NDC – WEB COMPONENTS

<https://www.webcomponents.org/> is the official site for the Web Components and defines web components as:

“Web components are a set of web platform APIs that allow you to create new custom, reusable, encapsulated HTML tags to use in web pages and web apps. Custom components and widgets build on the Web Component standards, will work across modern browsers, and can be used with any JavaScript library or framework that works with HTML.

Web components are based on existing web standards. Features to support web components are currently being added to the HTML and DOM specs, letting web developers easily extend HTML with new elements with encapsulated styling and custom behaviour.”

Browsers use a class called HTMLElement rom which HTML tags are formed. The HTMLElement interface represents any HTML element. Some elements directly implement this interface, while others implement it via an interface that inherits it.

Web Components also extend the HTMLElement class and have the same properties and methods. This is outlined in the Mozilla Developer Network, (MDN), at <https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>.

We can create our own custom HTML tag, for example <my-component></my-component>. It must contain at least one hyphen in the name so that the browser knows it is a custom element and to avoid a clash with any future HTML tag that may be introduced. Self-closing tags such as <my-component /> are not allowed in the specifications but some modern browsers to allow it. It is best to stick to the rule of no self-closing tags.

Web Components have been around a long time, since 2013, but as only a few browsers supported it, they were not widely used.

<https://www.webcomponents.org/> shows the current support as June 2020.

Polyfills are also available so that the Web Components produced work in all browsers.

A screenshot of a cell phone

Description automatically generated

Web Components can be used as we would any regular HTML tag. We can add properties liked id, class etc and add event listeners too.

In this workshop, we will learn how to create Web Components that encapsulate code and functionality that can be reused anywhere in HTML.

There is so much more we can do though. We can make a component a mini application if we want to and we have various other abilities not used in the above code.

However, good component design follows some or all of these principles, based on an article at <https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components>

1. [Address a common need.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#common-need)
2. [Do one job really well.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#one-job-really-well)
3. [Work predictably in a wide variety of circumstances.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#work-predictably)
4. [Be useful right out of the box.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#immediately-useful)
5. [Be](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#composable) able to handle content placed in the component tag by the user.
6. [Be styleable.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#styleable)
7. [Be](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#extensible) extensible allowing a user the option to build on it.
8. [Think small.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#think-small)
9. [Adapt to the user and device.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#user-and-device)
10. [Deliver the key benefit to HTML authors, not just coders.](https://github.com/basic-web-components/components-dev/wiki/Ten-Principles-for-Great-General-Purpose-Web-Components#html-authors)

As we progress through this book, we will see how each of these principles can be performed.

In the Code Samples in the Boilerplates folder, you will see examples of useful components that carry out certain functions.

**TRAINING:**

<https://www.manning.com/books/web-components-in-action> BOOK (very in depth)

<https://training.coryrylan.com/> BOOK and VIDEO

Academind – Web Components and Stencil.js on Udemy (wait for sale and around $15USD) VIDEO

<https://open-wc.org/> SITE

<https://developers.google.com/web/fundamentals/web-components> SITE

**COMPILERS:**

Not frameworks! They compile JS files.

* Ionic and Stencil.js
* Lit-Element
* Svelte.js

**FRAMEWORKS:**

Angular and Vue can export their apps as web components but they also need the js library on the page so they will be larger in size:

<https://vuejsdevelopers.com/2018/05/21/vue-js-web-component/>

A close up of a screen

Description automatically generated

<https://www.npmjs.com/package/react-web-component>

<https://medium.com/@IMM9O/web-components-with-angular-d0205c9db08f>

**CORE CONCEPTS**

There four core concepts to Web Components:

1. Custom Elements
2. Templates
3. Shadow Dom
4. JavaScript Modules

Originally, the fourth was HTML Imports where one could link in an HTML file as follows:

<link rel="import" href="import-file.html">

However, there was not universally vendor support so it was dropped from the specification as some vendors believed that JS Modules would be a better replacement.

Over the next few chapters, we will look at each of these. It will be a high-level ‘flyby’ that can show what Web Components can do. This will give is the groundwork to look at readymade components to be used in our applications. In Part Three when we build useful rather than instructional components, we will see in more detail how these core components work.

**01-Custom-Elements**

**What is a Custom Element?**

A Custom Element use the Custom Elements API to create our own HTML tag. It extends the HTMLElement Class of the browser so is a browser feature.

**What is a Web Component?**

Although Custom Elements and Web Components are used interchangeably, a Web Component uses a Shadow DOM, an HTML fragment that is encapsulated with certain rules that allow two-way communication between it and the rest of the page (the Light DOM). We will see this in a later section.

**Naming rules**

It must have at least one dash, (-), in the name so that the browser knows it is Custom Element and to prevent any future clash with new HTML Elements.

Thus <my-component></my-component> is a valid Custom Element. The tags cannot be self-closing like <br /> although modern browsers are allowing this.

**Script file**

Each Custom Element will have an associated JavaScript file or <script> block in the HTML page.

**Creating a Custom Element**

**01-create-custom-element.html**

**A screenshot of a cell phone

Description automatically generated**

We can see that the Custom Element is just a regular HTML tag except it has a dash in its name.

The JavaScript (JS) code is shown. It is a regular JS Class and there are several lifecycle events:

A screenshot of a social media post

Description automatically generated

In this example the main lifecycle event is connectedCallback() which occurs when the custom element is added to the DOM.

Some things can be declared in the constructor and some things must be declared in the connectedCallback lifecycle event when it needs to refer to the DOM, e.g. this.innerHTML will only work when the component is in the DOM.

This can be a useful way to have an HTML include file (import) which was part of the original specification but not adopted by all browser vendors as some felt JS modules would do the same job.

**02-simple-css-light.html**

We can see that we can use the custom tag and style it with regular CSS. All the rules of CSS still apply. Think of the custom tag as a <div> tag.

**03-on-event.html**

We can add events directly to the element or by adding an event listener.

**04-attributes.html**

We can add our own custom attribute names to the tag which can then be accessed by JavaScript (JS).

**05-define-component-name-page.html**

We can also allow a user to define their own name for the tag.

A screenshot of a cell phone

Description automatically generated

**02-TEMPLATE**

**01-template.html and 02-template-loaded.html** shows examples of how the template tag is inert until added to the DOM programmatically.

In future examples we will see that the template tag can be inside or outside of the class.

The Template API allows us to create neutral HTML templates in our document.

1. <template>
2. <h1>This is content in template tag.</h1>
3. </template>

What is a document fragment?

The DocumentFragment interface represents a minimal document object that has no parent. It is used as a lightweight version of [Document](https://developer.mozilla.org/en-US/docs/Web/API/Document) that stores a segment of a document structure comprised of nodes just like a standard document. The key difference is that because the document fragment isn't part of the active document tree structure, changes made to the fragment don't affect the document, cause [reflow](https://developer.mozilla.org/en-US/docs/Glossary/reflow), or incur any performance impact that can occur when changes are made.

A common use for DocumentFragment is to create one, assemble a DOM subtree within it, then append or insert the fragment into the DOM using [Node](https://developer.mozilla.org/en-US/docs/Web/API/Node) interface methods such as [appendChild()](https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild) or [insertBefore()](https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore). Doing this moves the fragment's nodes into the DOM, leaving behind an empty DocumentFragment. Because all of the nodes are inserted into the document at once, only one reflow and render is triggered instead of potentially one for each node inserted if they were inserted separately.

This interface is also of great use with Web components: [<template>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template) elements contain a DocumentFragment in their [HTMLTemplateElement.content](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTemplateElement/content) property.

An empty DocumentFragment can be created using the [document.createDocumentFragment()](https://developer.mozilla.org/en-US/docs/Web/API/Document/createDocumentFragment) method or the constructor. <https://developer.mozilla.org/en-US/docs/Web/API/DocumentFragment>

**The TEMPLATE tag of HTML5**

One of the new features of HTML5 is the <template> tag.

The MDN specifications states that ‘when you have to reuse the same mark-up structures repeatedly on web page, it makes sense to use a template rather than repeating the same structure over and over again.’

To be able to show the content we need to run the following code:

1. document.addEventListener(‘DOMContentLoaded’, () => {
3. let temp = document.getElementById(‘template01’);
4. let content = temp.content;
5. console.log(content);
6. document.body.appendChild(content);
7. });

All element tags in the DOM are nodes and carriage returns are also nodes. If we list out all the nodes, we may find extra ones due to carriage returns being nodes in the DOM.

In our Web Components, we can make use of the <template> tag and add it when needed. We see the use of <template> tags in the JS frameworks like Vue etc. It is faster than outputting a string to the page.

It is supported in all major browsers, but it can be tested as follows:

1. **if** (document.createElement(‘template’).content) {
2. // if true it is supported
3. } **else** {
4. //not supported so alternate code must be used
5. }

When we work with the ready made components, we will see the use of the template tag more fully.

**03-SHADOW-DOM**

**What is it?**

Shadow DOM provides:

1. Isolated DOM. The web component is encapsulated meaning that a document.querySelector() won’t return nodes in the component’s shadow DOM.
2. The CSS is scoped meaning it won’t affect the Light DOM.

When the browser loads a page, it transforms the HTML into a live document by creating a tree of nodes.

We can add nodes/elements with JavaScript and we saw how we could create a template tag and add it at run time.

This can be shown diagrammatically (image courtesy MDN):

A close up of a logo

Description automatically generated

The shadow dom has a shadow tree with a shadow root that can be appended to a node in the light dom.

**Can it access Light DOM?**

Yes. As we see in **01-shadow-dom.html** we can access the Light DOM in the Shadow DOM.

**Can Light DOM access HTML of Web Component?**

No. Shadow DOM elements are not visible to querySelector from the light DOM. Shadow DOM elements may have ids that conflict with those in the light DOM. They must be unique only within the shadow tree.

More detailed article on accessing the Shadow DOM via ‘back door’ approaches is in [**Appendix 01**](#appendix01)

**01-shadow-dom-template.html**

This is one of the boiler plates we will use to create Web Components

SLOTS and Named SLOTS

**03-default-slot.html**

This show how we can use the slot. If we don’t pass in any content yet have a <slot> element in our component, then it will use the default text.

**04-named-slot.html**

We can define named slots in our component and bind them with slots in the Light DOM.

NOTE: We have not yet covered CSS methods but the Light DOM always trumps Shadow DOM CSS. In this example, we see this with the slot=first, where the component has a CSS rule of .slot[name=”first”] but it is overruled by the Light DOM, making it red.

**05-slotchange-event.html**

We can access the slot nodes. CR is considered a node.

This could be one way to communicate data between Light DOM and component but not usually used.

Additional notes on slots:

<https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_templates_and_slots>

There are two kinds of slots:

* Named slots: <slot name="X">...</slot> – gets light children with slot="X".
* Default slot: the first <slot> without a name (subsequent unnamed slots are ignored) – gets unslotted light children.
* If there are many elements for the same slot – they are appended one after another.
* The content of <slot> element is used as a fallback. It’s shown if there are no light children for the slot.

The process of rendering slotted elements inside their slots is called “composition”. The result is called a “flattened DOM”.

Composition does not really move nodes, from JavaScript point of view the DOM is still same.

JavaScript can access slots using methods:

* slot.assignedNodes/Elements() – returns nodes/elements inside the slot.
* node.assignedSlot – the reverse method, returns slot by a node.

If we’d like to know what we’re showing, we can track slot contents using:

* slotchange event – triggers the first time a slot is filled, and on any add/remove/replace operation of the slotted element, but not its children. The slot is event.target.
* [MutationObserver](https://javascript.info/mutation-observer) to go deeper into slot content, watch changes inside it.

Now, as we know how to show elements from light DOM in shadow DOM, let’s see how to style them properly. The basic rule is that shadow elements are styled inside, and light elements – outside, but there are notable exceptions.

**04-CSS**

There are five major ways to style the component

1. Light CSS on the component tag – inheritable properties do pass through by default.
2. Pass CSS Variables into Shadow DOM.
3. ::slotted(selector).
4. :host(selector).
5. :host(context).

We will now look at each way to see how we can enable the user to add their own styling tour components.

Inheritable properties of CSS do pass into component.

This is an overall summary of how we can use CSS

A screenshot of a cell phone

Description automatically generated

**01-css-01-light.html**

If we have not Shadow DOM then the normal CSS rules apply as the Web Component is a regular element, even if it has been custom made.

SLOTS and Named SLOTS

**css-02-custom-properties-shadow.html**

Using Custom Variables we can allow the Light DOM (user) to alter the styling based on where we have used Custom CSS variables. This method is best used when the user does not know the internal structure of the component and we have specified CSS options for the user.

**css-03-default-slot.html**

This shows the use of the ::slotted() selector.

NOTE:

1. Light DOM trumps Shadow DOM. We saw that in 03-SHADOW-DOM/04-named-slot.html.
2. This selector is only for the top-level element in the slot. Nested elements are not styled as we can see in this code example **css-03-default-slot.html**.

**css-04-host.html**

We cannot access the host in the Web Component as it will not have been added to the Light DOM.

We can use the :host() selector as in this example.

The exercise will show how adding class or id attributes to the host element we can style the component using :host(<css selector>).

We can also use :

host([cool]) <my-component cool></my-component>

host([cool=’yes’]) <my-component cool=’yes’></my-component>

**css-05-host-context.html**

This style rule allows the Web Component to be styled depending on its lexical placing in the Light DOM. The exercise will demonstrate this.

The former shadow CSS rule has been deprecated.

New CSS rules like ::part and accessing a global stylesheet are in the pipeline.

Justin Fagnani, a long-time Polymer Project team member, will talk all about styling and theming web components. This is a critical topic for anyone building reusable web components.

<https://www.youtube.com/watch?v=FM7ROEVPA4k&t=1396s>

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A close up of a sign

Description automatically generated

A screenshot of a cell phone

Description automatically generated

@import considered an anti-pattern as they download sequentially and not in parallel.

A screenshot of a cell phone

Description automatically generated

<https://developers.google.com/web/updates/2019/02/constructable-stylesheets>

CSS Modules is a new proposal.

The future…?

A screenshot of a cell phone

Description automatically generated

**04-EVENTS**

PARENT

CHILD02

CHILD01

**red = custom event blue = set attributes on child**

Child components communicate with Parent by sending a Custom Event with the detail property containing the payload.

Parent communicates with child by setting attributes.

Siblings communicate by sending event to parent which then communicates this to the sibling.

The reason for this design pattern is that a Web Component is deployed without dependencies on other components. It does its job and emits events as needed. If two or more components are tied together, it may be more appropriate to combine them into one component.

However, as JS has no private classes and methods, there are other design patterns such as exposing methods and properties to outside elements.

We will cover the theory now although we will see more useful examples in the LIIBRARY section.

**01-light-events.html**

We can expose methods and properties to the Light DOM from within the Shadow DOM if we wish and we can also add regular events to the custom tag in the Light DOM just like any regular HTML tag.

**02-light-to-shadow-via-attribute-devtools.html**

In this hover tooltip component we can change the attribute ‘displaytext’ manually or programmatically.

In DEV > ELEMENTS, change the displaytext attribute to see this.

**01-ndc-show-post** is a very good example of this

**03-light-to-shadow-attribute-js.html**

This shows the attributeChangedCallback lifecyle event which is used to detect changes in attributes.

By default, components do not listen for attribute changes as this would be a performance load.

We must define what attributes to listen for in:

            static get observedAttributes() {

                return ['system'];

            }

**01-ndc-show-post** shows this well.

Reflection =>

From Web Components in Action, Manning

  attributeChangedCallback(name, oldValue, newValue) {

         if (name === 'checked')

            this.checked = newValue;

      }

      set checked(value) {

         const isChecked = Boolean(value);

         if (isChecked)

// OOPS! This will cause an infinite loop because

// it triggers the

// attributeChangedCallback() which then sets this property // again.

            this.setAttribute('checked', '');

         else

            this.removeAttribute('checked');

      }

// BETTER REFLECTION

// ATTRIBUTES AS SOURCE OF TRUTH

 static get observedAttributes() {

      return ['postid'];

   }

   get postID() {

      return this.getAttribute('postid');

   }

   set postID(val) {

      this.setAttribute('postid', val);

   }

   attributeChangedCallback(name, oldValue, newValue) {

      if (name === 'postid') {

         this.getPost();

      }

   }

**04-light-to-shadow-via-attribute-page.html**

This is a more visual example of this.

          this.dispatchEvent(new CustomEvent('childClick',

            {

              detail: 'Button clicked ' + this.\_currentCount +  ' times.<br>Child can trigger an event and send data to parent.',

              bubbles: true, // allows it to bubble up to top of child component where it can be heard in Light DOM

              composed: true // allows it to penetrate Shadow DOM and be heard in in tags outside of component

            }

            ));

By default, event bubbling is off in components and to penetrate the Light/Shadow boundary we need to add **composed: true**

**‘childClick’** event name is arbitrary and we can call the event whatever we like and listen for it in the Light DOM with usual event listeners.

**06-showAllPostsRange.html**

This is a more practical example that is also used in the library section. It can also show that every time any attribute is changed the AJAX request occurs. If we are setting two or more attributes at one time to carry out a single function, it is best to pass these parameters in once via an object in the attribute value, e.g. range=”{min:4, max: 10}”

**15-child-to-parent.html**

An example of Child -> Parent communication

**20-child01-parent-child02.html**

An example of Child01 -> Parent -> Child02 communication.

<https://coryrylan.com/blog/using-web-components-in-react-video-tutorial>

<https://coryrylan.com/blog/using-web-components-in-angular-video-tutorial>

<https://coryrylan.com/blog/using-web-components-in-vue>

**Appendix 01**

Terminology aside, the important takeaway is that we’re dealing with a new DOM

inside a document fragment. Unlike a document fragment used by the <template>

tag, however, this fragment is actually rendered in the browser, yet still maintains its

independence.

Once created, we can use the new and automatically created property of our component,

shadowRoot, to access any of our element’s properties, like innerHTML. This is

what we did in our example:

this.shadowRoot.innerHTML =

`<div class="inside-component">My Component</div>`

With just this change, we’ve now protected our component from accidental intrusions.

When we now run the same query selector and try to set the innerHTML, it fails:

document.querySelector('.inside-component').innerHTML +=

' has been hijacked';

Our error reads

*Uncaught TypeError: Cannot read property 'innerHTML' of null*

What happens now? Query-selecting our inside-component class comes up with nothing, and setting the innerHTML property is attempted on a null.

That’s because we’ve isolated the HTML inside our component with the

Shadow DOM.

***8.2.2 Closed mode***

Here’s the thing, though. If we wanted to be malicious, we still could be. The same

shadowRoot property is available from the outside. We could adjust our query selector

to be more complex and still set the innerHTML of that <div>:

document.querySelector('sample-component').shadowRoot.querySelector

Here, we’re showing JS that easily sets our component’s innerHTML. Can we stop those malicious users from coming in and manipulating our component in ways we don’t want? The answer appears to be no, but that’s where closed mode comes in.

To explain, let’s set mode to closed in the following listing.

connectedCallback () {

this.attachShadow({mode: 'closed'});

this.shadowRoot.innerHTML =

`<div class="inside-component">My Component</div>`

}

This won’t work as intended, however, without changing something else! With the

shadow root closed, the shadowRoot property doesn’t exist (it’s null), so we can’t set

the innerHTML through it. How, then, can we interact with our own component when

working from the inside?

The call to attachShadow does return a reference to the shadow root, whether

you’re in open or closed mode. If you only need a reference in the same function

where you created the shadow root, you can simply declare a variable, as follows.

Listing 8.3 Setting the shadow mode to **closed**

Figure 8.8 Attempting to query-select inside the Shadow DOM

.inside-component

querySelector('.inside-component')

(Rejected)

connectedCallback () {

const root = this.attachShadow(

{mode: 'closed'});

root.innerHTML = `<div class="inside-component">My Component</div>`

}

If that’s the only interaction point with your component’s Shadow DOM, problem

solved! You’ve taken steps to close off your component from malicious users . . .

except for one more thing. Let’s pretend we are malicious and will stop at nothing to

sabotage this component. We can change the function definition of attachShadow

after the component class is declared:

SampleComponent.prototype.attachShadow = function(mode) { return this; };

This is being very tricky indeed, but what we’ve done is change the attachShadow

function so that it doesn’t actually create a shadow root and instead does nothing but

pass back the Web Component’s natural scope. The original component creator, who

intended to create a closed shadow DOM, is not creating a shadow DOM at all. The

shadow root reference is what they were supposed to get back, but it ended up really

just being the component’s scope. This trickery still works the same because this, and

the shadow root, have approximately the same API.

And now we’re back to our original, easy way of taking over the component:

document.querySelector('.inside-component').innerHTML +=

' has been hijacked';

Should you expect people who use your component to try to break in in this way?

Probably not. But they could. It’s not real security because it’s so easily bypassed.

Recall at the start of this chapter when we talked about protecting your component

for real or doing so by convention. There, we discussed using the underscore to protect

private variables and methods in your class instead of using more secure ways.

Here, it’s the same thing, but instead of variable and methods, we’re talking about

your component’s DOM.

That’s why Google’s own documentation on Web Components says you shouldn’t use closed mode (<https://developers.google.com/web/fundamentals/web-components/shadowdom#closed>).

A screenshot of a social media post

Description automatically generated

You’re closing off the Shadow DOM to make it secure, but you’re trusting

that the folks who use your component won’t bypass it in some very simple ways. In

the end, you’re protecting your component by convention regardless of what you do;

it’s just that closed mode makes it more difficult to develop with.

Google claims that closed mode will make your component suffer for two reasons.

The first is that by allowing component users into your component’s Shadow DOM

through the shadowRoot property, you’re at least making an escape hatch. Whether

you’re making private class properties with underscores or keeping the Shadow DOM

open, it’s protecting your class or component by convention.

Listing 8.4 Using a variable to reference the shadow root

**Sets a variable to the newly**

**created shadow root**

**Licensed to Craig West <mrcraigwest@hotmail.com>**

**176** CHAPTER 8 ***The Shadow DOM***

Despite your best intentions for your component, you likely won’t accommodate

all use cases all the time. Having a way into your component allows some flexibility,

but it’s also important to recognize that this way in goes against your better judgement

as a component developer. It’s a signal to the developer who uses your component

that they should do so at their own risk. That’s ill-advised, of course, but when deadlines

are tight, and a web app needs to be shipped tomorrow, it’s nice to provide a

path forward with an open Shadow DOM using the shadowRoot property to access

things you don’t intend to be accessed at present. You’ll also see that an escape hatch

with the open mode is rather nice for reaching in to perform automated testing, as

we’ll discuss in chapter 13.

Google’s second gripe with closed mode is the claim that it makes your component’s

Shadow DOM inaccessible from inside your own component. But it’s more

complicated than that. The shadowRoot property is no longer available in closed

mode, but we can easily make a reference to it.

Our current example has a locally scoped variable in the next listing.

connectedCallback() {

const root = this.attachShadow(

{mode: 'closed'});

root.innerHTML = `<div class="inside-component">My Component</div>`

}

Now let’s change it to having a property on your class.

connectedCallback () {

this.root = this.attachShadow(

{mode: 'closed'});

this.root.innerHTML = `<div class="inside-component">My Component</div>`

}

On the other hand, making it a public property defeats the purpose. Again, you’re

back to having a public reference to the Shadow DOM; it just happens to be named

root (or any property name you choose) instead of the shadowRoot property, as created

by an open Shadow DOM. And again, it’s easy to access your component’s DOM

through it. That said, if you did use a stronger way of protecting your class properties,

like using Weak Maps to make your properties private, it’s still wouldn’t be foolproof,

but it would close things off pretty well and allow internal access to your closed DOM

just fine. It might be worth speculating that a truly closed Shadow DOM might be

achievable once we have native private class fields available in all browsers, but we just

aren’t there yet.

It’s clear that a closed Shadow DOM isn’t worth the trouble for most cases. There is

no bulletproof way to completely lock down your component, and protecting your

component by convention using the open Shadow DOM is the way to go.

Listing 8.5 Locally scoped shadow root variable

Listing 8.6 A public property containing the shadow root