Three Developer Behaviors to Eliminate 85 Percent of Accessibility Defects

Seán Kelly
Thomas F. Dinkel

CSUN 2018, San Diego
March 22, 2018

#CSUNATC2018

OPTUM® | User Experience Design Studio

Thomas F. Dinkel “Dink”

- Graphic / Industrial design degree
- Early career: User Experience Engineer
  - Part designer, part client-side developer (often mediating between business, design and development)
- First priority has always been the User Experience (followed by Standards/Browser Compatibility)
- Started paying attention to a11y when Dreamweaver was still made by “Macromedia”
- Accessibility Engineer since 2013 (focusing on helping developers implement accessible solutions)

Seán Kelly

- Former photographer and print/media artist
- Web since 1996, Accessibility since 1998
- 10+ Years in Public Sector
  and Higher Education
- 8 Years in State Government
- A11y at Optum since 2014

Outline

- Intro
- Behavior One: Perform HTML Validation
- Behavior Two: User A11y Page Scanning Tools
- Behavior Three: Test Keyboard Operation
- Bonus Behavior: Try it with a Screen Reader
- Conclusion: Benefits and Early Inclusion in Process
- Thank You!

Intro

During numerous development and remediation projects at Optum, we found that a surprising number of defects are preventable by changing just three developer behaviors.

85% ?!?

A brief summary of how we came up with 85%

- Number of defects tracked vs. number of WCAG Success Criteria or other measures
- Sample size and the point at which A11y was inserted into the process with the sample projects
- 85% sounds ambitious but these changes to developer habits can eliminate a surprising number of defects

Behavior One:

Perform HTML Validation

Good code is a major part of WCAG 2.0

Principle 4: Robust - Content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies.

> Guideline 4.1 Compatible: Maximize compatibility with current and future user agents, including assistive technologies.

HTML is a standard

Because when you write HTML, you are participating in a *standard*.

HTML is the *standard* markup language for creating Web pages.
-- w3schools.com

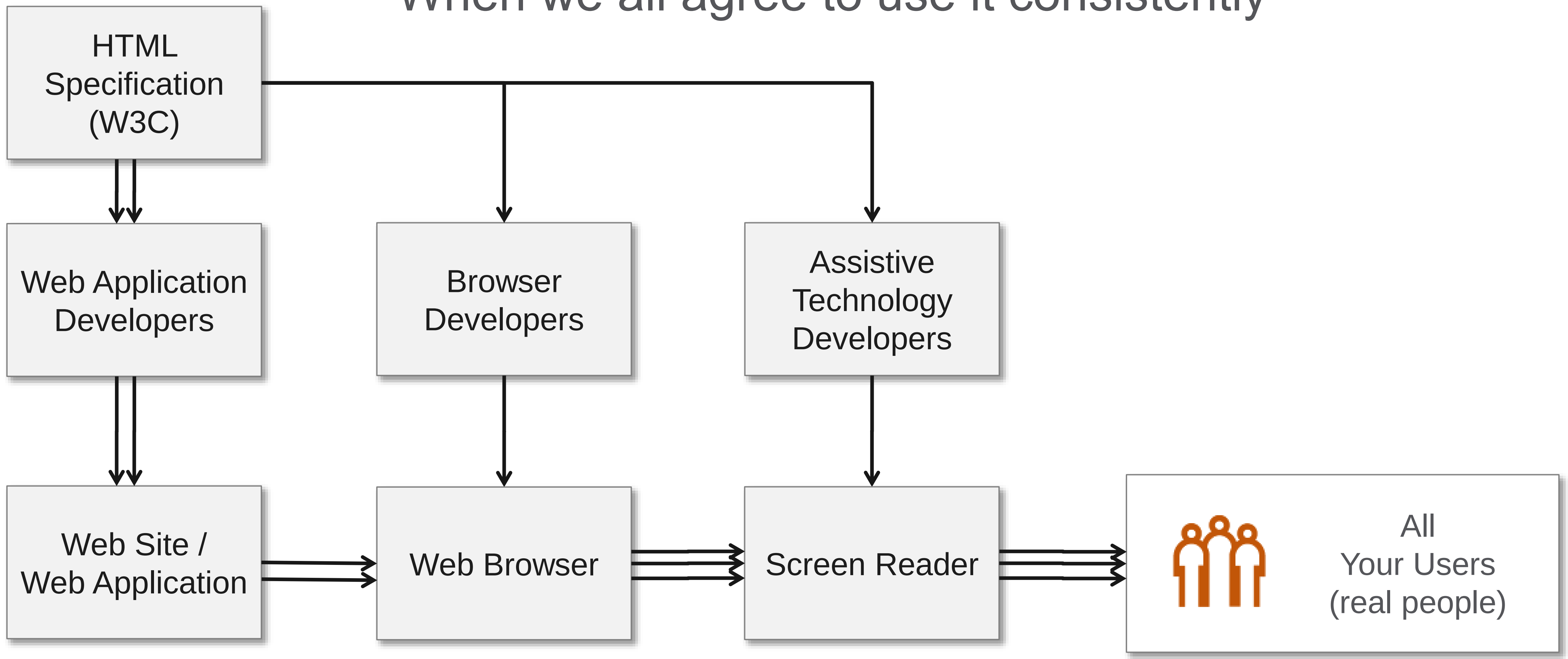
Hypertext Markup Language (**HTML**) is the *standard* markup language for creating web pages and web applications.
-- wikipedia.org

HTML - HyperText Markup Language: a set of *standards*, a variety of SGML, used to tag the elements of a hypertext document. It is the standard protocol for formatting and displaying documents on the World Wide Web.
-- dictionary.com

HTML - Hypertext Markup Language, a *standardized* system for tagging text files to achieve font, colour, graphic, and hyperlink effects on World Wide Web pages.
-- oxforddictionaries.com

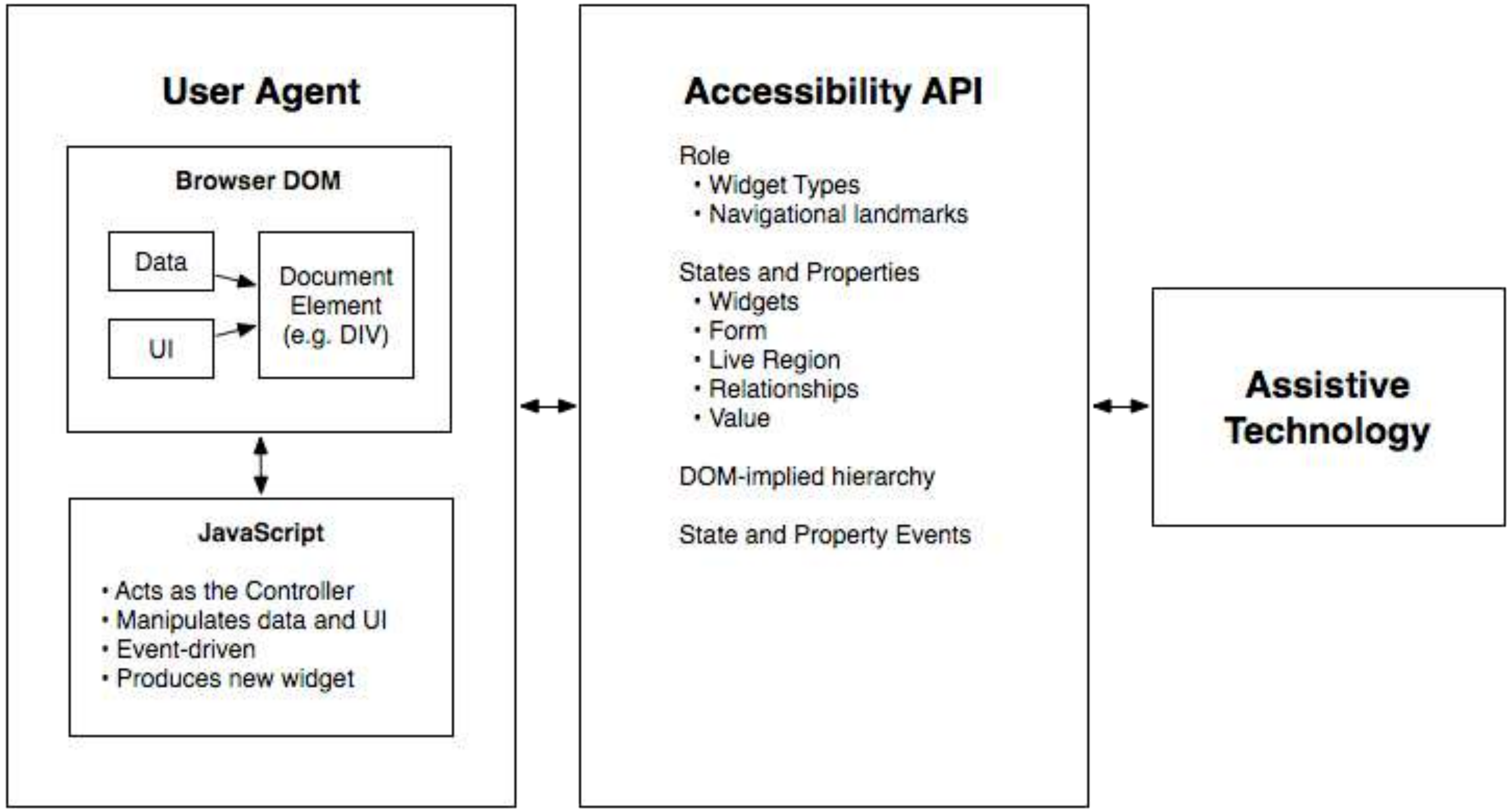
We all benefit from the standard

When we all agree to use it consistently


HTML Specification (W3C)
Web Application Developers
Browser Developers
Assistive Technology Developers
Web Site / Web Application
Web Browser
Screen Reader
All Your Users (real people)

The contract model with accessibility APIs

From the Web Accessibility Initiative's "Accessible Rich Internet Applications"



http://www.w3.org/TR/wai-aria-1.1/#intro_ria_accessibility

Browsers and Assistive Technologies

We want our stuff to work in the browsers our users use, not just the ones our developers use.

Take it a step further though: we want our stuff to work with browsers and the assistive technologies that our users use.

Internet Explorer 11
Microsoft

Firefox
moz://a

VoiceOver
Apple

Safari
Apple

JAWS
JAWS for Windows
Freedom Scientific

TalkBack
Google

DRAGON
NATURALLYSPEAKING

chrome
Google

ZoomText

NVDA
NV Access
Empowering lives through non-visual access to technology

Edge
Microsoft

Browser and Assistive Technology combinations (BrAT)

Sometimes, even when HTML is valid and follows the standard, users may get different experiences with different BrATs (for example: IE11+JAWS verses Firefox + JAWS).

In our experience, good, clean, standard HTML always does better than bad code.

Browsers can fix-up bad HTML to some degree so that it looks right, but it can still be presented incorrectly by different BrATs.

When things are a lot different between BrATs, there is often a code validation issue behind it.

Bad code example: a block breaking a block element

Nu Validator reported the following error:

No `p` element in scope but a `p` end tag seen.

```
<p>The quick brown fox
  <div><img src="image.jpg" alt="Fox" /></div>
  jumps over the lazy dog.</p>
```

The browser fix-up may be presented by AT as:
P: The quick brown fox
DIV: IMG
P: jumps over the lazy dog.

Bad code example: Bad parent-child relationship

Nu Validator reported the following error:

Element ______ not allowed as child of element ______ in this context.
(Suppressing further errors from this subtree.)

```
<button>
  <div>Alerts</div>
  <p> 3</p>
</button>
```

The browser fix-up may be presented by AT as:
BUTTON: Alerts
BUTTON: 3

Custom elements

The purpose of a custom element is unclear to the browser

- The browser doesn’t know if it’s a block- or an inline-element.
- It has no semantics, none of the technologies understand what it’s for.

Support for custom elements / web components is coming, but it’s not ready for prime time yet.

Bad code example: custom element causing an unknown relationship

Nu Validator reported the following error:

Element ______ not allowed as child of element ______ in this context. (Suppressing further errors from this subtree.)

```
<ul>
  <li>Notifications</li>
  <cust-notify>
    <li>3 messages need your attention </li>
    <li>2 items added to your queue </li>
  </cust-notify>
</ul>
```

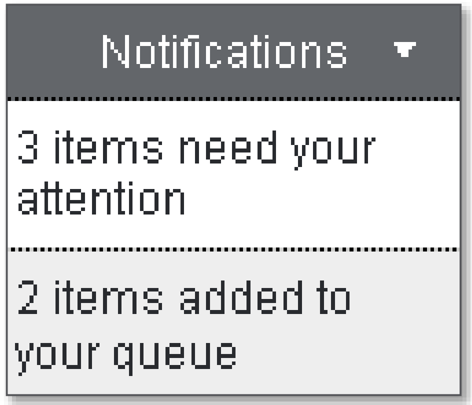
The browser fix-up may be presented by AT as:

- Notifications
- 3 messages need your attention
- 2 items added to your queue

It looked right due to CSS

They wanted it to show up like this:

- Notifications
  - 3 messages need your attention
  - 2 items added to your queue



But because the browser doesn't know if the custom tag is supposed to be an LI or a new UL, it wasn't showing up as it was intended to AT.

How strict should we be with custom attributes

The HTML specification says a couple interesting things about custom attributes:

- Custom attributes should be prefixed with “data-”.
- User agents must ignore attributes they don’t recognize.

This is our “absolute minimum” recommended practice:

- Don’t use attribute names that are already taken.
  - For example: No “href” on DIVs or SPANs.
- Please use some sort of custom prefix.
  - It makes it easier to know where attributes came from in a multiple-library world.
  - For example: Angular uses “ng-” and Optum’s UI Toolkit uses “tk-”.

Valid custom attributes

Basis for recommendation

- We don’t want custom attributes:
  - that might conflict with existing, or future, attributes.
  - that will be confused with existing attributes from other elements.
  - that might conflict with attributes from other libraries.

- The best practice – custom attributes should:
  - start with “data-” or “x-”; these will never conflict with anything from the HTML specification.
  - have your organization’s own unique identifier in it (vendor).
  - have a feature identifier in it.
  - provide an option to customize prefixes.

Good code example: custom attributes

For example:

```
<div role="dialog" …
  data-op-tk-dialogTitle="Your session is about
    to expire"
  data-op-tk-dialogIsModal="true">
```

- Vendor: op = Optum
- Product/Feature: tk = UI Toolkit

Why isn’t HTML validation being done?

“It looks right in my browser”
But will that still be true after the next major browser update?

“We’ve seen valid HTML that isn’t accessible”
And we’ve seen invalid HTML that is.

“With HTML5, all HTML is valid”
HTML5 is very flexible, but not *that* flexible.

“Most of the big player’s sites don’t pass”
Can you afford the same risk that they can?

“I don’t write HTML”
The JS framework we use generates the HTML, and they must know what they’re doing.

“HTML validation is:
- too hard
- takes too much time
- plagued with false positives

HTML Validation Options

Online with the NU validator:

Nu Html Checker

https://validator.w3.org/nu/

- No installation required
- Excellent "results filtering" feature

NU validator from the command line:

The Nu Html Checker (v.Nu)

https://validator.github.io/validator/

- Maybe make it part of your build process!
- Or part of QA automation!

Plug-in for Chrome and Firefox:

HTML Validator

http://users.skynet.be/mgueury/mozilla/

- Validates the HTML after JavaScript execution

IDE "linter" plug-ins:

Most IDEs have an HTML linter plug-in that can be added to help developers create valid code.

Things to Watch For

Static HTML file verses the DOM

HTML validation results are likely to vary when performed on the source file as opposed to what is in the DOM.

State and dynamic content changes

Anytime JavaScript is used to update some part of the screen, the HTML in the DOM has changed.

Other benefits of valid HTML

- Performance
- Browser’s “reader views”
- Voice assistants
- A11y checker-tools
- SEO (your page is more like data)

Behavior Two:

Use A11y Testing Tools

Accessibility Testing Tools

Page at a time:

- aXe https://www.deque.com/products/axe/
- WAVE https://wave.webaim.org/extension/
- WAT https://developer.paciellogroup.com/resources/wat/
- HTML CodeSniffer http://squizlabs.github.io/HTML_CodeSniffer/

Site scanning:

- Tenon, Compliance Sherriff, AMP, others

Accessibility Testing Tools

- For these tools to do their best work, make sure you've done HTML validation first!
- They can help identify roughly one third of accessibility issues
  - What this means: issues categorized by WCAG Success Criteria (or other checkpoint) vs. the number of issues
  - Why this fraction is important.
  - Types of issues identified, including ARIA.
- Make certain to check all possible states

  Any interaction that updates the page (opening menus, displaying and recovering from error messages, et al.) has an opportunity for to create issues.

aXe

There are a number of tools that one can use for scanning individual pages.

aXe Developer Tools (a browser plugin) is easy to get started with and easy to use and understand.

A related technology,—aXe Core—can be used for automated checking from the command line or as part of a Continuous Integration (CI) model.

Things to like:

- It makes it easy to focus on one issue at a time
- Built-in solution suggestions with external links—gentle introduction
- Displays info in the bottom of the screen in the panel with the code inspector

Behavior Three:

Test Keyboard Operations

WCAG Requirement

Principle 2 – Operable

User interface components and navigation must be operable.

Guideline 2.1 – Keyboard Accessible

Make all functionality available from a keyboard.

2.1.1 Keyboard – Level A

All functionality of the content is operable through a keyboard interface without requiring specific timings for individual keystrokes, except where the underlying function requires input that depends on the path of the user's movement and not just the endpoints.

2.1.2 No Keyboard Trap – Level A

If keyboard focus can be moved to a component of the page using a keyboard interface, then focus can be moved away from that component using only a keyboard interface, and, if it requires more than unmodified arrow or tab keys or other standard exit methods, the user is advised of the method for moving focus away.

2.1.3 Keyboard (No Exception) – Level AAA

All functionality of the content is operable through a keyboard interface without requiring specific timings for individual keystrokes.

AT Depends on Effective Keyboard Support

- Users that have visual impairments cannot depend on the position of a pointing device.
- Users with motor issues are often not able to use a pointing device—keyboard-only users.
- Assistive technologies such as voice input/control (e.g., Dragon Naturally Speaking) hardware devices (e.g., sip-and-puff switches), may emulate keyboard inputs.

All Users Benefit from Effective Keyboard Support

- Advanced- or power-users (developers and customer service representatives are often part of this group), benefit from the speed and efficiency of being able to stay on the keyboard without having to switch to a mouse.
- Data entry intensive work—any of us who are working within a form tend to focus on the keyboard.
- Users who are physically multitasking, e.g., holding a sandwich, using a telephone handset, or petting a cat may prefer being able to work using the keyboard alone.

Everything via Keyboard

Infrequently one will find situations where the same task is accomplished with a slightly different workflow, but the goal is to design the same experience for all users.

Sites are already being tested using a mouse, testing with the keyboard alone is a comparatively small addition.

Most-basic Keyboard Behaviors

- Visible Focus: You should be able to easily see where the current focus is.
- Focus Order: Matches visual order (i.e., left to right and then down)
- Controls: are in the tab order and they should appear in a natural order
- Active Controls: can be activated using the enter key (and spacebar for some specific controls).

Expected Keyboard Behaviors

- A Basic Primer: https://webaim.org/techniques/keyboard/

- Ensuring keyboard control for all functionality:
- https://www.w3.org/TR/2016/NOTE-WCAG20-TECHS-20161007/G202

- Very thorough and covers custom controls: https://www.w3.org/TR/wai-aria-practices-1.1/

Bonus Behavior:

Try it with a Screen Reader

Try it with a screen reader

Most developers have their favorite web browser that they use to test.

Some developers may want to understand more about the AT user’s experience, about why we’re doing all this.

Learn a little at a time

Not to master the screen reader, just to begin to understand.

For the screen reader of their choice, learn to:

1. As they “tab” through the screen, listen for each element’s *name*, *role*, *states / properties*, and additional descriptive information.
2. Move around the page using headings.
3. Browse / read text on the page.

Getting started with screen reader - Resources

- *WebAIM*
  - *https://webaim.org/techniques/screenreader/*
  - *https://webaim.org/articles/screenreader_testing/*

- JAWS
  - *https://webaim.org/articles/jaws/*
  - http://www.freedomscientific.com/Content/Documents/Manuals/JAWS/JAWS-Quick-Start-Guide.pdf

- NVDA
  - *https://webaim.org/articles/nvda/*
  - https://www.nvaccess.org/files/nvda/documentation/userGuide.html
  - NVDA help menu: User Guide and Commands Quick Reference

- VoiceOver
  - https://help.apple.com/voiceover/info/guide/10.12/
  - Definitely do the tutorial

Summary

Summary

We often refer to what we've covered here as "Three Plus One"

1. Perform HTML Validation
   - Good code should be part of every developer's job
   - This isn't even considered "extra effort" for accessibility
2. Use A11y testing tools
   - Great intro and insight into accessibility for developers
3. Test Keyboard Operations
   - Developers love good keyboard support in their IDEs
   - Developers already do mouse testing; check it out with a keyboard too!

- Try it with a Screen Reader
  - Start to get familiar with a screen reader
  - A little at a time

Benefits and Early Inclusion in Process

- Benefits of these three behaviors (+ the fourth ”bonus”):
  - Fewer total defects
  - Finding defects early is far cheaper than letting them get all the way to QA or A11y evaluation.
  - One can defend against these “programmatically discoverable defects”—developers fixing them is much more efficient than working with A11y engineers to do the same.
  - A11y engineers can focus on best practices and less on the repetition of the basics
  - Hopefully by following these updated behaviors you too will find a huge reduction in your A11y defects.

Thank you!

Thomas Dinkel

thomas.dinkel@optum.com

thomas.f.dinkel@gmail.com

@phrenishious

linkedin.com/in/thomas-dinkel-629a491/

Seán Kelly

sean_kelly@optum.com

sean.kelly.net@gmail.com

@sk55408

inkedin.com/in/seankellyweb/