WEB DEVELOPMENT

Web Accessibility

Semantic HTML & ARIA · Keyboard Navigation

Forms & Errors · Color & Contrast

Screen Readers · Accessible Components

Capstone: Auditing an Inaccessible Page

Philip Osztromok

Generated with Claude

Table of Contents

Web Accessibility (a11y) — 9 Chapters

♿ Why Accessibility Matters & The Legal/Ethical Landscape

Accessibility isn't a "nice to have" layered on afterward — roughly 1 in 6 people worldwide has some form of disability. Excluding them from a product's usability is both an ethical failure and, increasingly, a real legal liability. This chapter sets the foundation everything else in the course builds on.

WCAG: The Web Content Accessibility Guidelines

WCAG is the standard everything in this course maps back to — currently at version 2.1/2.2, organized into three conformance levels:

Level	What It Means
A	Minimum — a baseline, rarely sufficient on its own
AA	The common legal/practical target — most laws and standards reference this level
AAA	Highest — often impractical to apply site-wide, used selectively

The POUR Principles

WCAG organizes around four pillars — every technique in this course maps to one of these:

Perceivable

Content must be presentable in ways every user can perceive — alt text for images, captions for video, sufficient color contrast.

Operable

Interface components must be usable by everyone — full keyboard support, no interactions that require precise timing or fine motor control.

Understandable

Content and operation must be comprehensible — clear labels, predictable navigation, helpful error messages.

Robust

Content must work reliably across current and future tools — valid, semantic markup that assistive technology can correctly interpret.

⚖️ The Legal Landscape

Standard	Scope
ADA (Americans with Disabilities Act)	Applied to websites via US court precedent — the source of most inaccessible-website lawsuits
Section 508	Required for US federal agencies and their contractors
EN 301 549	The EU's accessibility standard, referencing WCAG directly

Lawsuits over inaccessible websites are a real, growing legal risk — not a hypothetical concern reserved for large enterprises.

Who These Techniques Actually Serve

Grounding the "why" behind specific techniques covered later: visual disabilities (blind, low-vision, color-blind) motivate screen reader support and contrast requirements; motor/mobility disabilities motivate full keyboard support; auditory disabilities motivate captions; cognitive disabilities motivate clear, predictable, jargon-free interfaces.

What Inaccessibility Actually Feels Like

Inaccessible

A screen reader user hits an unlabeled button and hears only "button" — no idea what it does. A keyboard user tabs into a dropdown and can't tab back out.

Accessible

Every control has a clear, spoken label. Every interactive element can be reached, operated, and exited using only a keyboard.

🛠️ Tooling Overview

Automated Scanners

axe DevTools (browser extension) and **Lighthouse's** accessibility audit both flag common, mechanically-detectable issues quickly.

Manual Testing

Keyboard-only navigation and real screen reader spot-checks (Chapter 7) — irreplaceable for the issues automated tools structurally can't catch.

💻 Coding Challenges

Challenge 1: Map Techniques to POUR

For each of (a) adding alt text to images, (b) ensuring a modal can be closed with the Escape key, and (c) writing clear, jargon-free error messages, identify which POUR principle it primarily addresses.

Goal: Practice connecting concrete techniques back to the four pillars they serve.

→ Solution

Challenge 2: Identify the Affected User Group

A site relies entirely on color (red vs green) to indicate form field errors, with no icon or text label. Identify which disability category this most directly excludes, and explain why.

Goal: Practice reasoning from a specific design choice back to who it actually harms.

→ Solution

Challenge 3: Choose a Conformance Target

A company is setting an internal accessibility standard for its public-facing website. Recommend WCAG level A, AA, or AAA as the target, and justify the choice using this chapter's conformance-level table.

Goal: Practice choosing a realistic, defensible standard rather than defaulting to the "best" one.

→ Solution

⚠️ Gotcha: Trusting Automated Scans Alone

Automated tools like axe and Lighthouse typically catch only around 30–40% of real accessibility issues — problems like confusing focus order, unclear link text ("click here"), or an illogical reading order require actual human judgment to catch, since a scanner has no way to evaluate whether something genuinely makes sense to a real user. Passing an automated scan with zero flagged issues is not the same as a page actually being accessible — treating a clean automated report as "done" is one of the most common, and most damaging, false senses of completion in this field.

🎯 What's Next

With the landscape and principles established, the next chapter goes hands-on: **Semantic HTML as the Foundation** — why div-soup breaks accessibility, and the semantic elements that fix it.

Semantic HTML as the Foundation

Chapter 1 established why accessibility matters. This chapter is the first concrete technique — and the foundation everything else builds on: using the right HTML element for the job.

Why Div-Soup Breaks Accessibility

A `<div>` carries no semantic meaning at all — to assistive technology, it's just a generic container, nothing more:

A "Button" Made of a Div

`<div onclick="...">`

No role announced to a screen reader, not in the tab order by default, no Enter/Space activation — sighted mouse users see a button; everyone else gets nothing.

`<button>`

Correct role, correct tab order, correct Enter/Space activation, correct focus styling — all for free, out of the box.

The browser and assistive technology get zero information about what an element *is* or *does* beyond its visual appearance — which sighted users take entirely for granted.

What Native Elements Give You for Free

```
<!-- Reimplementing a link with JS breaks things silently -->
<div onclick="location.href='/page'">Go to page</div>
<!-- No right-click "open in new tab," no Ctrl+click, no middle-click -->
<!-- The native element already does all of this -->
<a href="/page">Go to page</a>
```

This is the reason native semantics beat reimplementing behavior with JavaScript — a point Chapter 3 returns to directly for ARIA.

🗺️ Sectioning & Landmark Elements

Element	Landmark Role
`<header>`	banner
`<nav>`	navigation
`<main>`	main
`<aside>`	complementary
`<footer>`	contentinfo

Screen reader users can pull up a list of landmarks and jump straight to "main content," skipping navigation entirely — a genuinely powerful shortcut sighted users don't even notice they have. An all-`<div>` page forces linear listening through everything, with no way to skip around at all.

Heading Hierarchy

`h1`–`h6` form an outline of the page's actual content structure. Screen reader users frequently navigate *by heading level* — jumping to the next `h2`, for instance — which only works if headings are used hierarchically, not skipped or chosen for their default visual size.

Buttons vs Links: A Real Distinction

`<button>` is for actions that don't navigate — opening a modal, submitting a form, toggling a menu. `<a href>` is for navigation — a different page, or a different section via a `#` anchor. Using the wrong one confuses a screen reader user about what will actually happen on activation — directly undermining Chapter 1's Understandable principle.

💻 Coding Challenges

Challenge 1: Fix a Div-Based Button

Rewrite `<div class="btn" onclick="submitForm()">Submit</div>` using the correct native element, preserving the same visual class.

Goal: Practice the most fundamental semantic-HTML fix in this chapter.

→ Solution

Challenge 2: Add Landmark Structure to a Div-Only Page

Given a page built entirely from generic `<div>`s (one for the top nav, one for the main content, one for a footer), rewrite it using the correct landmark elements.

Goal: Practice replacing div-soup with the elements that create real, jumpable landmarks.

→ Solution

Challenge 3: Choose Button vs Link

For each of (a) a "Log out" control that clears a session and redirects to the homepage, and (b) a "Read more" control that expands a hidden paragraph in place without navigating anywhere, choose `<button>` or `<a href>` and justify each.

Goal: Practice applying the navigates-vs-acts distinction to realistic, slightly ambiguous cases.

→ Solution

⚠️ Gotcha: Picking a Heading Level for Its Visual Size

It's tempting to reach for `<h3>` somewhere simply because its default font size "looks right" for a given piece of text — skipping `<h2>` entirely in the process. This breaks heading-based screen reader navigation: jumping from an `h1` straight to an `h3` suggests to someone navigating by heading level that an entire section is missing. Choose the heading level based on the content's actual position in the document's hierarchy, then adjust its visual size with CSS separately — never the reverse.

🎯 What's Next

With the semantic foundation in place, the next chapter covers what to do when native HTML genuinely isn't enough: **ARIA: When and When Not to Use It** — the first rule of ARIA, and why misusing it can be worse than no ARIA at all.

🏷️ ARIA: When and When Not to Use It

Chapter 2 covered native semantic HTML. This chapter covers ARIA — Accessible Rich Internet Applications — the toolkit for filling genuine gaps native HTML can't cover, and the crucial discipline of not reaching for it first.

The First Rule of ARIA

No ARIA is better than bad ARIA. If a native HTML element already provides the semantics and behavior you need, use that instead of recreating it with a `<div>` plus ARIA attributes. Critically: **ARIA only announces information** — it provides none of the automatic keyboard behavior Chapter 2 covered. Adding `role="button"` to a `<div>` does not make it keyboard-operable; nothing about tab order or Enter/Space activation happens automatically just because a role was declared.

Three Categories of ARIA

Category	What It Describes	Example
Roles	What an element IS	`role="dialog"`, `role="tablist"`
States	Current condition, can change	`aria-expanded`, `aria-checked`
Properties	Relatively static characteristics	`aria-label`, `aria-describedby`

✅ Safe, High-Value ARIA

Icon-Only Button

```
<button aria-label="Close dialog">✕</button>


<!-- Announces dynamic updates without moving focus -->
<div aria-live="polite">3 items added to cart</div>

<!-- Communicates open/closed state on a custom disclosure toggle -->
<button aria-expanded="false">Show details</button>
```

`aria-label` for Icon Buttons

A button with only an icon and no visible text has no accessible name at all without something providing one — `aria-label` fills exactly that gap.

`aria-live` Regions

Lets a screen reader announce a content change — a chat message count, a toast — without requiring the user's focus to move there first.

⚠️ The Danger of ARIA Misuse

Adding `role="button"` to a `<div>` without also adding `tabindex="0"` and keydown handlers doesn't just fail to help — it actively makes things *worse* than no ARIA at all:

Announced as a Button, But Not One

`role="button"` Alone

A screen reader announces "button," creating an expectation of tab-focus and Enter/Space activation — neither of which actually works. Worse than saying nothing.

Just Use `<button>`

Chapter 2's own point, restated: the native element gives you everything correctly, with far less code and no way to get it half-wrong.

Redundant and Conflicting ARIA

`<button role="button">` is pointless — the native element already has that role. `<a href="#" role="button">` is actively harmful — the browser and assistive technology now receive contradictory signals about whether this is a link or a button.

💻 Coding Challenges

Challenge 1: Add an Accessible Name to an Icon Button

Given `<button>🔍</button>` used as a search trigger with no visible text, add the correct ARIA attribute so a screen reader announces its purpose.

Goal: Practice the most common, safest real-world use of ARIA.

→ Solution

Challenge 2: Fix a Broken Custom Toggle

A custom accordion toggle is written as `<div role="button" aria-expanded="false">Show more</div>` with a click handler but no keyboard support. Identify what's missing and fix it, or recommend a simpler alternative.

Goal: Practice recognizing and repairing the exact ARIA-without-behavior gap this chapter warns about.

→ Solution

Challenge 3: Identify Redundant or Conflicting ARIA

For each of (a) `<nav role="navigation">`, (b) `<a href="/settings" role="button">`, and (c) `<div role="tablist">`, state whether the ARIA is redundant, conflicting, or genuinely necessary, and why.

Goal: Practice distinguishing pointless, harmful, and correct ARIA usage.

→ Solution

⚠️ Gotcha: Reaching for ARIA First

The real skill this chapter teaches isn't memorizing ARIA attributes — it's knowing when *not* to reach for them. Before adding any ARIA role, state, or property, ask whether a native element already provides it: a custom "button" is almost always better as `<button>`; a custom "link" is almost always better as `<a href>`. ARIA is the correct, necessary tool specifically when no native element covers the pattern at all — there's no native tab interface, so `role="tablist"`/`role="tab"`/`role="tabpanel"` genuinely earn their place there (Chapter 8 builds exactly this). Reach for ARIA last, not first.

🎯 What's Next

With markup and ARIA covered, the next chapter goes hands-on with the interaction model that matters most for motor disabilities: **Keyboard Navigation** — tab order, focus management, and correctly trapping focus in modals.

⌨️ Keyboard Navigation

Chapters 2 and 3 established that native elements give keyboard behavior for free, and ARIA never provides it on its own. This chapter is about making that keyboard behavior actually work correctly across a whole page — the primary interaction model for motor and mobility disabilities from Chapter 1.

Tab Order Follows DOM Source Order

The default tab order follows **DOM source order**, not visual or CSS position. Flexbox `order`, absolute positioning, or grid placement can visually reposition an element without changing where it sits in the tab sequence — creating a confusing mismatch between what a keyboard user sees and where focus actually jumps.

`tabindex` Value	Effect
`0`	Adds the element to the natural tab order, at its actual DOM position
`-1`	Removes it from the tab order, but keeps it programmatically focusable via JS
Positive (`1`, `2`…)	Explicitly reorders the tab sequence — almost always an anti-pattern; avoid

👁 Focus Visibility

Browsers historically applied `:focus` styling to both keyboard and mouse interaction — leading many sites to strip it entirely with `outline: none` to "clean up" the look for mouse users. `:focus-visible` is the correct modern fix: it shows a focus indicator specifically for keyboard navigation, without applying it on a mouse click.

Removing Focus vs Styling It Correctly

`outline: none`

Clean for mouse users — and catastrophic for keyboard users, who now have zero visual indication of where focus currently is.

`:focus-visible`

Keyboard users get a clear focus indicator; mouse users never see it — the actual correct fix, not a removal.

Skip Links

A "Skip to Main Content" Link

```
<a href="#main-content" class="skip-link">Skip to main content</a>

/* Visually hidden until it receives focus */
.skip-link { position: absolute; left: -9999px; }
.skip-link:focus { left: 1rem; top: 1rem; }
```

A skip link is the very first focusable element on the page, letting keyboard users bypass repeated navigation without tabbing through it every single page. It's the keyboard-only user's equivalent of Chapter 2's landmark shortcut — someone who doesn't use a screen reader and can't jump by landmark experiences the exact same "re-tabbing through nav every time" problem otherwise.

🪟 Focus Management in Modals

A correct modal does three things, and missing any one creates a genuinely disorienting experience:

1. Move Focus In

On open, focus moves to the first focusable element or the modal's own heading — not left sitting on whatever triggered it.

2. Trap Focus

Tab and Shift+Tab cycle only among the modal's own focusable elements while it's open — never escaping to the (hidden) page behind it.

3. Restore Focus

On close, focus returns to whatever element originally opened the modal — never left "lost," resetting to the top of the document.

```
function openModal(modal, triggerButton) {
 const focusables = modal.querySelectorAll("button, a, input, [tabindex]");
 focusables[0].focus();  // Step 1 — move focus in

 modal.addEventListener("keydown", (e) => {
  if (e.key !== "Tab") return;
  const first = focusables[0], last = focusables[focusables.length - 1];
  if (e.shiftKey && document.activeElement === first) { e.preventDefault(); last.focus(); }
  else if (!e.shiftKey && document.activeElement === last) { e.preventDefault(); first.focus(); }
 });  // Step 2 — trap focus

 modal.dataset.trigger = triggerButton.id;  // remembered for Step 3 — restore focus on close
}
```

Custom Interactive Components Need Full Keyboard Support

Reinforcing Chapter 3's tablist example: a custom dropdown, combobox, or tabs component built without native elements needs someone to deliberately implement Tab, Enter, Space, Arrow keys, and Escape — matching whatever the ARIA Authoring Practices pattern specifies for that component. Chapter 8 builds several of these fully.

💻 Coding Challenges

Challenge 1: Diagnose a Tab-Order Mismatch

A form's fields are visually ordered Name, Email, Phone using CSS Grid with explicit `order` values, but the underlying HTML source order is Phone, Name, Email. Explain what a keyboard user experiences, and the fix.

Goal: Practice recognizing the DOM-order-vs-visual-order mismatch concretely.

→ Solution

Challenge 2: Write a Skip Link

Write the HTML and CSS for a skip link that's invisible until focused, jumps to a `<main id="content">` element, and appears in the top-left corner when focused.

Goal: Practice implementing the visually-hidden-until-focused pattern correctly.

→ Solution

Challenge 3: Identify a Missing Modal Step

A modal correctly moves focus in on open and traps focus while open, but on close, focus simply disappears (nothing is focused at all). Identify which of the three steps is missing and explain the user impact.

Goal: Practice diagnosing a partial, incomplete focus-management implementation.

→ Solution

⚠️ Gotcha: `outline: none` With No Replacement

This is one of the single most common and most damaging accessibility mistakes on the web — removing the default focus outline purely for visual cleanliness, with nothing put in its place. It takes an otherwise perfectly keyboard-operable page and makes it completely unusable for keyboard users, who now have no way to see where they currently are as they tab through it. The fix is never to remove focus styling — it's to replace it correctly with `:focus-visible`, exactly as this chapter's earlier section describes. If a codebase has `outline: none` anywhere without an accompanying focus style, that's a real, high-priority bug, not a minor visual nitpick.

🎯 What's Next

With navigation and focus covered, the next chapter turns to one of the most common places accessibility breaks down in practice: **Forms & Accessible Error Handling** — label association, accessible error messages, and the placeholder-as-label anti-pattern.

📝 Forms & Accessible Error Handling

Chapter 4 covered keyboard navigation generally. Forms are where accessibility most commonly breaks down in real-world sites — this chapter covers the specific, well-established patterns that fix it.

Label/Input Association Is Load-Bearing

```
<label for="email">Email address</label>
<input id="email" type="email">
```

A matched `<label for>`/`id` pair isn't a nice-to-have — clicking the label text focuses the input (a larger click target, helping motor disabilities too), and a screen reader announces the label the moment the input receives focus. Without it, an input is announced as just "edit text," with no indication of what it's for.

🚫 The Placeholder-as-Label Anti-Pattern

Placeholder Alone vs a Real Label

> **Placeholder Only**
>
> Disappears the moment typing starts (losing the field's purpose exactly when reviewing it matters most), typically lower contrast by default, and unreliably announced — or not announced at all — by screen readers.

> **Real `<label>`**
>
> Always visible, always announced, always contrast-compliant text — works correctly for everyone, no exceptions.

If a visually minimal design is genuinely wanted, use a visually-hidden label class — never remove the label element itself.

Fieldset/Legend for Grouped Controls

A Labeled Radio Group

```
<fieldset>
 <legend>Preferred contact method</legend>
 <label><input type="radio" name="contact" value="email"> Email</label>
 <label><input type="radio" name="contact" value="phone"> Phone</label>
</fieldset>
```

A screen reader announces the `<legend>` as the user navigates into the group — "preferred contact method" — before hearing each individual option, essential context that a plain nearby visual heading with no programmatic connection would never provide.

❌ Accessible Error Handling

An Invalid Field, Correctly Wired

```
<label for="email">Email address</label>
<input id="email" type="email"
aria-invalid="true" aria-describedby="email-error">
<span id="email-error">Enter a valid email address, like name@example.com</span>
```

`aria-invalid`

Announces the invalid state itself to a screen reader — not just a visual red border sighted users notice.

`aria-describedby`

Programmatically links the field to its specific error text, so the message is announced whenever the field is reached — not just displayed nearby with no connection.

Error messages should be specific and actionable, not just "invalid input" — Chapter 1's Understandable principle applies directly here. On a failed submission, move focus to the first invalid field (or an error summary), rather than leaving the user unaware anything went wrong — a direct extension of Chapter 4's focus-management theme.

💻 Coding Challenges

Challenge 1: Fix a Placeholder-Only Field

Given `<input type="text" placeholder="Full name">` with no label, rewrite it with a proper `<label>`.

Goal: Practice the single most common form-accessibility fix.

→ Solution

Challenge 2: Group a Checkbox Set

Given three unlabeled checkboxes for "Email me updates," "SMS me updates," and "Push notifications" with a nearby but unconnected visual heading "Notification preferences," wrap them correctly using `<fieldset>`/`<legend>`.

Goal: Practice applying fieldset/legend to a realistic checkbox group.

→ Solution

Challenge 3: Wire Up an Accessible Error

A password field fails validation with the message "Password must be at least 8 characters." Write the complete, correctly-associated HTML for the input and its error message.

Goal: Practice combining `aria-invalid` and `aria-describedby` correctly in one example.

→ Solution

⚠️ Gotcha: Reaching for `aria-label` Instead of a Visible Label

It's tempting to add `aria-label` to satisfy an accessibility checklist or scanner, when a genuinely visible `<label>` would have been just as easy to add and serves far more people. `aria-label` only helps screen reader users — it does nothing for users with cognitive disabilities, low vision, or anyone who simply benefits from seeing what a field is for rather than inferring it from context. A visible label serves everyone at once; reserve `aria-label` for cases where a visible label genuinely isn't appropriate (a well-known icon-only search field, per Chapter 3), not as a default shortcut to make an automated test pass.

🎯 What's Next

With forms covered, the next chapter turns to what's actually visible: **Color, Contrast & Visual Design** — WCAG contrast ratios, never relying on color alone, and respecting reduced-motion preferences.

🎨 Color, Contrast & Visual Design

Chapter 5 covered forms. This chapter covers what's actually visible — color, contrast, and motion — Chapter 1's Perceivable principle made concrete for users with low vision, color blindness, or motion sensitivity.

WCAG Contrast Ratio Requirements

Content Type	Minimum Ratio (Level AA)
Normal text	4.5:1
Large text (18pt+/24px+ regular, or 14pt+/18.66px+ bold)	3:1
UI components & graphical objects	3:1

The exact math isn't something to memorize — the thresholds and *why* they exist matter: low vision, aging eyes, and glare on mobile screens outdoors all make marginal contrast genuinely unreadable for a meaningful share of real users.

🚫 Never Relying on Color Alone

Any information conveyed via color must also be conveyed through at least one other channel — Chapter 1's own red/green form-error example is the classic case. This applies broadly:

Links in Body Text

Color alone doesn't reliably distinguish a link from surrounding text for a color-blind user — an underline or other non-color cue is needed too.

Status Indicators

A colored dot alone for "online/offline" needs an accompanying text label or icon — color-coded charts need patterns or direct labels, not just a color-keyed legend.

Color Alone vs Color Plus Another Channel

Color Only

A green dot means "online" — invisible information for anyone who can't distinguish that specific color.

Color + Text/Icon

A green dot *and* the word "Online" — the same information reaches everyone, regardless of color perception.

Text Resizing & Zoom Support

Users must be able to zoom or resize text up to 200% without losing content or functionality (WCAG 1.4.4). Common failures: fixed-height containers that clip text once font size increases, and text set with no responsive fallback at all.

```
<!-- Actively blocks a core accessibility feature — never do this -->
<meta name="viewport" content="width=device-width, user-scalable=no">

<!-- Correct: leaves zooming fully enabled -->
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

🌀 `prefers-reduced-motion`

A CSS media feature reflecting a user's OS-level preference to minimize animation — relevant for vestibular disorders, where parallax scrolling, large animated transitions, or auto-playing carousels can genuinely trigger motion sickness:

```
.carousel { animation: slide 8s infinite; }

@media (prefers-reduced-motion: reduce) {
  .carousel { animation: none; }
}
```

Respecting this preference means providing a reduced or eliminated-motion alternative for any significant animation — not just pure decoration, but often necessary functional transitions too.

💻 Coding Challenges

Challenge 1: Fix a Color-Only Status Indicator

A dashboard shows server status as only a colored circle (green = healthy, red = down), with no text or icon. Propose a fix that satisfies the "never color alone" principle.

Goal: Practice adding a second channel of information alongside an existing color signal.

→ Solution

Challenge 2: Identify the Contrast Failure

A button uses white text (#FFFFFF) on a light gray background (#D3D3D3) for its normal-size label. Explain why this likely fails WCAG AA, referencing this chapter's threshold table.

Goal: Practice reasoning about a contrast failure without needing to compute the exact ratio by hand.

→ Solution

Challenge 3: Respect Reduced Motion for a Functional Animation

A page uses a sliding animation to reveal a notification banner from off-screen. Write CSS that respects `prefers-reduced-motion` while still functionally showing the banner (not hiding it entirely) for users with that preference set.

Goal: Practice distinguishing "remove the motion" from "remove the content" for a functional (not purely decorative) animation.

→ Solution

⚠️ Gotcha: Disabling Zoom With `user-scalable=no`

Adding `user-scalable=no` (or `maximum-scale=1`) to the viewport meta tag explicitly disables pinch-to-zoom — one of the most direct, active violations of WCAG's zoom requirement possible. It's usually added by a developer trying to prevent "accidental zooming" during mobile interactions, without realizing it strips away a core, browser-provided accessibility feature that users with low vision genuinely depend on just to read the page at all. Never disable user scaling — if accidental zoom is a real concern, that's a UI/gesture-handling problem to solve some other way, not a reason to remove zoom entirely.

🎯 What's Next

With visual design covered, the next chapter goes directly to the assistive technology itself: **Screen Readers in Practice** — the accessibility tree vs the DOM, and testing with a real screen reader.

🔊 Screen Readers in Practice

Chapters 2 through 6 covered techniques. This chapter is about the assistive technology actually consuming all of it — screen readers — and how to genuinely test with one instead of guessing.

The Accessibility Tree, Not Just the DOM

Browsers build a separate **accessibility tree**, derived from the DOM but filtered and transformed: decorative elements (`aria-hidden`, empty `alt=""`) are removed, roles are computed by combining native semantics with any ARIA overrides (Chapters 2–3), and each element's **accessible name** is computed via a defined precedence order (`aria-labelledby`, then `aria-label`, then an associated `<label>`, then various fallbacks). This tree — not the visual DOM/CSSOM a sighted user's browser renders — is literally what a screen reader reads from. Every previous chapter has really been about correctly shaping this tree, not just "making things look right."

DOM / CSSOM	Accessibility Tree
What renders visually	What a screen reader announces
Includes every element	Decorative/hidden elements filtered out
Visual styling determines appearance	Computed roles and accessible names determine announcement

Testing With a Real Screen Reader

VoiceOver (built into macOS/iOS, free) and **NVDA** (free, Windows) are the standard tools. Browser DevTools' accessibility panel shows a useful static snapshot of a computed name or role — but it's not a substitute for actually navigating with a screen reader turned on.

Snapshot vs Real Session

DevTools Panel Only

Confirms an individual element's computed name/role looks correct in isolation — misses reading-order weirdness and awkward interaction flow entirely.

Real Screen Reader Session

Surfaces redundant or verbose announcements, confusing reading order, and interaction flows that look fine one element at a time but sound confusing in continuous use.

Visually-Hidden Text

A CSS class that hides content visually while keeping it in the accessibility tree — critically different from `display: none` or `visibility: hidden`, both of which remove content from the tree entirely:

The Standard Visually-Hidden Pattern

```
.visually-hidden {
  position: absolute;
  width: 1px; height: 1px;
  overflow: hidden;
  clip: rect(0, 0, 0, 0);
  white-space: nowrap;  /* prevents wrapping from expanding the clip box */
}
```

Used to add context sighted users get visually but a screen reader user tabbing through link names alone would miss entirely:

```
<a href="/policy">Read more<span class="visually-hidden"> about our return policy</span></a>
// Sighted users see "Read more" in context; a screen reader announces
// "Read more about our return policy" — not three identical "Read more"s
```

📢 `aria-live` Regions, Revisited

Value / Role	Behavior
`aria-live="polite"` / `role="status"`	Waits for a pause, doesn't interrupt current speech
`aria-live="assertive"` / `role="alert"`	Interrupts immediately — reserve for genuinely urgent information

```
<div role="status">12 results found</div>
```

💻 Coding Challenges

Challenge 1: Add Visually-Hidden Context to Repeated Links

A product listing page has three "View details" links, one per product ("Blue Sneakers," "Red Jacket," "Green Hat"). Rewrite each link so a screen reader announces which product it belongs to, without changing the visible text.

Goal: Practice the exact repeated-link-context pattern this chapter describes.

→ Solution

Challenge 2: Choose Polite or Assertive

For each of (a) a "Your changes have been saved" confirmation after a routine form edit, and (b) a "Your session is about to expire in 10 seconds" warning, choose `aria-live="polite"` or `aria-live="assertive"` and justify each.

Goal: Practice applying the urgency distinction to two realistically different cases.

→ Solution

Challenge 3: Explain Why a DevTools Check Wasn't Enough

A developer confirms every individual element's accessible name looks correct in the DevTools accessibility panel, then ships a page that real screen reader users report as confusing to navigate. Explain what kind of problem this scenario likely represents.

Goal: Practice articulating the gap between a per-element snapshot and a real, continuous screen reader session.

→ Solution

⚠️ Gotcha: Overusing `aria-live="assertive"`

It's tempting to reach for `aria-live="assertive"` (or `role="alert"`) for routine updates — a "form saved" confirmation, a "3 items in cart" counter update — because it feels like the more attention-grabbing, "important" choice. It isn't harmless: assertive interrupts whatever the screen reader was already announcing, which is disruptive and disorienting when used for anything that isn't genuinely urgent. Most updates belong under `aria-live="polite"` (or `role="status"`) — reserve assertive/alert strictly for the rare cases that truly warrant interrupting the user, like a critical error or a security warning.

🎯 What's Next

With the assistive technology itself understood, the next chapter puts everything together into real, working components: **Accessible Components & Patterns** — building a genuinely correct modal, dropdown/combobox, and tabs.

🧩 Accessible Components & Patterns

Chapters 2 through 7 covered individual techniques. This chapter builds three genuinely correct interactive components end-to-end — a modal, a combobox, and tabs — following the ARIA Authoring Practices Guide (APG), the reference documentation for exactly this kind of pattern.

The ARIA Authoring Practices Guide

A W3C resource documenting the correct keyboard model, roles, and states for common UI patterns. Checking the APG's pattern for a component is the right starting point when building from scratch — it exists precisely because these patterns require hand-built ARIA and JavaScript, exactly Chapter 3's "no native element covers this" case.

🪟 Component 1: Accessible Modal

Completing Chapter 4's Focus Trap

```
<div role="dialog" aria-modal="true" aria-labelledby="dialog-title">
 <h2 id="dialog-title">Confirm deletion</h2>
 <button>Cancel</button>
 <button>Delete</button>
</div>
```

`role="dialog"` plus `aria-modal="true"` announces this as a modal dialog; `aria-labelledby` gives it an accessible name from its own heading. Combined with Chapter 4's three-step focus management (move in, trap, restore) and an Escape-key handler to close it, this is a genuinely complete reference implementation.

🔽 Component 2: Accessible Dropdown/Combobox

```
<input role="combobox" aria-expanded="true"
aria-controls="suggestions" aria-activedescendant="option-2">

<ul id="suggestions" role="listbox">
  <li id="option-1" role="option">Apple</li>
  <li id="option-2" role="option">Banana</li>
</ul>
```

`aria-activedescendant`

Tracks which option is currently highlighted as arrow keys move through the list — **without** moving actual DOM focus away from the input. Real focus stays on the input the entire time.

Full Keyboard Model

Arrow keys navigate options, Enter selects the highlighted one, Escape closes the list — none of this comes from the roles alone.

📑 Component 3: Accessible Tabs

Completing Chapter 3's `role="tablist"` example:

Roving Tabindex

```
<div role="tablist">
 <button role="tab" aria-selected="true" tabindex="0">Details</button>
 <button role="tab" aria-selected="false" tabindex="-1">Reviews</button>
</div>

<div role="tabpanel" aria-labelledby="tab-details">...</div>
```

Only the *active* tab has `tabindex="0"`; every other tab has `tabindex="-1"` — one Tab press enters the whole tablist, and Left/Right arrow keys move the "roving" focus between individual tabs, not Tab itself. This is the general APG convention across list-like widgets, not just tabs.

Improvising vs Following the APG

> **Improvised**
>
> Guessing at roles and keyboard behavior invites inconsistency with what users expect from every other tab/combobox/dialog they've ever used.

> **APG Pattern**
>
> A documented, battle-tested reference for exactly this component — less to get wrong, and consistent with the rest of the web.

💻 Coding Challenges

Challenge 1: Add Escape-to-Close to the Modal

Write the keydown handler that closes the modal from Component 1 when Escape is pressed, and correctly restores focus per Chapter 4's third step.

Goal: Practice completing the modal pattern with the missing keyboard piece.

→ Solution

Challenge 2: Implement Roving Tabindex for Arrow Keys

Write the JavaScript for the tabs component that, on Right Arrow, moves `tabindex="0"` and `aria-selected="true"` to the next tab (setting the previous one to `tabindex="-1"`/`aria-selected="false"`) and focuses it.

Goal: Practice implementing the roving tabindex keyboard behavior, not just the static ARIA markup.

→ Solution

Challenge 3: Explain Why Focus Stays on the Combobox Input

Explain why the combobox pattern uses `aria-activedescendant` to track the highlighted option instead of simply moving real DOM focus to each option as the user presses arrow keys.

Goal: Practice articulating the reasoning behind this specific, non-obvious pattern.

→ Solution

⚠️ Gotcha: Getting the ARIA Half Right and Shipping a Broken Keyboard Half

This is Chapter 3's core lesson resurfacing specifically for these more complex components: it's entirely possible to add `role="tablist"`/`role="tab"` correctly, get every accessible name right, and still ship a broken experience by forgetting the arrow-key navigation the pattern requires — leaving only default Tab behavior, which cycles through every tab individually rather than the expected roving-tabindex model. The roles and states are only half of any APG pattern; the documented keyboard interaction model is the other half, and skipping it produces a component that looks correct in an accessibility tree inspection while still being genuinely broken to actually use.

🎯 What's Next

Every technique and component pattern comes together in the final chapter: **Capstone: Auditing and Fixing an Inaccessible Page** — walking a deliberately inaccessible example page through every fix from the course.

🏁 Capstone: Auditing and Fixing an Inaccessible Page

Every previous chapter built one piece in isolation. This capstone walks one deliberately inaccessible example page through a real audit — automated scan, semantic HTML, ARIA, keyboard, forms, visual design, screen-reader specifics, and custom components — using every technique from the course, then confirms it with a real screen reader.

The Starting Page

An e-commerce product page with a realistic pile of problems: div-based nav "buttons," a conflicting `<a role="button">`, no skip link, a modal that doesn't trap or restore focus, placeholder-only form fields, color-only stock status, `user-scalable=no`, an unthrottled carousel ignoring `prefers-reduced-motion`, repeated "Read more" links with no context, misused `aria-live="assertive"`, and a custom tabs widget with ARIA roles but no arrow-key support.

1️⃣ Automated Scan First

Run axe/Lighthouse (Chapter 1) to catch the mechanically-detectable ~30–40%: missing labels, low contrast, missing alt text. This is the starting point, not the finish line.

2️⃣ Fix Semantic HTML & ARIA

```
<!-- Before -->
<div onclick="goToCart()">Cart</div>
<a href="/wishlist" role="button">Wishlist</a>

<!-- After — Ch.2 native elements, Ch.3 no conflicting/redundant ARIA -->
<button onclick="goToCart()">Cart</button>
<a href="/wishlist">Wishlist</a>
```

3️⃣ Fix Keyboard Navigation

Add a skip link and `:focus-visible` styling (never `outline: none` alone), and fix the modal to move focus in, trap it, and restore it on close — Chapter 4's full three-step checklist, none of which was implemented before.

4️⃣ Fix Forms

```
<!-- Before -->
<input placeholder="Promo code">

<!-- After — Ch.5 real label + accessible error -->
<label for="promo">Promo code</label>
<input id="promo" aria-describedby="promo-error">
<span id="promo-error">Code expired</span>
```

5️⃣ Fix Visual Design

Raise low-contrast text to meet Chapter 6's 4.5:1 threshold, add a text label alongside the color-only "in stock/out of stock" indicator, remove `user-scalable=no`, and add a `prefers-reduced-motion` override for the carousel.

6️⃣ Fix Screen-Reader-Specific Issues

Add visually-hidden context to every repeated "Read more" link (Chapter 7), and downgrade the routine "item added to cart" announcement from `aria-live="assertive"` to `"polite"`.

7️⃣ Fix Custom Components

Add roving-tabindex arrow-key navigation to the product-details tabs widget (Chapter 8) — the ARIA roles were already present, but arrow-key support was missing entirely.

8️⃣ Manual Re-Test With a Real Screen Reader

Confirm every fix actually works by testing with VoiceOver or NVDA (Chapter 7) — not just re-running the automated scan, which would still miss reading order and interaction-flow issues no scanner can catch.

The Reusable Methodology

Automated scan → semantic/ARIA → keyboard → forms → visual → screen-reader specifics → components → manual re-test. Not specific to this one page — a repeatable process for auditing any real page.

What Changed the Experience

The page went from unusable for keyboard/screen-reader users to one where every control is reachable, operable, correctly announced, and never leaves someone stuck or lost.

💻 Coding Challenges

Challenge 1: Fix the Carousel's Reduced-Motion Gap

The page's product-image carousel auto-rotates every 4 seconds with a sliding animation and completely ignores `prefers-reduced-motion`. Write the CSS fix, citing which chapter's principle it applies.

Goal: Practice applying a specific earlier-chapter fix within this capstone's broader audit.

→ Solution

Challenge 2: Explain Why Step 2 Should Happen Before Step 3

Explain why converting div-based buttons to real `<button>` elements (Step 2) should happen before fixing the skip link and focus trap (Step 3), rather than in the reverse order.

Goal: Practice reasoning about why this methodology's step order isn't arbitrary.

→ Solution

Challenge 3: Apply the Methodology to a New Page

A completely different page — a SaaS dashboard with a data table, a settings form, and a notification bell icon — has accessibility problems. Walk through this chapter's eight-step methodology at a high level for this page, without writing code.

Goal: Practice applying the reusable audit process to a page this course never specifically covered.

→ Solution

⚠️ Gotcha: Trusting a Clean Automated Re-Scan as "Done"

After applying every fix, it's tempting to re-run axe/Lighthouse, see zero flagged issues, and call the audit complete — exactly the mistake Chapter 1 warned against from the very start of this course. Fixes can also interact: converting div-buttons to real `<button>` elements in Step 2 changes the page's actual tab order and count, which can shift where a skip link should land or where a focus trap's boundaries actually are — issues an automated scanner has no way to catch, since it doesn't understand what "makes sense" during real, continuous keyboard or screen reader use. Re-test manually after each round of fixes, not just once at the end, and never mistake an automated-clean report for a genuinely accessible page.

🎓 Course Complete

This closes out **Web Accessibility (a11y)** — from why accessibility matters and the POUR principles, through semantic HTML, ARIA, keyboard navigation, forms, color and contrast, screen readers, accessible components, and finally this capstone auditing a real page end to end.