

# Web Components

R. Mark Volkmann

Object Computing, Inc.



<https://mvolkmann.github.io/blog>



r.mark.volkmann@gmail.com



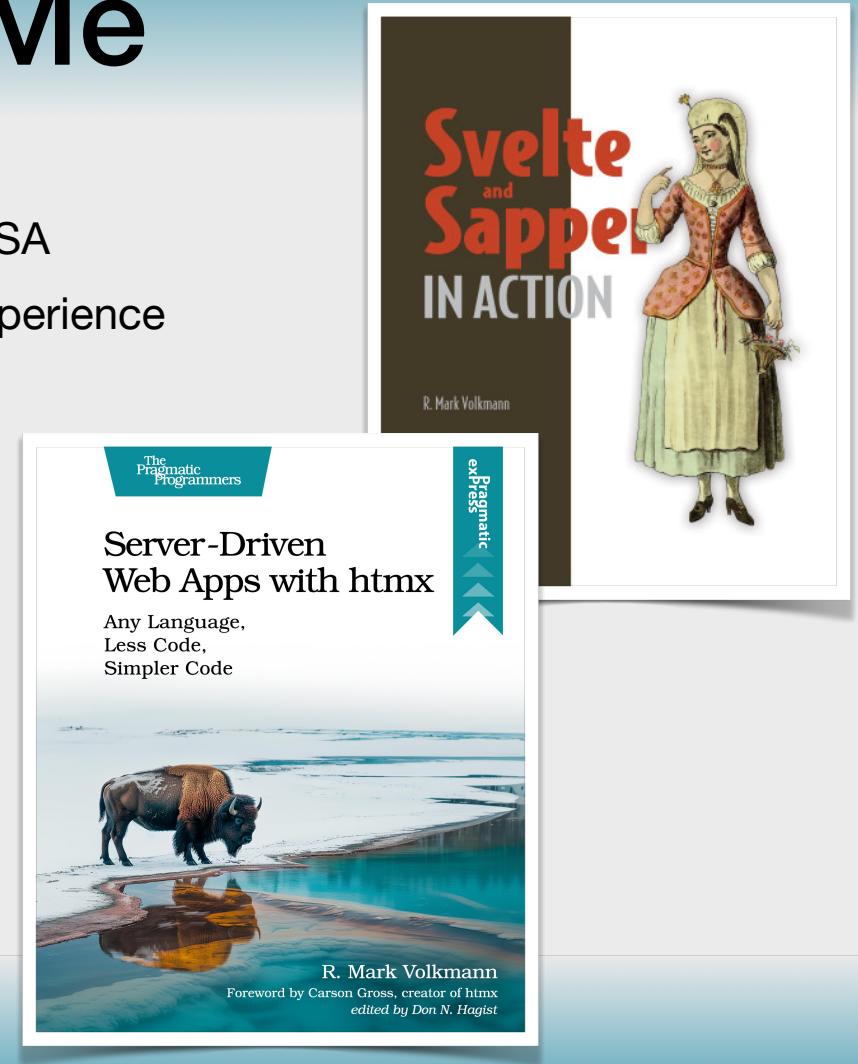
OBJECT COMPUTING  
YOUR OUTCOMES ENGINEERED

Slides at <https://github.com/mvolkmann/talks/>



# About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 44 years of professional software development experience
- Writer and speaker
- **Blog** at <https://mvolkmann.github.io/blog/>
- Author of Manning book “**Svelte ... in Action**”
- Author of Pragmatic Bookshelf book “**Server-Driven Web Apps with htmx**”



# Web Components



- Define **custom HTML elements** that can be used like standard HTML elements
- Names must be all lower-case and contain at least one hyphen
- Tags cannot be self-closing

```
ex. <radio-group></radio-group>, not <radio-group />
```

# Pros



- Standard way to implement web-based UI components
- Just use the “platform”
  - no libraries, tooling, or build process required
- Use in any web page, with any web framework, and in Markdown files
  - share between projects using different frameworks
- Shadow DOM provides CSS and DOM encapsulation
  - styles defined in component do not affect HTML outside it
  - can control ability to style component from outside it
  - can prevent code outside component from querying and modifying its DOM

Some issues remain when using in React-based frameworks, mainly due to use of virtual DOM.

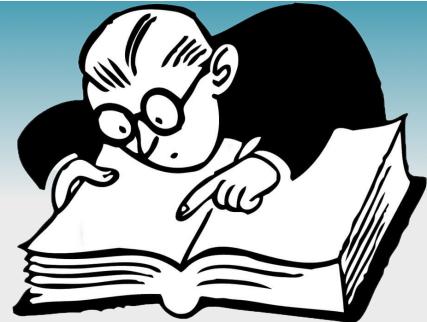
# Cons



- Learning curve, but relatively small
  - mostly learning more about the DOM, which is good to know
- Requires **more effort and code** than implementing components using a modern web framework
  - worthwhile for components that may someday be used in multiple apps that use multiple frameworks
  - not a great developer experience compared to using a framework
- Reactivity must be implemented manually
  - unless using a library like wrec
- Server-side rendering is currently challenging
  - Do you need it?

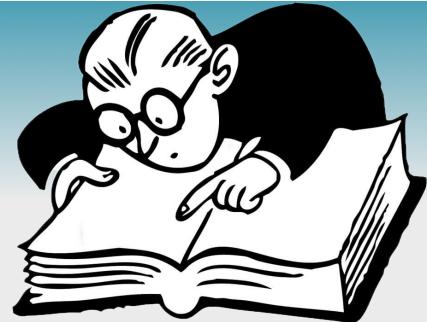
unless using a library like Lit

# Standards Based ...



- **Custom Elements**
  - enables defining custom HTML elements by creating a class that extends `HTMLElement` class or one of its subclasses
  - supports several lifecycle methods
  - each class defines a reusable component that has attributes, properties, and methods
- **Shadow DOM**
  - encapsulates styling and DOM of custom HTML elements
  - adds `attachShadow` method, `ShadowRoot` class, and `shadowRoot` property
  - supports “open” and “closed” modes more on these later

# ... Standards Based



- **HTML Templates**
  - adds HTML element `template` defined by `HTMLEmplateElement` class
  - `content` property is an instance of `DocumentFragment` that can be created `without rendering`
  - can be efficiently cloned and rendered later
- **ES Modules** (a.k.a. JavaScript Modules)
  - adds `import` and `export` syntax, module scope, asynchronous loading, and more
- **CSS Scoping**
  - defines scoping/encapsulation mechanisms for CSS, including scoped styles and the `@scope` rule, Shadow DOM selectors, and page/region-based styling
- **CSS Shadow Parts**
  - defines `::part()` pseudo-element on shadow hosts, allowing shadow hosts to selectively expose chosen elements from their shadow tree to outside page for styling

# Implementation Options



- **Vanilla** from W3C
- **wrec** from me (more on this later)

No tooling or build process is required.

- 
- **Lit** from Google
    - most popular web components library
    - can be used without tooling and a build process, but isn't typically
  - **Stencil** from Ionic
  - **FAST** from Microsoft

These require using tooling in a build process.

The name “**Lit**” is based on 3 things:

- it’s a **little** library
- it uses JS template **literals**
- it is “**lit**”, as in amazing

# Shadow DOM



- Encapsulates styling and content of a custom element
- Many standard HTML elements utilize a shadow DOM
  - `input`, `textarea`, `select`, `audio`, `video`, and more
- To use a shadow DOM in a web component

```
constructor() {  
  this.attachShadow({mode: 'open'});  
}
```

```
constructor() {  
  this.#root = this.attachShadow({mode: 'closed'});  
}
```

Inspect these elements in Chrome DevTools to see `#shadow-root` and their contents. This requires checking Settings ... Preferences ... Elements ... **Show user agent shadow DOM**. The `#shadow-root` of custom elements is always visible.

When mode is “closed”, capture shadow root in a private property so it can be accessed in `connectedCallback`.

- To add content

```
connectedCallback() {  
  this.shadowRoot.innerHTML = 'some HTML string';  
  // OR  
  this.shadowRoot.appendChild(someElement);  
}
```

```
connectedCallback() {  
  this.#root.innerHTML = 'some HTML string';  
  // OR  
  this.#root.appendChild(someElement);  
}
```

# Shadow DOM Access



- DOM provides many methods that find elements including `querySelector`, `querySelectorAll`, and `getElementById`
- Whether these methods can be used to find and manipulate elements in a web component depends on whether/how it uses a shadow DOM
  - see options on next slide

# DOM Access Options

- **No shadow DOM**
  - document **can** query into web component DOM
- **Open shadow DOM**
  - document **cannot accidentally** query into web component DOM
  - **can intentionally** query by calling methods on **shadowRoot** property
  - useful for implementing tests with tools like Playwright and Cypress
- **Closed shadow DOM**
  - document **cannot** query into web component DOM
  - **shadowRoot** property is not present
  - DevTools can still view and manipulate the elements



holds instance of  
**ShadowRoot** class which inherits  
from **DocumentFragment** class



# Shadow DOM Example

- Suppose a document contains an instance of **radio-group** web component that uses shadow DOM in “open” mode
  - this **does not** find **label** elements inside web component

```
const labels = document.querySelectorAll('label');
```

- this **does** find **label** elements inside web component

```
const labels = document.querySelector('radio-group').shadowRoot.querySelectorAll('label');
```

- this changes the **label** of first radio button inside web component

```
document
  .querySelector('radio-group')
  .shadowRoot
  .querySelector('label')
  .textContent = 'Rouge'
```

# Show Me Code!

- Simple web component

```
<hello-world></hello-world>
```

```
<hello-world name="Mark"></hello-world>
```

Hello, World!

Hello, Mark!

- We'll see five ways to implement

- vanilla with no shadow DOM, setting innerHTML
- vanilla with no shadow DOM, using DOM API
- vanilla with shadow DOM, setting innerHTML
- vanilla with shadow DOM, using DOM API
- using Lit (a bit later)

Each requires  
a small amount of code  
that is easy to understand.



# Starting Simple

```
<html>                                         index.html
  <head>
    <script src="hello-world1.js" type="module"></script>
    <script src="hello-world2.js" type="module"></script>
    <script src="hello-world3.js" type="module"></script>
    <script src="hello-world4.js" type="module"></script>
  </head>
  <body>
    <hello-world1 name="innerHTML"></hello-world1>
    <hello-world2 name="appendChild"></hello-world2>
    <hello-world3 name="shadow-innerHTML"></hello-world3>
    <hello-world4 name="shadow-appendChild"></hello-world4>
  </body>
</html>
```

```
class HelloWorld1 extends HTMLElement { hello-world1.js
  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    this.innerHTML = `<p>Hello, ${name}!</p>`;
  }
}
customElements.define("hello-world1", HelloWorld1);
```

```
class HelloWorld2 extends HTMLElement { hello-world2.js
  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    const p = document.createElement("p");
    p.textContent = `Hello, ${name}!`;
    this.appendChild(p);
  }
}
customElements.define("hello-world2", HelloWorld2);
```

None of these implement `attributeChangedCallback`, so modifying the `name` attribute in DevTools does not update the UI.

```
class HelloWorld3 extends HTMLElement { hello-world3.js
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    this.shadowRoot.innerHTML = `<p>Hello, ${name}!</p>`;
  }
}
customElements.define("hello-world3", HelloWorld3);
```

```
class HelloWorld4 extends HTMLElement { hello-world4.js
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const name = this.getAttribute("name") || "World";
    const p = document.createElement("p");
    p.textContent = `Hello, ${name}!`;
    this.shadowRoot.appendChild(p);
  }
}
customElements.define("hello-world4", HelloWorld4);
```

# innerHTML vs. DOM API

- Two approaches for building web component DOM
- Set `innerHTML`
  - typically using JavaScript template strings
  - less verbose
  - faster for bulk updates
  - destroys existing nodes and loses event listeners
  - builds DOM synchronously so it can be queried immediately
  - danger of XSS when using untrusted content
- Call **DOM API** methods
  - more fine-grained control
  - can preserve existing nodes and event listeners
  - avoids XSS from untrusted content

## Middle Ground

- set `innerHTML` to render initial DOM
- in event handlers, use DOM API to make targeted updates

# Template Strings



- JavaScript feature that can build a string of HTML for use as `innerHTML` value
- Capabilities include
  - conditional logic with ternary operator
    - one use is to conditionally add an attribute
  - iteration with `map` method
  - break up content rendering into multiple functions/methods

```
html`<input  
      type="radio"  
      id="${option}"  
      name="${this.#name}"  
      value="${option}"  
      ${option === this.value ? "checked" : ""}  
    />`
```

examples come from  
“Radio Group” component  
we will review later

```
html`<div class="radio-group">  
  ${options.map(option => this.#makeRadio(option)).join("")}  
</div>`
```

```
#makeRadio(option) {  
  return html`  
    ...  
  `;  
}
```

Preceding template string with tag `css` or `html` triggers  
**VS Code extension Prettier** to format code and  
**es6-string-html** extension to add syntax highlighting.



# Can AI Tools Generate?



- **Request:** Write a hello-world vanilla web component that has a name attribute with a default value of “World”.
- **Contenders:**
  - ChatGPT, Claude, Google Gemini, Microsoft Copilot, and Perplexity
- While the generated code uses slightly different approaches, each successfully generated code that meets the requirements
  - Google Gemini produced worst code

# Using Lit



```
package.json
{
  "name": "hello-world-web-components",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "lit": "^3.0.0"
  },
  "devDependencies": {
    "typescript": "^5.0.0",
    "vite": "^5.0.0"
  }
}
```

```
tsconfig.json
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "target": "ES2022",
    "useDefineForClassFields": false
  }
}
```

When using Lit, modifying the `name` attribute in DevTools automatically updates the UI!

```
index.html
<html>
  <head>
    <script src="hello-world5.js" type="module"></script>
    <script src="hello-world6.ts" type="module"></script>
  </head>
  <body>
    <hello-world5 name="Lit JS"></hello-world5>
    <hello-world6 name="Lit TS"></hello-world6>
  </body>
</html>
```

```
hello-world5.js
class HelloWorld5 extends LitElement {
  static properties = { name: { type: String } };

  render() {
    return html`<p>Hello, ${this.name} || "World"!</p>`;
  }
}
customElements.define("hello-world5", HelloWorld5);
```

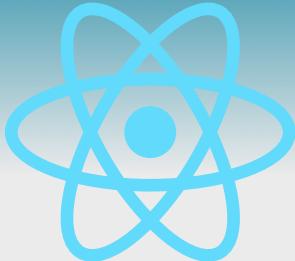
JavaScript  
version

```
hello-world6.ts
import { html, LitElement } from "lit";
import { customElement, property } from "lit/decorators.js";

@customElement("hello-world6")
export class HelloWorld6 extends LitElement {
  @property({ type: String }) name = "World";

  render() {
    return html`<p>Hello, ${this.name}!</p>`;
  }
}
```

TypeScript  
version



# React Comparison

- Steps to create
  - `npm create vite@latest react-hello-world`
    - select React and TypeScript
  - `cd react-hello-world`
  - `npm install`
  - create `src/HelloWorld.tsx` →
  - modify `App.tsx`
  - `npm run dev`
  - browse localhost:5173
- Component can only be used in React apps
- Lots of libraries and tools required

```
import React from "react";           HelloWorld.tsx

interface HelloWorldProps {
  name?: string;
}

const HelloWorld: React.FC<HelloWorldProps> =
  ({ name = "World" }) => {
    return <p>Hello, {name}!</p>;
};

export default HelloWorld;
```

```
import HelloWorld from "./HelloWorld";          App.tsx

function App() {
  return (
    <>
      <HelloWorld />
      <HelloWorld name="Mark" />
    </>
  );
}

export default App;
```

Hello, World!  
Hello, Mark!

# Including Web Components

- Include web components with `script` tags

```
<script src="hello-world.js" type="module"></script>
```

- Module benefits

- top-level variables are scoped to the module (useful when using `template` element)
- execution is deferred, same as when `defer` attribute is included
  - fetches in parallel, but delays execution until DOM is built
- uses strict mode for better error checking
- can use `import` and `export` keywords
- can use top-level `await`
- enforces Cross Origin Resource Sharing (CORS) restrictions
  - prevents testing web components with `file://` URLs
  - have to run a local server (like Vite which is described next)



# Vite Server

- Many options for starting an HTTP server that serves local files
- One option is to use Vite
  - has many other benefits including live reload
- Steps
  - make new directory and `cd` to it in a terminal window
  - create `package.json` file by entering `npm init`
  - install Vite by entering `npm i -D vite`
  - create `index.html`
  - start server by entering `npm run dev`
  - browse `localhost:5173`

If port is in use,  
Vite will try 5174, 5175, ...  
until it finds an open port.

minimal `package.json`  
after installing Vite

```
{ "name": "web-server", "scripts": { "dev": "vite" }, "devDependencies": { "vite": "^7.0.1" } }
```



# Web Component Libraries

- **Shoelace** - <https://shoelace.style/>
  - “forward-thinking library of web components”
- **Web Awesome** - <https://webawesome.com/>
  - “make something awesome with open-source web components”
  - eventual successor to Shoelace; still in beta
- **FAST** from Microsoft - <https://fast.design>
  - “dedicated to providing support for native Web Components and modern Web Standards”
- **Lion** - <https://lion.js.org/>
  - “fundamental white label web components for building your design system”





# Shoelace Example

```
<html>
  <head>
    <link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/@shoelace-style/shoelace@2.20.1/cdn/themes/light.css"
    />
    <style>
      sl-radio-group {
        margin-top: 1rem;
      }
      sl-radio-group::part(form-control),
      sl-radio-group::part(form-control-input) {
        display: flex;
        gap: 1rem;
      }
      sl-radio-group::part(form-control-label) {
        font-family: var(--sl-input-font-family);
        font-weight: bold;
      }
    </style>
    <script
      type="module"
      src="https://cdn.jsdelivr.net/npm/@shoelace-style/shoelace@2.20.1/cdn/shoelace-autoloader.js"
    ></script>
```

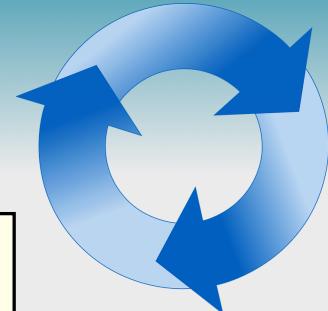
Happy?

Color:  Red  Green  Blue

```
<script>
  window.onload = () => {
    const slSwitch = document.querySelector("sl-switch");
    slSwitch.addEventListener("sl-change", (event) => {
      console.log("switch is",
        event.target.checked ? "on" : "off");
    });

    const slRadioGroup =
      document.querySelector("sl-radio-group");
    slRadioGroup.addEventListener("sl-change", (event) => {
      console.log("color =", event.target.value);
    });
  }
</script>
</head>
<body>
  <div>
    <sl-switch>Happy?</sl-switch>
  </div>
  <sl-radio-group label="Color:" value="red">
    <sl-radio value="red">Red</sl-radio>
    <sl-radio value="green">Green</sl-radio>
    <sl-radio value="blue">Blue</sl-radio>
  </sl-radio-group>
</body>
</html>
```

# Lifecycle Methods ...



- **constructor**

- called when an instance is created
- recommended place to attach a **ShadowRoot**

```
constructor() {
  super();
  this.attachShadow({ mode: "open" });
}
```

shadowRoot property is set when mode is "open", but not when it is "closed"

- **connectedCallback**

- called after an instance is added to DOM
- recommended place to populate web component DOM and register event handlers

```
connectedCallback() {
  this.shadowRoot.innerHTML = `...some HTML...`;
  const button = this.shadowRoot.querySelector('button');
  button.addEventListener('click', event => {
    alert('got click');
  });
}
```

- **attributeChangedCallback**

- called when the value of any “observed” attribute changes
- specify with

```
static get observedAttributes() {
  return ['name1', 'name2', ...];
}
```

```
attributeChangedCallback(name, oldValue, newValue) {
  ...
}
```

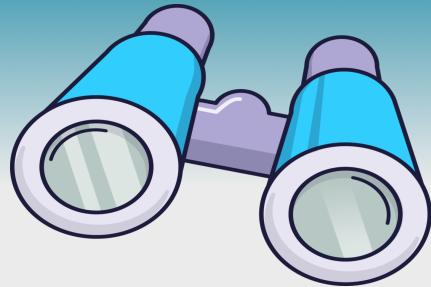


# ... Lifecycle Methods



- **disconnectedCallback** (rarely used)
  - called after an instance is removed from DOM
  - cleanup resources created in **connectedCallback**
    - resources could include event listeners, timeouts, intervals, and pending network requests
- **adoptedCallback** (rarely used)
  - called when an instance is moved to a different document

# Observing Attributes



- Let's modify the `HelloWorld1` class to update what it renders when the `name` attribute is modified
  - same approach works for other `HelloWorld*` classes

```
class HelloWorld1 extends HTMLElement {
    static get observedAttributes() {
        return ["name"];
    }

    connectedCallback() {
        const name = this.getAttribute("name") || "World";
        this.innerHTML = `<p>Hello, ${name}!</p>`;
    }

    attributeChangedCallback(name, oldValue, newValue) {
        if (name === "name") {
            const p = this.querySelector("p");
            if (p) p.textContent = `Hello, ${newValue}!`;
        }
    }
}

customElements.define("hello-world1", HelloWorld1);
```

# Cloning template Elements

- Faster than building component DOM from scratch for each instance

```
const template = document.createElement("template");
template.innerHTML = html`<p>Hello, <span id="name"></span>!</p>`;
;

class HelloWorld7 extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  connectedCallback() {
    const { shadowRoot } = this;
    shadowRoot.appendChild(template.content.cloneNode(true));
    const span = shadowRoot.querySelector("#name");
    span.textContent = this.getAttribute("name") || "World";
  }
}

customElements.define("hello-world7", HelloWorld7);
```

When `script` tag includes `type="module"`, the `template` variable is scoped to this module.



Dolly, the cloned sheep

deep clone



# Form Associated



- Instances of web components nested in a `form` element by default do not contribute to the set of name/value pairs that are submitted by the `form` focus still moves properly
  - prevented by shadow DOM
- Solution demonstrated in `radio-group` web component on next few slides
  - set static property `formAssociated` to `true`
  - declare private property `#internals`
  - initialize in constructor with  
`this.#internals = this.attachInternals();`
  - when value to contribute changes,  
call `this.#internals.setFormValue(newValue)` ;

# Radio Group Component

- Renders a set of associated radio buttons
- Uses a vanilla web component with an “open” shadow DOM
- Will use to demonstrate approaches for styling from outside component





# RadioGroup Component ...

Example instance

```
<radio-group  
  name="favoriteColor"  
  options="red,green,blue"  
  default="blue"  
  value="green"  
></radio-group>
```

default attribute is optional  
and defaults to first option;  
  
value attribute is optional  
and defaults to default

```
class RadioGroup extends HTMLElement {  
  static formAssociated = true;  
  #default;  
  #internals;  
  #name;  
  #value;  
  
  constructor() {  
    super();  
    this.attachShadow({ mode: "open" });  
    this.#internals = this.attachInternals();  
  }  
}  
radio-group.js
```

```
connectedCallback() {  
  this.#name = this.getAttribute("name");  
  const options = this.getAttribute("options")  
    .split(",")  
    .map((option) => option.trim());  
  this.#default = this.getAttribute("default") || options[0];  
  this.#value = this.getAttribute("value") || this.#default;  
  
  this.shadowRoot.innerHTML = html`  
    <style>  
      :not(:defined) {  
        visibility: hidden;  
      }  
  
      .radio-group {  
        display: flex;  
        gap: 0.25rem;  
  
        > div {  
          display: flex;  
          align-items: center;  
        }  
      }  
    </style>  
    <div class="radio-group">  
      ${options.map((option) => this.#makeRadio(option)).join("")}  
    </div>  
  `;  
  
  // Add event listeners to the radio buttons.  
  const inputs = this.shadowRoot.querySelectorAll("input");  
  for (const input of inputs) {  
    input.addEventListener("change", (event) => {  
      this.value = event.target.value;  
    });  
  }  
}
```

demonstrates factoring out some HTML generation to separate functions

invokes “set value” method on next slide



# ... RadioGroup Component

```
formResetCallback() {  
  const value = (this.value = this.#default);  
  for (const input of this.shadowRoot.querySelectorAll("input")) {  
    input.checked = input.value === value;  
  }  
}  
  
#makeRadio(option) {  
  return html`  
    <div>  
      <input  
        type="radio"  
        id="${option}"  
        name="${this.#name}"  
        value="${option}"  
        ${option === this.value ? "checked" : ""}  
      />  
      <label for="${option}">${option}</label>  
    </div>  
  `;  
}  
  
  called when containing form is reset by  
  clicking a button with type="reset" or  
  calling reset method on DOM Form object  
  
  called in connectedCallback  
  on previous slide to specify  
  the DOM to be built  
  
  get value() {  
    return this.#value;  
  }  
  
  set value(newValue) {  
    if (newValue === this.#value) return;  
    this.#value = newValue;  
    this.#internals.setFormValue(newValue);  
  
    // This demonstrates how a web component  
    // can contribute multiple values to a form.  
    /*  
      const data = new FormData();  
      data.append(this.#name, newValue);  
      data.append("favoriteNumber", 19);  
      this.#internals.setFormValue(data);  
    */  
  }  
  
  customElements.define("radio-group", RadioGroup);
```

# Shadow DOM Styling

- CSS rules defined in a web component do not leak out to affect external elements
  - enables using simple selectors like element names
- By default, web component styling is not affected by styles defined outside it
- Four ways to “pierce the Shadow DOM” to style from outside
  1. inherit CSS properties
  2. expose CSS variables
  3. use **part** attributes
  4. share CSS files





# Inherit CSS Properties



- Inheritable CSS properties are automatically used by web components unless they specify other values in their own CSS
  - include `color`, `cursor`, `font`, `font-family`, `font-size`, `font-style`, `font-variant`, `font-weight`, `letter-spacing`, `line-height`, `text-align`, `text-indent`, `text-transform`, `visibility`, `white-space`, and `word-spacing`
- Example
  - in HTML that uses the web component, include this CSS rule which causes every element the web component renders that does not specify `color` to use `blue`
    - in this case it will be the `label` elements

```
radio-group {  
  color: blue;  
}
```

Despite the fact that `inherit` is not the default value of inheritable CSS properties, they inherit the value from their parent element, unless explicitly overridden.

# Expose CSS Variables



- Web components can specify CSS properties in a way that allows them to be overridden by setting CSS variables (a.k.a. CSS custom properties)
- Example
  - in `radio-group.js`, include this CSS rule

```
label {  
  color: var(--radio-group-label-color, black);  
}
```

- in HTML that uses the web component, include this CSS rule

```
radio-group {  
  --radio-group-label-color: blue;  
}
```

# Use part Attributes



- Add part attributes to each web component element that wishes to allow external styling

- Example

- in `radio-group.js`, include this attribute

```
<label for="${option}" part="radio-label">${option}</label>
```

- in HTML that uses the web component, include this CSS rule

```
radio-group::part(radio-label) {  
  color: blue;  
}
```



# Share CSS Files

Sharing is caring



- Define styles to be shared by multiple pages and web components in a `.css` file
- Add `link` element in each page and web component that refers to the `.css` file
  - only downloaded once and cached
- Example
  - create `style.css` containing this

```
label {  
  color: blue;  
}
```

- in `radio-group.js`, include this `link` element

```
<link rel="stylesheet" href="style.css" />
```



# CSS Concerns



- To specify inheritable CSS properties on a custom element so they are not inherited, use `:host` selector

```
:host {  
  font-family: sans-serif;  
}
```

- Style custom elements with the following CSS rule to avoid “Flash of Undefined Custom Elements” (FOUCE) and layout shift

```
:not(:defined) {  
  visibility: hidden;  
}
```

# Events



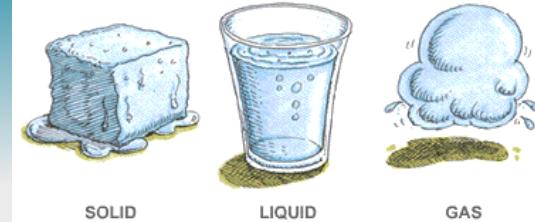
- Web components can dispatch any number of events, perhaps to advertise their state changes
- Code outside web components can listen for the events
- Example
  - in `radio-group.js`, add following at end of `set value` method

```
this.dispatchEvent(new Event("change"));
```

- listen for the event with code like this

```
const rg = document.querySelector("radio-group");
rg.addEventListener("change", (event) => {
  const { value } = event.target;
  console.log("value =", value);
});
```

# Sharing State ...



- Many approaches can be used to share state between web component instances
- A simple approach is to hold all shared state in a singleton object that
  - has a private property, getter method, and setter method for each piece of state
  - has a method that registers callback functions to be called when a specific piece of state changes
  - implements setter methods to call each registered callback function when the value is changed



# ... Sharing State ...

```
class State {
    static instance = new State();
    #favoriteColor = "transparent";
    #propertyToCallbacksMap = new Map();

    constructor() {
        if (State.instance) {
            throw new Error(
                "get singleton instance with State.instance"
            );
        }
        State.instance = this;
    }

    addCallback(property, callback) {
        let callbacks = this.#propertyToCallbacksMap.get(property);
        if (!callbacks) {
            callbacks = [];
            this.#propertyToCallbacksMap.set(property, callbacks);
        }
        callbacks.push(callback);
    }

    changed(property) {
        const callbacks =
            this.#propertyToCallbacksMap.get(property) || [];
        for (const callback of callbacks) {
            callback(this[property]);
        }
    }
}
```

Add a private instance variable for each piece of state to be shared.

Add getter and setter methods like these for each piece of state to be shared.

```
get favoriteColor() {
    return this.#favoriteColor;
}

set favoriteColor(color) {
    if (color === this.#favoriteColor) return;
    this.#favoriteColor = color;
    this.changed("favoriteColor");
}

window.State = State;
```





# ... Sharing State



Add setter methods similar to this in each web component that wishes to tie its state to the singleton **State** object.

```
set value(newValue) {
  if (newValue === this.#value) return;

  this.#value = newValue;
  this.#internals.setFormValue(newValue);
  const input = this.shadowRoot.getElementById(newValue);
  if (input) input.checked = true;
  this.dispatchEvent(new Event("change"));
}
```

Add code similar to this to wire up web components to the singleton **State** object.

```
window.onload = () => {
  const state = State.instance;

  const rgs = document.querySelectorAll("radio-group");

  // Add an event listener to each radio-group web component.
  for (const rg of rgs) {
    rg.addEventListener("change", (event) => {
      state.favoriteColor = event.target.value;
    });
  }

  // When the favoriteColor state changes,
  // update all the radio-group web components.
  state.addListener("favoriteColor", (color) => {
    for (const rg of rgs) {
      rg.value = color;
    }
  });
};
```

# Vanilla Counter Component



```
const template =
document.createElement("template");
template.innerHTML = html`

<button
    id="decrement-btn"
    type="button"
  >-</button>
  <span></span>
  <button
    id="increment-btn"
    type="button"
  >+</button>


`;
```

```
class CounterVanilla extends HTMLElement {
  static get observedAttributes() {
    return ["count"];
  }

  constructor() {
    super();
    this.attachShadow({ mode: "open" });
  }

  attributeChangedCallback() {
    this.#update();
  }

  connectedCallback() {
    const root = this.shadowRoot;
    root.appendChild(template.content.cloneNode(true));

    this.decrementBtn = root.querySelector("#decrement-btn");
    this.decrementBtn.addEventListener("click", () => {
      this.decrement();
    });
    root.querySelector("#increment-btn")
      .addEventListener("click", () => {
        this.increment();
      });
    this.span = root.querySelector("span");
    this.update();
  }

  decrement() {
    if (this.count > 0) {
      this.count--;
      this.update();
    }
  }

  increment() {
    this.count++;
    this.update();
  }

  update() {
    this.span.textContent = this.count;
  }
}
```



# Vanilla Counter Component

```
get count() {
  return this.getAttribute("count") || 0;
}

set count(newCount) {
  this.setAttribute("count", newCount);
}

decrement() {
  this.count--;
  if (this.count == 0){ ←
    this.decrementBtn.setAttribute("disabled", "disabled");
  }
  this.#update();
}

increment() {
  this.count++;
  this.decrementBtn.removeAttribute("disabled");
  this.#update();
}

#update() {
  if (this.span) this.span.textContent = this.count;
}
}

customElements.define("counter-vanilla", CounterVanilla);
```

treating `count` attribute as source of truth

`this.count` gets converted to a string, so we need to use `==` instead `==` here



# Simplifying Counter

- Previous example required a lot of code
- Can simplify using **wrec** (**Web REactive Component**)
  - superclass of `HTMLElement`
  - `npm install wrec`
- Let's see how Wrec simplifies the counter component



# Wrec Counter



```
import Wrec from "wrec"; 474 lines of code including comments and blank lines

class CounterWrec extends Wrec {
  static properties = {
    count: { type: Number },
  };

  static css = css` 
    button {
      background-color: lightgreen;
    }
    button:disabled {
      background-color: gray;
    }
  `;

  static html = html` 
    <button
      disabled="${this.count === 0}"
      onclick="${this.count--}"
      type="button"
    >-</button>
    <span>${this.count}</span>
    <button
      onclick="${this.count++}"
      type="button"
    >+</button>
  `;
}

CounterWrec.register();
```

- Looks for attribute values and text content containing `this.propertyName` and adds reactivity
- Looks for attributes whose name begins with “on” and whose value is a method name or JavaScript expression, and adds event listeners
- Add following class property to contribute to form submissions:  
`static formAssociated = true;`
- **More examples** of web components that use Wrec are in `examples` directory at <https://github.com/mvolkmann/wrec>

Lit requires much more than this.

# Wrec 2-Way Bindings

```
import Wrec, { css, html } from "../wrec.js";

class DataBinding extends Wrec {
  static properties = {
    color: { type: String },
    colors: { type: String, required: true },
  };

  static css = css`  
  :host {  
    display: flex;  
    flex-direction: column;  
    gap: 0.5rem;  
  }  
`;
```

```
    static html = html`  
      <div>  
        <label>Color Options (comma-separated) :</label>  
        <input value="this.options" />  
      </div>  
      <radio-group  
        name="color1"  
        options="this.colors"  
        value="this.color"  
      ></radio-group>  
      <select-list  
        name="color2"  
        options="this.colors"  
        value="this.color"  
      ></select-list>  
      <p>You selected the color <span>this.color</span>. </p>  
    `;  
  
  DataBinding.register();
```

Color Options (comma-separated):   
 red  green  blue

You selected the color blue.

See [radio-group.js](#) and [select-list.js](#) at  
<https://github.com/mvolkmann/wrec/tree/main/examples>.

# Slots ...

- Web component instances can supply content to be inserted using a default slot and any number of named slots

```
<html>
  <head>
    <script src="slots-demo.js" type="module"></script>
  </head>
  <body>
    <slots-demo color="mediumorchid" width="12rem">
      <span slot="title">Breaking News</span>
      <span>
        Wrec is a new library for building web components
        that has several advantages over Lit.
      </span>
      <span slot="footer">&copy; 2025 Object Computing, Inc.</span>
    </slots-demo>
  </body>
</html>
```

## Breaking News

Wrec is a new library for building web components that has several advantages over Lit.

© 2025 Object Computing, Inc.



# ... Slots

```
import Wrec, { css, html } from "wrec";

class SlotsDemo extends Wrec {
  static properties = {
    color: { type: String, value: 'cornflowerblue' },
    width: { type: String, value: '20rem' },
  };

  static css = css`  

    :host {  

      --border-radius: 1rem;  

      border: 1px solid var(--color);  

      border-radius: var(--border-radius);  

      display: inline-block;  

      font-family: sans-serif;  

      width: var(--width);  

    }  
  

    .content {  

      padding: 1rem;  

    }  
  

    .footer {  

      background-color: var(--color);  

      border-bottom-left-radius: var(--border-radius);  

      border-bottom-right-radius: var(--border-radius);  

      font-size: 0.7rem;  

      padding: 0.25rem;  

      text-align: center;  

    }  
  

    .header {  

      background-color: var(--color);  

      border-top-left-radius: var(--border-radius);  

      border-top-right-radius: var(--border-radius);  

      font-size: 2rem;  

      font-weight: bold;  

      text-align: center;  

    }  

`;
```

This demonstrates using slots in a web component defined using Wrec, but slots can be used in any web component.

```
static html = html`  

  <div class="card">  

    <div class="header">  

      <slot name="title"></slot>  

    </div>  

    <div class="content">  

      <slot></slot>  

    </div>  

    <div class="footer">  

      <slot name="footer"></slot>  

    </div>  

  </div>  
  

  connectedCallback() {  

    super.connectedCallback();  

    const { color, width } = this;  

    if (color) this.style.setProperty("--color", color);  

    if (width) this.style.setProperty("--width", width);  

  }  
  

  SlotsDemo.register();
```

default slot

# wrec Advantages Over Lit



- Simpler
  - just a single class to extend (**Wrec**)
- Much smaller
  - 5K (**wrec.min.js**) vs. 16K (**lit-core.min.js**)
- Cleaner syntax
  - don't need to surround JS expressions with `$(...)`
- Automatic 2-way data binding
  - no need to dispatch custom events and listen for them
- No special syntax for Boolean attributes
- Can put JS expressions in **textarea** elements

```
wrec
<button
  onClick="this.count--"
  type="button"
  disabled="this.count === 0"
>-</button>

-----
```

lit

```
<button
  @click={() => this.count--}
  type="button"
  ?disabled={this.count === 0}
>-</button>
```

```
wrec
<textarea>this.story</textarea>

-----
```

lit

```
<textarea
  .value=${this.story}
  @input=${this.handleStoryChange}
></textarea>
...
handleStoryChange(event) {
  this.story = event.target.value;
}
```

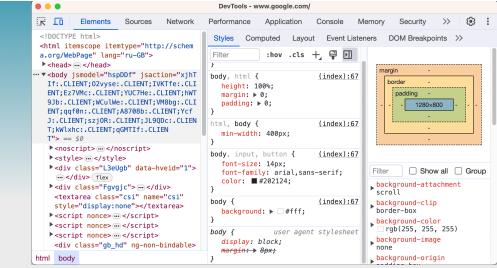
Boolean  
attribute

textarea  
content



# DevTools Tips

These work in Chrome, Firefox, and Safari.



- To change a web component attribute value
  - select Elements tab
  - select web component
  - double-click an attribute value and enter a new value
  - commit change by pressing return or tab key, or clicking elsewhere
  - verify that the component UI updates
- To change a web component property
  - select Elements tab
  - select web component
  - select Console tab
  - enter `$0.propertyName = newValue`
  - verify that the component UI updates
  - verify that the corresponding attribute updates
  - enter `el = document.querySelector('component-name');`
  - enter `el.propertyName = newValue`
  - verify that the component UI updates

just a different way  
to get a reference  
to a custom element  
that can be used in code



# Resources

- **My blog** - <https://mvolkmann.github.io/blog/>
  - select “Web Components”
- **These slides** - <https://github.com/mvolkmann/talks/blob/master/web-components.key.pdf>
- **Custom Elements Everywhere** - <https://custom-elements-everywhere.com/>
  - “making sure frameworks and custom elements can be BFFs”
- **Kinsta** - <https://kinsta.com/blog/web-components/>
  - “complete introduction to web components”
- **MDN docs** - [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_components](https://developer.mozilla.org/en-US/docs/Web/API/Web_components)
- **Open Web Components** - <https://open-wc.org/>
  - “guides, tools and libraries for developing web components”
- **WebComponents.org** - <https://www.webcomponents.org/>
  - “building blocks for the web”



# Wrap Up

- Web components
  - provide a **nice alternative** to implementing UI components using popular frameworks
  - are **standards-based** and “use the platform”
  - **can be shared** across apps that use frameworks
  - don’t necessarily require libraries, tooling, or build processes
  - have a relatively **small learning curve**
- Checkout my npm package “wrec”!
  - <https://www.npmjs.com/package/wrec>

