4 Polymer-Grundlagen

Wie das vorige Kapitel zeigte, ist die Arbeit mit den reinen Standards durchaus möglich, aber auch etwas mühsam: Wir mussten alles selbst aufbauen und die Komponenten zu verknüpfen, erforderte einiges an Code. In diesem Kapitel betrachten wir nun die Grundlagen des Polymer-Frameworks. Wir werden sehen, wie Polymer uns die Arbeit mit Web Components erleichtert und wie die vom Framework bereitgestellten Werkzeuge und Komponenten die Entwicklung vereinfachen und beschleunigen können.

In diesem Kapitel befassen wir uns zunächst mit den grundlegenden Konzepten, die in der Entwicklung mit Polymer eine Rolle spielen, wie Data-Binding, Properties, Events und Styling.

4.1 Aufbau einer Polymer-Anwendung

4.1.1 Das Polymer-Framework

Im Gegensatz zu den meisten Frameworks ist Polymer in mehrere Schichten aufgeteilt. Die Grundlage, also die unterste Schicht, bilden dabei die **Polyfills**, zusammengefasst in Webcomponents.js. Diese Polyfills stellen die wichtigsten Funktionen der Web-Components-Spezifikation auch in älteren Browsern bereit. Da die Spezifikationen aber tief in die Browser-Implementierung eingreifen, bestehen für die Polyfills bestimmte Einschränkungen, die wir im nächsten Abschnitt näher betrachten werden.

Die nächste Schicht bietet die **Polymer-Plattform**. Hier sind Grundfunktionen, wie das `Polymer.Base`-Element, das die Grundlage aller Polymer-Elemente bietet, Helfer-Funktionen für diverse Aufgaben und die Lifecycle-Callbacks implementiert.

Darüber liegen die **Polymer-Elemente**. Diese sind selbst in verschiedene Gruppen aufgeteilt: Iron-, Paper-, Google-, Gold-, Neon- und Platin-Elemente. Hier ist ein grober Überblick:

- *Iron*: grundlegende Elemente für UI und Struktur, zum Beispiel `<iron-icon>` oder `<iron-list>`

- *Paper*: Material-Design-Elemente, beispielsweise `<paper-slider>`, `<paper-card>` oder `<paper-menu>`
- *Google*: Elemente für Google-Dienste, etwa `<google-map>` oder `<google-hangout-button>`
- *Gold*: Elemente für E-Commerce-Anwendungen, etwa `<gold-cc-input>` für ein Kreditkarten-Formular
- *Neon*: ein Element für Animationen
- *Platin*: Elemente für Web-Apps mit teils experimentellen Funktionen, wie `<platinum-push-messaging>` oder `<platinum-bluetooth>`

Die Polymer-Elemente können (müssen aber nicht!) für eigene Elemente als Grundlage dienen. Eigene Elemente können demnach bestehende Funktionen erweitern oder modifizieren oder direkt auf der Polymer-Plattform aufbauen.

Es gibt bereits (unter dem Namen `Molecules` geführte) JavaScript-Bibliotheken, die als zusätzliche Polymer-Elemente implementiert sind. Ein Beispiel dafür ist die beliebte Code-Syntax-Highlighting-Bibliothek (`http://prismjs.com`).

4.1.2 Die Polymer-Polyfills: webcomponents.js

Die Polyfills, die von Polymer genutzt werden, sind unter dem Namen »webcomponents.js« zusammengefasst, bestehen aber aus einzelnen, separat nutzbaren Polyfills für:

- Custom Elements
- Shadow DOM
- HTML-Imports
- Template Element

4.1.3 Eigene Elemente, fremde Elemente und Pages

In Polymer ist alles ein Element, aber die Elemente können grundsätzlich in drei Kategorien eingeteilt werden. Dies erlaubt uns die Strukturierung unserer Anwendung in drei wesentliche Bestandteile:

1. *Eigene Elemente*: die Komponenten, die wir selbst spezifisch für die Anwendung erstellen. Sie liegen üblicherweise in einem `elements`-Verzeichnis in der Anwendung.
2. *Externe Elemente*: Komponenten von externen Anbietern, beispielsweise die Komponenten von Polymer selbst oder anderen Drittanbietern. Sie werden per `bower` installiert und liegen üblicherweise im `bower_components`-Verzeichnis.

3. *Pages*: die einzelnen »Seiten«, also die einzelnen Ansichten, unserer Anwendung sind ebenfalls als Polymer-Elemente umgesetzt, enthalten aber meist kaum eigene Logik, sondern kombinieren nur andere Elemente zu einer zusammenhängenen Ansicht. Üblicherweise werden diese im `pages`-Verzeichnis abgelegt.

4.2 Polymer-Elemente registrieren

4.2.1 Hallo-Polymer

Nach dieser ersten Einführung ist es jetzt an der Zeit, ein erstes Polymer-Element zu schreiben. Dafür erstellen wir zunächst ein Verzeichnis und installieren Polymer per Bower.

Hinweis

Es ist nicht zwingend notwendig, Polymer per Bower zu installieren und eigene Elemente mittels Bower bereitzustellen. Da dies aber die gängige Praxis in der Polymer-Landschaft ist, sollte man sich dies generell angewöhnen, um sich und anderen die Nutzung zu erleichtern. Im Anhang wird Bower näher vorgestellt.

Im Verzeichnis für unsere Komponente führen wir dafür folgenden Befehl aus:

```
bower init
  ? name name-tag
  ? description A name tag polymer element
  ? main file name-tag.html
  ? what types of modules does this package expose? globals
  ? keywords Polymer, name-tag
  ? authors John Doe <john.doe@example.org>
  ? license MIT
  ? homepage
  ? set currently installed components as dependencies? Yes
  ? add commonly ignored files to ignore list? Yes
  ? would you like to mark this package as private which prevents it from being ac
  cidentally published to the registry? Yes
```

Damit haben wir unsere »name-tag« Komponente als Bower-Paket initialisiert. Dieser Schritt legt hinter den Kulissen eine `bower.json`-Datei an, die neben Informationen zu unserer Komponente auch deren Abhängigkeiten festhält und anderen Entwicklern später die Installation und Verwendung unserer Komponente erleichtert.

Als Nächstes installieren wir Polymer:

```
bower install --save Polymer/polymer
```

Das installiert Polymer und die Polyfills (`webcomponentsjs`) in das `bower_components/`-Unterverzeichnis. Die Angabe von `--save` sorgt dafür, dass diese Abhängigkeit in unserer `bower.json` festgehalten wird.

Nun können wir die `name-tag.html` anlegen:

```
<link rel="import" href="bower_components/polymer/polymer.html">
 <script>
  Polymer({
    is: "name-tag",
    created: function() {
      this.textContent = "Hallo, mein Name ist Polymer"
    }
  });
 </script>
```

Die `Polymer`-Funktion ist der Dreh- und Angelpunkt der Polymer-Plattform. Das `is`-Attribut legt das Element-Tag fest, `created` ist einer der Lifecycle-Callbacks von Polymer.

4.2.2 Lifecycle-Callbacks

Die folgende Tabelle vergleicht die Lifecycle-Callbacks von Polymer-Elementen mit Standard-Webcomponents und erklärt die Unterschiede:

Polymer-Callback	Standard-Callback	Beschreibung
`created`	`createdCallback`	Einmaliger Aufruf, wenn das Element initialisiert wird. Wird aufgerufen, bevor Properties mit Werten befüllt werden.
`ready`	keiner	Einmaliger Aufruf, wenn das lokale DOM (dazu später mehr) und Properties ihre Werte erhalten haben.
`attached`	`attachedCallback`	Wird immer aufgerufen, wenn das Element ins DOM eingefügt wurde. Dies ist der beste Zeitpunkt, um die Darstellung des Elements vorzubereiten.
`detached`	`detachedCallback`	Wird immer aufgerufen, nachdem das Element aus dem DOM entfernt wurde. Hier lassen sich beispielsweise Timer oder Events deaktivieren.
`attributeChanged`	`attributeChangedCallback`	Wird immer aufgerufen, wenn sich Attribute des Elements geändert haben. Dieser Callback wird nur für Attribute aufgerufen, die nicht einem Property entsprechen.

Tab. 4–1 *Lifecycle-Callbacks*

Das Element ist im Augenblick noch nicht besonders sinnvoll; der Inhalt ist festgelegt und seine innere Struktur ist sehr simpel.

4.3 Local DOM und Light DOM

Verbessern wir das Element also zu einem Element mit definierbarem Namen und einer etwas interessanteren inneren Struktur, die uns auch ein Profilbild erlaubt:

```
<link rel="import" href="bower_components/polymer/polymer.html">
 <dom-module id="name-tag">
   <template>
     <div><content select="img"></content></div>
     <h1>Hallo, mein Name ist <span id="name"><content></content></span></h1>
   </template>
   <script>
   Polymer({
     is: "name-tag",
     attached: function() {
       console.log(this.$.name.textContent)
     }
   });
   </script>
 </dom-module>
```

Die neue Version sieht auf den ersten Blick recht anders aus als unser vorheriger Entwurf. Besonders auffällig ist sicher die Verschachtelung in das `<dom-module>`-Element. Dieses Element ist ein nützlicher Helfer für ein weiteres Feature der Polymer-Plattform: das **Local DOM**. Das Local DOM bezeichnet den inneren Aufbau eines Polymer-Elements und wird über ein `<template>`-Element innerhalb des `<dom-module>`-Elements mit Inhalt aufgebaut. Dafür müssen wir dem `<dom-module>` eine ID geben, die mit dem Wert des `is`-Attributs im Aufruf von `Polymer(…)` übereinstimmt. Polymer stellt dann die Elemente, die über eine `id` verfügen, innerhalb des `<template>`-Elements über `this.$.ID` bereit:

```
<dom-module id="demo-elem">
   <template>
     <div id="greeting"></div> ❶
   </template>
   <script>
     Polymer({
       is: 'demo-elem',
       ready: function() {
         this.$.greeting.textContent = 'Hallo Welt!' ❷
       }
     })
   </script>
 </dom-module>
```

❶ Das `<div>` im Local DOM hat die ID `greeting`.

❷ Mit `this.$.greeting` kann im Skript des Elements auf dieses `<div>` zugegriffen werden.

Dieser Zugriff funktioniert im `attached`- oder `ready`-Callback.

Im Übrigen kann das `<script>`-Tag, sozusagen der imperative Teil unserer Elementdefinition, auch in einer separaten Datei oder außerhalb des `<dom-module>` liegen. Für alle anderen Elemente können wir Polymers eigene Alternative für `querySelector` benutzen: `this.$$(selektor)`, z. B. `this.$$('h1')`, um das `<h1>`-Element aus unserem Local DOM des `<name-tag>` zu adressieren.

Die Inhalte, die über `<content>`-Elemente in das Local DOM eingefügt werden, nennt man auch **Light DOM**. Grundsätzlich verhält sich dieses Light DOM ähnlich wie die eingefügten Inhalten im normalen Shadow DOM. Wir werden die Eigenheiten im nächsten Kapitel im Detail betrachten.

Unser Element kann jetzt einfach in jede Webanwendung eingefügt werden, unabhängig davon, ob diese Polymer benutzt oder nicht! Das funktioniert so:

```
<!doctype html>
 <html>
   <head>
     <link rel="import" href="name-tag-v2/name-tag.html">
   </head>
   <body>
     <name-tag>
       John Doe
       <img src="img/john-doe.jpg" alt="John Doe">
     </name-tag>
   </body>
 </html>
```

Da die Komponente selbst Polymer lädt, benötigt eine Anwendung nur den HTML-Import-Link für unsere `<name-tag>`-Komponente.

4.4 Properties als Element-API

Im letzten Abschnitt haben wir eine Komponente entwickelt, die Inhalte über `<content>`-Elemente in ein Local DOM einbettet. Es gibt aber viele Anwendungsfälle, bei denen wir weitere Optionen angeben wollen, die nicht direkt zum Inhalt gehören.

Nehmen wir als Beispiel ein Element mit dem wir Unicode-Smileys abstrahieren wollen. Nutzer unserer Komponente sollen zwei Dinge angeben können:

1. Welcher Smiley in Unicode erscheinen soll
2. Ob der Smiley größer als normal angezeigt werden soll

Dies stellt die API unserer Komponente dar und lässt sich am besten mit **Properties** implementieren. Bei Properties handelt es sich um eigene HTML-Attribute für unsere Komponenten, deren Wert an Eigenschaften der Komponenten-Instanz gebunden sind.

Die Komponente sieht in der Anwendung mit den Properties `smiley` und `large` wie folgt aus:

```
<!doctype html>
 <html>
   <head>
     <link rel="import" href="unicode-smiley/unicode-smiley.html">
   </head>
   <body>
     <unicode-smiley smiley=":-D" large="true"></unicode-smiley>
     <unicode-smiley smiley=":-)" large="true"></unicode-smiley>
     <unicode-smiley smiley=":-|" large="true"></unicode-smiley>
     <unicode-smiley smiley=":-(" large="true"></unicode-smiley>
     <unicode-smiley smiley=":-]" large="true"></unicode-smiley>
     <unicode-smiley smiley=";-)" large="true"></unicode-smiley>
     <unicode-smiley smiley=":-O" large="true"></unicode-smiley>
     <unicode-smiley smiley="B-)" large="true"></unicode-smiley>
     <unicode-smiley smiley=":-*" large="true"></unicode-smiley>
   </body>
 </html>
```

Properties werden über das `properties`-Objekt in der Elementdefinition festgelegt. Jedem Property wird mindestens sein Datentyp (dazu später mehr) zugeordnet. Zusätzlich können bei Bedarf auch ein Standardwert sowie einige weitere Eigenschaften angegeben werden.

In unserer `<unicode-smiley>`-Komponente benutzen wir `smiley` als `String` und `large` als `Boolean`:

```
<link rel="import" href="bower_components/polymer/polymer.html">
 <dom-module id="unicode-smiley">
   <template>
     <span id="displayedEmoji"></span>
   </template>
   <script>
   var KNOWN_EMOJIS = { ❶
     ':-D':  '&#x1f600;',
     ':-)':  '&#x1f60a;',
     ':-(':  '&#x2639;',
     ';-)':  '&#x1f609;',
     ':-]':  '&#x1f60b;',
     'B-)':  '&#x1f60e;',
     ':-*':  '&#x1f618;',
     ':-|':  '&#x1f610;',
     ':-O':  '&#x1f62e;'
   }
   Polymer({
     is: "unicode-smiley",
     properties: {
       smiley: String,
       large: Boolean
     },
     attached: function() {
       var emoji = KNOWN_EMOJIS[this.smiley]
```

```
        if(!emoji) emoji = '&#xfffd;' ❷
        this.$.displayedEmoji.innerHTML = emoji ❸
        this.$.displayedEmoji.style.fontSize = this.large ? '2em' : '1em' ❹
      }
    });
    </script>
  </dom-module>
```

❶ Wir definieren eine (private) Liste von unterstützten Smileys, deren Unicode-Äquivalente die Komponente unterstützt.

❷ Falls ein unbekannter Smiley angegeben wird, zeigt die Komponente das Unicode-Zeichen für `REPLACEMENT_CHARACTER` (ein Fragezeichen in einer Raute).

❸ Das Unicode-Zeichen wird in das Local DOM eingefügt.

❹ Wird `large` im HTML angegeben, wird die Schriftgröße auf das Doppelte der normalen Schriftgröße gesetzt.

4.4.1 Attribute, Properties und Eigenschaften

Die drei Begriffe **Attribut**, **Property** und **Eigenschaften** sind sich relativ ähnlich und in ihrer Bedeutung sehr nahe zu einander. Dieser Abschnitt soll als Hilfestellung dienen, die Begriffe besser auseinanderhalten zu können.

- *Eigenschaft*: Instanzen eines Objektes haben bestimmte Eigenschaften. Das Objekt `var image = {src: 'a.jpg', width: 300, height: 200}` hat etwa die Eigenschaften `src`, `width` und `height`.
- *Attribut*: HTML-Elemente können verschiedene Attribute haben. Ein Beispiel ist `<a id="home" href="/">` mit den Attributen `id` und `href`.
- *Property*: Properties sind für Polymer die Verbindung zwischen Eigenschaften und Attributen von Polymer-Elementen.

4.4.2 Namensgebung

Properties können einen beliebigen Namen anlegen. Attributnamen, die komplett kleingeschrieben sind, erhalten ein identisches Attribut. Namen in »camel case« werden zu Attributnamen mit Bindestrich, beispielsweise wird aus dem Property `userName` das Attribut `user-name`.

4.4.3 Datentypen

Properties werden explizit mit einem Datentyp definiert. Die möglichen Datentypen sind:

Datentyp	Beispiele	Beschreibung
`String`	`Hallo Welt`	Eine beliebige Zeichenkette
`Number`	`-42` oder `3.1415`	Beliebige Ganz- oder Kommazahl, positiv oder negativ.
`Boolean`	Nur `true` oder `false`	Ist nur `false`, wenn es nicht als Attribut in HTML angegeben wird. Sobald es im HTML angegeben wird, ist der Wert immer `true`.
`Date`	`Sat, 26 Mar 2016 22:53:48 GMT`	Der Wert wird als String an `new Date()` übergeben, kann also jedes Format benutzen, das korrekt vom `Date`-Konstruktor verarbeitet werden kann.
`Array`	`[1,2,3]`	Wird als String angegeben und mit `JSON.parse` in ein Array überführt. Für Strings sind doppelte Anführungszeichen zu verwenden.
`Object`	`{"name": "John", "lastName": "Doe"}`	Wird als String angegeben und mit `JSON.parse` in ein Objekt überführt. Hier sind immer doppelte Anführungszeichen (wie im Beispiel) zu verwenden.

Tab. 4–2 *Property-Datentypen*

Achtung

Die Datentypen `Boolean`, `Array` und `Object` haben verschiedene, teils überraschende Eigenheiten, die im nächsten Kapitel näher erläutert werden.

Mit Properties können teils sehr komplexe Verhaltensweisen und Schnittstellen implementiert werden. Dies wird im Detail im nächsten Kapitel behandelt, wenn wir uns komplexe und zusammengesetzte Properties, Observe und einiges mehr anschauen.

4.5 Data-Binding

4.5.1 Was ist Data-Binding?

Bislang waren unsere Anwendungen und Komponenten sehr einfach. Einzelne Komponenten haben ihre Daten aus einer Datenschicht geladen, sie allein verwaltet und mithilfe der Datenschicht gespeichert. In der Praxis sind Anwendungen aber deutlich komplexer und Daten werden über mehrere Komponenten hinweg genutzt und verändert.

Nehmen wir uns einen Bestellvorgang als Beispiel: Zunächst haben wir einen Warenkorb, in den wir Produkte einfügen. Sobald wir alle Produkte in den Warenkorb gelegt haben, können wir uns eine Übersicht anzeigen lassen, in der wir meistens Produkte auch wieder entfernen oder mehr oder weniger Einheiten eines Produktes auswählen können.

Wenn wir den Bestellvorgang verlassen und zur Anwendung zurückkehren, wird uns meistens angezeigt, dass wir noch einen gefüllten Warenkorb haben und wie viele Produkte sich in diesem befinden.

Der Warenkorb enthält also Daten, die in unterschiedlichen Arten und Weisen angezeigt und manipuliert werden können:

- *Produktansicht*: Produkte zum Warenkorb hinzufügen
- *Bestellübersicht*: Liste aller Produkte, Produkte entfernen oder Anzahl verändern
- *Zahlungsansicht*: Gesamtpreis
- *Restlicher Shop*: Anzahl Produkte, Gesamtpreis

Das bedeutet: Wann immer wir eine Änderung am Inhalt des Warenkorbs vornehmen, muss diese Änderung an die anderen Ansichten weitergegeben werden. In großen Anwendungen führt dies oft zu Problemen und Fehlern, weil nicht immer ersichtlich ist, wo eine Änderung der Daten berücksichtigt werden muss.

Data-Binding ist ein Mechanismus, um diese Aufgabe zu erleichtern. Beim Data-Binding wird der Zustand in einer gemeinsamen Variable festgehalten und der Zugriff auf den Zustand in einzelnen Komponenten an diese Variable **gebunden**. Ändert sich der Zustand, wird diese Änderung automatisch an die gebundenen Komponenten weitergereicht.

Ein einfaches Beispiel ist ein Binding zwischen zwei Komponenten:

```
<!doctype html>
 <html>
   <head>
     <link rel="import" href="name-tag-v2/bower_components/polymer/polymer.
     html">
     <link rel="import" href="name-tag-v2/name-tag.html">
   </head>
   <body>
     <dom-module id="app-main">
       <template>
         <name-tag>[[name]]</name-tag>
       </template>
       <script>
       Polymer({
         is: "app-main",
         ready: function() {
           this.name = "Alice"
         }
       });
       </script>
     </dom-module>
     <app-main></app-main>
   </body>
 </html>
```

Im Beispiel erzeugen wir eine Polymer-Komponente `<app-main>`, die wiederum die `<name-tag>`-Komponente nutzt. Die `<app-main>`-Komponente hat eine Eigenschaft `name` mit dem Wert `Alice`. Der Wert der Eigenschaft wird durch die Notation `[[name]]` in das Markup eingefügt und dem `<name-tag>` als Inhalt übergeben. Diese Verknüpfung ist dynamisch, wie wir leicht in der JavaScript-Konsole überprüfen können:

```
document.querySelector("app-main").name = "John"
```

Dieser Aufruf verändert nicht nur den Wert in der Element-Instanz, sondern führt auch automatisch zu einer Aktualisierung des HTML-Markups. Das Besondere daran ist, dass wir dafür selbst keinen zusätzlichen Code benötigen, sondern dass Polymer diese Änderung für uns automatisch durchführt und das DOM aktualisiert.

4.5.2 Bidirektionale Bindings

Bei der Notation mit doppelten eckigen Klammern handelt es sich um ein **Einweg-Binding**. Das bedeutet, dass Änderungen am Wert des Bindings an der Stelle keine Auswirkungen auf den Wert des Bindings an anderer Stelle haben. Dies gilt dabei sowohl für Eltern-Kind-Beziehungen zwischen Bindings (z. B. wenn ein Wert an ein Property einer Kind-Komponente gebunden ist) als auch innerhalb einer Komponente. Einweg-Bindings sind also nur zur **Ausgabe** des gebundenen Wertes geeignet. Ein Beispiel:

```
<!doctype html>
 <html>
   <head>
     <link rel="import" href="bower_components/polymer/polymer.html">
   </head>
   <body>
   <app-main></app-main>
   <dom-module id="app-main">
     <template>
       <div>[[someNumber]]</div>
       <increment-button value="[[someNumber]]"></increment-button>
     </template>
     <script>
     Polymer({
       is: 'app-main',
       properties: { someNumber: Number },
       attached: function() {
         this.someNumber = 1
       }
     });
     </script>
   </dom-module>
   <dom-module id="increment-button">
     <template>
```

```
      <button id="increment">[[value]]+1</button>
    </template>
    <script>
    Polymer({
      is: 'increment-button',
      properties: { value: Number },
      attached: function() {
        var self = this
        this.$.increment.addEventListener('click', function() {
          self.value++
        })
      }
    });
    </script>
  </dom-module>
  </body>
 </html>
```

Der Button zeigt den Wert seines `value`-Property und erhöht diesen bei einem Klick. Das gebundene Property `someNumber` des Elternelements wird dabei jedoch nicht erhöht, weil es mit einem Einweg-Binding gebunden wurde.

Wollen wir die Änderung korrekt an das Elternelement weiterreichen, so müssen wir ein bidirektionales Binding nutzen:

```
<!doctype html>
 <html>
   <head>
     <link rel="import" href="bower_components/polymer/polymer.html">
   </head>
   <body>
   <app-main></app-main>
   <dom-module id="app-main">
     <template>
       <div>{{someNumber}}</div>
       <increment-button value="{{someNumber}}"></increment-button> ❶
     </template>
     <script>
     Polymer({
       is: 'app-main',
       properties: { someNumber: Number },
       attached: function() {
         this.someNumber = 1
       }
     });
     </script>
   </dom-module>
   <dom-module id="increment-button">
     <template>
       <button id="increment">[[value]]+1</button>
     </template>
     <script>
     Polymer({
```

```
      is: 'increment-button',
      properties: {
        value: { type: Number, notify: true } ❷
      },
      attached: function() {
        var self = this
        this.$.increment.addEventListener('click', function() {
          self.value++
        })
      }
    });
    </script>
  </dom-module>
  </body>
</html>
```

❶ Das Binding wird mit der Syntax für bidirektionale Bindings an das Kindelement gegegeben: {{…}}.

❷ Das Kindelement muss für das Property, für das es ein bidirektionales Binding erlaubt, die `notify`-Option auf `true` setzen.

Bidirektionale Bindings haben eine Reihe von interessanten Eigenschaften, aber auch Tücken bei ihrer Nutzung. Diese Eigenschaften und Stolpersteine untersuchen wir im nächsten Kapitel näher.

4.6 Events in Polymer-Elementen

Nachdem mit Properties und Data-Binding jetzt Inhalte und Daten definiert und zwischen Komponenten ausgetauscht werden können, sollen Komponenten nun noch interaktiv werden. Für diese Aufgabe werden **Events** verwendet, genau wie im nativen DOM. Ein paar besondere Events dienen internen Zwecken. Etwa die `property-changed`-Events, die Polymer intern nutzt, wenn bidirektionale Bindings aktualisiert werden müssen.

4.6.1 Event-Behandlung in Polymer

In Polymer-Komponenten werden Events durch die `listeners`-Eigenschaft mit Funktionen (sogenannten **Listenern**) verknüpft:

```
Polymer({
   is: "app-main",
   listeners: {
     'click': 'clickAnything'
   },
   clickAnything: function(evt) {
     console.log('clicked', evt)
   }
 })
```

Wann immer etwas innerhalb der Komponente angeklickt wird, wird nun die Funktion `clickAnything` aufgerufen. Der `evt`-Parameter enthält, genau wie ein nativer Event-Listener, das Event, welches den Listener ausgelöst hat. Für `click` ist dies ein normales, natives `MouseEvent`. Über `evt.target` erhalten wir Zugriff auf das Objekt, welches den Klick empfangen hat:

```
<dom-module id="click-counter">
    <template>
      <div id="clicksCount">[[clicks]]</div>
      <button id="countAClick">Hier klicken</button> ❶
    </template>
    <script>
      Polymer({
        is: 'click-counter',
        properties: {
          clicks: { type: Number, value: 0 }
        },
        listeners: {
          'click': 'countClick'
        },
        countClick: function(e) {
          if(e.target.id === 'countAClick') this.clicks++ ❷
        }
      })
    </script>
  </dom-module>
```

❶ Der Button erhält im Local DOM die ID `countAClick`.

❷ Im Listener wird geprüft, ob das `click`-Event auf dem Button geschehen ist.

Es gibt allerdings eine kürzere, etwas übersichtlichere Variante mit demselben Effekt: `countAClick.click` anstatt `click` als Event-Spezifikation angeben.

```
listeners: {
    'countAClick.click': 'countClick'
  },
  countClick: function(e) {
    this.clicks++
  }
```

Dies gilt generell: `id.eventType` kann verwendet werden, um Events nur auf dem Element mit der entsprechenden ID zu behandeln.

Hinweis

Je nach Browser ist das `click`-Event auf mobilen Geräten spürbar verzögert. Polymer bietet das `tap`-Event als Alternative an, die sowohl auf Touchscreen als auch auf Maus-Events sofort reagiert.

Neben dieser imperativen Variante gibt es auch die Möglichkeit, Listener deklarativ im HTML anzubringen:

```
<dom-module id="click-counter">
    <template>
      <div id="clicksCount">[[clicks]]</div>
      <button on-click="countClick">Hier klicken</button> ❶
    </template>
    <script>
      Polymer({
        is: 'click-counter',
        properties: {
          clicks: { type: Number, value: 0 }
        },
        countClick: function(e) {
          this.clicks++ ❷
        }
      })
    </script>
  </dom-module>
```

❶ Mit `on-click` wird der Listener deklarativ an das `<button>`-Element gebunden.

❷ Filtern ist nicht mehr nötig, denn es werden nur die relevanten Events für das `<button>`-Element abgefangen.

Da HTML-Attribute aber nicht zwischen Groß- und Kleinschreibung in ihrem Namen unterscheiden, wird bei deklarativen Event-Listenern der Attributname immer nur in Kleinbuchstaben akzeptiert. Beispielsweise ein `on-firstRender` würde zu `on-firstrender`.

Warnung

Der Name der Listener-Funktion hingegen macht einen Unterschied zwischen Groß- und Kleinschreibung!

4.6.2 Dynamische Listener

Listener können auch zur Laufzeit des Programms angelegt und dann je nach Belieben wieder abgeschaltet werden. Dafür stehen in Polymer die Funktionen `listen` und `unlisten` zur Verfügung:

```
Polymer({
    is: "click-counter",
    properties: { active: Boolean },
    activate: function() {
      if(!this.active) {
        this.listen(this.$.countAClick, 'tap', 'countClick') ❶
```

```
    } else {
      this.unlisten(this.$.countAClick, 'tap', 'countClick') ❷
    }
    this.active = !this.active
  },
  countClick: function(e) {
    this.clicks++
  }
})
```

❶ `this.listen` verknüpft `tap`-Events auf dem Element mit ID `countAClick` mit der `countClick` Funktion.

❷ `this.unlisten` löst die Verknüpfung von Event, Element und Listener und deaktiviert die Event-Behandlung wieder.

Dabei macht `unlisten` keinen Unterschied zwischen imperativen (über `listeners` oder `listen`) oder deklarativen Event-Listener-Definitionen:

```
Polymer({
  is: "click-counter",
  listeners: {
    'countAClick.tap': 'countClick'
  },
  killEvents: function() {
    this.unlisten(this.$.countAClick, 'tap', 'countClick')
  },
  countClick: function(e) {
    this.clicks++
  }
})
```

Der obige Code-Ausschnitt hat bei Aufruf von `killEvents` denselben Effekt, als wäre der Listener mittels `listen` erzeugt worden. Gleiches gilt für deklarative Definition, womit sich der folgende Code identisch verhält:

```
<template>
  <button on-tap="countClick">Klick mich</button>
</template>
<script>
Polymer({
  is: "click-counter",
  killEvents: function() {
    this.unlisten(this.$.countAClick, 'tap', 'countClick')
  },
  countClick: function(e) {
    this.clicks++
  }
})
</script>
```

Warnung

Ein wichtiger Unterschied zwischen `listeners`, `on-event` und `listen` besteht allerdings darin, dass Listener, die in `listeners` und `on-event` definiert werden, auch automatisch wieder entfernt werden, wenn das Element aus dem DOM entfernt wird. Bei Listenern, die per `listen` hinzugefügt worden sind, muss dies manuell geschehen. Ein guter Ort dafür ist der `detached`-Callback.

4.6.3 Eigene Events definieren

Manchmal sind Properties und Data-Binding nicht die besten Wege, Informationen zwischen Komponenten auszutauschen, etwa wenn es um die Behandlung von Netzwerk-Events (z. B. bei einem fehlgeschlagenen API-Request) oder um Änderungen des Zustands geht, die sich nicht direkt in der Darstellung der Anwendung äußern, sondern lediglich einen bestimmten Code-Pfad auslösen müssen. Diese Situationen lassen sich alle mit Properties und Observern realisieren, aber dies erscheint umständlich, wenn es eigentlich um die Behandlung eines Events geht. Eigene Events werden erzeugt, indem die `fire`-Methode mit dem frei wählbaren Namen des Events aufgerufen wird:

```
Polymer({
   is: 'hi-sayer',
   properties: {
     message: String,
   },
   listeners: {
     'tap.greetButton': 'sayHi'
   },
   sayHi: function(newContent) {
     this.fire('greetings')
   }
 })
```

Das Event kann im Elternelement jetzt wie üblich behandelt werden:

```
Polymer({
   is: 'app-main',
   listeners: {
     'greetings': 'sayHello'
   },
   sayHello: function(evt) {
     alert('Hallo Welt!')
   }
 })
```

Das funktioniert auch mit regulären DOM-Eventhandlern:

```
document.querySelector('hi-sayer').addEventListener('greetings', function() {
   alert('Hallo Welt!')
 })
```

Events können auch mit zusätzlichen Informationen versehen werden:

```
Polymer{
    // ...
    sayHi: function(newContent) {
      this.fire('greetings', {text: 'Hallo, ' + this.name})
    }
 })
```

Der Zugriff auf dieses zusätzliche Objekt ist im Listener über die `detail`-Eigenschaft des Event-Objektes möglich:

```
Polymer({
    is: 'app-main',
    listeners: {
      'greetings': 'sayHello'
    },
    sayHello: function(evt) {
      alert(evt.detail.text)
    }
 })
```

Analog dazu funktionieren sowohl Listener, die per `listen` definiert wurden, als auch native Listener per `addEventListener` und Listener aus einer deklarativen Definition.

4.6.4 Gesten in Polymer

Wir haben bereits ein Event kennengelernt, das Polymer mitbringt, um die Verarbeitung von Benutzerinteraktionen zu vereinfachen: das `tap`-Event, welches sowohl auf Klicks als auch auf Touch-Events reagiert.

Polymer bringt eine Reihe weiterer solcher Events mit, die helfen, sowohl Maus- als auch Touch-Events in konsistenter Art und Weise zu behandeln. Diese Events werden von Polymer als **Gesten** bezeichnet. Folgende Gesten sind verfügbar:

Event	Beschreibung
`down`	Finger auf dem Bildschirm bzw. Maustaste gedrückt (`touchstart` bzw. `mousedown`)
`up`	Finger vom Bildschirm entfernt bzw. Maustaste losgelassen (`touchend` bzw. `mouseup`)
`tap`	Behandelt Klicks bzw. »Taps« sowohl per Maus als auch durch Antippen eines Touchscreens mit dem Finger. Entspricht der Folge `down` & `up` bzw. dem `click`-Event.
`track`	Behandelt die Bewegung mit dem Finger auf dem Bildschirm (`touchmove`) bzw. die Mausbewegung bei gedrückter Maustaste (`mousemove` und `mousedown`).

Tab. 4–3 *Gesten-Events*

Die Events enthalten jeweils eine `detail`-Eigenschaft mit zusätzlichen Informationen. Alle Events enthalten `event.detail.x` und `event.detail.y` mit der Bildschirmposition, an der das Event ausgelöst wurde.

Die `up`-, `down`- und `tap`-Events haben darüber hinaus noch das `sourceEvent`, in dem das native Event (also z. B. `mouseUp` oder `touchend`) angesprochen werden kann.

4.6.5 Die track-Geste

Das `track`-Event wird mehrfach ausgelöst, während ein Finger auf dem Bildschirm oder eine Maustaste gedrückt ist und eine Bewegung stattfindet. Der Zustand der Geste ist in diversen Eigenschaften des `event.detail`-Objektes ablesbar:

In der Eigenschaft `state` werden die folgenden Werte geliefert:

Wert	Bedeutung
`start`	Es wurde ein `touchstart`- oder `mousedown`-Event zusammen mit einer Bewegung (es gibt einen Schwellenwert, ab dem eine Bewegung erst als solche gilt) registriert.
`track`	Die Bewegung wurde fortgesetzt.
`end`	Das Ende des Stroms aus `track`-Events, `touchend` / `touchcancel` oder `mouseup` wurde registriert.

Tab. 4–4 *Werte in* `event.detail.state` *des* `track`*-Events*

- `dx` und `dy` geben an, um wie viele Pixel der Cursor (bzw. Finger) seit dem *ersten* `track`-Event in diesem Zyklus (also zwischen `start`-`track`-`end`) bewegt wurde.
- `ddx` und `ddy` geben an, um wie viele Pixel der Cursor (bzw. Finger) seit dem unmittelbar *vorhergegangenen* `track`-Event bewegt wurde.

Außerdem können wir `hover()` auf dem Event aufrufen, um herauszufinden, über welchem Element der Cursor sich gerade befindet.

Warnung

Scrolling auf Elementen, die einen `track`-Listener nutzen, ist nicht per Touch-Events möglich! Das heißt: Sobald das `track`-Event auf einem Element aktiviert ist, kann innerhalb des Elements oder während der Finger über dem Element ist nicht mehr gescrollt werden.

Hier ist ein Beispiel, das `track` nutzt, um die Richtung der Bewegung bei einem Finger-Touch oder Ziehen der Maus mit gedrückter Maustaste in Textform anzuzeigen:

```
<link rel="import" href="bower_components/polymer/polymer.html">
  <dom-module id="pointer-tracker">
    <template>
      <div id="box" style="display:inline-block;width:500px;height:
      500px;background:gray"></div> ❶
      <div>[[direction]]</div>
    </template>
    <script>
      Polymer({
        is: 'pointer-tracker',
        properties: {
          direction: String
        },
        listeners: {
          'box.track': 'pickColour' ❷
        },
        pickColour: function(evt) {
          var yDirection = '', xDirection = ''
          if(evt.detail.ddy < 0) { ❸
            yDirection = 'upwards'
          } else if(evt.detail.ddy > 0) {
            yDirection = 'downwards'
          }
          if(evt.detail.ddx < 0) {
            xDirection = 'to the left'
          } else if(evt.detail.ddx > 0) {
            xDirection = 'to the right'
          }
          var textDescription = (evt.detail.state === 'end' ? 'Moved' :
          'Moving')
          if(yDirection !== '' && xDirection !== '') textDescription += ' ' +
          yDirection + ' and ' + xDirection
          else if(yDirection !== '') textDescription += ' ' + yDirection
          else if(xDirection) textDescription += ' ' + xDirection
          if(textDescription != this.direction) this.direction =
          textDescription
        }
      })
    </script>
  </dom-module>
```

❶ Da unsere Events nur auf dem Inhalt der Komponente ausgelöst werden, geben wir uns eine 500x500 Pixel große Box …

❷ … und konsumieren das `track`-Event auf dieser Box.

❸ `ddx` und `ddy` geben uns die aktuelle Bewegungsrichtung.

4.6.6 Events und das Shadow DOM

Bei der Betrachtung des Shadow DOM wurden bislang immer die Isolation und deren Nutzen erläutert. In der Verwendung von Events und dem speziellen Verhalten von Events im DOM ergibt sich aber eine Herausforderung:

Events enthalten ein `target`-Property, das auf ein DOM-Element verweist, in dem das Event aufgetreten ist. DOM-Events haben aber zwei spannende Verhaltensweisen: die **capture**- und die **bubble**-Phase (auch **Propagation** genannt).

Ein Event wird zunächst während der Capturing-Phase vom `<html>`-Element aus durch alle Elemente, die auf dem Pfad zum `target`-Element liegen, gereicht. Im `target`-Element geht es dann in die Propagation-Phase (»bubbling«) über, in der es so lange an das jeweilige Elternelement gereicht wird, bis es entweder wieder das `<html>`-Element erreicht hat oder ein Listener auf dem Weg dorthin `stopPropagation` aufruft und die Propagation stoppt.

Hinweis

Genaugenommen beginnen das Capturing und Bubbling im `window`-Objekt.

Dieses Verhalten ist sehr nützlich, weil so oft viele Listener eingespart werden können und so weniger Speicher und Rechenzeit verbraucht wird. Ein Beispiel ist eine Liste bei der durch einen Klick auf einen Eintrag der Inhalt dieses Eintrags separat dargestellt werden soll:

```
<!doctype html>
 <html>
 <head>
   <style>
     li article { display: none; }
   </style>
 </head>
 <body>
   <ul>
     <li> ❶
       Erster Artikel
       <article>Inhalt des ersten Artikels in voller Länge</article> ❷
     </li>
     <li>
       Zweiter Artikel
       <article>Inhalt des zweiten Artikels in voller Länge</article>
     </li>
     <li>
       Dritter Artikel
       <article>Inhalt des dritten Aritkels in voller Länge</article>
     </li>
   </ul>
   <div id="display"></div>
```

```
    <script>
    var list = document.querySelector('ul'),
        display = document.getElementById('display')
    list.addEventListener('click', function(evt) { ❸
      display.textContent = evt.target.querySelector('article').textContent ❹
    })
    </script>
  </body>
  </html>
```

❶ Die Listeneinträge sind später das Ziel des Events (also `event.target`).

❷ Die vollen Artikel werden ausgeblendet. Sie sind quasi nur Container, um den Inhalt später von hier auslesen zu können.

❸ Der Event-Listener wird auf die **Liste** gebunden, nicht auf einzelne Einträge.

❹ Der Inhalt des `<div>` wird auf den Inhalt des `<article>`-Elements innerhalb des angeklickten `<li>` Elements gesetzt.

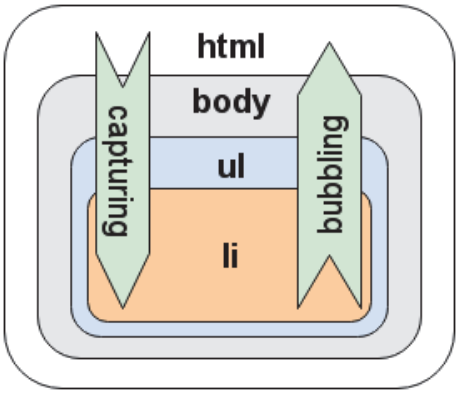

Abb. 4–1 *Event-Capture und -Bubbling*

Während der Propagation bleibt das `target` immer das Element, auf dem das Event ausgelöst worden ist, in diesem Fall das `<li>`. Das Problem ergibt sich durch die Tatsache, dass ein Event auf seinem »Weg nach oben« Shadow-DOM-Grenzen übertreten kann und das `target`-Property nach dem Überschreiten der Shadow-DOM-Grenze ein Element enthält, das vor Zugriffen aus dem darüberliegenden DOM geschützt ist und aus dessen Sicht es nicht existiert.

Dadurch würde die Isolation des Shadow DOM verletzt, da jedes Event die versteckten Elemente innerhalb des Shadow DOM preisgeben würde. Schlimmer: Man könnte sich nicht darauf verlassen, auf das `target`-Element eines Events zuzugreifen, da dies zu einem Fehler führen könnte, wenn dieses Element innerhalb eines Shadow DOM liegt.

Diesem Problem begegnen Browser mit dem sogenannten **Event Retargeting**. Überschreitet ein Event ein Shadow-Root, so wird das `target`-Element des

Events auf das Host-Element (also das Element, an welches der Shadow-Tree gebunden ist) gesetzt.

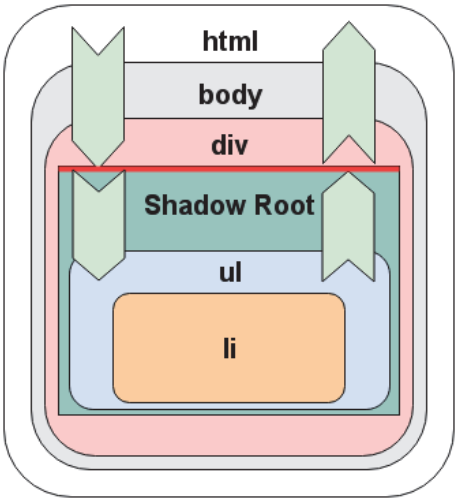

Abb. 4–2 *Event-Capture und -Bubbling und das Shadow DOM*

Hier kann folgendes beobachtet werden:

```
<div> ❶
   <ul></ul> ❷
 </div>
 <script>
   var shadowRoot = document.querySelector('ul').createShadowRoot()
   var item = document.createElement('li')
   item.textContent = 'Klick mich!'
   shadowRoot.appendChild(item)
   document.querySelector('div').addEventListener('click', function(evt) {
     console.log(evt.target) ❸
     console.log(evt.path) ❹
   })
 </script>
```

❶ Hier werden wir den Event-Listener anbringen.

❷ Die Liste erhält ein Shadow DOM.

❸ Das `target`-Property zeigt auf `<ul>` statt auf das `<li>`.

Das native Shadow DOM erlaubt aber dennoch einen gezielten Zugriff auf das originale Target über das `path`-Property des Events:

```
console.log(event.path)
 [li, document-fragment, ul, div, body, html, document, Window]
```

4.6.7 Event-Retargeting und Shady DOM

Event-Retargeting ist ein sehr nützliches Feature des nativen Shadow DOM, aber ein Polyfill (Polyfills) für dieses Verhalten ist mit einer gewissen Performanceeinbuße verbunden. Aus diesem Grund geschieht Event-Retargeting in Polymers Shady DOM Polyfill nicht automatisch. Es kann aber mit `Polymer.dom(event)` emuliert werden, wenngleich es sich auch nicht exakt identisch verhält:

- `event.target` enthält in Browsern mit Shadow DOM das Host-Element des Shadow DOM, aber in Browsern ohne natives Shadow DOM das Element innerhalb des Shadow DOM statt des Host-Elements des Shadow DOM.
- `event.rootTarget` enthält in jedem Fall das Element innerhalb des Shadow DOM.
- `event.localTarget` enthält in jedem Fall das Host-Element des Shadow DOM.
- `event.path` wird auch ohne natives Shadow DOM korrekt befüllt.

Damit stehen `event.rootTarget`, `event.localTarget` und `event.path` in einer einheitlichen Art und Weise zur Verfügung, unabhängig vom Vorhandensein einer nativen Shadow-DOM-Implementierung.

4.7 Styling und Theming in Polymer-Elementen

Ähnlich zu unseren nativen Web Components aus Kapitel 2 und 3 werden CSS-Definitionen in einem `<style>`-Element innerhalb des `<template>`-Elements angegeben.

Polymer sorgt dann dafür, dass diese CSS-Definitionen auch ohne natives Shadow DOM auf die Komponente beschränkt bleiben. Es ist aber auch möglich, CSS-Definitionen, die für mehrere Elemente genutzt werden sollen, in sogenannte **Style Modules** auszulagern. Mehr dazu später in diesem Kapitel.

4.7.1 Besonderheiten von CSS für Shady DOM

In nativem Shadow DOM stehen uns verschiedene Selektoren zur Verfügung, um innerhalb des Shadow DOM Elemente in CSS zu selektieren:

- die `:host`-Pseudoklasse und die `:host()`-Variante davon
- die `:host-context`-Pseudoklasse
- das `::content`-Pseudoelement

Damit kann Shadow DOM das sogenannte **scoping** der CSS-Definitionen durchführen. Das Shady DOM hingegen muss darauf zurückgreifen, die Style-Defini-

tionen auf die jeweilige Komponente zu spezifizieren. Dafür schreibt Polymer die CSS-Selektoren um, zum Beispiel:

```
<dom-module id="some-element">
   <template>
     <style>
       h1 {
         color: red;
       }
     </style>
 </dom-module>
```

Im Beispiel nutzt das native Shadow DOM die browserseitige Isolation des `h1`-Selektors und die Styles sind auf die Komponente isoliert. Falls kein natives Shadow DOM verfügbar ist, muss Shady DOM einspringen, indem es den Selektor umschreibt zu:

```
some-element h1 {
   color: red;
 }
```

Dadurch wird der Style ebenfalls auf die Komponente begrenzt. Dieses Umschreiben hat aber seine Grenzen:

- *Definierte CSS-Properties* (mehr dazu später in diesem Abschnitt) funktionieren nicht auf eingefügten Elementen.
- Das `::content`-Pseudoelement wird entfernt, was zu eigenartigen Nebeneffekten führen kann.

Achtung

Der Fakt, dass `::content` von Shady DOM vollständig aus dem Selektor entfernt wird, kann zu schwer identizierbaren Problemen mit fehlerhaftem CSS führen!

Insbesondere der Kindelement-Kombinator (`>`) kann Probleme verursachen, wenn Shady DOM das `::content`-Pseudoelement entfernt. Hier ein verdeutlichendes Beispiel:

```
<dom-module id="tricky-styles">
   <template>
     <style>
       span {
         color: red; ❶
       }
       ::content > span {
         color: green; ❷
       }
     </style>
     <span>Direct child, inside</span> ❸
```

```
      <content select=".first"></content> ❹
      <div>
        <span>Child of a child, inside</span> ❺
        <content></content> ❻
      </div>
    </template>
    <script>
      Polymer({is: 'tricky-styles'})
    </script>
  </dom-module>
```

❶ Alle `<span>`-Elemente innerhalb des Elements sollen rot sein.

❷ Alle direkten Kinder des `<content>`-Elements, die ein `<span>`-Element sind, sollen hingegen grün sein.

❸ Außerhalb des `<content>`-Elements, daher erwarten wir hier Rot.

❹ Innerhalb des `<content>`-Elements und dessen direktes Kind, daher erwarten wir hier Grün.

❺ Außerhalb des `<content>`-Elements, daher erwarten wir hier Rot.

❻ Innerhalb des `<content>`-Elements und dessen direktes Kind, deshalb erwarten wir hier Grün.

Im Shadow DOM erhalten wir die beschriebene Abfolge von Rot, Grün, Rot und **Grün** für die vier Textelemente. Im Shady DOM hingegen erhalten wir Rot, Grün, Rot und **Schwarz**. Ein Blick in die umgeschriebenen CSS-Definitionen verrät den Grund.

Unsere ursprünglichen CSS-Selektoren waren:

```
span { color: red; }
 ::content > span { color: green; }
```

Die umgeschriebenen Selektoren hingegen lauten:

```
span.tricky-styles { color: red; }
 tricky-styles > span { color: green; }
```

Hier ist was beim Umschreiben geschieht:

- Das Shady DOM hat allen direkten Kindelementen in der Polymer-Komponente eine Klasse hinzugefügt, die den selben Namen trägt, wie die Komponente (im obigen Beispiel also `tricky-styles`).
- Das `::content`-Pseudoelement wurde entfernt und der Name der Komponente wurde davorgesetzt, um nur innerhalb dieser Komponente Geltung zu haben.

Das Problem dabei ist, dass sich das `<content>`-Element innerhalb eines Kindelements befunden hat und somit kein direktes Kind von `<tricky-styles>`, sondern ein Kind des `<divs>` ist. Der Selektor greift aber nun anstatt auf den direkten

Kindelementen des `<content>`-Elements auf den direkten Kindelementen des `<tricky-styles>`-Elements.

Das Problem lässt sich über einen alternativen Selektor lösen, will aber gut überlegt sein. Beispielsweise hilft es nicht, den ursprünglichen Selektor von `::content > span` auf `div ::content span` zu ändern, weil dies nach dem Umschreiben auch das Kindelement betrifft, welches nicht per `<content>` eingefügt wurde (d. h., das Element mit dem Text »Child of a child, inside« würde ebenfalls grün dargestellt).

Eine Methode, um in diesem Fall eine konsistente Darstellung sowohl im Shadow- als auch im Shady DOM zu erzielen, ist die Nutzung einer Kombination aus einem Wrapper-Element und dem `::content` Pseudoelement:

```
<template>
   <style>
     span {
       color: red;
     }
     .content-wrapper ::content > span { ❶
       color: green;
     }
   </style>
   <span>Direct child, inside</span>
   <div class="content-wrapper"><content select=".first"></content></div> ❷
   <div>
     <span>Child of a child, inside</span>
     <div class="content-wrapper"><content></content></div> ❸
   </div>
 </template>
```

❶ Der Selektor ist sowohl auf die `content-wrapper`-Klasse als auch auch das `::content` Pseudoelement bezogen.

❷ Das `<content>`-Element wird in das Wrapper-Element eingebettet.

❸ Jedes `<content>`-Element, das von den Styles beeinflusst werden soll, muss in einem Wrapper platziert werden.

Damit wird der Selektor im Shady DOM umgeschrieben zu `.content-wrapper > span` und es werden keine zusätzlichen Klassen an das `<span>` angefügt. Im Shadow DOM ist der Selektor in der Wirkung identisch, da in dem Fall durch `::content` ebenfalls nur die direkten Kindelemente des Typs `<span>` selektiert werden.

4.7.2 CSS-Custom-Properties und Mixins

Im Zusammenhang mit Events kam schon ein Nachteil der Isolation durch das Shadow DOM zur Sprache und auch für CSS kann die Isolation hinderlich sein.

Die Isolation schützt vor versehentlichen Änderungen des Aussehens. Wie aber lassen sich gewollte Änderungen der CSS-Definitionen realisieren, um den Nutzern einer Komponente die Möglichkeit zu geben, bestimmte Eigenschaften des CSS anpassen zu können?

Polymer nutzt dafür **Custom-CSS-Properties**, die auch als **CSS-Variablen** bezeichnet werden, ein weiterer neuer Webstandard, der es erlaubt, Variablen in CSS zu definieren. Diese Variablen ermöglichen es, abhängig von den CSS-Definitionen im Kontext des Elements unterschiedliche Werte festzulegen, ohne das CSS anpassen zu müssen.

Ein Custom-CSS-Property wird so definiert:

```
menu-toolbar {
   background: var(--menu-toolbar-color);
 }
```

Das Property `--menu-toolbar-color` kann dann so mit einem Wert belegt werden:

```
body {
   --menu-toolbar-color: limegreen;
 }
```

Als Beispiel könnte eine `<toggle-button>`-Klasse eine Option für die Hintergrundfarbe angeben:

```
<dom-module id="toggle-button">
   <template>
     <style>
     button {
       display: inline-block;
       position: relative;
       width: 3em;
       margin: 0;
       padding: 0;
       border: 1px solid black;
     }
     .on {
       background: var(--toggle-on-color); ❶
       color: var(--toggle-on-text-color); ❷
       border-radius: 10px 0 0 10px;
       border-right: none;
     }
     .off {
       background: var(--toggle-off-color, black); ❸
       color: var(--toggle-off-text-color, white); ❹
       border-radius: 0 10px 10px 0;
       border-left: none;
     }
     </style>
     <button class="on">
```

```
        <span>I</span>
      </button><button class="off">
        <span>O</span>
      </button>
    </template>
    <script>
    Polymer({
      is: 'toggle-button'
    })
    </script>
  </dom-module>
```

❶ Es wird ein `--toggle-on-color`-Property definiert, dessen Wert für `background` im ersten Button genutzt wird.

❷ Mit `--toggle-on-color` kann dann das `color`-Property des ersten Buttons beeinflusst werden.

❸ Bei der Definition eines Custom-CSS-Property kann auch ein Standardwert angegeben werden, wie hier `black`.

❹ Auch das `color`-Property des zweiten Buttons wird mit einer CSS-Variablen konfigurierbar.

Warnung

Es gibt zwei Einschränkungen auf die man achten muss:

- Der Polyfill für CSS-Variablen ist nicht in der Lage, diese Properties auch auf Elemente innerhalb von `<content>`-Elementen einzusetzen.
- Die Belegung von CSS-Variablen für Polymer-Elemente funktioniert nur aus einem `dom-bind` oder einem anderen Polymer-Element heraus.

Die Custom-CSS-Properties bieten damit eine Art API für das Anpassen der Darstellung einer Polymer-Komponente, die durch einen Polyfill auf allen von Polymer unterstützten Browsern zur Verfügung steht und konsistent bei Verwendung von Shadow DOM und Shady DOM funktioniert.

Auch wenn die CSS-Variablen sehr mächtig und nützlich sind, so ist es doch auf Dauer etwas lästig, für jedes CSS-Property eine zugehörige CSS-Variable anzulegen. Der Polyfill (Polyfills) für CSS-Variablen unterstützt noch einen zusätzlichen Mechanismus, um statt eines einzelnen Werts eine ganze Reihe von CSS-Anweisungen zu definieren und über Komponenten hinweg zu nutzen: **Mixins** mit der `@apply` Regel.

Mixins werden genau wie CSS-Variablen definiert, können aber mehrere CSS-Definitionen enthalten:

```
<dom-module id="app-main">
   <template>
```

```
    <style>
      :host {
        --message-box-theme: {
          border: 1px solid blue;
          border-radius: 0.5em;
          background-color: #eef;
          padding: 0.5em 1em;
        }
      }
    </style>
    <message-box>Hello World</message-box>
    <!-- Rest des Templates -->
  </template>
  <script>
    Polymer({is: 'app-main'})
  </script>
</dom-module>
```

Warnung

Mixin-Definitionen können, genau wie CSS-Variablen-Definitionen, nur innerhalb von Polymer-Elementen eingesetzt werden.

Um eine Komponente mit einem Mixin zu stylen, muss die Komponente das Mixin mit `@apply` laden:

```
<dom-module id="message-box">
  <template>
    <style>
      :host {
        @apply(--message-box-theme);
      }
    </style>
    <!-- Rest des Templates -->
  </template>
  <script>
  Polymer({is: 'message-box'})
  </script>
</dom-module>
```

4.7.3 Styles dynamisch ändern und abrufen

CSS-Variablen und Mixins werden (zumindest im Polyfill) nur bei der Erzeugung von Elementen (`created`) angewendet. Der Wert einer CSS-Variablen innerhalb eines Elements kann mit `.getComputedStyleValue()` abgerufen werden, wobei der Name der gewünschten CSS-Variable übergeben werden muss:

```
elem.getComputedStyleValue("--unread-message-size")
```

Wird der Wert einer CSS-Variable über das `customStyle`-Objekt gesetzt, so ist der neue Wert erst aktiv, wenn `updateStyles` aufgerufen wurde:

```
elem.customStyle['--unread-message-size'] = '2em'
 elem.customStyle['--unread-message-color'] = '#ffe'
 elem.updateStyles()
```

Alternativ kann `updateStyles` auch direkt mit einem Objekt aufgerufen werden, in dem die zu ändernden Variablen aufgeführt werden:

```
elem.updateStyles({'--unread-message-size': '2em', '--unread-message-color':
  '#ffe'})
```

Darüber hinaus steht `updateStyles` auch global zur Verfügung, um alle Elemente zu updaten. Dafür wird `Polymer.updateStyles` statt `elem.updateStyles` genutzt. Über das `getComputedStyleValue` können die Werte von CSS-Variablen auch ausgelesen werden:

```
elem.getComputedStyleValue('--unread-message-color') ❶
```

❶ Ergibt zum Beispiel `#ffe`.

4.7.4 Einschränkungen des Polyfills

In nativen Custom-CSS-Properties ist der `updateStyles` Aufruf nicht notwendig, da der Browser dies automatisch handhabt. Aus Performancegründen verzichtet der Polyfill aber darauf dieses Verhalten zu imitieren.

Die Einschränkungen auf einen Blick:

- Nur eingeschränkt dynamische CSS-Variablen (Aufruf von `updateStyles` nötig)
- Elemente in `<content>` können nicht mit CSS-Variablen beeinflusst werden.
- Eine CSS-Variable kann nur genau einen Wert innerhalb eines CSS-Kontexts haben (mehr dazu im Anschluss).

Die eingeschränkte Dynamik von CSS-Variablen bezieht sich auf die *Änderung* der Werte von CSS-Variablen. Änderungen der verwendeten Variablen hingegen sind nicht davon betroffen.

Das Hinzufügen oder Entfernen der Klasse `unread` auf einem Element mit den folgenden CSS-Anweisungen im Template funktioniert also zur Laufzeit ohne `updateStyles`:

```
:host {
   background-color: var(--notification-color); ❶
 }
 :host.unread {
   background-color: var(--notification-unread-color); ❷
 }
```

❶ Es wird immer eine Variable angewendet.

❷ Im Fall der Klasse `unread` wird eine **andere** Variable verwendet.

Die Beschränkung der CSS-Custom-Properties auf einen Wert innerhalb eines CSS-Kontexts ist so zu verstehen, dass der Wert eines solchen Property sich nur durch Vererbung über mehrere Elemente hinweg ändern kann, nicht innerhalb eines Elements:

```
:host {
   --main-color: blue; ❶
 }
 .titlebar {
   --main-color: green; ❷
 }
```

❶ Erste Zuweisung innerhalb des aktuellen CSS-Kontexts.

❷ Erneute Zuweisung, die vom Polyfill ignoriert wird.

Vererbung funktioniert weiterhin, solange dabei der Kontext gewechselt wird:

```
<dom-module id="outer-element">
   <template>
     <style>
       :host {
         --toggle-on-color: #0f0;
       }
     </style>
     <toggle-button></toggle-button> ❶
     <inner-element></inner-element>
   </template>
   <script>Polymer({is: 'outer-element'})</script>
 </dom-module>
 <dom-module id="inner-element">
   <template>
     <style>
       :host {
         --toggle-on-color: #08f;
       }
     </style>
     <toggle-button></toggle-button> ❷
   </template>
   <script>Polymer({is: 'outer-element'})</script>
 </dom-module>
```

❶ Kontext des `<outer-element>`, `--toggle-on-color` hat den Wert `#0f0`.

❷ Kontext des `<inner-element>`, `--toggle-on-color` hat den Wert `#08f`.

4.7.5 Styles und Custom-CSS-Properties auf Dokumentenebene

Bei Verwendung des Shady DOM Polyfills wird die Isolation von CSS mittels zusätzlicher Klassen implementiert. Dies kann für unangenehme Überraschungen sorgen, wenn CSS direkt im Anwendungsdokument angegeben wird, da dort manuell darauf geachtet werden muss, diese besonderen Klassen zu berücksichtigen.

Hier ein Beispiel, in dem die Isolation versehentlich verletzt wird:

```
<dom-module id="my-element">
   <template>
     <div>Hallo Komponente!</div> ❶
   </template>
   <script>Polymer({is: 'my-element'})</script>
 </dom-module>
 <style>
   div { color: red; } ❷
 </style>
 <div>Hallo Welt!</div>
 <my-element></my-element>
```

❶ Wird unter Shady DOM zu `<div class="style-scope my-element">`.

❷ Ignoriert die `style-scoped`-Klasse und damit die Isolation.

Das lässt sich mit der `:not(.style-scope)`-Pseudoklasse beheben:

```
div:not(.style-scope) {
   color: red;
 }
```

Diese Pseudoklasse müsste an jede CSS-Definition angehängt werden, was in den meisten Fällen schnell lästig wird und ein stetig steigendes Risiko birgt, unerwartete Probleme zu erzeugen. Eine bessere Alternative ist Polymers `custom-style`-Element, welches als Typerweiterung für `<style>` Elemente dazu dient, solche dokumentenweiten CSS-Definitionen so anzupassen, dass diese nicht versehentlich im Shady DOM die Isolationsgrenzen überschreiten. Das vorige Beispiel mit `custom-style` sieht dann so aus:

```
<dom-module id="my-element">
   <template>
     <div>Hallo Komponente!</div> ❶
   </template>
   <script>Polymer({is: 'my-element'})</script>
 </dom-module>
 <style is="custom-style">
   div { color: red; } ❷
 </style>
 <div>Hallo Welt!</div>
 <my-element></my-element>
```

❶ Wird von Shady DOM umgeschrieben

❷ Wird automatisch umgeschrieben zu `div:not(.style-scope)`

Achtung

Diese Isolation funktioniert **nicht** für den Stern-Selektor (*)! Der Stern-Selektor funktioniert generell nicht mit der `:not`-Pseudoklasse und somit auch nicht mit der Korrektur durch `custom-style`.

Das `custom-style`-Element kann außerdem dazu genutzt werden, globale Werte für CSS-Variablen zu vergeben, die dann in allen Komponenten zur Verfügung stehen, sofern die Komponenten die Variable berücksichtigen:

```
<style is="custom-style">
   :root {
     --accent-color: #afe;
     --font-family: Helvetica, sans-serif;
   }
 </style>
```

Diese Werte stehen dann für alle Komponenten in der Anwendung zur Verfügung.

4.7.6 Styles wiederverwenden mit Style Modules

Wenn mehrere Elemente die selben CSS-Definitionen nutzen sollen, beispielsweise um ein Firmenfarbschema über mehrere Anwendungen hinweg nutzen zu können, ohne dies als Kopie in jeder Anwendung halten zu müssen, wäre es wünschenswert, diese CSS-Definitionen an einer zentralen Stelle zu definieren und wiederverwenden zu können.

Da Polymer aber die Verwendung von CSS aus `<link rel="stylesheet">` nicht automatisch auf das Local DOM einschränken kann, müsste auf verschiedene Hacks zurückgegriffen werden, um dies zu realisieren. Man könnte etwa das Stylesheet per AJAX als Text laden und dann mittels `document.createElement` ein neues `<style>`-Element einfügen und durch den Text aus der Datei zu ersetzen.

Glücklicherweise bietet Polymer aber einen Weg, dies ohne solche Tricks zu implementieren. Dazu wird eine neue Komponente angelegt, die nur über ein `<dom-module>` mit einem `<template>` und darin einem `<style>`-Element verfügt. Die ID des `<dom-module>` kann dann in anderen Komponenten, welche diese Komponente mit `<link rel="import">` importieren, genutzt werden, um die CSS-Definitionen zu laden.

Eine solche Komponente wird als **Style Module** bezeichnet, da sie keinerlei eigenes Verhalten oder eigenen Inhalt definiert, sondern nur die Darstellung anderer Komponenten beeinflussen kann.

Ein »dark«-Theme könnte so aussehen:

```
<dom-module id="company-theme">
   <template>
     <style>
     div, p, input, textarea {
       background-color: #000;
       color: #fff;
     }
     a: {
       color: #0f0;
     }
     a:visited {
       color: #0d0;
     }
     </style>
   </template>
 </dom-module>
```

Um dieses Style Module zu nutzen, kann eine andere Komponente es importieren und verwenden:

```
<link rel="import" href="../styles/company-theme.html">
 <dom-module id="my-component">
   <template>
     <style include="company-theme"></style> ❶
     <style>
       div {
         color: #f00; ❷
       }
     </style>
   </template>
   <!-- ... -->
 </dom-module>
```

❶ Für `include` muss die ID des `<dom-module>` eines Style Module angegeben werden.

❷ Es können mehrere Style Modules oder `<style>`-Elemente benutzt werden.

Ein `style`-Element, welches ein Style Module einbindet, kann gleichzeitig auch zusätzliche Regeln definieren (die Regeln aus dem Modul überschreiben können):

```
<style include="company-theme">
   div, p {
     border: 1px solid white;
   }
   a {
     color: #f00;
```

```
    }
  </style>
```

Auch `custom-style` und Style Module lassen sich so kombinieren:

```
<style include="company-theme" is="custom-style">
    div, p {
      border: 1px solid white;
    }
    a {
      color: #f00;
    }
  </style>
```

Die so geladenen CSS-Definitionen aus dem Style Module gelten für das Dokument, beeinflussen aber die Inhalte der Shady- bzw. Shadow-DOM-Kontexte innerhalb von Komponenten (außer es wird der Stern-Selektor eingesetzt).

4.7.7 Zusätzliche Bibliotheken, Polymer-Elemente und CSS

Wenn innerhalb des Local DOM eines Polymer-Elements Inhalte dynamisch hinzugefügt werden, so wird vom Shady DOM nicht automatisch die notwendige Anpassung durchgeführt, um die Isolation auch für die neuen Inhalte anzuwenden. Dieser **Scoping** genannte Vorgang muss daher unter Shady DOM für dynamisch erzeugte DOM-Elemente innerhalb von Polymer-Elementen manuell angestoßen werden.

Dazu gibt es die Polymer-Methode `scopeSubtree`, die einem Element innerhalb eines Local DOM vorschreibt, entweder einmalig das Scoping durchzuführen oder das Element auf neue Inhalte zu überwachen und dann, wenn notwendig, das Scoping durchzuführen, wenn neuer Inhalt eingefügt worden ist:

```
<dom-module id="news-feed">
    <template>
      <style>
        h2 {
          color: red; ❶
        }
      </style>
      <div id="news"></div> ❷
    </template>
    <script>
      Polymer({
        is: 'news-feed',
        ready: function() {
          setTimeout(function makeNews() {
            var headline = document.createElement('h2') ❸
            headline.textContent = 'Nichts passiert. Auch schön.'
            document.getElementById('news').appendChild(headline) ❹
            setTimeout(makeNews, 2000)
```

```
        ], 2000)
      }
    })
  </script>
</dom-magic>
```

❶ Die Überschriften vom Typ `h2` sind rot.

❷ Hier werden Inhalte dynamisch und mit nativen DOM-Methoden eingefügt (also nicht mittels Polymer oder dessen Bindings).

❸ Mit nativen DOM-Methoden wird ein neues `<h2>`-Element erzeugt.

❹ Die Verwendung von nativen DOM-Methoden soll verdeutlichen, wie eine Manipulation mittels externer Bibliotheken geschehen könnte.

Die im Beispiel neu hinzugefügten `<h2>`-Elemente sind entgegen des ersten Anscheins nicht rot, sondern schwarz beziehungsweise in der Farbe, die ihnen von außerhalb des Elements vorgegeben wurde. Eine Untersuchung und der Vergleich der dynamisch erzeugten `<h2>`-Elemente zeigt, dass hier kein Scoping stattfand. Mit `scopeSubtree` kann dieses Scoping durchgeführt werden:

```
<dom-module id="news-feed">
   <template>
     <style>
       h2 {
         color: red;
       }
     </style>
     <div id="news"></div>
   </template>
   <script>
     Polymer({
       is: 'news-feed',
       ready: function() {
         this.scopeSubtree(this.$.news, true) ❶
         setTimeout(function makeNews() {
           var headline = document.createElement('h2')
           headline.textContent = 'Nichts passiert. Auch schön.'
           document.getElementById('news').appendChild(headline) ❷
           setTimeout(makeNews, 2000)
         ], 2000)
       }
     })
   </script>
 </dom-magic>
```

❶ Durch `scopeSubtree` wird Polymer auf Änderungen im Inhalt von `<div id="news"></div>` achten und der zweite Parameter gibt an, ob dies kontinuerlich oder nur einmalig geschieht.

❷ Nach wie vor werden native DOM-Elemente für die Manipulation verwendet.

Hinweis

Dieses Problem ergibt sich aus dem Scoping des Shady DOM. **Bei Verwendung des Shadow DOM ist die Verwendung von** `scopeSubtree` **nicht notwendig.**

Durch die Anwendung von `scopeSubtree` mit `true` als Wert für den zweiten Parameter werden DOM-`MutationObserver` initialisiert, die das Element aus dem ersten Parameter auf Änderungen überwachen und im Fall einer Änderung seines Inhaltes entsprechend reagieren und das Scoping durchführen. Bei einem Aufruf von `scopeSubtree` mit `false` als zweitem Parameter wird das Scoping für den aktuellen Inhalt durchgeführt, aber keine `MutationObserver` für zukünftige Änderungen angelegt.

Achtung

`scopeSubtree` sollte nur auf Elementen aufgerufen werden, die **keine** Polymer-Komponenten oder eigene Komponenten enthalten! Komponenten, die von `scopeSubtree` angepasst werden, verhalten sich unerwartet, da ihre eigenen Style-Eigenschaften von denen des Kontextes, in dem `scopeSubtree` aufgerufen wurde, überschrieben werden.