

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in !\[\]\(d66ff64371a51729ac8c1cdaa685ba6f_img.jpg\)](#)

Handling common accessibility problems

[← Previous](#)[↑ Overview: Cross browser testing](#)[Next →](#)

Next we turn our attention to accessibility, providing information on common problems, how to do simple testing, and how to make use of auditing/automation tools for finding accessibility issues.

Prerequisites:	Familiarity with the core HTML, CSS, and JavaScript languages; an idea of the high level principles of cross browser testing.
Objective:	To be able to diagnose common Accessibility problems, and use appropriate tools and techniques to fix them.

What is accessibility?

When we say accessibility in the context of web technology, most people immediately think of making sure websites/apps are usable by people with disabilities, for example:


- Visually impaired people using screen readers or magnification/zoom to access text
- People with motor function impairments using the keyboard (or other non-mouse features) to activate website functionality.
- People with hearing impairments relying on captions/subtitles or other text alternatives for audio/video content.

However, it is wrong to say that accessibility is just about disabilities. Really, the aim of accessibility is to make your websites/apps are usable by as many people in as many contexts as possible, not just those users using high-powered desktop computers. Extreme examples might include:

- Users on mobile devices.
- Users on alternative browsing devices such as TVs, watches, etc.
- Users of older devices that might not have the latest browsers.
- Users of lower spec devices that might have slow processors.

In a way, this whole module is about accessibility — cross browser testing makes sure that your sites can be used by as many people as possible. [What is accessibility?](#) defines accessibility more completely and thoroughly than this article does.


That said, this article will cover cross browser and testing issues surrounding people with disabilities, and how they use the Web. We've already talked about other spheres such as [responsive design](#) and [performance](#) in other places in the module.

 **Note:** Like many things in web development, accessibility isn't about 100% success or not; 100% accessibility is pretty much impossible to achieve for all content, especially as sites get more complex. Instead, it is more about making an effort to make as much of your content accessible to as many people as possible, via defensive coding, and sticking to best practices.

Common accessibility issues

In this section we'll detail some of the main issues that arise around web accessibility, connected with specific technologies, along with best practices to follow, and some quick

tests you can do to see if your sites are going in the right direction.

 **Note:** Accessibility is morally the right thing to do, and good for business (numbers of disabled users, users on mobile devices, etc. present significant market segments), but it is also against the law in many parts of the world to not make web properties accessible to people with disabilities. Read [Accessibility guidelines and the law](#) for more information.

HTML

Semantic HTML (where the elements are used for their correct purpose) is accessible right out of the box — such content is readable by sighted viewers (provided you don't do anything silly like make the text way too small or hide it using CSS), but will also be usable by assistive technologies like screen readers (apps that literally read out a web page to their user), and confer other advantages too.

Semantic structure

The most important quick win in semantic HTML is to use a structure of headings and paragraphs for your content; this is because screen reader users tend to use the headings of a document as signposts to find the content they need more quickly. If your content has no headings, all they will get is a huge wall of text with no signposts to find anything. Examples of bad and good HTML:

```
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  <font size="5">My subheading</font>
8  <br><br>
9  This is the first subsection of my document. I'd love people to be able to
10 <br><br>
11 <font size="5">My 2nd subheading</font>
12 <br><br>
13 This is the second subsection of my content. I think is more interesting +
```



```
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 <h2>My subheading</h2>
8
9 <p>This is the first subsection of my document. I'd love people to be able
10
11 <h2>My 2nd subheading</h2>
12
13 <p>This is the second subsection of my content. I think is more interesting
```



In addition, your content 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.

As a test, you can turn off a site's CSS and see how understandable it is without it. You could do this manually by just removing the CSS from your code, but the easiest way is to use browser features, for example:

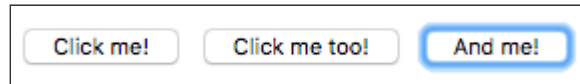
- Firefox: Select *View > Page Style > No Style* from the main menu.
- Safari: Select *Develop > Disable Styles* from the main menu (to enable the *Develop* menu, choose *Safari > Preferences > Advanced > Show Develop menu in menu bar*).
- Chrome: Install the Web Developer Toolbar extension, then restart the browser. Click the gear icon that will appear, then select *CSS > Disable All Styles*.
- Edge: Select *View > Style > No Style* from the main menu.

Using native keyboard accessibility

Certain HTML features can be selected using only the keyboard — this is default behavior, available since the early days of the web. The elements that have this capability are the common ones that allow user to interact with web pages, namely links, `<button>`s, and form elements like `<input>`.

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 that different browsers may have different keyboard control options available. Most modern browsers follow the tab pattern described above (you can also do Shift + Tab to move backwards through the focusable elements), but some browsers have their own idiosyncracies:

- Firefox for the Mac doesn't do tabbing by default. To turn it on, you have to go to *Preferences > Advanced > General*, then uncheck "Always use the cursor keys to navigate within pages". Next, you have to open your Mac's System Preferences app, then go to *Keyboard > Shortcuts*, then select the *All Controls* radio button.
- Safari doesn't allow you to tab through links by default; to enable this, you need to open Safari's *Preferences*, go to *Advanced*, and check the *Press Tab to highlight each item on a webpage* checkbox.

! Important: You should perform this kind of test on any new page you write — make sure the functionality can be accessed by the keyboard.

This example highlights the importance of using the correct semantic element for the correct job. It is possible to style *any* element to look like a link or button with CSS, and to behave like a link or button with JavaScript, but they won't actually be links or buttons, and you'll lose a lot of the accessibility these elements give you for free. So don't do it if you can avoid it.

Another tip — as shown in our example, you can control how your focusable elements look when focused, using the `:focus` pseudo-class. It is a good idea to double up focus and hover styles, so your users get that visual clue that a control will do something when activated, whether they are using mouse or keyboard:

```
1 a:hover, input:hover, button:hover, select:hover,  
2 a:focus, input:focus, button:focus, select:focus {
```

```
3 | font-weight: bold;  
4 | }
```

Note: If you do decide to remove the default focus styling using CSS, make sure you replace it with something else that fits in with your design better — it is a very valuable accessibility tool, and should not be removed.


Building in keyboard accessibility

Sometimes it is not possible to avoid losing keyboard accessibility. You might have inherited a site where the semantics are not very good (perhaps you've ended up with a horrible CMS that generates buttons made with `<div>`s), or you are using a complex control that does not have keyboard accessibility built in, like the HTML5 `<video>` element (amazingly, Opera is the only browser that allows you to tab through the `<video>` element's default browser controls). You have a few options here:

1. Create custom controls using `<button>` elements (which we can tab to by default!) and JavaScript to wire up their functionality. See [Creating a cross-browser video player](#) for some good examples of this.
2. Create keyboard shortcuts via JavaScript, so functionality is activated when you press certain keys on the keyboard. See [Desktop mouse and keyboard controls](#) for some game-related examples that can be adapted for any purpose.
3. Use some interesting tactics to fake button behaviour. Take for example our [fake-div-buttons.html](#) example (see [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"` (see WebAIM's [tabindex](#) article for more really useful details). This allows us to tab to the buttons, but not 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:** 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.

Text alternatives

Text alternatives are very important for accessibility — if a person has a visual or hearing impairment that stops them being able to see or hear some content, then this is a problem. The simplest text alternative available is the humble `alt` attribute, which we should include on all images that contain relevant content. This should contain a description of the image that successfully conveys its meaning and content on the page, to be picked up by a screenreader and read out to the user.

 **Note:** For more information, read [Text alternatives](#).

Missing alt text can be tested for in a number of ways, for example using accessibility Auditing tools.


Alt text is slightly more complex for video and audio content. There is a way to define text tracks (e.g. subtitles) and display them when video is being played, in the form of the `<track>` element, and the WebVTT format (see [Adding captions and subtitles to HTML5 video](#) for a detailed tutorial). [Browser compatibility](#) for these features is fairly good, but if you want to provide text alternatives for audio or support older browsers, a simple text transcript presented somewhere on the page or on a separate page might be a good idea.

Element relationships and context

There are certain features and best practices in HTML designed to provide context and relationships between elements where none otherwise exists. The three most common examples are links, form labels, and data tables.

The key to accessible link text is that people using screen readers will often use a common feature whereby they pull up a list of all the links on the page. In this case, the link text needs to make sense out of context. For example, a list of links labeled "click here", "click here", etc. is really bad for accessibility. It is better for link text to make sense in context and out of context.

Next on our list, the form `<label>` element is one of the central features that allows us to make forms accessible. The trouble with forms is that you need labels to say what data should be entered into each form input. Each label needs to be included inside a `<label>` to link it unambiguously to its partner form input (each `<label>` for attribute value needs to match the form element `id` value), and it will make sense even if the source order is not completely logical (which to be fair it should be).

 **Note:** For more information about link text and form labels, read [Meaningful text labels](#).

Finally, a quick word about data tables. A basic data table can be written with very simple markup (see `bad-table.html` [live](#), and [source](#)), but this has problems — there is no way for a screen reader 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 such a table.

If you instead look at our `punk-bands-complete.html` example ([live](#), [source](#)), you can see a few accessibility aids at work here, such as table headers (`<th>` and `scope` attributes), `<caption>` element, etc.

 **Note:** For more information about accessible tables, read [Accessible data tables](#).

CSS

CSS tends to provide a lot fewer fundamental accessibility features than HTML, but it can still do just as much damage to accessibility if used incorrectly. We have already mentioned a couple of accessibility tips involving CSS:

- Use the correct semantic elements to mark up different content in HTML; if you want to create a different visual effect, use CSS — don't abuse an HTML element to get the look you want. For example, if you want bigger text, use `font-size`, not an `<h1>` element.
- Make sure your source order makes sense without CSS; you can always use CSS to style the page any way you want afterward.
- You should make sure interactive elements like buttons and links have appropriate focus/hover/active states set, to give the user visual clues as to their function. If you remove the defaults for stylistic reasons, make sure you include some replacement styles.

There are a few other considerations you should take into account.

Color and color contrast

When choosing a color scheme for your website, you should make sure that the text (foreground) color contrasts well with the background color. Your design might look cool, but it is no good if people with visual impairments like color blindness can't read your content. Use a tool like WebAIM's [Color Contrast Checker](#) to check whether your scheme is contrasting enough.

Another tip is to not rely on color alone for signposts/information, as this will be no good for those who can't see the color. Instead of marking required form fields in red, for example, mark them with an asterisk and in red.

Note: A high contrast ratio will also allow anyone using a smartphone or tablet with a glossy screen to better read pages when in a bright environment, such as sunlight.


Hiding content

There are many instances where a visual design will require that not all content is shown at once. For example, in our [Tabbed info box example](#) (see [source code](#)) we have three panels of information, but we are positioning them on top of one another and providing tabs that can be clicked to show each one (it is also keyboard accessible — you can alternatively use Tab and Enter/Return to select them).



Screen reader users don't care about any of this — they are happy with the content as long as the source order makes sense, and they can get to it all. Absolute positioning (as used in this example) is generally seen as one of the best mechanisms of hiding content for visual effect, because it doesn't stop screen readers from getting to it.

On the other hand, you shouldn't use `visibility :hidden` or `display :none`, because they do hide content from screenreaders. Unless of course, there is a good reason why you want this content to be hidden from screenreaders.

 **Note:** [Invisible Content Just for Screen Reader Users](#) has a lot more useful detail surrounding this topic.

JavaScript

JavaScript has the same kind of problems as CSS with respect to accessibility — it can be disastrous for accessibility if used badly, or overused. We've already hinted at some accessibility problems related to JavaScript, mainly in the area of semantic HTML — you should always use appropriate semantic HTML to implement functionality wherever it is available, for example use links and buttons as appropriate. Don't use `<div>` elements with JavaScript code to fake functionality if at all possible — it is error prone, and more work than using the free functionality HTML gives you.

Simple functionality

Generally simple functionality should work with just the HTML in place — JavaScript should only be used to enhance functionality, not built it in entirely. Good uses of JavaScript include:

- Providing client-side form validation, which alerts users to problems with their form entries quickly, without having to wait for the server to check the data. If it isn't available, the form will still work, but validation might be slower.
- Providing custom controls for HTML5 `<video>`s that are accessible to keyboard-only users (as we said earlier, the default browser controls aren't keyboard-accessible in most browsers).

 **Note:** WebAIM's [Accessible JavaScript](#) provides some useful further details about considerations for accessible JavaScript.

More complex JavaScript implementations can provide issues with accessibility — you need to do what you can. For example, it would be unreasonable to expect you to make a complex 3D game written using [WebGL](#) 100% accessible to a blind person, but you could

implement [keyboard controls](#) so it is usable by non-mouse users, and make the color scheme contrasting enough to be usable by those with color deficiencies.

Complex functionality

One of the main areas problematic for accessibility is complex apps that involve complicated form controls (such as date pickers) and dynamic content that is updated often and incrementally.

Non-native complicated form controls are problematic because they tend to involve a lot of nested `<div>`s, and the browser does not know what to do with them by default. If you are inventing them yourself, you need to make sure that they are keyboard accessible; if you are using some kind of third-party framework, carefully review the options available to see how accessible they are before diving in. [Bootstrap](#) looks to be fairly good for accessibility, for example, although [Making Bootstrap a Little More Accessible](#) by Rhiana Heath explores some of its issues (mainly related to color contrast), and looks at some solutions.

Regularly updated dynamic content can be a problem because screenreader users might miss it, especially if it updates unexpectedly. If you have a single-page app with a main content panel that is regularly updated using [XMLHttpRequest](#) or [Fetch](#), a screenreader user might miss those updates.

WAI-ARIA

Do you need to use such complex functionality, or will plain old semantic HTML do instead? If you do need complexity, you should consider using [WAI-ARIA](#) (Accessible Rich Internet Applications), a specification that provides semantics (in the form of new HTML attributes) for items such as complex form controls and updating panels that can be understood by most browsers and screen readers.

To deal with complex form widgets, you need to use ARIA attributes like `roles` to state what role different elements have in a widget (for example, are they a tab, or a tab panel?), `aria-disabled` to say whether a control is disabled or not, etc.

To deal with regularly updating regions of content, you can use the `aria-live` attribute, which identifies an updating region. Its value indicates how urgently the screen reader should read it out:

- `off`: The default. Updates should not be announced.
- `polite`: Updates should be announced only if the user is idle.
- `assertive`: Updates should be announced to the user as soon as possible.

- **rude**: Updates should be announced straight away, even if this interrupts the user.

Here's an example:

```
1 | <p><span id="LiveRegion1" aria-live="polite" aria-atomic="false"></span></p></pre>
```

You can see an example in action at Freedom Scientific's [ARIA \(Accessible Rich Internet Applications\) Live Regions](#) example — the highlighted paragraph should update its content every 10 seconds, and a screenreader should read this out to the user. [ARIA Live Regions - Atomic](#) provides another useful example.

We don't have space to cover WAI-ARIA in detail here, you can learn a lot more about it at [WAI-ARIA basics](#).

Accessibility tools

Now we've covered accessibility considerations for different web technologies, including a few testing techniques (like keyboard navigation and color contrast checkers), let's have a look at other tools you can make use of when doing accessibility testing.

Auditing tools

There are a number of auditing tools available that you can point at your web pages, which will look over them and return a list of accessibility issues present on the page. Examples include:

- [Tenon](#): A rather nice online app that goes through the code at a provided URL and returns results on accessibility errors including metrics, specific errors along with the WCAG criteria they affect, and suggested fixes.
- [tota11y](#): An accessibility tool from the Khan Academy that takes the form of a JavaScript library that you attach to your page to provide a number of accessibility tools.
- [Wave](#): Another online accessibility testing tool that accepts a web address and returns a useful annotated view of that page with accessibility problems highlighted.





Let's look at an example, using Tenon.

1. Go to the [Tenon](#) homepage.
2. Enter the URL of our [bad-semantics.html](#) example into the text input at the top of the page (or the URL of another webpage you'd like to analyze) and press *Analyse Your Webpage*.
3. Scroll down until you find the error/description section, as shown below.


Error	Description
<pre><body style=""> My heading

 This is the first section of my document.

 I'll add another paragraph here too.

 <font size="5"</pre> <p>Test ID 97</p>	<p> Error / priority 100%</p> <p>Page has no headings</p> <p>(line: 8, 1)</p> <p>Web Content Accessibility Guidelines (WCAG) 2.0, Level A: 1.3.1 Info and Relationships</p> <p>This page has no heading elements. Heading elements provide a number of important benefits to users. They can be useful as wayfinding cues for in-page navigation, they can contribute to a summary of the page, and they can provide context to the content below them.</p> <p> Recommended Fix</p>
<pre><html><head> <meta charset="utf-8"> <title>Bad semantics example</title></pre>	<p> Error / priority 77%</p> <p>The language of this page is not set.</p> <p></p>

There are also some options that you can explore (see the *Show Options* link near the top of the page), as well as an API to use Tenon programmatically.

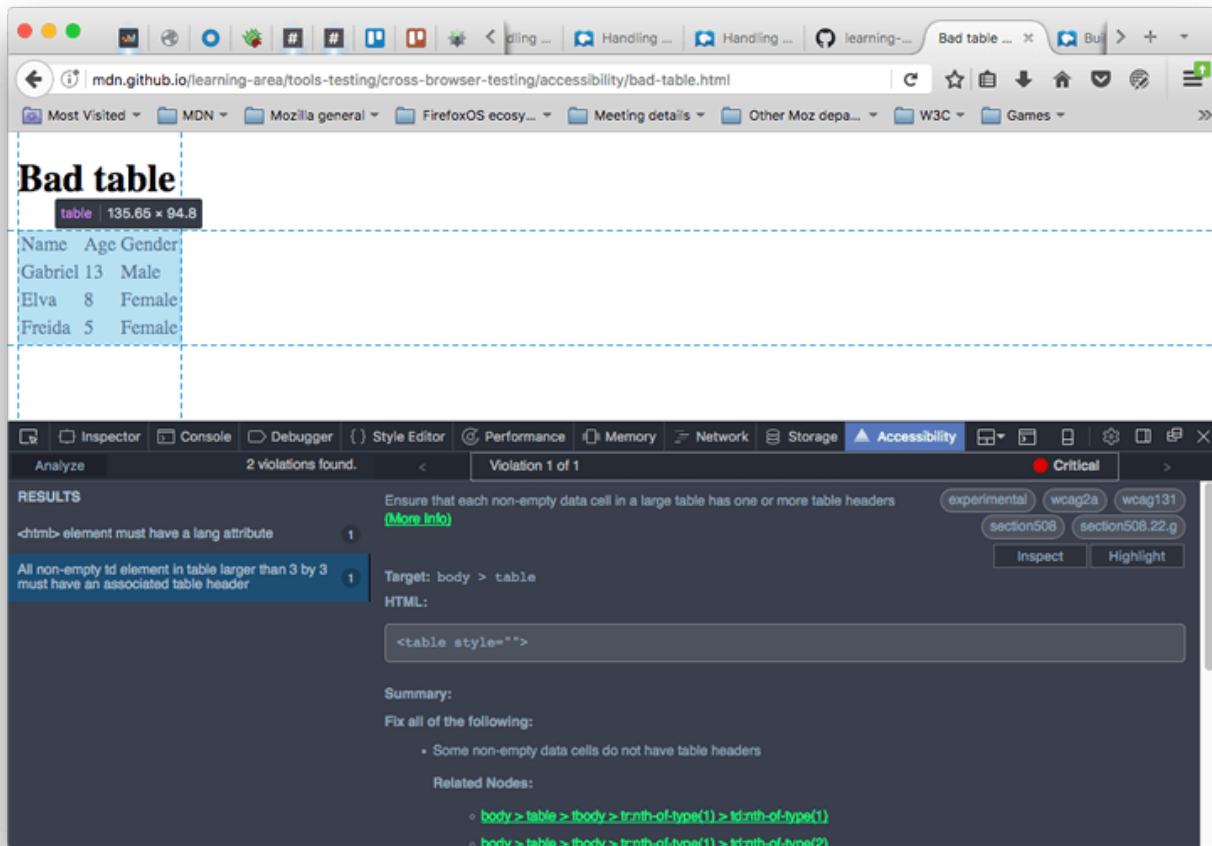
 **Note:** Such tools aren't good enough to solve all your accessibility problems on their own. You'll need a combination of these, knowledge and experience, user testing, etc. to get a full picture.

Automation tools

[Deque's aXe](#) tool goes a bit further than the auditing tools we mentioned above. Like the others, it checks pages and returns accessibility errors. Its most immediately useful form is probably the browser extensions:

- [aXe for Chrome](#)
- [aXe for Firefox](#)

These add an accessibility tab to the browser developer tools. For example, we installed the Firefox version, then used it to audit our [bad-table.html](#) example. We got the following results:



aXe is also installable using npm, and can be integrated with task runners like [Grunt](#) and [Gulp](#), automation frameworks like [Selenium](#) and [Cucumber](#), unit testing frameworks like [Jasmin](#), and more besides (again, see the [main aXe page](#) for details).


Screenreaders

It is definitely worth testing with a screenreader to get used to how severely visually impaired people use the Web. There are a number of screenreaders available:

- Some are paid-for commercial products, like [JAWS](#) (Windows) and [Window Eyes](#) (Windows).
- Some are free products, like [NVDA](#) (Windows), [ChromeVox](#) (Chrome, Windows, and Mac OS X), and [Orca](#) (Linux).
- Some are built into the operating system, like [VoiceOver](#) (Mac OS X and iOS), [ChromeVox](#) (on Chromebooks), and [TalkBack](#) (Android).

Generally, screen readers are separate apps that run on the host operating system and can read not only web pages, but text in other apps as well. This is not always the case (ChromeVox is a browser extension), but usually. Screenreaders tend to act in slightly different ways and have different controls, so you'll have to consult the documentation for your chosen screen reader to get all of the details — saying that, they all work in basically the same sort of way.

Let's go through some tests with a couple of different screenreaders to give you a general idea of how they work and how to test with them.

 **Note:** WebAIM's [Designing for Screen Reader Compatibility](#) provides some useful information about screenreader usage and what works best for screenreaders. Also see [Screen Reader User Survey #6 Results](#) for some interesting screenreader usage statistics.

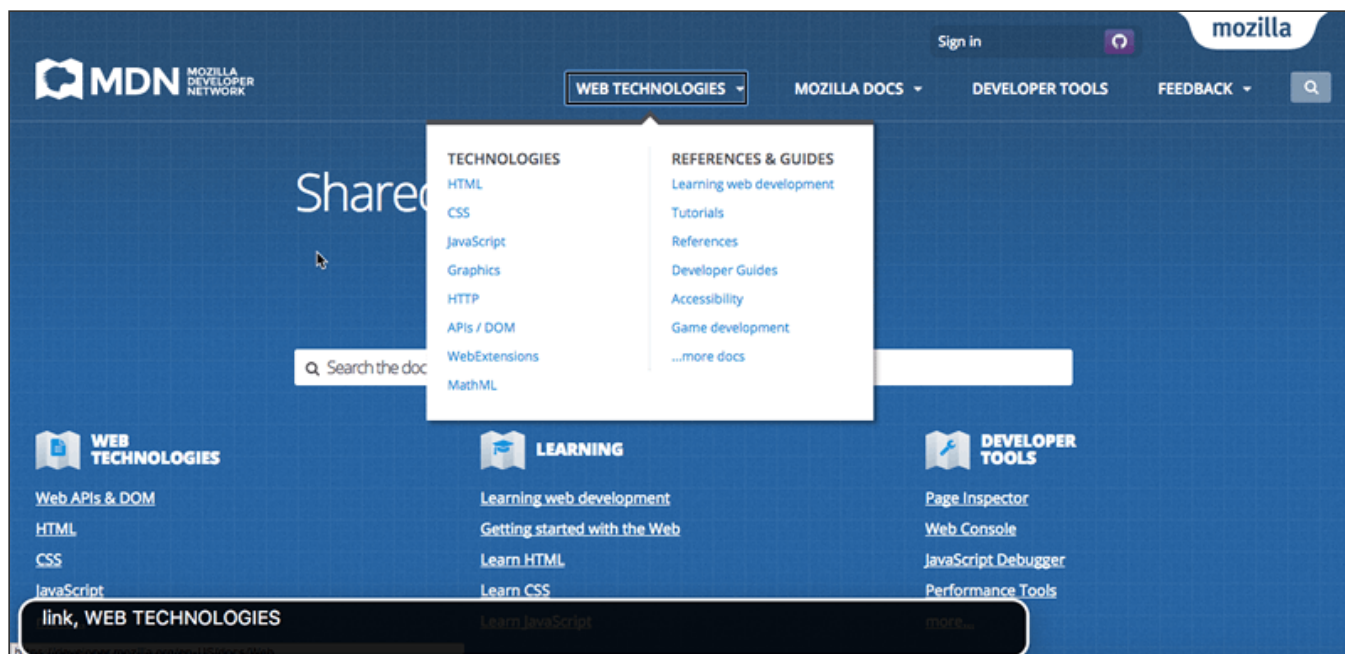
VoiceOver

VoiceOver (VO) comes free with your Mac/iPhone/iPad, so it's useful for testing on desktop/mobile if you use Apple products. We'll be testing it on Mac OS X on a MacBook Pro.

To turn it on, press Cmd + Fn + F5. If you've not used VO before, you will be given a welcome screen where you can choose to start VO or not, and run through a rather useful tutorial to learn how to use it. To turn it off again, press Cmd + Fn + F5 again.

 **Note:** You should go through the tutorial at least once — it is a really useful way to learn VO.

When VO is on, the display will look mostly the same, but you'll see a black box at the bottom left of the screen that contains information on what VO currently has selected. The current selection will also be highlighted, with a black border — this highlight is known as the **VO cursor**.



To use VO, you will make a lot of use of the "VO modifier" — this is a key or key combination that you need to press in addition to the actual VO keyboard shortcuts to get them to work. Using a modifier like this is common with screenreaders, to enable them to keep their commands from clashing with other commands. In the case of VO, the modifier can either be CapsLock, or Ctrl + Option.

VO has many keyboard commands, and we won't list them all here. The basic ones you'll need for web page testing are in the following table. In the keyboard shortcuts, "VO" means "the VoiceOver modifier".

Most common VoiceOver keyboard shortcuts

Keyboard shortcut	Description
VO + Cursor keys	Move the VO cursor up, right, down, left.
VO + Spacebar	Select/activate items highlighted by the VO cursor. This includes items selected in the Rotor (see below).
VO + Shift + down cursor	Move into a group of items (such as an HTML table, or a form, etc.) Once inside a group you can move around and select items inside that group using the above commands as normal.
VO + Shift + up cursor	Move out of a group.
VO + C	(when inside a table) Read the header of the current column.
VO + R	(when inside a table) Read the header of the current row.

Keyboard shortcut	Description
VO + C + C (two Cs in succession)	(when inside a table) Read the entire current column, including header.
VO + R + R (two Rs in succession)	(when inside a table) Read the entire current row, including the headers that correspond to each cell.
VO + left cursor, VO + right cursor	(when inside some horizontal options, such as a date or time picker) Move between options.
VO + up cursor, VO + down cursor	(when inside some horizontal options, such as a date or time picker) Change the current option.
VO + U	Use the Rotor, which displays lists of headings, links, form controls, etc. for easy navigation.
VO + left cursor, VO + right cursor	(when inside Rotor) Move between different lists available in the Rotor.
VO + up cursor, VO + down cursor	(when inside Rotor) Move between different items in the current Rotor list.
Esc	(when inside Rotor) Exit Rotor.
Ctrl	(when VO is speaking) Pause/Resume speech.
VO + Z	Restart the last bit of speech.
VO + D	Go into the Mac's Dock, so you can select apps to run inside it.

This seems like a lot of commands, but it isn't so bad when you get used to it, and VO regularly gives you reminders of what commands to use in certain places. Have a play with VO now; you can then go on to play with some of our examples in the [Screenreader testing section](#).


NVDA

NVDA is Windows-only, and you'll need to install it.

1. Download it from [nvaccess.org](https://www.nvaccess.org). You can choose whether to make a donation or download it for free; you'll also need to give them your e-mail address before you can download it.
2. Once downloaded, install it — you double click the installer, accept the license and follow the prompts.
3. To start NVDA, double click on the program file/shortcut, or use the keyboard shortcut Ctrl + Alt + N. You'll see the NVDA welcome dialog when you start it. Here you can choose from a couple of options, then press the *OK* button to get going.

NVDA will now be active on your computer.

To use NVDA, you will make a lot of use of the "NVDA modifier" — this is a key that you need to press in addition to the actual NVDA keyboard shortcuts to get them to work. Using a modifier like this is common with screenreaders, to enable them to keep their commands from clashing with other commands. In the case of NVDA, the modifier can either be Insert (the default), or CapsLock (can be chosen by checking the first checkbox in the NVDA welcome dialog before pressing *OK*).

 **Note:** NVDA is more subtle than VoiceOver in terms of how it highlights where it is and what it is doing. When you are scrolling through headings, lists, etc., items you are selected on will generally be highlighted with a subtle outline, but this is not always the case for all things. If you get completely lost, you can press Ctrl + F5 to refresh the current page and begin from the top again.

NVDA has many keyboard commands, and we won't list them all here. The basic ones you'll need for web page testing are in the following table. In the keyboard shortcuts, "NVDA" means "the NVDA modifier".

Most common NVDA keyboard shortcuts

Keyboard shortcut	Description
NVDA + Q	Turn NVDA off again after you've started it.
NVDA + up cursor	Read the current line.
NVDA + down cursor	Start reading at the current position.
Up cursor and down cursor, or Shift + Tab and Tab	Move to previous/next item on page and read it.
Left cursor and right cursor	Move to previous/next character in current item and read it.
Shift + H and H	Move to previous/next heading and read it.

Keyboard shortcut	Description
Shift + K and K	Move to previous/next link and read it.
Shift + D and D	Move to previous/next document landmark (e.g. <nav>) and read it.
Shift + 1–6 and 1–6	Move to previous/next heading (level 1–6) and read it.
Shift + F and F	Move to previous/next form input and focus on it.
Shift + T and T	Move to previous/next data table and focus on it.
Shift + B and B	Move to previous/next button and read its label.
Shift + L and L	Move to previous/next list and read its first list item.
Shift + I and I	Move to previous/next list item and read it.
Enter/Return	(when link/button or other activatable item is selected) Activate item.
NVDA + Space	(when form is selected) Enter form so individual items can be selected, or leave form if you are already in it.
Shift Tab and Tab	(when inside form) Move between form inputs.
Up cursor and down cursor	(when inside form) Change form input values (in the case of things like select boxes).
Spacebar	(when inside form) Select chosen value.
Ctrl + Alt + cursor keys	(when a table is selected) Move between table cells.

Screenreader testing

Now you've gotten used to using a screenreader, we'd like you to use it to do some quick accessibility tests, to get an idea of how screenreaders deal with good and bad webpage features:

- Look at [good-semantics.html](#), and note how the headers are found by the screenreader and available to use for navigation. Now look at [bad-semantics.html](#), and note how the screenreader gets none of this information. Imagine how annoying this would when trying to navigate a really long page of text.
- Look at [good-links.html](#), and note how they make sense when viewed out of context. This is not the case with [bad-links.html](#) — they are all just "click here".
- Look at [good-form.html](#), and note how the form inputs are described using their labels because we've used `<label>` elements properly. In [bad-form.html](#), they get

an unhelpful label along the lines of "blank".

- Look at our [punk-bands-complete.html](#) example, and see how the screenreader is able to associate columns and rows of content and read them out all together because we've defined headers properly. In [bad-table.html](#), none of the cells can be associated at all. Note that NVDA seems to behave slightly strangely when you've only got a single table on a page; you could try [WebAIM's table test page](#) instead.
- Have a look at the [WAI-ARIA live regions example](#) we saw earlier, and note how the screen reader will keep reading out the constantly updating section as it updates.

User testing

As mentioned above, you can't rely on automated tools alone for determining accessibility problems on your site. It is recommended that when you draw up your testing plan, you should include some accessibility user groups if at all possible (see our [User Testing](#) section earlier on in the course for some more context). Try to get some screenreader users involved, some keyboard-only users, some non-hearing users, and perhaps other groups too, as suits your requirements.

Accessibility testing checklist

The following list provides a checklist for you to follow to make sure you've carried out the recommended accessibility testing for your project:

1. Make sure your HTML is as semantically correct as possible. [Validating](#) it is a good start, as is using an [Auditing tool](#).
2. Check that your content makes sense when the CSS is turned off.
3. Make sure your functionality is [keyboard accessible](#). Test using Tab, Return/Enter, etc.
4. Make sure your non-text content has [text alternatives](#). an [Auditing tool](#) is good for catching such problems.
5. Make sure your site's [color contrast](#) is acceptable, using a suitable checking tool.
6. Make sure [hidden content](#) is visible by screenreaders.
7. Make sure that functionality is usable without JavaScript wherever possible.
8. Use ARIA to improve accessibility where appropriate.
9. Run your site through an [Auditing tool](#).
10. Test it with a screenreader.

11. Include an accessibility policy/statement somewhere findable on your site to say what you did.
-

Finding help

There are many other issues you'll encounter with accessibility; the most important thing to know really is how to find answers online. Consult the HTML and CSS article's [Finding help](#) section for some good pointers.

Summary

Hopefully this article has given you a good grounding in the main accessibility problems you might encounter, and how to test and overcome them.

In the next article we'll look at feature detection in more detail.

[← Previous](#)[↑ Overview: Cross browser testing](#)[Next →](#)

In this module

- [Introduction to cross browser testing](#)
- [Strategies for carrying out testing](#)
- [Handling common HTML and CSS problems](#)
- [Handling common JavaScript problems](#)
- [Handling common accessibility problems](#)
- [Implementing feature detection](#)
- [Introduction to automated testing](#)

- [Setting up your own test automation environment](#)
-