cause of a lot of wasted authoring time, and by encouraging authors to avoid making them, authors can save time in future efforts.

For example, a [script](#) element's [src](#) attribute causes the element's contents to be ignored. However, this isn't obvious, especially if the element's contents appear to be executable script — which can lead to authors spending a lot of time trying to debug the inline script without realizing that it is not executing. To reduce this problem, this specification makes it non-conforming to have executable script in a [script](#) element when the [src](#) attribute is present. This means that authors who are validating their documents are less likely to waste time with this kind of mistake.

Errors that involve areas that affect authors migrating to and from XHTML

Some authors like to write files that can be interpreted as both XML and HTML with similar results. Though this practice is discouraged in general due to the myriad of subtle complications involved (especially when involving scripting, styling, or any kind of automated serialization), this specification has a few restrictions intended to at least somewhat mitigate the difficulties. This makes it easier for authors to use this as a transitional step when migrating between HTML and XHTML.

For example, there are somewhat complicated rules surrounding the [lang](#) and [xml:lang](#) attributes intended to keep the two synchronized.

Another example would be the restrictions on the values of `xmlns` attributes in the HTML serialization, which are intended to ensure that elements in conforming documents end up in the same namespaces whether processed as HTML or XML.

Errors that involve areas reserved for future expansion

As with the restrictions on the syntax intended to allow for new syntax in future revisions of the language, some restrictions on the content models of elements and values of attributes are intended to allow for future expansion of the HTML vocabulary.

For example, limiting the values of the [target](#) attribute that start with an U+005F LOW LINE character (`_`) to only specific predefined values allows new predefined values to be introduced at a future time without conflicting with author-defined values.

Errors that indicate a mis-use of other specifications

Certain restrictions are intended to support the restrictions made by other specifications.

For example, requiring that attributes that take media queries use only *valid* media queries reinforces the importance of following the conformance rules of that specification.

1.10 Recommended reading

This section is non-normative.

The following documents might be of interest to readers of this specification.

Character Model for the World Wide Web 1.0: Fundamentals [\[CHARMOD\]](#)

This Architectural Specification provides authors of specifications, software developers, and content developers with a common reference for interoperable text manipulation on the World Wide Web, building on the Universal Character Set, defined jointly by the Unicode Standard and ISO/IEC 10646. Topics addressed include use of the terms 'character', 'encoding' and 'string', a reference processing model, choice and identification of character encodings, character escaping, and string indexing.

Unicode Security Considerations [\[UTR36\]](#)

Because Unicode contains such a large number of characters and incorporates the varied writing systems of the world, incorrect usage can expose programs or systems to possible security attacks. This is especially important as more and

more products are internationalized. This document describes some of the security considerations that programmers, system analysts, standards developers, and users should take into account, and provides specific recommendations to reduce the risk of problems.

Web Content Accessibility Guidelines (WCAG) 2.0 [\[WCAG\]](#)

Web Content Accessibility Guidelines (WCAG) 2.0 covers a wide range of recommendations for making Web content more accessible. Following these guidelines will make content accessible to a wider range of people with disabilities, including blindness and low vision, deafness and hearing loss, learning disabilities, cognitive limitations, limited movement, speech disabilities, photosensitivity and combinations of these. Following these guidelines will also often make your Web content more usable to users in general.

Authoring Tool Accessibility Guidelines (ATAG) 2.0 [\[ATAG\]](#)

This specification provides guidelines for designing Web content authoring tools that are more accessible for people with disabilities. An authoring tool that conforms to these guidelines will promote accessibility by providing an accessible user interface to authors with disabilities as well as by enabling, supporting, and promoting the production of accessible Web content by all authors.

User Agent Accessibility Guidelines (UAAG) 2.0 [\[UAAG\]](#)

This document provides guidelines for designing user agents that lower barriers to Web accessibility for people with disabilities. User agents include browsers and other types of software that retrieve and render Web content. A user agent that conforms to these guidelines will promote accessibility through its own user interface and through other internal facilities, including its ability to communicate with other technologies (especially assistive technologies). Furthermore, all users, not just users with disabilities, should find conforming user agents to be more usable.

Polyglot Markup: HTML-Compatible XHTML Documents [\[POLYGLOT\]](#)

A document that uses polyglot markup is a document that is a stream of bytes that parses into identical document trees (with the exception of the `xmlns` attribute on the root element) when processed as HTML and when processed as XML. Polyglot markup that meets a well defined set of constraints is interpreted as compatible, regardless of whether they are processed as HTML or as XHTML, per the HTML5 specification. Polyglot markup uses a specific DOCTYPE, namespace declarations, and a specific case — normally lower case but occasionally camel case — for element and attribute names. Polyglot markup uses lower case for certain attribute values. Further constraints include those on empty elements, named entity references, and the use of scripts and style.