Blackboard

# 10 Common Gotchas in Accessible Development

2018

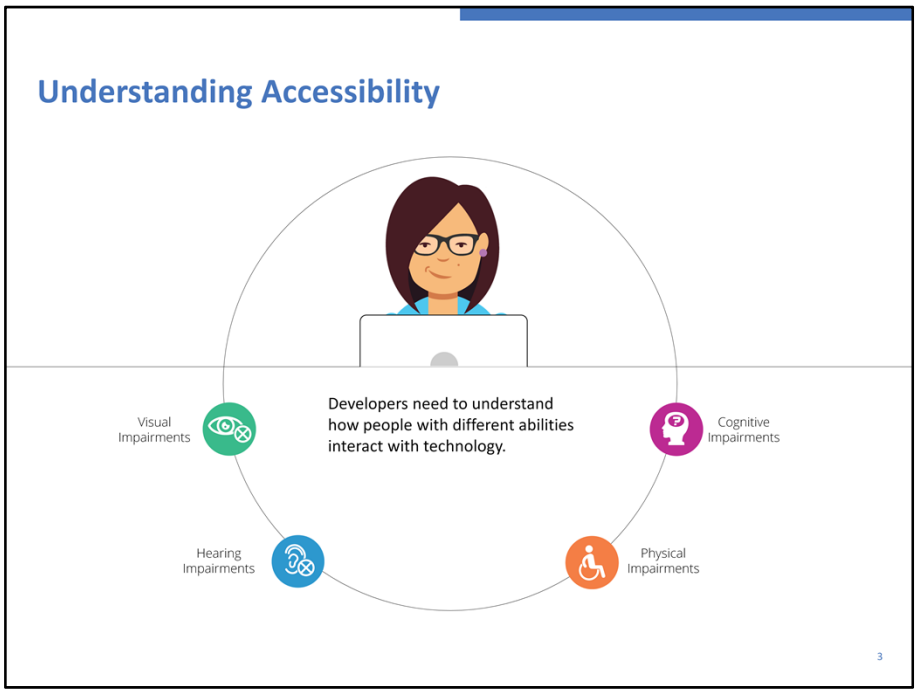
## Who am I?

Elizabeth Simister
Product Accessibility Manager

Bringing now 14 years of experience in assisting educational and government institutions in understanding what it means to be accessible.

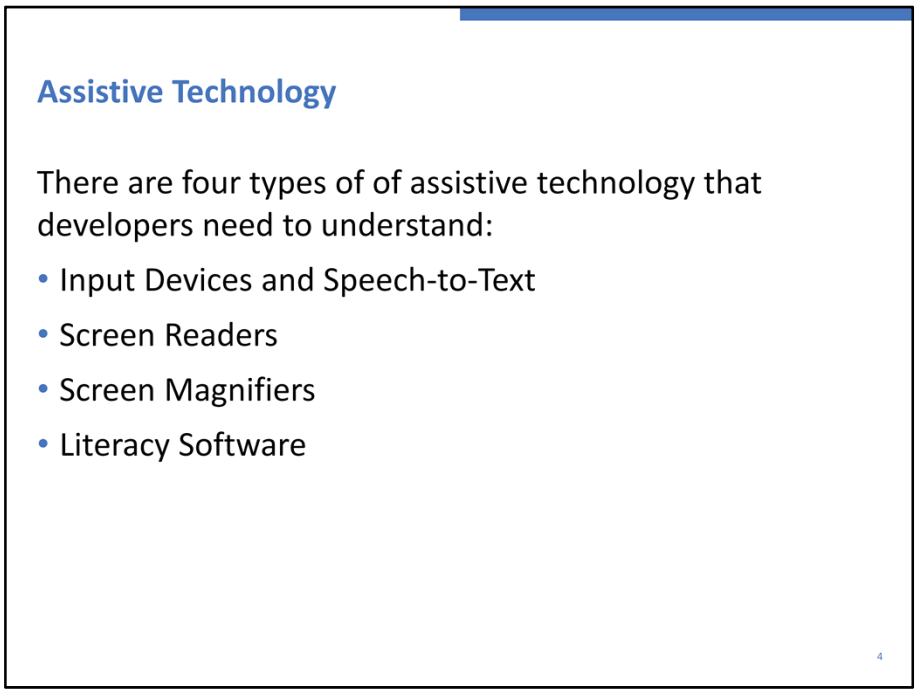
2

My name is Elizabeth Simister and I am the current product accessibility manager at Blackboard. I got my start in accessibility in 2004 in what is now the K. Lisa Yang and Hock E. Tan Institute on Employment and Disability Institute at Cornell University. One of my biggest responsibilities was understanding how to take scanned journal pages with lots of charts, graphs, tables, and math formulas and make them in accessible PDFs. After Cornell, I spent a number of years working as a contractor helping different organizations figure out what they needed to do to be accessible. I've been at Blackboard for about 9 months and my primary focus is on working with our development teams to get them to the point where all of our products are as accessible as possible.




More and more attention is being placed on building technology that works for all types of users. To do that developers need to understand how different types of people interact with their tools and how to apply accessibility coding best practices to create experiences that work for everyone.

Let’s start by understanding a little more about how people with disabilities interact with technology.

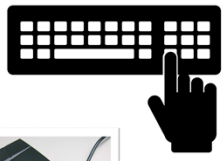



Although there may be similarities between each of these types of ATs, such as screen magnifiers with screen reader keyboard commands, overall each of these technologies comes with its own unique characteristics that can be impacted if code is not designed or developed with accessibility in mind.

## Input Devices and Speech-to-Text

Alternative input devices and speech-to-text tools are used by people with limited mobility or fine muscle control. They can include:

- Keyboards
- Mouth sticks
- Foot pedals
- Voice recognition tools
  - Dragon Naturally Speaking
  - Dictation



DRAGON
NATURALLYSPEAKING

5

There are a variety of input devices that could be used be people with limited mobility. They could range from keyboard and voice recognition tools, to foot pedals and mouth sticks. All of them are intended to assist a user in moving around the interface and interacting with the elements of your site.

## Screen Readers

These tools are used for navigation and text-to-speech by individuals who are blind, have low-vision, or cognitive issues typically via the keyboard or braille refresh devices.


nvaccess

6

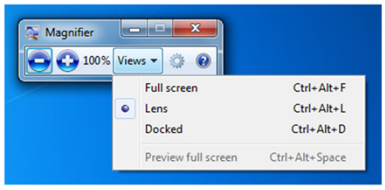
Screen readers that assist with both “reading” and navigation are generally used by individuals with significant vision loss or total blindness. These tools (like JAWS, VoiceOver, and NVDA) allow non-visual users to understand the structure and content within your application as well interact with the elements of the interface. Many of these tools have features built within them that help individuals interact with technology. They have the ability to quickly understand the structure of a page; view a list of links or buttons; move quickly between headings; and many more. The availability of these tools means the way non-visual people interact with technology is very different than how visual people do.

This is where some of the biggest impact for accessible development lives.

## Screen magnifiers

Tools allow low-vision users or users with color perception impairments to configure the “zoom” and color contrast settings. In some instances, these users can access keyboard commands to more easily find information.

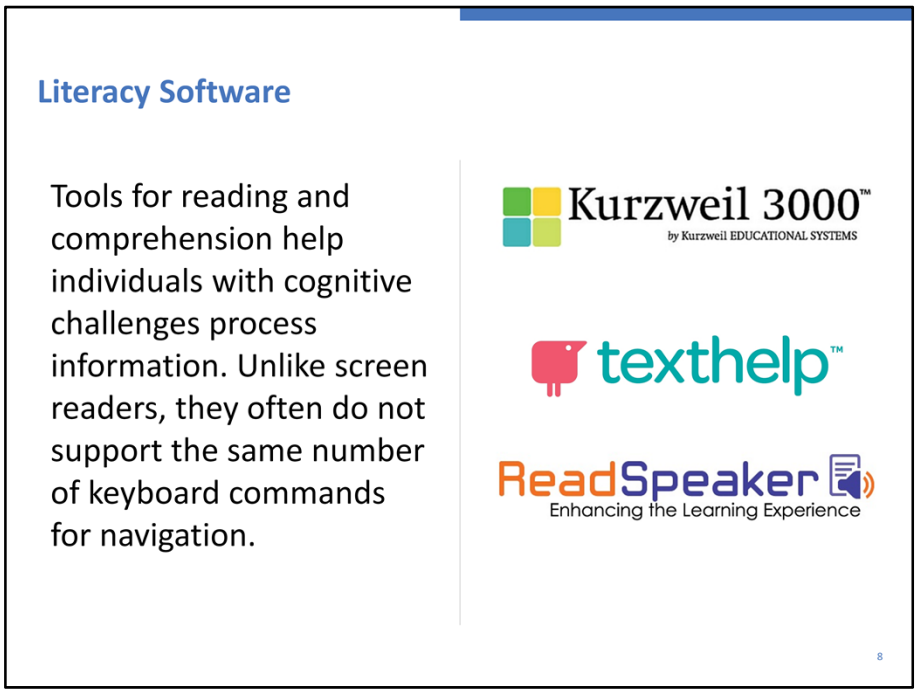
ZoomText




**MAGic® Screen Magnification Software with Speech**

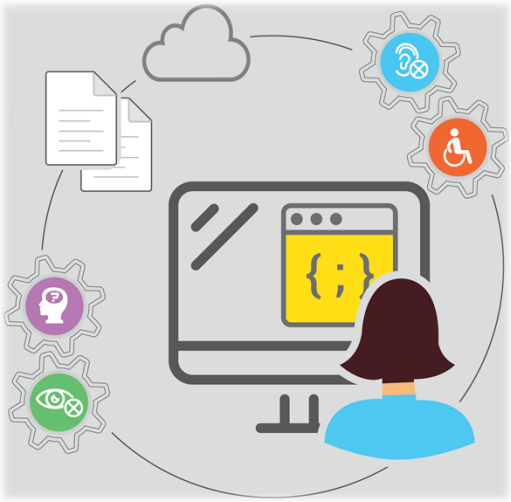
7

Although there are related features between screen readers and screen magnifiers, I want to point out the differences. Th biggest is the ability of most screen magnifier tools to alter the color contrast of content. When color is not considered as part of the design or development process, it can cause these tools not to work smoothly with alternative user set contrasts.




Literacy software is used for reading and comprehension (like TextHelp, Kurzweil, and ReadSpeaker) and is generally used by sighted individuals with cognitive disabilities who need the content to be read aloud to them so they can internalize and understand the information. There are a number of additional features within most of these tools, but for our purposes today, there isn't much else you need to know. If you do everything else right, these will work properly.

## Impact to Developers

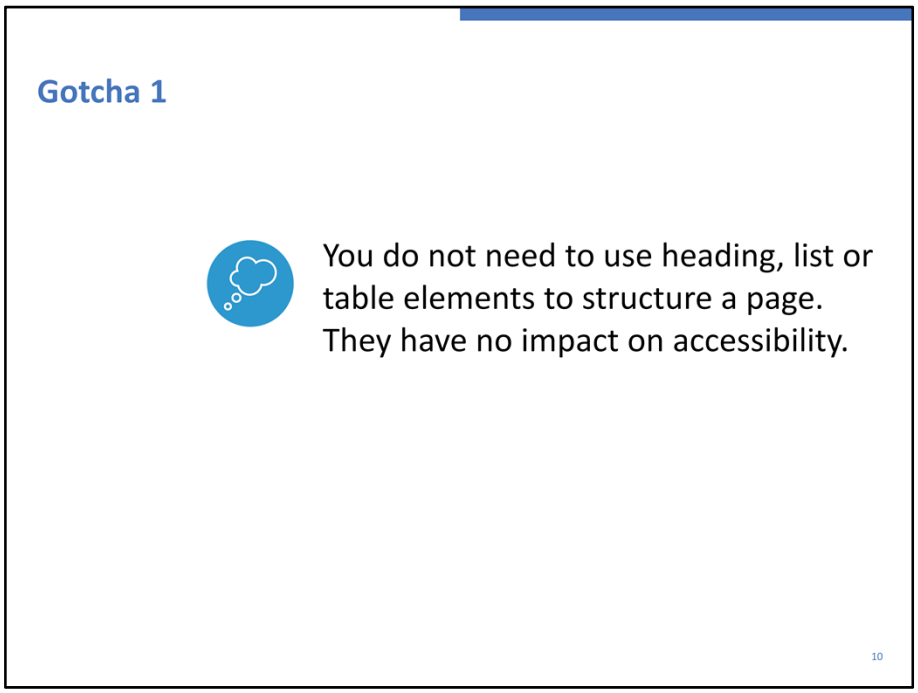



9

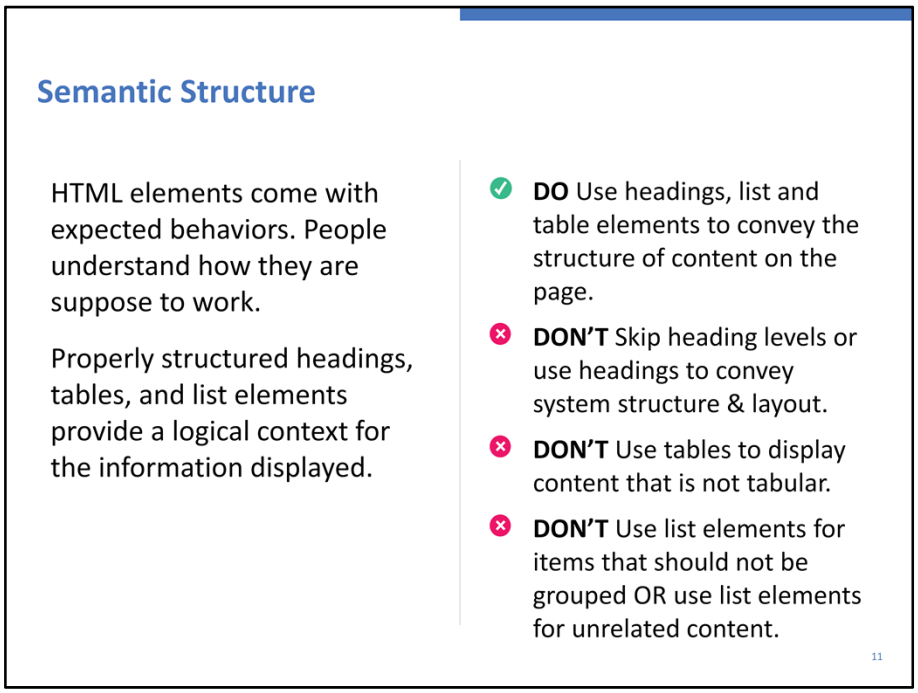
So what does this all mean for developers?

Developers execute a design vision and really bring it to life. That means that they are the foundation for how everything works. An understanding of coding best practices related to accessibility is key to building a solid foundation. But, in today's development world it's still often considered a specialized skill set that needs to be learned. As with learning anything new, there are several easily misunderstood, or misapplied, concepts that can trip up any developer getting started with accessible coding.

Let's spend the rest of the session talking through the 10 most common gotchas in accessible development.




One of the most basic misconceptions is that you can build custom HTML elements or skip over semantic structures like heading levels without impacting accessibility.




This one is more about good quality code than accessibility. Good semantics always matter. HTML has a number of "elements" which contain expected behaviors, styles, etc. This could include things like layout containers, tables, links, buttons, etc.

Non-visual users rely on headings to help them understand the nested structure of content in a page. They rely on headings being logically and consistently used to help them gain context and awareness throughout the application.

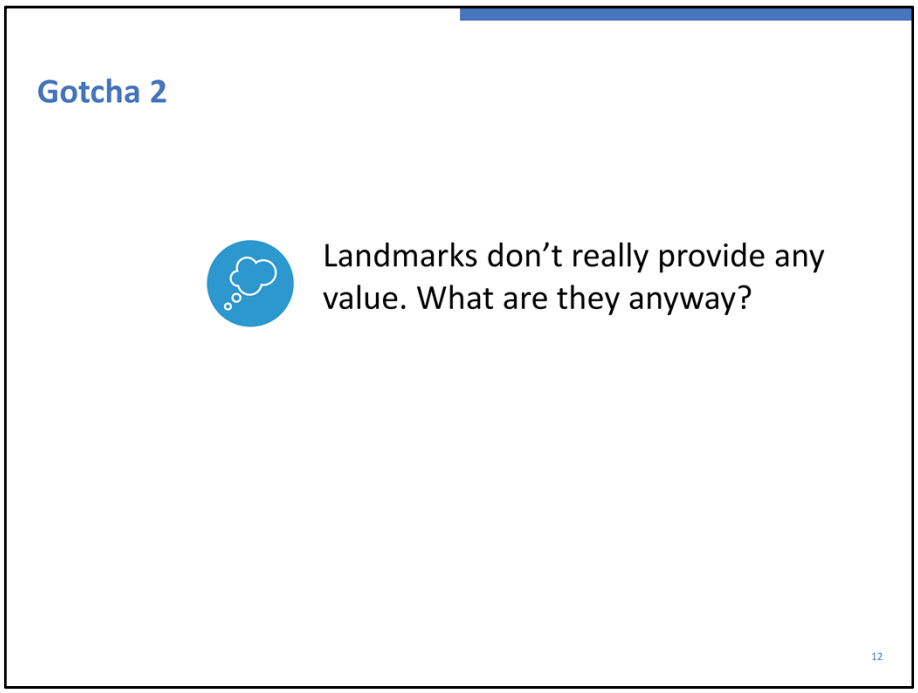
Make good use of Headings to convey structure of CONTENT (think an old-fashioned paper newspaper structure)

But don't skip heading levels, or if you do, do it consistently. And avoid using headings to convey system layout as that can cause additional work for AT users trying to navigate the application.

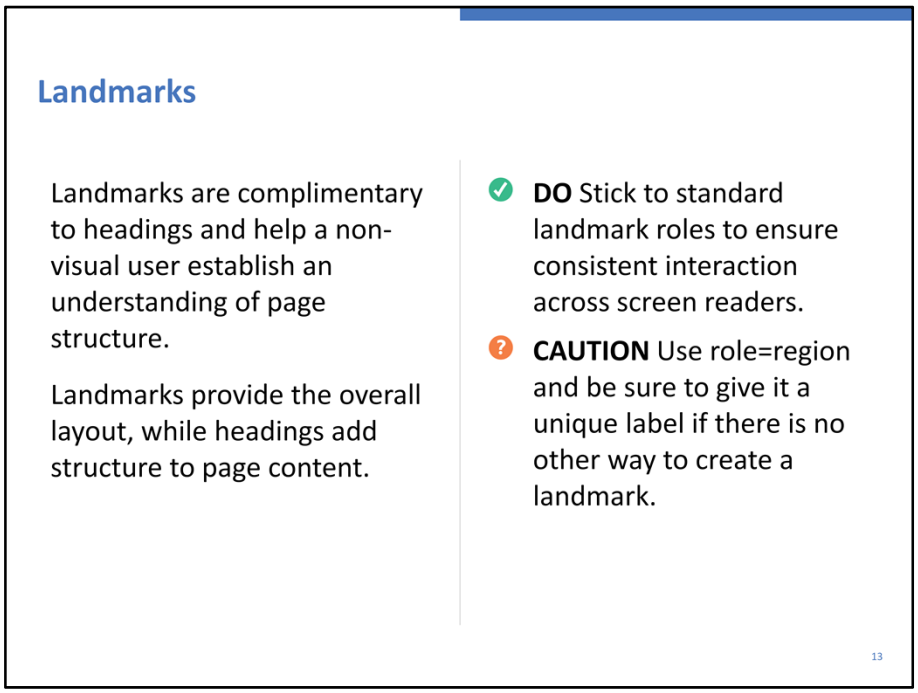
When tables are used to provide a visual structure of a page, screen reader users can become confused trying to figure out how content relates or try using table keyboard commands. Nested tables are some of the worst for screen reader users

as they can become “trapped” in the cells.

Using lists incorrectly or not all can be problematic for several types of users. Think about it from a cognitive perspective, your tool tells you that you are in a list but visually you just see text. ??? Confusing right.




But what about landmarks? The question I hear most often is “What are they used for and are they truly useful”?




Landmarks are an ARIA construct that allows developers to create semantically valid "regions" on a page that can help support screen reader navigation and provide structure within a page or application. They are complimentary to headings. Landmarks focus on the layout and types of information within the page. Headings provide structure to the content.

There are 8 roles in ARIA that are designated as "Landmarks".

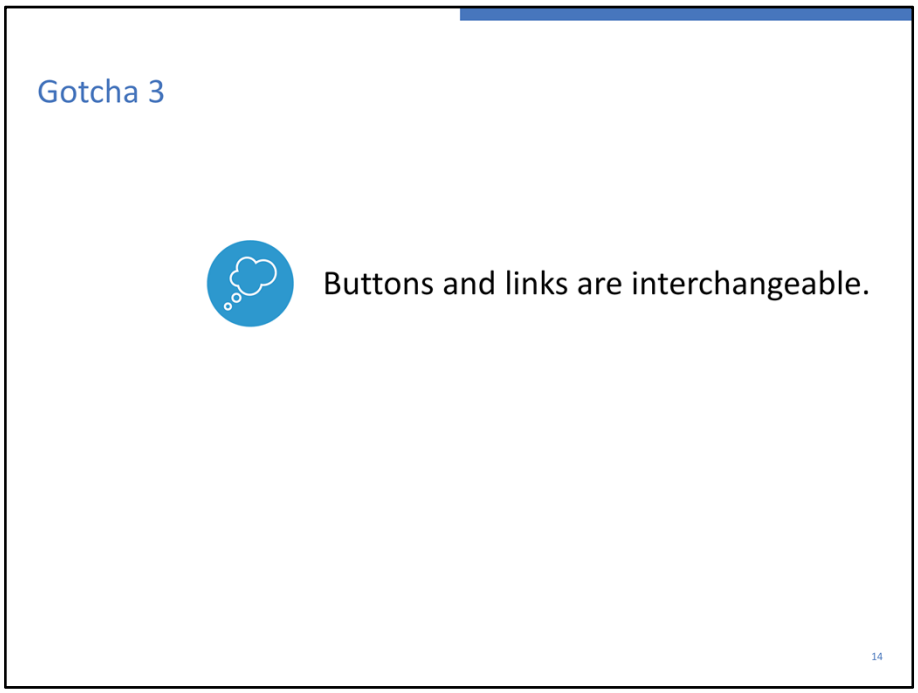
- Banner
- Main
- Navigation
- Contentinfo
- Complementary
- Form
- Search

There are also 6 default elements in HTML that automatically map to these landmarks – saving time. Those include.

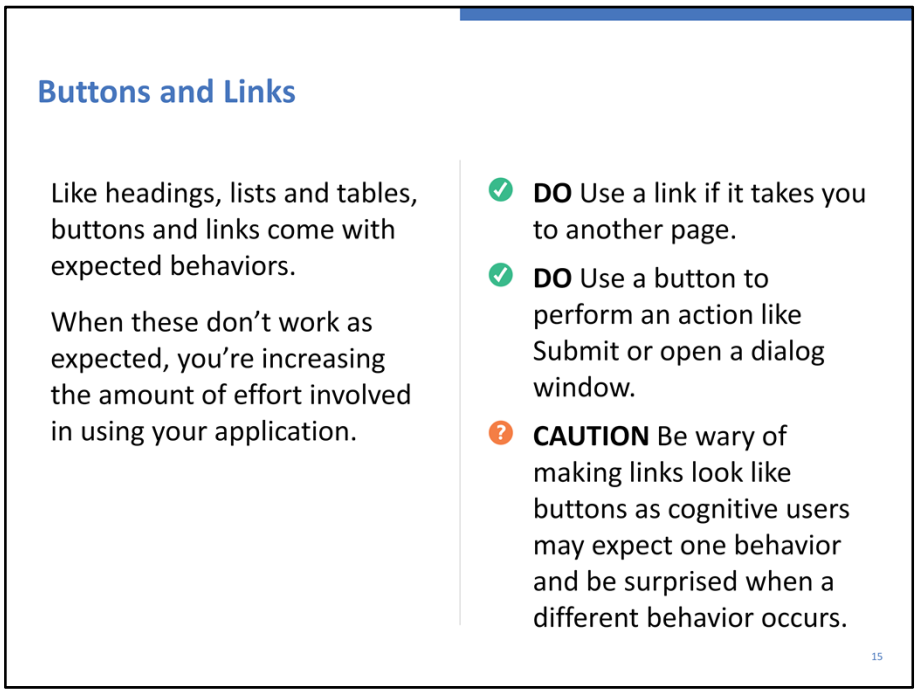
- Header (maps to banner)

- Main
- Nav
- Footer (maps to contentinfo)
- Aside
- Form

Landmarks can be applied to any generic element of custom control to expose accessibility information, but the most common misconception, is that you can create your own landmarks. Stick to the standards to ensure screen reader users can properly interact with your application. If you really need to create something not provided in the list above, use role=region and be sure to give it a unique label.




And the last of the “semantic” elements – buttons and links.
Strangely, there is a lot of confusion among developers about when something should be a link and when it should be a button.




Here's the simplest break down.

A button is used when you're executing some action on a page and NOT changing the context.

A link is used when you're changing context – like opening another page or layer in the application.

There are a couple of reasons this matters.

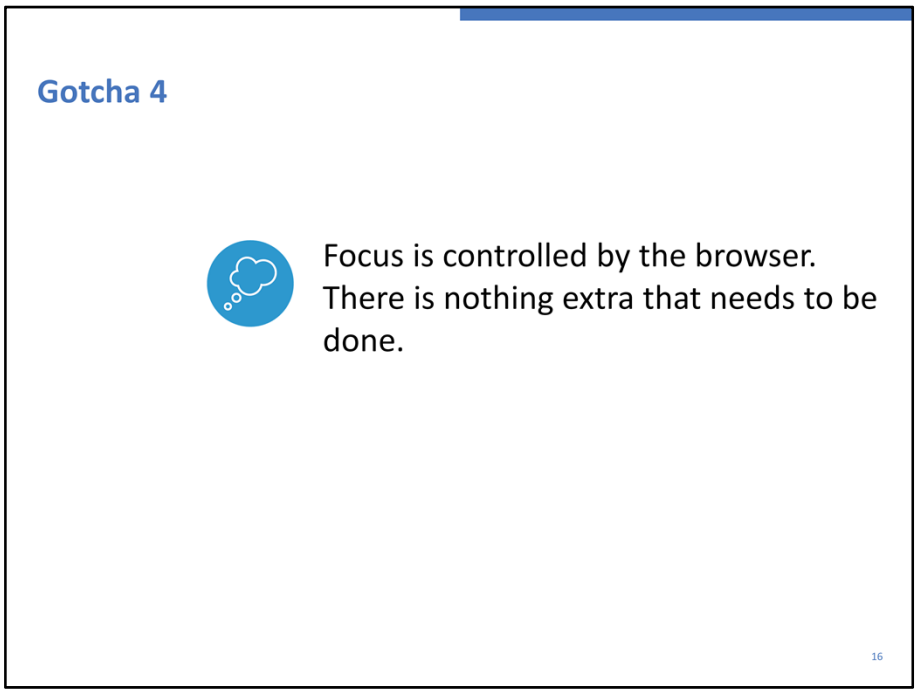
Non-visual users have developed mental models about what each of these elements do. Their screen readers will read out WHAT it is (a link or a button) and that allows them to make an assumption about what will happen. When a button navigates them to a new page, it can be jarring and confusing because it's not what's expected.

Sighted users not using a mouse SEE something that looks like a button or a link and they have a preconceived understanding of how to activate it.

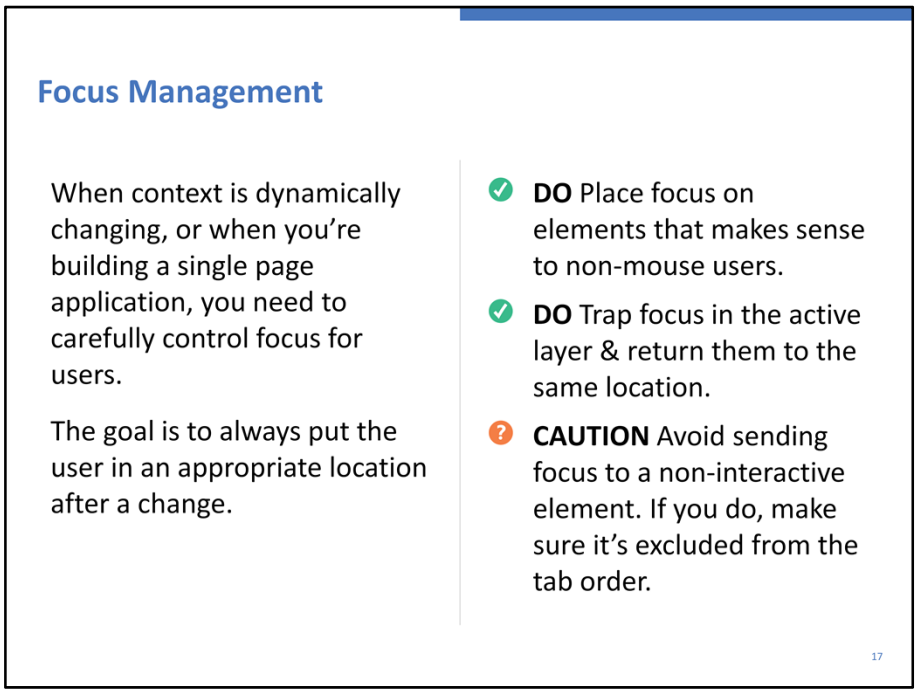
- Pressing SPACE or saying “Click BUTTON” to activate something that looks like a button.
- Pressing ENTER or saying “Click LINK” to activate something that looks like a link.

When these don’t work as expected, you’re increasing the amount of effort it takes people to use your application.

So, if the element takes the user to another context, make it look and work like a link. If it executes an action on a page (like saving data on a form) – make it look and work like a button. If you really need to break this rule, at least do it consistently across your application.




Let’s move now into Focus Management. One of the most common misunderstandings I continue to see is the assumption that the browser will handle the focus for you. It’s critical that focus is never dropped or unexpectedly moved. While this can sometimes happen by default when pages refresh, more often you need to programmatically control it.



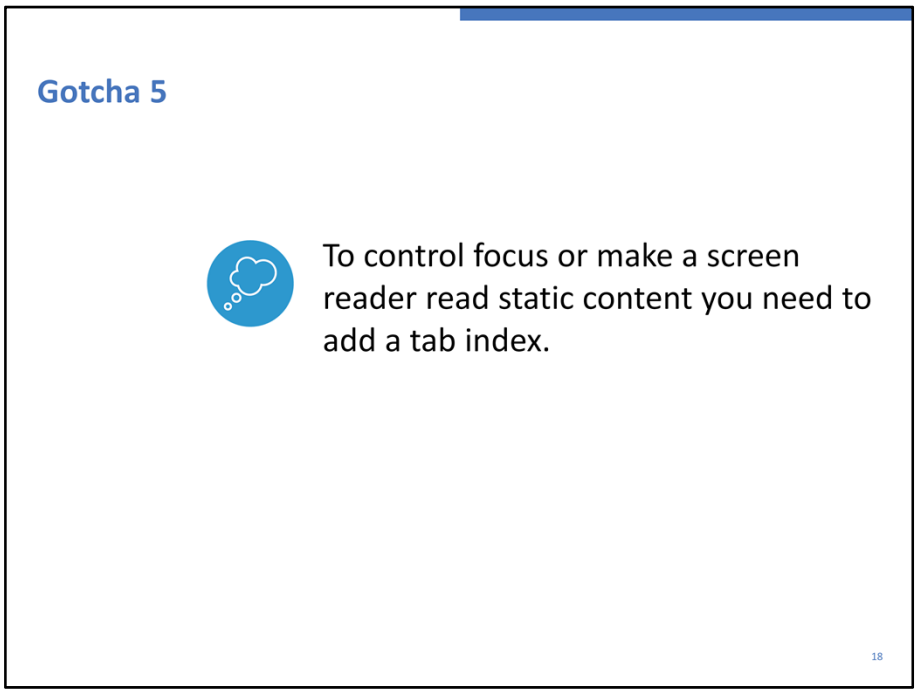

Controlling focus is the key to helping users understand where they are at all times. But controlling focus for screen reader users and keyboard only users can be a bit different. The goal is to always put the user in an appropriate place when the screen changes. For keyboard users, it's best to send them to the first interactive element in the new context when you're moving them between pages or layers.

But for screen reader users, they may need more context, as the first interactive element may come AFTER the page heading, or descriptive information they need to understand where they are.

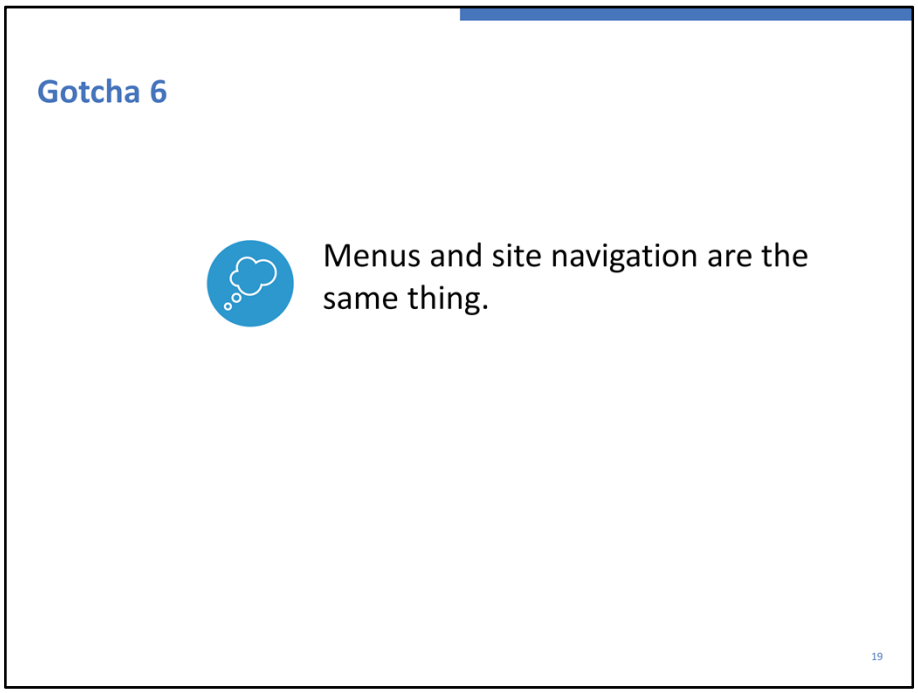
A general best practice is to send focus to the first heading or div in the new context to ensure non-visual users know where they are. But you also need to make sure this element doesn't end up in the tab order and create clutter for a keyboard only users. We'll talk more about how to do that in the next section.

One final, but critical element of controlling focus is returning the user to the right place in the previous context when they move "back". This is especially important when you're working with modal dialogs or layers in a single page application. You

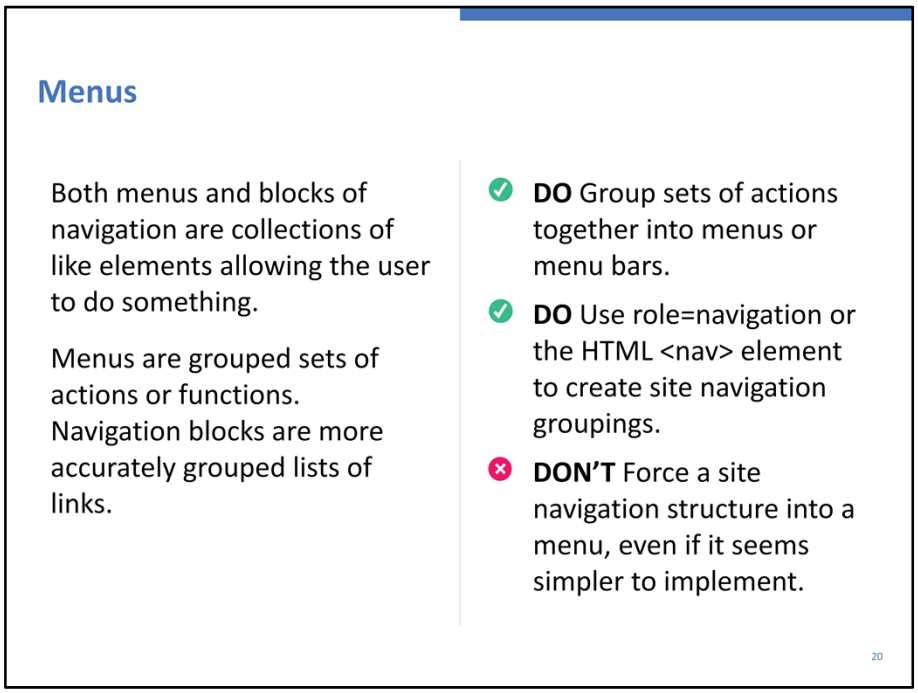
also need to ensure you're trapping focus in the active layer for the duration it's open. One of the most frustrating things is to lose your place by having focus drop back into the background layer and having no idea where you are or how to get back.




Ok. We just talked about controlling focus. One of the most common ways to set a focus target is with the tab index attribute. But it’s also something to be careful with.




The biggest gotcha with menus, is using them to build primary site or application navigation elements. These are technically not menus.

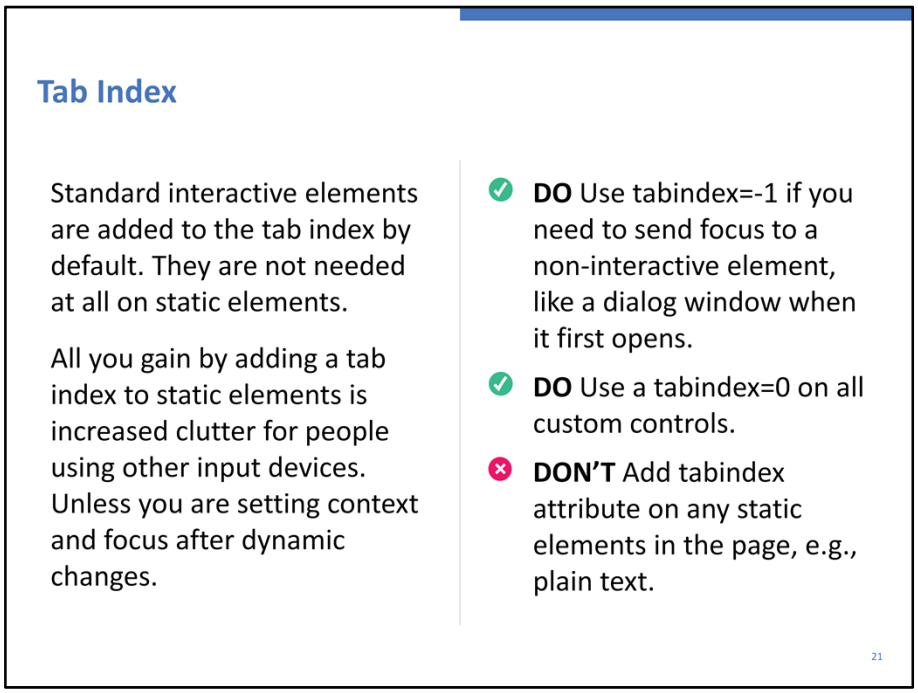



Menus are widgets that offer a list of choices to a user, such as a set of actions or functions. They contain "menu items" and can be contained in "menu bars" to create more robust constructs. Menu bars are generally persistent visible elements that are presented horizontally and intended to mimic the action menus in many desktop applications.

The mental models for interacting with menus have been around for a long time and are pretty well understood. Once the menu has focus, use the arrow keys to move around it. Use up/down to move within the menu items. In a menu bar, right and left cycle you to the next parent menu or a sub-menu of the item in focus if appropriate. Escape will close or exit the menu. Tab will move you to the next interactive element on the page – after the menu.

Navigation constructs are not technically "menus". The are more accurately organized groups of links. This distinction goes back to the differentiation between buttons and links – like buttons, menus are about taking action in the same context. Navigation structures are about just that, navigating to a new context.

Rather than forcing these elements into a menu construct, wrap them in a navigation landmark. Use ARIA roles and states to provide context and awareness around “sub groups” and allow the standard tab order to control the keyboard interactions.

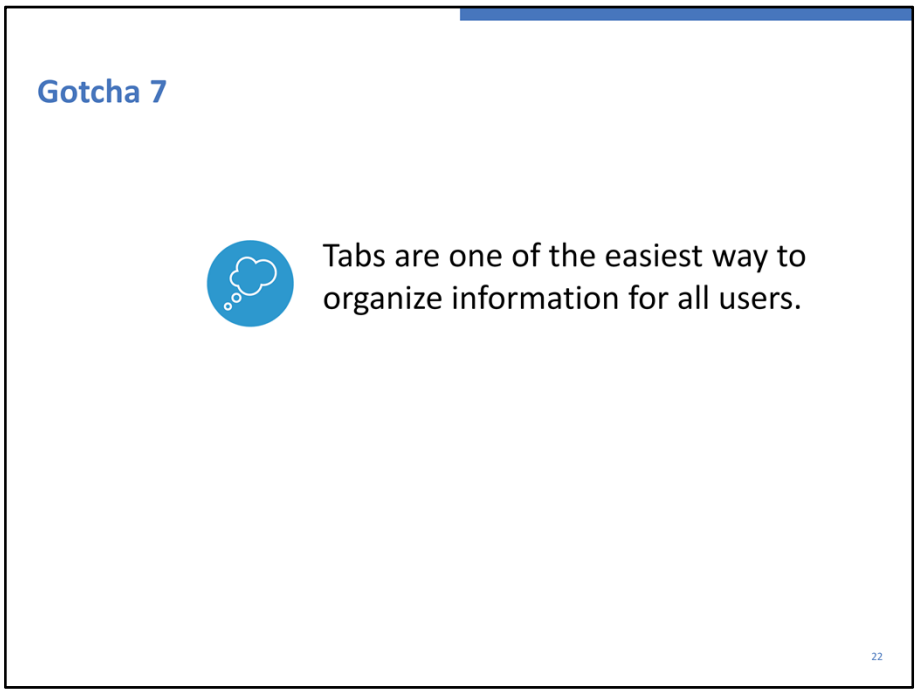



Standard interactive elements (like links, buttons, and menus) are automatically added into the tab order so you don't need to add a tab index attribute.

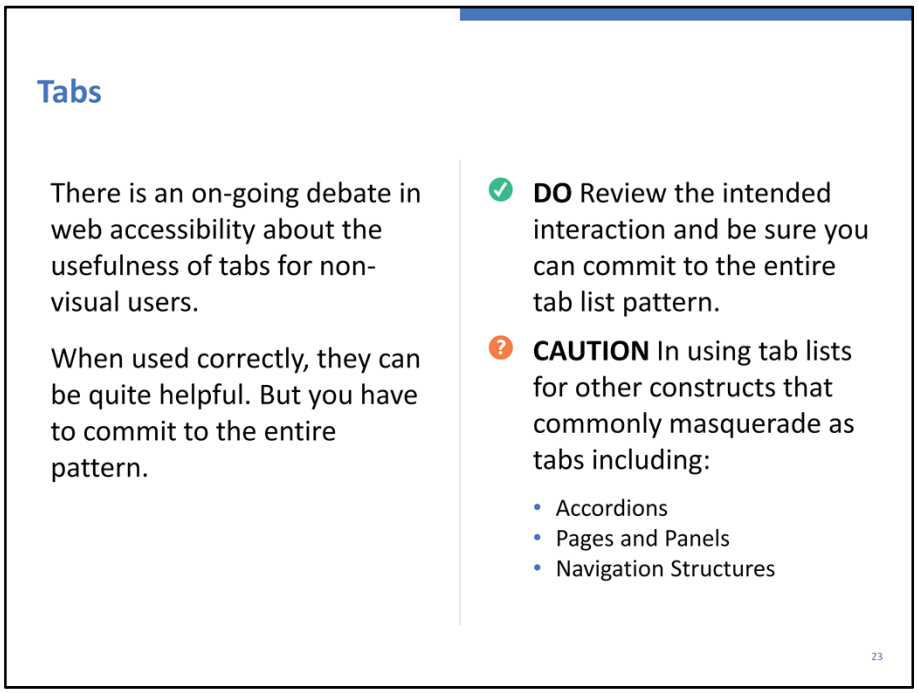
But, tab index can be added to any element. One of the most common misunderstandings is that you need to add tabindex=0 to non-interactive elements in order for screen reader users to be able to focus on it and read the text. That's not true.

Screen reader users have other ways of finding and reading text that's in proximity to other elements they can navigate to directly (like headings, tables, links, buttons, etc). All you gain by adding tabindex=0 to non-interactive elements is extra tab stops for sighted keyboard users, which quickly becomes very annoying.

If you're programmatically sending the user to a non-interactive element (which is a good practice to control focus and ensure an awareness of context) add tabindex=-1 rather than tabindex=0. This will allow focus to be sent there, providing context to screen reader users, but it will keep it OUT of the tab order and reduce the clutter for sighted keyboard users.




There is an on-going debate in web accessibility circles about whether or not tabs are a useful construct at all due to the complexity and confusion they often incur for screen reader users. But a lot of the confusion comes in when they are not used properly. So let's dig into that.




Tab lists and panels are used to organize information into separate sections within the same page.

If you have a design that seems to suggest the need for a tabs, review the full interaction model for tabs. If you can't commit to implementing the entire pattern, it's not a tab construct and you should explore other implementation options.

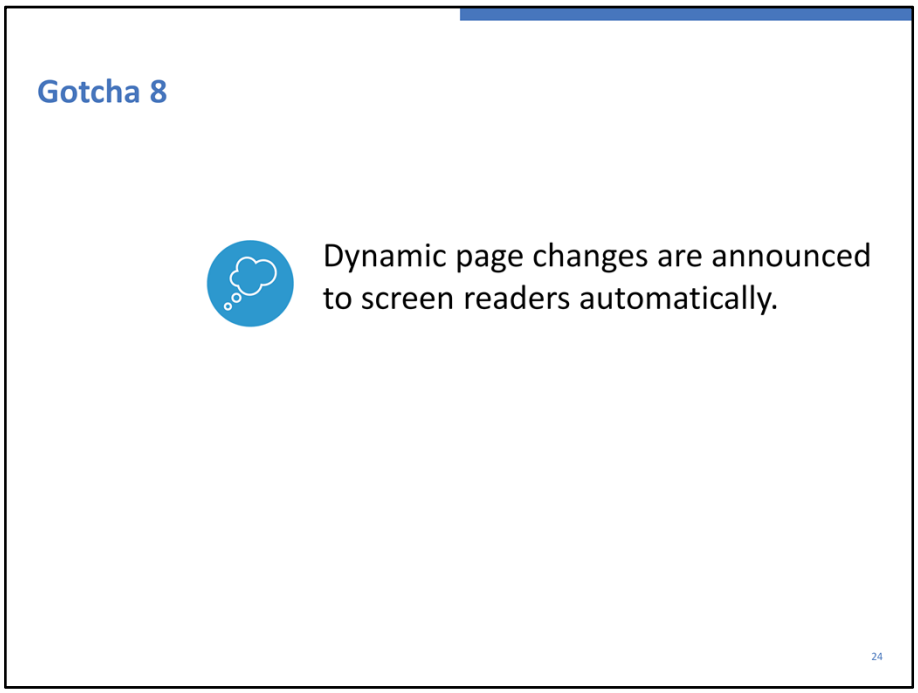
Some of the critical elements of tabs and tab lists include:

- A need for at least two tabs and tab panels
- One tab needs to be selected at all times
- Only content from selected tab can be available to screen reader
- Moving between tabs does not change the page, only some content within the page
- The tab list should be treated as a single interactive element
    - Arrow keys are used to move between the tabs.
    - Use SPACE key interaction to load a content within a tab
    - The next tab press on the keyboard should move to the next interactive element on the page, which may be within the tab, or further down the

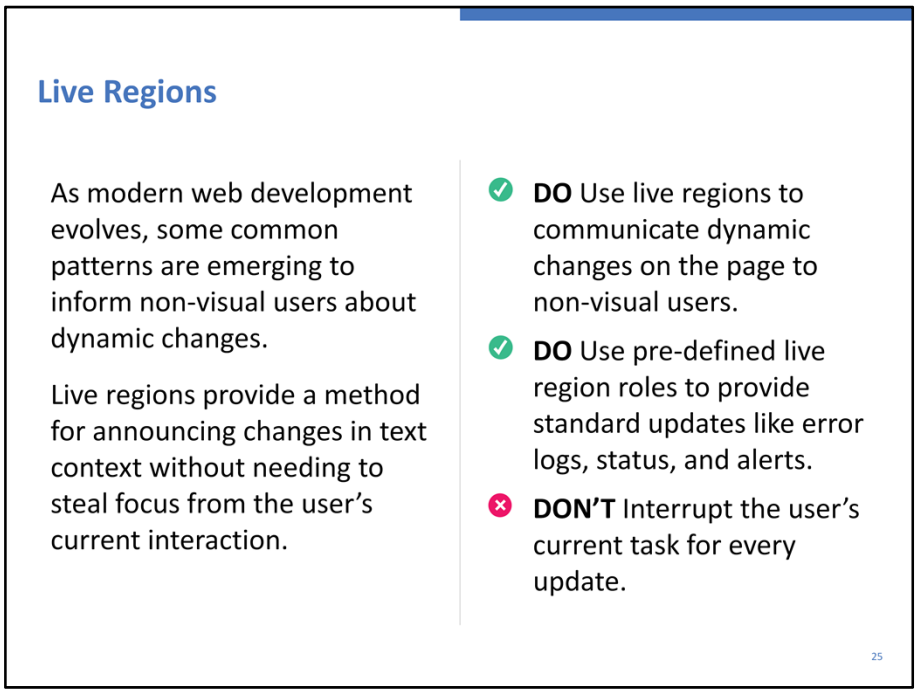
page.

If you can’t commit to using the entire pattern for tab lists, the alternatives you consider will be based on design objectives. Some options might include:

- Accordions
- Pages and panels
- Navigation elements




The idea of changing content dynamically on a web page or within an application has been around so long it's become the standard. The challenge is, assistive technology – like screen readers – aren't able to detect when these dynamic changes are made. They need to be programmatically told.




Which brings us to live regions

Live regions are an HTML container that screen readers can subscribe to. They provide a method for announcing changes in text context without needing to steal focus from the user's current interaction. The simplest use of live regions is simply to assign a polite or assertive attribute to the content being sent to the container.

When aria-live=polite the screen reader will wait for a pause in the user's interaction before announcing the content in the live region. This is the most common, and least intrusive way of alerting a user to dynamic changes in content within a page or application.

If you cannot wait for the user to pause, you can use aria-live=assertive which will interrupt the user's current interaction and read the content of the live region immediately. This method is most useful for errors or important system alerts. It should be used sparingly.

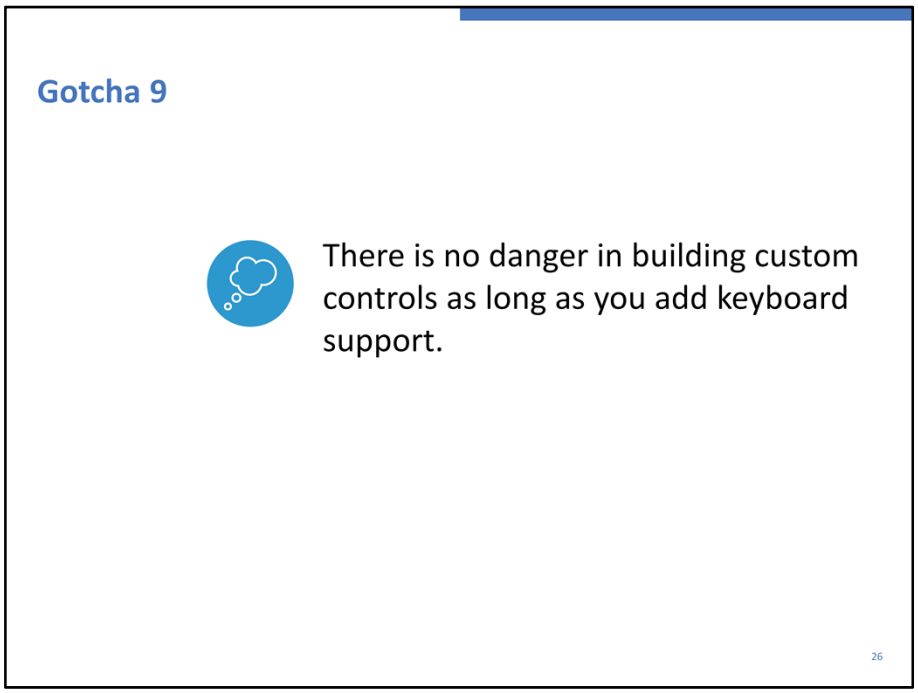
As modern web development evolves, some common patterns are emerging around

when and how to inform non-visual users about dynamic content changes. For these use cases, some pre-defined roles and behaviors are being defined for live regions.

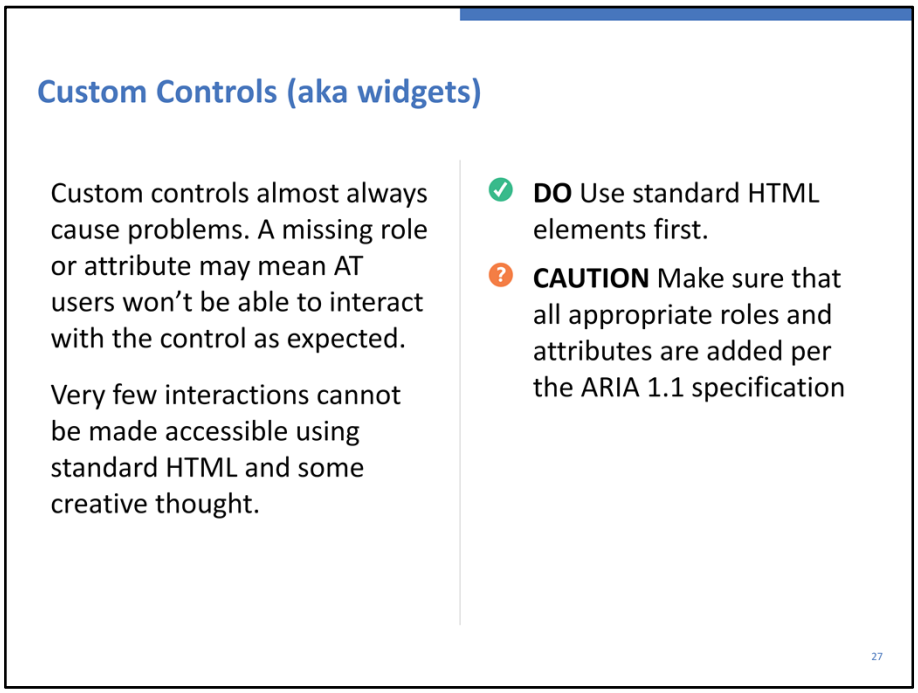
A few of the most common include:

- Log: used to inform the user of logged content such as chat message, non-critical errors, game interactions, etc.
- Status: used to present information in a persistent status bar or regular updates on the page.
- Alert: most commonly used for serious error messages.

Live regions can be incredibly useful, but you should be thoughtful in their use so as not to overwhelm a non-visual user with information they may not need. When they are used too often in a short period of time, they have a tendency to confuse the screen readers.




When designers give developers complex and challenging interactions, it can seem simpler to just go and build custom widgets to handle them. But this can cause serious problems for screen reader users, and often keyboard users as well if every scenario is not considered.




Enter "Application Mode".

The intent of application mode is to allow the web tool to take control of the keyboard interactions & force them to behave in a way more inline with desktop applications.

Application mode is invoked automatically when a user focuses on a form element in a web page, but this change in context is tightly controlled by the screen reader and expected by non-visual users to enable them to complete a form.

Many developers are introduced to role=application when heading down the path of custom widget development. Theoretically, this can be quite useful when you have a custom construct that you're creating a non-standard keyboard interaction for. But this is a VERY advanced technique that should be used with considerable caution.

Application mode steals control of the keyboard and prevents a screen reader user from using their standard interactions within their screen reader. When not handled

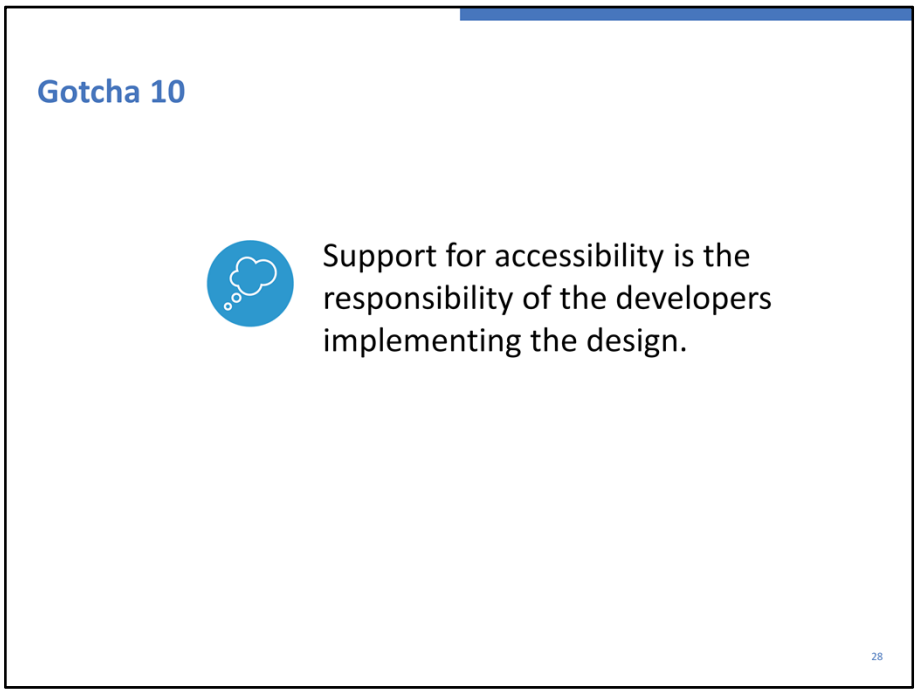
carefully the constantly changing context can cause enough confusion for the screen reader as to render your tool completely unusable.

There are very view interactions that cannot be made accessible using standard HTML elements and ARIA roles and attributes. With some creative thought and combination of elements it's possible to build a very complicated web application without ever using role=application.

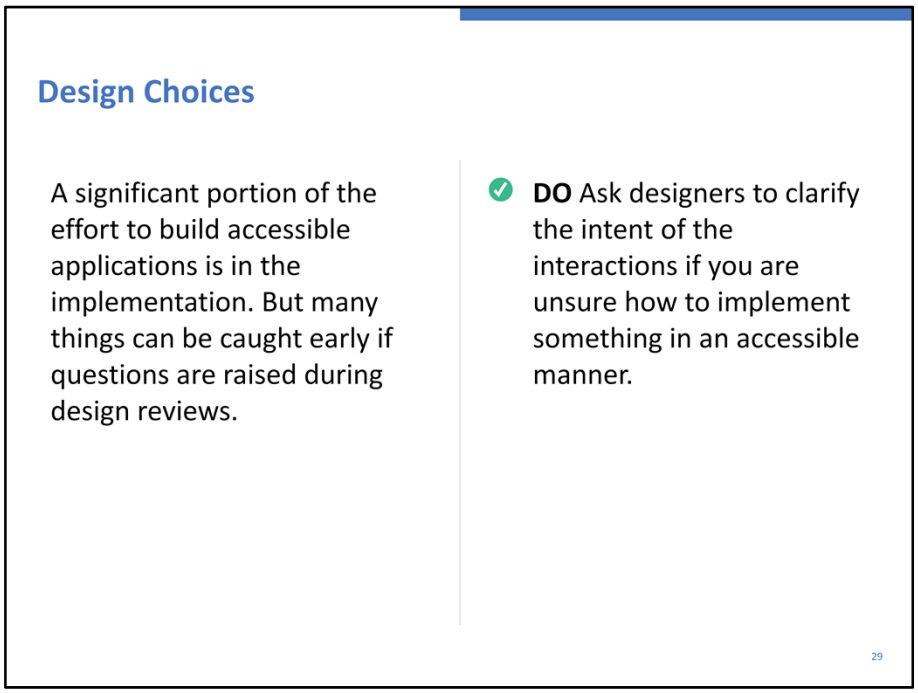
However, if you've attempted all other alternatives and really need to use application mode to achieve your goal there are a few key things to consider:

- You can add application mode on any element in the page by adding role=application to the element. But then you need to define all potential keyboard interactions.
- If you have multiple custom constructs, every instance of role=application will be listed in the page landmarks, but they cannot be distinguished from one another and often create serious confusion and conflicting commands for the screen readers
- You should never put role=application on the body of a page unless the entire page is a custom widget that does not use ANY standard HTML elements.

Application mode is very challenging to implement correctly as you need to consider EVERY keyboard interaction and build a custom navigation pattern that considers every possibility.




And last but not least, many design and development teams still believe that accessibility is an implementation challenge only.




While a significant portion of building accessible applications is in the implementation, many things can be caught early if questions are raised in design reviews.

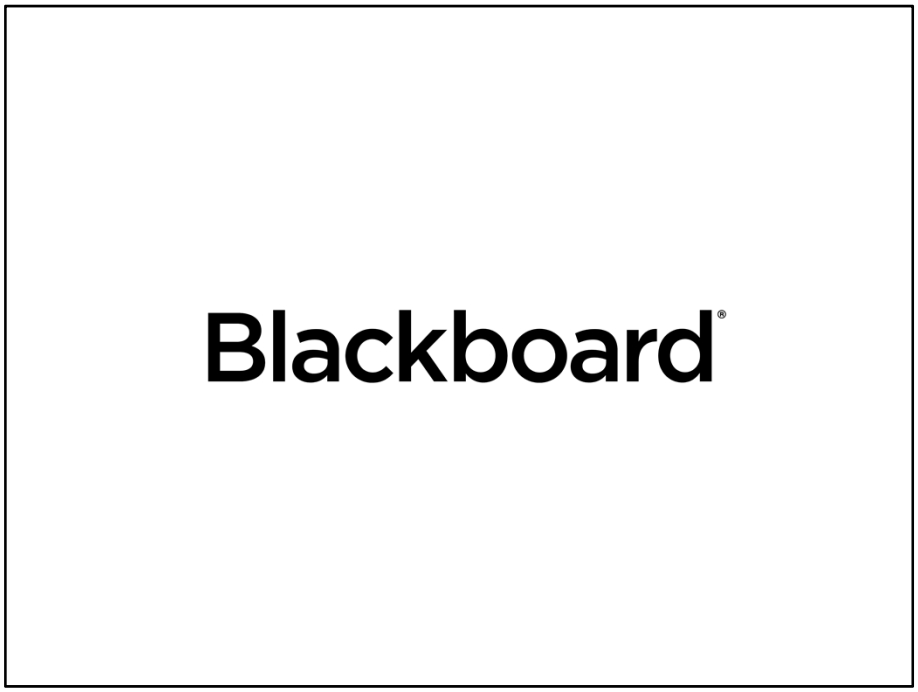
Some of the most common challenges that can be caught before implementation begins are

- Color contrast issues
- Forms without labelled fields
- Text inputs without borders
- Images without alternative text
- Links that look like buttons and vice versa
- Reading order, tab order and keyboard expectations

Developers should feel empowered to question designs that include these common accessibility mistakes.

Always think through how you would need to implement a proposed design and ask for clarification of the intent if you're unsure of how to make it work for a keyboard

or screen reader user.


Blackboard®