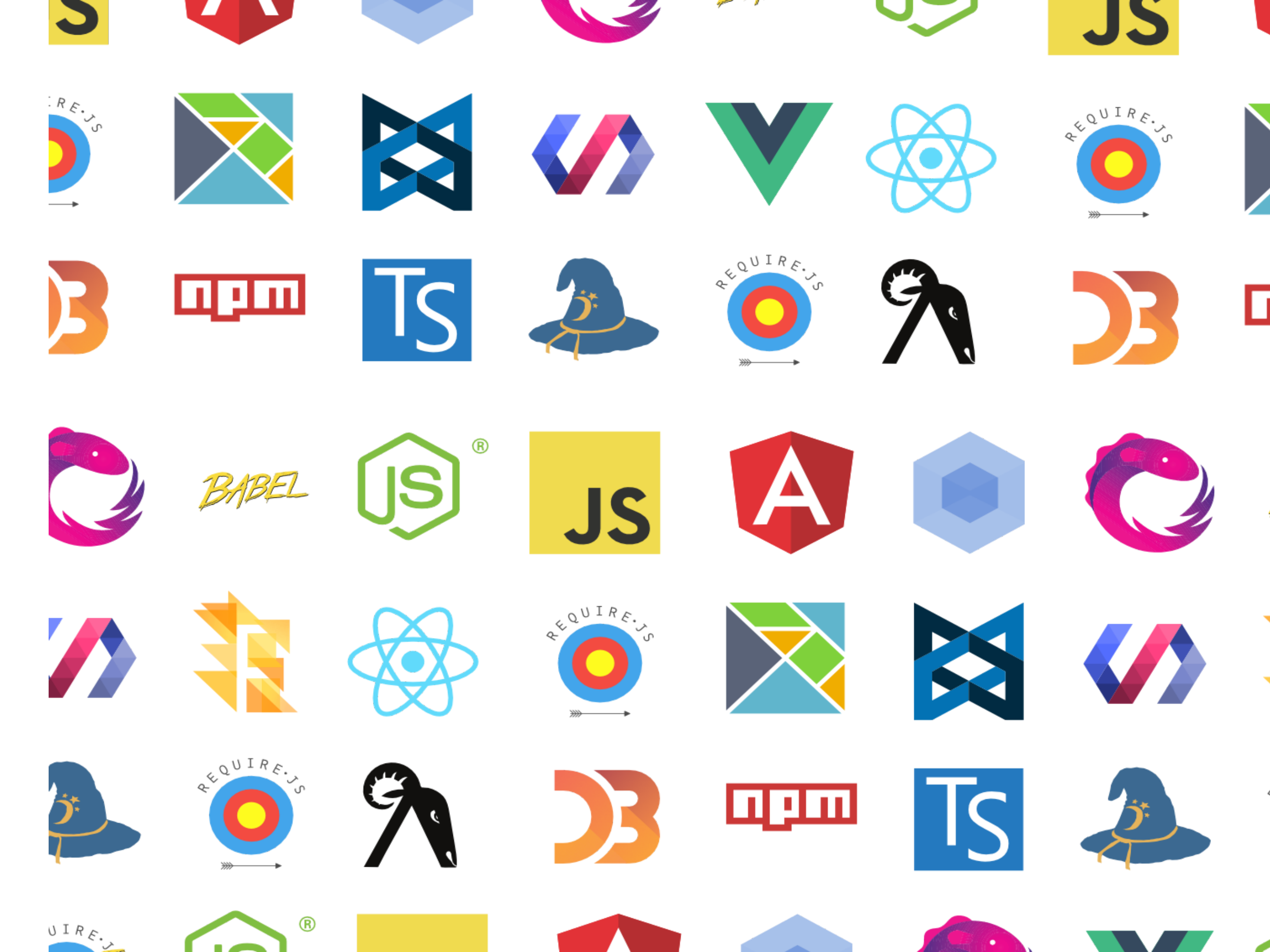


# Web Components Workshop

Irhad Kulanic & Lukas Korten

**Ein weiteres Framework?**





# Web Components

- › Eine Gruppe von Webtechnologien
- › benutzerdefinierte und wiederverwendbare HTML-Elemente
- › in sich gekapselte Funktionalität
- › einsetzbar auf allen Webseiten und in allen Frameworks



# Die Haupttechnologien

- › Custom Elements
- › Shadow DOM
- › HTML-Templates
- › (ES Modules)

# Custom Elements

Benutzerdefinierte HTML-Elemente

# Custom Elements

`CustomElementRegistry`

- Controller für benutzerdefinierte Elemente
- global über `customElements` zugreifbar

`CustomElementRegistry.define()` - registriert ein neues Custom Element

## Argumente

- Name des Elements (Ein `-` im Namen ist Pflicht)
- Das Verhalten definierende Klassenobjekt

# Custom Elements

## Das Klassenobjekt

### › ES 2015 Standardsyntax für Klassen

```
class MyPopup extends HTMLElement {  
  constructor() {  
    super(); // muss immer als erstes aufgerufen werden  
  
    // Funktionalität des Elements  
    ...  
  }  
}
```





# Custom Elements

## Zwei Arten

- autonome Custom Elements
- erweiterte Standardelemente (built-in elements)



# Custom Elements

## Autonome Custom Elements

- › erben direkt vom `HTMLElement`
- › nicht von Standardelementen wie z.B. `HTMLInputElement`
- › Name kann als Tag verwendet werden



# Custom Elements

```
class MyPopup extends HTMLElement {  
  constructor() { ... }  
}
```

```
customElements.define('my-popup', MyPopup);
```



# Custom Elements

Kann mit dem Namen `my-popup` erzeugt werden

## › JavaScript

```
document.createElement('my-popup');
```

## › HTML

```
<my-popup></my-popup>
```

# Custom Elements

## Erweiterte Standardelemente (Built-in Elements)

- erben von Standardelementen
- Beim Erstellen Angabe des erweiterten Standardelement-Tags erforderlich
- Mit dem Attribut `is` wird der Name des Elements angegeben

🔍 📄

Elements Console Sources Network Performance Memory Application Security >>

<search-box-component>

#shadow-root (open)

>>

>>

Styles

Computed

Properties

▶ input.web-search-input

▶ HTMLInputElement

▶ HTMLElement

▶ Element

▶ Node

▶ EventTarget

▶ Object

▶ <style>...</style>

▼ <div class="web-search-container">  
 <style type="text/css">@import  
 url("chrome://startpage/searchstyle.css?  
 ts=1561974834046")</style>  
 ▼ <form class="web-search">  
 ▼ <div class="web-search-inner-container">  
 ▶ <span class="web-search-logo-  
 container">...</span>  
 ▼ <div class="suggestion-input-  
 container">  
 ...  
 <input class="web-search-input"  
 autocomplete="off" incremental  
 placeholder="Suche im Web" tabindex=  
 "2000" type="search" value> == \$0  
 </div>  
 <button class="web-search-submit"  
 tabindex="2000" type="submit">Suchen  
 </button>  
 </div>  
 </form>  
 </div>



# Custom Elements

```
class MyAlert extends HTMLParagraphElement {  
  constructor() { ... }  
}
```

```
customElements.define('my-alert', MyAlert, { extends: 'p' });
```



# Custom Elements

Beim Erzeugen muss das Attribut `is` angegeben werden

## › JavaScript

```
document.createElement('p', { is: 'my-alert' });
```

## › HTML

```
<p is="my-alert"></p>
```



# DEMO 1

Custom Element

# Übung 1 - Ein Custom Element implementieren

Implementiert eine Visitenkarte als ein **automares Custom Element**. Die Visitenkarte soll die folgende HTML-Struktur haben:

```
<div class="contact-card">
  
  <div class="content">
    <h1>Max Mustermann</h1>
    <p class="mail">max.mustermann@example.com</p>
    <button>Profilbeschreibung</button>
    <div class="details"><p>Lorem ipsum...</p></div>
  </div>
</div>
```

Einstiegsdatei: `Uebung_1-CustomElements/index.js`.

# Shadow DOM

Kapselung von CSS und HTML

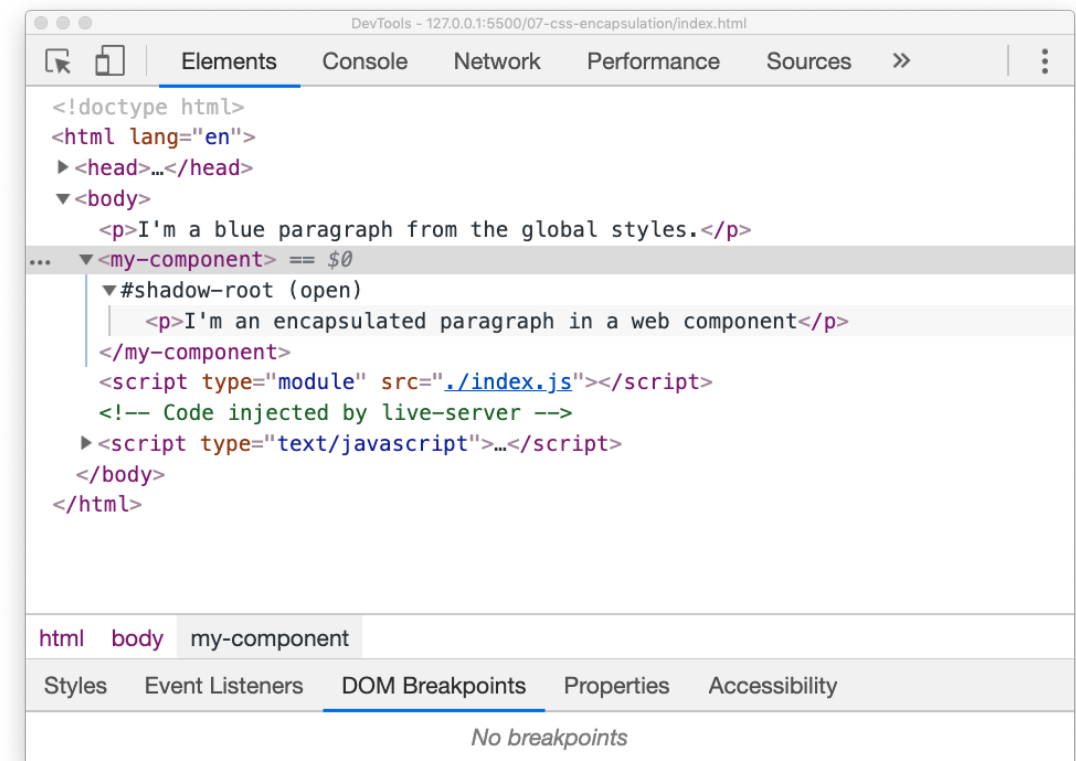
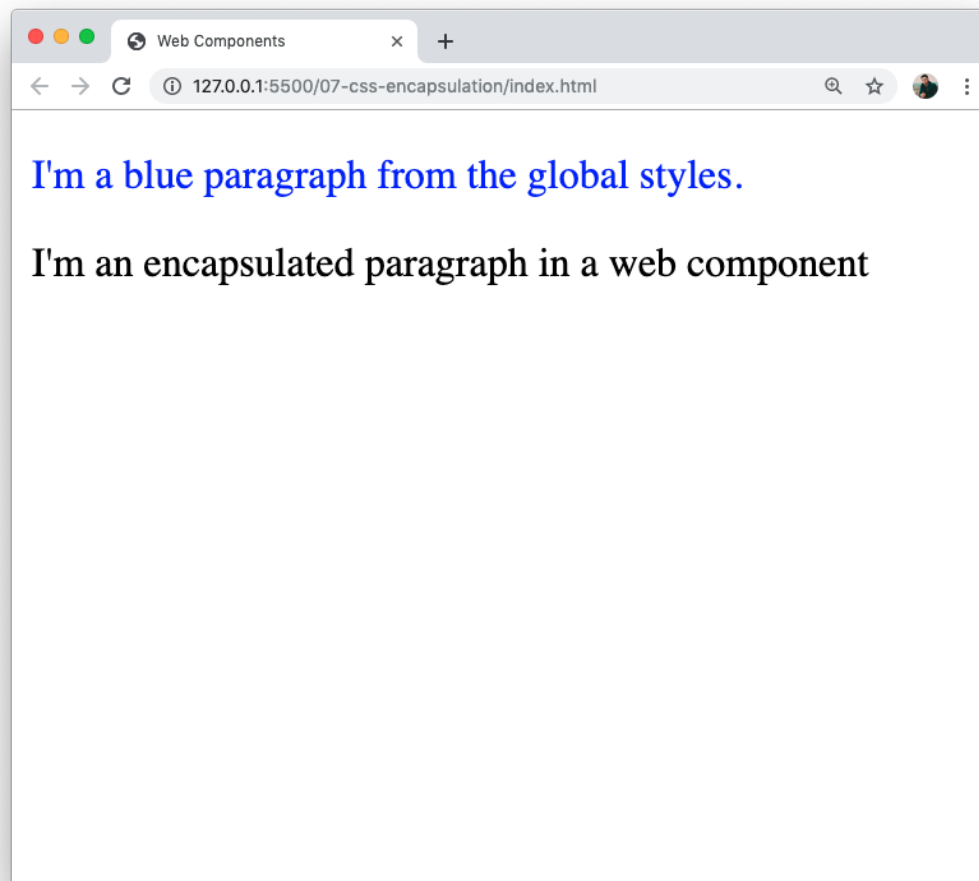


# Shadow DOM

- › ein isolierter *Subtree* mit eigenem Markup und Styling
- › wird unabhängig vom DOM des Hauptdokuments gerendert
- › kollidiert nicht mit Skripten und Styles des Hauptdokuments
- › Elemente im Shadow DOM sind über `document.querySelector()` nicht zugreifbar



# Shadow DOM



# Shadow DOM

- **Shadow host** - Der DOM-Knoten, an dem das Shadow DOM hängt
- **Shadow tree** - Der DOM-Tree (Struktur) des Shadow DOM
- **Shadow boundary** - Der Geltungsbereich des Shadow DOM
- **Shadow root** - Der Root-Knoten des DOM-Tree



# Shadow DOM

```
class MyComponent extends HTMLElement {  
  constructor() {  
    super();  
  
    this.attachShadow({ mode: 'open' });  
    const childElement = document.createElement('p');  
    this.shadowRoot.appendChild(childElement);  
  }  
}  
  
customElements.define("my-component", MyComponent);
```

# Shadow DOM

› open vs closed

I'm in the Shadow Root!

Elements Console Sources Network Timeline Profiles Application Security

```
<!DOCTYPE html>
<html lang="en">
  ▶ #shadow-root (open)
  ▶ <head>...</head>
  ▼ <body>
    ▼ <my-web-component>
      ... ▼ #shadow-root (open) == $0
        | <p>I'm in the Shadow Root!</p>
        </my-web-component>
      </body>
    </html>
```

html body my-web-component #shadow-root

⋮ Console Sensors

🚫 🔍 top ▼ ☐ Preserve log

> |



# DEMO 2

Shadow DOM



## Übung 2 - Shadow DOM

In dieser Übung soll das bereits entwickelte Custom Element `MyContractCard` mit Shadow DOM erweitert werden. Einstiegsdatei mit zusätzlichen Hinweisen: `uebungen/Uebung_2-ShadowDOM/index.js`



# Shadow DOM Styles

- Im Shadow DOM definiertes CSS wirkt sich nur auf angehängte Elemente aus
- globales CSS hat keinen Einfluss auf Elemente im Shadow DOM
- Mit `:host` - Das Wurzel-Element der Web Component selektieren

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



# Shadow DOM Styles

```
constructor(){  
  super();  
  
  const style = document.createElement('style');  
  style.textContent = `p { color: red; }`;  
  
  this.attachShadow({ mode: 'open' });  
  this.shadowRoot.appendChild(style);  
}
```

# DEMO 3

Shadow DOM - Styles im Shadow DOM definieren



## Übung 3 - Shadow DOM Styles

Eigenes `style`-Element auf der `shadowRoot` definieren und damit das Darstellungsproblem aus der Übung 2 korrigieren. Die Einstiegsdatei für diese Übung ist `uebungen/Uebung_3-ShadowDOM_Styles/index.js`.

# HTML Templates und Slots

Einsatz von `<template>` und `<slot>` Elementen

# HTML Templates

- › wiederverwendbare Markup-Vorlagen
- › sind im HTML-Dokument enthalten, werden aber vom Browser nicht angezeigt
- › dienen als Grundlage für wiederverwendbare Elemente
- › können sehr effizient geklont werden
- › können im HTML oder JavaScript definiert werden



# HTML Templates

- › neuer HTML-Tag `<template>`
- › bleibt auf der Seite unsichtbar bis es initialisiert wurde
- › benötigt eine `id`

## HTML

```
<!-- Bleibt auf der Seite unsichtbar -->  
<template id="dw-template">  
  <p>Das ist ein Text</p>  
</template>
```

## JavaScript

```
const template = document.querySelector('#dw-template');
```



# HTML Templates

...

```
constructor(){  
  super();  
  this.attachShadow({mode: 'open'});  
  
  const template = document.querySelector('#dw-template');  
  this.shadowRoot.appendChild(template.content.cloneNode(true))  
}
```

...



# HTML Templates

```
const myTemplate = document.createElement('template');
myTemplate.innerHTML = `
  <p>I'm an encapsulated paragraph in a web component</p>
`;
...
constructor(){
  super();
  this.attachShadow({mode: 'open'});
  this.shadowRoot.appendChild(myTemplate.content.cloneNode(true)
}
...
```

# DEMO 4

HTML Templates



## Übung 4 - HTML Template verwenden

In dieser Übung wollen wir uns mit Templates befassen. Lagert alle Kindelemente aus `MyContractCard` in ein Template aus. Das Template soll in der JavaScript-Datei `index.js` implementiert und an die `shadowRoot` angehängt werden.

Einstieg: `uebungen/Uebung_4-HTMLTemplate/index.js`

# HTML Slots

➤ Markup-*Einschübe* an vordefinierten Stellen im `<template>`

```
<template id="info-alert-template">
  <div>
    <h1>Info</h1>
    <slot></slot>
  </div>
</template>
```

```
<my-info-alert>
  <p>Eine ganz wichtige Meldung</p>
</my-info-alert>
```

# HTML Slots

➤ Markup-*Einschübe* an vordefinierten Stellen im `<template>`

```
<template id="info-alert-template">
  <div>
    <h1>Info</h1>
    <slot><p>Kein Inhalt ...</p></slot> <!-- Defaultwert -->
  </div>
</template>
```

```
<my-info-alert>
  <p>Eine ganz wichtige Meldung</p>
</my-info-alert>
```

# HTML Slots

- › mehrere `<slot>`-Tags innerhalb eines Templates möglich
- › mit dem Attribut `name` den Namen des `<slot>` definieren

```
<template id="info-alert-template">
  <div>
    <slot name="title">
      <h1>Info</h1>
    </slot>
    <slot name="message">
      <p>Kein Inhalt ...</p>
    </slot>
  </div>
</template>
```



# HTML Slots

➤ mit `slot="name-des-slots"` den passenden `<slot>` ansprechen

```
<my-info-alert>
```

```
  <h1 slot="title">Benutzerdaten aktualisiert</h1>
```

```
  <p slot="message">
```

```
    Die Benutzerdaten für Max Mustermann konnten ...
```

```
  </p>
```

```
</my-info-alert>
```

# DEMO 5

HTML Slots

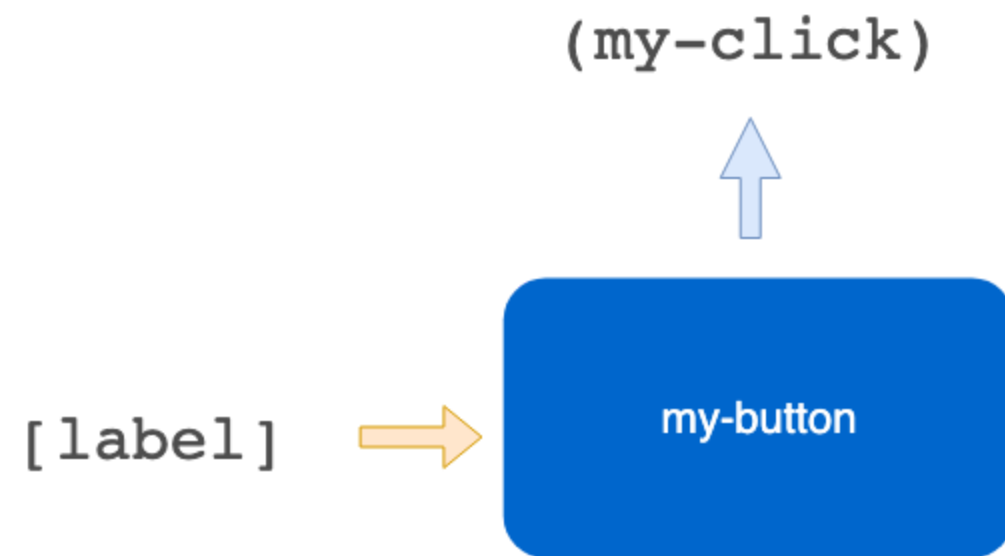
## Übung 5 - HTML Slots

In dieser Übung soll das vorhandene Template mit einem `slot`-Element erweitert werden. Das benutzerdefinierte Markup, welches die Benutzer zwischen den Tags der `my-contact-card`-Web Component eingeben, soll an der Stelle des Standardtextes im `.details`-Container erscheinen.

Einstiegsdatei: `uebungen/Uebung_5-HTMLSlot/index.js`

# Properties & Events

Input und Output einer Web Component



# Events

- › gehört zum öffentlichen API einer Web Component
- › andere Komponenten über eigene Ereignisse informieren
- › Einsatz der `CustomEvent` API
- › mit `EventTarget.dispatchEvent()` Event auslösen



# Events

```
// my-button.js
```

```
const template = document.createElement('template');  
template.innerHTML = `<button>Klick mich</button>`;
```

```
class MyButton extends HTMLElement {  
    ...  
}
```

```
customElements.define('my-button', MyButton);
```



# Events

```
// my-button.js
```

```
...
```

```
constructor() {  
  super();  
  this.attachShadow({mode: 'open'});  
  this.shadowRoot.appendChild(template.content.cloneNode(true))  
  this.buttonElement = this.shadowRoot.querySelector('button')  
  this.buttonElement.addEventListener('click', () => this.handleClick()  
}
```

```
handleClick() {  
  this.dispatchEvent(new CustomEvent('my-click', { detail: 'Klick' }  
}
```

```
...
```





# Events

```
// index.js
```

```
import './my-button.js';
```

```
const button = document.querySelector('my-button');
```

```
button.addEventListener('my-click', e => console.log(e));
```

# Properties

- › öffentliches API einer Web Component
- › dynamisch Eigenschaften verändern
- › `set` und `get` -Methoden für jede Property definieren
- › beim Setzen (wenn nötig) Elemente aktualisieren



# Properties

```
// my-button.js
```

```
const template = document.createElement('template');  
template.innerHTML = `<button>Klick mich</button>`;
```

```
class MyButton extends HTMLElement {  
    ...  
}
```

```
customElements.define('my-button', MyButton);
```

# Properties

```
const template = document.createElement('template');  
template.innerHTML = `<button>Klick mich</button>`;
```

```
class MyButton extends HTMLElement {  
  set label(value) {  
    this._label = value;  
    this.buttonElement.innerText = this._label;  
  }  
  get label() {  
    return this._label;  
  }  
  constructor() { ... }  
  ...  
}
```



# Properties

```
// index.js
```

```
import './my-button.js';
```

```
const button = document.querySelector('my-button');
```

```
button.label = 'Speichern';
```

```
button.addEventListener('my-click', e => console.log(e));
```

# Attribute

Eigenschaften im HTML festlegen



# Attribute

```
<!-- index.html -->
```

```
<my-button></my-button>
```



# Attribute

```
<!-- index.html -->
```

```
<my-button label="Speichern"></my-button>
```





# Attribute

```
class MyButton extends HTMLElement {  
  
    static get observedAttributes() {  
        return ['label'];  
    }  
  
    ...  
  
}
```



# Attribute

```
class MyButton extends HTMLElement {  
  
    static get observedAttributes() {  
        return ['label'];  
    }  
  
    ...  
    attributeChangedCallback(attrName, oldValue, newValue) {  
        if (attrName === 'label') {  
            this.label = newValue;  
        }  
    }  
}
```

# DEMO 6

Attribute, Properties & Events

# Callbacks

Der Lebenszyklus eines Custom Elements



# Callbacks

- › in jedem Custom Element enthalten
- › steuern das Verhalten einer Web Component
- › ermöglichen Eingriffe bei speziellen Ereignissen und Änderungen

# Callbacks

- › connectedCallback
- › disconnectedCallback
- › attributeChangedCallback
- › adoptedCallback

können im Lebenszyklus **mehrmals** aufgerufen werden

# Callbacks

```
connectedCallback
```

- wird aufgerufen, wenn das Element in das DOM des Dokuments eingebunden wird
- nützlich, wenn das Element vom innerem oder äußerem DOM abhängig ist

...

```
connectedCallback() {  
    // z.B. berechnete Styles abfragen  
    // oder auf das DOM zugreifen  
}
```

...

# Callbacks

```
disconnectedCallback
```

- wird aufgerufen, wenn das Element vom DOM des Dokuments getrennt wird

...

```
disconnectedCallback() {  
    // z.B. Properties zurücksetzen  
    // oder EventListener entfernen  
}
```

...



# Callbacks

- › wenn eines der Element-Attribute hinzugefügt, gelöscht oder verändert wurde

```
attributeChangedCallback
```

...

```
attributeChangedCallback(attrName, oldValue, newValue) {  
    if (attrName === 'label') {  
        this.label = newValue;  
    }  
}
```

...

# Callbacks

`adoptedCallback`

- › wenn das Element in das DOM eines anderen Dokuments verschoben wird (z.B. `iframe`)
- › nur da wo `connectedCallback` nicht ausreichend ist

...

```
adoptedCallback() {  
    // Beispiel: Das Element wurde mit adoptNode() in ein iframe  
}
```

...

# DEMO 7

Callbacks



# Übung 6 - Properties, Attribute und Events

In dieser Übung geht es um Deklaration benutzerdefinierter Properties und Attribute und das Auslösen von benutzerdefinierten Events.

Einstiegsdatei: `uebungen/Uebung_6-Callbacks-Attributen/index.js`










# Browser-Support & Polyfills

Support für Web Components

# Browser-Support

- alle Webkit und Chromium Browser (Chrome, Safari, Opera)
- Firefox ab Version 63
- Edge arbeitet aktiv an der Implementierung
- IE kein Support

# Browser-Support

Browser support	 CHROME	 OPERA	 SAFARI	 FIREFOX	 EDGE
 HTML TEMPLATES	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE
 CUSTOM ELEMENTS	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE	✓ POLYFILL • DEVELOPING
 SHADOW DOM	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE	✓ POLYFILL • DEVELOPING
 ES MODULES	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE	✓ STABLE



# Browser-Support

## Zwei Probleme:

- kein Support für ES2015 Features, wie Klassen und Module
- Web Component APIs werden nicht unterstützt





# Browser-Support

## Lösung:

- Support für ES2015 Features: **Babel** oder **Typescript** einsetzen
- Web Component APIs unterstützen: **Polyfills** einsetzen

# Polyfills

- › kleine JavaScript-Bibliotheken, welche die fehlenden Technologien im Browser nachbilden
- › ältere Browser Versionen können somit unterstützt werden
- › können zu Einschränkungen der Performanz führen
- › einige Technologien lassen sich nicht nachbilden, z.B. shadow DOM im IE 11

Empfohlene Polyfills: <https://www.webcomponents.org/polyfills>

# Polyfills

## Einbindung

➤ `webcomponents-loader` lädt das passende Polyfill-Bundle nach

```
<!-- webcomponents-loader einbinden -->
```

```
<script src="../../webcomponentsjs/webcomponents-loader.js"></script>
```

```
<!-- Web Component laden -->
```

```
<script type="module" src="my-element.js"></script>
```

```
<!-- Web Component nutzen -->
```

```
<my-element></my-element>
```

# DEMO 8

Polyfills

# Frameworks

effiziente Entwicklung von Web Components



# Frameworks

- › effizientes Arbeiten
- › kein Boilerplate Code
- › je nach Library oder Framework `npm` erforderlich



# Frameworks

- lit-html und LitElement (Polymer)
- Angular Elements
- Stencil

# lit-html

- HTML-Templating-Library für JavaScript
- entwickelt vom Polymer-Project-Team
- extrem schnell
- kein virtuelles DOM Diffing
- aktualisiert nur geänderte Template-Zweige
- Verschachtelung und Komposition möglich



# lit-html

- › nutzt HTML-Strings
- › Templates werden in Funktionen ausgelagert
- › können wiederholt mit unterschiedlichen Daten aufgerufen werden
- › `TemplateResult` ist das Ergebnis eines solchen Aufrufs
- › mit `render()` kann das Ergebnis endgültig gerendert werden

```
let template = (name) => html`  
  <h1>Hallo {name}!</h1>`;
```

```
const result = template('Joe');  
render(result, document.body);
```

# LitElement

- › Basisklasse für eigene Custom Elements
- › zusätzliche API, um Properties und Attribute zu verwalten
- › bei Änderungen von Properties werden entsprechende Elemente automatisch aktualisiert

```
import { LitElement, html } from 'lit-element';  
  
class MyComponent extends LitElement {  
    ...  
}  
customElements.define('my-component', MyComponent);
```

# LitElement

```
class MyComponent extends LitElement {  
  static get properties() {  
    return {  
      name: {type: String}  
    }  
  }  
  render() {  
    return html`<p>Hallo {this.name}</p>`;  
  }  
}
```



# Angular Elements

- `createCustomElement()` konvertiert eine Angular Component in ein Custom Element
- mit allen Abhängigkeiten und vollständiger Funktionalität
- das Ergebnis ist ein neues Custom Element
- muss über `customElements.define()` registriert werden
- besonders interessant für dynamische Komponenten



# Angular Elements

**Beispiel:** eine `PopupComponent` in ein `PopupElement` konvertieren

```
@Component({ ... })  
export class AppComponent {  
  constructor(injector: Injector) {  
  
    const PopupElement = createCustomElement(PopupComponent, {in  
    customElements.define('popup-element', PopupElement);  
  
  }  
}
```



# Angular Elements

## Mapping

- `Input()` -Properties werden automatisch in Attribute umgewandelt

```
@Input('myInputProperty') inputProp;  
// => my-input-property
```

- `Output()` -EventEmitter werden in `dispatchEvent` -Aufrufe mit `CustomEvent` 's übersetzt

```
@Output('myClick') click = new EventEmitter<string>();  
this.click.emit('Hallo Welt!');  
// => this.dispatchEvent(new CustomEvent('myClick', { detail:
```



# Angular Elements

## Das Popup-Beispiel

# Stencil

- Eine Sammlung verschiedener Entwicklungs-Tools für die Entwicklung von:
- Komponentenbibliotheken, Applikationen und Designer Systemen
- stellt einen Compiler zur Verfügung, um optimierte Web Components zu generieren
- wurde von Ionic entwickelt



# Stencil

**Stencil CLI** ermöglicht:

- die Entwicklung eigener Komponentenbibliotheken
- Einsatz von Typescript und JSX
- Auslagerung von CSS und Sass in eigene Dateien



# Stencil

```
import { Component, Prop, h } from '@stencil/core';

@Component({
  tag: 'my-component',
  styleUrls: 'my-component.css',
  shadow: true
})
export class MyComponent {
  @Prop() title: string;
  render() {
    return <div>Hello, World! I'm {this.name}</div>;
  }
}
```

# Stencil

## Dekoratoren

- `@Component()` deklariert eine neue Web Component
- `@Prop()` eine öffentliche Property bzw. ein Attribut
- `@State()` eine Property, bei deren Änderung das Template aktualisiert werden soll
- `@Watch()` eine Hook-Methode für Änderungen an `Prop` und `State` - Properties
- `@Element()` deklariert eine Referenz auf das Host-Element
- `@Method()` für öffentliche Methoden
- `@Event()` ein DOM Event, welches von der Component ausgelöst werden kann
- `@Listen()` Listener für DOM-Events

# Stencil

## Callbacks

- `connectedCallback()`
- `disconnectedCallback()`
- `componentWillLoad()`
- `componentDidLoad()`
- `componentWillRender()`
- `componentDidRender()`
- `componentWillUpdate()`
- `componentDidUpdate()`
- `render()`

# DEMO 9

Stencil aufsetzen

# Einsatz von Fremdbibliotheken

Veröffentlichte Web Components im eigenen Projekt nutzen

# Einsatz von Fremdbibliotheken

- Web Components sind valides HTML
- lassen sich in jedem HTML-basierten Framework einsetzen
- große Auswahl veröffentlichter Web Components:
  1. <https://www.webcomponents.org/elements>
  2. <https://www.npmjs.com/search?q=keywords:web-components>

# Einsatz von Fremdbibliotheken

➤ verschiedene Ansätze eine Web Component zu importieren:

1. als npm Paket ins Projekt importieren
2. über `<script>`-Tag in HTML-Dokument einbinden
3. als ES-Modul in die eigene JavaScript-Datei importieren: `import *`  
`from`



# DEMO 10

Vaadin-Web Components einbinden