

MDN's new design is in Beta! A sneak peek: <https://blog.mozilla.org/opendesign/mdns-new-design-beta/>

## Learn web development

# HTML: A good basis for accessibility

[← Previous](#)[↑ Overview: Accessibility](#)[Next →](#)

A great deal of web content can be made accessible just by making sure the correct HTML elements are used for the correct purpose at all times. This article looks in detail at how HTML can be used to ensure maximum accessibility.

<b>Prerequisites:</b>	Basic computer literacy, a basic understanding of HTML (see <a href="#">Introduction to HTML</a> ), and understanding of <a href="#">what accessibility is</a> .
<b>Objective:</b>	To gain familiarity with what features of HTML have accessibility benefits, and how to use them appropriately in your web documents.

## HTML and accessibility

As you learn more about HTML — read more resources, look at more examples, etc. — you'll keep seeing a common theme: the importance of using semantic HTML (sometimes called POSH, or Plain Old Semantic HTML). This means using the correct HTML elements for their correct purpose as much as possible.

You might wonder why this is so important. After all, you can use a combination of CSS and JavaScript to make just about any HTML element behave in whatever way you want. For example, a control button to play a video on your site could be marked up like this:

1 | `<div>Play video</div>`

But as you'll see in greater detail later on, it makes much sense to use the correct element for the job:


## 1 | `<button>Play video</button>`

Not only do HTML `<button>`s have some suitable styling applied by default (which you will probably want to override), they also have built-in keyboard accessibility — they can be tabbed between, and activated using Return/Enter.

Semantic HTML doesn't take longer to write than non-semantic (bad) markup if you do it consistently from the start of your project, and it also has other benefits beyond accessibility:

1. **Easier to develop with** — as mentioned above, you get some functionality for free, plus it is arguably easier to understand.
2. **Better on mobile** — semantic HTML is arguably lighter in file size than non-semantic spaghetti code, and easier to make responsive.
3. **Good for SEO** — search engines give more importance to keywords inside headings, links, etc., than keywords included in non-semantic `<div>`s, etc., so your documents will be more findable by customers.

Let's get on and look at accessible HTML in more detail.

 **Note:** It is a good idea to have a screenreader set up on your local computer, so you can do some testing of the examples shown below. See our [Screenreaders guide](#) for more details.

## Good semantics

We've already talked about the importance of good semantics, and why we should use the right HTML element for the right job. This cannot be ignored, as it is one of the main places that accessibility is badly broken if not handled properly.

Out there on the web, the truth is that people do some very strange things with HTML markup. Some abuses of HTML are due to legacy practices that have not been completely forgotten, and some are just plain ignorance. Whatever the case, you should replace such bad code wherever you see it, whenever you can.

Sometimes you are not always in the position to get rid of bad markup — your pages might be generated by some kind of server-side framework that you don't have full control over, or you might have third party content on your page (such as ad banners) that you don't have control over.

The goal isn't "all or nothing", however — every improvement you are able to make will help the cause of accessibility.

## Text content

One of the best accessibility aids a screenreader user can have is a good content structure of headings, paragraphs, lists, etc. A good semantic example might look something like the following:

```

1  <h1>My heading</h1>
2
3  <p>This is the first section of my document.</p>
4
5  <p>I'll add another paragraph here too.</p>
6
7  <ol>
8    <li>Here is</li>
9    <li>a list for</li>
10   <li>you to read</li>
11 </ol>
12
13 <h2>My subheading</h2>
14
15 <p>This is the first subsection of my document. I'd love people to be abl
16
17 <h2>My 2nd subheading</h2>
18
19 <p>This is the second subsection of my content. I think is more interes😊

```

We've prepared a version with longer text for you to try out with a screenreader (see [good-semantics.html](#)). If you try navigating through this, you'll see that this is pretty easy to navigate:

1. The screenreader reads each header out as you progress through the content, notifying you what is a heading, what is a paragraph, etc.
2. It stops after each element, letting you go at whatever pace is comfortable for you.
3. You can jump to next/previous heading in many screenreaders.
4. You can also bring up a list of all headings in many screenreaders, allowing you to use them like a handy table of contents to find specific content.

People sometimes write headings, paragraphs, etc. using presentational HTML and line breaks, something like the following:

```

1  <font size="7">My heading</font>
2  <br><br>
3  This is the first section of my document.
4  <br><br>
5  I'll add another paragraph here too.
6  <br><br>
7  1. Here is
8  <br><br>
9  2. a list for
10 <br><br>
11 3. you to read
12 <br><br>
13 <font size="5">My subheading</font>

```

```

14 <br><br>
15 This is the first subsection of my document. I'd love people to be able t
16 <br><br>
17 <font size="5">My 2nd subheading</font>
18 <br><br>
19 This is the second subsection of my content. I think is more interestin🙄

```

If you try our longer version out with a screenreader (see [🔗 bad-semantics.html](#)), you'll not have a very good experience — the screenreader hasn't got anything to use as signposts, so you can't retrieve a useful table of contents, and the whole page is seen as a single giant block, so it is just read out in one go, all at once.

There are other issues too beyond accessibility — it is harder to style the content using CSS, or manipulate it with JavaScript for example, because there are no elements to use as selectors.

## Using clear language

The language you use can also affect accessibility. In general you should use clear language that is not overly complex, and doesn't use unnecessary jargon or slang terms. This not only benefits people with cognitive or other disabilities; it benefits readers for whom the text is not written in their first language, younger people ... everyone in fact! Apart from this, you should try to avoid using language and characters that don't get read out clearly by the screenreader. For example:

- Don't use dashes if you can avoid it. Instead of writing 5–7, write 5 to 7.
- Expand abbreviations — instead of writing Jan, write January.
- Expand acronyms, at least once or twice. Instead of writing HTML in the first instance, write Hypertext Markup Language, or HTML.

## Page layouts

In the bad old days, people used to create page layouts using HTML tables — using different table cells to contain the header, footer, side bar, main content column, etc. This is not a good idea because a screenreader will likely give out confusing readouts, especially if the layout is complex and has many nested tables.

Try our example [🔗 table-layout.html](#) example, which looks something like this:

```

1 <table width="1200">
2   <!-- main heading row -->
3   <tr id="heading">
4     <td colspan="6">
5
6       <h1 align="center">Header</h1>
7
8     </td>
9   </tr>
10  <!-- nav menu row -->
    <tr id="nav" bgcolor="#ffffff">

```

```
11 <td width="200">
12   <a href="#" align="center">Home</a>
13 </td>
14 <td width="200">
15   <a href="#" align="center">Our team</a>
16 </td>
17 <td width="200">
18   <a href="#" align="center">Projects</a>
19 </td>
20 <td width="200">
21   <a href="#" align="center">Contact</a>
22 </td>
23 <td width="300">
24   <form width="300">
25     <input type="search" name="q" placeholder="Search query" width="250" />
26   </form>
27 </td>
28 <td width="100">
29   <button width="100">Go!</button>
30 </td>
31 </tr>
32 <!-- spacer row -->
33 <tr id="spacer" height="10">
34   <td>
35
36   </td>
37 </tr>
38 <!-- main content and aside row -->
39 <tr id="main">
40   <td id="content" colspan="4" bgcolor="#ffffff">
41
42     <!-- main content goes here -->
43   </td>
44   <td id="aside" colspan="2" bgcolor="#ff80ff" valign="top">
45     <h2>Related</h2>
46
47     <!-- aside content goes here -->
48
49   </td>
50 </tr>
51 <!-- spacer row -->
52 <tr id="spacer" height="10">
53   <td>
54
55   </td>
56 </tr>
57 <!-- footer row -->
58 <tr id="footer" bgcolor="#ffffff">
59   <td colspan="6">
```

```
60         <p>©Copyright 2050 by nobody. All rights reversed.</p>
61     </td>
62 </tr>
63 </table>
64
```

If you try to navigate this using a screenreader, it will probably tell you that there's a table to be looked at (although some screenreaders can guess the difference between table layouts and data tables). You'll then likely (depending on which screenreader you're using) have to go down into the table as an object and look at its features separately, then get out of the table again to carry on navigating the content.

Table layouts are a relic of the past — they made sense back when CSS support was not widespread in browsers, but they create confusion for screenreader users, as well as being bad for many other reasons (abuse of tables, arguably requires more markup, make designs more inflexible). Don't do it!

You can verify these claims by comparing your previous experience with a [more modern website structure example](#), which could look something like this:

```
1  <header>
2    <h1>Header</h1>
3  </header>
4
5  <nav>
6    <!-- main navigation in here -->
7  </nav>
8
9  <!-- Here is our page's main content -->
10 <main>
11
12   <!-- It contains an article -->
13   <article>
14     <h2>Article heading</h2>
15
16     <!-- article content in here -->
17   </article>
18
19   <aside>
20     <h2>Related</h2>
21
22     <!-- aside content in here -->
23   </aside>
24
25 </main>
26
27 <!-- And here is our main footer that is used across all the pages of our
28
29 <footer>
30
```

```
31 | <!-- footer content in here -->
    | </footer>
```

If you try our more modern structure example with a screenreader, you'll see that the layout markup no longer gets in the way and confuses the content readout. It is also much leaner and smaller in terms of code size, which means easier to maintain code, and less bandwidth for the user to download (particularly prevalent for those on slow connections).

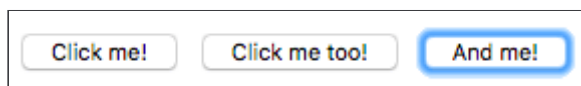
Another consideration when creating layouts is using HTML5 semantic elements as seen in the above example (see [content sectioning](#)) — you can create a layout using only nested `<div>` elements, but it is better to use appropriate sectioning elements to wrap your main navigation (`<nav>`), footer (`<footer>`), repeating content units (`<article>`), etc. These provide extra semantics for screenreaders (and other tools) to give user extra clues about the content they are navigating (see [Screen Reader Support for new HTML5 Section Elements](#) for an idea of what screen reader support is like).

**Note:** As well as your content having good semantics and an attractive layout, it should make logical sense in its source order — you can always place it where you want using CSS later on, but you should get the source order right to start with, so what screenreader users get read out to them will make sense.

## UI controls

By UI controls, we mean the main parts of web documents that users interact with — most commonly buttons, links, and form controls. In this section we'll look at the basic accessibility concerns to be aware of when creating such controls. Later articles on WAI-ARIA and multimedia will look at other aspects of UI accessibility.

One key aspect to the accessibility of UI controls is that by default, browsers allow them to be manipulated by the keyboard. You can try this out using our [native-keyboard-accessibility.html](#) example (see the [source code](#)) — open this in a new tab, and try pressing the tab key; after a few presses, you should see the tab focus start to move through the different focusable elements; the focused elements are given a highlighted default style in every browser (it differs slightly between different browsers) so that you can tell what element is focused.



You can then press Enter/Return to follow a focused link or press a button (we've included some JavaScript to make the buttons alert a message), or start typing to enter text in a text input (other form elements have different controls, for example the `<select>` element can have its options displayed and cycled between using the up and down arrow keys).

**Note:** Different browsers may have different keyboard control options available. See [Using native keyboard accessibility](#) for more details.

You essentially get this behavior for free, just by using the appropriate elements, e.g.

```
1 <h1>Links</h1>
2
3 <p>This is a link to <a href="https://www.mozilla.org">Mozilla</a>.</p>
4
5 <p>Another link, to the <a href="https://developer.mozilla.org">Mozilla D
6
7 <h2>Buttons</h2>
8
9 <p>
10   <button data-message="This is from the first button">Click me!</button>
11   <button data-message="This is from the second button">Click me too!</bu
12   <button data-message="This is from the third button">And me!</button>
13 </p>
14
15 <h2>Form</h2>
16
17 <form>
18   <div>
19     <label for="name">Fill in your name:</label>
20     <input type="text" id="name" name="name">
21   </div>
22   <div>
23     <label for="age">Enter your age:</label>
24     <input type="text" id="age" name="age">
25   </div>
26   <div>
27     <label for="mood">Choose your mood:</label>
28     <select id="mood" name="mood">
29       <option>Happy</option>
30       <option>Sad</option>
31       <option>Angry</option>
32       <option>Worried</option>
33     </select>
34   </div>
35 </form>
```



This means using links, buttons, form elements, and labels appropriately (including the `<label>` element for form controls).

However, it is again the case that people sometimes do strange things with HTML. For example, you sometimes see buttons marked up using `<div>`s, for example:

```
1 <div data-message="This is from the first button">Click me!</div>
2 <div data-message="This is from the second button">Click me too!</div>
3 <div data-message="This is from the third button">And me!</div>
```





But using such code is not advised — you immediately lose the native keyboard accessibility you would have had if you'd just used `<button>` elements, plus you don't get any of the default CSS styling that buttons get.

## Building keyboard accessibility back in

Adding such advantages back in takes a bit of work (you can see an example code in our [fake-div-buttons.html](#) example — also see the [source code](#)). Here we've given our fake `<div>` buttons the ability to be focused (including via tab) by giving each one the attribute `tabindex="0"`:

```
1 | <div data-message="This is from the first button" tabindex="0">Click me!<
2 | <div data-message="This is from the second button" tabindex="0">Click me
3 | <div data-message="This is from the third button" tabindex="0">And me!</d
```


Basically, the `tabindex` attribute is primarily intended to allow tabbable elements to have a custom tab order (specified in positive numerical order), instead of just being tabbed through in their default source order. This is nearly always a bad idea, as it can cause major confusion. Use it only if you really need to, for example if the layout shows things in a very different visual order to the source code, and you want to make things work more logically. There are two other options for `tabindex`:

- `tabindex="0"` — as indicated above, this value allows elements that are not normally tabbable to become tabbable. This is the most useful value of `tabindex`.
- `tabindex="-1"` — this allows not normally tabbable elements to receive focus programmatically, e.g. via JavaScript, or as the target of links.

Whilst the above addition allows us to tab to the buttons, it does not allow us to activate them via the Enter/Return key. To do that, we had to add the following bit of JavaScript trickery:

```
1 | document.onkeydown = function(e) {
2 |     if(e.keyCode === 13) { // The Enter/Return key
3 |         document.activeElement.onclick(e);
4 |     }
5 | };
```

Here we add a listener to the `document` object to detect when a button has been pressed on the keyboard. We check what button was pressed via the event object's `keyCode` property; if it is the keycode that matches Return/Enter, we run the function stored in the button's `onclick` handler using `document.activeElement.onclick()`. `activeElement` gives us the element that is currently focused on the page.

 **Note:** You should bear in mind that this technique will only work if you set your original event handlers via event handler properties (e.g. `onclick`). `addEventListener` won't work.

This is a lot of extra hassle to build the functionality back in. And there's bound to be other problems with it. **Better to just use the right element for the right job in the first place.**

## Meaningful text labels

UI control text labels are very useful to all users, but getting them right is particularly important to users with disabilities.

You should make sure that your button and link text labels are understandable and distinctive. Don't just use "Click here" for your labels, as screenreader users sometimes get up a list of buttons and form controls. The following screenshot shows our controls being listed by VoiceOver on Mac.




Make sure your labels make sense out of context, read on their own, as well as in the context of the paragraph they are in. For example, the following shows an example of good link text:

```
1 | <p>Whales are really awesome creatures. <a href="whales.html">Find out 😊
```

but this is bad link text:

```
1 | <p>Whales are really awesome creatures. To find more out about whales, 😞
```

 **Note:** You can find a lot more about link implementation and best practices in our [Creating hyperlinks](#) article. You can also see some good and bad examples at [good-links.html](#) and [bad-links.html](#).

Form labels are also important, for giving you a clue what you need to enter into each form input. The following seems like a reasonable enough example:

```
1 | Fill in your name: <input type="text" id="name" name="name">
```



However, this is not so useful for disabled users. There is nothing in the above example to associate the label unambiguously with the form input, and make it clear how to fill it in if you cannot see it. If you access this with some screenreaders, you'll may only be given a description along the lines of "edit text".

The following is a much better example:

```
1 | <div>
2 |   <label for="name">Fill in your name:</label>
3 |   <input type="text" id="name" name="name">
4 | </div>
```



With the code like this, the label will be clearly associated with the input; the description will be more like "Fill in your name: edit text".

**Fill in your name: edit text**

As an added bonus, in most browsers associating a label with a form input means that you can click the label to select/activate the form element. This gives the input a bigger hit area, making it easier to select.

 **Note:** you can see some good and bad form examples in [good-form.html](#) and [bad-form.html](#).

## Accessible data tables

A basic data table can be written with very simple markup, for example:

```
1 | <table>
2 |   <tr>
3 |     <td>Name</td>
4 |     <td>Age</td>
5 |     <td>Gender</td>
6 |   </tr>
7 |   <tr>
8 |     <td>Gabriel</td>
9 |     <td>13</td>
10 |    <td>Male</td>
11 |   </tr>
12 | </table>
```

```
13     <td>Elva</td>
14     <td>8</td>
15     <td>Female</td>
16 </tr>
17 <tr>
18     <td>Freida</td>
19     <td>5</td>
20     <td>Female</td>
21 </tr>
22 </table>
```

But this has problems — there is no way for a screenreader user to associate rows or columns together as groupings of data. To do this you need to know what the header rows are, and if they are heading up rows, columns, etc. This can only be done visually for the above table (see [bad-table.html](#) and try the example out yourself).

Now have a look at our [punk bands table example](#) — you can see a few accessibility aids at work here:

- Table headers are defined using `<th>` elements — you can also specify if they are headers for rows or columns using the `scope` attribute. This gives you complete groups of data that can be consumed by screen readers as single units.
- The `<caption>` element and `<table>` `summary` attribute both do similar jobs — they act as alt text for a table, giving a screen reader user a useful quick summary of the table's contents. `<caption>` is generally preferred as it makes it's content accessible to sighted users too, who might also find it useful. You don't really need both.

 **Note:** See our [HTML table advanced features and accessibility](#) article for some more details around accessible data tables.


## Text alternatives

Whereas textual content is inherently accessible, the same cannot necessarily be said for multimedia content — image/video content cannot be seen by visually-impaired people, and audio content cannot be heard by hearing-impaired people. We'll cover video and audio content in detail in the Accessible multimedia article later on, but for this article we'll look accessibility for the humble `<img>` element.

We have a simple example written up, [accessible-image.html](#), which features four copies of the same image:

```
1 
2
3 
9
10
11 
12
13 <p id="dino-label">The Mozilla red Tyrannosaurus Rex: A two legged dinosa
```


The first image, when viewed by a screen reader, doesn't really offer the user much help — VoiceOver for example reads out `/dinosaur.png, image`. It reads out the filename to try to provide some help. In this example the user will at least know it is a dinosaur of some kind, but often files may be uploaded with machine generated file names (e.g. from a digital camera) and these file names would likely provide no context to the image's content.

 **Note:** This is why you should never include text content inside an image — screen readers simply can't access it. There are other disadvantages too — you can't select it and copy/paste it. Just don't do it!

When a screen reader encounters the second image, it reads out the full alt attribute — "A red Tyrannosaurus Rex: A two legged dinosaur standing upright like a human, with small arms, and a large head with lots of sharp teeth."

This highlights the importance of not only using meaningful file names in case so-called **alt text** is not available, but also making sure that alt text is provided in alt attributes wherever possible. Note that the contents of the alt attribute should always provide a direct representation of the image and what it conveys visually. Any personal knowledge or extra description shouldn't be included here, as it is not useful for people who have not come across the image before.

One thing to consider is whether your images have meaning inside your content, or whether they are purely for visual decoration, so have no meaning. If they are decorative, it is better to just include them in the page as CSS background images.

 **Note:** Read [Images in HTML](#) and [Responsive images](#) for a lot more information about image implementation and best practices.


If you do want to provide extra contextual information, you should should put it in the text surrounding the image, or inside a title attribute, as shown above. In this case, most screenreaders will read out the alt text, the title attribute, and the filename. In addition, browsers display title text as tooltips when moused over.



Let's have another quick look at the fourth method:

```
1 | 
2 |
3 | <p id="dino-label">The Mozilla red Tyrannosaurus ... </p>
```

In this case, we are not using the `alt` attribute at all — instead, we have presented our description of the image as a regular text paragraph, given it an `id`, and then used the `aria-labelledby` attribute to refer to that `id`, which causes screenreaders to use that paragraph as the alt text/label for that image. This is especially useful if you want to use the same text as a label for multiple images — something that isn't possible with `alt`.

 **Note:** `aria-labelledby` is part of the [WAI-ARIA](#) spec, which allows developers to add in extra semantics to their markup to improve screenreader accessibility where needed. To find out more about how it works, read our [WAI-ARIA Basics](#) article.

## Other text alternative mechanisms

Images also have another mechanisms available for providing descriptive text. For example, there is a `longdesc` attribute that is meant to point to a separate web document containing an extended description of the image, for example:

```
1 | 
```

This sounds like a good idea, especially for infographics like big charts with lots of information on that could perhaps be represented as an accessible data table instead (see previous section). However, `longdesc` is not supported consistently by screenreaders, and the content is completely inaccessible to non-screenreader users. It is arguably much better to include the long description on the same page as the image, or link to it with a regular link.

HTML5 includes two new elements — `<figure>` and `<figcaption>` — which are supposed to associate a figure of some kind (it could be anything, not necessarily an image) with a figure caption:

```
1 <figure>
2   
3   <figcaption>A red Tyrannosaurus Rex: A two legged dinosaur standing upr
4 </figure>
```


Unfortunately, most screenreaders don't seem to associate figure captions with their figures yet, but the element structure is useful for CSS styling, plus it provides a way to place a description of the image next to it in the source.

## Empty alt attributes

```
1 <h3>
2   
3   Tyrannosaurus Rex: the king of the dinosaurs
4 </h3>
```

There may be times where an image is included in a page's design, but its primary purpose is for visual decoration. You'll notice in the code example above that the image's `alt` attribute is empty — this is to make screen readers recognize the image, but not attempt to describe the image (instead they'd just say "image", or similar).

The reason to use an empty `alt` instead of not including it is because many screen readers announce the whole image URL if no `alt` is provided. In the above example, the image is acting as a visual decoration to the heading its associated with. In cases like this, and in cases where an image is only decoration and has no content value, you should put an empty `alt` on your images. Another alternative is to use the `aria` role attribute `role="presentation"` — this also stops screens readers from reading out alternative text.

 **Note:** if possible you should use CSS to display images that are only decoration.

## Summary

You should now be well-versed in writing accessible HTML for most occasions. Our WAI-ARIA basics article will also fill in some gaps in this knowledge, but this article has taken care of the basics. Next up we'll explore CSS and JavaScript, and how accessibility is affected by their good or bad use.

[< Previous](#)[↑ Overview: Accessibility](#)[Next >](#)

Was this article helpful?



# Learn the best of web development



Get the latest and greatest from MDN delivered straight to your inbox.

SIGN UP NOW









