

**WBE: UI-BIBLIOTHEK**

**TEIL 2: IMPLEMENTIERUNG**

# ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele

# ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele

# RÜCKBLICK

- Ziel: eigene kleine Bibliothek entwickeln
- Komponentenbasiert und datengesteuert
- An Ideen von React.js und ähnlicher Systeme orientiert
- Motto: „Keep it simple!”
- Bezeichnung:

SuiWeb

Simple User Interface Toolkit for Web Exercises

# RÜCKBLICK

- Notation für den Aufbau der Komponenten
  - **JSX**: in React.js verwendet
  - **SJDON**: eigene Notation
- SuiWeb soll beide Varianten unterstützen

```
// jsx
const element = (<h1 title="foo">Hello</h1>)
```

```
// sjdon
const element = ["h1", {title: "foo"}, "Hello"]
```

# ANSTEHENDE AUFGABEN

- Interne Repräsentation der Komponenten
- Konvertierung von JSX und SJDON in diese Repräsentation
- Abbildung interne Repräsentation ins DOM
- Daten steuern Komponenten: Properties
- Hierarchie von Komponenten
- Komponenten mit Zustand

Anregungen und Code-Ausschnitte aus:

Rodrigo Pombo: Build your own React

<https://pomb.us/build-your-own-react/>

Zachary Lee: Build Your Own React.js in 400 Lines of Code

<https://webdeveloper.beehiiv.com/p/build-react-400-lines-code>

## Speaker notes

Rodrigo Pombo nennt seine React-ähnliche Implementierung *Didact*.

Wichtig: unsere Implementierung ist erheblich einfacher. Basierend auf *Didact* wurde eine Serie von Prototypen entwickelt. Diese waren definitiv nicht perfekt und vermutlich auch nicht fehlerfrei. Sie konnten aber bereits verwendet werden, um Demos für WBE damit umzusetzen.

Aufbauend auf den Prototypen wurde im Rahmen einer Projektarbeit eine neue Implementierung von SuiWeb in TypeScript vorgenommen:

<https://github.com/suiweb/suiweb>

Wenn Sie sich dafür interessieren, wie ein etwas umfangreicheres React-ähnliches Framework umgesetzt werden kann, können Sie die Sources und das Tutorial dieser Version von SuiWeb studieren. Auch *Build your own React* ist ein sehr empfehlenswertes Tutorial.

Da React sich schnell weiter entwickelt und die Ressourcen fehlen, SuiWeb 1.0 aktuell zu halten, wurde im Sommer 2024 eine neue Arbeitsversion auf der Basis des Tutorials von Zachary Lee „Build Your Own React.js in 400 Lines of Code“ erstellt. Die aktuellen WBE-Demos basieren auf dieser Version.

Eine Konsolidierung der verschiedenen Implementierungen zu einer einheitlichen Code-Basis ist derzeit eine offene Aufgabe... 😊

# AUSGANGSPUNKT

```
// jsx
/** @jsx createElement */
const element = (<h1 title="foo">Hello</h1>)

// jsx babel output (React < 17)
const element = createElement(
  "h1",
  { title: "foo" },
  "Hello"
)

// sjdon
const element = ["h1", {title: "foo"}, "Hello"]
```



## Speaker notes

Mit React 17 wurde eine neue Art, JSX zu transformieren, eingeführt. Hier ein Beispiel. Als Ausgangspunkt wird folgender JSX-Code verwendet:

```
function App() {  
  return <h1>Hello World</h1>;  
}
```

Das wird bis React 16 transformiert zu:

```
import React from 'react'  
  
function App() {  
  return React.createElement('h1', null, 'Hello world')  
}
```

Und ab React 17:

```
import {jsx as _jsx} from 'react/jsx-runtime'  
  
function App() {  
  return _jsx('h1', { children: 'Hello world' })  
}
```

Die Implementierung in SuiWeb bezieht sich auf die JSX-Transformation, wie sie bis React 16 verwendet wurde. Um diese zu verwenden, muss in Babel die Runtime von Automatic auf Classic gestellt werden.

Der Kommentar

```
/** @jsx createElement */
```

im JSX-Code weist Babel an, Elemente mit der Funktion createElement anzulegen. Die Einstellung für React.js ist:

```
/** @jsx React.createElement */
```

# INTERNE REPRÄSENTATION

// jsx babel output

```
const element = createElement(  
  "h1",  
  { title: "foo" },  
  "Hello"  
)
```

// internal representation

```
const element = {  
  type: "h1",  
  props: {  
    title: "foo",  
    children: ["Hello"],  
  },  
}
```

## Speaker notes

`props.children` ist normalerweise ein Array von Elementen, welche wieder auf die gleiche Art aufgebaut sind. Auf diese Weise entsteht eine Baumstruktur. Das Beispiel ist noch nicht ganz fertig. Im Array von `props.children` tauchen dann keine Strings mehr auf, sondern Repräsentationen der Kindelemente, also selbst wieder Objekte wie das vorliegende für `element`.

# INTERNE REPRÄSENTATION

```
{  
  type: "h1",  
  props: {  
    title: "foo",  
    children: ["Hello"],    /* noch anzupassen */  
  },  
}
```

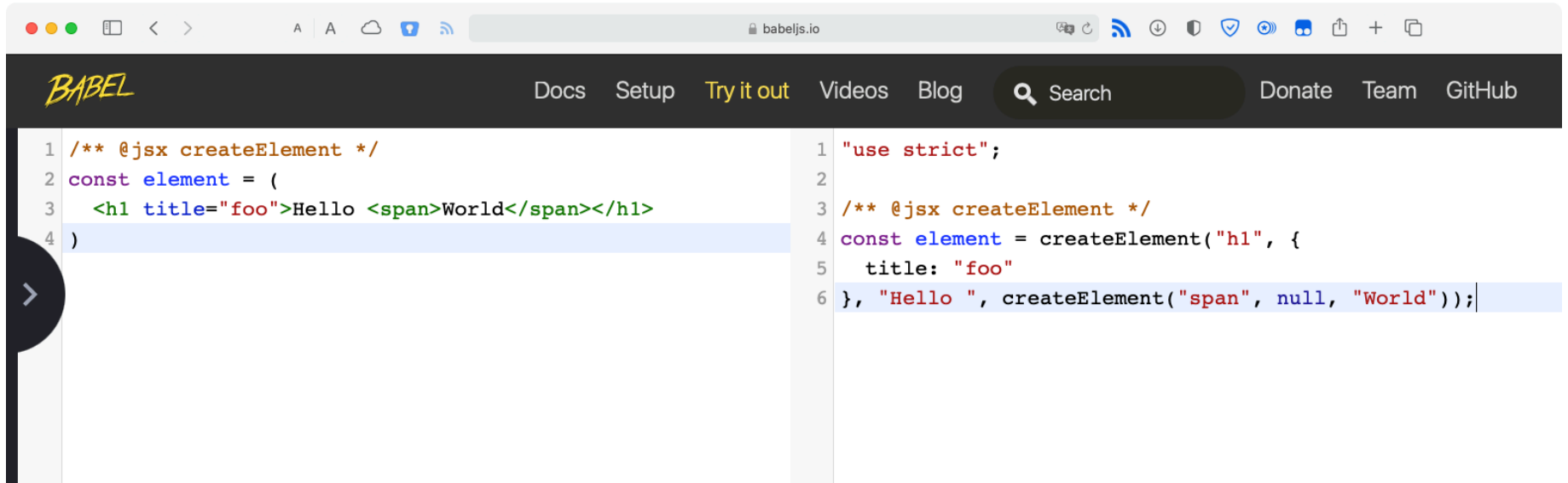
- Element: Objekt mit zwei Attributen, `type` und `props`
- `type`: Name des Elements ("body", "h1", ...)
- `props`: Attribute des Elements
- `props.children`: Kindelemente (Array)

# TEXT-ELEMENT

```
{  
  type: "TEXT_ELEMENT",  
  props: {  
    nodeValue: "Hello",  
    children: [],  
  },  
}
```

- Aufbau analog zu anderen Elementen
- Spezieller Typ: `"TEXT_ELEMENT"`

# VERSCHACHTELTE ELEMENTE



The screenshot shows the Babel website's 'Try it out' interface. The left pane contains the input JSX code, and the right pane shows the transformed code.

```
1 /** @jsx createElement */
2 const element = (
3   <h1 title="foo">Hello <span>World</span></h1>
4 )
```

```
1 "use strict";
2
3 /** @jsx createElement */
4 const element = createElement("h1", {
5   title: "foo"
6 }, "Hello ", createElement("span", null, "World"));
```

- Mehrere Kindelemente:  
ab drittem Argument von `createElement`
- Verschachtelte Elemente:  
rekursive Aufrufe von `createElement`

# KONVERTIERUNG VON JSX

```
function createElement (type, props,
                        ...children) {
  return {
    type,
    props: {
      ...props,
      children: children.map(child =>
        typeof child === "object"
          ? child
          : createTextElement(child)
      ),
    },
  },
}
```

```
function createTextElement (text) {
  return {
    type: "TEXT_ELEMENT",
    props: {
      nodeValue: text,
      children: [],
    },
  }
}
```



# CREATEELEMENT: BEISPIEL

```
// <div>Hello<br></div>
createElement("div", null, "Hello", createElement("br", null))

// returns
{
  type: 'div',
  props: {
    children: [
      {
        type: 'TEXT_ELEMENT',
        props: { nodeValue: 'Hello', children: [] }
      },
      { type: 'br', props: { children: [] } }
    ]
  }
}
```

# KONVERTIERUNG VON SJDON

```
1 function parseSJDON ([type, ...rest]) {
2   const isObj = (obj) => typeof(obj)=== 'object' && !Array.isArray(obj)
3   const children = rest.filter(item => !isObj(item))
4
5   return createElement(type,
6     Object.assign({}, ...rest.filter(isObj)),
7     ...children.map(ch => Array.isArray(ch) ? parseSJDON(ch) : ch)
8   )
9 }
```

- Abbildung auf `createElement`-Funktion
- Attribute in einem Objekt zusammengeführt
- Kindelemente bei Bedarf (Array) ebenfalls geparst

Tatsächlich geschieht hier in SuiWeb (ab Version 1.1) noch etwas mehr:

- Da `createElement` ein Objekt erstellt, welches normalerweise in `props.children` abgelegt wird, Objekte in SJDON aber für eine Sammlung von Attributwerten stehen, entsteht hier eine Mehrdeutigkeit. Zu diesem Zweck erhält die Ausgabe von `createElement` neben `type` und `props` noch ein weiteres Attribut `sjdon` mit dem Wert `"noprops"`.
- Um mit CSS-Stilen in SJDON etwas flexibler umgehen zu können, wird das `style`-Attribut in einer Funktion `combineStyles` nach Bedarf angepasst.

Damit ergibt sich die folgende Funktion:

```
function parseSJDON ([type=Fragment, ...rest]) {
  const isObj = (obj) =>
    typeof(obj)=== 'object'
    && !Array.isArray(obj)
    && obj.sjdon !== "noprops"
  const props = Object.assign({}, ...rest.filter(isObj))
  const children = rest.filter(item => !isObj(item))

  if (props.style !== undefined) {
    props.style = combineStyles(props.style)
  }

  const repr = createElement(type, props,
    ...children.map(ch => Array.isArray(ch) ? parseSJDON(ch) : ch)
  )
  return repr
}
```

# ZWISCHENSTAND

- Einheitliche Repräsentation für Elemente unabhängig von der ursprünglichen Syntax (JSX or SJDON)
- Baumstruktur von Elementen
- Text-Elemente mit leerem Array `children`
- DOM-Fragment im Speicher repräsentiert (virtuelles DOM?)

## Zu tun:

- Abbildung der Baumstruktur ins DOM

# RENDER TO DOM

```
1 function render (element, container) {
2   /* create DOM node */
3   const dom =
4     element.type == "TEXT_ELEMENT"
5       ? document.createTextNode("")
6       : document.createElement(element.type)
7
8   /* assign the element props */
9   const isProperty = key => key !== "children"
10  Object.keys(element.props)
11    .filter(isProperty)
12    .forEach(name => { dom[name] = element.props[name] })
13
14  /* render children */
15  element.props.children.forEach(child => render(child, dom))
16  /* add node to container */
17  container.appendChild(dom)
18 }
```

## Speaker notes

So funktioniert es im Prinzip und so könnte man es in einer sehr einfachen Bibliothek auch umsetzen. Tatsächlich wird der Render-Prozess in Arbeitspakete aufgeteilt, welche asynchron ausgeführt werden. Eine Beschreibung finden Sie hier:

<https://webdeveloper.beehiiv.com/p/build-react-400-lines-code>

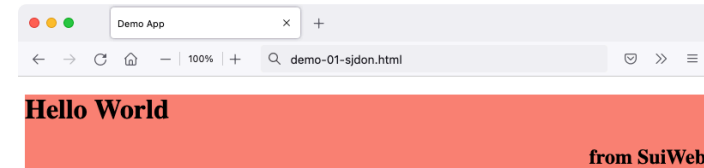
# HTML-ELEMENTE

- Komponenten können HTML-Elemente verwenden
- Tagnamen in Kleinbuchstaben
- Gross-/Kleinschreibung ist relevant
- Übliche Attribute für HTML-Elemente möglich
- Wenig Ausnahmen: `className` statt `class`



# BEISPIEL

```
1 import { render } from "../lib/suiweb-1.1.js"
2
3 const element =
4   [ "div", {style: "background:salmon"},
5     [ "h1", "Hello World"],
6     [ "h2", {style: "text-align:right"}, "from SuiWeb" ] ]
7
8 const container = document.getElementById("root")
9 render(element, container)
```



## Speaker notes

In folgendem Beispiel wird h1 noch mit einer Klasse versehen. Es wird `className` als Attribut verwendet. In SJDON können Attribute beliebig zwischen den Kindelementen vorkommen.

```
const element =  
  ["div", {style: "background: salmon"},  
    ["h1", "Hello World", {className: "title"}],  
    ["h2", {style: "text-align:right"}, "from SuiWeb"]  ];
```

# ZWISCHENSTAND

- Interne Struktur aufbauen
- Ins DOM rendern

SuiWeb: JSX, SJDON

demo-01-jsx.html →

demo-01-sjdon.html →

Didact: (Rodrigo Pombo)

<https://codesandbox.io/s/didact-2-k6rbj?file=/src/index.js>

# ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele

# FUNKTIONSKOMPONENTEN

```
1 const App = (props) =>
2   ["h1", "Hi ", props.name]
3
4 const element =
5   [App, {name: "foo"}]
```

- App ist eine Funktionskomponente
- Die zugehörige Repräsentation erzeugt keinen DOM-Knoten
- Ergebnis des Funktionsaufrufs wird als *child* eingehängt
- Konvention: eigene Komponenten mit grossen Anfangsbuchstaben

# PROBLEM

- Komponenten in JSX retournieren mittels `createElement` erzeugte interne Strukturen
- Unter SJDON liefern sie allerdings SJDON-Code, der nach Aufruf der Komponente noch geparkt werden muss
- Abhilfe: SJDON-Komponenten erhalten ein Attribut `sjdon`, welches die Konvertierung (`parseSJDON`) ergänzt
- Dieses Attribut lässt sich mit einer kleinen Hilfsfunktion anbringen

# SJDON-KONVERTIERUNG ERWEITERT

```
1 function useSJDON (...funcs) {  
2   for (let f of funcs) {  
3     const fres = (...args) => parseSJDON(f(...args))  
4     f.sjdon = fres  
5   }  
6 }
```

- Kann für mehrere Komponentenfunktionen aufgerufen werden, indem sie als Argumente übergeben werden
- Diese werden um das `sjdon`-Attribut ergänzt

## Speaker notes

Eine weitere kleine Anpassung im Vergleich zu der Version weiter oben in den Slides ist an der Funktion `createElement` nötig, speziell zur Verarbeitung von JSX: Wenn in einer Komponente `{props.children}` als Kindelement eingefügt wird, resultiert ein Array im Array. Das wird hier mit dem Aufruf von `flat()` behoben. In SJDON kann das Problem mit dem Spread-Operator `...props.children` umgangen werden.

```
function createElement(type, props, ...children) {
  if (children.length > 0 && Array.isArray(children[0])) children=children[0]
  return {
    type,
    props: {
      ...props,
      children: children.flat().map(child =>
        typeof child === "object"
          ? child
          : createTextElement(child)
      ),
    },
  },
}
```



# FUNKTIONSKOMPONENTEN

- Funktion wird mit `props`-Objekt aufgerufen
- Ergebnis ggf. als SJDON geparst

```
1  switch (typeof type) {  
2    case 'function': {  
3      let children  
4  
5      if (typeof(type.sjdon) === 'function') {  
6        children = type.sjdon(props)  
7      } else {  
8        children = type(props)  
9      }  
10  
11      reconcileChildren(...)  
12      break  
13    }  
14    ...  
15  }
```

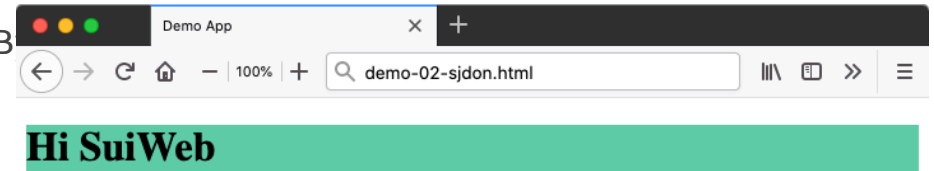
# BEISPIEL

```
const App = (props) =>
  ["h1", {style: "background: mediumaquamarine"}, "Hi ", props.name]

const element =
  [App, {name: "SuiWeb"}]

// notify SuiWeb that the App component returns SJDON
useSJDON(App)

const container = document.getElementById("container")
render(element, container)
```



demo-02-jsx.html →

demo-02-sjdon.html →

# WERTE STEUERN UI-AUFBAU

```
const App = () => {  
  const enabled = false  
  const text = 'A Button'  
  const placeholder = 'input value...'  
  const size = 50  
  
  return (  
    ["section",  
      ["button", {disabled: !enabled}, text],  
      ["input", {placeholder, size, autofocus: true}] ]  
  )  
}
```

demo-03-values →

# ARRAY ALS LISTE AUSGEBEN

```
const List = ({items}) =>  
  ["ul", ...items.map((item) => ["li", item]) ]
```

```
const element =  
  [List, {items: ["milk", "bread", "sugar"]} ]
```

```
useSJDON(List)
```

- Die `props` werden als Argument übergeben
- Hier interessiert nur das Attribut `items`

demo-04-liste →

# OBJEKT ALS TABELLE

```
const ObjTable = ({obj}) =>
  ["table", {style},
    ...Object.keys(obj).map((key) =>
      ["tr", ["td", key], ["td", obj[key]]])]

const style = {
  width: "8em",
  background: "lightblue",
}

const element =
  [ObjTable, {obj: {one: 1111, two: 2222, three: 3333}}]
```

demo-05-object →

# VERSCHACHTELN VON ELEMENTEN

```
/* JSX */  
<MySection>  
  <MyButton>My Button Text</MyButton>  
</MySection>
```

- Eigene Komponenten können verschachtelt werden
- `MyButton` ist mit seinem Inhalt in `props.children` von `MySection` enthalten

# VERSCHACHTELN VON ELEMENTEN

```
1  const MySection = ({children}) =>
2    ["section", ["h2", "My Section"], ...children]
3
4  const MyButton = ({children}) =>
5    ["button", ...children]
6
7  const element =
8    [MySection, [MyButton, "My Button Text"]]
9
10 useSJDON(MyButton, MySection)
```

demo-06-nested →

# TEILBÄUME WEITERGEBEN

```
1  const Main = ({header, name}) =>
2    ["div",
3      [...header, name],
4      ["p", "Welcome to SuiWeb"] ]
5
6  const App = ({header}) =>
7    [Main, {header, name: "web developers"}]
8
9  const element = [App, {header: ["h2", "Hello "]}]
10
11 useSJDON(App, Main)
```

demo-07-subtree →



# ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- **Darstellung von Komponenten**
- Defaults und weitere Beispiele

# DARSTELLUNG

- Komponenten müssen ggf. mehrere Styles mischen können
- Neben Default-Darstellung auch via `props` eingespeist
- Daher verschiedene Varianten vorgesehen:
  - CSS-Stil als String
  - Objekt mit Stilangaben
  - Array mit Stil-Objekten

## Speaker notes

Im Array werden die Stil-Angaben der Objekte zusammengemischt, bei gleichen Attributen haben spätere Angaben (höherer Index des Objekts im Array) Vorrang.

# DARSTELLUNG

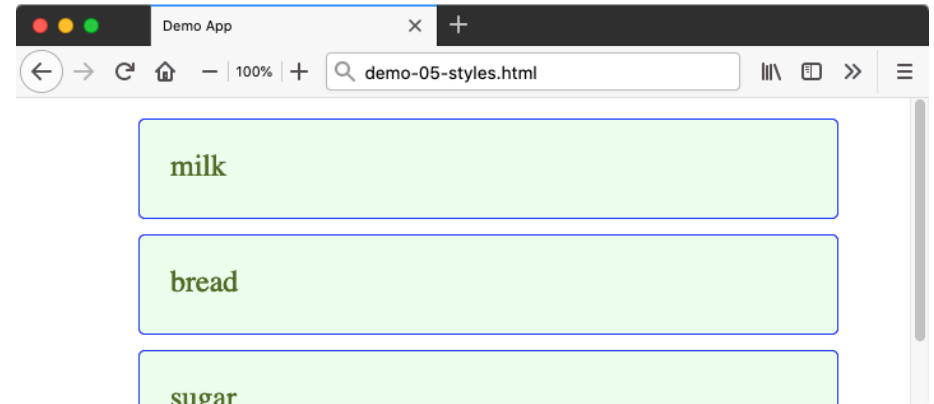
```
1 function combineStyles (styles) {
2   let styleObj = {}
3   if (typeof(styles)=="string") return styles
4   else if (Array.isArray(styles)) styleObj = Object.assign({}, ...styles)
5   else if (typeof(styles)=="object") styleObj = styles
6   else return ""
7
8   let style = ""
9   for (const prop in styleObj) {
10     style += prop + ":" + styleObj[prop] + ";"
11   }
12   return style.replace(/([a-z])([A-Z])/g, "$1-$2").toLowerCase()
13 }
```

## Speaker notes

Es könnte gut sein, dass das Kombinieren von verschiedenen Styleangaben noch etwas eleganter möglich wäre ... :)

# BEISPIEL

```
1  const StyledList = ({items}) => {
2    let style = [styles.listitem, {color: "#556B2F"}]
3    return (
4      ["ul", ...items.map((item) => ["li", {style}, item]) ]
5    )
6  }
7
8  const element =
9    [StyledList, {items: ["milk", "bread", "sugar"]}]]
10
11 const styles = {
12   listitem: {
13     padding: "1em",
14     margin: "0.5em 2em",
15     fontSize: "1.5em",
16     ... }
17 }
```



demo-08-styles →

## Speaker notes

Eine mögliche Erweiterung ist, die Styles einer Komponente zusammenzusetzen aus solchen, die für die Komponente voreingestellt sind und solchen, die zusätzlich von aussen in die Komponente eingespeist werden, wie hier über das zusätzliche Attribut `style`:

```
const StyledList = ({items,style={}}) => {  
  style = [styles.listitem, {color: "#556B2F"}, style]  
  return (  
    ["ul", ...items.map((item) => ["li", {style}, item]) ]  
  )  
}
```

Die lokale Variable `style` wird hier durch Destrukturierung des Parameter-Objekts angelegt, weshalb kein `let` vor `style` stehen darf. Nun kann der Default-Style nach Bedarf überschrieben werden:

```
const element =  
  [StyledList, {items: ["milk", "bread", "sugar"], style: {color:"red"}}]
```

# ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele



# DEFAULT PROPERTIES

```
1 const App = () => (  
2   [ "main",  
3     [MyButton, {disabled: true, text: 'Delete'}],  
4     [MyButton] ]  
5 )  
6  
7 const MyButton = ({disabled=false, text='Button'}) => (  
8   [ "button", disabled ? {disabled} : {}, text]  
9 )
```

demo-09-defaultprops →

## Speaker notes

In *MyButton* wird ein Objekt destrukturiert mit Defaultangabe – es handelt sich also nicht um einen Default-Parameter. So wird hier erreicht, dass Buttons normalerweise nicht *disabled* sind, ausser dies wird explizit auf *true* gesetzt. Ebenso tragen sie den Standard-Text *Button*, wenn das *text*-Attribut nicht überschrieben wird.

Ohne Destructurieren mit Defaults könnte das auch folgendermassen erreicht werden:

```
const MyButton = (props) => {
  const defaultProps = {
    disabled: false,
    text: 'Button',
  }
  const myprops = {...defaultProps, ...props}
  const { disabled, text } = myprops
  return (
    ["button", disabled ? {disabled} : {}, text]
  )
}
```

# DEFAULT PROPERTIES

- Übergebene Properties überschreiben Defaults
- Selbst zu implementieren (ist einfach, s. Beispiel)
- In React.js können Defaults an Funktion gehängt werden:  
(in SuiWeb nicht umgesetzt, wäre aber möglich)

```
const MyButton = (props) => { ... }
```

```
MyButton.defaultProps = {  
  text: 'My Button',  
  disabled: false,  
}
```

# WEITERES BEISPIEL

```
1  const MyButton = ({children, disabled=true}) =>
2    ["button", {style: "background: khaki", disabled}, ...children]
3
4  const Header = ({name, children}) =>
5    ["h2", "Hello ", name, ...children]
6
7  const App = (props) =>
8    ["div",
9      [Header, {name: props.name}, " and", ["br"], "web developers"],
10     [MyButton, "Start", {disabled:false}],
11     [MyButton, "Stop"] ]
12
13  useSJDON(App, Header, MyButton)
14  render([App, {name: "SuiWeb"}], container)
```

demo-10-children →

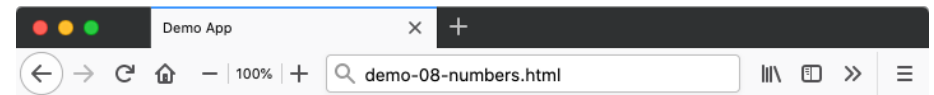
# ZAHLEN IN PROPS

```
const App = ({num1, num2}) =>  
  ["h1", num1, " * ", num2, " = ", num1*num2]
```

```
const element = [App, {num1: 3, num2: 9}]
```

- Beim Funktionsaufruf als Zahlen behandelt
- Beim Rendern in Textknoten abgelegt

demo-11-numbers →



**3 \* 9 = 27**

# AKTUELLER STAND

- Notationen, um Komponenten zu definieren: **JSX, SJDON**
- Funktionen zur Anzeige im Browser: **render**-Funktion
- Daten können Komponenten steuern: Argument **props**
- Ausserdem: Verarbeiten von Styles, Default-Properties
- Also: UI-Aufbau mit Komponenten
- Was noch fehlt: Mutation, Zustand  
→ nächste Woche 😊

# VERWEISE

- Build Your Own React.js in 400 Lines of Code  
<https://webdeveloper.beehiiv.com/p/build-react-400-lines-code>
- Rodrigo Pombo: Build your own React  
<https://pomb.us/build-your-own-react/>
- SuiWeb - An Educational Web Framework (Inspired by React)  
<https://github.com/suiweb/suiweb>





