ARIA

INTRODUCTION

Some common terms

Before we dive into ARIA, we are going to look at some **common terms** that will come up doing this course.

Assistive technology

“Assistive technology” is an overall term that includes assistive, adaptive, and rehabilitative devices.

Assistive technologies are designed to enable people with disabilities to **perform tasks that they were formerly unable to accomplish**, or had great difficulty accomplishing.

There are a **wide range of Assistive Technologies** available to support different disabilities.

Output devices:

1. Text-based browsers,
2. Screen Readers,
3. Magnifiers,
4. Refreshable Braille Devices
5. And more…

Input devices:

1. Accessible keyboards
2. Track pads
3. Head wands
4. Puffers & Switches
5. Touch screens
6. Voice activation software
7. And more…

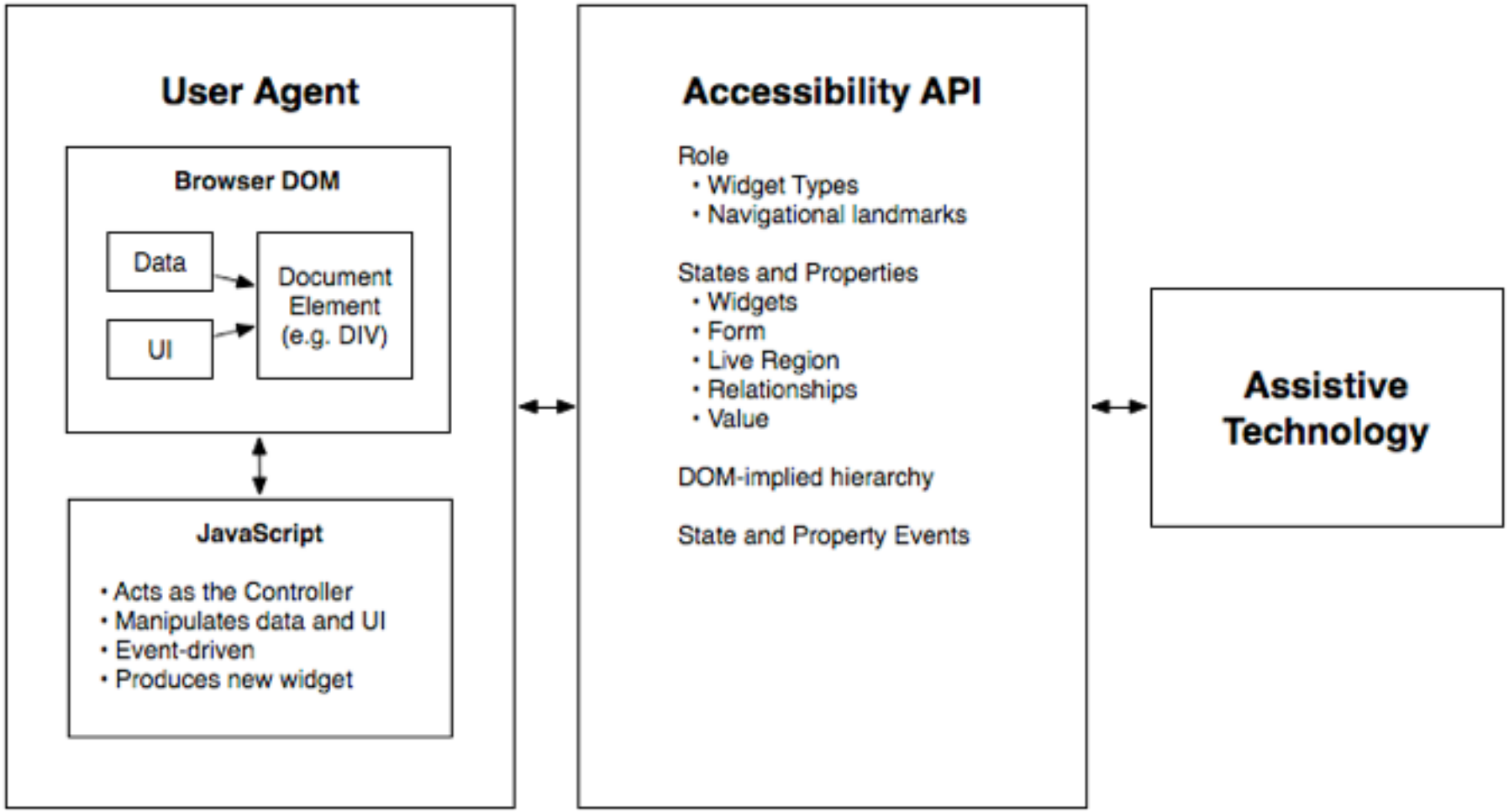
Accessibility API

Accessibility application programming interfaces (APIs) are used to **communicate semantic information about the user interface** to Assistive Technologies.

“Accessibility APIs constitute a **contract between applications and assistive technologies**, to enable them to access the appropriate semantics needed to produce a usable alternative to interactive applications.”

https://www.w3.org/TR/2012/WD-wai-aria-implementation-20120816/#intro

For example, the Accessibility API **helps screen reading software** determine whether a particular UI widget is a menu, button, text field, list box, etc.


User Agent
Browser DOM
Data
UI
Document Element (e.g. DIV)
JavaScript
• Acts as the Controller
• Manipulates data and UI
• Event-driven
• Produces new widget
Accessibility API
Role
• Widget Types
• Navigational landmarks
States and Properties
• Widgets
• Form
• Live Region
• Relationships
• Value
DOM-implied hierarchy
State and Property Events
Assistive Technology

Accessibility APIs **represent objects in a user interface**, exposing information about each object within the application.

Typically, there are **several pieces of information for an object**, including:

1. The object's **role** (e.g. a menu, a button, an input, an image).

2. A **name that identifies it within the interface** (e.g. a visible label or a name that has been encoded directly in the object).

3. The object's **state** (e.g. selected, unselected, checked, unchecked).

A brief history of APIs

In 1997, Microsoft introduced **Microsoft Active Accessibility** (MSAA) into Windows 95.

In 1998, IBM and Sun Microsystems built **a cross-platform accessibility API** for Java.

In 2001, the **Assistive Technology Service Provider Interface** (AT-SPI) for Linux was released.

In 2002 Apple included the **NSAccessibility** protocol with Mac OS X (10.2 Jaguar).

More than one API?

Browsers typically support **one or more of the available accessibility APIs** for the platform they're running on.

Safari and Chrome support **NSAccessibility** on OS X and **UIAccessibility** on iOS.

In Windows, Firefox and Chrome support **MSAA/IAccessible** and **IAccessible2**.

Internet Explorer supports **MSAA/IAccessible** and **UIAExpress**.

This is why you should always test against **more than one** Browser/ Assistive Technology combination.

Windows

IE: JAWS & NVDA
FireFox: JAWS & NVDA
Chrome: JAWS & NVDA

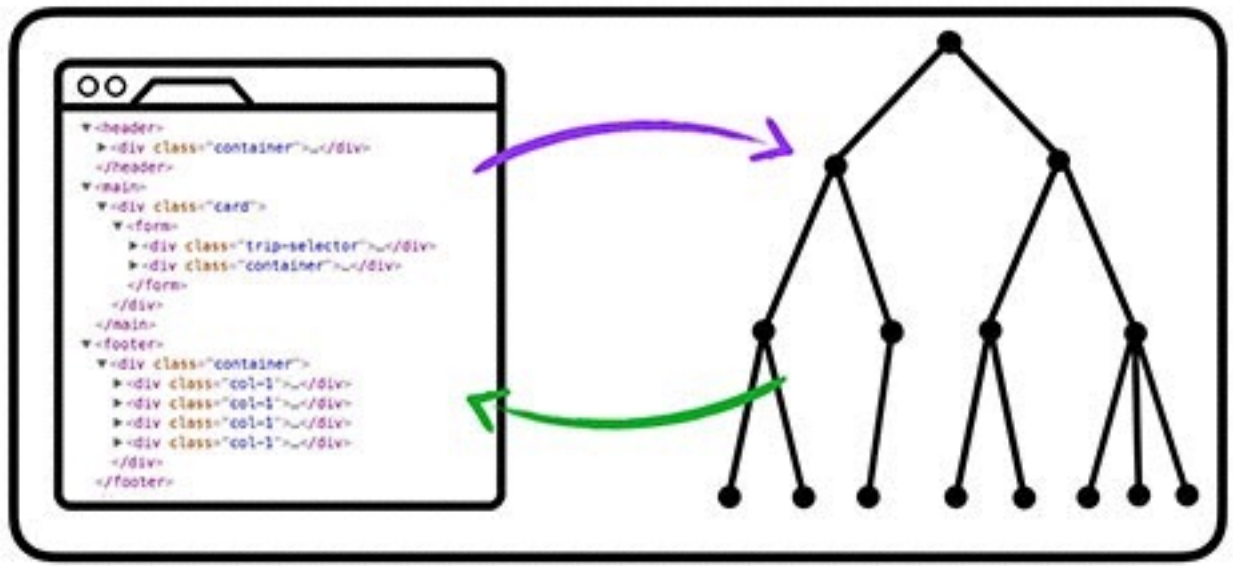
OSX

Safari: VoiceOver
FF: VoiceOver
Chrome: VoiceOver

Accessibility Tree

Browsers take the DOM tree **and modify it**, to turn it into a form that is useful for assistive technologies.

This modified tree, is referred to as **the accessibility tree** - a subset of of the DOM tree.


▼<header>
▶ …
</header>
▼<main>
▼
▼<form>
▶ …
▶ …
</form>

</main>
▼<footer>
▼
▶ …
▶ …
▶ …
▶ …

</footer>
DOM
accessibility tree

The accessibility tree contains **only “Accessible objects”**. These are nodes that have states, properties or events.

All other DOM nodes (that do not have states, properties or events) **are not presented in the accessibility tree**.

For example, **a section within the DOM tree** could be:

```
<div class="container">
  <form action="#">
    <div class="form-container">
      <label for="name">Name</label>
      <input id="name" type="text">
    </div>
    <div class="form-container">
      <button type="submit">Submit</button>
    </div>
  </form>
</div>
```

The Accessibility tree would **only present the following**:

```
<form action="#">

    <label for="name">Name</label>
    <input id="name" type="text">


    <button type="submit">Submit</button>

</form>
```

Each browser could potentially present a **slightly different accessibility tree**.

Widgets

Within the various WAI ARIA specifications, there are multiple references to **“widgets”**.

Within the ARIA specifications, a widget is defined as a **component of an overall interface** that enables a user to perform a function or access a service.

For example, a widget could be any **stand-alone UI component** such as a dropdown menu, a modal or a tooltip.

Why is ARIA needed?

The HTML markup language has a **very limited set of interface controls** and interactions.

As the demand for rich interactions has increased, **JavaScript has become our saviour**!

JavaScript provides us with **many things** including:

dynamic interactions:
such as drag and drop, resizing, hide and show, open and shut, switch views etc.

widgets and components:
such as modals, in-page tabs, button drop-downs, date pickers, page loaders, sliders and much more.

However, many of these dynamic interactions and widgets are **problematic for Assistive Technologies**.

Issues with dynamic content

Assistive Technologies may not be aware that **content that has been dynamically updated** after the initial page has loaded.

This means that this dynamically added content may be **totally inaccessible** to some audiences.

Issues understanding the purpose

Assistive Technologies **may not be aware of the purpose of non-native widgets** - such as dropdown buttons, in-page tabs, accordions etc.

This means that **the purpose of a widget may be confusing** for some users.

This is especially true for widgets that use incorrect HTML elements and **rely on JavaScript in order to function**.

In these cases, the entire widget may **totally inaccessible** for some users.

Issues understanding the state

Assistive Technologies **may not be aware of the state of a widget** - such as an item that has been selected or checked.

This means that users **may not be able to tell whether they have selected or checked an option** within a widget.

ARIA to the rescue!

What is ARIA?

“WAI-ARIA is a technical specification that provides a framework to **improve the accessibility and interoperability** of web content and applications.”

https://www.w3.org/WAI/intro/aria

WAI:
Web Accessibility Initiative

ARIA:
Accessible Rich Internet Applications

WAI-ARIA 1.0 was published as a completed **W3C Recommendation** on 20 March 2014.

https://www.w3.org/TR/wai-aria/

WAI-ARIA 1.1 was published as a **Candidate Recommendation** on 27 October 2016.

https://www.w3.org/TR/wai-aria-1.1/

Published **WAI-ARIA 1.0** working drafts include:

WAI-ARIA technical specification

Primarily for developers of Web browsers, assistive technologies, and accessibility evaluation tools.

https://www.w3.org/TR/wai-aria/

WAI-ARIA User Agent Implementation Guide

Describes how browsers and other user agents should support WAI-ARIA

https://www.w3.org/TR/wai-aria-implementation/

WAI-ARIA Authoring Practices

Describes how Web developers can develop accessible rich internet applications using ARIA, with detailed advice and examples.

https://www.w3.org/TR/wai-aria-practices/

WAI-ARIA Primer

Introduces developers to the accessibility problems that WAI-ARIA is intended to solve.

https://www.w3.org/TR/wai-aria-primer/

WAI-ARIA Roadmap
defines the path to make rich Web content accessible, including steps already taken, remaining future steps, and a timeline.

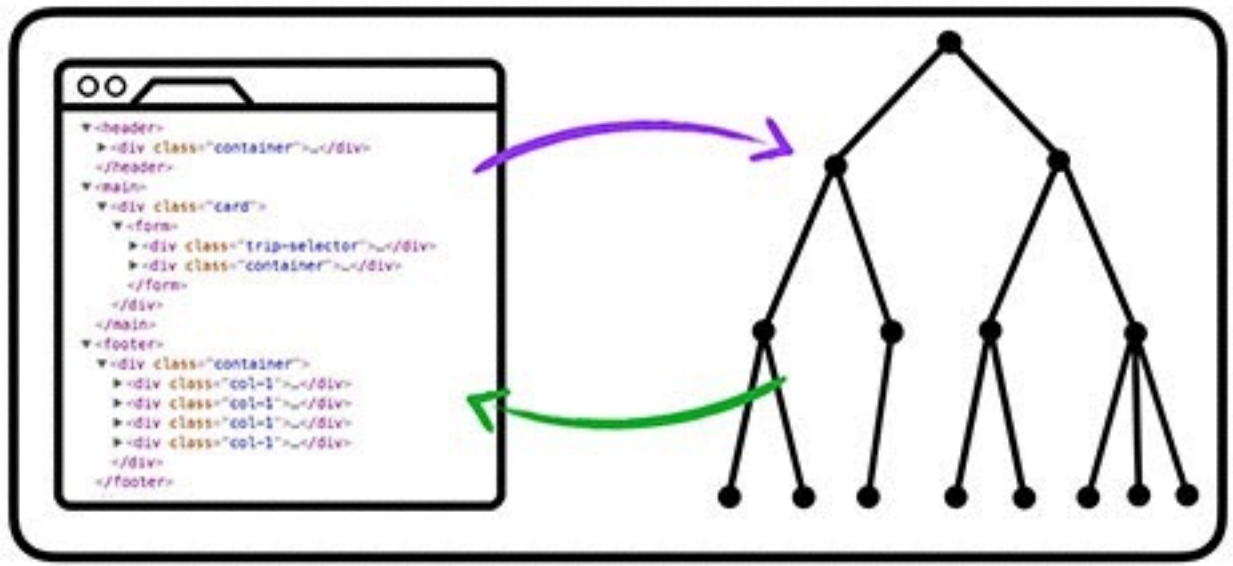
https://www.w3.org/TR/wai-aria-roadmap/

How does ARIA work?

ARIA uses a range of HTML attributes to convey **additional semantics** to HTML elements.

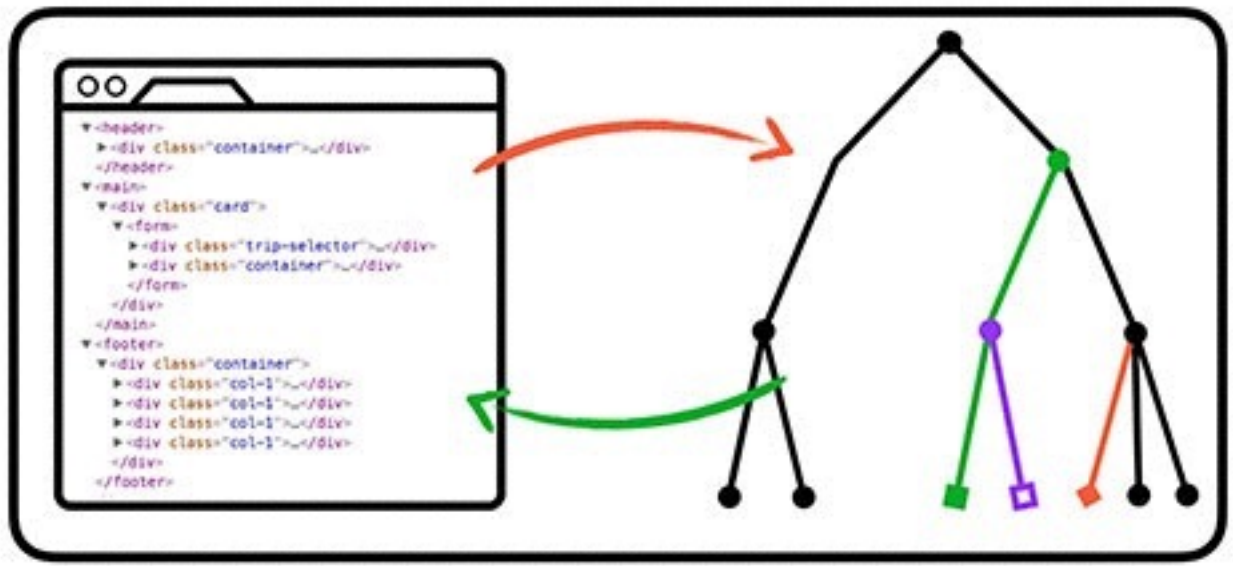
```
<ul role="menu">
</ul>

<input aria-checked="true" type="radio">

<button aria-label="Close application">
  Close
</button>
```

These HTML attributes **change and augment the Accessibility Tree**.


▼<header>
▶ …
</header>
▼<main>
▼
▼<form>
▶ …
▶ …
</form>

</main>
▼<footer>
▼
▶ …
▶ …
▶ …
▶ …

</footer>
DOM
accessibility tree


▼<header>
▶ …
</header>
▼<main>
▼
▼<form>
▶ …
▶ …
</form>

</main>
▼<footer>
▼
▶ …
▶ …
▶ …
▶ …

</footer>
DOM
+
ARIA
accessibility
tree

ARIA allows us to **adjust the Accessibility Tree** to do the following:

1. Add semantics
2. Modify existing semantics
3. Provide extra labelling
4. Provide extra descriptions
5. Establish relationships
6. Inform ATs of different states
7. Inform ATs of live updates

ARIA only modifies the Accessibility Tree. It **does not**:

1. Modify an element’s appearance
2. Modify the element’s behaviour
3. Add focusability
4. Add keyboard event handlers

Roles, States and Properties

ARIA attributes are broken down into **roles, states and properties**.

Role attributes allow us to inform Assistive Technologies what type of widget it is.

Is it a menu, slider, progress bar? Does it provide the structure of a web page?

```
<!-- defining a widget -->
<ul role="menu">
    ...
</ul>

<!-- defining some aspect of the page structure -->
<main role="main">
    ...
</main>
```

State attributes allow us to inform Assistive Technologies what the current state of the widget is.

Is it checked, disabled, something else?

```
<!-- defining different states-->
<input aria-disabled="true" type="text">

<input aria-checked="true" type="radio">
```

Property attributes allow us to inform Assistive Technologies the purpose of the element, or whether it has a relationship to other elements.

What is it? Does it interact with other elements?

```
<!-- describe a relationship -->
<input aria-describedby="format"
type="text" name="a" id="a">
<span id="format">
  (must be mm/dd/yyyy)
</span>

<!-- providing additional labelling -->
<button aria-label="Close and return to application">
  Close
</button>
```

State vs Property?

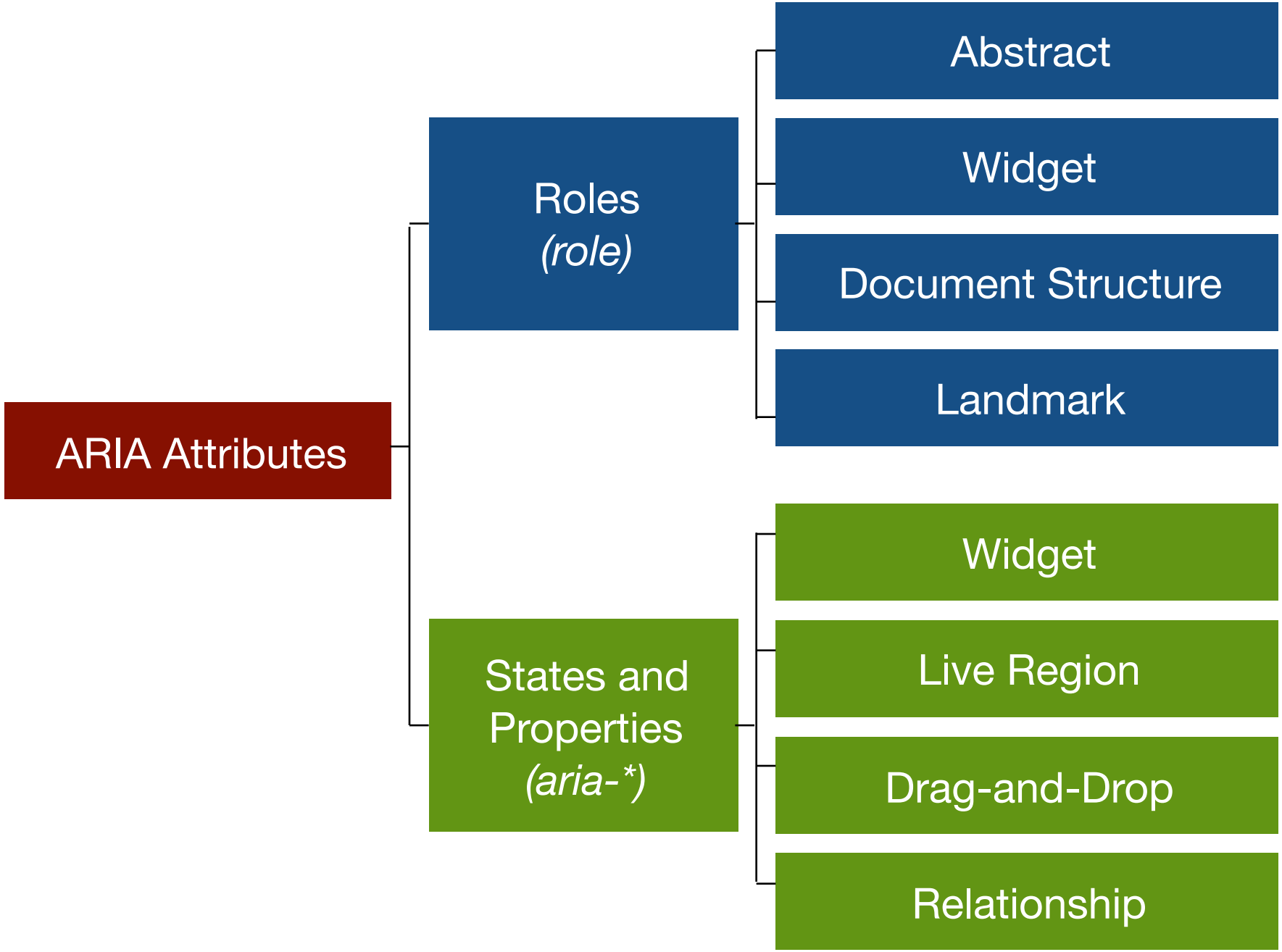
The terms “states” and “properties” refer to similar features. However, there are some **subtle differences** in their meaning.

States (such as `aria-checked`) may change frequently depending on user interaction.

Properties (such as `aria-labelledby`) very rarely change.

To avoid confusion, the WAI-ARIA specifications refer to “states” and “properties” simply as **“attributes”** whenever possible.

In reality, all of ARIA is just **HTML attributes**.


ARIA Attributes
Roles
(role)
Abstract
Widget
Document Structure
Landmark
States and
Properties
(aria-*)
Widget
Live Region
Drag-and-Drop
Relationship

ARIA support?

Browsers that support ARIA:

Firefox 3+
Internet Explorer 8+
Safari 4+ (Mac)
Chrome 17+

Assistive Technologies that support ARIA:

JAWS 8+ (Win)
Windows Eyes 5.5+ (Win)
ZoomText
VoiceOver (OS X/iOS)
NVDA (Win)
ORCA (Linux)

Keep in mind that **“support” is a general term**. Each browser/Assistive Technology has it’s own minor quirks and inconsistencies.

Also, the ARIA specification is constantly evolving, so the concept of “support” is a shifting landscape.

When in doubt, test!

Exercise 1:
Fixing a fake checkbox

Open **exercise01/start.html** in a browser and also in a text editor.

We'll use an example where a developer may use a `<div>` element **instead of a checkbox**.

(Let's not get bogged down discussing "why" anyone would do such a thing at this point. It can and does happen)

JavaScript and CSS have been used to **make the element look and operate like a checkbox** - at least to sighted, mouse users.

Native checkbox

- [x] Apples
- [ ] Bananas

Fake checkbox

- [x] Apples
- [ ] Bananas

```
<!-- Native checkbox -->
<form action="#">

  <div>
    <input type="checkbox" id="e1" checked>
    <label for="e1">Apples</label>
  </div>

  <div>
    <input type="checkbox" id="e2">
    <label for="e2">Bananas</label>
  </div>
```

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div class="checkbox checked">Apples</div>

  <div class="checkbox">Bananas</div>

</div>
```

However, there is no semantic meaning associated with the `<div>` element, so Assistive Technologies have **no way of understanding it's purpose** and then conveying this to the user.

Assuming we **cannot change** the `<div>` element, how could we (1) add some functionality for keyboard users and (2) add some additional semantic meaning for Assistive Technologies?

Step 1:

Adding `tabindex` with a value of `"0"` makes the element able to receive focus without changing the overall tab order of the page.

(This has nothing to do with ARIA but it is very important for keyboard only users.)

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div tabindex="0" class="checkbox checked">Apples</
div>

  <div tabindex="0" class="checkbox">Bananas</div>

</div>
```

Step 2:

Adding `role="checkbox"` makes sure the element is announced as a checkbox to Assistive Technologies.

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div tabindex="0" role="checkbox" class="checkbox
checked">Apples</div>

  <div tabindex="0" role="checkbox"
class="checkbox">Bananas</div>

</div>
```

Step 3:

Adding `aria-checked="true"` to the first fake checkbox means that it's checked state will be announced to Assistive Technologies.

```
<!-- Fake checkbox -->
<div class="checkbox-container">

  <div tabindex="0" role="checkbox" aria-
checked="true" class="checkbox checked">Apples</div>

  <div tabindex="0" role="checkbox"
class="checkbox">Bananas</div>

</div>
```

Keeping in mind that using a `<div>` here is very poor practice, we have now managed to **make the widget slightly more accessible**.

We used ARIA attributes to **change some nodes in the accessibility tree** so that they have a meaningful role and state.

More importantly, adding these ARIA attributes **has not changed anything** about the the appearance or on-screen behaviour of these elements.

However, there are **still one major issue**. One that we cannot resolve using ARIA. Can anyone find it?

Bad ARIA?

Over the last few years there has been a growing trend for developers to include ARIA attributes in **all sorts of areas in applications**.

Unfortunately, **ARIA attributes can be misused**, and this can lead to all sorts of problems for Assistive Technologies.

Issues with Redundancy

ARIA is sometimes used by “over-zealous” developers to provide additional ARIA attributes to native HTML elements **that already have accessible APIs**.

This can lead to **problems for Assistive Technologies**, such as the role of an element being announced more than once.

```
<!-- Do not do this -->
<input type="radio" role="radio">

<!-- This is preferred -->
<input type="radio">
```

```
<!-- Do not do this -->
<label for="name">Name</label>
<input id="name" type="text"
  required aria-required="true">

<!-- This is preferred -->
<label for="name">Name</label>
<input id="name" type="text" required>
```

Issues with being too verbose

There are times when developers use ARIA attributes to provide **detailed additional information** for Assistive Technologies.

This can sometimes lead to **excessive amounts of information** being presented to AT users.

```
<!-- Do not do this -->
<label for="email">Email</label>
<input id="email" type="email"
  aria-describedby="tip">
<span id="tip">Tip: Always include an AT
symbol as part of the email address so
that it is valid.</span>
```

Issues with copy and paste

This is where developers simple copy chunks of code from existing pattern libraries, **without understanding how the ARIA attributes in this code work**.

This can lead to problems such as `aria-label` attributes pointing to non-existent `ID` attributes, **so the ARIA does not work**.

```
<!-- Do not do this -->
<div role="dialog" aria-labelledby="modalLabel">
  <h5>Modal title</h5>
</div>

<!-- This is preferred -->
<div role="dialog" aria-labelledby="modalLabel">
  <h5 id="modalLabel">Modal title</h5>
</div>
```

Rules of using ARIA

Rule 1

If you can use a native HTML element or attribute with the semantics and behaviour you require already built in, **then do so**.

```
<!-- Do not do this -->
<div role="button">...</div>

<!-- This is preferred -->
<button>...</button>
```

Rule 2

Do not change native semantics,
unless you really have to.

```
<!-- Do not do this -->
<h2 role="tab">heading tab</h2>

<!-- This is preferred -->
<div role="tab">
  <h2>heading tab</h2>
</div>
```

Rule 3

All interactive ARIA controls **must be usable with the keyboard**.

If you create a widget that a user can click or tap or drag or drop or slide or scroll, a user **must also be able to navigate to the widget and perform an equivalent action** using the keyboard.

Rule 4

Do not use `role="presentation"` or `aria-hidden="true"` on **visible focusable elements**.

```
<!-- Do not do this -->
<button role="presentation">press me</button>
<button aria-hidden="true">press me</button>
```

Rule 5

All interactive elements **must have an accessible name**.

An interactive element **only has an accessible name** when its Accessibility API accessible name (or equivalent) property has a value.

```
<div role="region" aria-label="weather portlet">
  ...
</div>
```

Exercise 2: Providing extra context for buttons

Imagine you have a modal window and in the top right corner there is a `<button>` element with an **“X” symbol**.

Or, in the middle of a banking application screen there is a `<button>` element with the words **“Delete”**.


Delete

These two `<button>` elements are meaningful for sighted users as there are **visual clues nearby** that provide additional context.

But what about a Screen Reader user who uses the `TAB` keystroke and focuses on either of these `<button>` elements **without any additional context**?

Ideally, we should provide these `<button>` elements with additional context so that Screen Reader users **can understand their purpose**.

“Close and return to Account details”

“Delete this Personal Savings Account Transaction”

We can do this by **applying** the `aria-label` attribute to both of the `<button>` elements.

The `aria-label` value will provide a hidden label that is only available to Assistive Technologies. This value will **override the actual button content**.

```
<button type="button" aria-label="Close and return to
account details">
  x
</button>

<button type="button" aria-label="Delete this
Personal Savings Account Transaction">
  Delete
</button>
```

The `aria-live` attribute can be used to inform screenreader users when an area of content has been **dynamically updated**.