## NAME

HTML::Element − Class for objects that represent HTML elements

## VERSION

This document describes version 5.07 of HTML::Element, released August 31, 2017 as part of HTML-Tree.

## SYNOPSIS

```
use HTML::Element;
$a = HTML::Element->new('a', href => 'http://www.perl.com/');
$a->push_content("The Perl Homepage");

$tag = $a->tag;
print "$tag starts out as:",  $a->starttag, "\n";
print "$tag ends as:",  $a->endtag, "\n";
print "$tag\'s href attribute is: ", $a->attr('href'), "\n";

$links_r = $a->extract_links();
print "Hey, I found ", scalar(@$links_r), " links.\n";

print "And that, as HTML, is: ", $a->as_HTML, "\n";
$a = $a->delete;
```

## DESCRIPTION

(This class is part of the HTML::Tree dist.)

Objects of the HTML::Element class can be used to represent elements of HTML document trees. These objects have attributes, notably attributes that designates each element's parent and content. The content is an array of text segments and other HTML::Element objects. A tree with HTML::Element objects as nodes can represent the syntax tree for a HTML document.

## HOW WE REPRESENT TREES

Consider this HTML document:

```
<html lang='en-US'>
  <head>
    <title>Stuff</title>
    <meta name='author' content='Jojo'>
  </head>
  <body>
   <h1>I like potatoes!</h1>
  </body>
</html>
```

Building a syntax tree out of it makes a tree-structure in memory that could be diagrammed as:

```
                   html (lang='en-US')
                    / \
                  /     \
                /         \
              head        body
             /\               \
           /    \               \
         /        \               \
       title     meta              h1
        |       (name='author',     |
     "Stuff"    content='Jojo')    "I like potatoes"
```

This is the traditional way to diagram a tree, with the "root" at the top, and it's this kind of diagram that people have in mind when they say, for example, that "the meta element is under the head element instead of under the body element''. (The same is also said with ``inside'' instead of ``under'' — the use of ``inside'' makes more sense when you're looking at the HTML source.)

Another way to represent the above tree is with indenting:

```
html (attributes: lang='en-US')
  head
    title
      "Stuff"
    meta (attributes: name='author' content='Jojo')
  body
    h1
      "I like potatoes"
```

Incidentally, diagramming with indenting works much better for very large trees, and is easier for a program to generate. The `$tree->dump` method uses indentation just that way.

However you diagram the tree, it's stored the same in memory — it's a network of objects, each of which has attributes like so:

```
element #1:  _tag: 'html'
             _parent: none
             _content: [element #2, element #5]
             lang: 'en-US'

element #2:  _tag: 'head'
             _parent: element #1
             _content: [element #3, element #4]

element #3:  _tag: 'title'
             _parent: element #2
             _content: [text segment "Stuff"]

element #4   _tag: 'meta'
             _parent: element #2
             _content: none
             name: author
             content: Jojo

element #5   _tag: 'body'
             _parent: element #1
             _content: [element #6]

element #6   _tag: 'h1'
             _parent: element #5
             _content: [text segment "I like potatoes"]
```

The ``treeness'' of the tree-structure that these elements comprise is not an aspect of any particular object, but is emergent from the relatedness attributes (_parent and _content) of these element-objects and from how you use them to get from element to element.

While you could access the content of a tree by writing code that says "access the 'src' attribute of the root's *first* child's *seventh* child's *third* child", you're more likely to have to scan the contents of a tree, looking for whatever nodes, or kinds of nodes, you want to do something with. The most straightforward way to look over a tree is to "traverse" it; an HTML::Element method (`$h->traverse`) is provided for this purpose; and several other HTML::Element methods are based on it.

(For everything you ever wanted to know about trees, and then some, see Niklaus Wirth's *Algorithms + Data Structures = Programs* or Donald Knuth's *The Art of Computer Programming, Volume 1*.)

### Weak References

TL;DR summary: `use HTML::TreeBuilder 5 −weak;` and forget about the `delete` method (except for pruning a node from a tree).

Because HTML::Element stores a reference to the parent element, Perl’s reference-count garbage collection doesn’t work properly with HTML::Element trees. Starting with version 5.00, HTML::Element uses weak references (if available) to prevent that problem. Weak references were introduced in Perl 5.6.0, but you also need a version of Scalar::Util that provides the `weaken` function.

Weak references are enabled by default. If you want to be certain they’re in use, you can say `use HTML::Element 5 −weak;`. You must include the version number; previous versions of HTML::Element ignored the import list entirely.

To disable weak references, you can say `use HTML::Element −noweak;`. This is a global setting. **This feature is deprecated** and is provided only as a quick fix for broken code. If your code does not work properly with weak references, you should fix it immediately, as weak references may become mandatory in a future version. Generally, all you need to do is keep a reference to the root of the tree until you’re done working with it.

Because HTML::TreeBuilder is a subclass of HTML::Element, you can also import `−weak` or `−noweak` from HTML::TreeBuilder: e.g. `use HTML::TreeBuilder: 5 −weak;`.

## BASIC METHODS

### new

```
  $h = HTML::Element−>new('tag', 'attrname' => 'value', ... );
```

This constructor method returns a new HTML::Element object. The tag name is a required argument; it will be forced to lowercase. Optionally, you can specify other initial attributes at object creation time.

### attr

```
  $value = $h−>attr('attr');
  $old_value = $h−>attr('attr', $new_value);
```

Returns (optionally sets) the value of the given attribute of `$h`. The attribute name (but not the value, if provided) is forced to lowercase. If trying to read the value of an attribute not present for this element, the return value is undef. If setting a new value, the old value of that attribute is returned.

If methods are provided for accessing an attribute (like `$h−>tag` for ‘‘_tag’’, `$h−>content_list`, etc. below), use those instead of calling attr `$h−>attr`, whether for reading or setting.

Note that setting an attribute to `undef` (as opposed to "", the empty string) actually deletes the attribute.

### tag

```
  $tagname = $h−>tag();
  $h−>tag('tagname');
```

Returns (optionally sets) the tag name (also known as the generic identifier) for the element `$h`. In setting, the tag name is always converted to lower case.

There are four kinds of ‘‘pseudo-elements’’ that show up as HTML::Element objects:

Comment pseudo-elements

These are element objects with a `$h−>tag` value of ‘‘˜comment’’, and the content of the comment is stored in the ‘‘text’’ attribute (`$h−>attr("text")`). For example, parsing this code with HTML::TreeBuilder...

```
  <!−− I like Pie.
     Pie is good
  −−>
```

produces an HTML::Element object with these attributes:

```
"_tag",
"~comment",
"text",
" I like Pie.\n     Pie is good\n  "
```

Declaration pseudo-elements

Declarations (rarely encountered) are represented as HTML::Element objects with a tag name of "~declaration", and content in the "text" attribute. For example, this:

```
<!DOCTYPE foo>
```

produces an element whose attributes include:

```
"_tag", "~declaration", "text", "DOCTYPE foo"
```

Processing instruction pseudo-elements

PIs (rarely encountered) are represented as HTML::Element objects with a tag name of "~pi", and content in the "text" attribute. For example, this:

```
<?stuff foo?>
```

produces an element whose attributes include:

```
"_tag", "~pi", "text", "stuff foo?"
```

(assuming a recent version of HTML::Parser)

~literal pseudo-elements

These objects are not currently produced by HTML::TreeBuilder, but can be used to represent a "super-literal" — i.e., a literal you want to be immune from escaping. (Yes, I just made that term up.)

That is, this is useful if you want to insert code into a tree that you plan to dump out with `as_HTML`, where you want, for some reason, to suppress `as_HTML`'s normal behavior of amp-quoting text segments.

For example, this:

```
my $literal = HTML::Element->new('~literal',
  'text' => 'x < 4 & y > 7'
);
my $span = HTML::Element->new('span');
$span->push_content($literal);
print $span->as_HTML;
```

prints this:

```
<span>x < 4 & y > 7</span>
```

Whereas this:

```
my $span = HTML::Element->new('span');
$span->push_content('x < 4 & y > 7');
  # normal text segment
print $span->as_HTML;
```

prints this:

```
<span>x &lt; 4 &amp; y &gt; 7</span>
```

Unless you're inserting lots of pre-cooked code into existing trees, and dumping them out again, it's not likely that you'll find `~literal` pseudo-elements useful.

**parent**

```
$parent = $h->parent();
$h->parent($new_parent);
```

Returns (optionally sets) the parent (aka "container") for this element. The parent should either be undef, or should be another element.

You **should not** use this to directly set the parent of an element. Instead use any of the other methods under "Structure-Modifying Methods", below.

Note that `not($h->parent)` is a simple test for whether `$h` is the root of its subtree.

**content_list**

```
@content = $h->content_list();
$num_children = $h->content_list();
```

Returns a list of the child nodes of this element — i.e., what nodes (elements or text segments) are inside/under this element. (Note that this may be an empty list.)

In a scalar context, this returns the count of the items, as you may expect.

**content**

```
$content_array_ref = $h->content(); # may return undef
```

This somewhat deprecated method returns the content of this element; but unlike content_list, this returns either undef (which you should understand to mean no content), or a *reference to the array* of content items, each of which is either a text segment (a string, i.e., a defined non-reference scalar value), or an HTML::Element object. Note that even if an arrayref is returned, it may be a reference to an empty array.

While older code should feel free to continue to use `$h->content`, new code should use `$h->content_list` in almost all conceivable cases. It is my experience that in most cases this leads to simpler code anyway, since it means one can say:

```
@children = $h->content_list;
```

instead of the inelegant:

```
@children = @{$h->content || []};
```

If you do use `$h->content` (or `$h->content_array_ref`), you should not use the reference returned by it (assuming it returned a reference, and not undef) to directly set or change the content of an element or text segment! Instead use content_refs_list or any of the other methods under "Structure-Modifying Methods", below.

**content_array_ref**

```
$content_array_ref = $h->content_array_ref(); # never undef
```

This is like `content` (with all its caveats and deprecations) except that it is guaranteed to return an array reference. That is, if the given node has no `_content` attribute, the `content` method would return that undef, but `content_array_ref` would set the given node's `_content` value to `[]` (a reference to a new, empty array), and return that.

**content_refs_list**

```
@content_refs = $h->content_refs_list;
```

This returns a list of scalar references to each element of `$h`'s content list. This is useful in case you want to in-place edit any large text segments without having to get a copy of the current value of that segment value, modify that copy, then use the `splice_content` to replace the old with the new. Instead, here you can in-place edit:

```
foreach my $item_r ($h->content_refs_list) {
    next if ref $$item_r;
    $$item_r =~ s/honour/honor/g;
}
```

You *could* currently achieve the same affect with:

```
    foreach my $item (@{ $h->content_array_ref }) {
        # deprecated!
        next if ref $item;
        $item =~ s/honour/honor/g;
    }
```

...except that using the return value of `$h->content` or `$h->content_array_ref` to do that is deprecated, and just might stop working in the future.

**implicit**

```
  $is_implicit = $h->implicit();
  $h->implicit($make_implicit);
```

Returns (optionally sets) the "_implicit" attribute. This attribute is a flag that's used for indicating that the element was not originally present in the source, but was added to the parse tree (by HTML::TreeBuilder, for example) in order to conform to the rules of HTML structure.

**pos**

```
  $pos = $h->pos();
  $h->pos($element);
```

Returns (and optionally sets) the "_pos" (for "current *pos*ition") pointer of `$h`. This attribute is a pointer used during some parsing operations, whose value is whatever HTML::Element element at or under `$h` is currently "open", where `$h->insert_element(NEW)` will actually insert a new element.

(This has nothing to do with the Perl function called `pos`, for controlling where regular expression matching starts.)

If you set `$h->pos($element)`, be sure that `$element` is either `$h`, or an element under `$h`.

If you've been modifying the tree under `$h` and are no longer sure `$h->pos` is valid, you can enforce validity with:

```
    $h->pos(undef) unless $h->pos->is_inside($h);
```

**all_attr**

```
  %attr = $h->all_attr();
```

Returns all this element's attributes and values, as key-value pairs. This will include any "internal" attributes (i.e., ones not present in the original element, and which will not be represented if/when you call `$h->as_HTML`). Internal attributes are distinguished by the fact that the first character of their key (not value! key!) is an underscore ("_").

Example output of `$h->all_attr()` : `'_parent',` *[object_value]* `, '_tag', 'em', 'lang', 'en-US', '_content',` *[array-ref value]*.

**all_attr_names**

```
  @names = $h->all_attr_names();
  $num_attrs = $h->all_attr_names();
```

Like `all_attr`, but only returns the names of the attributes. In scalar context, returns the number of attributes.

Example output of `$h->all_attr_names()` : `'_parent', '_tag', 'lang', '_content', `.

**all_external_attr**

```
  %attr = $h->all_external_attr();
```

Like `all_attr`, except that internal attributes are not present.

**all_external_attr_names**

```
  @names = $h->all_external_attr_names();
  $num_attrs = $h->all_external_attr_names();
```

Like `all_attr_names`, except that internal attributes' names are not present (or counted).

**id**

```
$id = $h->id();
$h->id($string);
```

Returns (optionally sets to `$string`) the “id” attribute. `$h->id(undef)` deletes the “id” attribute.

`$h->id(...)` is basically equivalent to `$h->attr('id', ...)`, except that when setting the attribute, this method returns the new value, not the old value.

**idf**

```
$id = $h->idf();
$h->idf($string);
```

Just like the `id` method, except that if you call `$h->idf()` and no “id” attribute is defined for this element, then it’s set to a likely-to-be-unique value, and returned. (The “f” is for “force”.)

## STRUCTURE-MODIFYING METHODS

These methods are provided for modifying the content of trees by adding or changing nodes as parents or children of other nodes.

**push_content**

```
$h->push_content($element_or_text, ...);
```

Adds the specified items to the *end* of the content list of the element `$h`. The items of content to be added should each be either a text segment (a string), an HTML::Element object, or an arrayref. Arrayrefs are fed thru `$h->new_from_lol(that_arrayref)` to convert them into elements, before being added to the content list of `$h`. This means you can say things concise things like:

```
$body->push_content(
  ['br'],
  ['ul',
    map ['li', $_], qw(Peaches Apples Pears Mangos)
  ]
);
```

See the “new_from_lol” method’s documentation, far below, for more explanation.

Returns `$h` (the element itself).

The push_content method will try to consolidate adjacent text segments while adding to the content list. That’s to say, if `$h`’s `content_list` is

```
('foo bar ', $some_node, 'baz!')
```

and you call

```
 $h->push_content('quack?');
```

then the resulting content list will be this:

```
('foo bar ', $some_node, 'baz!quack?')
```

and not this:

```
('foo bar ', $some_node, 'baz!', 'quack?')
```

If that latter is what you want, you’ll have to override the feature of consolidating text by using splice_content, as in:

```
$h->splice_content(scalar($h->content_list),0,'quack?');
```

Similarly, if you wanted to add ’Skronk’ to the beginning of the content list, calling this:

```
 $h->unshift_content('Skronk');
```

then the resulting content list will be this:

```
('Skronkfoo bar ', $some_node, 'baz!')
```

and not this:

```
('Skronk', 'foo bar ', $some_node, 'baz!')
```

What you'd to do get the latter is:

```
$h->splice_content(0,0,'Skronk');
```

**unshift_content**

```
$h->unshift_content($element_or_text, ...)
```

Just like `push_content`, but adds to the *beginning* of the `$h` element's content list.

The items of content to be added should each be either a text segment (a string), an HTML::Element object, or an arrayref (which is fed thru `new_from_lol`).

The unshift_content method will try to consolidate adjacent text segments while adding to the content list. See above for a discussion of this.

Returns `$h` (the element itself).

**splice_content**

```
@removed = $h->splice_content($offset, $length,
                              $element_or_text, ...);
```

Detaches the elements from `$h`'s list of content-nodes, starting at `$offset` and continuing for `$length` items, replacing them with the elements of the following list, if any. Returns the elements (if any) removed from the content-list. If `$offset` is negative, then it starts that far from the end of the array, just like Perl's normal `splice` function. If `$length` and the following list is omitted, removes everything from `$offset` onward.

The items of content to be added (if any) should each be either a text segment (a string), an arrayref (which is fed thru "new_from_lol"), or an HTML::Element object that's not already a child of `$h`.

**detach**

```
$old_parent = $h->detach();
```

This unlinks `$h` from its parent, by setting its 'parent' attribute to undef, and by removing it from the content list of its parent (if it had one). The return value is the parent that was detached from (or undef, if `$h` had no parent to start with). Note that neither `$h` nor its parent are explicitly destroyed.

**detach_content**

```
@old_content = $h->detach_content();
```

This unlinks all of `$h`'s children from `$h`, and returns them. Note that these are not explicitly destroyed; for that, you can just use `$h->delete_content`.

**replace_with**

```
$h->replace_with( $element_or_text, ... )
```

This replaces `$h` in its parent's content list with the nodes specified. The element `$h` (which by then may have no parent) is returned. This causes a fatal error if `$h` has no parent. The list of nodes to insert may contain `$h`, but at most once. Aside from that possible exception, the nodes to insert should not already be children of `$h`'s parent.

Also, note that this method does not destroy `$h` if weak references are turned off — use `$h->replace_with(...)->delete` if you need that.

**preinsert**

```
$h->preinsert($element_or_text...);
```

Inserts the given nodes right BEFORE `$h` in `$h`'s parent's content list. This causes a fatal error if `$h` has no parent. None of the given nodes should be `$h` or other children of `$h`. Returns`$h`.

**postinsert**

```
$h->postinsert($element_or_text...)
```

Inserts the given nodes right AFTER `$h` in `$h`'s parent's content list. This causes a fatal error if `$h` has no parent. None of the given nodes should be `$h` or other children of `$h`. Returns`$h`.

**replace_with_content**

```
$h->replace_with_content();
```

This replaces `$h` in its parent's content list with its own content. The element `$h` (which by then has no parent or content of its own) is returned. This causes a fatal error if `$h` has no parent. Also, note that this does not destroy `$h` if weak references are turned off — use `$h->replace_with_content->delete` if you need that.

**delete_content**

```
$h->delete_content();
$h->destroy_content(); # alias
```

Clears the content of `$h`, calling `$h->delete` for each content element. Compare with `$h->detach_content`.

Returns `$h`.

`destroy_content` is an alias for this method.

**delete**

```
$h->delete();
$h->destroy(); # alias
```

Detaches this element from its parent (if it has one) and explicitly destroys the element and all its descendants. The return value is the empty list (or `undef` in scalar context).

Before version 5.00 of HTML::Element, you had to call `delete` when you were finished with the tree, or your program would leak memory. This is no longer necessary if weak references are enabled, see "Weak References".

**destroy**

An alias for "delete".

**destroy_content**

An alias for "delete_content".

**clone**

```
$copy = $h->clone();
```

Returns a copy of the element (whose children are clones (recursively) of the original's children, if any).

The returned element is parentless. Any '_pos' attributes present in the source element/tree will be absent in the copy. For that and other reasons, the clone of an HTML::TreeBuilder object that's in mid-parse (i.e, the head of a tree that HTML::TreeBuilder is elaborating) cannot (currently) be used to continue the parse.

You are free to clone HTML::TreeBuilder trees, just as long as: 1) they're done being parsed, or 2) you don't expect to resume parsing into the clone. (You can continue parsing into the original; it is never affected.)

**clone_list**

```
@copies = HTML::Element->clone_list(...nodes...);
```

Returns a list consisting of a copy of each node given. Text segments are simply copied; elements are cloned by calling `$it->clone` on each of them.

Note that this must be called as a class method, not as an instance method. `clone_list` will croak if called as an instance method. You can also call it like so:

```
    ref($h)->clone_list(...nodes...)
```

**normalize_content**

```
$h->normalize_content
```

Normalizes the content of `$h` — i.e., concatenates any adjacent text nodes. (Any undefined text segments are turned into empty-strings.) Note that this does not recurse into `$h`'s descendants.

**delete_ignorable_whitespace**

```
$h->delete_ignorable_whitespace()
```

This traverses under `$h` and deletes any text segments that are ignorable whitespace. You should not use this if `$h` is under a `<pre>` element.

**insert_element**

```
$h->insert_element($element, $implicit);
```

Inserts (via push_content) a new element under the element at `$h->pos()`. Then updates`$h->pos()` to point to the inserted element, unless `$element` is a prototypically empty element like `<br>`, `<hr>`, `<img>`, etc. The new `$h->pos()` is returned. This method is useful only if your particular tree task involves setting `$h->pos()`.

## DUMPING METHODS

**dump**

```
$h->dump()
$h->dump(*FH)  ; # or *FH{IO} or $fh_obj
```

Prints the element and all its children to STDOUT (or to a specified filehandle), in a format useful only for debugging. The structure of the document is shown by indentation (no end tags).

**as_HTML**

```
$s = $h->as_HTML();
$s = $h->as_HTML($entities);
$s = $h->as_HTML($entities, $indent_char);
$s = $h->as_HTML($entities, $indent_char, \%optional_end_tags);
```

Returns a string representing in HTML the element and its descendants. The optional argument `$entities` specifies a string of the entities to encode. For compatibility with previous versions, specify `'<>&'` here. If omitted or undef, *all* unsafe characters are encoded as HTML entities. See HTML::Entities for details. If passed an empty string, no entities are encoded.

If `$indent_char` is specified and defined, the HTML to be output is intented, using the string you specify (which you probably should set to "\t", or some number of spaces, if you specify it).

If `\%optional_end_tags` is specified and defined, it should be a reference to a hash that holds a true value for every tag name whose end tag is optional. Defaults to `\%HTML::Element::optionalEndTag`, which is an alias to `%HTML::Tagset::optionalEndTag`, which, at time of writing, contains true values for `p, li, dt, dd`. A useful value to pass is an empty hashref, `{}`, which means that no end-tags are optional for this dump. Otherwise, possibly consider copying `%HTML::Tagset::optionalEndTag` to a hash of your own, adding or deleting values as you like, and passing a reference to that hash.

**as_text**

```
$s = $h->as_text();
$s = $h->as_text(skip_dels => 1);
```

Returns a string consisting of only the text parts of the element's descendants. Any whitespace inside the element is included unchanged, but whitespace not in the tree is never added. But remember that whitespace may be ignored or compacted by HTML::TreeBuilder during parsing (depending on the value of the `ignore_ignorable_whitespace` and `no_space_compacting` attributes). Also, since whitespace is never added during parsing,

```
HTML::TreeBuilder->new_from_content("<p>a</p><p>b</p>")
                  ->as_text;
```

returns `"ab"`, not `"a b"` or `"a\nb"`.

Text under `<script>` or `<style>` elements is never included in what's returned. If `skip_dels` is true, then text content under `<del>` nodes is not included in what's returned.

**as_trimmed_text**

```
$s = $h->as_trimmed_text(...);
$s = $h->as_trimmed_text(extra_chars => '\xA0'); # remove &nbsp;
$s = $h->as_text_trimmed(...); # alias
```

This is just like `as_text(...)` except that leading and trailing whitespace is deleted, and any internal whitespace is collapsed.

This will not remove non-breaking spaces, Unicode spaces, or any other non-ASCII whitespace unless you supply the extra characters as a string argument (e.g. `$h->as_trimmed_text(extra_chars => '\xA0')`). `extra_chars` may be any string that can appear inside a character class, including ranges like `a-z`, POSIX character classes like `[:alpha:]`, and character class escapes like `\p{Zs}`.

**as_XML**

```
$s = $h->as_XML()
```

Returns a string representing in XML the element and its descendants.

The XML is not indented.

**as_Lisp_form**

```
$s = $h->as_Lisp_form();
```

Returns a string representing the element and its descendants as a Lisp form. Unsafe characters are encoded as octal escapes.

The Lisp form is indented, and contains external (“href”, etc.) as well as internal attributes (“_tag”, “_content”, “_implicit”, etc.), except for “_parent”, which is omitted.

Current example output for a given element:

```
("_tag" "img" "border" "0" "src" "pie.png" "usemap" "#main.map")
```

**format**

```
$s = $h->format; # use HTML::FormatText
$s = $h->format($formatter);
```

Formats text output. Defaults to HTML::FormatText.

Takes a second argument that is a reference to a formatter.

**starttag**

```
$start = $h->starttag();
$start = $h->starttag($entities);
```

Returns a string representing the complete start tag for the element. I.e., leading “<”, tag name, attributes, and trailing “>”. All values are surrounded with double-quotes, and appropriate characters are encoded. If `$entities` is omitted or undef, *all* unsafe characters are encoded as HTML entities. See HTML::Entities for details. If you specify some value for `$entities`, remember to include the double-quote character in it. (Previous versions of this module would basically behave as if `'&">'` were specified for `$entities`.) If `$entities` is an empty string, no entity is escaped.

**starttag_XML**

```
$start = $h->starttag_XML();
```

Returns a string representing the complete start tag for the element.

**endtag**

```
$end = $h->endtag();
```

Returns a string representing the complete end tag for this element. I.e., “</”, tag name, and “>”.

**endtag_XML**

```
$end = $h->endtag_XML();
```

Returns a string representing the complete end tag for this element. I.e., “</”, tag name, and “>”.

## SECONDARY STRUCTURAL METHODS

These methods all involve some structural aspect of the tree; either they report some aspect of the tree's structure, or they involve traversal down the tree, or walking up the tree.

**is_inside**

```
$inside = $h−>is_inside('tag', $element, ...);
```

Returns true if the `$h` element is, or is contained anywhere inside an element that is any of the ones listed, or whose tag name is any of the tag names listed. You can use any mix of elements and tag names.

**is_empty**

```
$empty = $h−>is_empty();
```

Returns true if `$h` has no content, i.e., has no elements or text segments under it. In other words, this returns true if `$h` is a leaf node, AKA a terminal node. Do not confuse this sense of "empty" with another sense that it can have in SGML/HTML/XML terminology, which means that the element in question is of the type (like HTML's `<hr>`, `<br>`, `<img>`, etc.) that *can't* have any content.

That is, a particular `<p>` element may happen to have no content, so `$that_p_element`−>is_empty will be true — even though the prototypical `<p>` element isn't "empty" (not in the way that the prototypical `<hr>` element is).

If you think this might make for potentially confusing code, consider simply using the clearer exact equivalent: `not($h−>content_list)`.

**pindex**

```
$index = $h−>pindex();
```

Return the index of the element in its parent's contents array, such that `$h` would equal

```
$h−>parent−>content−>[$h−>pindex]
# or
($h−>parent−>content_list)[$h−>pindex]
```

assuming `$h` isn't root. If the element `$h` is root, then `$h−>pindex` returns `undef`.

**left**

```
$element = $h−>left();
@elements = $h−>left();
```

In scalar context: returns the node that's the immediate left sibling of `$h`. If`$h` is the leftmost (or only) child of its parent (or has no parent), then this returns undef.

In list context: returns all the nodes that're the left siblings of `$h` (starting with the leftmost). If `$h` is the leftmost (or only) child of its parent (or has no parent), then this returns an empty list.

(See also `$h−>preinsert(LIST)`.)

**right**

```
$element = $h−>right();
@elements = $h−>right();
```

In scalar context: returns the node that's the immediate right sibling of `$h`. If`$h` is the rightmost (or only) child of its parent (or has no parent), then this returns `undef`.

In list context: returns all the nodes that're the right siblings of `$h`, starting with the leftmost. If `$h` is the rightmost (or only) child of its parent (or has no parent), then this returns an empty list.

(See also `$h−>postinsert(LIST)`.)

**address**

```
$address = $h−>address();
$element_or_text = $h−>address($address);
```

The first form (with no parameter) returns a string representing the location of `$h` in the tree it is a member of. The address consists of numbers joined by a '.', starting with '0', and followed by the pindexes of the nodes in the tree that are ancestors of `$h`, starting from the top.

So if the way to get to a node starting at the root is to go to child 2 of the root, then child 10 of that, and then child 0 of that, and then you're there — then that node's address is "0.2.10.0".

As a bit of a special case, the address of the root is simply "0".

I forsee this being used mainly for debugging, but you may find your own uses for it.

```
  $element_or_text = $h->address($address);
```

This form returns the node (whether element or text-segment) at the given address in the tree that `$h` is a part of. (That is, the address is resolved starting from `$h->root`.)

If there is no node at the given address, this returns `undef`.

You can specify "relative addressing" (i.e., that indexing is supposed to start from `$h` and not from `$h->root`) by having the address start with a period — e.g., `$h->address(".3.2")` will look at child 3 of `$h`, and child 2 of that.

**depth**

```
  $depth = $h->depth();
```

Returns a number expressing `$h`'s depth within its tree, i.e., how many steps away it is from the root. If `$h` has no parent (i.e., is root), its depth is 0.

**root**

```
  $root = $h->root();
```

Returns the element that's the top of `$h`'s tree. If `$h` is root, this just returns `$h`. (If you want to test whether `$h` *is* the root, instead of asking what its root is, just test `not($h->parent)`.)

**lineage**

```
  @lineage = $h->lineage();
```

Returns the list of `$h`'s ancestors, starting with its parent, and then that parent's parent, and so on, up to the root. If`$h` is root, this returns an empty list.

If you simply want a count of the number of elements in `$h`'s lineage, use `$h->depth`.

**lineage_tag_names**

```
  @names = $h->lineage_tag_names();
```

Returns the list of the tag names of `$h`'s ancestors, starting with its parent, and that parent's parent, and so on, up to the root. If `$h` is root, this returns an empty list. Example output: `('em', 'td', 'tr', 'table', 'body', 'html')`

Equivalent to:

```
  map { $_->tag } $h->lineage;
```

**descendants**

```
  @descendants = $h->descendants();
```

In list context, returns the list of all `$h`'s descendant elements, listed in pre-order (i.e., an element appears before its content-elements). Text segments DO NOT appear in the list. In scalar context, returns a count of all such elements.

**descendents**

This is just an alias to the `descendants` method, for people who can't spell.

**find_by_tag_name**

```
  @elements = $h->find_by_tag_name('tag', ...);
  $first_match = $h->find_by_tag_name('tag', ...);
```

In list context, returns a list of elements at or under `$h` that have any of the specified tag names. In scalar context, returns the first (in pre-order traversal of the tree) such element found, or undef if none.

**find**

This is just an alias to `find_by_tag_name`. (There was once going to be a whole find_* family of methods, but then `look_down` filled that niche, so there turned out not to be much reason for the verboseness of the name ‘‘find_by_tag_name’’.)

**find_by_attribute**

```
  @elements = $h->find_by_attribute('attribute', 'value');
  $first_match = $h->find_by_attribute('attribute', 'value');
```

In a list context, returns a list of elements at or under `$h` that have the specified attribute, and have the given value for that attribute. In a scalar context, returns the first (in pre-order traversal of the tree) such element found, or undef if none.

This method is **deprecated** in favor of the more expressive `look_down` method, which new code should use instead.

**look_down**

```
  @elements = $h->look_down( ...criteria... );
  $first_match = $h->look_down( ...criteria... );
```

This starts at `$h` and looks thru its element descendants (in pre-order), looking for elements matching the criteria you specify. In list context, returns all elements that match all the given criteria; in scalar context, returns the first such element (or undef, if nothing matched).

There are three kinds of criteria you can specify:

(attr_name, attr_value)

This means you’re looking for an element with that value for that attribute. Example:`"alt", "pix!"`. Consider that you can search on internal attribute values too: `"_tag", "p"`.

(attr_name, qr/.../)

This means you’re looking for an element whose value for that attribute matches the specified Regexp object.

a coderef

This means you’re looking for elements where coderef−>(each_element) returns true. Example:

```
  my @wide_pix_images = $h->look_down(
    _tag => "img",
    alt  => "pix!",
    sub { $_[0]->attr('width') > 350 }
  );
```

Note that `(attr_name, attr_value)` and `(attr_name, qr/.../)` criteria are almost always faster than coderef criteria, so should presumably be put before them in your list of criteria. That is, in the example above, the sub ref is called only for elements that have already passed the criteria of having a ‘‘_tag’’ attribute with value ‘‘img’’, and an ‘‘alt’’ attribute with value ‘‘pix!’’. If the coderef were first, it would be called on every element, and *then* what elements pass that criterion (i.e., elements for which the coderef returned true) would be checked for their ‘‘_tag’’ and ‘‘alt’’ attributes.

Note that comparison of string attribute-values against the string value in `(attr_name, attr_value)` is case-INsensitive! A criterion of`('align', 'right')` *will* match an element whose ‘ ‘align’’ value is ‘‘RIGHT’’, or ‘‘right’’ or ‘‘rIGhT’’, etc.

Note also that `look_down` considers "" (empty-string) and undef to be different things, in attribute values. So this:

```
  $h->look_down("alt", "")
```

will find elements *with* an ‘‘alt’’ attribute, but where the value for the ‘‘alt’’ attribute is "". But this:

```
  $h->look_down("alt", undef)
```

is the same as:

```
$h->look_down(sub { !defined($_[0]->attr('alt')) } )
```

That is, it finds elements that do not have an "alt" attribute at all (or that do have an "alt" attribute, but with a value of undef — which is not normally possible).

Note that when you give several criteria, this is taken to mean you're looking for elements that match *all* your criterion, not just *any* of them. In other words, there is an implicit "and", not an "or". So if you wanted to express that you wanted to find elements with a "name" attribute with the value "foo" *or* with an "id" attribute with the value "baz", you'd have to do it like:

```
@them = $h->look_down(
  sub {
    # the lcs are to fold case
    lc($_[0]->attr('name')) eq 'foo'
    or lc($_[0]->attr('id')) eq 'baz'
  }
);
```

Coderef criteria are more expressive than `(attr_name, attr_value)` and `(attr_name, qr/.../)` criteria, and all `(attr_name, attr_value)` and `(attr_name, qr/.../)` criteria could be expressed in terms of coderefs. However, `(attr_name, attr_value)` and `(attr_name, qr/.../)` criteria are a convenient shorthand. (In fact, `look_down` itself is basically "shorthand" too, since anything you can do with `look_down` you could do by traversing the tree, either with the `traverse` method or with a routine of your own. However, `look_down` often makes for very concise and clear code.)

**look_up**

```
@elements = $h->look_up( ...criteria... );
$first_match = $h->look_up( ...criteria... );
```

This is identical to `$h->look_down`, except that whereas `$h->look_down` basically scans over the list:

```
($h, $h->descendants)
```

`$h->look_up` instead scans over the list

```
($h, $h->lineage)
```

So, for example, this returns all ancestors of `$h` (possibly including `$h` itself) that are `<td>` elements with an "align" attribute with a value of "right" (or "RIGHT", etc.):

```
$h->look_up("_tag", "td", "align", "right");
```

**traverse**

```
$h->traverse(...options...)
```

Lengthy discussion of HTML::Element's unnecessary and confusing `traverse` method has been moved to a separate file: HTML::Element::traverse

**attr_get_i**

```
@values = $h->attr_get_i('attribute');
$first_value = $h->attr_get_i('attribute');
```

In list context, returns a list consisting of the values of the given attribute for `$h` and for all its ancestors starting from `$h` and working its way up. Nodes with no such attribute are skipped. ("attr_get_i" stands for "attribute get, with inheritance".) In scalar context, returns the first such value, or undef if none.

Consider a document consisting of:

```
<html lang='i-klingon'>
  <head><title>Pati Pata</title></head>
  <body>
    <h1 lang='la'>Stuff</h1>
    <p lang='es-MX' align='center'>
      Foo bar baz <cite>Quux</cite>.
    </p>
    <p>Hooboy.</p>
  </body>
</html>
```

If `$h` is the `<cite>` element, `$h->attr_get_i("lang")` in list context will return the list `('es-MX', 'i-klingon')`. In scalar context, it will return the value `'es-MX'`.

If you call with multiple attribute names...

```
@values = $h->attr_get_i('a1', 'a2', 'a3');
$first_value = $h->attr_get_i('a1', 'a2', 'a3');
```

...in list context, this will return a list consisting of the values of these attributes which exist in `$h` and its ancestors. In scalar context, this returns the first value (i.e., the value of the first existing attribute from the first element that has any of the attributes listed). So, in the above example,

```
$h->attr_get_i('lang', 'align');
```

will return:

```
 ('es-MX', 'center', 'i-klingon') # in list context
or
 'es-MX' # in scalar context.
```

But note that this:

```
$h->attr_get_i('align', 'lang');
```

will return:

```
 ('center', 'es-MX', 'i-klingon') # in list context
or
 'center' # in scalar context.
```

**tagname_map**

```
$hash_ref = $h->tagname_map();
```

Scans across `$h` and all its descendants, and makes a hash (a reference to which is returned) where each entry consists of a key that's a tag name, and a value that's a reference to a list to all elements that have that tag name. I.e., this method returns:

```
{
  # Across $h and all descendants...
  'a'   => [ ...list of all <a>   elements... ],
  'em'  => [ ...list of all <em>  elements... ],
  'img' => [ ...list of all <img> elements... ],
}
```

(There are entries in the hash for only those tagnames that occur at/under `$h` — so if there's no `<img>` elements, there'll be no "img" entry in the returned hashref.)

Example usage:

```
          my $map_r = $h->tagname_map();
          my @heading_tags = sort grep m/^h\d$/s, keys %$map_r;
          if(@heading_tags) {
            print "Heading levels used: @heading_tags\n";
          } else {
            print "No headings.\n"
          }
```

**extract_links**

```
        $links_array_ref = $h->extract_links();
        $links_array_ref = $h->extract_links(@wantedTypes);
```

Returns links found by traversing the element and all of its children and looking for attributes (like "href" in an `<a>` element, or "src" in an `<img>` element) whose values represent links. The return value is a *reference* to an array. Each element of the array is reference to an array with*four* items: the link-v alue, the element that has the attribute with that link-value, and the name of that attribute, and the tagname of that element. (Example:`['http://www.suck.com/',` *`$elem_obj`* `, 'href', 'a']`.) Y ou may or may not end up using the element itself — for some purposes, you may use only the link value.

You might specify that you want to extract links from just some kinds of elements (instead of the default, which is to extract links from *all* the kinds of elements known to have attributes whose values represent links). For instance, if you want to extract links from only `<a>` and `<img>` elements, you could code it like this:

```
        for (@{  $e->extract_links('a', 'img')  }) {
            my($link, $element, $attr, $tag) = @$_;
            print
              "Hey, there's a $tag that links to ",
              $link, ", in its $attr attribute, at ",
              $element->address(), ".\n";
        }
```

**simplify_pres**

```
        $h->simplify_pres();
```

In text bits under PRE elements that are at/under `$h`, this routine nativizes all newlines, and expands all tabs.

That is, if you read a file with lines delimited by `\cm\cj`'s, the text under PRE areas will have `\cm\cj`'s instead of `\n`'s. Calling `$h->simplify_pres` on such a tree will turn `\cm\cj`'s into `\n`'s.

Tabs are expanded to however many spaces it takes to get to the next 8th column — the usual way of expanding them.

**same_as**

```
        $equal = $h->same_as($i)
```

Returns true if `$h` and `$i` are both elements representing the same tree of elements, each with the same tag name, with the same explicit attributes (i.e., not counting attributes whose names start with "_"), and with the same content (textual, comments, etc.).

Sameness of descendant elements is tested, recursively, with `$child1->same_as($child_2)`, and sameness of text segments is tested with `$segment1 eq $segment2`.

**new_from_lol**

```
        $h = HTML::Element->new_from_lol($array_ref);
        @elements = HTML::Element->new_from_lol($array_ref, ...);
```

Recursively constructs a tree of nodes, based on the (non-cyclic) data structure represented by each `$array_ref`, where that is a reference to an array of arrays (of arrays (of arrays (etc.))).

In each arrayref in that structure, different kinds of values are treated as follows:

- Arrayrefs

  Arrayrefs are considered to designate a sub-tree representing children for the node constructed from the current arrayref.

- Hashrefs

  Hashrefs are considered to contain attribute-value pairs to add to the element to be constructed from the current arrayref

- Text segments

  Text segments at the start of any arrayref will be considered to specify the name of the element to be constructed from the current arrayref; all other text segments will be considered to specify text segments as children for the current arrayref.

- Elements

  Existing element objects are either inserted into the treelet constructed, or clones of them are. That is, when the lol-tree is being traversed and elements constructed based what's in it, if an existing element object is found, if it has no parent, then it is added directly to the treelet constructed; but if it has a parent, then `$that_node->clone` is added to the treelet at the appropriate place.

An example will hopefully make this more obvious:

```
my $h = HTML::Element->new_from_lol(
  ['html',
    ['head',
      [ 'title', 'I like stuff!' ],
    ],
    ['body',
      {'lang', 'en-JP', _implicit => 1},
      'stuff',
      ['p', 'um, p < 4!', {'class' => 'par123'}],
      ['div', {foo => 'bar'}, '123'],
    ]
  ]
);
$h->dump;
```

Will print this:

```
<html> @0
  <head> @0.0
    <title> @0.0.0
      "I like stuff!"
  <body lang="en-JP"> @0.1 (IMPLICIT)
    "stuff"
    <p class="par123"> @0.1.1
      "um, p < 4!"
    <div foo="bar"> @0.1.2
      "123"
```

And printing `$h`->as_HTML will give something like:

```
<html><head><title>I like stuff!</title></head>
<body lang="en-JP">stuff<p class="par123">um, p &lt; 4!
<div foo="bar">123</div></body></html>
```

You can even do fancy things with `map`:

```
  $body->push_content(
    # push_content implicitly calls new_from_lol on arrayrefs...
    ['br'],
    ['blockquote',
      ['h2', 'Pictures!'],
      map ['p', $_],
      $body2->look_down("_tag", "img"),
        # images, to be copied from that other tree.
    ],
    # and more stuff:
    ['ul',
      map ['li', ['a', {'href'=>"$_.png"}, $_ ] ],
      qw(Peaches Apples Pears Mangos)
    ],
  );
```

In scalar context, you must supply exactly one arrayref. In list context, you can pass a list of arrayrefs, and new_from_lol will return a list of elements, one for each arrayref.

```
  @elements = HTML::Element->new_from_lol(
    ['hr'],
    ['p', 'And there, on the door, was a hook!'],
  );
   # constructs two elements.
```

**objectify_text**

```
  $h->objectify_text();
```

This turns any text nodes under `$h` from mere text segments (strings) into real objects, pseudo-elements with a tag-name of "~text", and the actual text content in an attribute called "text". (For a discussion of pseudo-elements, see the "tag" method, far above.) This method is provided because, for some purposes, it is convenient or necessary to be able, for a given text node, to ask what element is its parent; and clearly this is not possible if a node is just a text string.

Note that these "~text" objects are not recognized as text nodes by methods like "as_text". Presumably you will want to call `$h->objectify_text`, perform whatever task that you needed that for, and then call `$h->deobjectify_text` before calling anything like `$h->as_text`.

**deobjectify_text**

```
  $h->deobjectify_text();
```

This undoes the effect of `$h->objectify_text`. That is, it takes any "~text" pseudo-elements in the tree at/under `$h`, and deletes each one, replacing each with the content of its "text" attribute.

Note that if `$h` itself is a "~text" pseudo-element, it will be destroyed — a condition you may need to treat specially in your calling code (since it means you can't very well do anything with `$h` after that). So that you can detect that condition, if `$h` is itself a "~text" pseudo-element, then this method returns the value of the "text" attribute, which should be a defined value; in all other cases, it returns undef.

(This method assumes that no "~text" pseudo-element has any children.)

**number_lists**

```
  $h->number_lists();
```

For every UL, OL, DIR, and MENU element at/under `$h`, this sets a "_bullet" attribute for every child LI element. For LI children of an OL, the "_bullet" attribute's value will be something like "4.", "d.", "D.", "IV.", or "iv.", depending on the OL element's "type" attribute. LI children of a UL, DIR, or MENU get their "_bullet" attribute set to "*". There should be no other LIs (i.e., except as children of OL, UL, DIR, or MENU elements), and if there are, they are unaffected.

**has_insane_linkage**

```
$h->has_insane_linkage
```

This method is for testing whether this element or the elements under it have linkage attributes (_parent and _content) whose values are deeply aberrant: if there are undefs in a content list; if an element appears in the content lists of more than one element; if the _parent attribute of an element doesn't match its actual parent; or if an element appears as its own descendant (i.e., if there is a cyclicity in the tree).

This returns empty list (or false, in scalar context) if the subtree's linkage methods are sane; otherwise it returns two items (or true, in scalar context): the element where the error occurred, and a string describing the error.

This method is provided is mainly for debugging and troubleshooting — it should be *quite impossible* for any document constructed via HTML::TreeBuilder to parse into a non-sane tree (since it's not the content of the tree per se that's in question, but whether the tree in memory was properly constructed); and it *should* be impossible for you to produce an insane tree just thru reasonable use of normal documented structure-modifying methods. But if you're constructing your own trees, and your program is going into infinite loops as during calls to **traverse()** or any of the secondary structural methods, as part of debugging, consider calling `has_insane_linkage` on the tree.

**element_class**

```
$classname = $h->element_class();
```

This method returns the class which will be used for new elements. It defaults to HTML::Element, but can be overridden by subclassing or esoteric means best left to those will will read the source and then not complain when those esoteric means change. (Just subclass.)

## CLASS METHODS

**Use_Weak_Refs**

```
$enabled = HTML::Element->Use_Weak_Refs;
HTML::Element->Use_Weak_Refs( $enabled );
```

This method allows you to check whether weak reference support is enabled, and to enable or disable it. For details, see ''Weak References''. `$enabled` is true if weak references are enabled.

You should not switch this in the middle of your program, and you probably shouldn't use it at all. Existing trees are not affected by this method (until you start modifying nodes in them).

Throws an exception if you attempt to enable weak references and your Perl or Scalar::Util does not support them.

Disabling weak reference support is deprecated.

## SUBROUTINES

**Version**

This subroutine is deprecated. Please use the standard VERSION method (e.g. `HTML::Element->VERSION`) instead.

**ABORT OK PRUNE PRUNE_SOFTLY PRUNE_UP**

Constants for signalling back to the traverser

## BUGS

* If you want to free the memory associated with a tree built of HTML::Element nodes, and you have disabled weak references, then you will have to delete it explicitly using the ''delete'' method. See ''Weak References''.

* There's almost nothing to stop you from making a ''tree'' with cyclicities (loops) in it, which could, for example, make the traverse method go into an infinite loop. So don't make cyclicities! (If all you're doing is parsing HTML files, and looking at the resulting trees, this will never be a problem for you.)

* There's no way to represent comments or processing directives in a tree with HTML::Elements. Not yet, at least.

* There's (currently) nothing to stop you from using an undefined value as a text segment. If you're

running under `perl -w`, however, this may make HTML::Element's code produce a slew of warnings.

## NOTES ON SUBCLASSING

You are welcome to derive subclasses from HTML::Element, but you should be aware that the code in HTML::Element makes certain assumptions about elements (and I'm using ''element'' to mean ONLY an object of class HTML::Element, or of a subclass of HTML::Element):

* The value of an element's _parent attribute must either be undef or otherwise false, or must be an element.

* The value of an element's _content attribute must either be undef or otherwise false, or a reference to an (unblessed) array. The array may be empty; but if it has items, they must ALL be either mere strings (text segments), or elements.

* The value of an element's _tag attribute should, at least, be a string of printable characters.

Moreover, bear these rules in mind:

* Do not break encapsulation on objects. That is, access their contents only thru `$obj->`attr or more specific methods.

* You should think twice before completely overriding any of the methods that HTML::Element provides. (Overriding with a method that calls the superclass method is not so bad, though.)

## SEE ALSO

HTML::Tree; HTML::TreeBuilder; HTML::AsSubs; HTML::Tagset; and, for the morbidly curious, HTML::Element::traverse.

## ACKNOWLEDGEMENTS

Thanks to Mark-Jason Dominus for a POD suggestion.

## AUTHOR

Current maintainers:

- Christopher J. Madsen `<perl AT cjmweb.net>`
- Jeff Fearn `<jfearn AT cpan.org>`

Original HTML-Tree author:

- Gisle Aas

Former maintainers:

- Sean M. Burke
- Andy Lester
- Pete Krawczyk `<petek AT cpan.org>`

You can follow or contribute to HTML-Tree's development at <https://github.com/kentfredric/HTML−Tree>.

## COPYRIGHT AND LICENSE