Web-Entwicklung

Silko Kruse

Inhalt

- Einführung in JavaScript, Datentypen, Kontrollstrukturen, Schleifenanweisungen, Blöcke und Gültigkeitsbereiche, Typumwandlung, Felder, Assoziative Felder, Mehrdimensionale Felder, Funktionen, Objektorientierte Programmierung, Konstruktorfunktionen, Komposition, Klassensyntax, Vererbung, Canvas, Ereignisbehandlung, Farbverläufe, Dynamisches SVG, Chartist, D3, JSON, XML, AJAX, Bedienoberflächen, Bedienelemente platzieren, Geräteabhängige Ereignisse, Allgemeine Bedienelemente, Formulare, Zeichenketten, Formularüberprüfung, Templates, HTML erweitern, Komponententechnologien, Datenspeicherung, Verlaufsverwaltung, Audio, Literatur

Webauftritt: Früher

- HTML-Seiten
  - Überwiegend textuelle Information
  - Bilder
  - Formatierungsanweisungen
- Gestalterischer Aufwand vergleichsweise gering
- Inhalte statisch
- Keine Interaktivität

Webauftritt: Heute

- Webauftritt ist Benutzeroberfläche einer potentiell komplexen Webanwendung
- Hoher gestalterischer Aufwand
- Hoher Grad an Interaktivität
- Dynamische Seiteninhalte als Folge von Anwendungssteuerung
- Im Browser primär erreicht durch Einbettung von JavaScript in das HTML
- Ausführung durch JavaScript-Interpreter im Webbrowser

JavaScript: Geschichte

- Ursprünglich von der Firma Netscape unter dem Namen LiveScript entwickelt
  - Überprüfung von Formulardaten
  - Kommunikation mit Plug-Ins
- Kooperation zwischen Netscape und Sun Microsystems zwecks Schaffung einer Schnittstelle zwischen LiveScript und Java-Applets
- Umbenennung von LiveScript in JavaScript
- Einreichung der Sprache zwecks Standardisierung bei der ECMA (European Computer Manufacturers Association)
- 1997 Standardisierung als **ECMAScript** (ECMA-262)

JavaScript-Code in eine Webseite einbinden

- Intern
  - Mit Hilfe eines `<script>`-Tags
  - Als HTML-Link
  - Als Parameter von HTML-Tags
- Extern
  - Programmcode wird in einer Datei mit der Endung `.js` gespeichert
  - JavaScript-Code wird so dateiübergreifend verfügbar
  - JavaScript-Datei darf kein HTML enthalten
  - Einbindung der Datei mit `<script>`-Tag

Platzierung des `<script>`-Tags

- Möglich sowohl im Head- als auch im Body-Bereich
- Konvention:
  - Head-Bereich
    - Nicht anzuzeigender Teil der HTML-Seite
    - Definition von Variablen und Funktionen sowie von Code, der nicht aktiv in die Seitengestaltung eingreift
    - Stellt sicher, dass Variablen etc. vor der ersten Benutzung bekannt sind
  - Body-Bereich
    - Anzuzeigender Teil der HTML-Seite
    - Definition von Code, der aktiv, z. B. durch Ausgaben, in die Seitengestaltung eingreift

<script>- und <noscript>-Tag

```
<script [type="javascript"]>
  <!--
       JavaScript-Anweisungen
  // -->
</script>

<noscript>
      <p>JavaScript ist deaktiviert.</p>
</noscript>
```

- **type**-Attribut: Optional. Voreinstellungswert ist JavaScript. Angabe nur bei anderen Skriptsprachen erforderlich
- **noscript**: Das hier notierte Markup wird ausgeführt, falls JavaScript im Browser deaktiviert oder die angegebene Sprache nicht verfügbar ist

Ausgaben in die HTML-Seite

- JavaScript basiert auf der Arbeit mit Objekten
  - Entsprechend existiert eine umfangreiche Objektreferenz mit Objekten, die für die Programmierung zur Verfügung stehen
- Der Zugriff auf den im Browser-Fenster angezeigten Inhalt erfolgt mit Hilfe des **`document`**-Objekts und des Punktoperators
- Beispiel:
  - **`document.write("Hello World");`**
- Webseiteninhalt wird beim Einlesen der Seite vom Browser fortlaufend interpretiert
- Ausgabe von **`write`** erscheint deshalb an der Position der Seite, an der der JavaScript-Code steht

Beispiel

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript-Beispiel</title>
    <script>
      document.write("<p>Ausgabe 1</p>");
    </script>
    </head>
  <body>
    <script>
      document.write("<p>Ausgabe 2</p>");
    </script>
    <p>Ein Absatz.</p>
    <script>
      document.write("<p>Ausgabe 3</p>");
    </script>
  </body>
</html>
```

- Ausgabe in Browserfenster (Titel erscheint nur in der Titelzeile):

  Ausgabe 1
  Ausgabe 2
  Ein Absatz.
  Ausgabe 3

- Ausgaben im Head-Bereich erscheinen auch im Browserfenster
- HTML-Tags in **write**-Aufrufen werden interpretiert

Externe JavaScript-Datei einbinden

- `input.js` enthält:

```
document.write("<p>Ausgabe 1</p>");
```

- `index.html` enthält:

```
<html>
  <head>
    <script src="input.js"></script>
  </head>
  <body>
    <script>
      document.write("<p>Ausgabe 2</p>");
    </script>
    <p>Ein Absatz.</p>
    <script>
      document.write("<p>Ausgabe 3</p>");
    </script>
  </body>
</html>
```

- Ausgabe:

```
Ausgabe 1
Ausgabe 2
Ein Absatz.
Ausgabe 3
```

- Ausgaben in externer JavaScript-Datei erscheinen im Browserfenster an der Einbindungsposition

`<script>`-Tag: Externe Skripte

- Externe Scripte werden mit **`src`**-Attribut eingebunden
- **`<script src="script.js"></script>`**
  - Script wird geladen und ausgeführt. Erst dann wird das Laden und Interpretieren, d. h. *Parsen*, der Seite fortgesetzt
- **`<script src="script.js" async></script>`**
  - Script wird asynchron, d. h. parallel ausgeführt, während Webseite weiter geladen und geparst wird
- **`<script src="script.js" defer></script>`**
  - Script wird erst ausgeführt, wenn Webseite vollständig geladen und geparst wurde

HTML5: `async`

- Voreinstellung: Script wird geladen und ausgeführt. Erst dann wird das Laden und Interpretieren, d. h. *Parsen*, der Seite fortgesetzt
- Bei umfangreichen Scripts und/oder vielen, einzelnen Scripten führt dies zu einem spürbar verlangsamten Seitenaufbau
- Abhilfe vor HTML5: Scripte am Ende einer Webseite einbinden, um trotzdem eine schnelle Seitenanzeige zu gewährleisten
- Durch Einsatz von **`async`** ist dies nicht mehr erforderlich
- Nachteil **`async`**: Reihenfolge der Ausführung der Scripte nicht determiniert. Nur einsetzen, wenn Scripte voneinander unabhängig

HTML5: `defer`

- Scripte, die auf Seitenelemente zugreifen, können dies erst, wenn diese schon geparsed wurden
- Abhilfe vor HTML5: Scripte hinter den betreffenden Elementen oder am Ende einer Webseite einbinden
- Durch Einsatz von **`defer`** ist dies nicht mehr erforderlich

Testausgaben

- Ausgaben aus dem Programmcode müssen nicht zwangsläufig in die Webseite geschrieben werden
  - Können die Seitendarstellung beim Testen stören
- Alternativen:
  - Ausgabe durch Meldung
  - Schreiben in die Browser-Konsole
- Beispiel für Meldung:

```
<script>
  window.alert("Summe = " + (2 + 3));
</script>
```

  - Verfügbar über das Window-Objekt des Browsers

Browser-Konsole

- Üblicherweise verfügbar als Teil der browserspezifischen Entwicklertools
  - Chrome: Einschalten mit *Weitere Tools* > *Entwicklertools* oder *JavaScript-Konsole*
  - Firefox: *Extras* > *Web-Entwickler* > *Web-Konsole*
  - Browser allgemein: Ausgabe erscheint in einem eigenen Konsolen- bzw. Ausgabefenster
- Beispiel:

```
<script>
  console.log("Produkt = " + (4 * 5));
</script>
```

Einführung in JavaScript

Kommentare

- // bzw. /* */: An C, C++, Java angelehnt

```
//  Dies ist ein einzeiliger Kommentar

/*
    Dies ist ein mehrzeiliger Kommentar.
    Das Ende des Kommentars wird gesondert
    angezeigt:
*/
```

- In HTML-Seiten funktionieren zusätzlich HTML-Kommentare:

```
<!-- Dies ist ein HTML-Kommentar -->
```

- Kommentare sind zum Verstehen des Codes wichtig!

Variablenname

- Erlaubt: Die Buchstaben **`a`** bis **`z`** bzw. **`A`** bis **`Z`**, die Ziffern **`0`** bis **`9`** und die Sonderzeichen _ und **`$`**
  - Groß- und Kleinbuchstaben werden unterschieden
- Nicht erlaubt: Leerzeichen und reservierte Wörter
  - Z. B. **`if`**, **`while`**, **`for`**, etc.
- Variablenname darf nicht mit einer Ziffer anfangen
- Instruktive Namen wählen!

Variablendeklaration und -definition

- JavaScript-Variablen sind **typenlos**
- Deklaration erfolgt explizit mit Hilfe des Schlüsselworts **`var`** oder implizit mittels einfacher Zuweisung
  - Das Gleichheitszeichen ist der **Zuweisungsoperator**
- Beispiele:

```
var x = 2;
y = 10;
var z = x + y;
```

Gültigkeit einer Variablen

- Ohne **`var`** deklarierte Variablen sind automatisch global gültig
- Durch Verwenden von **`var`** kann die globale Gültigkeit einer Variablen eingeschränkt werden
  - Z. B. in einem Funktionsrumpf
- Der konsequente Gebrauch von durch **`var`** deklarierten Variablen wird empfohlen

Variablendeklaration: `strict mode` (1)

- Eingeführt in ECMAScript 5 (ES5)
- Im strikten Modus ist die Deklaration einer Variablen durch einfache Zuweisung verboten
  - Verhindert z. B. Probleme durch falsch geschriebene Variablen. Dies führt dann nicht zur Deklaration einer neuen Variable
  - Beispiel:

```
var aVar = 20;
avar = 30; // erzeugt neue Variable
```

- Wird aktiviert durch Hinzufügen von **`use strict`** an den Anfang eines JavaScript-Abschnitts oder einer JavaScript-Datei oder einer Funktion
  - Bei Funktionen gilt der Modus dann nur lokal in der Funktion

Variablendeklaration: `strict mode` (2)

- Beispiel:

```
"use strict"; // Alte Interpreter überlesen dies

x = 10;       // Scriptabbruch und Fehlermeldung
              // in Browser-Konsole

var aVar = 20;

avar = 30;    // Scriptabbruch und Fehlermeldung
              // in Browser-Konsole
```

Variablendeklaration: Voreinstellungswert

- Variablen können auch ohne Initialisierung deklariert werden
- `var tmp;`
- Solange einer Variablen noch kein Wert zugewiesen wurde, liefert sie den Voreinstellungswert **`undefined`**
- Der Wert **`undefined`** wird in logischen Ausdrücken als **`false`** interpretiert

Datentypen

- JavaScript ist eine locker typisierte Sprache
- Typen werden implizit benutzt
- Jede Variable kann jeglichen Datentyp enthalten
  - Zahl, Zeichenkette etc.
- "Typ" einer Variable darf sich während der Lebenszeit ändern

Typ	Wertebereich	Beschreibung
`Boolean`	`true, false`	Wahrheitswerte
`Number`		IEEE 754 64-Bit Gleitkommazahl
`Object`		Objektliterale
`String`		UTF-16-Zeichenkette
`Undefined`	`undefined`	Variable ohne Wert
`Null`	`null`	Leere Referenz

- Zu den Objekten zählen auch die Felder bzw. Arrays

Datentypwechsel

- "Typ" einer Variable darf sich während der Lebenszeit ändern

```
var myvar = 3; // Definition von myvar
// JavaScript tolerant bei doppelten
// Deklarationen
var myvar = "Drei"; // Datentypwechsel
var var1 = myvar;
```

`typeof`-Operator

- Liefert den Typ des folgenden Werts
- Beispiele:
  - **`document.write`** wird in der Folge durch **`write`** abgekürzt

```
write('<br>10: ', typeof 10); // number
write('<br>10.0: ', typeof 10.0); // number
write('<br>zehn: ', typeof "zehn"); // string
write('<br>true: ', typeof true);  // boolean
var myVar=3.57;
write("<br>myVar: " + typeof myVar);
// myVar: number
```

Typ: `String`

- Strings können beliebig lang sein
- Schreibweisen
  - Einfache und doppelte Anführungszeichen erlaubt (single/double quoted)
    - `'Das ist ein einfacher String'`
    - `"Das ist ein einfacher String"`
- Zeichenketten verbinden (konkatenieren)
- Verkettungsoperator: +
- Beispiel:

```
var str = "Hello " + "World"; // str enthält
                              // "Hello World"
```

Escape-Sequenzen

- Einfügen spezieller Zeichen in einen String

`\b`: Rückschritt

`\t`: Tabulator

`\n`: Zeilenvorschub

`\f`: Seitenvorschub

`\r`: Zeilenrücklauf

`\"`: Doppelter Anführungsstrich

`\'`: Einfacher Anführungsstrich

`\\`: Backslash

Konstanten

- Schlüsselwort **`const`**
  - Seit ECMAScript Version 6 (2015)
- Beispiel: **`const EINE_KONSTANTE = 10;`**

Geschachtelte Zuweisung

- Der Wert einer Zuweisung ist der zugewiesene Wert
- Dieser kann in einem übergeordneten Ausdruck verwendet werden
- Beispiel:
- `a = (b = 4) + 5;`
  - Nach Auswertung ist `a` gleich `9` und `b` gleich `4`

Arithmetische Operatoren

`a + b`	Addition:	Summe von a und b
`a - b`	Subtraktion:	Differenz von a und b
`a * b`	Multiplikation:	Produkt von a und b
`a / b`	Division:	Quotient von a und b
`a % b`	Modulus:	Rest von a geteilt durch b

Beispiele:

```
var var1 = 1 + 1;
var var2 = 100.5 - 2;
var var3 = var1 + var2;
var var4 = var3 / 27.2;
var var4 = var4 * 2;
```

Logische Operatoren

- a && b Und
- a || b Oder
- ! a Nicht

Variablen		Verknüpfungsergebnis		
`A`	`B`	`A && B`	`A || B`	`!A`
`false`	`false`	`false`	`false`	`true`
`false`	`true`	`false`	`true`	`true`
`true`	`false`	`false`	`true`	`false`
`true`	`true`	`true`	`true`	`false`

Vergleichsoperatoren

- a == b Wert gleich: **`true`**, wenn Wert von a gleich Wert von b
- a != b Wert ungleich: **`true`**, wenn Wert von a nicht gleich Wert von b
- a < b Kleiner: **`true`**, wenn a kleiner als b
- a > b Größer: **`true`**, wenn a größer als b
- a <= b Kleiner-gleich: **`true`**, wenn a kleiner oder gleich b
- a >= b Größer-gleich: **`true`**, wenn a größer oder gleich b

Kontrollstrukturen

Kontrollstrukturen

- Programmverzweigungen
  - `if, if-else`
  - `switch`
- Schleifenkonstrukte
  - `while`
  - `do … while`
  - `for`

Programmverzweigungen

if-else

- Erlaubt Verzweigungen im Programmablauf:

```
if ( Bedingung )
  // Anweisung, die ausgeführt wird, wenn
  // die Bedingung erfüllt, d. h. wahr, ist
else
  // Anweisung, die ausgeführt
  // wird, wenn die Bedingung nicht
  // erfüllt, d. h. falsch, ist
```

- Der **`else`**-Zweig ist optional

`if-else` mit Block

- Block: In Klammern eingeschlossene Reihe von Anweisungen

```
if ( Bedingung ) {
  // Anweisungen, die ausgeführt werden, wenn
  // die Bedingung erfüllt, d. h. wahr, ist
}
else {
  // Optionale Anweisungen, die ausgeführt
  // werden, wenn die Bedingung nicht
  // erfüllt, d. h. falsch, ist
}
```

if-else: Beispiele (1)

```
if (a > b)
  document.write("a ist größer als b <br>");

if (a > b) {
  document.write("a ist größer als b <br>");
  max = a;
}

if (a > b) {
  document.write("a ist größer als b <br>");
  max = a;
}
else {
  document.write("a ist kleiner-gleich b <br>");
  max = b;
}
```

`if-else`: Beispiele (2)

- Browser-Typ auslesen:
- `document.write(window.navigator.userAgent);`
- Abfrage des Browser-Typs:

```
if (window.navigator.userAgent.indexOf("Firefox") > -1){
  document.write("Sie benutzen den Firefox <br>");
}
```

- `indexOf()`: Zeichenkettenfunktion, die den Indexwert des ersten Zeichens der gesuchten Zeichenkette liefert
- Liefert -1, wenn gesuchte Zeichenkette nicht im String vorhanden ist

Traditionelle Browserweiche

```
var browser = window.navigator.userAgent;

if (browser.indexOf("Firefox") > -1) {
  document.write("<p>Firefox</p>");
}
else if (browser.indexOf("Chrome") > -1) {
  document.write("<p>Chrome</p>");
}
else {
  document.write("<p>Unbekannter Browser</p>");
}
```

Nachteile

- Browser-Erkennungscode muss laufend an die aktuellen Versionen der Browser angepasst werden
- Erfolgt dies nicht stringent, so können sich Annahmen über Browser-Eigenschaften als falsch erweisen
- Dann reagiert Script falsch auf die aktuellen Eigenschaften eines Browsers
  - Um dies abzumildern, lieferte Microsoft z. B. zurückliegend beim Internet Explorer einen "falschen" Erkennungsstring, damit Browserweichen nicht alte Eigenschaften des Internet Explorers voraussetzten, die unter neuen Versionen nicht mehr funktionierten
- Beispiel: Internet Explorer 11
  - `Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; .NET4.0E; .NET4.0C; .NET CLR 3.5.30729; .NET CLR 2.0.50727; .NET CLR 3.0.30729; InfoPath.3; rv:11.0) like Gecko`
- Heutzutage wird die programmatische "Feature-Erkennung" favorisiert

Beispiel: Kreditkartenfirma abfragen

- Schreibweise mit einer Reihe von **`if`**-Anweisungen
  - (Mit: **`document.write()`** abgekürzt zu **`write()`**)

```
if (card == "Mastercard") {
  write("Betrag wird von Mastercard abgebucht");
}
else if (card == "Visa") {
  write("Betrag wird von Visa-Karte abgebucht");
}
else if (card == "Eurocard") {
  write("Betrag wird von Eurocard abgebucht");
}
else {
  write("Kreditkartenfirma unbekannt");
}
```

Beispiel: Kreditkartenfirma abfragen

- Schreibweise mit **`switch`**-Anweisung

```
switch (card) {
  case "Mastercard":
    write("Betrag wird von Mastercard abgebucht");
    break;
  case "Visa":
    write("Betrag wird von Visa-Karte abgebucht");
    break;
  case "Eurocard":
    write("Betrag wird von Eurocard abgebucht");
    break;
  default:
    write("Kreditkartenfirma unbekannt");
}
```

Schleifenanweisungen

Prä- und Post-Inkrement und -Dekrement

- **Post-Inkrement: i++**
- **Post-Dekrement: i--**
- **Prä-Inkrement: ++i**
- **Prä-Dekrement: --i**
- Beispiel:

```
var i = 1;
write(i++); // Ausgabe: ?
write(++i); // Ausgabe: ?
write(i--); // Ausgabe: ?
write(--i); // Ausgabe: ?
```

Weitere Abkürzungen

```
var a = 3;

a += 5;  // setzt a auf den Wert 8
         // entspricht: a = a + 5;
```

- Auch: `-=`, `*=`, `/=`, `%=` etc.

`while`-Schleife

- Grundform einer **`while`**-Anweisung:

```
while (ausdr)
   Anweisung;
```

- Anweisung kann auch ein Block sein

```
while (ausdr) {
   // Anweisungen;
}
```

while-Schleife: Beispiele

```
// Beispiel 1
var i = 1;
while (i <= 10)
  write(i++); // erst wird i ausgegeben, dann der
              // Wert erhöht (Post-Inkrement)
// Beispiel 2:
var i = 1;
var text = "<br>";
while (i <= 10) {
  text += "The number is " + i + "<br>";
  i++;
}
write(text);
```

`do-while`-Schleife

- Wie **`while`**-Schleife
- Wahrheitswert des Ausdrucks wird indes erst am Ende jeder Iteration geprüft
- Schleife wird in jedem Fall einmal durchlaufen
- Grundformen einer **`do-while`**-Anweisung:

1.
```
do Anweisung while (ausdr);
```

2.
```
do {
    // Anweisungen
} while (ausdr);
```

`for`-Schleife

- Schleifenanweisung für Zählschleifen

1. `for (Initial.; Bedingung; Aktualisier.) Anweisung`
2. `for (Initial.; Bedingung; Aktualisier.) {`

```
    // Anweisungen
  }
```

`Initialisierung:` Zählvariable initialisieren

`Bedingung:` Prüfen, ob Zählvariable im zulässigen Bereich

`Aktualisierung:` Wert der Zählvariable erhöhen oder verringern

`for`-Schleife: Beispiel

- Beispiel:

```
var text = "<br>";
for (var i=0; i<5; i++) {
  text += "The number is " + i + "<br>";
}
write(text);
```

- Deklaration der Zählvariable mittels **`var`** kann im Schleifenkopf erfolgen

Blöcke und Gültigkeitsbereiche

`for`-Schleife: Gültigkeit der Zählvariable

- Beispiel:

```
var text = "<br>";
for (var i=0; i<5; i++) {
  text += "The number is " + i + "<br>";
}
write(text + "Final i: " + i);
```

Ausgabe:

```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
Final i: 5
```

- Gültigkeit von Variablen ist bei JavaScript nicht automatisch blockgebunden (kein *block-level scope*)
  - Variable `i` daher auch außerhalb der `for`-Schleife gültig
  - Deklaration wird an den Anfang des Blocks gezogen

Heben (Hoisting)

- Eine Variable kann bereits vor ihrer Deklaration im Code verwendet werden
  - Der Interpreter bewegt bzw. *hebt* alle Deklarationen an den Beginn des Blocks
- Beispiel:

```
"use strict";
num = 5;
write(num + "<br>"); // 5
var num = 10;
write(num + "<br>"); // 10
```

- Der Variable `num` kann bereits vor ihrer Deklaration ein Wert zugewiesen werden

Ausgabe:

```
5
10
```

Hoisting und Zuweisungen

- Der Interpreter praktiziert Hoisting nur bei Deklarationen, aber nicht bei Zuweisungen

- Beispiel:

```
write(num + "<br>"); // liefert: undefined

var num = 10;

write(num + "<br>"); // liefert: 10
```

- Die erste Ausgabe liefert **`undefined`**, da die Variable zwar bereits gültig, aber noch keine Zuweisung erfolgt ist

`let`-Befehl

- Block-level scope ab ECMAScript 6 über den **`let`**-Befehl möglich
- Beispiel:

```
var text = "<br>";
for (let i=0; i<5; i++) {
  text += "The number is " + i + "<br>"; // ok
}
write(text + "Final i: " + i);
```

- Programmabbruch in der letzten Zeile: **`i`** existiert nicht
- Browserkompatibilität des Codes ggf. hinsichtlich **`let`** prüfen
  - Nicht kompatible Browser brechen den Code schon in der Zeile mit dem **`let`** ab

`let`-Befehl: Weiteres Beispiel (1)

```
var text = "<br>";
for (let i=0; i<5; i++) {
  text += "The number is " + i + "<br>"; // ok
}
write(text + "i: " + typeof i);
```

- Liefert: `i: undefined`

`let`-Befehl: Weiteres Beispiel (2)

```
for (var i=0; i<5; i++) {
  text += "The number is " + i + "<br>";
}
write(text + "Final i: " + i);

if (i==5) {
  let x = 10;
  i += x; // i = 15
  // let i=10; // Abbruch: i existiert schon
  let i1 = 0;
  let i2 = 5;
  if (i1 >= 0) {
    let i2; // i2 oben verschattet
    write("<br>i2: " + i2); // i2 = undefined
    i2 = i1+10; // i1 ist hier auch gueltig
    write("<br>i2: " + i2);
  }
}
write("<br>x= " + x); // Programmabbruch
```

Ausgabe:

```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
Final i: 5
i2: undefined
i2: 10
```

Typumwandlung

Typumwandlung

- Typumwandlung tritt auf, wenn ein Wert in den Wert eines anderen Datentyps umgewandelt wird
- Arten:
  - Implizite Typumwandlungen (**Coercion**)
  - Explizite Typumwandlungen ((Type) **Casting**)

Implizite Typumwandlungen

- Wird automatisch vom JavaScript-Interpreter zwischen kompatiblen Typen durchgeführt
- Beispiele:
  - `write("x = " + x);`
    - x wird automatisch in eine Zeichenkette umgewandelt
  - `var i = "7" * "4";`
    - Variable `i` besitzt danach den Wert 28, da Zeichenketten automatisch in Zahlen konvertiert werden
  - Aber: `var j = "7" + "4";`
    - `j` besitzt danach den Wert `"74"`, da zwei Zeichenketten konkateniert werden

Umwandlung in den Datentyp `Boolean`

- Folgende Werte werden als *falsy* ("falsch-artig") interpretiert und liefern damit **`false`**:
  - der Wert **`false`** des Typs **`Boolean`**
  - die Integerzahlen 0 und -0
  - die Fließkommazahl 0.0
  - die leere Zeichenkette
  - **`null`**, **`undefined`** und **`NaN`**
- Jeder andere Wert wird als *truthy* ("wahr-artig") interpretiert und liefert damit **`true`**

Logische Operatoren und Typumwandlung

- a && b Und
- a || b Oder
- ! a Nicht

Variablen		Verknüpfungsergebnis		
`A`	`B`	`A && B`	`A || B`	`!A`
`falsy`	`falsy`	`A`	`B`	`true`
`falsy`	`truthy`	`A`	`B`	`true`
`truthy`	`falsy`	`B`	`A`	`false`
`truthy`	`truthy`	`B`	`A`	`false`

Vergleichsoperatoren und Typumwandlung

- a `==` b Wert gleich
  - **`true`**, wenn Wert von a gleich Wert von b
- a `!=` b Wert ungleich
  - **`true`**, wenn Wert von a nicht gleich Wert von b
- a `===` b Strikt gleich: Identischer Typ und Wert
  - **`true`**, wenn Wert von a gleich Wert von b ist und beide vom gleichen Typ sind
- a `!==` b Strikt ungleich: Nicht identischer Typ und/oder Wert
  - **`true`**, wenn Wert von a nicht gleich Wert von b ist, oder wenn beide nicht vom gleichen Typ sind

Explizite Umwandlung in Zahlenwerte

- „Casting“
- Jeder Datentyp, dessen Literal als Zahl interpretierbar ist, kann in eine Zahl umgewandelt werden
- Ist Literal nicht als Zahlenwert interpretierbar, ist der resultierende Zahlenwert **`NaN`**
- Beispiele:
  - **`write(Number(" 27 "));`**
    - Liefert **`27`**, Leerzeichen werden ignoriert
  - **`write(Number("3.7")); // liefert 3.7`**
  - **`write(Number("ten")); // liefert NaN`**
  - **`write(Number("17 63")); // liefert NaN`**

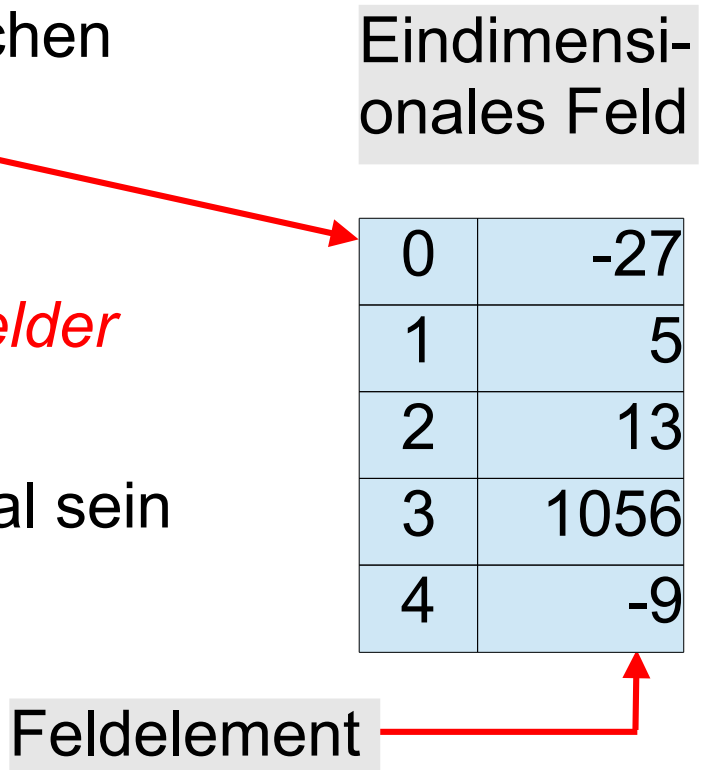
Explizite Umwandlung von Wahrheitswerten

- Für Boolsche Literale und **`Number`** gilt: **`true`** → **`1`**, **`false`** → **`0`**
- Beispiele:
  - **`write(Number(true));`**
    - Liefert **`1`**
  - **`write(Number(false));`**
    - Liefert **`0`**

Felder

Felder in JavaScript

- (engl.: *array*)
- Datentyp, welcher die Speicherung einer fortlaufenden Liste von Werten erlaubt
- Werte werden über *Schlüssel* angesprochen
- Schlüssel sind *Indizes*,
  d. h. nicht-negative ganze Zahlen
- Derartige Felder werden als *indizierte Felder* bezeichnet
- Felder können ein- oder mehrdimensional sein



Felder benutzen

- Feld als Feldliteral deklarieren und initialisieren:

```
var a = [23, "String", 14.7, -3];
```

  - Für Felder gilt die gleiche lockere Typisierung wie für Variablen

- Feldelemente mit Schleife verarbeiten:

```
for (var i = 0; i < a.length; i++)
  write("a[" + i + "]: " + a[i] + "<br>");
```

  - Die Länge eines Feldes, d. h. die Anzahl seiner Feldelemente, kann aus der Eigenschaft **`length`** mit Hilfe des Punktoperators abgefragt werden

Felder dynamisch erweitern

- Zuweisungen an nicht existierende Feldelemente führen zu einer nachträglichen Erweiterung des Felds

- Beispiel:

```
var a = []; // leeres Feld
a[0] = -27;
a[1] =  "String";
a[2] =  13.0;
a[5] =  210;
a[8] = -7;

for (var i = 0; i < a.length; i++)
  write("<br> a[" + i + "]: " + a[i]);
```

Ausgabe:

```
a[0]: -27
a[1]: String
a[2]: 13
a[3]: undefined
a[4]: undefined
a[5]: 210
a[6]: undefined
a[7]: undefined
a[8]: -7
```

Feldelemente dynamisch anhängen

- **`push()`**: Ein oder mehr Elemente anhängen
- Syntax:
  - **`array.push(item1)`**
  - **`array.push(item1, item2, ..., itemX)`**
- Beispiel:

```
var a = [];
var length = a.push(-27,"String",13.0,210,-7);
```

- Fügt Elemente fortlaufend hinter dem letzten Feldelement ein
- Liefert die aktuelle Länge des Felds als Rückgabewert

Feldelemente dynamisch entnehmen: `shift()`

- Syntax:
  - **`array.shift()`**: Liefert erstes Feldelement
  - Ist das Feld leer, so liefert **`shift()`** den Wert **`undefined`** und damit **`false`**
- Beispiel:

```
var a = [-27, "String", 13.0, 210, -7];
var erstesElement = a.shift();
write("Erstes Element: " + erstesElement +
      ", Length: " + a.length);
```

- Ausgabe: **`Erstes Element: -27, Length: 4`**
- mit: **`a`** → **`["String", 13.0, 210, -7]`**

```
          a[0]     a[1] a[2] a[3]
```

Feldelemente dynamisch entnehmen: `pop()`

- Syntax:
  - **`array.pop()`**: Liefert letztes Feldelement
  - Ist das Feld leer, so liefert **`pop()`** den Wert **`undefined`** und damit **`false`**
- Beispiel:

```
var a = [-27, "String", 13.0, 210, -7];
var letztesElement = a.pop();
write("Letztes Element: " + letztesElement +
      ", Length: " + a.length);
```

- **`Ausgabe: Letztes Element: -7, Length: 4`**
- mit: **`a → [-27, "String", 13.0, 210]`**

Dynamische Feldelemente und `for`-Schleife

- **`shift()`** und **`pop()`** bewirken einen Seiteneffekt auf **`length`**
- Folgendes geht nicht:

```
var a = [-27, "String", 13.0, 210, -7];
for (var i=0; i<a.length; i++)
  write("a[" + i + "]: " + a.shift() + ", ");
```

- Ausgabe: **`a[0]: -27, a[1]: String, a[2]: 13,`**
- Anstatt dessen folgende Lösung mit Hilfsvariable nehmen:

```
var length = a.length;
for (var i=0; i<length; i++)
  write("a[" + i + "]: " + a.shift() + ", ");
```

Beispiel: Maximumsuche mit Feld

```
var array = [7, 3, 99, 20];
var max = array[0];

for (var i = 1; i < array.length; i++)
  if (array[i] > max) // Feldelement mit Max.
                      // vergleichen
    max = array[i];

write("<br> Maximum: " + max);
```

Feld destrukturieren (1)

- Seit ES6 (2015) können Feldelemente und -abschnitte mittels *Destrukturierung (engl.: destructuring)* einzelnen Konstanten für Lesezugriff zugewiesen werden
- Beispiel: `var arr = ["Markus", "Mustermann"];`
  - `const [vorname, nachname] = arr;`

    `// => vorname="Markus", nachname="Mustermann"`
- Ermöglicht bessere Codelesbarkeit:
  - `vorname` und `nachname` an Stelle von `arr[0]` und `arr[1]`
- Anstatt `const` ist auch `var` möglich
  - Schreiboperationen auf Variablen haben aber keinen Seiteneffekt auf die Feldelemente

Feld destrukturieren (2)

- Weitere Möglichkeiten:
- `var arr = [23, "String", 14.7, -3, 60, 45];`
  - `const [b1,...b] = arr;`
    - `b1=23, b=["String", 14.7, -3, 60, 45]`
    - `...` liefert den Rest
  - `const [c1, ,c2,...[c, d]] = arr;`
    - `c1=23, c2=14.7, c=-3, d=60`
    - Leerzeichen `(, ,)` lässt Element aus
    - `...[c, d]` liefert den Rest als Feld und von diesem Feld die beiden ersten Elemente in `c` und `d`

Assoziative Felder

Assoziative Felder

- Schlüssel sind *Zeichenketten* (engl.: *Slots*) →

Nr. 1	-27
Nr. 2	5
Nr. 3	13
Nr. 4	1056
Nr. 5	-9

```
var data = [];

data["Nr. 1"] = -27;
data["Nr. 2"] =  5;
data["Nr. 3"] =  13;
data["Nr. 4"] =  1056;
data["Nr. 5"] = -9;

write(data["Nr. 1"]);
```

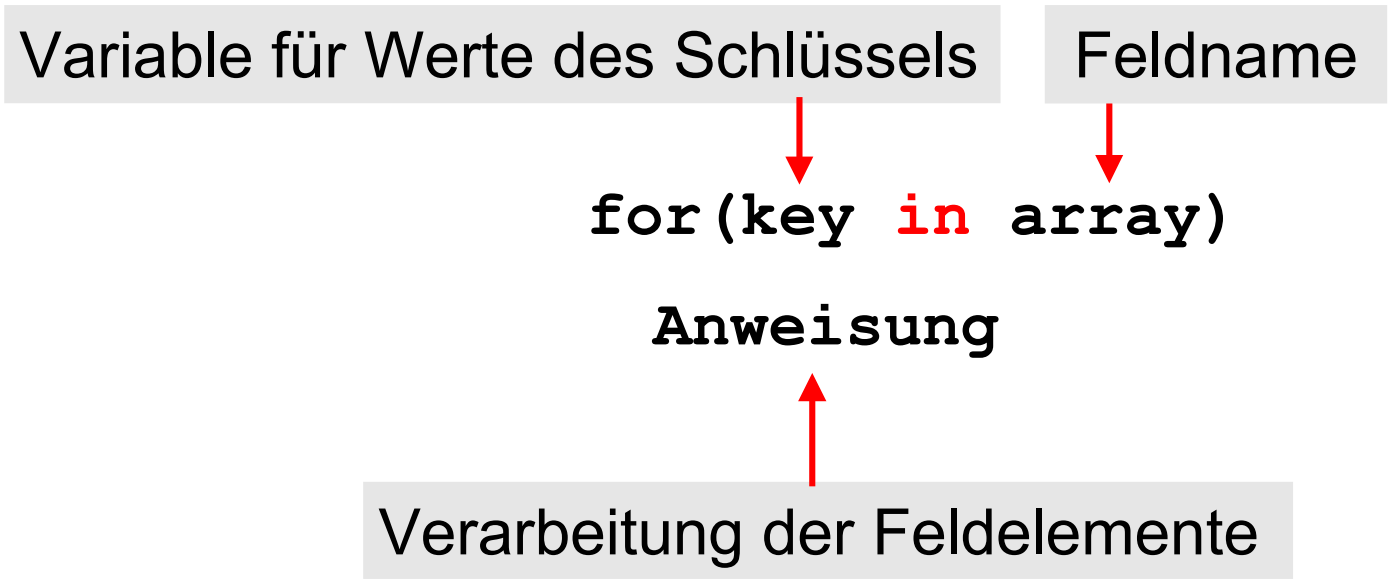
Ausgabe:

```
-27
```

- Wie lassen sich die Elemente eines assoziativen Felds mit einer Schleife verarbeiten?

`for-in`-Kontrollstruktur

- Ermöglicht es, die Elemente eines Felds ohne Zählvariable zu durchlaufen
- Syntax:



Beispiel

```
var data = [];
data["Nr. 1"] = -27;
data["Nr. 2"] =  5;
data["Nr. 3"] =  13;
data["Nr. 4"] =  1056;
data["Nr. 5"] = -9;
for (var key in data) {
  write(key + ": " + data[key]);
}
```

Ausgabe:

```
Nr. 1: -27
Nr. 2: 5
Nr. 3: 13
Nr. 4: 1056
Nr. 5: -9
```

`for-in` und indiziertes Feld

```
var a = [1, 2, 3, 4];
var sum = 0;
```

- Mittels `for-in` wird die folgende `for`-Schleife

```
for(var i=0; i<a.length; i++) {
  write("<br> a[" + i + "] = " + a[i]);
  sum += a[i];
}
```

- wie folgt geschrieben:

```
for(var i in a) {
  write("<br> a[" + i + "] = " + a[i]);
  sum += a[i];
}
```

- Kürzere Schreibweise möglich, sofern alle Feldelemente verarbeitet werden sollen

Dynamische Feldelemente und `for-in`-Schleife

- **`shift()`** und **`pop()`** bewirken einen Seiteneffekt auf **`length`**
- **`for-in`**-Schleife nicht einsetzbar. Folgendes geht nicht:

```
var a = [-27, "String", 13.0, 210, -7];
for (var i in a)
  write("a[" + i + "]: " + a.shift() + ", ");
```

- Ausgabe: **`a[0]: -27, a[1]: String, a[2]: 13,`**

Assoziatives Feld und Feldlänge

- ```
  var data = [];

  data["Nr. 1"] = -27;
  data["Nr. 2"] =  5;
  data["Nr. 3"] =  13;
  data["Nr. 4"] =  1056;
  data["Nr. 5"] = -9;

  write(data.length); // Ausgabe: 0!
  ```
- Assoziative Felder besitzen keine Länge!
- Sie verfügen ebenfalls nicht über **`push()`**, **`pop()`**, **`shift()`**
- Assoziative Felder sind in JavaScript tatsächlich Objekte
  ```

Mehrdimensionale Felder

Mehrdimensionale Felder

- Zusätzliche Dimensionen können bei der Initialisierung eines Felds durch Zuweisung von Feldern an die Feldelemente erzeugt werden
- Beispiel für ein zweidimensionales Feld:

```
var arr = [ [ 1, 9, 4 ],
            [ 0, 2 ],
            [ 0, 1, 2, 3, 4 ] ];

write("<br> arr[0][2] is " + arr[0][2]); // OK
write("<br> arr[1][1] is " + arr[1][1]); // OK
write("<br> arr[1][2] is " + arr[1][2]); // WRONG
write("<br> arr[2][4] is " + arr[2][4]); // OK
```

`length`-Wert und Zeilen unterschiedlicher Länge

- Der **`length`**-Eintrag eines zweidimensionalen Felds entspricht der Anzahl seiner Zeilen
- Jedes Zeilenfeld besitzt seinen eigenen **`length`**-Eintrag
- Bei mehrdimensionalen Feldern entsprechend

Zeilenlängen: Beispiel

```
var arr = [ [ 1, 9, 4 ],
            [ 0, 2],
            [ 0, 1, 2, 3, 4 ] ];
var str = "";
str += "No of rows: " + arr.length;
str += "<br>Length of row[0]: " + arr[0].length;
str += "<br>Length of row[1]: " + arr[1].length;
str += "<br>Length of row[2]: " + arr[2].length;
write(str);
```

Ausgabe:

```
No of rows: 3
Length of row[0]: 3
Length of row[1]: 2
Length of row[2]: 5
```

Ausgabe der Feldelemente mit `for`

```
var arr = [ [ 1, 9, 4 ],
            [ 0, 2],
            [ 0, 1, 2, 3, 4 ] ];
var str = "";
for (var row = 0; row < arr.length; row++) {
  str += "<br>Row " + row + ": ";
  for (var col = 0; col < arr[row].length; col++)
    str += arr[row][col] + " ";
}
write(str);
```

Ausgabe:

```
Row 0: 1 9 4
Row 1: 0 2
Row 2: 0 1 2 3 4
```

Ausgabe der Feldelemente mit `for-in`

```
var arr = [ [ 1, 9, 4 ],
            [ 0, 2],
            [ 0, 1, 2, 3, 4 ] ];
var str = "";
for (var row in arr) {
  str += "<br>Row " + row + ": ";
  for (var col in arr[row])
    str += arr[row][col] + " ";
}
write(str);
```

Ausgabe:

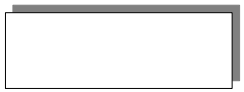
```
Row 0: 1 9 4
Row 1: 0 2
Row 2: 0 1 2 3 4
```

Mehrdimensionale Felder dynamisch erzeugen

- Wenn die Feldelementwerte nicht vorab bekannt sind, kann ein Feld mit unterschiedlicher Zeilenlänge auch **dynamisch** erzeugt werden
- Um dies zu erreichen, werden die Zeilen getrennt angelegt

Zeilen dynamisch erzeugen

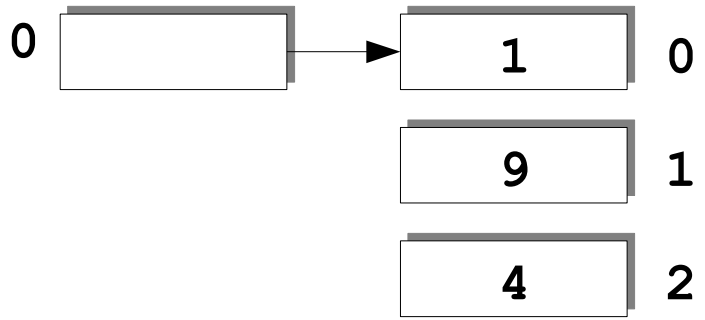
- `var myArray = [];`

  myArray 

  Leeres Feld für die Aufnahme weiterer eindimensionaler Felder als Feldelemente

- `myArray[0] = [];`

```
myArray[0][0] = 1;
myArray[0][1] = 9;
myArray[0][2] = 4;
```



- Oder:

```
myArray[0] = [];
myArray[0].push(1,9,4);
```

- Speicherzelle 0 verweist jetzt auf ein Feld mit 3 Elementen

Felder als Feldelemente

- Die nächste und alle weiteren Zeilen werden auf dieselbe Weise angelegt

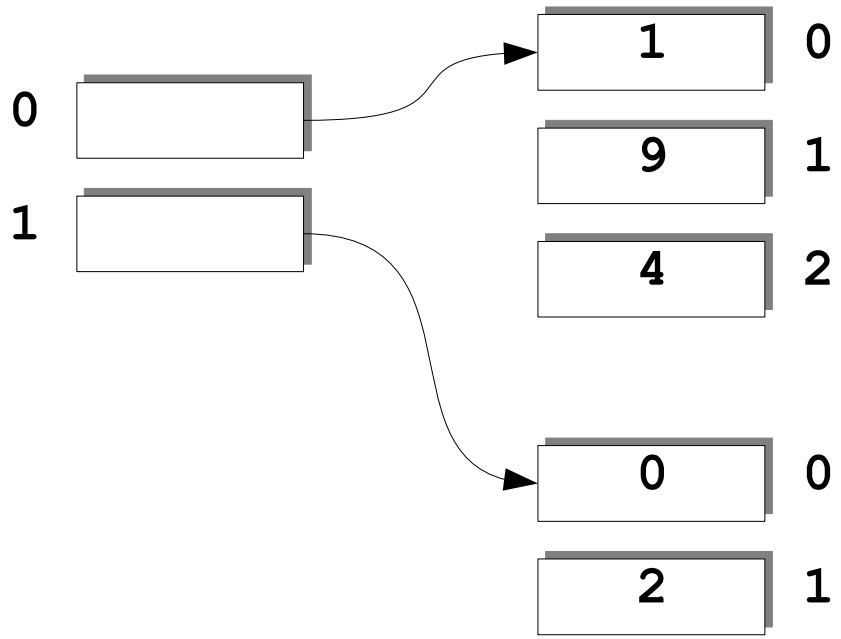
- `myArray[1] = [];`

```
myArray[1][0] = 0;

myArray[1][1] = 2;

// oder:
myArray[1].push(0,2);
```

- Speicherzelle 1 verweist jetzt auf ein Feld mit 2 Elementen
- usw.

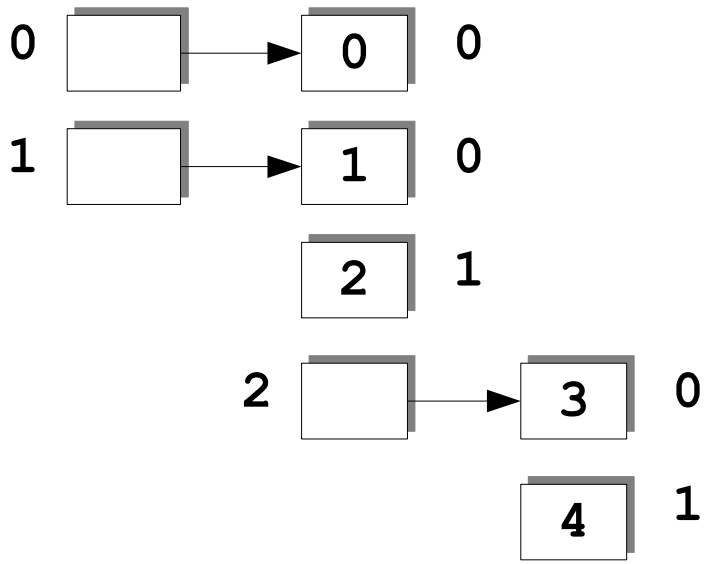


Ein zweidimensionales Feld ist ein Feld aus eindimensionalen Feldern

Mehrdimensionale Felder

- Wie zweidimensionale Felder mit entsprechend mehr Indizes
- Beispiel: Dreidimensionales Feld

```
var arr3d = [];
arr3d[0] = [];
arr3d[0][0] = 0;
arr3d[1] = [];
arr3d[1][0] = 1;
arr3d[1][1] = 2;
arr3d[1][2] = [];
arr3d[1][2][0] = 3;
arr3d[1][2][1] = 4;
```



Funktionen

Funktionen

- Funktionen bestehen aus einem **Funktionskopf** und einem **Funktionsrumpf**
- Bevor eine Funktion benutzt werden kann, muss sie deklariert werden
  - Die Funktionsdeklaration kann nach dem ersten Aufruf der Funktion im Quellcode stehen ("Function Hoisting")
- Syntax der Funktionsdeklaration:

```
function myFunction() {
  // Funktionsrumpf mit Anweisungen
}
```

  - Schlüsselwort `function`
  - Parameterlose Funktion: Leere runde Klammern

Parametrische Funktion

```
function myFunction(par1, par2,..., parn) {
  // Funktionsrumpf mit Anweisungen
}
```

- Daten werden an die Funktion mittels einer *Parameterliste* übergeben
  - Die Parameterliste ist eine durch Kommas getrennte Liste von Variablen und/oder Konstanten
  - Parameter werden auch als *Argumente* bezeichnet
  - Parameter sind Wertparameter (call by value): Zuweisungen an die Parameter im Funktionsrumpf haben keinen Seiteneffekt auf übergebene Variablen
  - Sofern vorhanden, wird ein Funktionswert mit `return` Befehl zurückgegeben

Funktion: Beispiele

```
function writeln(str) {
  document.write(str + "<br>");
}
function parSum(par1, par2, par3) {
  return par1+par2+par3;
}
writeln("Summe: " + parSum(1, 2, 3));
```

- Trotz **`return`** in **`parSum`** kein Rückgabetyp erforderlich
- Die Funktion **`writeln()`** wird auf den weiteren Folien als gegeben vorausgesetzt

Funktionsaufruf in Funktion

```
function parSum(par1, par2, par3) {
    return par1+par2+par3;
}

function starttag(tag) {
    return "<" + tag + ">";
}

function endtag(tag) {
    return "</" + tag + ">";
}

function paragraph(string) {
    return starttag("p") + string + endtag("p");
}

write(paragraph("Parametersumme: " + parSum(1, 2, 3)));
```

Vorgabewerte

- Parametern können seit ES6 Vorgabewerte zugewiesen werden
  - Die Ersetzung der Vorgabe- durch Parameterwerte beim Aufruf erfolgt dabei von links nach rechts

```
function parSum(par1 = 1, par2 = 2, par3 = 3) {
  return par1+par2+par3;
}
writeln("Sum1: " + parSum(2,4,6)); // = 12
writeln("Sum2: " + parSum());      // =  6
writeln("Sum3: " + parSum(10));    // = 15
writeln("Sum4: " + parSum(10,20)); // = 33
```

- Kompatibilität der Browser jeweils prüfen

Vorgabewerte vor ES6

- Funktionsparameter können keine Vorgabewerte erhalten
  - Zuweisung von Vorgabewerten mit Hilfe von zusätzlichem Ausdruck
- Beispiel:

A	B	A || B
falsy	falsy	B
falsy	truthy	B
truthy	falsy	A
truthy	truthy	A

```
function fvorgabewert(par) {
  if (!par) {
    // par ohne Wert => undefined => false
    par = 100;
  } // alternativ als logischer Ausdruck:
  par = par || 100;
  return par;
}
```

Variable Anzahl von Parameter

- Funktionen können eine variable Anzahl von Parametern besitzen
- Hierfür ist keine spezielle Syntax erforderlich
- Der Zugriff auf die Parameter ist möglich mit Hilfe des integrierten `arguments`-Felds
  - Die Parameterwerte stellen Werte dieses Felds dar
  - Die Anzahl der Parameter ist über die **`length`**-Eigenschaft des Felds abrufbar

Variable Parameteranzahl: Beispiel

```
function maximumsuche() {
    var max = arguments[0];

    for (var i = 1; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }

    return max;
}

writeln(maximumsuche(7, 3, 99, 20));
```

Funktionen als Werte

- Funktionsnamen können Variablen als Werte zugewiesen werden
- Die Funktion lässt sich dann ebenfalls über den Namen der Variable aufrufen
- Beispiel:

```
function square(x) { return x*x; }
var s = square;
square(4); // liefert 16
s(4); // liefert 16
```

- Der Funktionsname entspricht der Speicheradresse des Anweisungsblocks für den der Funktionsrumpf steht
- Durch die Zuweisung wird diese Adresse dupliziert

Anonyme Funktionen

- Funktionen, die keinen eigenen Namen besitzen
- Beispiel:

```
var square = function(x) { return x*x; };
var s = square;
square(4); // liefert 16
s(4); // liefert 16
```

- Die Funktionsdeklaration liefert eine Adresse, die einer oder mehreren Variablen zugewiesen werden kann
- *Funktionsausdruck* (engl.: *function expression*)
- Über die Variablennamen erfolgt dann der Funktionsaufruf
- Anonyme Funktionen werden traditionell in Objekten für die Definition von Methoden oder auch als Parameterwerte verwendet

Pfeilfunktionen (1)

- Seit ES6 existiert eine abkürzende Syntax für anonyme Funktionen (engl.: *Arrow Function*)
- Beispiel:

```
var square = (x) => { return x*x; };
var s = square;
square(4); // liefert 16
s(4); // liefert 16
```

- Allgemeine Form:
  - `(param1, param2, …, paramN) => { statements }`

Pfeilfunktionen (2)

- Entspricht der Funktionsrumpf einem Ausdruck, so lässt sich der Funktionsausdruck weiter vereinfachen
- An Stelle von beispielsweise

```
var square = (x) => { return x*x; };
```

  kann folgendes geschrieben werden:

```
var square = (x) => x*x;
```

- Allgemeine Form:
  - `(param1, param2, …, paramN) => expression`
    - an Stelle von: `=> { return expression; }`

Funktionen und Gültigkeitsbereiche

- Jede in einer Funktion mit **`var`** deklarierte Variable ist auf den Rumpf der Funktion beschränkt:

```
var a = 1; // Globaler Gültigkeitsbereich
function test() {
  var a = 2;
  writeln(a); // Lokaler Gültigkeitsbereich!
}
test();
writeln(a);
```

Ausgabe:

```
2
1
```

- Verhalten bei Einsatz von **`let`** an Stelle von **`var`** identisch

Funktion: Zugriff auf globale Variable

- Ohne **`var`** erfolgt ein Zugriff auf die gleichnamige globale Variable

```
var a = 1; // Globaler Gültigkeitsbereich

function test() {
  a = 2; // var fehlt: Zugriff auf globale
         //              Variable

  writeln(a);

}

test();

writeln(a);
```

Ausgabe:

```
2
2
```

Funktion: Deklaration einer globalen Variable

- Wird im nicht strikten Modus in einer Funktion eine Variable implizit durch Zuweisung deklariert, erzeugt dies eine globale Variable
  - Im strikten Modus erfolgt eine Fehlermeldung (IDE) oder Abbruch des Scripts (Browser)

```
function test() {
  a = 2; // var fehlt: Globale Variable erzeugt
  writeln(a);
}
test();
writeln(a);
```

Ausgabe:

```
2
2
```

Innere Funktionen (Engl.: Nested Functions)

- Funktionsrümpfe können Definitionen von Funktionen enthalten
  - Die inneren Funktionen sind nur im Rumpf der äußeren Funktion verfügbar
- Beispiel:

```
function arithmetic (operation, x, y) {
    function add(x,y) { return x + y; }
    function subtract(x,y) { return x - y; }
    // etc.
    switch (operation) {
      case 1: return add(x, y);
      case 2: return subtract(x, y);
      // etc.
    }
}
arithmetic(1, 2, 3); // liefert 5
arithmetic(2, 2, 3); // liefert -1
```

Closure ("Verschluss")

- JavaScript nutzt einen lexikalischen Gültigkeitsbereich (engl.: *lexical scoping*)
- Funktionen sind zusammen mit dem Gültigkeitsbereich eingeschlossen, in dem sie deklariert sind
  - Werte globaler Variablen werden darum am *Deklarationsort* aufgelöst, nicht am *Aufrufort*
- Beispiel:

```
var scope = "global"; // A global variable

function f() { return scope; }

function checkscope() {
  var scope = "local"; // A local variable
  return f();
}
checkscope(); // liefert "global"
```

Gültigkeitsbereich und innere Funktionen

- Durch innere Funktionen lässt sich der Sichtbereich einer Funktion auf den Rumpf der äußeren Funktion einschränken
- Beispiel:

```
var scope = "global"; // A global variable

function checkscope() {
  var scope = "local"; // A local variable
  function f() { return scope; }
  return f();
}

checkscope(); // liefert "local"
```

- Die Funktion **`f`** "sieht" die Variable in **`checkscope`**

Closure und anonyme Funktion

- Beispiel:

```
function counter() {
  var i = 0;
  return function() {
    return ++i;
  }
}
var count = counter();
```

- Zurückgegeben wird von **`counter()`** ein Verweis auf den Codeblock der inneren Funktion inklusive aller hier sichtbaren internen Variablen
- Interne Variablen "überleben" auf diese Weise die Termination der Funktion **`counter()`**

Aufruf

```
function counter() {
   var i = 0;
   return function() {
     return ++i;
   }
}
var count = counter();
writeln(count());
writeln(count());
writeln(count());
writeln(counter());
```

- Ersatz für statische Variablen in z. B. Java, C++
- Ausgabe:

```
1
2
3

function () {
  return ++i;
}
```

Nachgelagerter Aufruf der inneren Funktion

- Beispiel:

```
function outer(a) {
  return function(b, c) {
    return a+b+c;
  }
}
writeln(outer(1)(2, 3)); // liefert 6
```

  - Die äußere Funktion und die innere anonyme Funktion besitzen Parameter
  - Der Wert des Parameters der äußeren Funktion wird dem Codeblock der inneren Funktion eingeschrieben

Funktion als Gültigkeitsbereich

- JavaScript besitzt traditionell keine blockbasierten, sondern funktionsbasierte Gültigkeitsbereiche
  - Kein *block level scope*, sondern *function scope*
- Hierdurch kann es zu Kollisionen durch doppelt vergebene Namen kommen
  - Z. B. beim Einbinden externer Programmmodule
- Abhilfe: Einkapselung von Variablen und Funktionen in einer umgebenden anonymen Funktion
  - Umgebende Funktion definiert einen eigenen Gültigkeitsbereich
  - Funktion lediglich als Container gedacht, aber nicht für Funktionsaufrufe

Funktion als Gültigkeitsbereich: Umsetzung

- Einkapselung geschieht in Form einer anonymen Funktion
- Syntaktisches Konstrukt:

Umschließende Klammern bewirken, dass das Konstrukt als auszuwertender Gesamtausdruck betrachtet wird und nicht als Funktionsdeklaration mit nachfolgendem leeren Ausdruck. Dies ermöglicht den Funktionsaufruf

```
(function() {
  // Lokale Variablen und Funktionen
})();
```

Deklaration der anonymen Funktion

Klammern stehen für den erforderlichen einmaligen Funktionsaufruf, der den Code zur Ausführung bringt

Objektorientierte Programmierung in JavaScript

Objektorientierte Programmierung in JavaScript

- Objektorientierte Programmierung erfolgt in JavaScript ohne Klassen auf der Grundlage von **Objektliteralen**
  - Objektliteral kann in der Folge als **Prototyp** für weitere Objekte dienen
- Klassenlose Objektorientiertheit
  - Seit ES6 existiert zusätzlich eine klassenbasierte Syntax
  - Diese ändert nichts am prototypenbasierten Modell
  - Sie erleichtert den Umstieg von klassenbasierten Sprachen
    - Im Hinblick auf die Grundfunktionen ist sie "syntaktischer Zucker"
  - Unterstützung in den gewünschten Zielbrowsern jeweils prüfen

Objekte als Literal erzeugen

- Ein Objektliteral ist eine in geschweiften Klammern notierte Liste von durch Kommas getrennten Eigenschaften bzw. Attributen
  - Eigenschaft: Durch Doppelpunkt getrenntes Paar aus Name und Wert
- Beispiel:

```
var person = {
    vorname:"Markus",
    nachname:"Mustermann",
    alter:25
};
```

Auf Objekteigenschaften zugreifen

- Der Zugriff erfolgt mittels Punktoperator oder im Sinne eines assoziativen Feldelements
- Beispiel:

```
var person = {
    vorname:"Markus",
    nachname:"Mustermann",
};

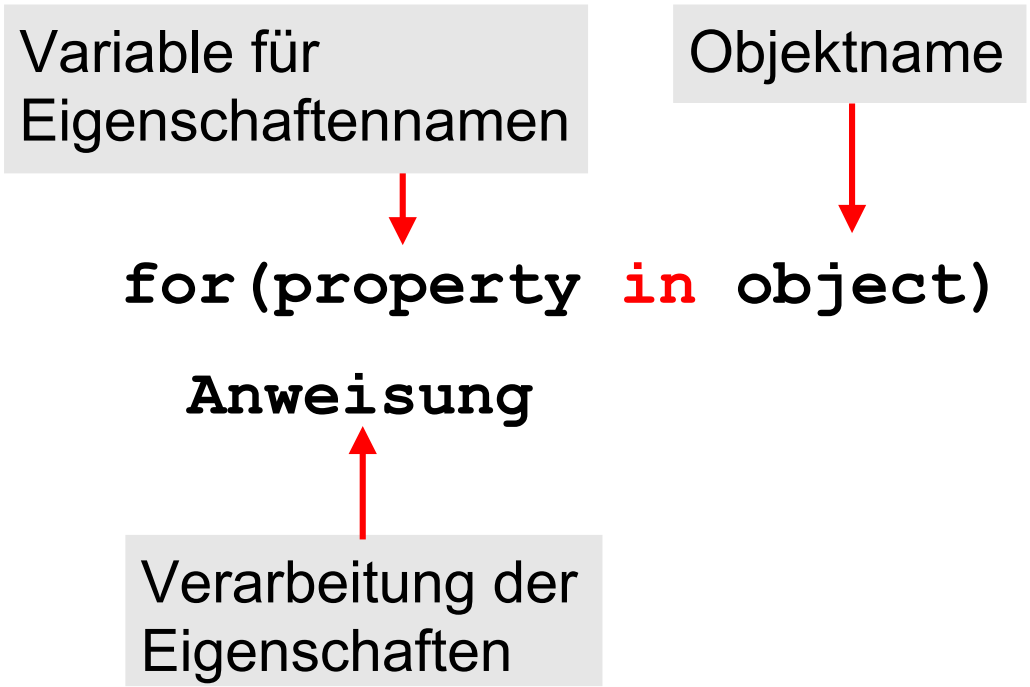
writeln(person.vorname + " " + person.nachname);

writeln(person["vorname"] + " " + person["nachname"]);
```

- Ausgabe jeweils: **Markus Mustermann**

`for-in`-Kontrollstruktur

- Wegen der Analogie zwischen assoziativen Feldern und Objekten ermöglicht es die `for-in`-Kontrollstruktur ebenfalls die Eigenschaften eines Objekts auszulesen
- Syntax:



Beispiel

```
var person = {
  vorname:"Markus",
  nachname:"Mustermann",
};

for (var property in person) {
  writeln(property + ": " + person[property]);
}
```

Ausgabe:

```
vorname: Markus
nachname: Mustermann
```

Kein Punktoperator, d. h. kein **`person.property`** und keine Anführungsstriche erlaubt hier

Objekte erweitern

- JavaScript-Objekte sind dynamisch, d. h. strukturell veränderbar
  - Ihre Eigenschaftsmenge kann zur Laufzeit erweitert werden
- Sehr flexibler Ansatz, der aber keine Code-Optimierungen erlaubt, wie sie bei statischen Sprachen möglich sind

Beispiel

```
var person = {
  vorname:"Markus",
  nachname:"Mustermann",
};
person.alter = 25;

for (var property in person) {
  writeln(property + ": " + person[property]);
}
```

Ausgabe:

```
vorname: Markus
nachname: Mustermann
alter: 25
```

Eigenschaften löschen

- Eigenschaften können mit Hilfe des **delete**-Operators aus einem Objekt gelöscht werden
- Beispiel:

```
var person = {
  vorname:"Markus", nachname:"Mustermann", alter:25
};
delete person.alter;
for (var property in person) {
  writeln(property + ": " + person[property]);
}
```

Ausgabe:

```
vorname: Markus
nachname: Mustermann
```

Objekte destrukturieren

- Wie Felder können auch Objekte seit ES6 destrukturiert werden
- Beispiel:

```
var person = {
    vorname:"Markus",
    nachname:"Mustermann",
    alter:25
};
```

- Varianten:
  - Einzelne Eigenschaft:

    ```
    const {vorname} = person;
    ```

  - Untermenge der Eigenschaften:

    ```
    const {vorname, alter} = person;
    ```

  - Alle Eigenschaften:

    ```
    const {vorname, nachname, alter} = person;
    ```

- Auf die Objekteigenschaften kann nach der Destrukturierung über die Konstanten lesend zugegriffen werden
- Verwenden von Variablen (**`var`** an Stelle von **`const`**) mit anschließendem Schreibzugriff ändert nicht die Objekteigenschaften

Methoden deklarieren

- Objektmethoden werden durch Zuweisen einer anonymen Funktion an eine Objekteigenschaft deklariert
- Beispiel:

```
var person = {
   vorname:"Markus",
   nachname:"Mustermann",
   ausgabe:function() {
      return this.vorname + " " + this.nachname;
   } // Keine Pfeilfunktionen möglich hier
};
writeln(person.ausgabe());
```

- liefert: **Markus Mustermann**

Zugriff auf Objekteigenschaften mittels **this**-Referenz

Aufruf der Methode mittels Punktoperator

Seit ES6: Vereinfachte Syntax

- Methoden können ohne anonyme Funktion deklariert werden
- Beispiel:

```
var person = {
   vorname:"Markus",
   nachname:"Mustermann",
   ausgabe() {
      return this.vorname + " " + this.nachname;
   }
};
writeln(person.ausgabe());
```

- liefert: **Markus Mustermann**

Zugriff auf Objekt-eigenschaften mittels **this**-Referenz

Aufruf der Methode mittels Punktoperator

Methodenparameter

- Methoden können wie Funktionen parametriert werden
- Beispiel:

```
var person = {
  vorname:"Markus", nachname:"Mustermann",
  ausgabe(htmltag) {
    return "<" + htmltag + ">" +
      this.vorname + " " + this.nachname +
      "</" + htmltag + ">";
  }
};
write(person.ausgabe("p"));
```

- liefert: `<p>Markus Mustermann</p>`

Mehr Objekte

- Beispiel:

```
var eineWeiterePerson = {
  vorname:"Marion", nachname:"Musterfrau",
  ausgabe(htmltag) {
    return "<" + htmltag + ">" +
      this.vorname + " " + this.nachname +
      "</" + htmltag + ">";
  }
};
write(eineWeiterePerson.ausgabe("p"));
```

- Codeduplikation durch Duplizierung der Methode **ausgabe**

Codeduplikation verringern

- Eine Funktion sei Methode unterschiedlicher Objekte
  - Die daraus resultierende Codeduplikation kann durch Deklarieren einer freien Funktion außerhalb der Objekte verringert werden
  - Die Funktion verweist auf das konkrete Objekt mittels **this**-Referenz
  - Nachteil: Der inhaltliche Bezug zwischen den Objekten und der Funktion geht verloren
- Beispiel:

```
var ausgabefkt = function () {
  return this.vorname + " " + this.nachname;
};
var person = {
  vorname:"Markus", nachname:"Mustermann",
  ausgabe:ausgabefkt
};
writeln(person.ausgabe()); //Markus Mustermann
```

Objekte mit `create` erzeugen

- Objekte können seit ES5 ebenfalls mit Hilfe der statischen Funktion **`Object.create()`** angelegt werden
- Beispiel:

```
var person = Object.create(null);

person.vorname = "Markus";
person.nachname = "Mustermann";
```

- Dieser Ansatz ist hauptsächlich sinnvoll für das Duplizieren (auch: Klonen) von Objekten
  - An Stelle von **`null`** wird dann ein prototypisches Objekt als Argument übergeben

Objekte duplizieren

- Existierendes Objekt, welches als Vorlage bzw. **Prototyp** dient, wird als Argument an `create` übergeben
- Das Duplikat übernimmt alle Eigenschaften des Prototyps
- Es wird ein komplett neues Objekt erzeugt, nicht nur eine weitere Referenz auf das gleiche Objekt
  - D. h. alle darauf folgenden Änderungen an den beiden Objekten betreffen nicht das jeweils andere Objekt

Objekte duplizieren mit `create()`: Beispiel

```
var person = {
   vorname: "Markus",
   nachname: "Mustermann",
   alter: 25,
   ausgabe() {
      return this.vorname + " " + this.nachname +
             ": " + this.alter;
   }
};
var duplikat = Object.create(person);
writeln(duplikat.ausgabe());
```

- Ausgabe: **`Markus Mustermann: 25`**

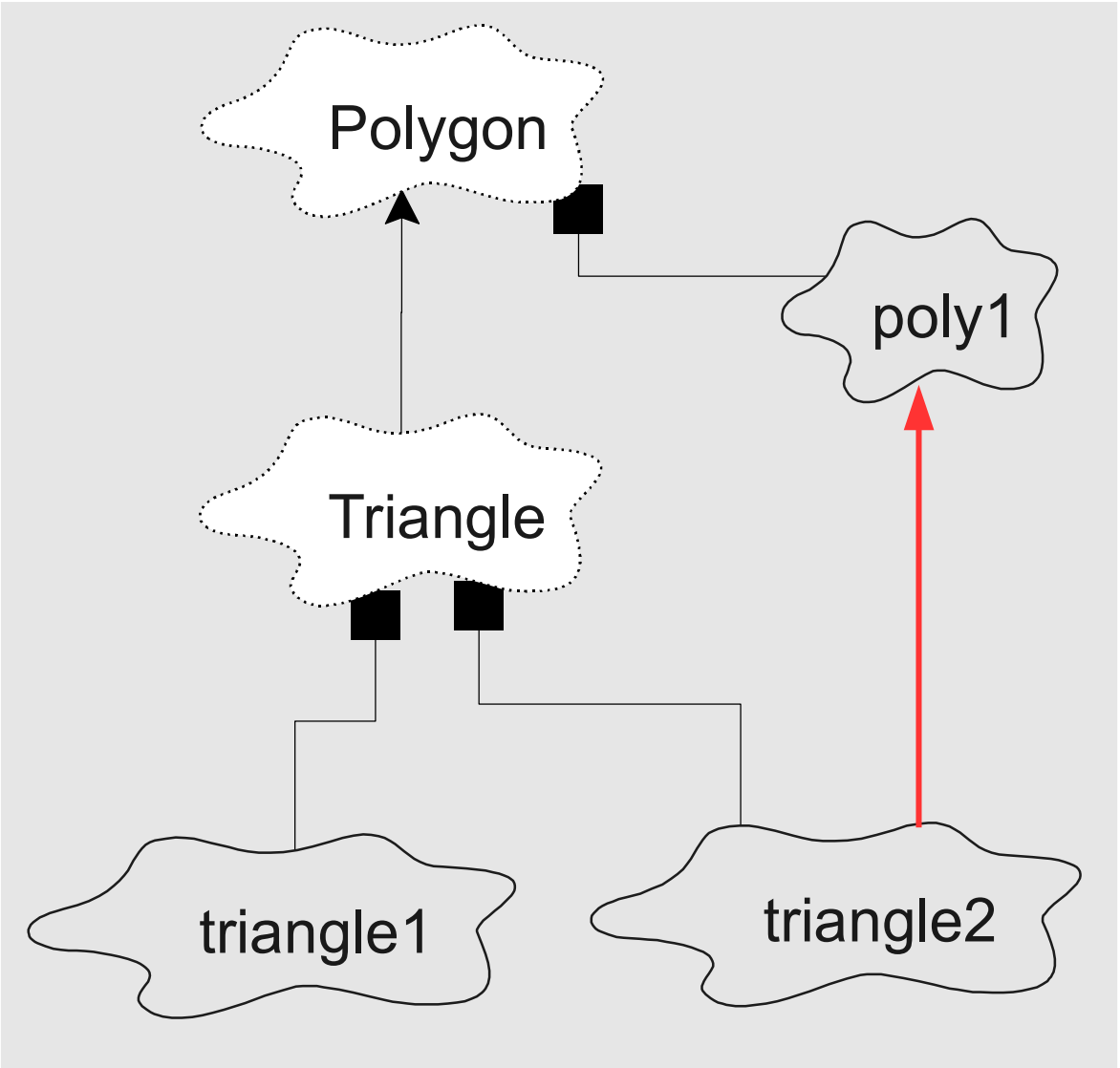
Objekt erweitern

- Duplikate können nach ihrer Erstellung um individuelle Eigenschaften erweitert werden
- Beispiel:

```
var duplikat = Object.create(person);
duplikat.wohnort = "Ulm";
duplikat.plz = 89075;
```

- Das Duplikat besitzt danach folgende Eigenschaften:
  - **`vorname`**, **`nachname`**, **`alter`**: Diese wurden vom Prototyp übernommen bzw. *geerbt*
  - **`wohnort`**, **`plz`**: Zusätzliche individuelle Eigenschaften, die später hinzugefügt wurden
  - Vererbung mit Erweiterung auf Objektbasis

Beispiel



- Java, C++: Vererbung findet zwischen Klassen, aber nicht zwischen Objekten statt
- JavaScript: Vererbung zwischen Objekten

Duplikateigenschaften anpassen

- Ersatz für die Initialisierung der Klassenvariablen durch den Konstruktor bei klassenbasierten Sprachen
- ```
  var person = {
    vorname: "Markus", nachname: "Mustermann",
    alter: 25,
    ausgabe() {/* wie zuvor */}
  };
  var duplikat = Object.create(person);
  duplikat.vorname = "Marion";
  duplikat.nachname = "Musterfrau";
  duplikat.alter = 30;
  writeln(duplikat.ausgabe());
  ```
- Ausgabe: Marion Musterfrau: 30
  ```

Eigenschaften mittels Methode initialisieren

```
var person = {
  init(vorname, nachname, alter) {
    this.vorname = vorname;
    this.nachname = nachname;
    this.alter = alter;
  },
  ausgabe() {/* wie zuvor */}
};
var einePerson = Object.create(person);
einePerson.init("Marion", "Musterfrau", 30);
write(einePerson.ausgabe());
```

- Ausgabe: Marion Musterfrau: 30

Konstruktoren

- Parametrische Objekte können auch dynamisch mit **Konstruktorfunktionen** erstellt werden
  - Beim Aufruf der Funktion mit **`new`** wird ein zu den übergebenen Funktionsargumenten passendes neues Objekt erzeugt
  - Diese Vorgehensweise ist effizienter als das Kopieren eines vorhandenen Objekts, wenn eine Reihe modifizierter Kopien dieses Objekts benötigt werden
  - Durch die Konstruktorfunktion wird ein **Prototyp** für Objekte einer gegebenen Art festgelegt
    - Konstruktorfunktionen repräsentieren die größte Annäherung des klassischen JavaScript an klassenbasierte Sprachen

Konstruktorfunktion: Beispiel

```
function Person(vorname, nachname, alter) {
  this.vorname = vorname;
  this.nachname = nachname;
  this.alter = alter;
  this.ausgabe = function() {
    return this.vorname + " " +
      this.nachname + ": " + this.alter;
  };
}
```

- Die Eigenschaften des Objekts werden mit Hilfe von **`this`**-Referenzen definiert
- Konvention: Konstruktorfunktionsnamen werden groß geschrieben

Objekt erstellen und benutzen

- Objekte erstellen: Aufruf der Konstruktorfunktion mit **`new`**

```
var p1 = new Person("Marianne", "Musterfrau",
                    30);

var p2 = new Person("Otto", "Normalverbraucher",
                    50);
```

- Objekte benutzen

```
writeln(p1.ausgabe());
writeln(p2.ausgabe());
```

- Ausgabe:

```
Marianne Musterfrau: 30
Otto Normalverbraucher: 50
```

Bedeutung von `new`

- Der Aufruf der Konstruktorfunktion mit **`new`** führt zur Erstellung eines neuen leeren **`Object`**-Objekts, als Hülle (engl.: *wrapper*) für das neue Objekt
  - Alle JavaScript-Objekte sind abgeleitet von **`Object`** als übergeordnetem Prototyp
- Die Konstruktorfunktion wird im Kontext dieses neuen Objekts ausgeführt und erweitert über die **`this`**-Referenzen dessen Eigenschaften

```
function Person(vorname, nachname, alter) {
  this.vorname = vorname;
  //... etc.
}
```

- Zurückgegeben wird eine Referenz auf das neue Objekt

```
var p1 = new Person("Marianne", "Musterfrau", 30);
```

Aufruf mit und ohne `new`

```
function Person(vorname, nachname, alter) {
  this.vorname = vorname;
  //... etc.
  writeln("this:" + this);
}
var p1 = new Person("Marianne", "Musterfrau", 30);
Person("Marianne", "Musterfrau", 30);
```

- Ausgabe:

this:[object Object]

this:[object Window]

Ohne **`new`** werden die Eigenschaften dem Scope des umgebenden Objekts, d. h. dem Browser-Window hinzugefügt

Danach existiert z. B. **`window.vorname`**!

Klassensyntax

- Erleichtert den Umstieg von klassenbasierten Sprachen
- Beispiel:

```
class Person {
  constructor(vorname, nachname, alter) {
    this.vorname = vorname;
    this.nachname = nachname;
    this.alter = alter;
  }
  ausgabe() {
    return this.vorname + " " +
           this.nachname + ": " + this.alter;
  }
}
var person = new Person("Markus","Mustermann",25);
```

Erzeugen vordefinierter Objekte

- Beispiele:
- `var o = new Object();`
  - Leeres Objekt erzeugen. Äquivalent zu {}
- `var d = new Date();`
  - Neues Datumsobjekt erzeugen
- `var a = new Array();`
  - Leeres Feld erzeugen. Äquivalent zu []

Felder: Deklaration und Initialisierung

- Variante 1 als Array-Literal:

```
var a = [23, 38, 14, -3, 0,
         14, 9, 103, 0, -56];
```

- Variante 2 mit Zuweisung von Elementen:

```
var a = new Array(); // oder: var a = [];
a[0] = 23;
...
a[9] = -56;
```

- Variante 3 mit Array-Konstruktor:

```
var a = new Array(23, 38, 14, -3, 0,
                  14, 9, 103, 0, -56);
```

Komposition

Komposition

- Komposition ist auf der Grundlage von Objektliteralen, Konstruktorfunktionen oder Klassen möglich
- Beispiel: Erstellt wird ein Objekt für ein Auto
  - Das Auto besteht aus Teilen, die für sich selber wieder durch Objekte definiert sind:
  - **`car`**
    - **`engine`**
    - **`wheel`**
    - **`door`**
      - **`window`**

Auto mit internem Motor als Objektliteral

```
var car = {
  engine: {
    start() {
      writeln("Engine started");
    },
    accelerator(rpm) {
      writeln(rpm + "rpm");
    },
    stop() {
      writeln("Engine stopped");
    }
  },
  //...
};
```

Komposition

- Objektdefinition kann Definition weiterer Objektliterale enthalten (im Beispiel **`engine`**)

Auto: Räder hinzufügen

```
var car = {
  engine:{…}, // wie zuvor
  wheel1:{
    inflate(pressure) {
      writeln("Wheel pressure: " + pressure);
    }
  },
  wheel2:{ /* Codeduplikation */ },
  wheel3:{ /* Codeduplikation */ },
  wheel4:{ /* Codeduplikation */ }
}
```

Codeduplikation verringern durch Auslagern

```
var inflator = function(pressure) {
  writeln("Func Wheel pressure: " + pressure);
};
var car = {
  engine:{…}, // wie zuvor
  wheel1:{inflate:inflator},
  wheel2:{inflate:inflator}, // usw.
};
```

- Problem: Der zu den **`wheelx`**-Objekten gehörende Code ist jetzt außerhalb der Objektstruktur definiert und unabhängig von dieser benutzbar

Konstruktorfunktionen

- Eigenschaften können auch mit **`new`** erzeugte Objekte sein

```
function Car() {
  this.engine = new Engine();
  this.leftDoor = new Door("Left door");
  this.rightDoor = new Door("Right door");
  this.wheels = [new Wheel(),new Wheel(),
                 new Wheel(),new Wheel()];
}
```

- Codeduplikation vermieden und Code inhaltlich an der richtigen Stelle

Motor

```
function Engine() {
  this.start = function() {
    writeln("Engine started");
  };
  this.accelerator = function(rpm) {
    writeln(rpm + "rpm");
  };
  this.stop = function() {
    writeln("Engine stopped");
  };
}
```

Räder

```
function Wheel() {
  this.inflate = function(pressure) {
    writeln("Wheel pressure: " + pressure);
  };
}
```

Fenster

```
function Window() {
  this.rollup = function() {
    writeln("Window up");
  };

  this.rolldown = function() {
    writeln("Window down");
  };
}
```

Tür

```
function Door(whichDoor) {
  // Komposition
  this.window = new Window();

  this.whichDoor = whichDoor;

  this.open = function() {
    writeln(this.whichDoor + " open");
  };

  this.close = function() {
    writeln(this.whichDoor + " closed");
  };
}
```

Hauptprogramm

```
var car = new Car();
car.leftDoor.window.rollup();
car.wheels[0].inflate(72);
car.leftDoor.close();
car.rightDoor.close();
```

Ausgabe:

```
Window up
Wheel pressure: 72
Left door closed
Right door closed
```

Umsetzung mit Klassensyntax (1)

```
class Window {
  rollup() {
    writeln("Window up");
  }
  rolldown() {
    writeln("Window down");
  }
}
```

```
class Engine {
  start() {
    writeln("Engine started");
  }
  accelerator(rpm) {
    writeln(rpm + "rpm");
  }
  stop() {
    writeln("Engine stopped");
  }
}
```

Umsetzung mit Klassensyntax (2)

```
class Wheel {
  inflate(pressure) {
    writeln("Wheel pressure: " + pressure);
  }
}
```

```
class Door {
  constructor(whichDoor) {
    this.whichDoor = whichDoor;
    this.window = new Window();
  }
  open() {
    writeln(this.whichDoor + " open");
  }
  close() {
    writeln(this.whichDoor + " closed");
  }
}
```

- Komposition
- Erzeugung des Fenster-Objekts muss im Konstruktor erfolgen
- Hierfür ist eine **`this`**-Referenz notwendig
- Diese ist nur in Methoden verfügbar, nicht außerhalb

Umsetzung mit Klassensyntax (3)

```
class Car {
  constructor() {
    this.engine = new Engine();
    this.leftDoor = new Door("Left door");
    this.rightDoor = new Door("Right door");
    this.wheels = [new Wheel(),new Wheel(),
                   new Wheel(),new Wheel()];
  }
}
```

- Hauptprogramm:

```
var car = new Car();
car.leftDoor.window.rollup();
car.wheels[0].inflate(72);
car.leftDoor.close();
car.rightDoor.close();
```

Vererbung

Vererbung

- Als Voreinstellung nutzen Konstruktorfunktionen das allgemeine **`Object`** als Prototyp
- Mit Hilfe der speziellen Eigenschaft **`prototype`** kann der Prototyp für eine Konstruktorfunktion festgelegt werden
- Beispiel: Benanntes Auto: Zusätzlich zu den Eigenschaften von **`Car`** besitze dieses einen Namen

```
function NamedCar(name) {
  this.name = name;
  this.write = function() {
    writeln(this.name);
  };
}
NamedCar.prototype = new Car();
var aNamedCar = new NamedCar("Named car");
aNamedCar.write(); // Ausgabe: Named car
```

Vererbungshierarchien

- Möglich über wechselseitiges Eintragen der Prototypen: Es entsteht eine „Prototypenkette“
- Beispiel: Amphibienfahrzeug

```
function NamedCar(name) {/* wie zuvor */}
NamedCar.prototype = new Car(); // wie zuvor

function AmphiCar(name) {
  this.name = name;
  this.swim = function() {
    writeln(this.name + " swims");
  };
}
AmphiCar.prototype = new NamedCar("Named car");
```

  - Der Prototyp muss mit Parameterwerten versehen werden

Hauptprogramm

```
var amphiCar = new AmphiCar("Amphicar");

amphiCar.leftDoor.window.rollup();
amphiCar.wheels[0].inflate(72);
amphiCar.leftDoor.close();
amphiCar.rightDoor.close();
amphiCar.write(); // NamedCar
amphiCar.swim();  // AmphiCar
```

- Ausgabe:

```
Window up
Wheel pressure: 72
Left door closed
Right door closed
Amphicar
Amphicar swims
```

- Verdeckung: Gleichnamige Eigenschaften des Prototyps werden durch eigene Eigenschaften überlagert
- **`write`** aus **`NamedCar`** liefert **`name`** aus **`AmphiCar`**

Zugriff auf Eigenschaften des Prototyps

- Möglich mit Hilfe der Funktion `getPrototypeOf()`
-
```
var amphiCar = new AmphiCar("Amphicar");
//...
var acproto = Object.getPrototypeOf(amphiCar);
writeln(acproto.name);
acproto.write();
```

- Ausgabe:

```
Named car
Named car
```

Verdeckung und Löschen von Eigenschaften

- Werden verdeckende Eigenschaften im erbenden Objekt gelöscht, so treten die ursprünglichen Eigenschaften des Prototyps wieder zu Tage
- ```
  var amphiCar = new AmphiCar("Amphicar");

  //...

  amphiCar.write();
  amphiCar.swim();
  delete amphiCar.name;
  amphiCar.swim();
  ```
- ```
  Ausgabe:
  Amphicar
  Amphicar swims
  Named car swims
  ```

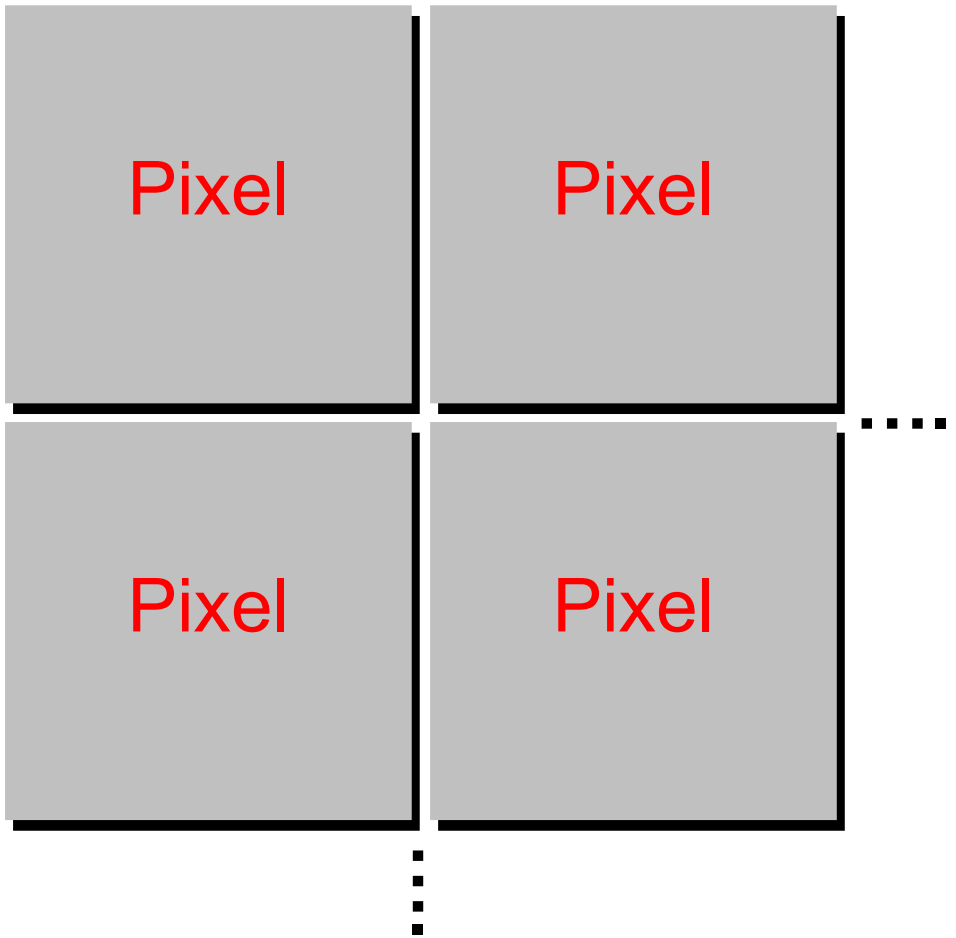
Vererbungshierarchien und Klassensyntax

- `class Car { ... } // wie zuvor`
- ```
  class NamedCar extends Car {
    constructor(name) {
      // Aufruf des Oberklassenkonstruktors erforderlich!
      super(); // Erstellung des Car-Objekts
      this.name = name;
    }
    write() {writeln(this.name);}
  }
  ```
- ```
  class AmphiCar extends NamedCar {
    constructor(name) {
      super(name);
    }
    swim() {writeln(this.name + " schwimmt");}
  }
  ```
- `var amphiCar = new AmphiCar("Amphicar");`

Canvas

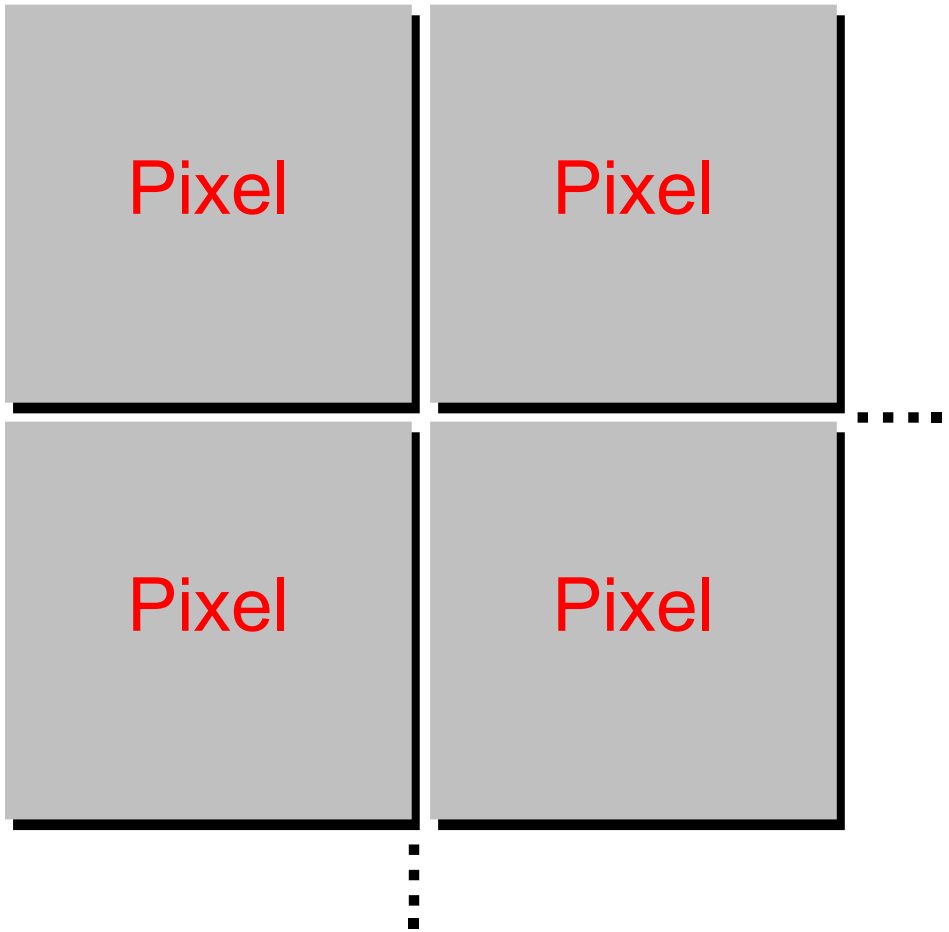
Raster Display

- Die Bildschirmanzeige besteht aus einer Matrix sogenannter **Pixel** oder **Pel**
  - *“Picture Elements”*
  - Diese sind quadratisch
- Die Pixel besitzen eine bestimmte Farbe
- Die auf dem Bildschirm darzustellenden Bildinhalte werden also aus einer Menge unterschiedlich getönter Pixel zusammengesetzt



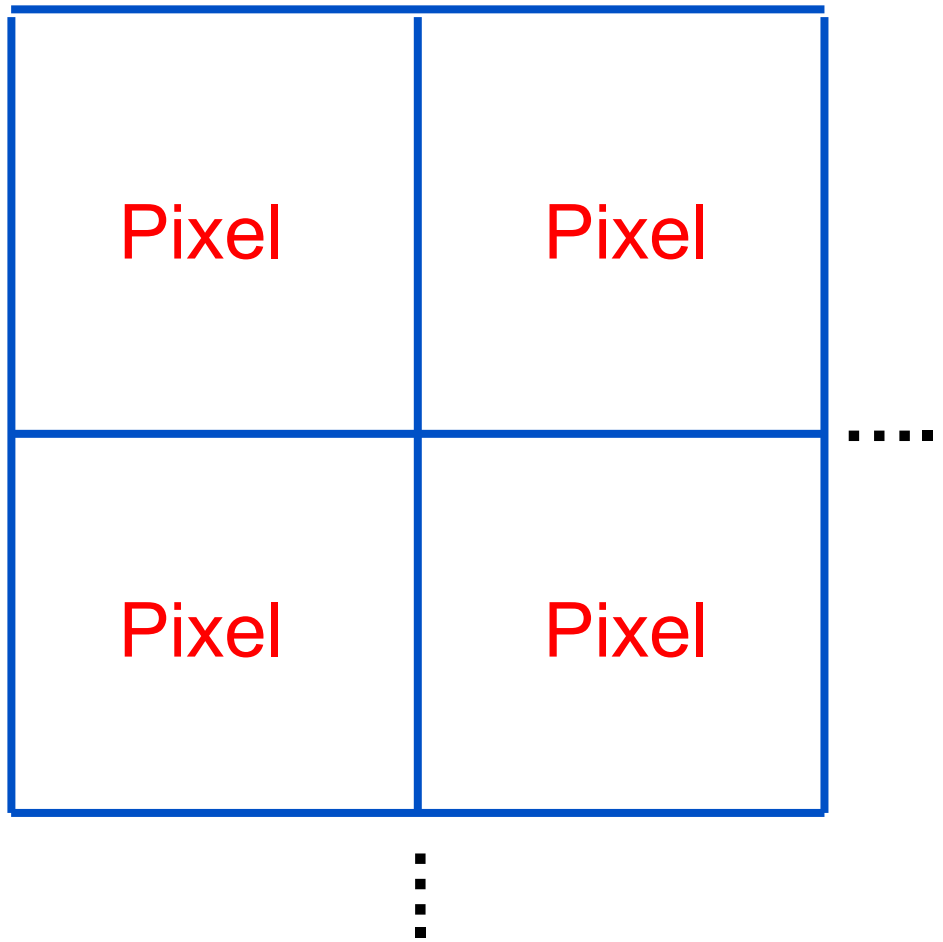
Raster Display: Zugriff auf einzelne Pixel (1)

- Während des Betriebs ändert sich die Bildschirmanzeige laufend
- D. h. die Farbwerte der Pixel müssen ständig aktualisiert werden
- Hierfür ist ein Zugriff auf die Pixel erforderlich
- Die Pixel müssen irgendwie adressiert werden können



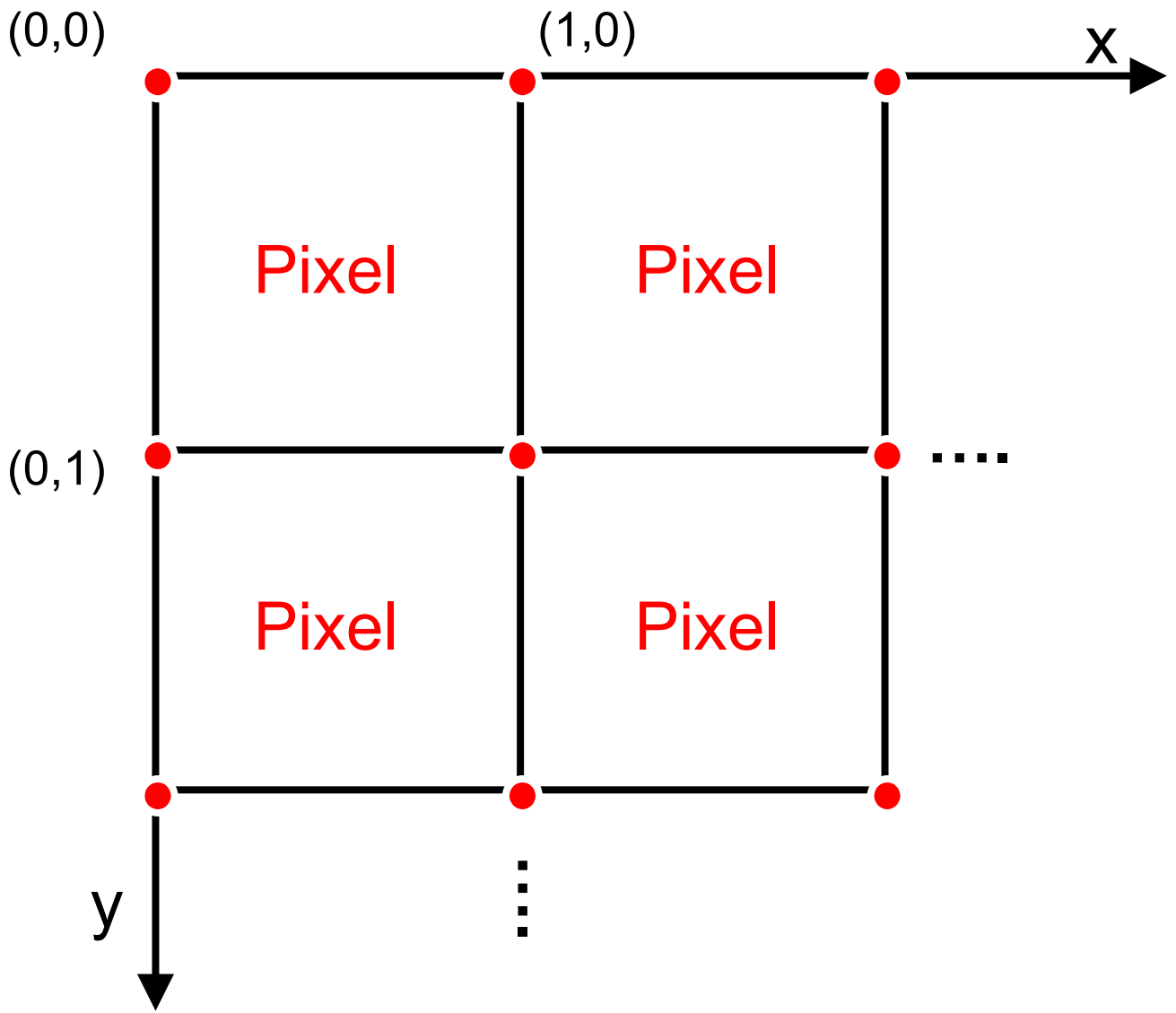
Raster Display: Zugriff auf einzelne Pixel (2)

- Benachbarte Pixel werden hierfür durch ein gedachtes **Gitternetz** voneinander separiert
  - Das Gitternetz definiert horizontale und vertikale Linien
  - Diese Linien können durchnummeriert werden
  - Die Nummern entsprechen dann Koordinaten



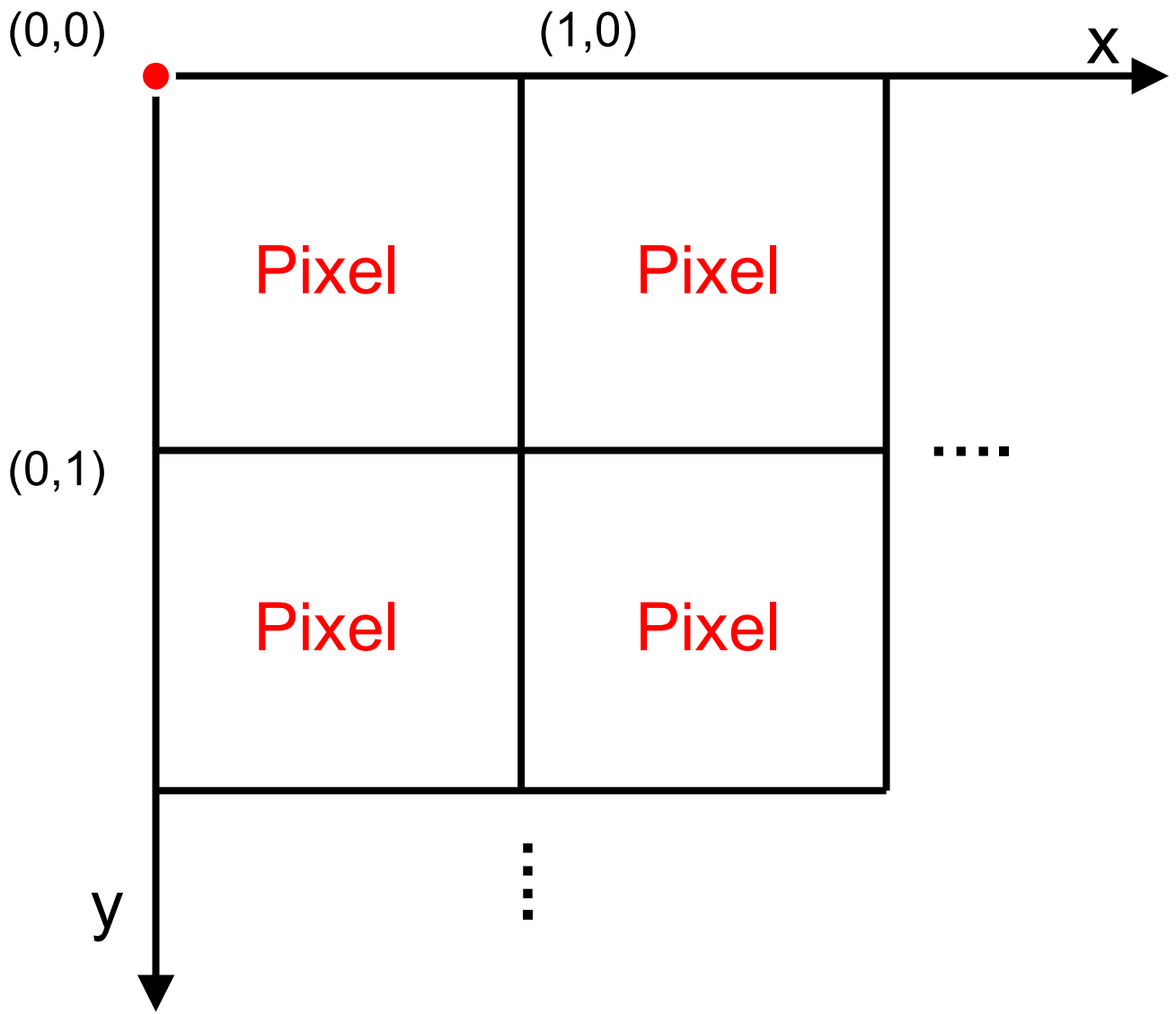
Raster Display – Koordinaten

- Die Schnittpunkte des Gitternetzes legen die Koordinaten eines kartesischen Koordinaten-systems fest
- Die y-Achse dieses Koordinaten-systems zeigt nach unten



Raster Display – Pixelkoordinaten

- Die Koordinate eines Pixels entspricht der Koordinate des Gitterpunktes der in der linken oberen Ecke des Pixels liegt
- Dies ist eine Konvention



JavaScript-Koordinatensysteme

- Dokumenten-(Document-)Koordinatensystem
  - Koordinatensystem des im Browser angezeigten Dokuments
  - Ursprung in der linken oberen Ecke des Dokuments
- Fenster-(Window-)Koordinatensystem
  - Koordinatensystem des im Fenster angezeigten Teils des Dokuments
  - Ursprung in der linken oberen Ecke des Sichtbereichs

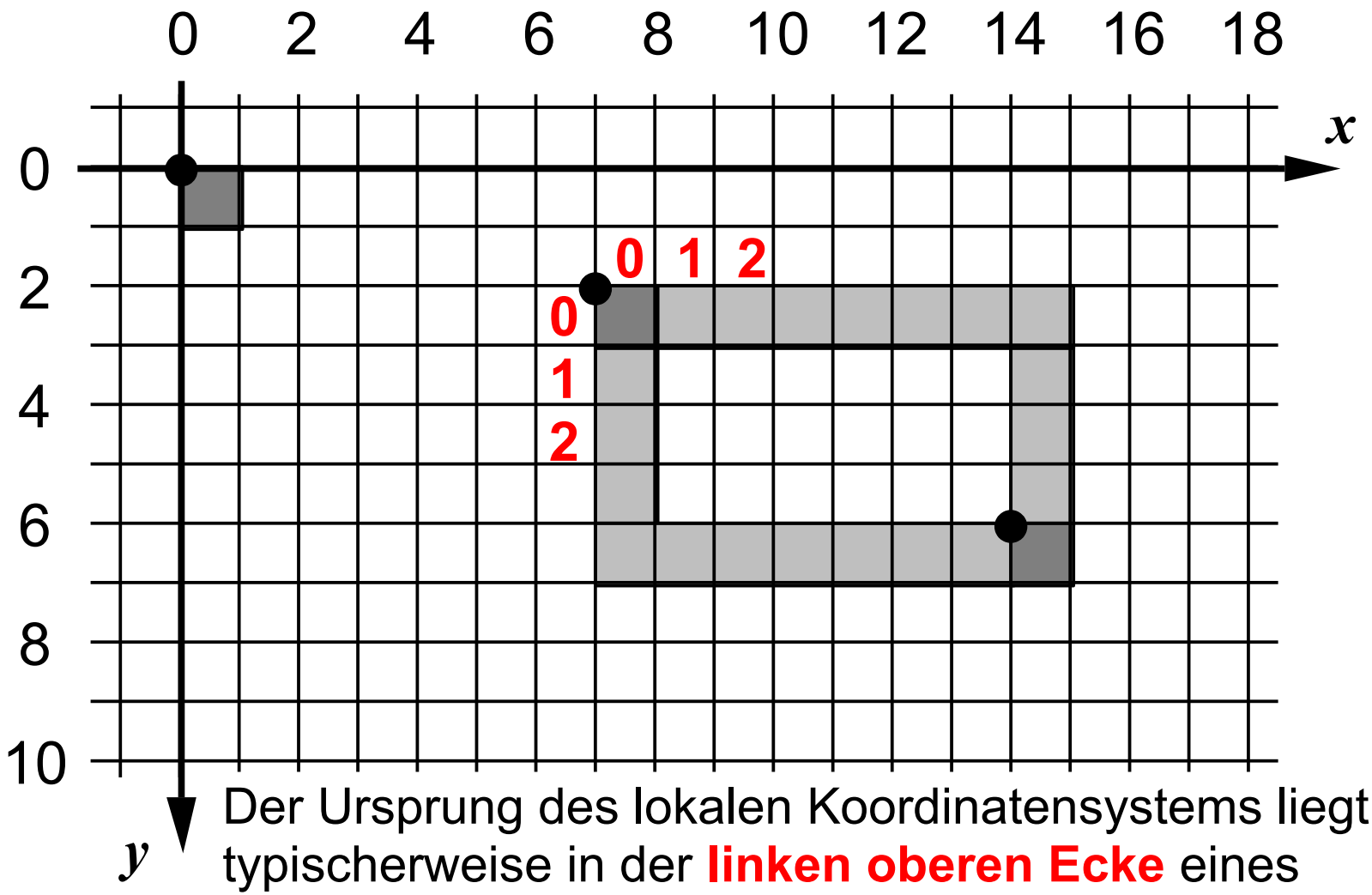
Canvas

- Im Webdokument als ein HTML-Element frei platzierbare zunächst leere "Leinwand", auf die gezeichnet werden kann
- Das Zeichnen im Canvas-Element erfolgt mit Hilfe von JavaScript-Befehlen in Form von Pixelgrafiken
  - Pixelgrafiken bestehen aus einer Ansammlung farbiger Pixel
- Jedes Canvas-Element verfügt über ein eigenes Canvas-Koordinatensystem
  - Der Ursprung dieses Systems liegt in der linken oberen Ecke des Canvas-Elements

Canvas-Koordinatensystem

- Im Canvas gezeichnete grafische Objekte besitzen eigene lokale Koordinatensysteme mit jeweils eigenem Ursprung
- Dieser Ursprung besitzt bezüglich des Canvas-Koordinatensystems zumeist von (0,0) verschiedene globale Koordinaten
  - Je nachdem, wo sich das Objekt im Canvas befindet
- Pixelkoordinaten können also lokal bezüglich eines grafischen Objekts und global bezüglich des Canvas-Elements interpretiert werden
  - Hier ist eine beliebig tiefe Schachtelung möglich

Globale und lokale Koordinaten



Der Ursprung des lokalen Koordinatensystems liegt typischerweise in der **linken oberen Ecke** eines grafischen Objekts

Zeichnen von Grafikobjekten

Canvas: Grafischer Kontext

- Für das Zeichnen stellt das Canvas-Element einen parametrierbaren grafischen Kontext zur Verfügung
- Dieser erlaubt das Erstellen von 2D-Grafiken sowie 3D-Grafiken mit Hilfe von WebGL
  - Wir betrachten hier nur die Erstellung von 2D-Grafiken
  - Im 2D-Kontext können Rechtecke und allgemeine Pfade gezeichnet werden

Beispiel: Zeichnen eines Rechtecks (1)

- Schritt 1: Canvas-Element in die HTML-Seite integrieren

```
<head>
  <!-- defer: Script erst ausführen, wenn canvas
       dargestellt -->
  <script src="script.js" defer></script>
</head>

<body>
  <canvas id="myCanvas" width="500" height="500">
    <p>Leider kein Canvas verfügbar</p>
  </canvas>
</body>
```

Beispiel: Zeichnen eines Rechtecks (2)

- HTML: `<canvas id="myCanvas" width="500" height="500">`
- Schritt 2: Mit JavaScript auf Canvas zugreifen

```
var canvas = document.getElementById("myCanvas");
```

- `getElementById()`: Zugriff auf das HTML-Element mit dem Identifikator `id` in der Dokumentenstruktur
  - DOM (Document Object Model): Schnittstelle zwischen HTML und JavaScript
  - Alle HTML-Elemente werden zu Objekten
  - Die Schachtelung der HTML-Elemente wird auf eine geschachtelte Struktur der Objekte abgebildet
  - Die Eigenschaften der Elemente können über die zugehörenden Objekte durch JavaScript ausgelesen und verändert werden

Beispiel: Zeichnen eines Rechtecks (3)

- Schritt 3: Zeichenkontext des Canvas-Elements definieren und Zeichenoperationen durchführen

```
var context = canvas.getContext("2d");
context.strokeStyle = "#FF0000"; // Farbe rot
context.lineWidth=10; // Strichstärke in Pixel

context.strokeRect(20,20,150,100); // oder:

// Rechteck definieren:
// context.rect(20,20,150,100);
// Rechteck zeichnen
// context.stroke();
```

Rechteck: Gefüllt und ungefüllt zeichnen

- Ungefüllt:

```
context.strokeStyle = "#FF0000";

context.strokeRect(x, y, width, height)
```

  - Zeichnet ein nicht gefülltes Rechteck
  - **`x`**, **`y`**: Position der linken oberen Ecke
  - **`width`**, **`height`**: Ausdehnung in Pixel in horizontaler und vertikaler Richtung

- Gefüllt:

```
context.fillStyle = "#FF0000";

context.fillRect(x, y, width, height)
```

  - Zeichnet ein gefülltes Rechteck

Beispiel: Mehrere Rechtecke zeichnen

```
context.lineWidth=10; // Strichstärke in Pixel
context.strokeStyle = "#FF0000"; // Farbe rot
context.strokeRect(20,20,150,100);

context.fillStyle = "#00FF00"; // Farbe grün
context.fillRect(20,150,150,100);

context.fillStyle = "#0000FF"; // Farbe blau
context.fillRect(20,275,150,100);
```

Rechteckobjekt: Gefüllt und ungefüllt zeichnen

- Alternative: Rechteck zunächst definieren

```
context.rect(x, y, width, height);
```

- Dann Rechteck ungefüllt zeichnen:

```
context.stroke();
```

- Oder gefüllt zeichnen:

```
context.fill();
```

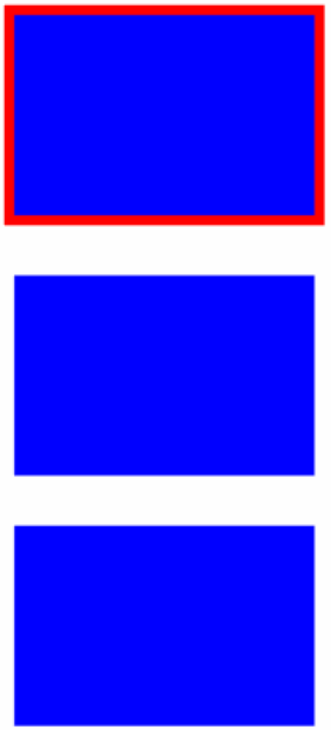
Rechteckobjekt: Mehrere Rechtecke zeichnen

```
context.rect(20,20,150,100); //Rechteck definieren
context.lineWidth = 10;
context.strokeStyle = "#FF0000";
context.stroke(); // Rechteck ungefüllt zeichnen
context.fillStyle = "#FFFF00"; // gelb

context.rect(20,150,150,100);
context.fillStyle = "#00FF00"; // grün

context.rect(20,275,150,100);
context.fillStyle = "#0000FF"; // blau

context.fill(); // Rechteck gefüllt zeichnen*/
```



- **`fill`** bezieht sich jeweils auf alle zuvor definierten Rechtecke
- Die zwischenzeitlichen Füllfarben werden ignoriert und alle Rechtecke werden mit der Farbe blau gefüllt
- Wiederholung der **`fills`** bringt nichts, da sich diese immer auf alle bislang definierten Rechtecke beziehen

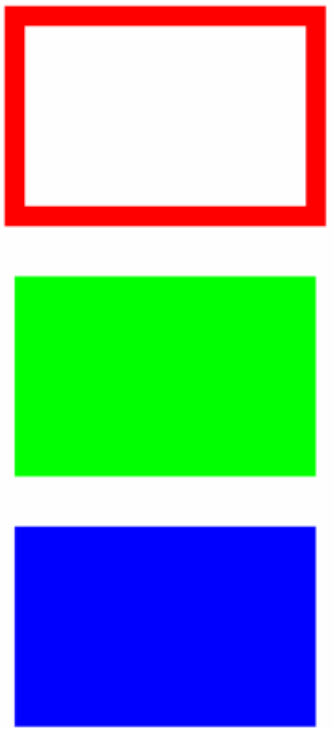
Unterschiedliche Stile

- Jeweils durch Aufruf von **`beginPath()`** einen neuen Pfad anlegen

```
context.rect(20,20,150,100);
context.lineWidth = 10;
context.strokeStyle = "#FF0000";
context.stroke(); // Rechteck zeichnen

context.beginPath(); // Neuer Pfad
context.rect(20,150,150,100);
context.fillStyle = "#00FF00";
context.fill(); // Rechteck zeichnen

context.beginPath(); // Neuer Pfad
context.rect(20,275,150,100);
context.fillStyle = "#0000FF";
context.fill(); // Rechteck zeichnen
```



Zeichnen von Kreisen

- Kreise werden als Pfad für ein Kreissegment gezeichnet:

  `arc(x,y,r,sAngle,eAngle,counterclockwise);`

  - **`x`**, **`y`**: Koordinaten des Kreismittelpunkts
  - **`r`**: Kreisradius
  - **`sAngle`**: Startwinkel in Radiant. 0°-Position ist die positive x-Achse
  - **`eAngle`**: Endwinkel in Radiant.
  - **`counterclockwise`**: Optionaler Parameter. Falls **`true`** erfolgt das Zeichnen im Gegenuhrzeigersinn. Voreingestellt ist Uhrzeigersinn
- Ein Vollkreis entspricht einem Kreispfad von 360° bzw. 2π

Kreis zeichnen: Beispiel

```
context.lineWidth = 5;

// Einen neuen Pfad für einen Kreis anlegen
context.beginPath();

// Kreisbogen zum Pfad hinzufügen
context.arc(100,100,50,0,2*Math.PI);

context.stroke(); // Pfad ungefüllt zeichnen
// context.fill(); // Pfad gefüllt zeichnen
```

- Pfadoperationen erweitern solange den aktuellen Pfad, bis durch Aufruf von **`beginPath()`** ein neuer Pfad angelegt wird

Zeichnen von freien Pfaden

(1) Mit **`context.beginPath()`** einen neuen Pfad anlegen

(2) "Zeichenstift" an die gewünschte Anfangsposition verschieben, sofern er sich dort noch nicht befindet: **`moveTo(x, y)`**

(3) Gewünschte Kurvenform definieren bis Endpunkt erreicht

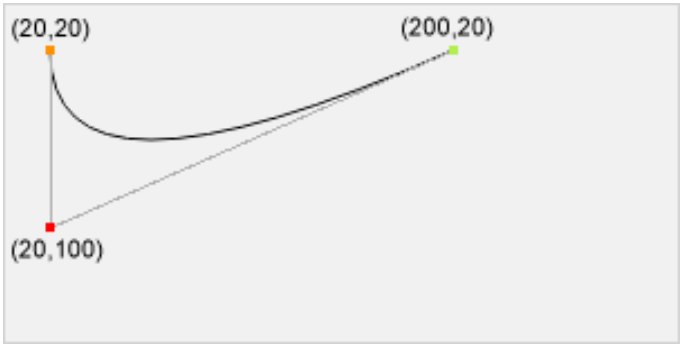
- **`lineTo(x, y)`**
- Weitere: **`quadraticCurveTo()`**, **`bezierCurveTo()`**, **`arcTo()`**

(4) Pfad mit **`context.stroke()`** oder **`context.fill()`** zeichnen

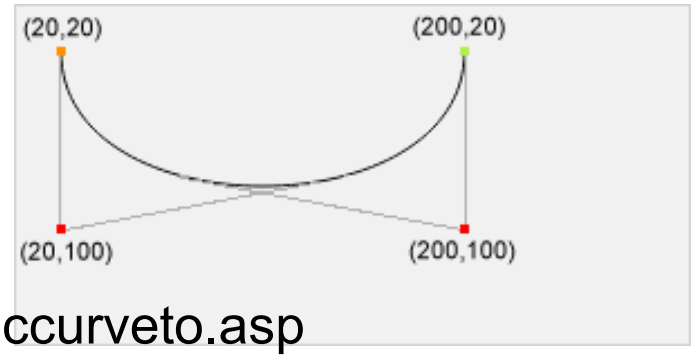
- Endpunkt ist jeweils Startpunkt für die nächste Zeichenoperation
- Pfade können mit **`closePath()`** automatisch geschlossen werden
- Siehe auch Dokumentation unter: www.w3schools.com/tags/ref_canvas.asp

Polynomiale Kurven

- Quadratische Bézier-Kurve:
  - **`context.quadraticCurveTo(cpx,cpy,x,y);`**
  - Ein Kontrollpunkt (**`cpx`**, **`cpy`**)
  - Immer in Kombination mit **`moveto`** für den Startpunkt und (**`x`**, **`y`**) als Endpunkt



- Kubische Bézier-Kurve:
  - **`context.bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y);`**
  - Zwei Kontrollpunkte



- Siehe hierzu auch:
  https://www.w3schools.com/tags/canvas_quadraticcurveto.asp
  https://www.w3schools.com/tags/canvas_beziercurveto.asp

Stiftparameter einstellen

- Stiftparameter werden vor der Zeichenoperation eingestellt
- Stile für ungefüllte und gefüllte Pfade oder andere grafische Objekte, wie z. B. Rechtecke, zuweisen an die Variablen
  - **`strokeStyle`** und/oder **`fillStyle`**
- Es kann entweder eine **Farbe**, ein **Farbverlauf** oder ein **Muster** angegeben werden
- Einstellungen bleiben bis zur Neudefinition eines Stils erhalten
- Weiteres zu Stilen: siehe www.w3schools.com/tags/ref_canvas.asp

Stiftparameter: Farbe

- CSS-Farbwert
  - Rot z. B. **`"#FF0000"`**, **`"red"`**, **`"rgb(255, 0, 0)"`**
- Beispiel:

```
context.lineWidth = 5;
context.beginPath();
context.moveTo(100,100);
context.lineTo(150,100);
context.stroke();

context.strokeStyle = "red";
context.beginPath();
context.moveTo(100,200);
context.lineTo(150,200);
context.stroke();
```

- Voreinstellungswert der Farbe ist schwarz

Bilder anzeigen

- Ein Canvas kann nicht nur für Zeichenoperationen, sondern auch für die Anzeige von Bildern genutzt werden
- Die Bilder können entweder direkt in der Canvasfläche angezeigt werden oder das Muster für eine Zeichenoperation liefern
- Zunächst müssen die Bilder eingelesen werden
- Geschieht dies mit JavaScript, dann sinnvollerweise auf der Grundlage von ereignisbasierter Programmierung
- Hierfür bietet JavaScript ein Ereignisbehandlungssystem (engl.: *event handling system*)

Ereignisbehandlung

- Ereignisse werden durch Ereignisquellen (engl.: *event source*) ausgelöst
- Mögliche Quellen für Ereignisse im Browser:
  - Elemente der grafischen Bedienoberfläche einer Webseite
  - Script, welches Daten aus dem Internet lädt
  - Bewegen der Maus innerhalb einer Webanwendung
  - Drücken einer Taste auf der Tastatur
- Ereignisse werden durch zugeordnete Funktionen behandelt:
  - Ereignisbehandler (engl.: *event handler*),
    auch: Ereignishörer (engl.: *event listener*)
- Zur Klassifikation gehören Ereignisse einem bestimmten Typ an und besitzen ein zugehörendes Ereignisobjekt

Ereignistypen

- Formularereignis (engl.: *form event*)
  - Ältester Ereignistyp: Formulare und Hyperlinks waren die ersten scriptfähigen Seitenelemente
  - Formular abschicken, abbrechen: submit, reset event
  - Hauptaufgabe für JavaScript: Überprüfung und ggf. auch Verarbeitung der Formulareingaben
  - Button, Check Box drücken: click event
  - Text eingeben:
    - input event bei Zeicheneingabe; change event nach Abschluss der Texteingabe bzw. Wechsel des Formularelements
- `onload`-Ereignis für das Nachladen externer Dokumente

Externes Bild anzeigen

```
var img = new Image();
img.src = "einBild.jpg";
img.onload = function() { // Ereignisbehandler
  // Direktes Anzeigen des Bilds im Canvas:
  // context.drawImage(VerweisAufBild, x, y);
  context.drawImage(img, 200, 50);
};
```

- Sobald das Bild fertig geladen wurde, wird das **`onload`**-Ereignis ausgelöst
- Die für dieses Ereignis zugewiesene Funktion wird dann ausgeführt

Externes Bild als Stift-Muster

- `context.createPattern(image,"repeat|repeat-x|repeat-y|no-repeat");`
  - **`image`**: Geladenes oder mit Webseite verlinktes Bild
  - **`repeat...`**: Ob Bild horizontal bzw. vertikal fortgesetzt werden soll, um Zeichnungsbereich zu füllen (Standard: **`repeat`**)

```
var img = new Image();
img.src = "einBild.jpg";
img.onload = function() { // Ereignisbehandler
  var pat=context.createPattern(this,"repeat");
  context.fillStyle=pat;
  context.fillRect(20,20,150,100);
};
```

Verlinktes Bild als Muster nutzen

```
<body>
  <img style="display:none" id="myImage"
       src="einBild.jpg">
  <canvas id="cva" width="300" height="200"></canvas>
  <script>
    var canvas  = document.getElementById("cva");
    var context = canvas.getContext("2d");
    var img=document.getElementById("myImage");
    var pat=context.createPattern(img,"repeat");
    context.fillStyle=pat;
    context.fillRect(0,0,300,200);
  </script>
</body>
```

- Bildelement unsichtbar, da Bild nur im Canvas sichtbar sein soll

Stiftparameter: Linearer Farbverlauf

- `grad = context.createLinearGradient(xs,ys,xe,ye);`
  - **`xs,ys,xe,ye`**: Koordinaten des Start- und Endpunkts des Farbverlaufs
  - Die Ausdehnung des Farbverlaufs an die Größe des gezeichneten Bereichs anpassen
- Festlegung der Farben des Farbverlaufs:
- `grad.addColorStop(stop,color);`
  - **`stop`**: Wert zwischen 0 und 1.
    0 steht für den Anfang und 1 für das Ende des Farbverlaufs.
    Wert steuert die Position der gewählten Farbe im Farbverlauf
  - **`color`**: Farbwert der gewählten Farbe.
    - CSS-Farbwert, d. h. **`#FF0000`**, **`rgb(255, 0, 0)`**, **`red`**, ...

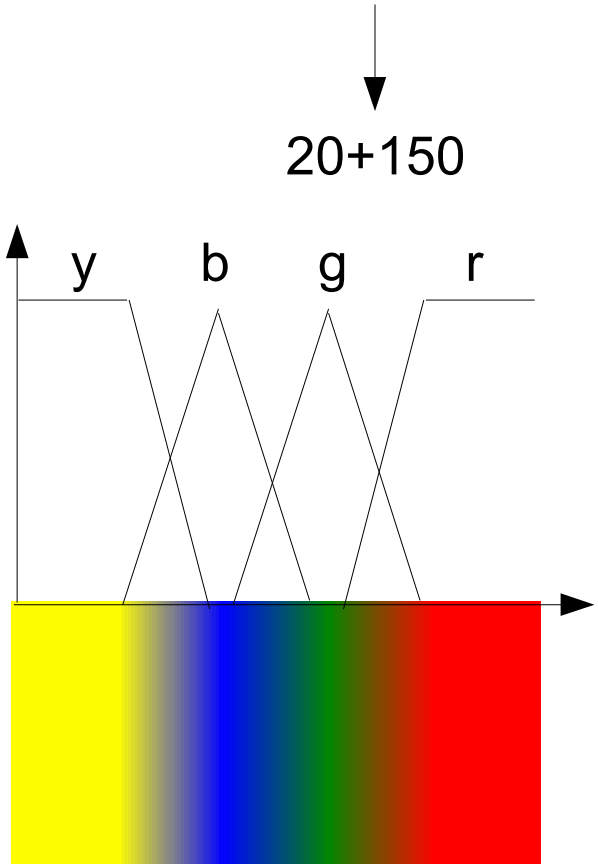
Linearer Farbverlauf: addColorStop

- Ohne Color Stops ist der Farbverlauf schwarz mit 100% Transparenz
- Ein Color Stop: Der gesamte Farbverlauf besitzt diese eine Farbe
- Zwei oder mehr Color Stops: 0, cs1, cs2,…, csi, 1
- 0...cs1: Die Farbe des ersten Stopps wird angezeigt
- csi...1: Die Farbe des letzten Stopps wird angezeigt
- cs1, cs2,…, csi: Die Farben werden gleichmäßig interpoliert zwischen den Stopps

Linearer Farbverlauf: Beispiel

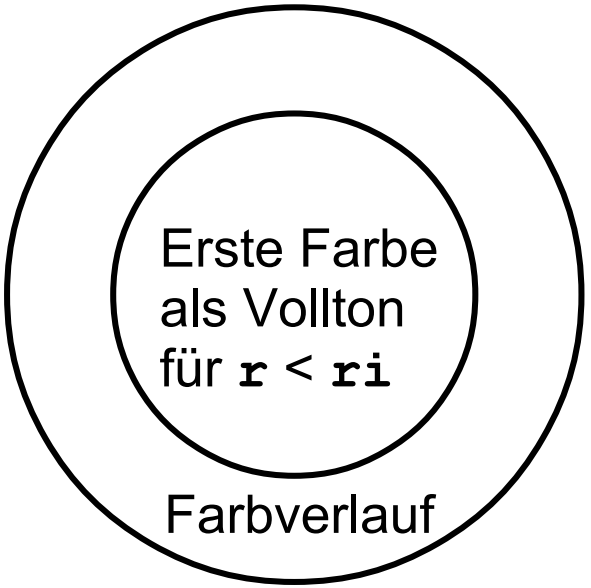
- Position und Ausdehnung des Farbverlaufs an die Größe des gezeichneten Bereichs anpassen
-

```
var grd=context.createLinearGradient(20,0,170,0);
grd.addColorStop(0.2,"yellow");
grd.addColorStop(0.4,"blue");
grd.addColorStop(0.6,"green");
grd.addColorStop(0.8,"red");
context.fillStyle=grd;
context.fillRect(20,20,150,100);
```



Stiftparameter: Radialer Farbverlauf

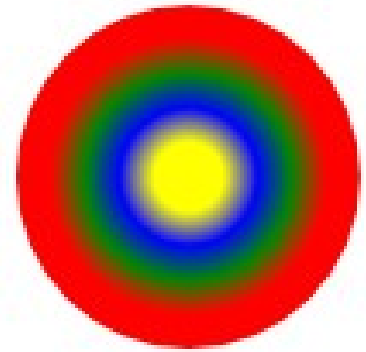
- "Farbverlaufsring", definiert auf der Grundlage zweier, überlagerter Kreise
- **`context.createRadialGradient(`**
  `xi,yi,ri,xa,ya,ra`
  `);`
  - **`xi,yi,xa,ya`**: Koordinaten des Mittelpunkts des inneren und äußeren Kreises
  - **`ri`**, **`ra`**: Radius des inneren und äußeren Kreises
  - Interpolation der Farben zwischen **`ri`** und **`ra`**
- Festlegung der Farben des Farbverlaufs: wie zuvor mit **`addColorStop()`**

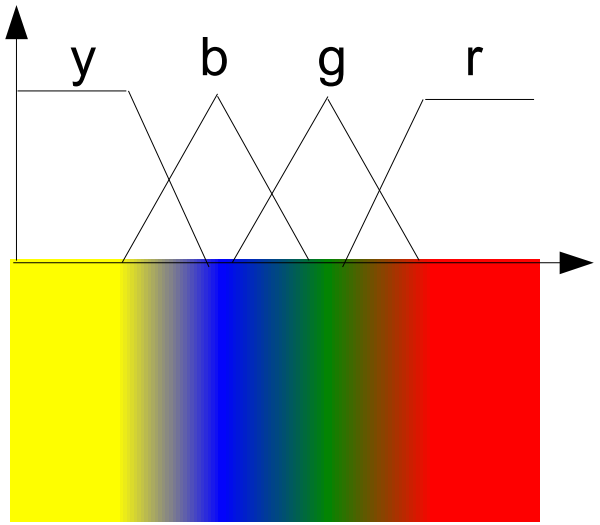


Letzte Farbe als Vollton für **`r`** > **`ra`**

Radialer Farbverlauf: Beispiel

```
var radgrd=context.createRadialGradient(
                100,100,0,100,100,50);
radgrd.addColorStop(0.2,"yellow");
radgrd.addColorStop(0.4,"blue");
radgrd.addColorStop(0.6,"green");
radgrd.addColorStop(0.8,"red");
context.fillStyle=radgrd;
context.beginPath();
context.arc(
  100,100,50,0,2*Math.PI
);
context.fill();
```





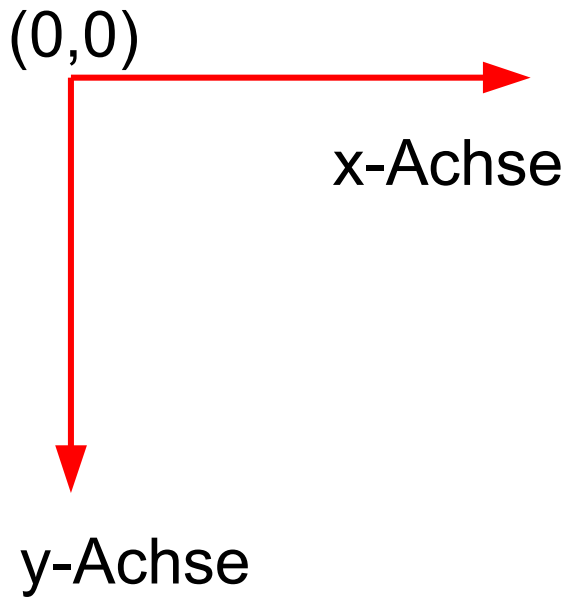
Dynamisches SVG

SVG

- **S**calable **V**ector **G**raphics ("Skalierbare Vektorgrafiken")
- Standard des World Wide Web Consortium (W3C) für verlustfrei skalierbare, zweidimensionale Vektorgrafiken
  - Für Anwendung im Web konzipiert
  - Mit Texteditor bearbeitbare Auszeichnungssprache
  - Heutzutage direkte Unterstützung in den gängigen Browsern. Früher waren Plug-Ins hierfür erforderlich
  - Animationen werden von SVG mittels SMIL unterstützt
    - Synchronized Multimedia Integration Language
    - Auszeichnungssprache für zeitsynchronisierte, multimediale Inhalte

SVG: Koordinatensystem

- SVG-Grafiken werden in einem kartesischen Koordinatensystem platziert
  - y-Achse ist nach unten gerichtet
- Koordinatenwerte können Gleitkommazahlen sein
  - Rastergrafik erlaubt demgegenüber nur ganzzahlige Koordinatenwerte
- Für Längenangaben stehen die von CSS bekannten Einheiten zur Verfügung
  - **`pt`**, **`mm`**, **`cm`**, **`px`**, **`em`** etc.



SVG-Grafiken

- Beruhen auf einfachen Grundelementen, den so genannten grafischen Primitiven
- Zu diesen gehören
  - Rechteck: **`<rect />`**
  - Kreis: **`<circle />`**
  - Pfad: **`<path />`**
  - Polygon: **`<polygon />`**
  - Text: **`<text />`**
  - Bild: **`<image />`**
- Grafische Primitiven werden zu komplexen Grafiken kombiniert

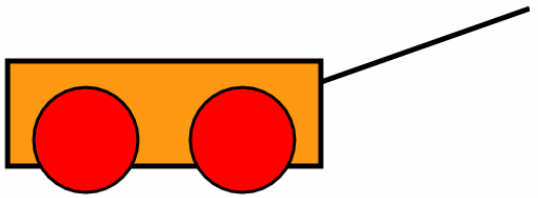
Beispiel: Handwagen

```
<svg width="600" height="600">
  <g transform="scale(1, 1)">
    <rect style="stroke-width:5; opacity:1;
      fill:#ff950e; fill-opacity:1; stroke:#000000;
      stroke-opacity:1"
      width="300" height="100" x="100" y="100" />

    <line x1="400" y1="120" x2="600" y2="50"
      stroke="black" stroke-width="5" />

    <circle cx="175" cy="175" r="50" stroke="black"
      stroke-width="3" fill="red" />

    <circle cx="325" cy="175" r="50" stroke="black"
      stroke-width="3" fill="red" />
  </g>
</svg>
```



SVG: Integration in Webseite

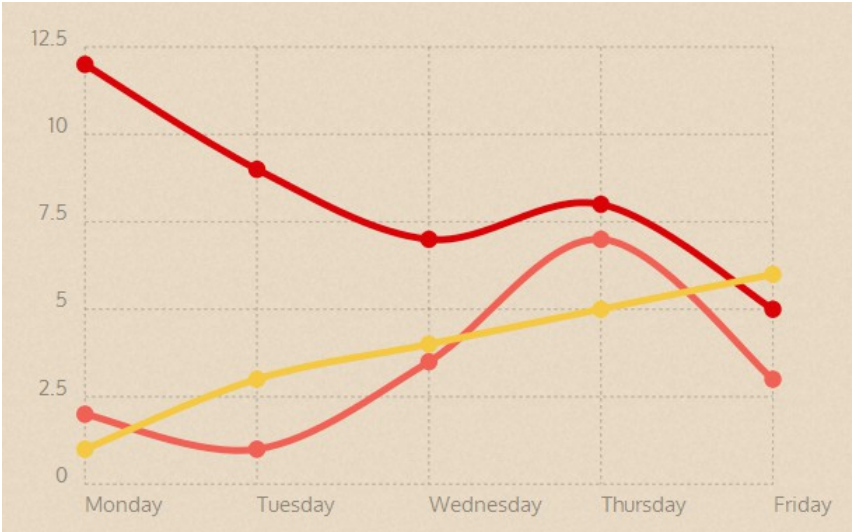
- Unter HTML5 können SVG-Grafiken direkt in eine Webseite geschrieben werden
- Beispiel:

```
<!DOCTYPE html>
<html>
<body>
<h1>SVG-Grafik</h1>
<svg width="100" height="100">
<circle cx="50" cy="50" r="40" stroke="red"
stroke-width="4" fill="yellow" />
</svg>
</body>
</html>
```

Chartist

Chartist

- Autor: Gion Kunz
- Diagrammsoftware, die SVG für die grafische Darstellung nutzt
- SVG-Grafiken werden in Chartist mittels JavaScript-Methoden produziert und in die DOM-Struktur einer Webseite eingebettet



  - Das SVG erscheint nicht explizit im Markup
  - SVG-Grafiken können mit Hilfe der Entwicklerwerkzeuge eines Browsers aus dem DOM als SVG-Code herauskopiert werden, um diese anderweitig zu nutzen
- Die Formatierung der Grafiken geschieht mit CSS bzw. SASS
- Abb.: Beispiel aus der Chartist-Dokumentation

Chartist-Diagramme in Webseite einbinden

- Im Kopfbereich JavaScript- und Stylesheet-Datei einbinden:

```
<head>
<link rel="stylesheet"
      href="chartist.min.css">
<script src="chartist.min.js"></script>
</head>
```

- Im Körperbereich für jedes Diagramm ein ** **-Element vorsehen:

```
<body>
<div class="mychart ct-perfect-fourth"></div>
</body>
```

Lokale Kopie oder CDN

- Einbinden lokaler Kopien von **`chartist.min.js`** bzw. **`-.css`**
- Alternative: Content Delivery Network (CDN)
  - Einbinden durch "Hotlinking" bzw. "Inline Linking"
  - Eingebundene Medien, z. B. Video, Musik oder JavaScript entstammen einem anderen Server, als die sie enthaltende Seite
- CDN: Anbieter derartiger Medien (z. B. YouTube)
- Häufig kostenlose Dienstleistung, da über die angebotenen Daten Einkünfte generiert werden können (z. B. durch Werbung)
- CDN für JavaScript: JSDelivr
  - **`https://www.jsdelivr.com/`**

Chartist über CDN JSDelivr

```
<head>
  <link rel="stylesheet"
    href="https://cdn.jsdelivr.net/chartist.js/
          latest/chartist.min.css">
  <script
    src="https://cdn.jsdelivr.net/chartist.js/
         latest/chartist.min.js">
  </script>
</head>
```

Diagrammgröße

- ```
  <body>

  </body>
  ```
- Chartist-Diagramme sind responsiv
  - Um diese Möglichkeit zu nutzen, erfolgt die Größenanpassung auf der Grundlage von Größenverhältnissen und nicht mit festen Maßangaben
- Beispiele für vorgegebene Größenverhältnisse ("aspect ratios"):

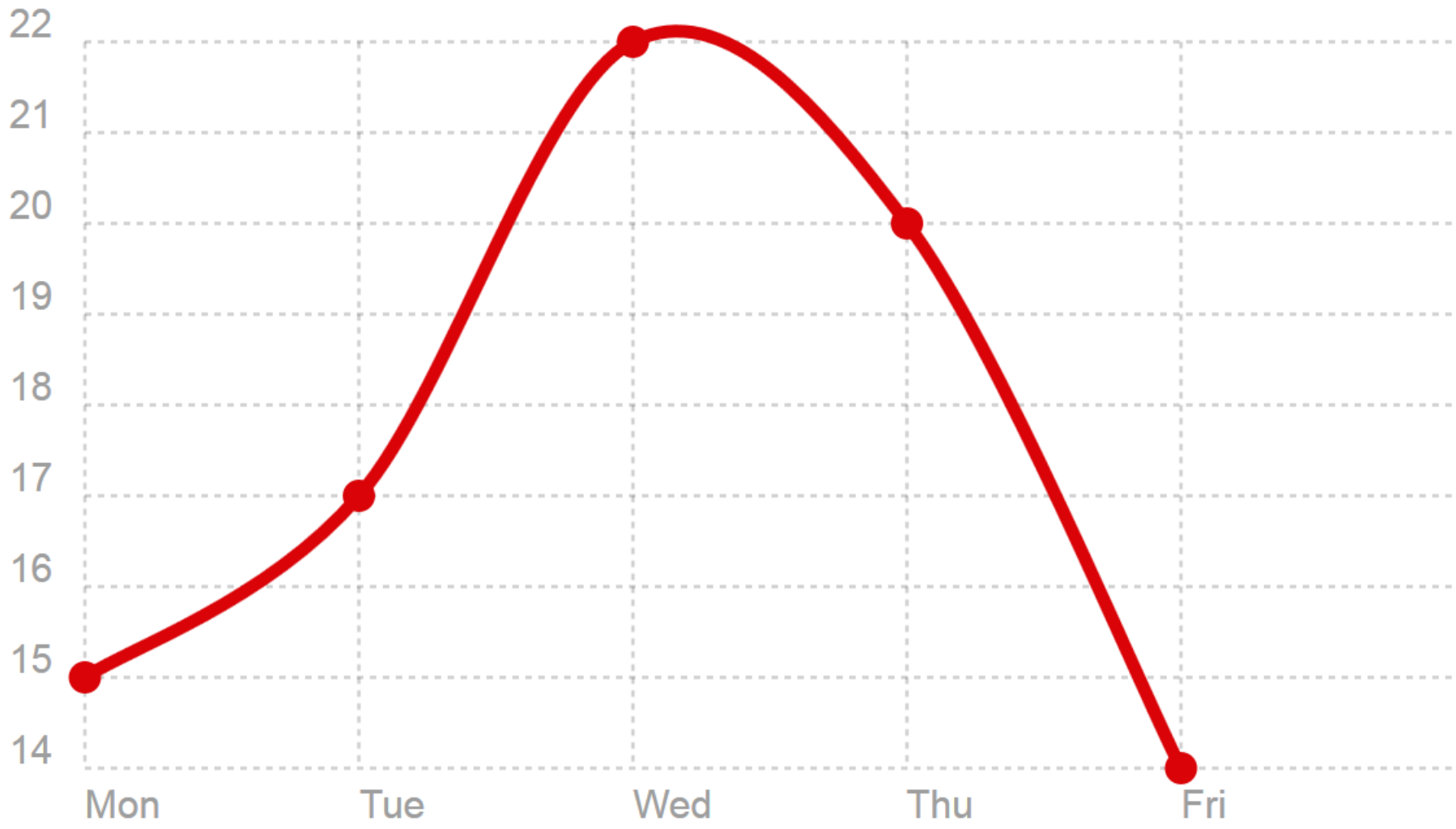
Containerklasse	Größenverhältnis
`ct-minor-seventh`	`9:16`
`ct-perfect-fourth`	`3:4`
`ct-golden-section`	`1:1.618`
  ```

Beispieldiagramm: Temperaturverlauf

```
<body>
<div class="mychart ct-minor-seventh"></div>
<script>
var data = {
  // Array containing any sort of labels
  labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],

  // Array(s) of temperature measures
  series: [
    [15, 17, 22, 20, 14]
  ]
};
new Chartist.Line('.mychart', data);
</script></body>
```

Temperaturverlauf: Ergebnis



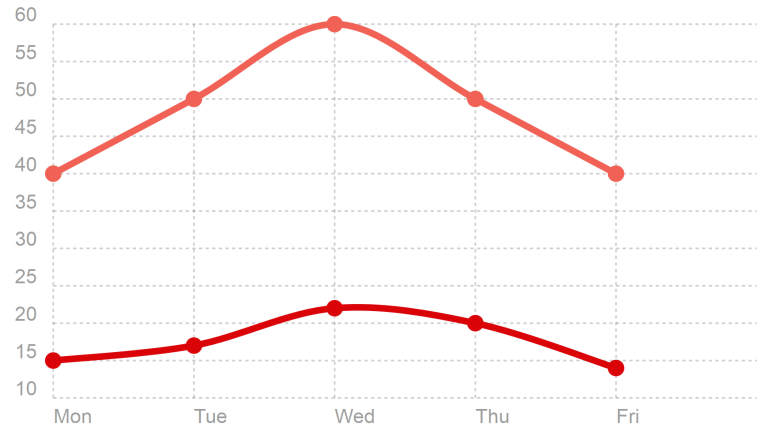
Mehrere Kurven in einem Diagramm

- Beispiel: Verlauf der Temperatur und Luftfeuchte

```
var data = {
  labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
  series: [
    [15, 17, 22, 20, 14],
    [40, 50, 60, 50, 40],
  ]
};
```

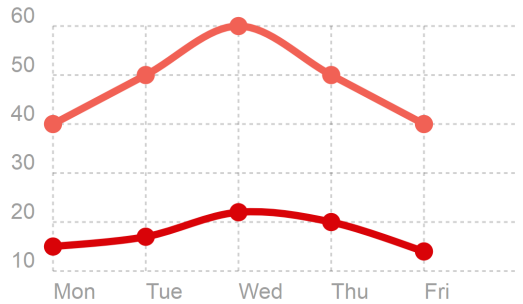
Serie B

Serie A



Optionen: Diagrammgröße festlegen

- ```
  <script>
  var options = {
    width: 320,
    height: 180
  };
  new Chartist.Line('.mychart', data, options);
  </script>
  ```



- Sollte einher gehen mit angepasster Containerklasse und -größe:

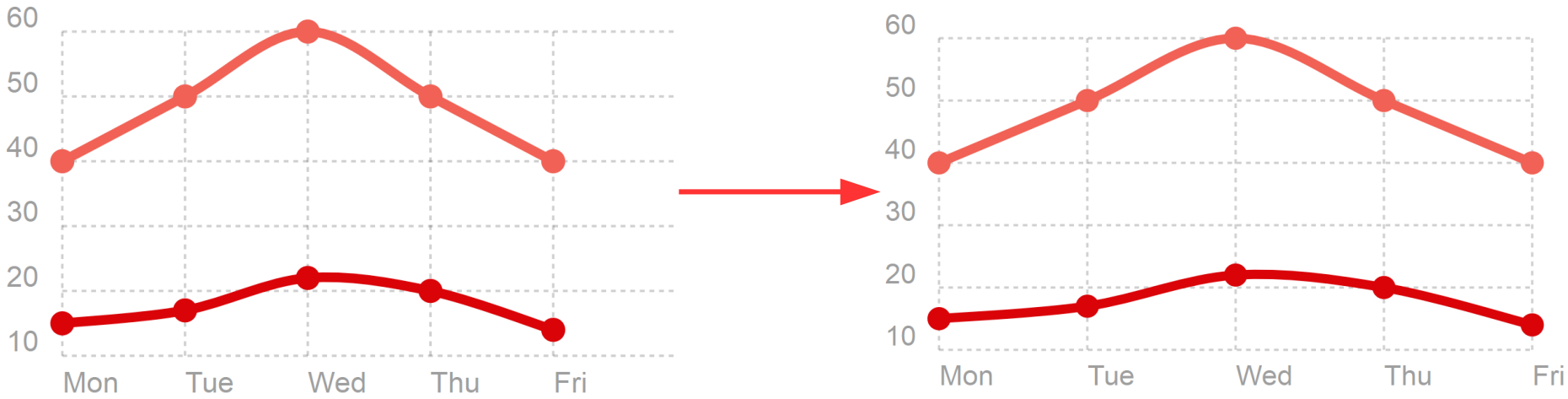
```
<head><style type="text/css">
   .mychart {
     width: 320px;
   }
</style></head>
```

- ```
  <body>
  <div class="mychart"></div>
  </body>
  ```

Keine Klasse **`ct-minor-seventh`** etc. verwenden hier

Optionen: Diagrammfüllbereich festlegen

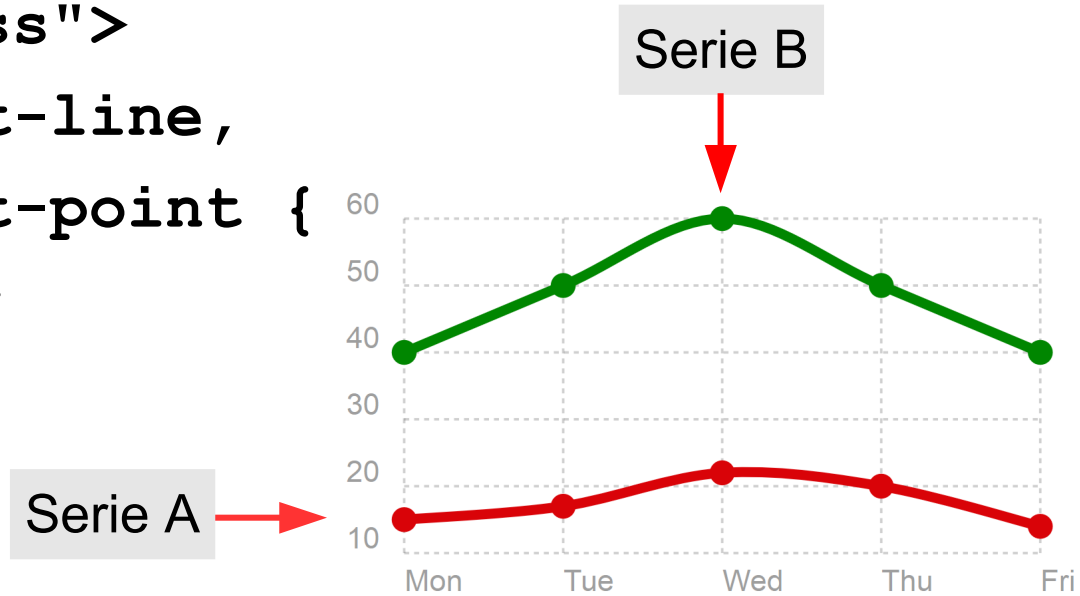
```
<script>
var options = {
   fullWidth: true,
};
</script>
```



Formatieren

- Geschieht mit Hilfe von CSS-Formatdefinitionen
  - Z. B. im Kopf der HTML-Datei, die die Diagramme enthält, einbinden
- Beispiel:

```
<head>
<style type="text/css">
     .ct-series-b .ct-line,
     .ct-series-b .ct-point {
       stroke: green;
     }
</style>
</head>
```

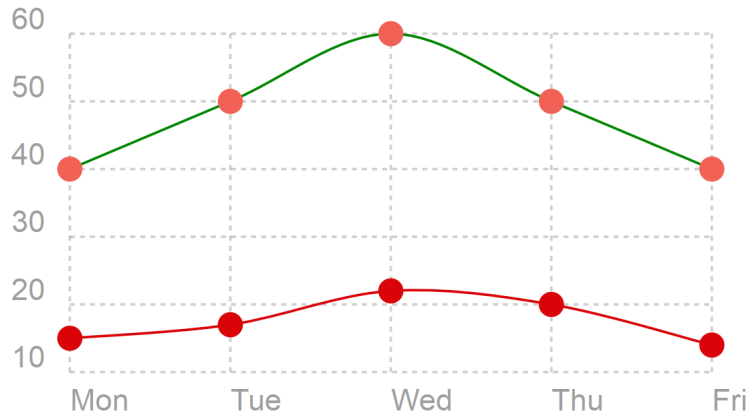


Linienformat

- Linienfarbe und -stärke der Graphen A und B:

```
.ct-series-a .ct-line {
    stroke-width: 1px;
}

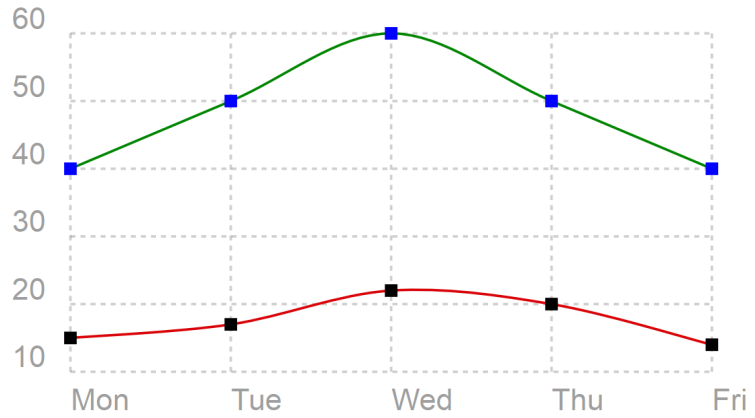
.ct-series-b .ct-line {
    stroke-width: 1px;
    stroke: green;
}
```



Punktformat

- Punktfarbe und -stärke der Graphen A und B:

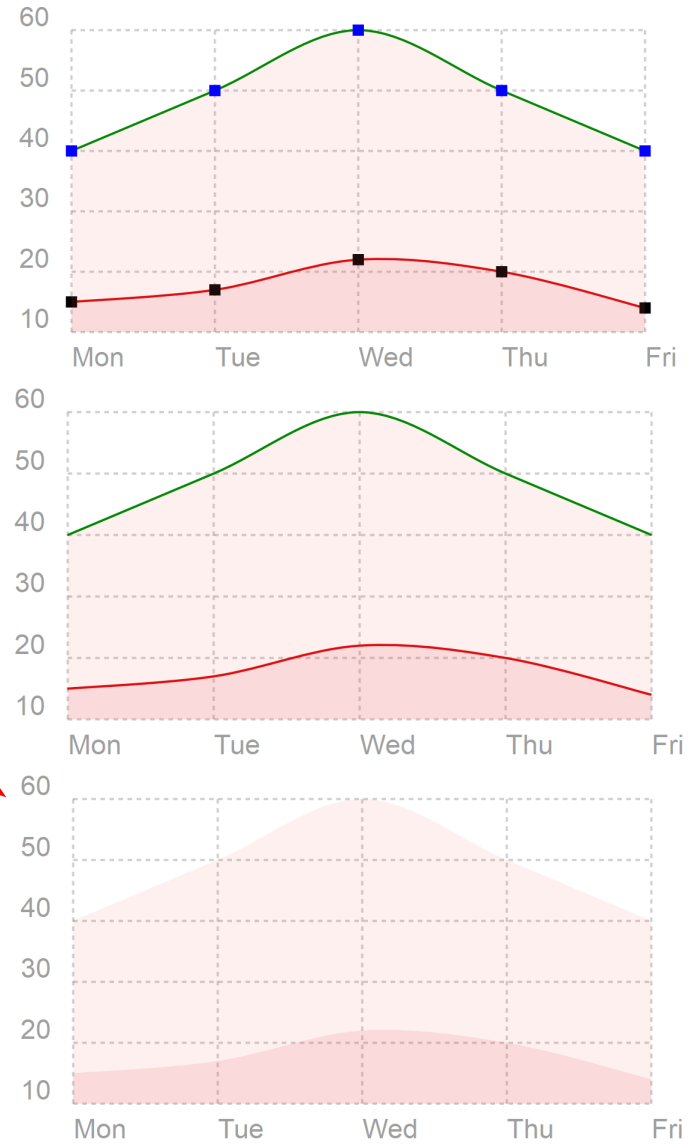
```
.ct-series-a .ct-point {
    stroke: black; // Punktfarbe
    stroke-width: 5px; // Punktgröße
    stroke-linecap: square; // Punktform
}
.ct-series-b .ct-point {
    stroke: blue;
    stroke-width: 5px;
    stroke-linecap: square;
}
```



Formatzuweisungen als Optionen

- Optionen:

```
var options = {
    showArea: true,
    showPoint: false,
    showLine: false,
};
```

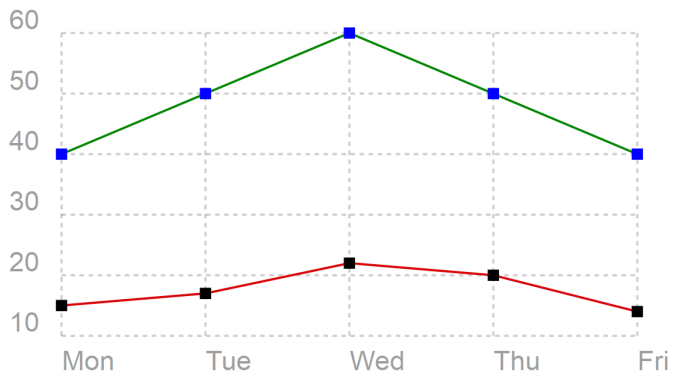


Interpolationsmethoden

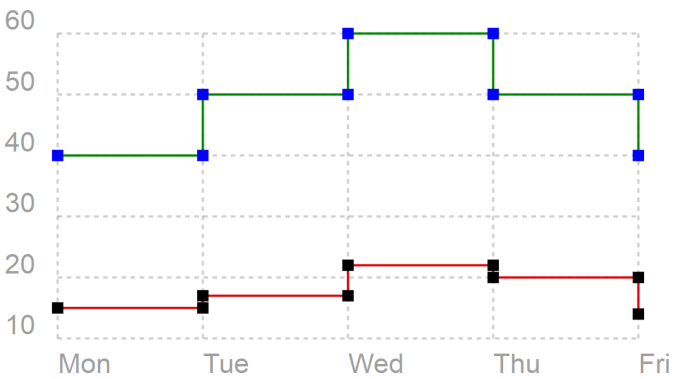
```
var options = {
    // Default: Polynomiale Kurven (Splines)
    // Interpolation durch Geraden, siehe (1)
    lineSmooth: Chartist.Interpolation.none(),
    // Rechteckfunktion, siehe (2)
    lineSmooth: Chartist.Interpolation.step(),
};
```

- siehe auch http://gionkunz.github.io/chartist-js/api-documentation.html#chartistinterpolation-method-none

(1)



(2)



Achsenbeschriftungen

- Optionen:

```
var options = {
    axisY: {
      // nur ganze Zahlen in der Legende
      onlyInteger: true,
      offset: 20 // Abstand zum linken Rand
    }
};
```

Responsivität: Media Queries (1)

- `var data = {`
  `labels: ['Monday', 'Tuesday', 'Wednesday',...]`
  `//...};`
- Syntax: `[[mediaQueryString, optionsObject],[more...]]`
- Beispiel aus der Chartist-Dokumentation:

```
var responsiveOptions = [
  ['screen and (min-width: 641px) and (max-width: 1024px)',
   {
    showPoint: false,
    axisX: {
     labelInterpolationFnc: function(value) {
      // Will return Mon, Tue, Wed etc. on medium screens
      return value.slice(0, 3);
     }
    }
   }], // Fortsetzung: siehe nächste Folie
```

Responsivität: Media Queries (2)

```
  ['screen and (max-width: 640px)', {
    showLine: false,
    axisX: {
      labelInterpolationFnc: function(value) {
        // Will return M, T, W etc. on small screens
        return value[0];
      }
    }
  }]
];

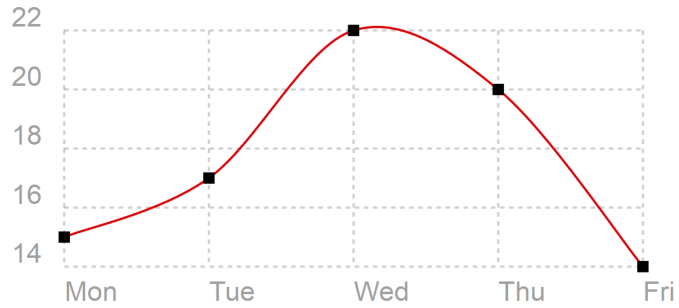
new Chartist.Line('.mychart', data, null,
                  responsiveOptions);
```

Optionen

Diagrammarten

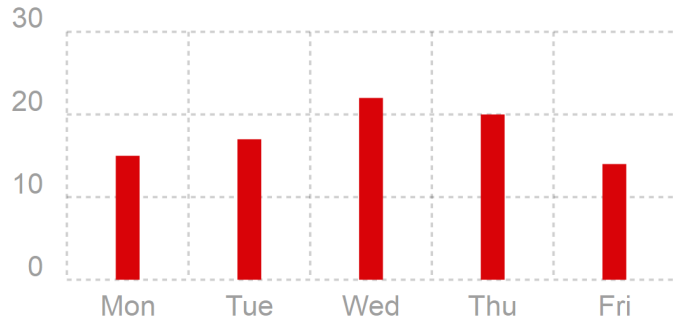
- Kurvendiagramm

```
new Chartist.Line('.mychart',
data, {fullWidth: true});
```



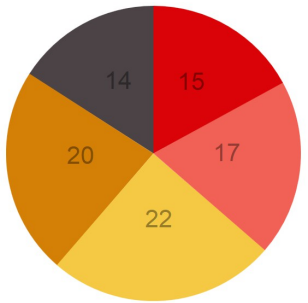
- Säulendiagramm

```
new Chartist.Bar('.mychart',
data);
```



- Tortendiagramm

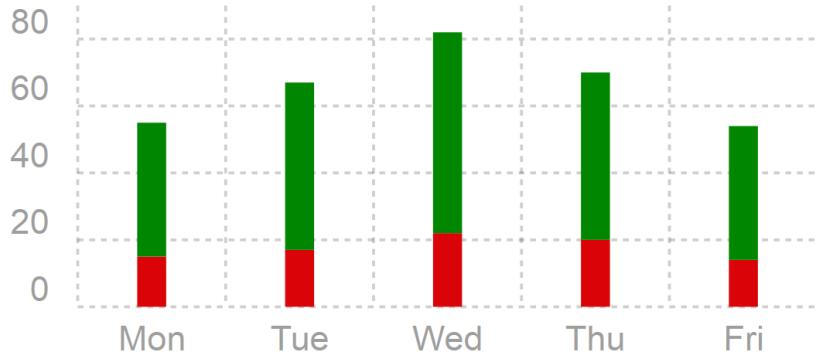
```
data = {
  // labels leer lassen
  series:
    [15, 17, 22, 20, 14]
};
new Chartist.Pie('.mychart', data);
```



Gestapeltes Säulendiagramm

- CSS:

```
.ct-series-b .ct-bar {
  stroke: green;
}
```



- JavaScript:

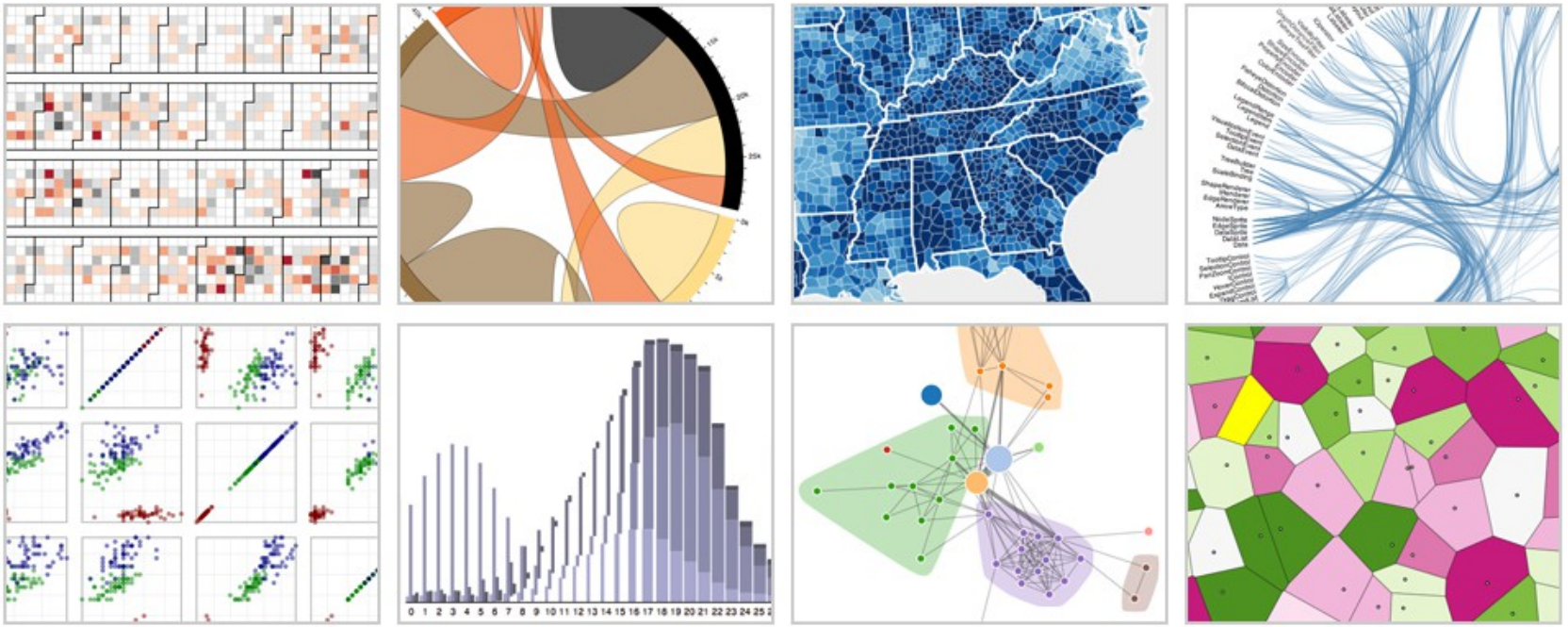
```
var data = {
  labels: ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
  series: [[15, 17, 22, 20, 14],
           [40, 50, 60, 50, 40]]
};
```

- `new Chartist.Bar('.mychart', data, {stackBars: true});`

D3

D3

- Abkürzung für Data-Driven Documents



- siehe: *D3: Data-Driven Documents*, M. Bostock, V. Ogievetsky, J. Heer, IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis), 2011, vis.stanford.edu/papers/d3, https://d3js.org/, https://github.com/d3/d3/wiki/Gallery

D3

- Umfangreiches Softwarepaket für Datenvisualisierung
  - https://d3js.org/
  - D3 API Reference: https://github.com/d3/d3/blob/master/API.md
- Wie bei Chartist werden Visualisierungen mittels JavaScript-Methoden als SVG-Grafiken in das DOM einer Webseite eingebettet
- Neben dem Einsatz vorgefertigter Diagramme ist der Gebrauch von SVG-Syntax und die Formatierung von Seitenelementen möglich
- Modulkonzept
  - Diagramme bestehen aus einzelnen "Bausteinen"

Formatierung existierender HTML-Elemente

- Im Kopfbereich JavaScript-Datei einbinden

```
<html>
  <head>
    <!-- sonstiges -->
    <script src="d3.min.js"></script>
  </head>
  <body>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
    <div id="aDiv">Div 1</div>
    <script>
      d3.selectAll("p").style("color", "red");
      d3.select("#aDiv").style("color", "blue");
    </script>
  </body>
</html>
```

Method Chaining

- Um den Code kompakt zu halten, unterstützt D3 die Verkettung von Methodenaufrufen ("method chaining")
  - Attributwerte müssen nicht durch einzelne Objektzugriffe gesetzt werden
  - Anstatt:

```
obj.attr(...);
...
obj.attr(...);
```

  - Aufrufe verkettbar:

```
obj.attr(...).attr(...).attr(...)....
```

  - Der Rückgabewert einer Methode ist das selektierte Objekt selbst

Animation

```
d3.selectAll("p")
    .transition() // Farbe über 2s nach rot
      .duration(2000)
      .style("color", "red")
    .transition() // Dann grün
      .duration(2000)
      .style("color", "green")
    .transition()
      .duration(2000)
      .style("color", "blue") // Schließlich blau
      .remove(); // P's von der Seite entfernen
```

Beispiel: Hinzufügen eines Rechtecks

- SVG-Syntax kann direkt in D3-Methodenaufrufe übernommen werden
- Im Körperbereich z. B. ein **`div`**-Element als Container vorsehen

```
<body>
  <div id="rectangle"></div>
  <script>
    var rectangle = d3.select("#rectangle").
                       append("svg:svg").
                       attr("width", 300).
                       attr("height", 300);
    rectangle.append("svg:rect").
                       attr("x", 50).
                       attr("y", 50).
                       attr("height", 100).
                       attr("width", 200).
                       attr("fill","green");
  </script>
</body>
```

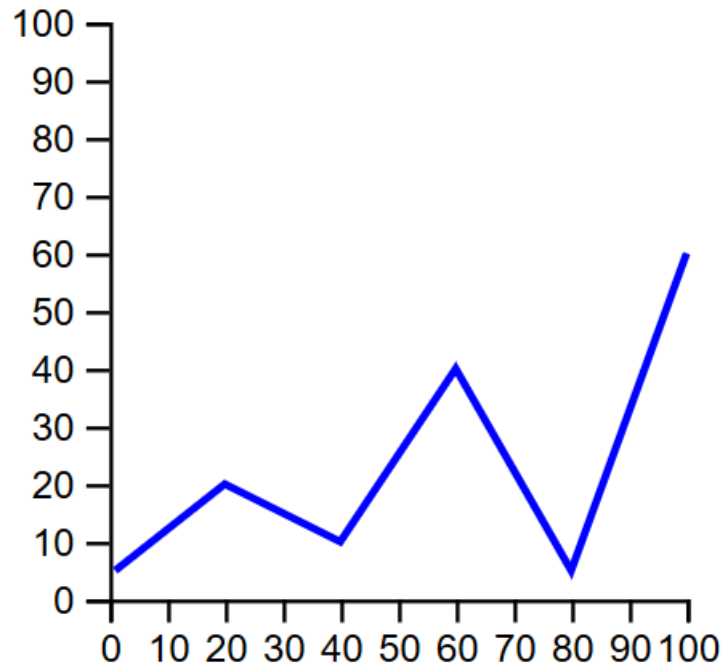
Beispiel: Handwagen mit D3

```
<body><div id="cart"></div><script>
  var cart = d3.select("#cart").append("svg:svg").
                attr("width", 500).attr("height", 500);
  cart.append("svg:rect").attr("x", 100).attr("y", 100).
       attr("height", 100).attr("width", 300).
       attr("fill","#ff950e").attr("fill-opacity",1).
       attr("stroke","#000000").attr("stroke-width","5").
       attr("opacity",1).attr("stroke-opacity",1);
  cart.append("svg:line").attr("x1", 400).attr("y1", 120).
       attr("x2", 600).attr("y2", 50).
       attr("stroke","black").attr("stroke-width","5");
  cart.append("svg:circle").attr("cx", 175).attr("cy", 175).
       attr("r", 50).attr("fill","red").
       attr("stroke","black").attr("stroke-width","3");
  cart.append("svg:circle").attr("cx", 325).attr("cy", 175).
       attr("r", 50).attr("fill","red").
       attr("stroke","black").attr("stroke-width","3");
   </script></body>
```

Liniendiagramm

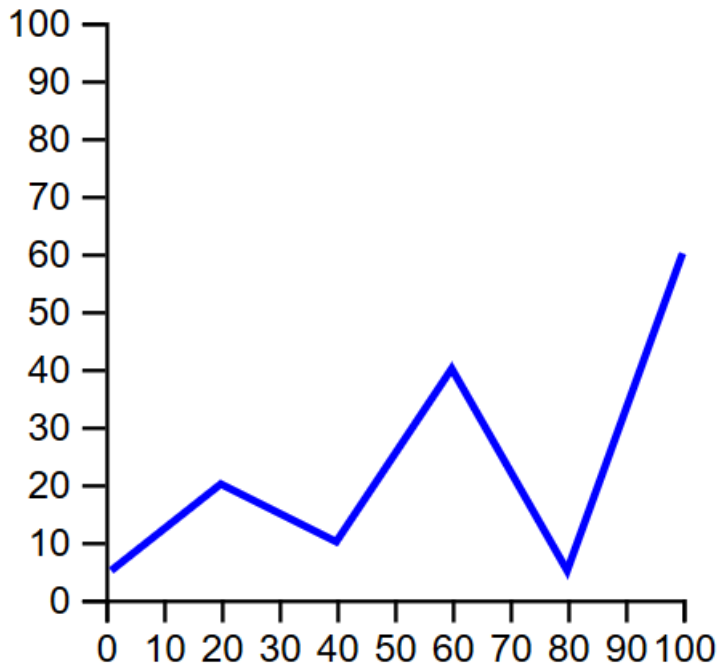
- Daten als Feld von Koordinaten:

```
var lineData = [{x:1,  y:5}, {x:20,  y:20},
                {x:40, y:10},{x:60,  y:40},
                {x:80, y:5}, {x:100, y:60}];
```



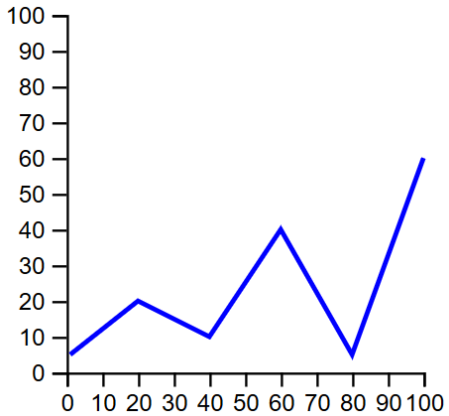
Darstellungsbereich: Skalierung

- Abbildung eines Eingabewertebereichs (domain) auf einen Ausgabewertebereich (range)
- Hier: Zuordnung eines Pixelbereichs in der SVG-Grafik (range) zu dem darzustellenden Wertebereich in x- und y-Richtung
-
```
var xRange =
    d3.scaleLinear().
    domain([0,100]).
    range([50, 200]);
```
-
```
var yRange =
    d3.scaleLinear().
    domain([0,100]).
    range([200, 50]);
```
  - y-Achse weist nach unten.
    Darum die Zuordnung 0 → 200



Achsen

- HTML: `<svg id="svg" width="500" height="500"></svg>`
- Achsen werden mit Hilfe eines "Generators" auf der Grundlage der definierten Skalierung erzeugt (**`axisBottom`**, **`axisLeft`**)



- x-Achse:

```
var axisb = d3.axisBottom(xRange);
d3.select("#svg").append("g")
    .attr("transform", "translate(0,200)").call(axisb);
```

- y-Achse

```
var axisl = d3.axisLeft(yRange);
d3.select("#svg").append("g")
    .attr("transform", "translate(50,0)").call(axisl);
```

- Für die Achsen wird jeweils eine SVG-Gruppe in der Grafik angelegt (**`append("g")`**) und an die gewünschte Position verschoben (**`"transform"`**)

Kurve

- `line`: Generator für Kurven
  - Kurvenpunkte werden als Feld übergeben
  - Die als Parameter gelieferte Zugriffsfunktion wird automatisch für jedes Feldelement aufgerufen
- ```
  var line = d3.line()
      .x(function(d) { return xRange(d.x); })
      .y(function(d) { return yRange(d.y); });
  ```
- ```
  d3.select("#svg").append('svg:path')
    .attr('d', line(lineData))
    .attr('stroke', 'blue')
    .attr('stroke-width', 2)
    .attr('fill', 'none');
  ```

JSON

JSON

- **J**ava**S**cript **O**bject **N**otation
- Datenaustauschformat für JavaScript
  - Insbesondere für den Transfer von Daten zwischen Server und Client
  - Script einer Webseite liest im Hintergrund Daten von einem Webdienst und fügt diese dynamisch in die Seite ein (z. B. Anfrageergebnisse)
- Übertragung und Speicherung strukturierter Daten in Form von JavaScript-Objekten
- JSON beruht daher auf der JavaScript-Syntax

Beispiel: Personendaten

- Datei **`personendaten.json`** mit folgenden Inhalt:

```
{
   "vorname":"Markus",
   "nachname":"Mustermann",
   "alter":25
}
```

- Daten werden als Objektliteral in einer Textdatei gespeichert
  - Vorteil: Einfach les- und bearbeitbar
  - Endung beliebig, üblicherweise **`.json`**
- Im Gegensatz zu JavaScript müssen Eigenschaftsnamen bei JSON in Anführungszeichen gesetzt werden
- Reine Eigenschafts-/Wert-Paare, keine Methoden möglich

JSON-Dateien einlesen

- Erfolgt mit Hilfe von AJAX-Technologien
  - Ursprünglich für: **A**synchronous **J**avaScript **A**nd **X**ML
  - XML: Format für die Definition von Auszeichnungssprachen
  - JSON: Reines Datenaustauschformat
  - Im Vergleich zu XML ist die JSON-Notation einfacher und darum kompakter
- Laden der Datendateien erfolgt mit Hilfe des `XMLHttpRequest`-Objekts
  - Ermöglicht das asynchrone Laden von Daten aus einer Webseite heraus, ohne dass hierfür ein Link erforderlich ist
- Interpretation der Daten durch `JSON.parse()`-Methode

JSON-Dateien einlesen: Beispiel

```
<script>
var person;
var xmlhttp = new XMLHttpRequest();
var url = "personendaten.json";

xmlhttp.onreadystatechange = function() {

  if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {

    person = JSON.parse(xmlhttp.responseText);

    writeln("Vorname: " + person.vorname);
    writeln("Nachname: " + person.nachname);
    writeln("Alter: " + person.alter);
  }
};
xmlhttp.open("GET", url, true); // true: asynchr. transfer
xmlhttp.send();
</script>
```

XMLHttpRequest-Objekt: Eigenschaften

- **`onreadystatechange`**: Behandler für **`readyState`**-Ereignisse
  - Wird automatisch bei Änderungen des **`readyState`** aufgerufen
- **`readyState`**: Speichert den Status des XMLHTTPRequest.
  - Mögliche Werte:
    - 0: Anfrage nicht initialisiert (request not initialized)
    - 1: Verbindung aufgebaut (server connection established)
    - 2: Anfrage erhalten (request received)
    - 3: Anfrage in Bearbeitung (processing request)
    - 4: Bearbeitung fertig (request finished and response is ready)
- **`status`**
  - Mögliche Werte z. B.: 200: "OK", 404: Page not found
- **`responseText`**: Enthält eingelesene Daten als String

Felder einlesen

- Neben Objekten können auch Felder, Zeichenketten, Zahlen, Wahrheitswerte und das **`null`**-Literal in JSON-Dateien abgelegt werden
- Sollen mehrere Objekte bzw. Literale abgelegt werden, so geschieht dies in verschachtelter Form
  - Häufig in Form von Kombinationen unterschiedlicher Datentypen, z. B. von Objekten und Feldern
- Beispiel: Chartist-Diagramm
  - Temperaturverlauf als Feld in JSON-Datei
  - **`temperaturverlauf.json`** mit folgendem Inhalt:

    ```
    [[15, 17, 22, 20, 14]]
    ```

  - Chartist arbeitet mit mehrdimensionalen Feldern

Diagramm mit externen Temperaturdaten

```
var url = "temperaturverlauf.json";
var data = {};
xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
    data.labels = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'];
    data.series = JSON.parse(xmlhttp.responseText);
    new Chartist.Line('.mychart', data);
  }
};
xmlhttp.open("GET", url, true);
xmlhttp.send();
```

- Body mit: `<div class="mychart ct-perfect-fourth"></div>`

Objekte mit Feldern

- Beispiel: Alle Daten für Chartist-Diagramm in einem Objektliteral
  - Datei **`labelsAndTemp.json`** mit folgendem Inhalt:

```
{
  "labels":["Mo", "Di", "Mi", "Do", "Fr"],
  "series":[[15, 17, 22, 20, 14]]
}
```

- Nur doppelte Anführungszeichen sind hier erlaubt, keine einfachen

Chartist-Diagramm mit JSON

```
var url = "labelsAndTemp.json";

xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState === 4 &&
      xmlhttp.status === 200) {

    var data = JSON.parse(xmlhttp.responseText);

    new Chartist.Line('.mychart', data);

  }
};

xmlhttp.open("GET", url, true);
xmlhttp.send();
```

Feld von Objektliteralen

- Neben Objekten, die Felder enthalten, sind auch Felder von Objekten möglich
- Dateiname: **`personendaten2.json`**

```
[
   {
      "vorname":"Markus",
      "nachname":"Mustermann",
      "alter":25
   },
   {
      "vorname":"Marianne",
      "nachname":"Musterfrau",
      "alter":30
   }
]
```

Personendaten einlesen

```
var url = "personendaten2.json";

xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState === 4 &&
      xmlhttp.status === 200) {
    var person = JSON.parse(xmlhttp.responseText);
    for (var i in person) {
      writeln("Vorname: " + person[i].vorname);
      writeln("Nachname: " + person[i].nachname);
      writeln("Alter: " + person[i].alter);
    }
  }
};
```

XML

HTML

- Auszeichnungssprachen wie HTML dienen dazu Dokumente logisch zu gliedern
  - Z. B. in Absätze, Aufzählungen, Überschriften, etc.
- Die logischen Bestandteile definieren die Art der Ausgabe bzw. Darstellung auf dem Zielmedium
- Sie liefern keine Information zur möglicherweise vorhandenen Bedeutung bzw. Semantik der Daten, obwohl diese ursprünglich den Daten beigefügt gewesen sein kann
  - HTML-Seite wurde möglicherweise dynamisch mit Daten aus einer Datenbank erzeugt
  - Innerhalb dieser Datenbank besaßen die Daten eine Semantik, die jetzt verloren ist

Beispiel: Wetterdienstanfrage

- Serveranfrage kann z. B. zu folgender Antwort führen:

```
<HTML>
<HEAD><TITLE>Wetterbericht</TITLE></HEAD>
<BODY>
  <P> Ulm, Messpunkt Böfingen </P>
  <P> Mo, der 11.11.2011 um 11:11 </P>
  <P> Sonnig </P>
  <P> 10 Grad Celsius </P>
  <P> Wind SO 10 km/h </P>
  <P> 1033 mbar </P>
  <P> 90% </P>
  <P> Sicht: 19 km </P>
</BODY>
</HTML>
```

- Ursprünglich vorhandene Semantik ist verloren

XML

- Verlust der semantischen Information erschwert die maschinelle Weiterverarbeitung der Daten
- Mit Hilfe von XML (eXtensible Markup Language) lassen sich demgegenüber eigene Auszeichnungssprachen definieren, die es gestatten
  - Daten entsprechend ihrer Semantik bzw. Bedeutung zu strukturieren,
  - unterschiedlichen Bedeutungsklassen Darstellungsklassen zuzuordnen, um die Daten für die Ausgabe auf einem Medium aufzubereiten

Beispiel: Wetterdaten mit XML

```
<?xml version="1.0" encoding="UTF-8"?>
<Wetter>
  <Ort Messpunkt="Böfingen">Ulm</Ort>
  <Datum>
    <Tag>11</Tag>
    <Monat>11</Monat>
    <Jahr>2011</Jahr>
    <Zeit>11:11</Zeit>
  </Datum>
  <Himmel>Sonnig</Himmel>
  <Temperatur Einheit="Celsius">10</Temperatur>
  <Wind Einheit="km/h">SO 10</Wind>
  <Luftdruck Einheit="mbar">1033</Luftdruck>
  <Luftfeuchtigkeit>90%</Luftfeuchtigkeit>
  <Sicht Einheit="km">19</Sicht>
</Wetter>
```

HTML versus XML versus JSON

- HTML → Darstellung
  - Präsentation von Daten steht im Vordergrund
  - Paragraf-Tags besitzen z. B. keinen Bezug zur Information
- XML → Datenspeicherung und -archivierung
  - Semantik (Bedeutung) der Daten steht im Vordergrund
- JSON → Datenaustausch zwischen Programmen
  - Einfacher Datenaustausch zwischen Programmen zur Laufzeit steht im Vordergrund
  - Datenrepräsentation auf der Grundlage einer eingeschränkten Zahl von Datentypen

XML: Typische Anwendungsfälle

- Textuell lesbare Datenspeicherformate
  - Z. B. Scalable Vector Graphics (SVG), MathML, Open Document Format (ODF)
- Setup-Dateien für Programme
- Erweiterung der HTML-Syntax mit Hilfe von Webkomponenten
- Definition grafischer Bedienoberflächen

XML: Syntax

- XML verlangt eine wesentlich strengere syntaktische Beschreibung als HTML
- Zu jedem öffnenden Tag muss beispielsweise auch ein zugehöriges schließendes Tag vorhanden sein
- Browser interpretieren auch "sloppy HTML"
  - Z. B. weggelassenes oder vergessenes schließendes Tag **`</p>`**
- Bei XML wird üblicherweise eine rigide Syntaxprüfung durchgeführt, entsprechend zu Programmiersprachen
  - Syntaktische Fehler führen zu Fehlermeldungen des Parsers
- XML ist im Gegensatz zu HTML case-sensitiv

XML

- Daten werden durch definierte **Elemente** gefasst und gegliedert
  - Element entspricht einem durch eine Markierung gekennzeichneten Bereich
  - Die Markierung erfolgt mit Hilfe von **Tags**
  - Der Name des Elements wird im öffnenden und schließenden Tag notiert
- Die Klasse eines Elements bzw. dessen Semantik geht aus den umschließenden Tags hervor
- Beispiel:

```
<Textelement>
  Dieser Text gehört zur Klasse Textelement
</Textelement>
```

Dokumentbaum

- Elemente werden auch als **Knoten** bezeichnet
- Der Dokumentinhalt besitzt eine hierarchische Baumstruktur mit Wurzelknoten (*Wurzelelement*, *Root Element*)
- Das Wurzelelement enthält den kompletten Dokumenteninhalt als Unterbaum
- Im Gegensatz zu HTML ist das Wurzelelement nicht vordefiniert
- Beispiel: **`<Wetter>...</Wetter>`**
- Der Dokumentbaum liegt für den Zugriff auf die Knoten üblicherweise auch als Datenstruktur vor
- Der Zugriff erfolgt mit Hilfe des standardisierten **Document Object Models (DOM)**

Attribute und leere Elemente

- Elemente können mit Hilfe von Attributen erweitert werden
- Beispiel:
  - `<Temperatur Einheit="Celsius">10</Temperatur>`
- Neben den Standardelementen gibt es auch **leere Elemente** (engl.: *empty element*)
  - Werden auch als *terminale* oder *standalone* Tags bezeichnet
  - Leere Elemente besitzen kein explizit schließendes Tag
  - Der Schrägstrich ist Teil des öffnenden Tags
- Der Einsatz leerer Elemente ist dann sinnvoll, wenn die Bedeutung eines Elements aus seinen Attributen folgt
- Beispiel:
  - `<Kunde Nachname="Brause" Vorname="Klaus" />`

Auszeichnungssprachen definieren

- Tags bzw. Markierungen sind frei definierbar, d. h. die Syntax der Auszeichnungssprache lässt sich an den Anwendungsfall anpassen
  - Folglich ist ein "Universum" an Auszeichnungssprachen möglich
- Es sind deshalb syntaktische Regeln erforderlich, die die Grammatik dieser Auszeichnungssprachen vorschreiben, d. h. wie diese aufgebaut sind
- Die syntaktischen Regeln werden in einer Metasprache formuliert und in **Dokumenttypdefinitionen** (engl.: Document Type Definitions: DTD) niedergeschrieben

XML: Korrektheit

- Sofern zu einer XML-Datei eine Dokumenttypdefinition vorliegt, kann ihre syntaktische Korrektheit überprüft werden
- Daneben ist auch das Formulieren von neuen Dokumenttypen ohne DTD möglich
- Dokumente dieses Typs müssen dann zumindest **wohlgeformt** *(well-formed*) sein, d. h. dem allgemeinen Aufbau einer jeden XML-Datei folgen
- Auf dieser Grundlage sind allgemeine syntaktische Tests möglich
  - Z. B. ob zu einem öffnenden Tag ein schließendes Tag vorhanden ist
  - Es ist aber beispielsweise keine Überprüfung möglich, ob ein gelesenes Element tatsächlich Teil der Auszeichnungssprache ist

XML-Datei: Allgemeiner Aufbau

- Jedes XML-Dokument beginnt mit einem **Prolog**
- Beispiel:

```
<?xml version="1.0" encoding="utf-8"?>
```

  - Der Prolog enthält die XML-Versionsnummer und i. d. R. die Zeichencodebasis
- Der Inhalt des XML-Dokuments besteht aus einer Reihe von hierarchisch gegliederten Elementen
  - D. h. Elemente enthalten weitere Elemente
- Zu jedem öffnenden Tag gehört ein schließendes, es sei denn, es handelt sich um ein leeres Element
- Elemente können Attribute besitzen
- Es gibt ein Wurzelelement

XML-Datei: Strenger syntaktischer Test

- Um ein XML-Dokument streng hinsichtlich seines korrekten Aufbaus validieren zu können, ist die Angabe einer Beschreibung des Dokumenttyps, d. h. eine DTD, erforderlich
- Es kann dann nicht nur die Wohlgeformtheit, sondern ebenfalls die **Gültigkeit** nachgewiesen werden
- Beispiel:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Kundendaten SYSTEM "Kundendaten.dtd">
```

- Schlüsselwörter:
  - **`SYSTEM`**: XML-Typ, der nur lokal verwendet wird
  - **`PUBLIC`**: Allgemein bekannter und verwendeter XML-Typ

Beispiel: XML-Datei mit Kundendaten

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Kundendaten SYSTEM "Kundendaten.dtd">

<Kundendaten>
```

Wurzelelement

```
  <Kunde>
    <Nachname>Mustermann</Nachname>
    <Vorname>Markus</Vorname>
    <Strasse>Musterstr. 12</Strasse>
    <PLZ>78101</PLZ>
    <Ort>Musterstadt</Ort>
    <Alter>32</Alter>
  </Kunde>

  <Kunde>
    ...
  </Kunde>

</Kundendaten>
```

DTD zu Kundendaten

```
<!ELEMENT Kundendaten (Kunde*)>
<!ELEMENT Kunde (Nachname, Vorname,
          (Strasse|Postfach), PLZ, Ort, Alter?)>
<!ELEMENT Nachname (#PCDATA)>
<!ELEMENT Vorname (#PCDATA)>
<!ELEMENT Strasse (#PCDATA)>
<!ELEMENT Postfach (#PCDATA)>
<!ELEMENT PLZ (#PCDATA)>
<!ELEMENT Ort (#PCDATA)>
<!ELEMENT Alter (#PCDATA)>
<!ATTLIST PLZ Land (D|A|F|I|GB|CH) "D">
```

DTD zu den Wetterdaten

```
<!ELEMENT Wetter (Ort, Datum, Himmel, Temperatur, Wind,
                  Luftdruck, Luftfeuchtigkeit, Sicht)>
<!ELEMENT Ort (#PCDATA)>
<!ELEMENT Datum (Tag, Monat, Jahr, Zeit)>
<!ELEMENT Himmel (#PCDATA)>
<!ELEMENT Temperatur (#PCDATA)> <!ELEMENT Wind (#PCDATA)>
<!ELEMENT Luftdruck (#PCDATA)>
<!ELEMENT Luftfeuchtigkeit (#PCDATA)>
<!ELEMENT Sicht (#PCDATA)>
<!ELEMENT Tag (#PCDATA)>
<!ELEMENT Monat (#PCDATA)> <!ELEMENT Jahr (#PCDATA)>
<!ELEMENT Zeit (#PCDATA)>
<!ATTLIST Ort Messpunkt CDATA #REQUIRED>
<!ATTLIST Temperatur Einheit
          (Celsius|Fahrenheit|Kelvin) "Celsius">
<!ATTLIST Wind Einheit CDATA "km/h"> <!-- CDATA wg. / -->
<!ATTLIST Luftdruck Einheit (mbar|atm) "mbar">
<!ATTLIST Sicht Einheit (m|km|sm) "km">
```

PCDATA und CDATA

- **PCDATA** (*Parsed Character Data*)
  - Normal interpretierter Text innerhalb z. B. eines Elements
- **CDATA**
  - Es gibt **CDATA**-Abschnitte und den **CDATA**-Datentyp
    - Texte in CDATA-Abschnitten werden vom XML-Parser nicht interpretiert
    - **CDATA**-Abschnitte beginnen mit der Zeichenfolge **<![CDATA[** und enden mit der Zeichenfolge **]]>**
    - Dazwischen kann beliebiger Text stehen
    - Zeichen, die sonst als Metazeichen interpretiert werden, können als normale Zeichen verwendet werden (z. B. <, >, =)
  - Als Datentyp dient **CDATA** für Attributdeklarationen

Elementtyp-Deklaration

- Definiert ein Element und seinen möglichen Inhalt
- Syntax: **<!ELEMENT *name inhalt* >**
- *name*: Für die Tags zu verwendender Bezeichner
- *inhalt*: Mögliche Inhalte eines Elements
  - **`#PCDATA`**
  - **element_name**: Ein bestimmtes anderes Element
  - **(expr)**: Ein Ausdruck, der weitere Klammern enthalten darf
    - Wird z. B. eingesetzt, um eine Liste von Elementen festzulegen, die von diesem Element umschlossen werden
  - **`ANY`**: Irgendein Inhalt, der aber interpretiert wird
  - **`EMPTY`**: Definiert ein leeres Element

Ausdrücke

- **(expr)**: Klammern umhüllen einen möglicherweise komplexen Ausdruck, innerhalb dessen es weitere Klammern geben kann
- **(expr, expr, ...)**: Das Komma trennt eine Reihe von Ausdrücken, die in der angegebenen Reihenfolge erwartet werden
- **(expr | expr)**: Der linke oder rechte Ausdruck ist erlaubt, aber nicht beide gleichzeitig
- **expr?**: Bezeichnet einen optionalen Ausdruck, der einmal vorkommen darf
- **expr***: Der Ausdruck darf beliebig oft vorkommen und auch nicht
- **expr+**: Der Ausdruck darf beliebig oft, muss aber mindestens einmal vorkommen

Attribute

- Attribute werden in einer **Attributlistendeklaration** (*Attribute List Declaration*) definiert
- Der Name der Deklaration entspricht dem Namen des Elements
- **<!ATTLIST element_name attribut_name attribut_typ attribut_vorgabe ... >**
- Beispiel:
  - `<!ATTLIST PLZ Land (D|A|F|I|GB|CH) "D">`

Attributtypen

- **`CDATA`**: Beliebige Zeichenkette
- **Aufzählungstyp** (Enumerated Attribute Type)
  - Das Attribut kann einen der Werte aus der angegebenen Liste annehmen
- **Tokenized Type**
  - **`NMTOKEN`** oder **`NMTOKENS`**: Einer oder mehrere durch Leerzeichen getrennte Namen können zugewiesen werden
    - Ein Name enthält keine Leerzeichen und beginnt mit einem Buchstaben, einer Ziffer oder dem Zeichen . (Punkt), - (Bindestrich), _ (Unterstrich) oder : (Doppelpunkt)
  - Andere Tokenized Types: ID, IDREFS, ENTITY, ENTITIES
    - ID: Attributwert ist ein eindeutiger Identifikationswert
    - siehe auch https://wiki.selfhtml.org/wiki/XML/DTD/Attribute_und_Wertzuweisungen

Attributvorgaben

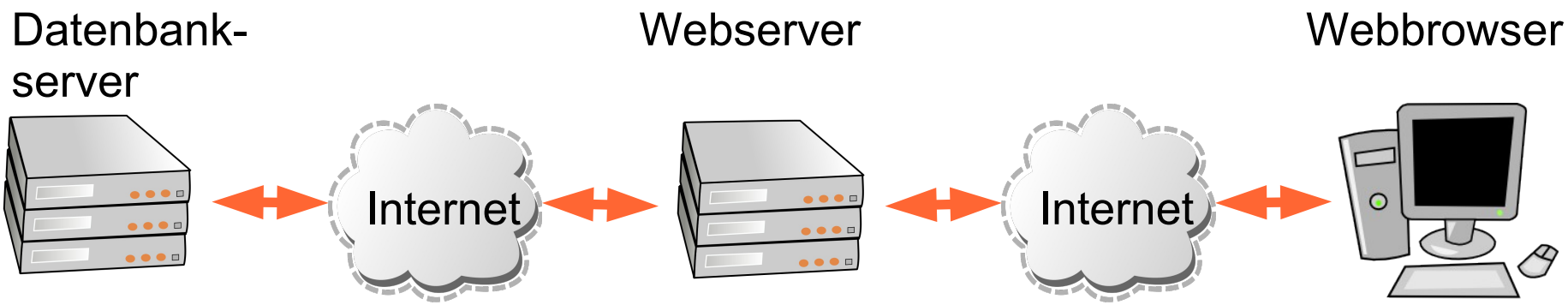
- *Attribute Defaults*
- **`#REQUIRED`**: Das Attribut muss im Element angegeben werden
- **`#IMPLIED`**: Das Attribut kann angegeben werden, muss aber nicht
- **"Wert"**: Ein Vorgabewert, der automatisch gewählt wird, wenn kein Attributwert angegeben wurde
- **`#FIXED`** "**Wert**": Das Attribut hat immer diesen Wert, es kann kein anderer Wert zugewiesen werden
  - Sinnvoll, wenn andere Attributwerte als "Wert" erst in der Zukunft unterstützt aber bereits jetzt etabliert werden sollen

XML: Verarbeitung in JavaScript

- Analog zu HTML werden XML-Daten auf eine Objektstruktur gemäß des DOM abgebildet
  - Folglich kann auf die Daten mit den üblichen DOM-Methoden zugegriffen werden
  - Das Einlesen der Daten geschieht mit der gleichen AJAX-Technologie, wie bei JSON-Dateien
    - AJAX ursprünglich Akronym für *Asynchronous JavaScript And XML*
- XML-Dateien ergänzen die durch JSON und Datenbanken gegebenen Möglichkeiten der Datenhaltung
  - Der Transfer von Daten aus externen Datenbanken geschieht häufig basierend auf JSON- oder XML-Syntax

AJAX

- Webseiten verarbeiten im Hintergrund extern eingelesene Daten
  - Der Inhalt der Seite ändert sich, ohne dass diese neu geladen werden muss
  - Der/die Benutzer/-in kann weiter mit der Anwendung interagieren, während die Daten vom Server geladen werden

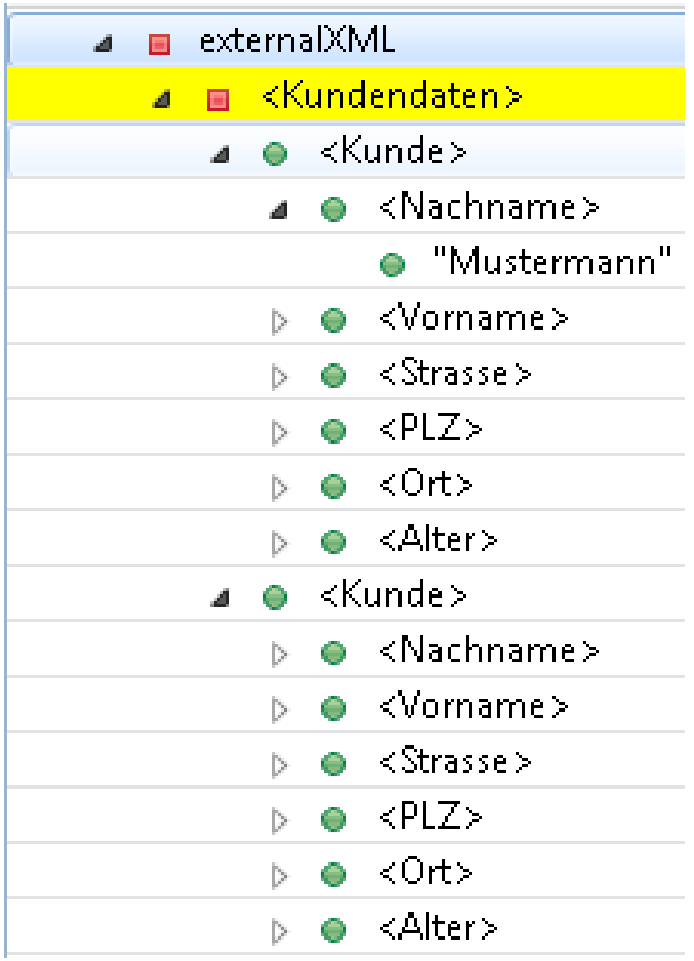


MVC

- MVC-Designmuster
- Model-View-Controller („Modell-Präsentation-Steuerung“)
  - Eine Anwendung besteht aus dem Datenmodell (Modell), dem Präsentationsteil (View) und der Programmsteuerung (Controller), die Model und View miteinander verbindet
- Im Webbereich
  - Die Datenhaltung erfolgt auf dem Server, der Browser bzw. die Webseite ist nur die Präsentationsschicht
  - Die Programmsteuerung wird zumeist mit JavaScript umgesetzt

Beispiel: Datenbasis für Kundendaten

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Kundendaten SYSTEM "Kundendaten.dtd">
<Kundendaten> Wurzelelement
  <Kunde>
    <Nachname>Mustermann</Nachname>
    <Vorname>Markus</Vorname>
    <Strasse>Musterstr. 12</Strasse>
    <PLZ>78101</PLZ>
    <Ort>Musterstadt</Ort>
    <Alter>32</Alter>
  </Kunde>
  <Kunde>
    ...
  </Kunde>
</Kundendaten>
```



Datentransfer vorbereiten

```
<div id = "output"></div>
<script>
  var xmlhttp = new XMLHttpRequest();
  // Ereignisbehandler fuer die Auswertung der Daten
  xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState === 4 &&
      xmlhttp.status === 200) {
    // Get the type of the response
    var type =
      xmlhttp.getResponseHeader("Content-Type");
    // Check if type is XML and not JSON, HTML,...
    if (type.indexOf("xml") !== -1 &&
                           xmlhttp.responseXML) {
      var externalXML = xmlhttp.responseXML;
      // weiter auf der nächsten Folie
```

Auswertung der Daten (1)

```
// Die einzelnen XML-Elemente über ihren Namen auslesen
var kunden = externalXML.getElementsByTagName("Kunde");
for (var i = 0; i < kunden.length; i++) {
  var lName =
  kunden[i].getElementsByTagName("Nachname")[0].textContent;
  var fName =
  kunden[i].getElementsByTagName("Vorname")[0].textContent;
  var str =
  kunden[i].getElementsByTagName("Strasse")[0].textContent;
  var pcode =
  kunden[i].getElementsByTagName("PLZ")[0].textContent;
  var loc =
  kunden[i].getElementsByTagName("Ort")[0].textContent;
  var age =
  kunden[i].getElementsByTagName("Alter")[0].textContent;
```

Auswertung der Daten (2)

```
// Zugriff auf Ausgabeelement herstellen
var output = document.getElementById("output"); // div

// Eingelesene Daten in Ausgabeelement schreiben
output.innerHTML +=
  fName + " " + lName + ", " + age + ": " +
  str + ", " + pcode + " " + loc + "<br>";

}}}}
</script>
```

- Ausgabe:

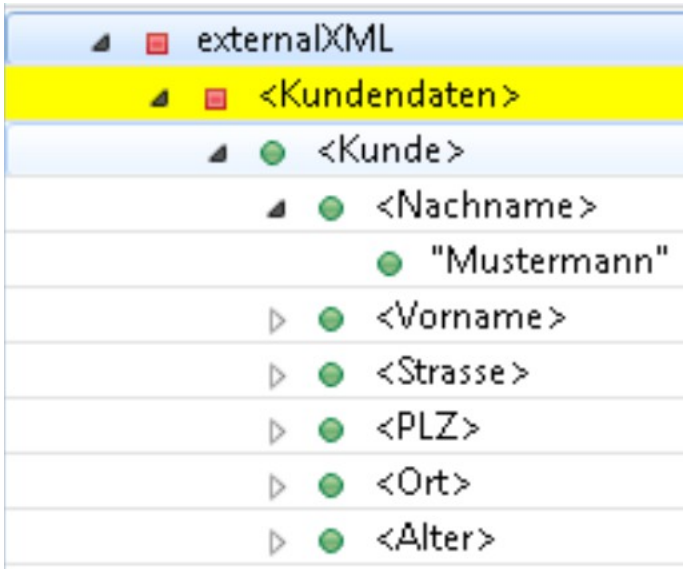
```
Markus Mustermann, 32: Musterstr. 12, 78101 Musterstadt
Marianne Musterfrau, 35: Musterweg. 14, 89102 Musterweiler
```

Datentransfer durchführen

- `xmlhttp.open("GET", "Kundendaten.xml", true);`
  `xmlhttp.send();`
- Verbindung zur Zieldatei herstellen:
  - URL der Datei und Sendemethode: GET oder POST
  - Aus Sicherheitsgründen können nur Daten aus derselben Domain geladen werden, d. h. XML-Datei muss vom gleichen Server wie die ladende HTML-Datei stammen
  - Asynchrone (true) oder synchrone (false) Übertragung

Alternativer Datenzugriff über Unterelemente

- Zugriff erfolgt hier unter Ausnutzung der Struktur des Dokumentenbaums
- Unterelemente eines Elements sind childNodes
- childNodes von Kunde: Nachname, Vorname, etc.
  - Diese bilden ein weiteres Feld
- Damit ergibt sich für das Beispiel:
  - Feldelement 1: Nachnameknoten,
  - Feldelement 3: Vornameknoten, usw.
  - Feldelement 2 etc.:
    Knoten für Zeilenumbrüche
- Der Texteintrag eines Knotens steht in der Variable textContent

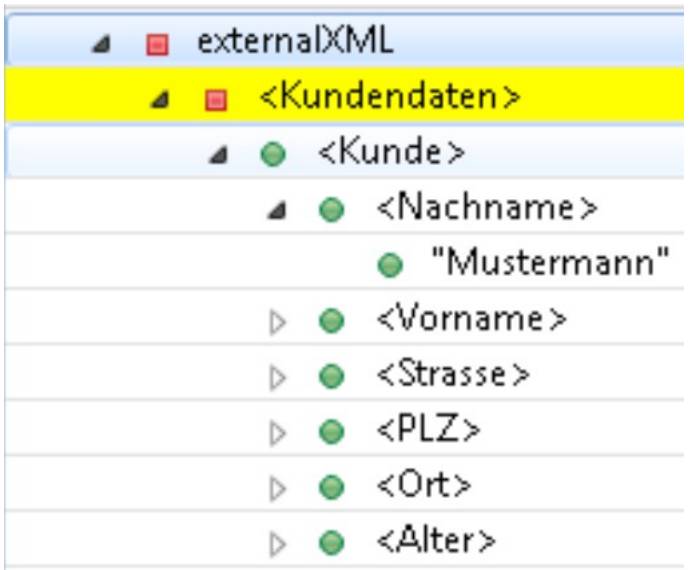


Datenzugriff: Beispiel 1

```
var kunden = externalXML.getElementsByTagName("Kunde");
var str = "";
for (var i = 0; i < kunden.length; i++) {
  str += kunden[i].childNodes[1].textContent + ", ";
  str += kunden[i].childNodes[3].textContent + ", ";
  str += kunden[i].childNodes[11].textContent + "<br>";
  str += kunden[i].childNodes[5].textContent + "<br>";
  str += kunden[i].childNodes[7].textContent + " ";
  str += kunden[i].childNodes[9].textContent + "<br>";
}
```

Ausgabe in einem :

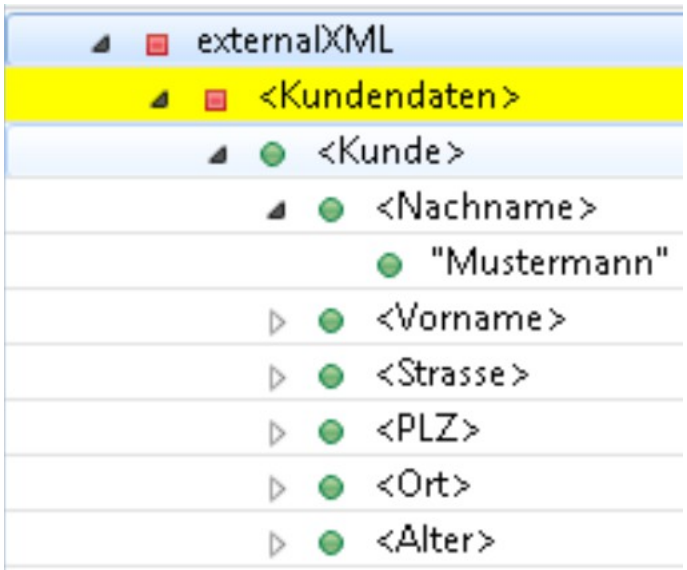
```
Mustermann, Markus, 32
Musterstr. 12
78101 Musterstadt
Musterfrau, Marianne, 35
Musterweg 14
89102 Musterweiler
```



Datenzugriff: Beispiel 2

```
var kunden = externalXML.childNodes[1];  // (*)
var str = "";
for (var i=1; i<kunden.childNodes.length; i+=2) {
  str += kunden.childNodes[i].childNodes[1].textContent + ", ";
  str += kunden.childNodes[i].childNodes[3].textContent + ", ";
  str += kunden.childNodes[i].childNodes[11].textContent + "<br>";
  str += kunden.childNodes[i].childNodes[5].textContent + "<br>";
  str += kunden.childNodes[i].childNodes[7].textContent + " ";
  str += kunden.childNodes[i].childNodes[9].textContent + "<br>";
}
```

(*): Indexwert abhängig von der Position der Kundendaten im XML-Code

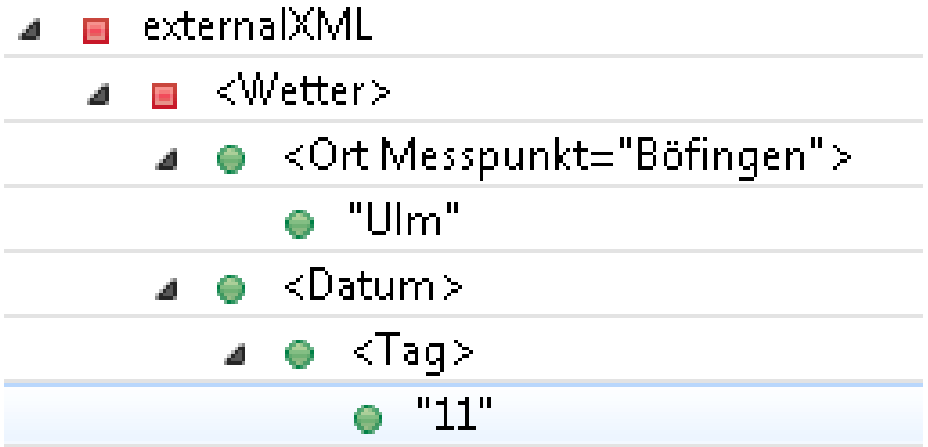


Elementattribute

- Zugriff mittels der Funktion

  `getAttribute("Attributname")`

- Attribute eines Knotens stehen ebenfalls nacheinander im Feld **`attributes`**
  - Attribut besteht hier aus einem Namen (**`name`**, **`nodeName`**) sowie einem Wert (**`value`**, **`nodeValue`**, **`textContent`**)
  - Namen und Werte stehen unter verschiedenen Bezeichnungen zur Verfügung



Beispiel Wetterdaten

- Ort und Messpunkt auslesen mit Funktion **`getAttribute()`**:

```
<div id = "wetter"></div>
<script>
// Code wie zuvor
var externalXML = xmlfile.responseXML;
var wetter = externalXML.getElementsByTagName("Wetter");
var str = "Ort: " +
          wetter[0].childNodes[1].textContent + "<br>";
str += "Messpunkt: " +
wetter[0].childNodes[1].getAttribute("Messpunkt");
document.getElementById("wetter").innerHTML = str; // div
// Code wie zuvor
</script>
```

- Ausgabe:

```
Ort: Ulm
Messpunkt: Böfingen
```

Beispiel Wetterdaten

- Ort und Messpunkt auslesen mit Attributfeld:

```
<div id = "wetter"></div>
<script>
// Code wie zuvor
var externalXML = xmlfile.responseXML;
var wetter = externalXML.getElementsByTagName("Wetter");
var str = "Ort: " +
          wetter[0].childNodes[1].textContent + "<br>";
var attr = wetter[0].childNodes[1].attributes[0];
str += attr.nodeName + ": " + attr.nodeValue + "<br>";
document.getElementById("wetter").innerHTML = str; // div
// Code wie zuvor
</script>
```

- Ausgabe:

```
Ort: Ulm
Messpunkt: Böfingen
```

Bedienoberflächen

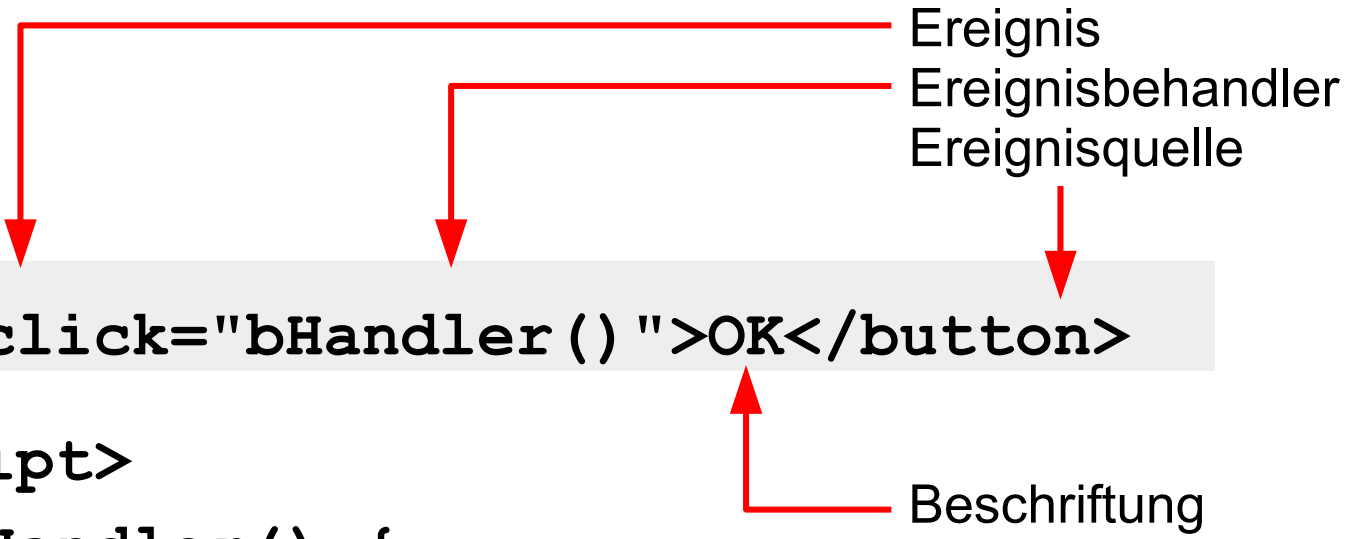
Bedienoberflächen

- Spielen eine zentrale Rolle hinsichtlich der Interaktivität einer Webanwendung
  - Aufgabe 1: Steuern einer Webanwendung
  - Aufgabe 2: Formularerstellung für die Datenerfassung
- Bedienoberflächen bestehen aus Bedienelementen bzw. Widgets ("Window Gadget")
- Die im Folgenden beschriebenen HTML-Bedienelemente können für beide oben angegebenen Aufgaben gleichermaßen eingesetzt werden
  - Es stehen alle wesentlichen grafischen Bedienelemente zur Verfügung, die in der Konzeption grafischer Bedienoberflächen üblich sind

Button

- Einfachstes Bedienelement: Wird benutzt, um eine bestimmte Aktion auszulösen

- Beispiel:




```
<button onclick="bHandler()">OK</button>

<head><script>
function bHandler() {

    //...
}
</script></head>
```

Ereignisbehandler mit Parameter

- Ereignisbehandler können Parameter besitzen
- Auf diese Weise ist z. B. eine einfache Identifikation der Ereignisquelle möglich
  - Sowie die Behandlung der Ereignisse einer Reihe von Buttons mit einem einzigen Ereignisbehandler
- Beispiel:

```
<script>
function bHandler(name) {
  writeln("Button " + name + " clicked");
  //...
}
</script>

<button onclick="bHandler('B1')">OK</button>
```

In getrennten Dateien: HTML

- Button und JavaScript mit Ereignisbehandler in getrennten Dateien
-

```
<html>
  <head>
    <script src="ButtonExample.js" defer>
    </script>
  </head>
  <body>
    <div>Button Example</div>
    <button id="btn">OK</button>
    <div id="output"></div>
  </body>
</html>
```

In getrennten Dateien: JavaScript

- **ButtonExample.js:**

```
var btn = document.getElementById("btn");
var output = document.getElementById("output");
btn.onclick = function () {
    output.innerHTML = "Button clicked";
}
```

- **defer** stellt beim Einbinden dieser Datei sicher, dass der Code in dieser Datei erst ausgeführt wird, wenn die **div**-Elemente in der HTML-Datei vollständig verarbeitet wurden und damit bekannt sind

Auswertung des Ereignisobjekts

- Der Browser erzeugt zu jedem Ereignis automatisch ein Ereignisobjekt
- Im Ereignisobjekt wird u. a. gespeichert, welches Objekt das Ereignis auslöste
- Im Ereignisbehandler können diese Daten ausgewertet werden
- Hierfür wird der Ereignisbehandler parametriert (Parametername beliebig)
- Beispiel:

```
<head><script>
function bHandler(event) {
  writeln("Button " + event.currentTarget.id + " clicked");
}
</script></head>
```

-
```
<body>
<button id="B1" onclick="bHandler(event)">OK1</button><br>
<button id="B2" onclick="bHandler(event)">OK2</button><br>
</body>
```

- Funktioniert sowohl für JavaScript in derselben als auch in einer getrennten Datei

Texteingabe: Einzeiliges Eingabefeld

- Dient zur Aufnahme einer Zeichenkette (Wörter, Zahlen)

  - Standalone-Tag (d. h. kein Ende-Tag erforderlich)

- `<input type="text" size="30" maxlength="40">`
  - **`size`**: Anzeigebreite des Feldes in Zeichen (automatisches Scrolling, falls **`size < maxlength`**)
  - **`maxlength`**: Maximale Anzahl einzugebender Zeichen

- Vorbelegungstext kann mit **`value="..."`** oder **`placeholder="..."`** (HTML5) eingefügt werden

Beispiel

- Button kopiert Text von einem Eingabe- in ein Ausgabetextfeld
- HTML:

```
<input type="text" id="str" size="30" maxlength="40">
<button onclick="textHandler()">OK</button>
<input type="text" id="output">
```

- JavaScript:

```
function textHandler() {
  var str = document.getElementById("str").value;
  document.getElementById("output").value = str;
}
```

Addierer

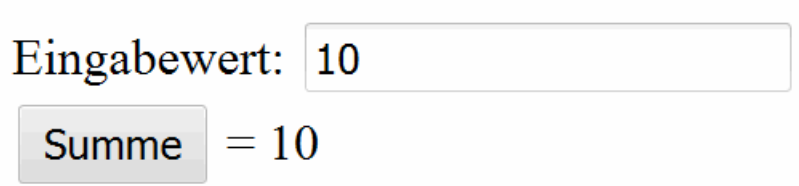
```
<html><head><script>
  var summe = 0;
  function sum() {
    summe += parseInt(
      document.getElementById("input").value);
    document.getElementById("output").value = summe;
  }
</script></head>

<body>
  <input id="input"  type="text">Eingabewert<br>
  <input id="output" type="text">
  <button onclick="sum()">Summe</button>
</body>
</html>
```

- Kommazahlen: **`parseFloat()`**
- Funktionen liefern NaN, wenn String nicht als Zahl interpretiert werden kann

Addierer: Ausgabe in Textelement

```
<html>
<head>
<script>
  var summe = 0;
  function sum() {
    summe += parseInt(
      document.getElementById("input").value);
    document.getElementById("output").innerHTML =
      " = " + summe;
  }
</script>
</head>
<body>
  Eingabewert: <input id="input" type="text"><br>
  <button onclick="sum()">Summe</button>
  <span id="output"> = </span>
</body></html>
```



Ausgaben können auch in gewöhnliche Textelemente einer HTML-Seite erfolgen

Mehrzeiliges Eingabefeld

- Dient zur Aufnahme von längeren Textabschnitten

- `<textarea id="textfeld" cols="50" rows="20"></textarea>`
  - Erfordert abschließendes `</textarea>`-Tag
    - Ermöglicht Eingabe von Vorbelegungstext
  - **`cols`**: Zeichen pro Zeile
  - **`rows`**: Anzahl der angezeigten Zeilen

Beispiel

- Button kopiert Text von einem Eingabe- in ein Ausgabetextarea
- HTML:

```
<textarea id="taIn" cols="50" rows="20"></textarea>
<button onclick="taHandler()">OK</button>
<textarea id="taOut" cols="50" rows="20"></textarea>
```

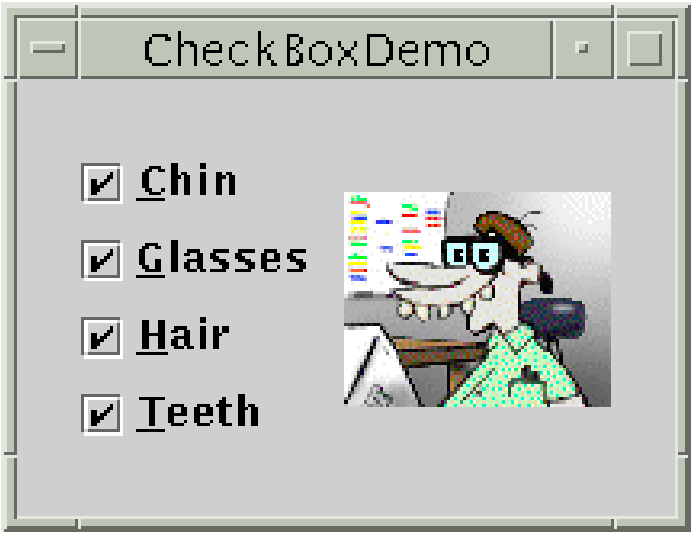
- JavaScript:

```
function taHandler() {
  var str = document.getElementById("taIn").value;
  document.getElementById("taOut").value = str;
}
```

  - Bei Ausgabe in ein Textfeld bleiben Zeilenumbrüche erhalten, bei Ausgabe z. B. in ein ** **-Element nicht

Check Box

- Besteht aus einer kleinen, benannten Box zum Ankreuzen
  - Wird i. d. R. verwendet, um aus einer Palette von Optionen die gewünschten auszuwählen
  - Optionen sind häufig zu thematischen Gruppen zusammengefasst
- Die gewählten Optionen sind durch einen Indikator, z. B. durch einen Haken oder ein kleines Kreuz, markiert

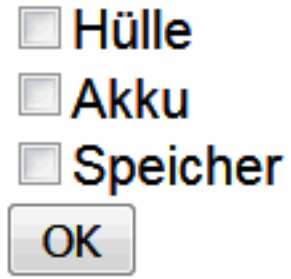


[2]

Check Box: HTML

```
<body>
<input type="checkbox" id="cb1" value="cover">Hülle<br>
<input type="checkbox" id="cb2" value="accu">Akku<br>
<input type="checkbox" id="cb3" value="mc">Speicher<br>
<button onclick="cbHandler()">OK</button>
<div id="output"></div>
</body>
```

- Box vorauswählen: `<input type=... checked>Name<br>`
- **`value`**: Wert, der mit JavaScript aus dem Element ausgelesen werden kann

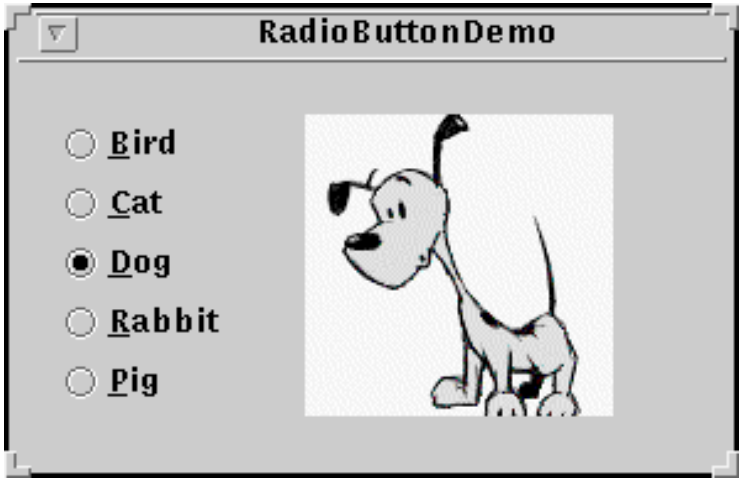


Check Box: JavaScript

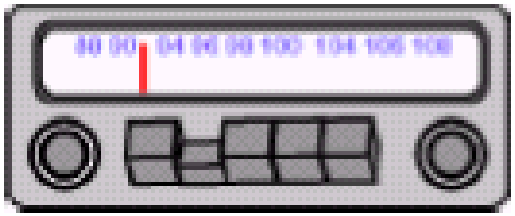
```
<script>
function cbHandler() {
  var str = "";
  if (document.getElementById("cb1").checked)
    str += document.getElementById("cb1").value + ", ";
  if (document.getElementById("cb2").checked)
    str += document.getElementById("cb2").value + ", ";
  if (document.getElementById("cb3").checked)
    str += document.getElementById("cb3").value;
  document.getElementById("output").innerHTML = str;
}
</script>
```

Radio Button

- Besteht aus einem kleinen, benannten Kreis
- Konzept stammt vom Programm-wähler alter Kofferradios ab
  - Wenn ein Button gedrückt wird, springt der gerade selektierte heraus
  - Radio Buttons sind nur als Gruppe sinnvoll
  - Nur eine Option in der Gruppe kann selektiert sein
  - Wird z. B. benutzt, um zwischen verschiedenen Werkzeugen oder Ausgabeformaten zu wählen

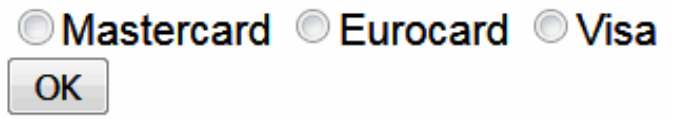


[2]



Radio Button: HTML

- Beispiel:




```
<input type="radio" id="rb1" name="rbgroup1"
  value="Mastercard">Mastercard

<input type="radio" id="rb2" name="rbgroup1"
  value="Eurocard">Eurocard

<input type="radio" id="rb3" name="rbgroup1"
  value="Visa">Visa<br>

<button onclick="rbHandler()">OK</button>

<div id="output"></div>
```

- Alle Radio Buttons mit dem gleichen Namen gehören zur gleichen Gruppe
- **`id`** und **`value`**: Auswertung entsprechend zu Check Boxes
- Radio Button vorauswählen mit **`checked`**

Radio Button: JavaScript

```
function rbHandler() {
  var card = "";

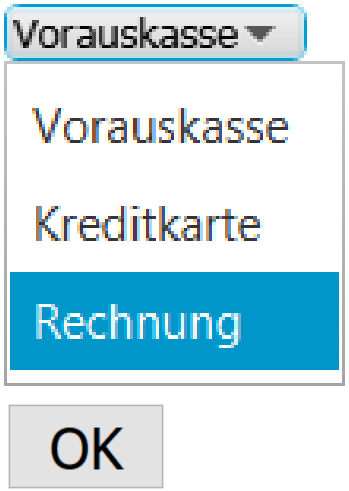
  if (document.getElementById("rb1").checked)
    card = document.getElementById("rb1").value;

  if (document.getElementById("rb2").checked)
    card = document.getElementById("rb2").value;

  if (document.getElementById("rb3").checked)
    card = document.getElementById("rb3").value;

  document.getElementById("output").innerHTML = card;
}
```

Auswahlliste: HTML



- Durch Mausklick zu öffnende Liste von Einträgen, von denen einer ausgewählt werden kann
  - **<select>** mit Optionen als **<option>**
- Beispiel:

```
<select id="zahlweise">
  <option value="Kreditkarte">Kreditkarte</option>
  <option value="Vorauskasse">Vorauskasse</option>
  <option value="Rechnung" selected>Rechnung</option>
</select>
<button onclick="selectHandler()">OK</button>
<div id="output"></div>
```

- Vorselektion: **selected**, Auswahl steht in Variable **value**

Auswahlliste: JavaScript

```
function selectHandler() {
  var selected =
    document.getElementById("zahlweise").value;

  document.getElementById("output").innerHTML =
    selected;
}
```

Bedienelemente platzieren

Bedienelemente platzieren

- Das CSS-Layoutmodell **Flexible Box Layout** (kurz: **Flexbox**) erlaubt responsive Bedienoberflächen, z. B. für Apps
- Bei diesem Modell werden die Bedienelemente in Containern in Zeilen und Spalten angeordnet
- Die Bedienelemente werden dabei automatisch in den zur Verfügung gestellten Raum eingepasst
- Deklaration erfolgt in CSS durch die Eigenschaft `display:flex;`
  - Browserspezifische Erweiterungen der Syntax können erforderlich sein
- Siehe:
  - https://css-tricks.com/snippets/css/a-guide-to-flexbox/
  - https://wiki.selfhtml.org/wiki/CSS/Tutorials/Flexbox
  - https://www.w3schools.com/cssref/css3_pr_justify-content.asp

Flexbox

- Als Voreinstellung erfolgt die Anordnung zeilenweise
- Beispiel: Drei Radio Buttons in einer Zeile 
- CSS:

```
.hcontainer {
  display:flex;
}
```

- HTML:

```
<div class="hcontainer">
  <input type="radio" name="rbgrp">
  <input type="radio" name="rbgrp">
  <input type="radio" name="rbgrp">
</div>
```

Flexbox: Optionen

- **`justify-content:flex-start|flex-end|center|space-between|space-around;`**
  - **`flex-start/flex-end/center`**: Elemente werden dicht gepackt links/rechts/zentriert im Container ausgerichtet
  - **`space-between`**: Elemente werden mit Zwischenraum angeordnet, sodass sie den Container füllen
  - **`space-around`**: Elemente werden mit Raum davor, dazwischen und dahinter angeordnet, sodass sie den Container füllen
- **`flex-direction:column`**
  - Elemente werden spaltenweise angeordnet
- Ausprobieren möglich auf den jeweils zugeordneten Webseiten unter https://www.w3schools.com/css/css3_flexbox.asp

Flexbox: Anpassung an Browser

- Erfolgt durch Präfixe, sofern erforderlich:
  - Webkit-Browser (Safari): **`-webkit-`**
  - Als zusätzliche Anweisungen möglich
- Beispiel:

```
display: -webkit-flex;
display: flex;
-webkit-flex-direction: column;
flex-direction: column;
-webkit-justify-content: space-between;
justify-content: space-between;
```

Geräteabhängige Ereignisse

Geräteabhängige Ereignisse

- Die bisherigen Beispiele berücksichtigten:
  - Geräteunabhängige Ereignisse
    - Button-Event **`onclick`**
  - Ladeereignisse
    - Bild laden: **`onload`**
    - JSON- bzw. XML-Datei laden: **`readyState`**
- Daneben gibt es geräteabhängige Ereignisse
  - Hierzu gehören z. B. Mausereignisse

Mausereignisse

- **`onmousedown`**: Ein Mausbutton wurde gedrückt
  - Nr. des Buttons aus der **`event.button`** Eigenschaft des Ereignisobjekts auslesen (0: links, 1: Rad, 2: rechts).
- **`onmouseup`**: Ein Mausbutton wurde losgelassen
- **`onmousewheel`**: Das Mausrad wurde rotiert
- **`onmousemove`**: Die Maus wurde bewegt
- **`onmouseover`**: Die Maus betritt den Bereich des Elements, für welches Mausereignisse ausgewertet werden
- **`onmouseout`**: Die Maus verlässt den Bereich des Elements, für welches Mausereignisse ausgewertet werden

Beispiel: Einfache Zeichenanwendung

- Grundlage ist ein Canvas-Element
- HTML:

```
<canvas id="canvas" width="400" height="300">
  <p>Leider kein Canvas verfügbar</p>
</canvas>
```

- Ziel: Bei gedrücktem Mausbutton wird ein freier Pfad mit vorgegebener Strichbreite und -farbe gezeichnet

Zeichenanwendung: JavaScript

- Mit JavaScript auf den Canvas zugreifen:

```
var canvas = document.getElementById('canvas');
var context = canvas.getContext("2d");
```

- Zeichenparameter festlegen:

```
context.fillStyle = "#FFFFFF"; // weißer Hintergrund

context.strokeStyle = "#ff0000"; // rote Linie

context.lineJoin = "round"; // Liniensegmentenden
                            // abgerundet verbinden
                            // bevel: Gerade Linie
                            // miter: Gehrungs-
                            //        verbindung
                            //        (Bilderrahmen)

context.lineWidth = 5; // Linienbreite
```

Ereignisbehandler für die Maus (1)

- Wenn Maus gedrückt wird, neuen Zeichenvorgang beginnen

```
canvas.onmousedown = function(event) {
  // Mauskoordinaten berechnen
  var mouseX = event.pageX - this.offsetLeft;
  var mouseY = event.pageY - this.offsetTop;

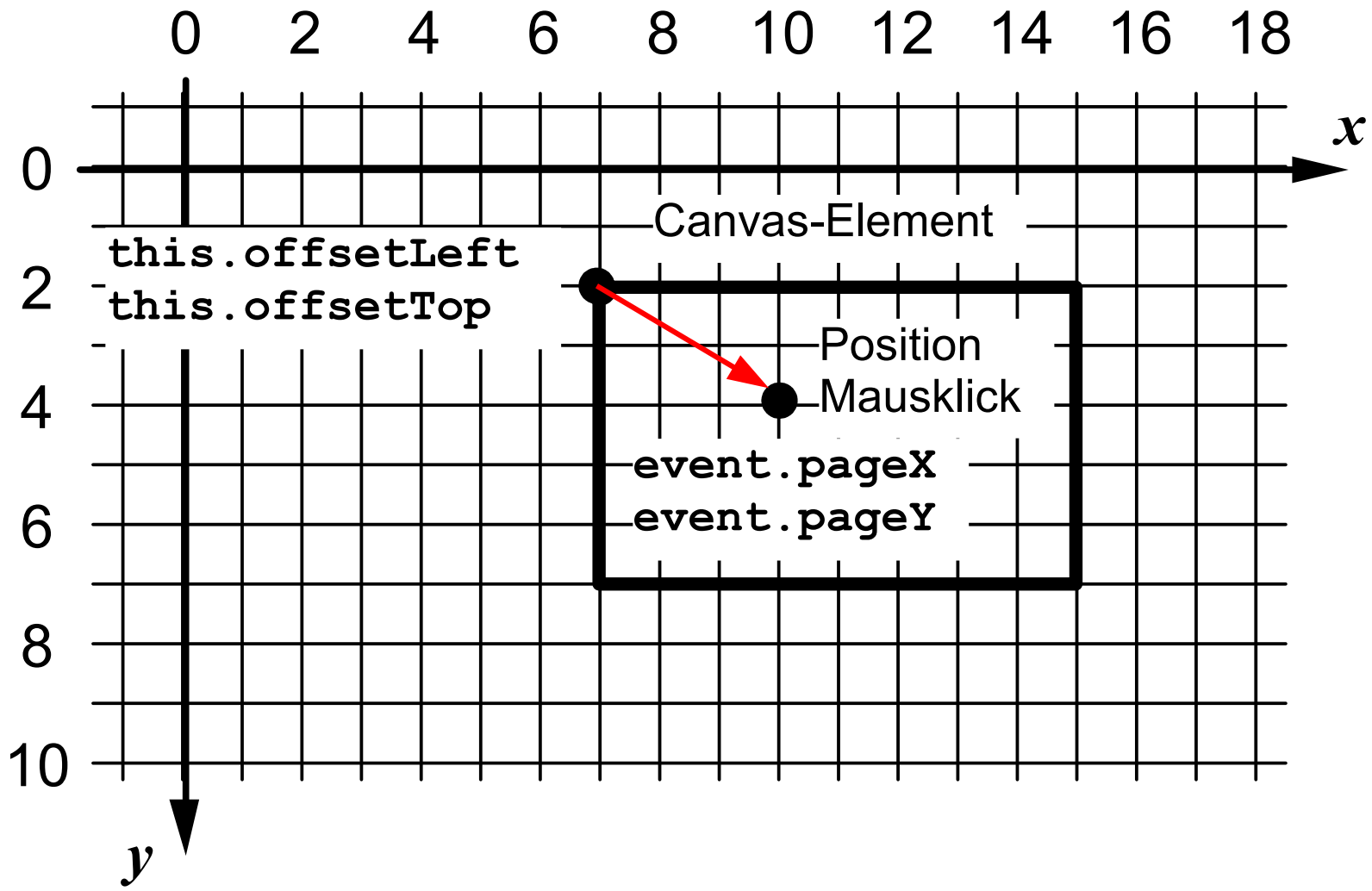
  paint = true; // Zeichenmodus aktivieren

  // Vorhandenen Canvasinhalt löschen
  context.fillRect(0, 0, this.width,
                         this.height);

  context.beginPath();
  context.moveTo(mouseX,mouseY);
}
```

  - `this` ist das Canvas-Element selbst

Berechnung der relativen Elementkoordinaten



Ereignisbehandler für die Maus (2)

- Bei gedrückter Maus Pfad zeichnen

```
canvas.onmousemove = function(event) {
  if(paint){
    var mouseX = event.pageX - this.offsetLeft;
    var mouseY = event.pageY - this.offsetTop;
    context.lineTo(mouseX,mouseY);
    context.stroke();
  }
};
```

Ereignisbehandler für die Maus (3)

- Wenn Mausbutton losgelassen wird oder Maus Canvas verlässt: Zeichnen beenden

```
canvas.onmouseup = function() {
  paint = false;
};

canvas.onmouseout = function() {
  paint = false;
};
```

Allgemeine Bedienelemente

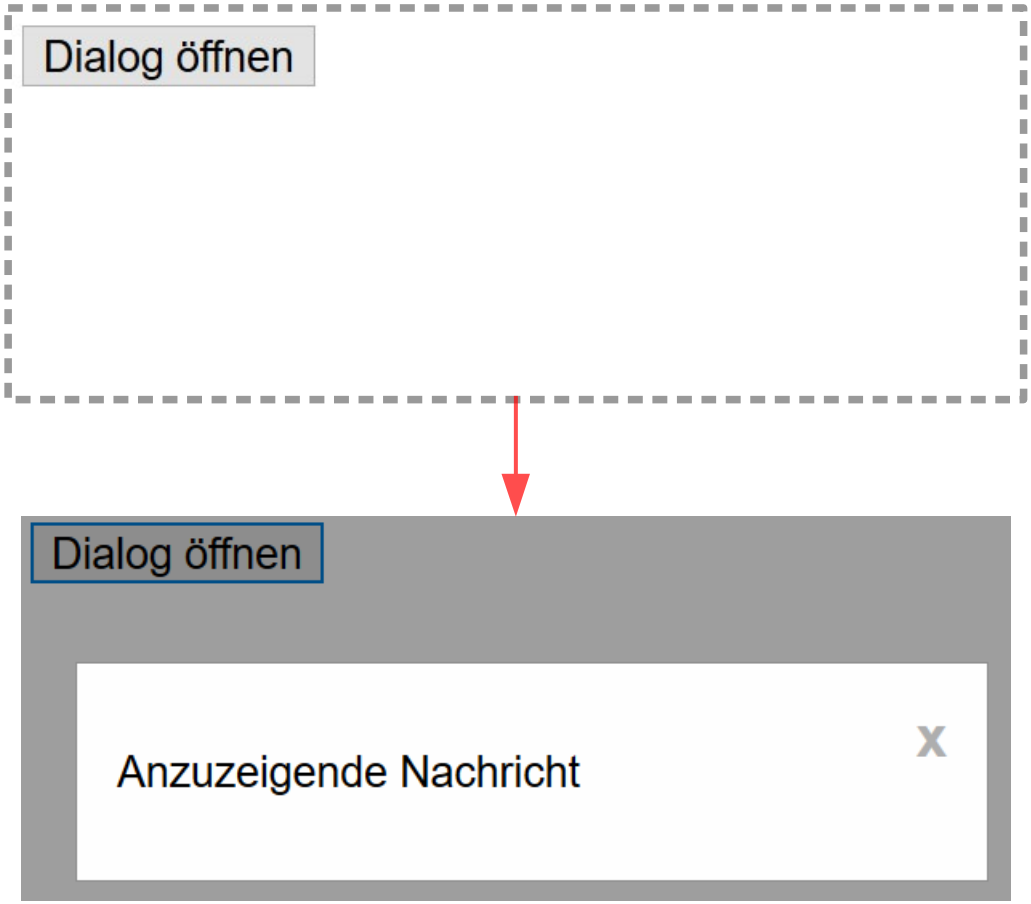
Bedienelemente mit HTML, CSS und JavaScript

- HTML definiert eine Anzahl grundlegender Bedienelemente
- Für die effektive Strukturierung von Bedienoberflächen existiert in modernen Entwicklungsumgebungen eine Vielzahl weiterer Standard-Bedienelemente
- Mit Hilfe von HTML, CSS und JavaScript lassen sich diese Bedienelemente auch für die Entwicklung von Webanwendungen implementieren
- Unter www.w3schools.com findet sich eine Reihe von Beispielimplementierungen
  - Siehe hierzu https://www.w3schools.com/howto/
  - Von diesen werden im weiteren einige behandelt

Modale Bedienelemente

Modale Dialoge bzw. Boxen

- Dienen dazu, den normalen Programmablauf zu unterbrechen
- Während der modale Dialog angezeigt wird, kann das Hauptfenster einer Anwendung nicht benutzt werden
- Beispiel: Webseite mit Button
  - Drücken des Buttons öffnet modalen Dialog



Modale Dialoge bzw. Boxen: Einsatz

- Beispiele:
  - Benachrichtigung oder verlangte Information anzeigen
    - Fehleranzeige, Terminerinnerung, Bild anzeigen
  - Benutzereingabe durchführen
    - Z. B. Account erstellen oder Einloggen in Account
    - Kontaktformular
- Beruht auf dem Schalten von z. B. einer **`div`**-Box in den sichtbaren oder unsichtbaren Zustand mit Hilfe von JavaScript

Umsetzung: HTML

```
<head>
  <link rel="stylesheet" href="ModalBox.css">
  <script src="ModalBox.js" defer></script>
</head>

<body>
<!-- Button -->
<button id="myBtn">Dialog öffnen</button>

<div id="myModal" class="modal"><!-- Dialogbox -->
  <div class="modal-content"><!-- Dialoginhalt -->
    <span class="close" id="close">&times;</span>
    <p>Anzuzeigende Nachricht</p>
  </div>
</div>
</body>
```

Script erst ausführen, wenn alle Elemente geladen. Sonst funktioniert der Zugriff nicht!

Entity-Referenz

Umsetzung: CSS

```
.modal {
  display: none; /* Voreinstellung: Unsichtbar */
  z-index: 1; /* obenauf liegend (schwebend) */
  /* weitere Formatdefinitionen */
}
```

Umsetzung: JavaScript (1)

- Programmcode wird in eine eigene JavaScript-Datei ausgelagert

```
// Zugriff auf Elemente des modalen Dialogs
// herstellen:

// Button zum Öffnen des Dialogs
var btn = document.getElementById("myBtn");

// Dialogbox
var modal = document.getElementById('myModal');


// <span>-Element für das Schließen des Dialogs
var span = document.getElementById('close');
```

Umsetzung: JavaScript (2)

```
// Ereignisbehandler für Button: Dialog öffnen
btn.onclick = function () {
  modal.style.display = "block";
}

// Dialog schließen, wenn <span> (x) angeklickt
span.onclick = function () {
  modal.style.display = "none";
}
```

Modales Bild

- CSS & JavaScript: Entsprechend zum modalen Dialog
- HTML: Bildelement statt ** **-Box im modalen Dialog:

```
<head>
  <link rel="stylesheet" href="ModalImage.css">
  <script src="ModalImage.js" defer></script>
</head>

<body>

  <button id="myBtn">Bild öffnen</button>

  <div id="myModal" class="modal"> <!-- Box -->
    <!-- Schließen-Button -->
    <span class="close" id="close">&times;</span>
    <img src="bild.jpg">
    <div><!-- Bildbeschriftung --></div>
  </div></body>
```

Modales Bild: Nachteile

- HTML:

```
<div id="myModal" class="modal"> <!-- Box -->
  <!-- ... -->
  <img id="myImg" src="bild.jpg">
  <!-- ... -->
</div>
```

- Das `<img>`-Element steht im Markup
  - Bild wird zusammen mit HTML-Seite geladen
  - Bild potentiell groß
  - Für mobile Verbindungen potentiell unnötiger Traffic
- Bild erst dann laden, wenn es wirklich angeschaut werden soll

Modales dynamisches Bild: HTML

- HTML:

```
<body>

<!-- Entweder weiterhin Button oder
     kleines Vorschaubild: Fungiert als Button,
     um großes Bild nachzuladen -->

<img id="myImg" src="bildKlein.jpg" alt="-">

<div id="myModal" class="modal"> <!-- Box -->
  <!-- Schließen-Button -->
  <span class="close" id="close">&times;</span>
</div>
</body>
```

- CSS: wie gehabt

Modales dynamisches Bild: JavaScript

```
var modal = document.getElementById('myModal');
var img = document.getElementById('myImg');
var bigImg = null;
img.onclick = function () { // Vorschaubild
  // Mehrfaches Laden des Bilds verhindern
  if (bigImg === null) {
    bigImg = new Image();
    bigImg.src = "bildGroß.jpg";
    modal.appendChild(bigImg);
  }
  modal.style.display = "block";
};
```

- Rest wie gehabt

- Nachteil: Bildbox wird sofort angezeigt
- Leere Box bei langsam ladendem Bild
- Besser: Box erst anzeigen, wenn Bild geladen

Modales Bild mit ereignisgesteuerter Anzeige

- HTML und CSS: Wie gehabt
- Änderungen am JavaScript:

```
var bigImg = null;
img.onclick = function () {
  if (bigImg === null) {
    bigImg = new Image();
    bigImg.src = "bildGroß.jpg";
    bigImg.onload = function() { // Ereignisbeh.
      modal.appendChild(bigImg);
      // Erst anzeigen, wenn geladen!
      modal.style.display = "block";
    };
  }
  else
    modal.style.display = "block";
};
```

- Nachteil: Nach dem Klicken auf das kleine Bild passiert eine Zeitlang nichts
- Besser: Visuelles Feedback geben, dass Ladevorgang läuft

Modales Bild mit Ladeanzeige: CSS

- Ladeanzeige bzw. Loader umgesetzt mit Hilfe von CSS
- Animierter Kreis (siehe Übungsblatt und www.w3schools.com)

```
.loader {
    border: 16px solid #f3f3f3; /* light grey */
    border-top: 16px solid #3498db; /* blue */
    border-radius: 50%; /* transforms into circle */
    width: 120px;
    height: 120px;
    animation: spin 2s linear infinite;
}

@keyframes spin {
    0% { transform: rotate(0deg); }
    100% { transform: rotate(360deg); }
}
```

Modales Bild mit Ladeanzeige: JavaScript

```
img.onclick = function () {
  if (bigImg===null) {
    var loader = document.createElement("div");
    loader.className = "loader"; // CSS zuweisen
    modal.appendChild(loader); // Loader laden
    modal.style.display = "block";

    bigImg = new Image();
    bigImg.src = "bildGroß.jpg";
    bigImg.onload = function() {
      modal.removeChild(loader); // Loader entladen
      modal.appendChild(bigImg);
    };
  }
  else modal.style.display = "block";
};
```

Loader testen: Zeitgesteuerte Vorgänge

- Loader zu Testzwecken eine Zeitlang anzeigen:

```
img.onclick = function () {
  if (bigImg===null) {
    // … wie zuvor
    bigImg.src = "bildGroß.jpg";
    bigImg.onload = function() {
      function removeLoader() {
        modal.removeChild(loader);
        modal.appendChild(bigImg);
      }
      setTimeout(removeLoader, 4000);
    };
  }
  modal.style.display = "block";
};
```

Simulation des Bildladevorgangs

- `setTimeout`: Timerfunktion, die nach Ablauf der angegebenen Zeit in ms die als erstes Argument gelieferte Funktion aufruft

Code zeitgesteuert aufrufen

- **`setTimeout(fkt, t)`**: Funktion **`fkt`** wird einmal nach **`t`** Millisek. aufgerufen
- **`var id = setInterval(fkt, t)`**: Funktion **`fkt`** wird wiederkehrend nach **`t`** Millisek. aufgerufen. Funktion **`setInterval`** liefert Intervallidentifikationscode **`id`**.
- **`clearInterval(id)`**: Funktion beendet das wiederkehrende Ereignis, welches durch den Intervallidentifikationscode **`id`** definiert ist
- Die Funktion **`setInterval`** kann z. B. dafür benutzt werden, um einen animierten Ladebalken (engl.: *Progress Bar*) zu implementieren

Ladebalken (Progress Bar): HTML


```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="ProgressBar.css">
    <script src="ProgressBar.js" defer></script>
  </head>
  <body>
  <div>Progress Bar</div>
  <div id="myProgress">
    <div id="myBar">
      <div id="label">10%</div>
    </div>
  </div>
  </body>
</html>
```

- **myProgress**: Grau getönter Container für **myBar**
- Breite von **myBar** wird wiederkehrend erhöht
- **label** wird als Teil von **myBar** bei Verbreiterung mitbewegt

- CSS: Siehe Übungsblatt

Ladebalken: JavaScript

```
var myBar = document.getElementById("myBar");
var label = document.getElementById("label");
var width = 0;
var id = setInterval(frame, 100); // 100ms-Raster
function frame() {
  if (width >= 100) { // Animation beenden
    clearInterval(id);
  }
  else {
    width++;
    myBar.style.width = width + '%';
    label.innerHTML = width + '%';
  }
}
```

- Nachteil: Progress Bar läuft sofort bis zum Endwert 100%

Parametrischer Ladebalken

- Vorgabe eines Prozentwerts
  - Animation des Ladevorgangs läuft bis zum vorgegebenen Wert
-
```
function moveTo(percent) {
  if (0 <= percent && percent <= 100) {
    var myBar = document.getElementById("myBar");
    var label = document.getElementById("label");
    var width = 0;
    var id = setInterval(frame, 100);
    function frame() {
      if (width >= percent) {
        clearInterval(id);
      } else {
        width++;
        myBar.style.width = width + '%';
        label.innerHTML = width + '%';
}} }  }
```

Snack Bar

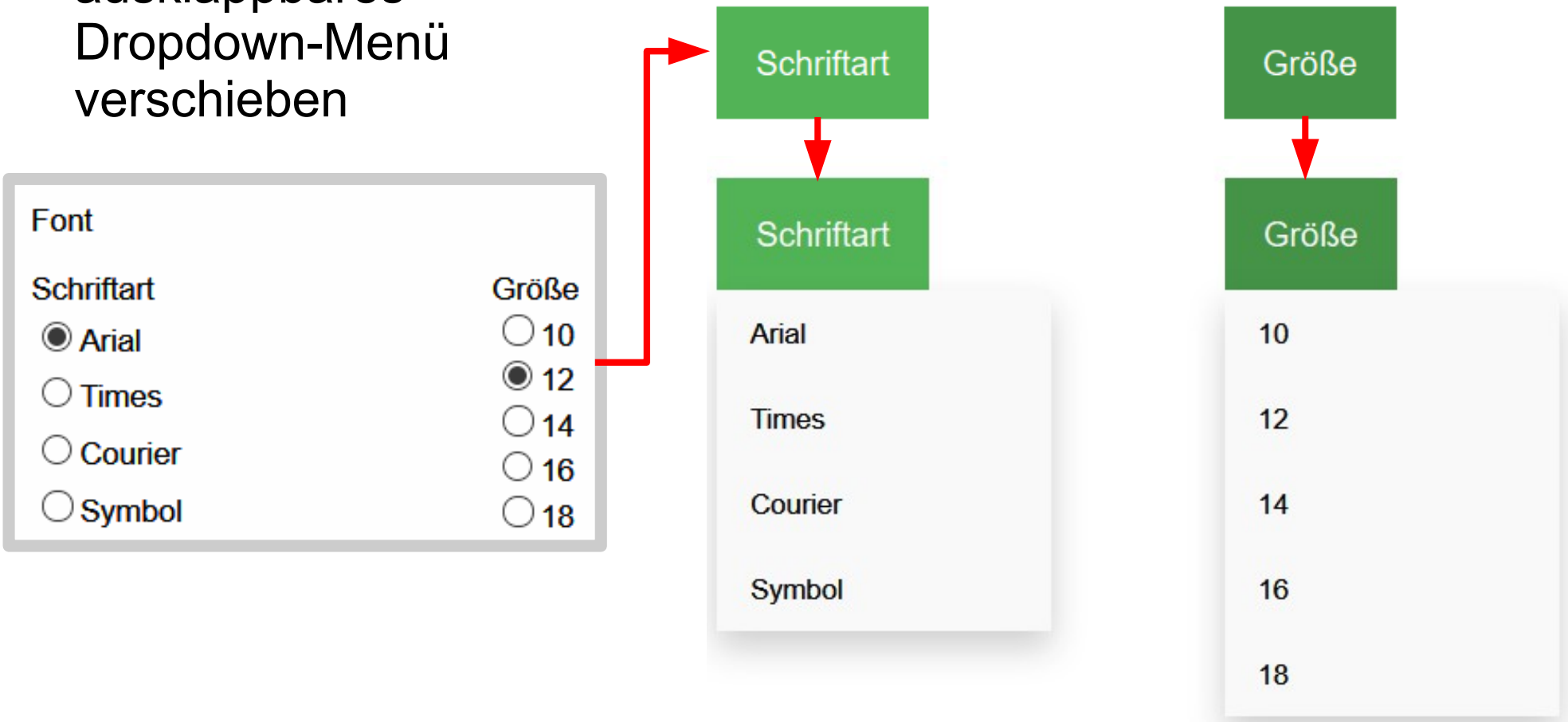
- Meldung anzeigen und nach einer Weile automatisch ausblenden
  - Z. B. bei erfolgreich abgeschlossenem Vorgang (Download, Druckvorgang,…)
-
```
function showSnackbar(message) {
  // Get the snackbar DIV
  var sb = document.getElementById("snackbar");

  // Defined in CSS-File: "show" class sets
  // visibility from default 'hidden' to 'visible'
  sb.className = "show";
  sb.innerHTML = message;

  // After 3 seconds, remove show class from DIV
  setTimeout(function(){ sb.className = ""; }, 3000);
}
```
- HTML: `<div id="snackbar"></div>`, CSS: Siehe Übungsblatt

Dropdown-Menü

- Radio Buttons nehmen viel Platz in der Bedienoberfläche ein
- Alternativen: HTML-Auswahlliste oder Buttons in ein ausklappbares Dropdown-Menü verschieben



Dropdown-Menü

- HTML
  - Menü aufklappen, wenn Button angeklickt wird

```
<div class="dropdown">
  <button onclick="doDropDown('font')"
    class="dropbtn">Schriftart
  </button>
  <div id="font" class="dropdown-content">
    <a onclick="handle('f1')">Arial</a>
    <a onclick="handle('f2')">Times</a>
    <a onclick="handle('f3')">Courier</a>
    <a onclick="handle('f4')">Symbol</a>
  </div>
</div>
```

Dropdown-Menü: CSS

- ```
  /* Dropdown Content (Hidden by Default) */

  .dropdown-content {
      display: none;
      /* … */
  }
  ```
- ```
  /* Show the dropdown menu (use JavaScript and
  className-Property to add this class to
  .dropdown-content container) */

  .show {display: block;}
  ```
- Restliches CSS: siehe Übungsblatt

Dropdown-Menü: Klasse dynamisch zuweisen

- Dropdown-Menü durch Zuweisen einer Klasse sichtbar schalten
- HTML-Elemente können zu mehr als einer CSS-Klasse gehören
  - Die Namen der zugeordneten Klassen im DOM sind über die **`className`**-Eigenschaft verfügbar
  - Bei mehreren Klassen: Liste von Namen
  - Dann irreführende Bezeichnung (besser: **`classNames`**)
- HTML5 beseitigt dieses Problem durch die Definition der **`classList`**-Eigenschaft für Elemente
  - Nur lesbares Feld bzw. **`DOMTokenList`** mit Elementen für jede zugeordnete CSS-Klasse
  - Klassen können zu **`DOMTokenList`** mit Hilfe von Methoden hinzugefügt und gelöscht werden

`classList`-Eigenschaft

- Klassen hinzufügen oder entfernen:
  - **`add(class1,class2,…)`**, **`remove(class1,class2,…)`**
  - **`toggle(class)`**: Klasse hinzufügen, falls nicht vorhanden und umgekehrt
- Eigenschaften abfragen:
  - **`contains(class)`**: Ist Klasse in der Liste? Liefert **`true`** oder **`false`**
  - Anzahl der Klassen auslesen: **`length`**
- Beispiel:
  `document.getElementById("aDIV").classList.add("newClass");`
- Siehe hierzu auch:
  www.w3schools.com/jsref/prop_element_classlist.asp

Dropdown-Menü: JavaScript (1)

- Anklicken des Menütitels führt zum Aufruf der Funktion **`doDropDown`** und damit zum Zuweisen oder Entfernen der Klasse **`show`**:
- ```
  function doDropDown(menu) {
      document.getElementById(menu).classList.
      toggle("show");
  }
  ```
  ```

Dropdown-Menü: JavaScript (2)

- Ereignisbehandler für Schriftartwahl:

```
function handle(font) {
  switch (font) {
    case "f1":alert("Arial gewählt");break;
    case "f2":alert("Times gewählt");break;
    case "f3":alert("Courier gewählt");break;
    case "f4":alert("Symbol gewählt");break;
  }
}
```

Gefaltete Menüs: Akkordeon

- Engl: Accordion
- Das Akkordeon verwaltet eine Reihe von Bereichen, von denen jeweils immer nur einer sichtbar ist
- Der gewünschte Bereich wird mit Hilfe eines Navigations-Buttons ausgewählt
- Standard-Container für Bedienelemente
  - Unter vielen Entwicklungsumgebungen verfügbar
- Mehrfachnutzung des gleichen Bereichs der Bedienoberfläche für die Darstellung unterschiedlicher Funktionen oder Informationen und den jeweils zugeordneten Bedienelementen

Akkordeon: HTML

```
<button class="accordion">Schriftart</button>
<div class="panel">
  <span><input type="radio" name="bg">Arial</span><br>
  <span><input type="radio" name="bg">Times</span><br>
  <!-- usw. -->
</div>

<button class="accordion">Größe</button>
<div class="panel">
  <!-- content -->
</div>

<button class="accordion">Farbe</button>
<div class="panel">
  <!-- content -->
</div>
```

- CSS: Siehe Übungsblatt

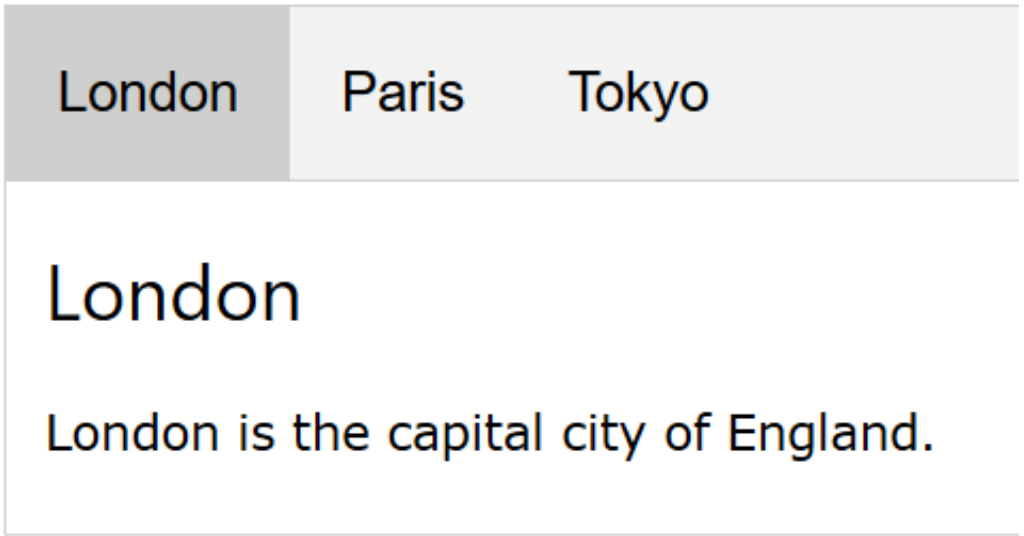
Akkordeon: JavaScript

- Ähnlich zu Dropdown-Menü etc.: ** **-Box durch Zuweisen einer definierten Klasse sichtbar schalten
-
```
var acc =
document.getElementsByClassName("accordion");

for (var i = 0; i < acc.length; i++) {
  acc[i].onclick = function() {
    /* assign color of active mode to button */
    this.classList.toggle("active");
    /* make element following button visible */
    this.nextElementSibling.classList.
      toggle("show");
}}
```
- **nextElementSibling**: Liefert folgendes Element
  - Siehe: www.w3schools.com/jsref/prop_element_nextelementsibling.asp

Reiter

- Reiter (auch: Laschen, engl.: *Tabs*) erlauben eine Ebenendarstellung von Inhalten
  - Entsprechend zu einem Karteikartensystem
- Beispiel: Informationen zu unterschiedlichen Städten auswählen



HTML

- Reiter hier umgesetzt mit Hilfe einer Liste von Links:

```
<ul class="tab">
  <li><a href="#" class="tablinks"
    onclick="openCity(event, 'London')">London</a></li>
  <li><a href="#" class="tablinks"
    onclick="openCity(event, 'Paris')">Paris</a></li>
  <li><a href="#" class="tablinks"
    onclick="openCity(event, 'Tokyo')">Tokyo</a></li>
</ul>
```

  - **`href="#"`**: Ermöglicht Rücksprung auf den Seitenanfang und vermeidet das Neuladen der Seite, was bei einem leeren Link der Fall wäre

- Listeneinträge mit Textumfluss nebeneinander anzeigen:
  CSS: **`ul.tab li {float:left;}`**
- Beim Klicken auf einen Link wird Karteikarte sichtbar geschaltet

Einträge

- ** **-Boxen der Karteikarten sind zunächst unsichtbar
- HTML:

```
<div id="London" class="tabcontent">
  <h3>London</h3>
  <p>London is the capital city of England.</p>
</div>
<div id="Paris" class="tabcontent">...etc.</div>
<div id="Tokyo" class="tabcontent">...etc.</div>
```

- CSS:

```
.tabcontent {
    display: none;
    /* … */
}
.active { background-color: #CCCCCC }
```

JavaScript

```
function openCity(evt, cityName) {
  var i, tabcontent, tablinks;
  // Hide elements with class="tabcontent"
  tabcontent =
    document.getElementsByClassName("tabcontent");
  for (i = 0; i < tabcontent.length; i++) {
    tabcontent[i].style.display = "none";
  }
  // Remove class "active" from elements with class="tablinks"
  tablinks = document.getElementsByClassName("tablinks");
  for (i = 0; i < tablinks.length; i++) {
    tablinks[i].classList.remove("active");
  }
  // Show selected tab, and add class "active" to the link
  // that opened the tab to emphasize it
  document.getElementById(cityName).style.display = "block";
  evt.currentTarget.classList.add("active");
}
```

Formulare

Formular definieren

- Formular kann an beliebiger Stelle innerhalb eines HTML-Body definiert werden:

```
<form action="process.php" method="get">
  <!-- hier folgen die Formularelemente -->
</form>
```

- `<form>...</form>` definiert das Formular
- `action`:
  - Datei, die die Formulardaten verarbeiten soll
  - Wird durch Abschicken des Formulars aufgerufen
  - Ist kein Ziel angegeben, werden die Daten an die abschickende Datei zur Verarbeitung zurückgeschickt
- `method="get"`: Encodierung der Daten erfolgt als URL-Querystring

Formularelement: <input type="submit"...>

- Zum Abschicken des ausgefüllten Formulars
- Beispiel:

```
<input type="submit" value="Abschicken">
```

Definiert Absende-knopf

Beschriftung des Knopfes

- Werte aus Eingabefeldern werden aneinandergehängt und an das unter `"action"` angegebene Ziel geschickt:

```
<form action="process.php" method="get">
```

Formularelement: <input type="reset"...>

- Zum Abbrechen der Eingabe
  - Eingegebene Daten werden verworfen und nicht abgeschickt
- Beispiel:

```
<input type="reset" value="Abbrechen">
```

Definiert Knopf zum Abbrechen der Eingabe

Beschriftung des Knopfes

Formulareinträge prüfen

- Originäre Aufgabe von JavaScript vor dem Abschicken der Formulardaten
- Fragestellungen:
  - Sind alle erforderlichen Felder ausgefüllt worden?
  - Sind in den Feldern die korrekten Datentypen verwendet worden?
    - Wurde z. B. in einem Feld für die Altersangabe ein Zahlenwert in einem realistischen Bereich eingegeben?
  - Wurden Datumsangaben korrekt eingetragen?
- Formularüberprüfungen können auch auf der Serverseite stattfinden (z. B. mit PHP)
  - Wir behandeln hier nur die Clientseite

Formulareinträge prüfen

- Einbinden von JavaScript:

```
<form onsubmit="return check()">
  <!-- hier folgen die Formularelemente -->
</form>
```

- **`onsubmit`**:
  - Ereignis, welches beim Abschicken des Formulars ausgelöst wird. Führt im Beispiel oben zum Aufruf der Funktion **`check()`** als Ereignisbehandler
  - **`return`** erforderlich, damit Rückgabewert der Funktion durch Browser ausgewertet wird im Hinblick auf einen Abbruch des Sendevorgangs bei **`false`**
    - Sonst wird auch im Fehlerfall (**`check()`** liefert **`false`**) das Formular abgeschickt

HTML5-Formularelemente

- Vor HTML5 waren nur relativ wenige Formularelemente vorhanden
- Für viele Eingaben, für die kein spezielles Eingabefeld zur Verfügung stand, wurde das Texteingabefeld gewählt
- Seit HTML5 wurden die Möglichkeiten bei der Formulardefinition erheblich erweitert
- Für unterschiedliche Eingabemuster stehen nun passende *semantische Eingabefelder* zur Verfügung
- Diese berücksichtigen die Bedeutung der Eingabedaten und kommen mit beigeordneten syntaktischen Prüfverfahren
- JavaScript hierfür nicht mehr erforderlich

Semantische Eingabefelder (1)

- `<input type="search">`
  - Keine Einschränkung bei der Eingabe. Ggf. visuell anderes Erscheinungsbild gegenüber einfacher Texteingabe
- `<input type="email">`
  - Korrekte Formatierung der Email-Adresse wird geprüft (@-Zeichen)
- `<input type="url">`
  - Korrekte Formatierung der URL wird geprüft, inklusive Protokoll (https://www.hs-ulm.de)
- Fehler bei der Formatierung führen zur Zurückweisung des Formulars mit passender Fehlermeldung des Browsers

Semantische Eingabefelder (2)

- `<input type="number" min="1" max="5">`
  - Eingabe einer Nummer im zulässigen Bereich wird geprüft
- `<input type="date">`
- `<input type="time">`
- `<input type="tel">`

`usw...`

- Nachlesen und ausprobieren unter:
  - `www.w3schools.com/html/html_form_input_types.asp`
  - `https://wiki.selfhtml.org/wiki/HTML/Tutorials/Formulare/browsereigene_Validierung`

Eingaben erzwingen

- Attribut **`required`**: Eingabefeld muss bei der Eingabe ausgefüllt werden. Ein leeres Feld ist ungültig
- Beispiele:
  - **`<input type="email" required>`**
  - Auswahlliste **`<select>`**
    - Eine Option muss gewählt werden
  - Check Box **`<input type="checkbox">`**
    - Check Box muss angeklickt werden, sonst wird Formular nicht akzeptiert
    - Beispiel: Allgemeine Geschäftsbedingungen akzeptieren

Ausgabefeld

- Tag: **`<output></output>`**
  - Vor HTML5 wurden Ausgaben häufig über Eingabefelder durchgeführt
  - Diese sind stets sichtbar und suggerieren eine Eingabemöglichkeit
  - Im Gegensatz zum **`<input>`**-Feld erscheint ein Ausgabefeld erst, wenn eine konkrete Ausgabe hineingeschrieben wird

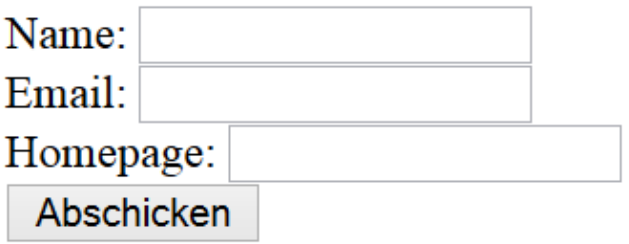
Überprüfung abschalten

- Prüfung auf korrekte Eingabe kann lästig sein
  - Z. B. die zwingende Protokolleingabe beim URL-Feld
- Abschaltung der Prüfung für die Eingabe und das Absenden:
  - `<form novalidate>...</form>`
  - Es werden keine Fehler mehr visualisiert und das Formular kann mit Fehlern abgeschickt werden
- Abschaltung der Prüfung nur für das Absenden:
  - `<button type="submit" formnovalidate>Abschicken</button>`
  - Fehler werden visualisiert, aber Formular kann mit Fehlern oder nur teilweise ausgefüllt abgeschickt werden
  - Sinnvoll z. B. für die Zwischenspeicherung eines schon teilweise ausgefüllten Formulars

Beispiel: Einfache Visitenkarte

```
<form onsubmit="return check()">
  Name: <input type="text" name="name"><br>
  Email: <input type="email" name="email"><br>
  Homepage: <input type="url" name="hpage"><br>
  <input type="submit" value="Abschicken">
</form>
```

- Die automatische Prüfung testet nur Grundsätzliches, wie z. B. Vorhandensein von @ in Email und **`http`** in **`url`**
- Feinere syntaktische und inhaltliche Tests z. B. mit JavaScript
  - Beispiele: Eingabefelder sind nicht leer, Punkte in Email und in URL gegeben, bestimmter Provider oder Nationalität



Zeichenketten

Zeichenketten: Allgemeines

- Zeichenketten sind in JavaScript unveränderliche Folgen von 16-Bit-Zeichen, d. h. eine Operation auf einer Zeichenkette liefert eine neue Zeichenkette
  - Unicode-Zeichen im Format UTF-16
- Zeichenketten können in doppelte " " oder in einfache ' ' Anführungszeichen gesetzt werden
- JavaScript kennt keinen Datentyp für einzelne Zeichen (Character)
  - Einzelne Zeichen werden als Zeichenkette der Länge 1 gespeichert

Anführungszeichen

- Zeichenketten in doppelten Anführungszeichen dürfen einfache Anführungszeichen enthalten und umgekehrt
  - `"<span class='bold'>JavaScript</span>"`
  - `'<span class="heavy">JavaScript</span>'`
- Sollen die gleichen Anführungszeichen benutzt werden, in die die Zeichenkette gesetzt wurde, so ist dies mit einem umgekehrten Schrägstrich \ möglich
  - `"<span class=\"heavy\">JavaScript</span>"`
  - `'<span class=\'heavy\'>JavaScript</span>'`

Vergleich von Zeichenketten

- Zeichenketten lassen sich zeichenweise mit den Operatoren **==** und **===** vergleichen
  - Dabei wird Groß- und Kleinschreibung berücksichtigt
- Beispiel:

```
var strA = "Example String";
var strB = "Example String";

if (strA === strB)
  writeln("String A und B: identisch");
else
  writeln("String A und B: verschieden");

// Ausgabe: String A und B: identisch
```

Zeichenkettenobjekte

- Zeichenketten können mit **`new`** explizit als Objekt erzeugt werden
- Beispiel: **`var str = new String("Example String");`**
  - Sogenanntes Wrapper-Object: "Verbindungsobjekt" zwischen dem primitiven und dem abstrakten Datentyp
  - Erlaubt Zugriff auf Verarbeitungsmethoden
- Explizites Erzeugen vermeiden: Verringert Ausführungsgeschwindigkeit des Codes
- Laufzeitumgebung erzeugt automatisch temporäre Wrapper-Objekte, wenn eine String-Methode aufgerufen wird
  - Wrapper-Objekt wird nach Methodenaufruf wieder freigegeben

Zeichenkettenliterale

- Die folgende Schreibweise erzeugt ein sogenanntes Zeichenkettenliteral:

```
var str = "Example String";
```

- Ein Feld von 16-Bit-Werten
- Über die automatische Erstellung der Wrapper-Objekte stehen für Zeichenkettenliterale die gleichen Verarbeitungsmethoden zur Verfügung, wie für Zeichenkettenobjekte

Zeichenkettenobjekt und -literal: Vergleich (1)

- Hinsichtlich des Vergleichs verhalten sich Zeichenkettenliterale und -objekte beim strengen Vergleich `===` unterschiedlich
- Beispiel:

```
var strA = "Example String"; //String-Literal
var strB = new String("Example String"); //Objekt

if (strA === strB)
  writeln("String A und B: identisch");
else
  writeln("String A und B: verschieden");

// Ausgabe: String A und B: verschieden
```

Zeichenkettenobjekt und -literal: Vergleich (2)

- Demgegenüber wird beim einfachen Vergleich **==** nur der Ergebniswert und nicht der Typ verglichen
- Beispiel:

```
var strA = "Example String"; //String-Literal
var strB = new String("Example String"); //Objekt

if (strA == strB)
  writeln("String A und B: identisch");
else
  writeln("String A und B: verschieden");

// Ausgabe: String A und B: identisch
```

Zeichenketten: Eigenschaften & Methoden

- Die `String`-Wrapperklasse stellt viele Methoden und Informationen für Standardaufgaben zur Verfügung
  - Diese gelten gleichermaßen für Stringobjekte und Stringliterale

- Die Länge eines Strings lässt sich mit Hilfe der Eigenschaft `length` bestimmen
  - `length` liefert die Anzahl von Zeichen im String

- Beispiel:

```
var str = "Example String";

writeln(str.length); // Ausgabe: 14
```

Groß- und Kleinschreibung ändern

- `toLowerCase()`: Wandelt alle Zeichen im String in kleine Schriftzeichen um
- `toUpperCase()`: Wandelt alle Zeichen im String in große Schriftzeichen um
- Beispiel:
- ```
  var str = "Example String";

  var newStr = str.toLowerCase();

  // newStr -> "example string"
  ```
  ```

Strings modifizieren

- Strings sind unveränderbar
- Es wird immer eine modifizierte Kopie des Originals zurückgegeben
- Allgemeiner Ansatz:

```
modStr = origStr.operation(par1,...,parN);
```

- Übersicht zu Zeichenkettenmethoden, siehe:
  - **http://www.w3schools.com/js/js_strings.asp**

Strings modifizieren: `trim`

- Führende oder abschließende Leerzeichen beseitigen:

  `trim()`-Methode

- Beispiel:

```
var str = "      745    ";
var fixedStr = str.trim();
```

- Der neue String `fixedStr` enthält nur noch `"745"`

Strings modifizieren: `replace`

- Teilzeichenkette im String ersetzen:

  `replace(oldString, newString)`

  - Erzeugt einen neuen String, in dem das *erste* Vorkommen von **`oldString`** durch **`newString`** ersetzt wurde
  - Ersetzen aller Vorkommen durch Verwenden eines regulären Ausdrucks möglich
  - Groß- und Kleinschreibung wird beim Ersetzen berücksichtigt
  - Siehe hierzu auch:
    `www.w3schools.com/jsref/jsref_replace.asp`

`replace`: Beispiele

- ```
  var str1 = "Äpfel! Äpfel und Birnen im Angebot!";
  var str2 = str1.replace("Äpfel","Bananen");
  ```
- Liefert für `str2`:

  ```
  "Bananen! Äpfel und Birnen im Angebot!"
  ```
- Regulärer Ausdruck:

  ```
  var str3 = str1.replace(/Äpfel/g,"Bananen");
  ```
- Liefert für `str3`:

  ```
  "Bananen! Bananen und Birnen im Angebot!"
  ```
  ```

In Strings suchen: `indexOf`

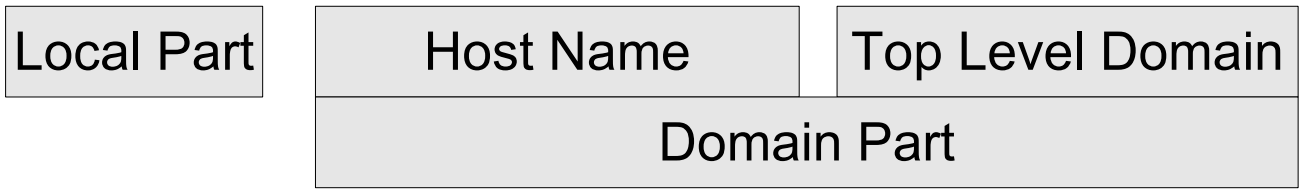
- Prüfen, ob eine Teilzeichenkette in einer anderen Zeichenkette enthalten ist
- **`indexOf(searchStr, startIndex)`**

  `lastIndexOf(searchStr, startIndex)`
  - Liefert Position des Anfangszeichens des ersten bzw. letzten Vorkommens der Zeichenkette **`searchStr`** im zugrundeliegenden String
  - Das erste Zeichen im String besitzt die Position 0
  - Ist **`searchStr`** nicht im zugrundeliegenden String enthalten, wird -1 zurückgegeben
  - Ist die Position **`startIndex`** angegeben, so startet die Suche ab dieser Position, sonst ab Position 0 bzw. length-1
  - Groß- und Kleinschreibung wird bei der Suche berücksichtigt

`indexOf`: Beispiel

- Prüfen einer Email-Adresse:

```
var str = "brause@wasserundsaft.de";
```




```
writeln("'@': " + str.indexOf("@"));
writeln("'com': " + str.indexOf("com");
```

- Ausgabe:

```
'@': 6
'com': -1
```

In Strings suchen: `search`

- Neben **`indexOf`** existiert auch die Methode **`search(searchStr)`**
  - Suchanfragen können hier auch in Form regulärer Ausdrücke formuliert werden
  - Dafür gibt es keine Startposition der Suche, wie bei **`indexOf`** bzw. **`lastIndexOf`**
    - Suchalgorithmus mächtiger, dafür nicht so schnell
    - Verhalten ansonsten analog zu **`indexOf`** bzw. **`lastIndexOf`**
  - Beispiele:
    - **`str1.search("Birnen");  // Zeichenkette`**
    - **`str1.search(/Birnen/); // Regulärer Ausdruck`**
    - Beides liefert entweder Indexwert oder -1

Teilzeichenkette auslesen

- **`substr(startIndex, length)`**
  - Liefert Teilzeichenkette mit der Länge **`length`**, die an der Position **`startIndex`** anfängt
  - Beispiel:

```
var str = "xy@hs-ulm.de";
str.substr(3); // returns "hs-ulm.de"
```

- **`substring(startIndex, endIndex)`**
  - Liefert Teilzeichenkette, die an der Position **`startIndex`** anfängt und bis **`endIndex-1`** reicht
  - Beispiel: **`str.substring(0,2); // returns "xy"`**
- Ohne **`length`** bzw. **`endIndex`** wird die restliche Teilzeichenkette ab **`startIndex`** geliefert

String zerlegen

- Zeichenkette an der Stelle **`cutter`** aufspalten:

  `split(cutter)`

  - Erzeugt ein **Feld** von Teil-Strings

- Beispiel:

```
var expression = "12.3 + 3*x + sin(y)";
var subexp = expression.split(" + ");
writeln(subexp[0]);
writeln(subexp[1]);
writeln(subexp[2]);
```

- Ausgabe:

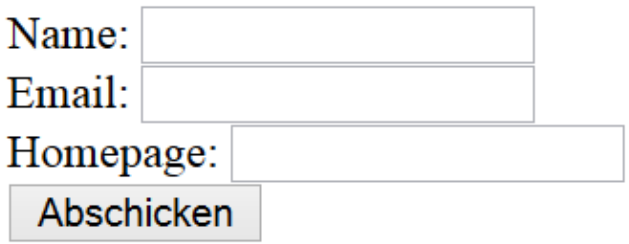
```
12.3
3*x
sin(y)
```

Formularüberprüfung: Einfache Visitenkarte

```
<form onsubmit="return check()">
Name: <input type="text" id="name" name="name"><br>
Email: <input type="email" id="email" name="email"><br>
Homepage: <input type="url" id="hpage" name="hpage"><br>
<input type="submit" value="Abschicken">
</form>

<output id="output"></output>
```

- **`id`**: Für Zugriff auf das Formularelement unter JavaScript
- **`name`**: Für Verarbeitung des Elementwerts nach dem Abschicken im Browser oder in einem Webserver
- Rückgabewert der Funktion **`check()`** bestimmt, ob Formulardaten abgeschickt werden oder nicht



Einfache Visitenkarte: Script (1)

```
<script>

function check() {

  document.getElementById("output").innerHTML = "";

  var email = document.getElementById("email").value;
  var hpage = document.getElementById("hpage").value;
  var valid = true;

  if (email.indexOf(".") < 0) {
    document.getElementById("output").innerHTML =
      "Email-Adresse ungültig: " + email + "<br>";
    valid = false;
  }

  //...
```

Einfache Visitenkarte: Script (2)

```
  //...

  if (hpage.indexOf(".") < 0) {
    document.getElementById("output").innerHTML +=
      "URL ungültig: " + hpage + "<br>";
    valid = false;
  }

  return valid;

}
</script>
```

- Liefert **`valid`** den Wert **`true`**, so werden die Formulardaten abgeschickt, sonst nicht

URL-Querystring

- Versand der Fomulardaten voreingestellt als URL-Querystring:

```
http://localhost:8383/.../Visitenkarte.html?
name=Markus+Mustermann&email=markus
%40mustermann.de&hpage=http%3A%2F
%2Fwww.mustermann.de
```

  - Formulartag ohne Zielangabe: Der Versand geschieht an die Formulardatei selbst. Hier: **`Visitenkarte.html`**
  - Variablennamen im Querystring entsprechen den Namen der Formularelemente
    - Fehlt Angabe eines Namens, wird der Wert nicht übertragen
  - Die Auswertung der Daten erfolgt bei diesem Beispiel durch den Webbrowser mittels JavaScript

URL-Querystring: Auswertung im Browser

- Zunächst im Scriptteil der Formulardatei die Zeichenkette **`window.location.href`** auswerten
- Beispiel:

```
<script>
if (window.location.href.indexOf("?") > -1)
  document.write("Formulardaten erhalten: " +
    window.location.href.substr(
      window.location.href.indexOf("?")+1) +
      "<br>");
</script>
```

- Variablenwerte können mit Hilfe von Zeichenkettenbefehlen ausgewertet werden

URL-Querystring: Variablen auslesen

- Auswertung mit **`split()`** und Schleife:

```
if (window.location.href.indexOf("?") > -1) {
  var querystring = window.location.href.split("?");
  var entries = querystring[1].split("&");
  for (var i in entries) {
    var entry = entries[i].split("=");
    document.write(entry[0]+": "+entry[1]+"<br>");
  }
}
```

Ausgabe:

```
name: Markus+Mustermann
email: markus%40mustermann.de
hpage: http%3A%2F%2Fwww.mustermann.de
```

URL-Querystring: Decodierung

- Erfolgt mit **`decodeURIComponent()`**:

```
if (window.location.href.indexOf("?") > -1) {
  var querystring = window.location.href.split("?");
  var entries = querystring[1].split("&");
  for (var i in entries) {
    var entry = entries[i].split("=");
    var name = decodeURIComponent(entry[0]);
    var val = decodeURIComponent(entry[1]);
    document.write(name + ": " + val + "<br>");
  }
}
```

- **`URI`**: Universal Resource Identifier

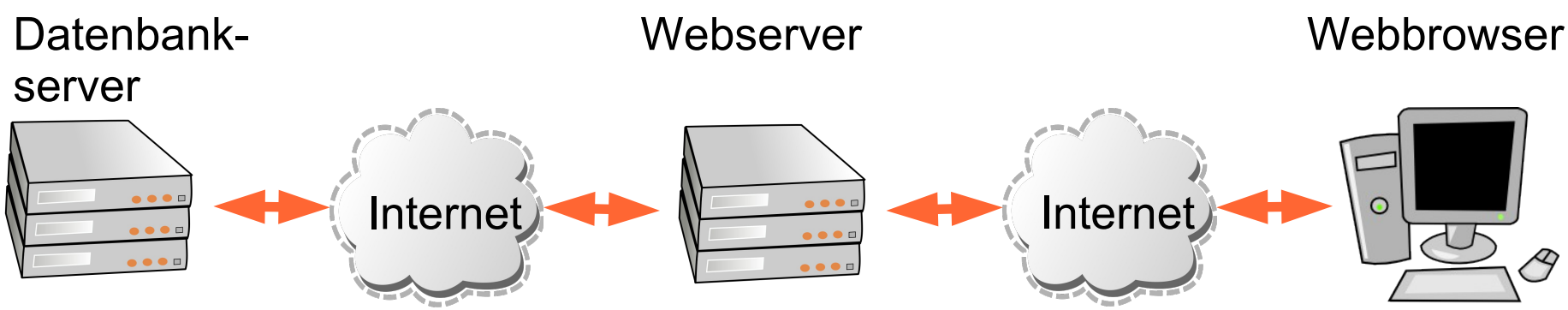
Ausgabe:

```
name: Markus+Mustermann
email: markus@mustermann.de
hpage: http://www.mustermann.de
```

Templates

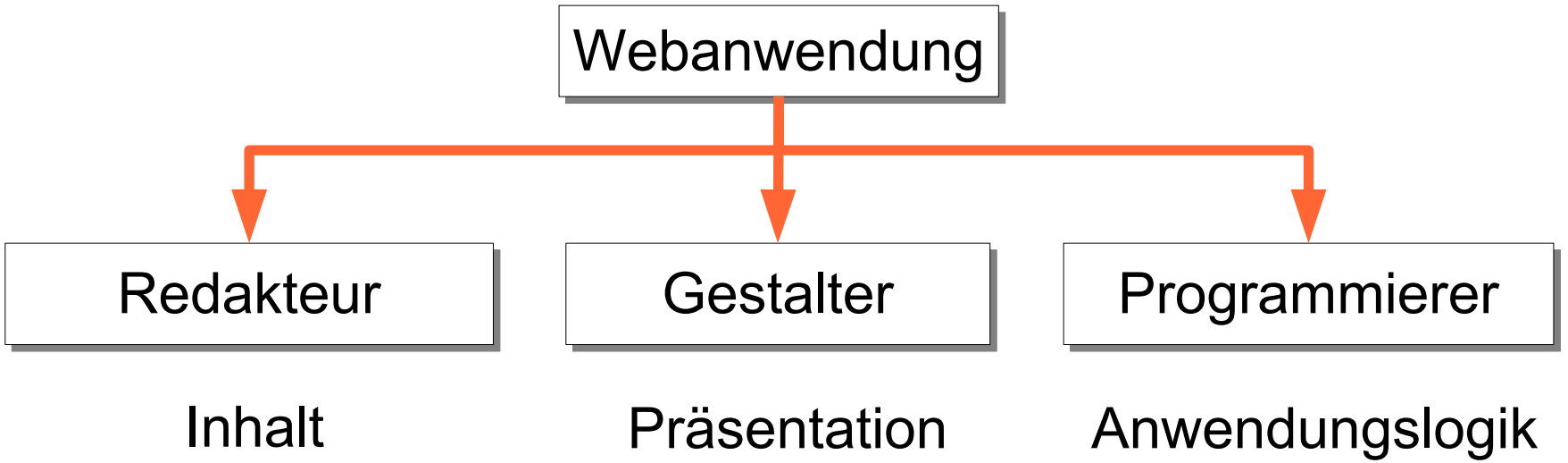
AJAX

- Webseiten verarbeiten im Hintergrund extern eingelesene Daten
  - Es gilt nicht mehr das *Seitenwechselparadigma*: Der Inhalt der Seite ändert sich, ohne dass diese neu geladen werden muss
  - Es gilt das *Anwendungsparadigma*: Die Webseite verhält sich wie ein Desktopprogramm
  - Es ist möglich, mit der Anwendung weiter zu interagieren, während neue Daten vom Server geladen werden



Webanwendung

- Webanwendungen erfordern die Zusammenarbeit von Spezialisten unterschiedlicher Disziplinen



- Die Möglichkeit eines asynchronen Arbeitens an den HTML-, CSS- und JavaScript-Inhalten ist daher wünschenswert

MVC

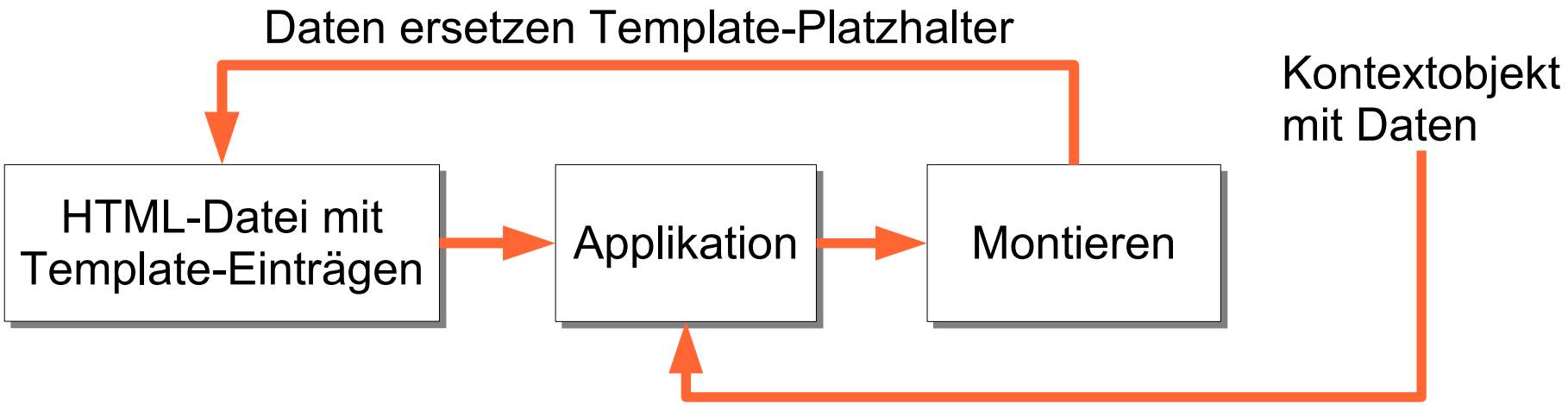
- MVC-Designmuster
- Model-View-Controller („Modell-Präsentation-Steuerung“)
  - Eine Anwendung besteht aus dem Datenmodell (Modell), dem Präsentationsteil (View) und der Programmsteuerung (Controller), die Model und View miteinander verbindet
- Impliziert die Möglichkeit, diese Programmteile unabhängig voneinander zu entwickeln und zu pflegen
- Die verschiedenen Codebestandteile werden hierfür soweit wie möglich voneinander getrennt

Templates

- Templates erlauben die Erstellung von Schablonen, die das Layout oder Layoutbestandteile von Webseiten repräsentieren
  - Templates begünstigen hierdurch ein einheitliches Layout
- Die Schablonen enthalten Platzhalter, die bei der dynamischen Erzeugung der Webseite durch Inhalte ersetzt werden
- Die Inhalte stammen häufig aus JSON-Dateien oder Datenbanken
- Vorlagen bzw. Templates unterstützen daher auch die Trennung zwischen Layout und Inhalten
- Für die Umsetzung von Templates existieren unterschiedliche Frameworks, u. a. Handlebars und Vue.js

Templateverarbeitung mit Vue.js

- https://vuejs.org/



Zu jedem Template
wird eine sogenannte
Applikation erzeugt

Applikation ersetzt
beim Montieren die Platzhalter
im Template-HTML durch die
übergebenen Kontext-Daten

Bsp: Dynamisch eingefügter Benutzername (1)

- **Schritt 1:** Vue einbinden:

```
<head>
  <!--…-->
  <script src="vue.global.min.js"></script>
</head>
```

- **Schritt 2:** Templates jeweils an passender Stelle als HTML-Element mit Platzhaltern in den HTML-Body einfügen:
- `<div id="tpl">{{firstname}} {{lastname}}</div>`
  - Platzhalter werden durch `{{}}` ausgezeichnet („Mustache“-Notation)
  - Werte gleichnamiger Kontextvariablen ersetzen später Platzhalter

Bsp: Dynamisch eingefügter Benutzername (2)

- **Schritt 3:** Template verarbeiten

```
<script>
1. const { createApp } = Vue;

2. createApp(
3. {
4.   data() {
       return {
         firstname: "Mark",
         lastname: "Muster"
       }
     }
5. }).mount('#tpl');
</script>
```

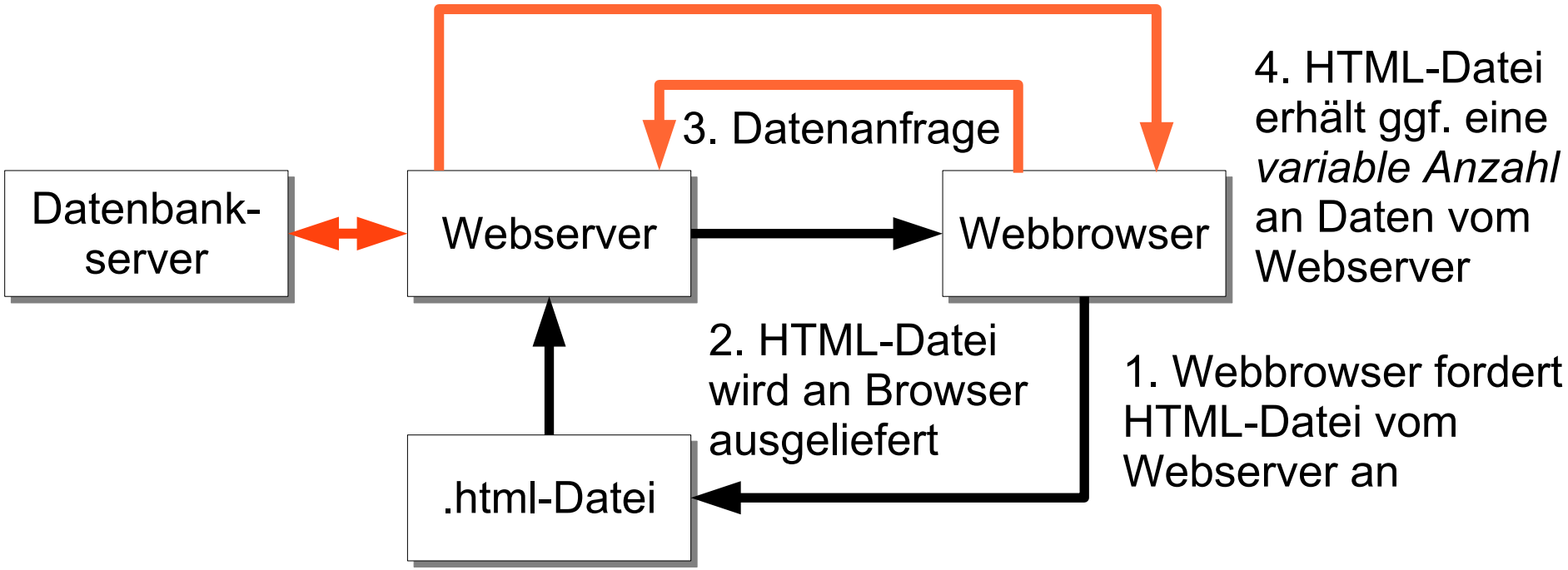
1. Konstruktorfunktion **`createApp`** aus Vue destrukturieren
2. Applikation erzeugen
   - Standardvorgang bei Vue
3. Objektliteral für die Daten ist Parameter für das Applikationsobjekt
4. Daten werden durch Methode **`data()`** geliefert
5. Durch Aufruf von **`mount()`** werden die Daten in das HTML-Element mit der passenden ID geschrieben

Vollständiges Beispiel

```
<html>
  <head>
    <meta charset="UTF-8">
    <script src="vue.global.min.js"></script>
  </head>
  <body><!-- JavaScript-frei -->
    <div>Template Example</div>
    <div id="tpl">{{firstname}} {{lastname}}</div>
    <script><!-- HTML-frei => kann in externe JS-Datei -->
      const { createApp } = Vue;
      createApp({data() {
        return {
            firstname: "Mark", lastname: "Muster"
        }}}).mount('#tpl');
    </script>
  </body>
</html>
```

Webanwendung: AJAX-Datenaustausch

- Externe JSON-Dateien oder Datenbankanfragen können eine variable Anzahl von Daten liefern
  - Z. B. eine Tabelle mit variabler Anzahl von Zeilen
  - Folge: Die Anzahl der Einträge im HTML-Kontext des zugehörenden Templates variiert



Variable Anzahl von Daten

- Templates müssen in der Lage sein, in Abhängigkeit von der Anzahl vorliegender Daten eine passende Anzahl von HTML-Elementen zu erzeugen
- Beispiele
  - Eine Tabelle oder Liste mit variabler Anzahl von Zeilen
  - Optionale Tabellen- oder Listeneinträge
- Zu diesem Zweck besitzt Vue spezielle Direktiven
  - Mit ihrer Hilfe können HTML-Elemente in Abhängigkeit von einer logischen Bedingung oder auch mehrfach im Rahmen einer Iteration erzeugt werden

Direktiven

- **`v-for`**: Inhalt wird entsprechend einer gegebenen Anzahl von Feldelementen gerendert
- **`v-if`**: Inhalt wird nur bei zutreffender Bedingung gerendert
- **`v-else`**: Definiert eine Alternative zu einem vorhergehenden **`v-if`**
- **`v-else-if`**: Definiert ein else-if zu einem vorhergehenden **`v-if`**
- **`v-show`**: Wie **`v-if`**. Inhalt wird aber immer in das DOM gerendert und bei Bedarf über **`display`** sichtbar gemacht
- **`v-html`**: Dient dazu, das innere HTML (**`innerHTML`**) eines Elements mit Daten zu beschreiben
- **`v-text`**: Wie **`v-html`** für **`textContent.`** Alternative zu Mustache-Notation

Beispiel: Gästeliste ausgeben

```
<div id="app">
  <p v-for="guest in guests">
    {{guest.firstname}} {{guest.lastname}}
  </p>
</div>

<script>
  createApp({
    data() {
      return {guests:[
            { firstname: 'Mark',  lastname: 'Muster'},
            { firstname: 'Petra', lastname: 'Pattern'},
            { firstname: 'Susan', lastname: 'Sample'}
      ]}
    }
  }).mount('#app');
</script>
```

Beispiel: Gästeliste ausgeben mit `v-html`

```
<div id="app">
  <p v-for="guest in guests">
    <span v-html="guest.firstname"></span>
    <span> </span>
    <span v-html="guest.lastname"></span>
  </p>
</div>

<script>
  createApp({
    data() {
      return {guests:[
            {firstname: 'Mark<br>',  lastname: 'Muster'},
            {firstname: 'Petra<br>', lastname: 'Pattern'},
            {firstname: 'Susan<br>', lastname: 'Sample'}
      ]}
    }
  }).mount('#app');
</script>
```

Zusätzliches HTML wird verarbeitet

v-if: Beispiel

```
<div id="app">
  <p v-for="guest in guests">
    {{guest.firstname}} {{guest.lastname}}
    <span v-if="guest.age">, {{guest.age}}</span>
    <span v-else>, keine Altersangabe</span>
  </p>
</div>
<script>
  createApp({ data() { return { guests:[
    {firstname:"Mark",  lastname:"Muster", age: 32},
    {firstname:"Petra", lastname:"Pattern"},
    {firstname:"Susan", lastname:"Sample", age:24}
  ]}}}).mount('#app');
</script>
```

- Falls im Kontext die Variable **age** vorhanden ist, wird deren Inhalt angezeigt, sonst eine Standardmeldung

Komposition

- Vue erlaubt den Zugriff auf eingeschachtelte Objekte
- Beispiel:

```
createApp({
  data() {
    return {guests:[
      {firstname:"Mark", lastname:"Muster",
       address: {
         location:"Munich",
         street:"Maximilianstr.",
         no:"1"}
      },
      ...usw.
    ]}
  }
}).mount('#app');
```

Zugriff auf eingeschachtelte Objekte

- Der Zugriff erfolgt im Template mit Hilfe des Punktoperators
- Beispiel:

```
<div id="app">
  <p v-for="guest in guests">
    {{guest.firstname}} {{guest.lastname}}:
    {{guest.address.location}},
    {{guest.address.street}}
    {{guest.address.no}}
  </p>
</div>
```

HTML erweitern

HTML: Beschränkungen

- HTML
  - Menge der vorhandenen Elemente ist in einer gegebenen HTML-Version beschränkt und nicht erweiterbar
  - Neue Elemente müssen einen Standardisierungsprozess durchlaufen
  - Nur Elemente, die eine breite Unterstützung erlangen, schaffen es schließlich in eine neue Version des Standards
- Die Erfordernisse der Webentwicklung wachsen schneller, als der Standard folgen kann
- Neue Anforderungen werden deshalb häufig mit Hilfe einer Kombination aus **`<div>`**-Element + CSS + JavaScript umgesetzt
- Nachteil: HTML-Seiten, die aus "**`<div>`**-Wüsten" bestehen

HTML + CSS

- Formatierter Absatz mit blauem, fett gesetzten Text:

```
<p style="color:blue; font-weight:bold">
  Blauer, fetter Text
</p>
```

- Weitere Absätze erfordern Wiederholung der Formatierungsanweisungen. Alternative: CSS-Klasse definieren:

```
<style>
  p.blueBold {color:blue; font-weight:bold;}
</style>
<p class="blueBold">Blauer, fetter Text</p>
```

- Das HTML-Element und die zugehörende CSS-Klasse existieren unabhängig voneinander
- In beiden Fällen erschließt sich die Semantik des Elements nicht aus der Syntax

CSS: Nachteile

- CSS-Befehle sind global
  - Gleichnamige Klassen können Seiteneffekte erzeugen
- CSS wächst unaufhörlich
  - Jeder Spezialfall erfordert eine eigene Klasse
  - Die Referenzierung von CSS-Klassen bzw. -Befehlen erfolgt verdeckt
    - Nach einer gewissen Zeit ist es schwer zu entscheiden, welche der Klassen bzw. Befehle aktiv genutzt wird
  - Folge: Es ist schwierig CSS-Befehle zu löschen: Die möglichen Seiteneffekte sind nicht abschätzbar

Mögliche Lösungen

- Vermeidung globaler CSS-Direktiven
  - Eine Lösung: Inline-Styles. D. h., erforderliche Styles werden in dem betreffenden HTML-Element notiert
  - Nachteile:
    - Codeduplikation
    - Vermischung von HTML + CSS
- Besser: Inline-Styles als Teil einer HTML-Komponente definieren
  - Die gewünschten Styles werden parametrierbare Eigenschaften der Komponente
  - Codeduplikation wird dabei vermieden
- Komponententechnologien erlauben das Einkapseln von Styles
  - Gekapselte Lösungen erleichtern die Codewiederverwendbarkeit und vermeiden Seiteneffekte

Lösungen weiterverwendbar machen

- Die bislang behandelten Lösungen zur Entwicklung eigener Bedienelemente basieren beispielsweise auf getrennten HTML-, CSS- und JavaScript-Inhalten (z. B. Modales Bild)
- Die Lösungen liegen nicht in gekapselter Form vor, die leicht weitergereicht und -verwendet werden können
- Eine Kapselung lässt sich durch Umsetzen der Bedienelemente in Form von Komponenten erreichen

Komponenten: Allgemeines

- Komponenten bieten eine Möglichkeit, den Sprachumfang von HTML durch eigene Elemente zu erweitern ("Custom Tags")
  - Der Sprachumfang des HTML-Standards bleibt davon unberührt
- Die neuen HTML-Elemente werden entsprechend ihrer Funktion benannt
  - Ausdruckslose "`<div>`-Wüsten" werden vermieden
- Ihre Implementierung bringen die Komponenten selber mit
  - Diese liegt in gekapselter Form vor und kann leicht weitergereicht werden

HTML durch neue Elemente erweitern

- Es besteht bereits eine Möglichkeit, neue HTML-Elemente zu definieren
- Es ist z. B. möglich, in einer HTML-Datei ein willkürlich benanntes neues Element zu verwenden
  - Beispiel: **`<newelem>...</newelem>`**
  - Dieses Element kann mit CSS formatiert werden
  - Das Element besitzt aber keine zugeordnete Funktionalität
- Der Browser behandelt es als **`HTMLUnknownElement`**
  - Vorgabe für unbekannte Elemente, damit diese keinen Abbruch bei der Interpretation einer Webseite verursachen
- Mit Hilfe von JavaScript kann dem unbekannten Element eine eigene Funktionalität hinzugefügt werden

Beispiel: Eigene Absatzkomponente

```
<p>Normaler Absatz</p>
<p-blue-bold> <!-- Blau getönter Absatz -->
  Dies ist ein blau getönter Absatz.
</p-blue-bold>
<p-blue-bold>
  Dies ist ein weiterer blau getönter Absatz.
</p-blue-bold>
<p-blue-bold>
  <i>Blauer kursiver Text.</i>
  <p>Eingekapselter Absatz.</p>
</p-blue-bold>
```

- Konvention: Neue HTML-Elemente werden mit Bindestrich-Notation geschrieben

JavaScript für blau getönten Absatz

- Code selektiert passende HTML-Elemente und formatiert diese dynamisch
-
```
(function() {
  var pblues =
    document.getElementsByTagName("p-blue-bold");
  for (let i=0; i<pblues.length; i++) {
    pblues[i].style.color = "blue";
    pblues[i].style.fontWeight = "bold";
    pblues[i].style.display ="block";
    pblues[i].style.lineHeight = "200%";
  }
}());
```

- Code in externe JavaScript-Datei schreiben und diese z. B. mittels **`defer`** einbinden

Beispiel: W3Schools-Button

```
<head>
  <script>
    function handleClick() {
      document.getElementById("output").innerHTML =
        "Button clicked"

    }
  </script>
</head>

<body>
  <button class="button" onclick="handleClick()">
    Click
  </button>
  <div id="output"></div>
</body>
```

W3Schools-Button: CSS (1)

```
<!-- siehe http://www.w3schools.com/css/css3_buttons.asp -->
<!DOCTYPE html>
<html>
<head>
<style>
.button {
  display: inline-block;
  padding: 15px 25px;
  font-size: 24px;
  font-family: sans-serif;
  cursor: pointer;
  text-align: center;
  text-decoration: none;
  outline: none;
  color: #fff;
  background-color: #4CAF50;
  border: none;
  border-radius: 15px;
  box-shadow: 0 9px #999;
}
```

W3Schools-Button: CSS (2)

```
<!DOCTYPE html>
<html>
<head>
<style>

  <!-- siehe vorige Folie -->

  .button:hover {background-color: #3e8e41}

  .button:active {
    background-color: #3e8e41;
    box-shadow: 0 5px #666;
    transform: translateY(4px);
  }
</style>

</head>
```

W3Schools-Button als Komponente

```
<script>
function handleClick() {
  document.getElementById("output").innerHTML =
    "Button clicked"
}
</script>
```

```
<press-button onclick="handleClick()">
  Click
</press-button>
```

- Konvention bei Komponententechnologien: Tag-Namen werden mit Bindestrich geschrieben, um sie von echten HTML-Elementen zu unterscheiden

```
<div id="output"></div>
```

Dynamische Formatierung (1)

```
(function() {
  var pbarr = document.getElementsByTagName("press-button");
  for (let i=0; i<pbarr.length; i++) {
    pbarr[i].style.display = "inline-block";
    pbarr[i].style.padding = "15px 25px";
    pbarr[i].style.marginBottom = "20px";
    pbarr[i].style.fontSize = "24px";
    pbarr[i].style.fontFamily = "sans-serif";
    pbarr[i].style.cursor = "pointer";
    pbarr[i].style.textAlign = "center";
    pbarr[i].style.textDecoration = "none";
    pbarr[i].style.outline = "none";
    pbarr[i].style.color = "#fff";
    pbarr[i].style.backgroundColor = "#4CAF50";
    pbarr[i].style.border = "none";
    pbarr[i].style.borderRadius = "15px";
    pbarr[i].style.boxShadow = "0 9px #999";
                              ⋮
```

Dynamische Formatierung (2)

- Alternative: CSS mit Hilfe einer Zeichenkette zuweisen

```
(function() {
  var pbarr = document.getElementsByTagName("press-button");
  for (let i=0; i<pbarr.length; i++) {
    pbarr[i].style = "display: inline-block;\
      padding: 15px 25px; marginBottom: 20px;\
      font-size: 24px; font-family: sans-serif;\
      cursor: pointer; text-align: center;\
      text-decoration: none; outline: none;\
      color: #fff; background-color: #4CAF50;\
      border: none; border-radius: 15px;\
      box-shadow: 0 9px #999";
                          ⋮
```

- Die Gültigkeit der CSS-Anweisungen ist auf die Komponente beschränkt (Scoped-CSS)

Dynamische Formatierung (3)

- Kein Zugriff auf CSS-Pseudoklassen unter JavaScript möglich. Folgendes funktioniert daher nicht:

```
pbarr[i].style.hover.backgroundColor = "#3e8e41";
pbarr[i].style.active.backgroundColor = "#3e8e41";
```

- Alternative Lösung: Ereignisbehandler

```
pbarr[i].onmouseover = function () {
  pbarr[i].style.backgroundColor = "#3e8e41";
}
pbarr[i].onmouseout = function () {
  pbarr[i].style.backgroundColor = "#4CAF50";
}
```

Dynamische Formatierung (4)

- Status gedrückt und ungedrückt mittels Ereignisbehandler:

```
    pbarr[i].onmousedown = function () {
      pbarr[i].style.backgroundColor = "#3e8e41";
      pbarr[i].style.boxShadow = "0 5px #666";
      pbarr[i].style.transform = "translateY(4px)";
    }

    pbarr[i].onmouseup = function () {
      pbarr[i].style.backgroundColor = "#4CAF50";
      pbarr[i].style.boxShadow = "0 9px #999";
      pbarr[i].style.transform = "translateY(0px)";
    }
  }

}());
```

W3Schools-Button mit zwei Zuständen

- Button verbleibt im eingeschalteten Zustand mit roter Beschriftung, bis er durch ein nochmaliges Drücken wieder ausgeschaltet wird

```
(function() {
  var pbarr = document.getElementsByTagName("press-button");
  for (let i=0; i<pbarr.length; i++) {
    pbarr[i].switchon = true;
                        ⋮ (wie zuvor)
    pbarr[i].onmousedown = function () {
      // wie zuvor
      if (pbarr[i].switchon) {
        pbarr[i].switchon = false;
        pbarr[i].style.color = "red";
      }
      else {
        pbarr[i].switchon = true;
        pbarr[i].style.color = "white";
      }
  } // usw.
```

Entwicklungsumgebungen für Komponenten

Komponententechnologien (1)

- Neben der Möglichkeit, eigene Lösungen zu programmieren, können auch existierende Komponententechnologien genutzt werden
- Vue bietet hierfür z. B. zwei Möglichkeiten (zusätzlich zu Templates)
  - Web Components
  - Vue Components
- Weitere Lösungen: Google Polymer, React (Facebook), X-Tag (Mozilla, Microsoft), Riot

Komponententechnologien (2)

- Wichtige Merkmale
  - Custom Elements: Selbst geschriebene Komponenten bzw. Elemente, die wie alle anderen HTML-Elemente auch benutzt und formatiert werden können
  - Shadow DOM: Ein spezieller DOM-Bereich für die Darstellung der neuen Elemente
    - Kapselt den zur Komponente gehörenden Teil des DOM zusammen mit den zugehörenden CSS-Befehlen
    - Ermöglicht Gültigkeitsbereiche für CSS
    - Vermeidet Seiteneffekte durch gleichlautenden CSS-Code
    - Extern definiertes CSS kann nicht in die Komponente diffundieren und so ihr Aussehen ändern
    - Für die Komponente definiertes CSS beeinflusst nicht den Seiteninhalt

Beispiel: Vue-Web-Komponente

- JavaScript für ein neues HTML-Element definieren:

```
const { defineCustomElement } = Vue;

const PBlueBoldElement = defineCustomElement({
  template: "<div class='text'><slot></slot></div>",
  styles: [".text {color: blue;font-weight:bold}"]
})
```

- Element einen Tag-Namen zuweisen:

```
customElements.define('p-blue-bold',PBlueBoldElement);
```

- Später vorhandener Elementinhalt wird mittels `<slot />` eingebettet
- HTML:

```
<p-blue-bold>
    Blauer fetter Text
</p-blue-bold>
```

Elementinhalte

- Elementinhalte werden durch **<slot/>** eingebettet
- Elementinhalt in der HTML-Datei darf beliebig viele HTML-Elemente enthalten: Diese werden einfach durchgereicht
- Beispiel:
-

```
<html>

<p-blue-bold>
  <i>Blauer, fetter, kursiver Text</i>
</p-blue-bold>

</html>
```

  - Das **<i>**-Element wird zusammen mit dem Text an der Position von **<slot/>** eingefügt

W3Schools-Button als Vue-Komponente (1)

- JavaScript:

```
const PressButton = defineCustomElement({
template: "<span class='button'><slot></slot></span>",
styles: [
  ".button {\
     display: inline-block;\
     padding: 15px 25px;\
     font-size: 24px; font-family: sans-serif;\
     cursor: pointer;\
     text-align: center; text-decoration: none;\
     outline: none;\
     color: #fff; background-color: #4CAF50;\
     border: none;\
     border-radius: 15px; box-shadow: 0 9px #999;\
   }\
                              :
```

W3Schools-Button als Vue-Komponente (2)


```
                                  ⋮
    .button:hover {background-color: #3e8e41}\

    .button:active {\
      background-color: #3e8e41;\
      box-shadow: 0 5px #666;\
      transform: translateY(4px);\
    }"
  ]
})

customElements.define(
  'press-button', PressButton);
```

- CSS ist eingekapselt
- Die Gültigkeit der CSS-Anweisungen ist auf die Komponente beschränkt (Scoped-CSS)
- Die globale CSS-Klasse `.button` ist nicht länger erforderlich

W3Schools-Button als Vue-Komponente (3)

```
<head><script>
  function bHandler(name) {
    document.getElementById("output").innerHTML =
      "Button clicked";
  }
</script></head>

<body>
  <press-button>Click</press-button>
  <script>
    var barr = document.getElementsByTagName("press-button");
    barr[0].addEventListener("click", bHandler);
  </script>
  <div id="output"></div>
</body>
</html>
```

Komponententechnologien: Riot

- Erweiterung von HTML durch Custom Tags
- Ziel: Möglichst geringe Komplexität
- Einfache Installation, einfache Syntax
- siehe https://riot.js.org/

Getönter Absatz als Riot-Element

- Datei **`p-blue-bold.tag`** erzeugen mit folgendem Inhalt:

```
<p-blue-bold>
  <p><yield /></p>
  <style>
    p {
      color:blue;
      font-weight:bold;
    }
  </style>
</p-blue-bold>
```

- CSS ist eingekapselt
- Die Gültigkeit der CSS-Anweisungen ist auf die Komponente beschränkt (Scoped-CSS)

- Später vorhandener Elementinhalt wird mittels **`<yield />`** eingebettet

Elementinhalte

- Elementinhalte werden durch **<yield/>** eingebettet
- Elementinhalt in der HTML-Datei darf beliebig viele HTML-Elemente enthalten: Diese werden einfach durchgereicht
- Beispiel:
-

```
<html>

<p-blue-bold>
  <i>Blauer, fetter, kursiver Text</i>
</p-blue-bold>

</html>
```

  - Das **<i>**-Element wird zusammen mit dem Text an der Position von **<yield/>** eingefügt

Verwendung des neuen Elements in HTML-Datei

```
<html>
  <head>
    <script src="riotpluscompiler.js"></script>
    <script type="riot/tag" src="p-blue-bold.tag">
    </script>
  </head>
  <body>
    <p-blue-bold>Blauer, fetter Text</p-blue-bold>
    <script>riot.mount('p-blue-bold')</script>
  </body>
</html>
```

- Neu-definierte Auszeichnungselemente müssen durch den Riot-Compiler zunächst in JavaScript übersetzt werden, damit der Webbrowser diese interpretieren kann
- Durch das Montieren wird der Code übersetzt und auf die Vorkommen des neuen Elements angewendet

Riot-Compiler

- Es gibt zwei Varianten:
  - **`riotpluscompiler.js`**, **`riot+compiler.js`**
    - In-browser compilation zur Auffrufzeit
  - **`riot.js`**
    - Pre-compilation, d. h. Code wurde bereits zuvor durch den Riot-Compiler übersetzt
    - Auf dem Server ist bereits fertiges Kompilat hinterlegt

Erstellung: Allgemeine Regeln

- Elemente können frei definierte Namen besitzen und bestehen aus HTML, CSS und JavaScript
-
```
<neues-element>
  ...
</neues-element>
```
- Code des Elements wird in einer eigenen Datei mit dem *korrespondierenden* Namen `neues-element.tag` gespeichert
- Konvention: Tag-Namen werden mit Bindestrich geschrieben, um sie von echten HTML-Elementen zu unterscheiden
  - Konvention entsprechend zu anderen Komponenten-Entwicklungsumgebungen

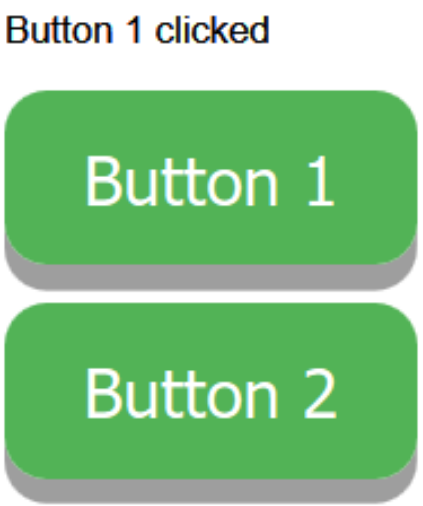
W3Schools-Button als Riot-Komponente

```
<press-button>
  <span><yield/></span>
  <style>
    span {
      display: inline-block;
      padding: 15px 25px;
      /* etc. */
    }
    span:hover {
      background-color: #3e8e41
    }
    span:active { /* etc. */ }
  </style>
</press-button>
```

- Steht in Datei **`press-button.tag`**
- CSS ist eingekapselt
- Die Gültigkeit der CSS-Anweisungen ist auf die Komponente beschränkt (Scoped-CSS)
- Die globale CSS-Klasse **`.button`** ist nicht länger erforderlich

Anwendung: HTML

- <head>
  <!-- Riot-Compiler + Pressbutton
  + JavaScript laden -->
  </head>



- <body>
  <output id="output">Output</output>
  <press-button onclick="f1()">Button 1
  </press-button>
  <press-button onclick="f2()">Button 2
  </press-button>
  <script>riot.mount('press-button')</script>
  </body>

Anwendung: JavaScript

```
function f1() {
  document.getElementById("output").innerHTML =
    "Button 1 clicked"
}

function f2() {
  document.getElementById("output").innerHTML =
    "Button 2 clicked"
}
```

Beispiel: Absatzelement

```
<html>
  <head>
    <!-- Inkludieren wie zuvor mit:
         absatz-element.tag -->
  </head>

  <body>
    <absatz-element>1. Absatz</absatz-element>
    <absatz-element text="2. Absatz"></absatz-element>
    <absatz-element></absatz-element>
    <script>riot.mount('absatz-element')</script>
  </body>
</html>
```

Ausgabe:

```
1. Absatz
2. Absatz
Leerer Absatz
```

Internes JavaScript

- Beispiel: Neues Absatzelement
- **<absatz-element>**

```
<p>{value}</p>
if (opts.text)
  this.value = opts.text;
else if (this.root._innerHTML)
  this.value = this.root._innerHTML;
else
  this.value = "Leerer Absatz";
```

</absatz-element>

- JavaScript-Code steht am Ende des Element-Codes oder in **<script>**-Tags
- Zugriff auf Attribute mit **opts**
- Zugriff auf Elementinhalt mit **this.root._innerHTML**

Riot-Element mit mehreren Attributen

- Beispiel: Element **<user-name>** für Benutzernamen
-
```
<head>
  <!-- Inkludieren wie zuvor mit:
       user-name.tag -->
</head>
```
-
```
<body>
  <user-name firstname="Mark"
             lastname="Muster"></user-name>

  <script>
    riot.mount('user-name');
  </script>
</body>
```

Ausgabe:

Mark Muster

Element <user-name>: Definition

- Inhalt der Datei **user-name.tag**:

```
<user-name>
  <p>{opts.firstname} {opts.lastname}</p>
</user-name>
```

  - Attributnamen dürfen nicht in CamelCaps geschrieben werden
  - Attributwerte werden dann nicht ausgewertet

Zusätzlich Elementinhalt

- `<head>`

```
  <script type="riot/tag" src="user-name.tag">
  </script>
</head>
```

- `<body>`

```
  <user-name firstname="Mark"
             lastname="Muster">
    Name:
  </user-name>
  <script>riot.mount('user-name');</script>
</body>
```

Ausgabe:

Name: Mark Muster

Interne Verarbeitung mit `<yield/>`

- Inhalt der Datei **`user-name.tag`** mit **`yield`**:

```
<user-name>
  <p><yield/> {opts.firstname} {opts.lastname}</p>
</user-name>
```

- Zusätzliche HTML-Elemente werden wieder durchgereicht
- Beispiel:

```
<user-name firstname="Mark"
           lastname="Muster">
  <p>Name:</p>
</user-name>
```

Ausgabe:

```
Name:
Mark Muster
```

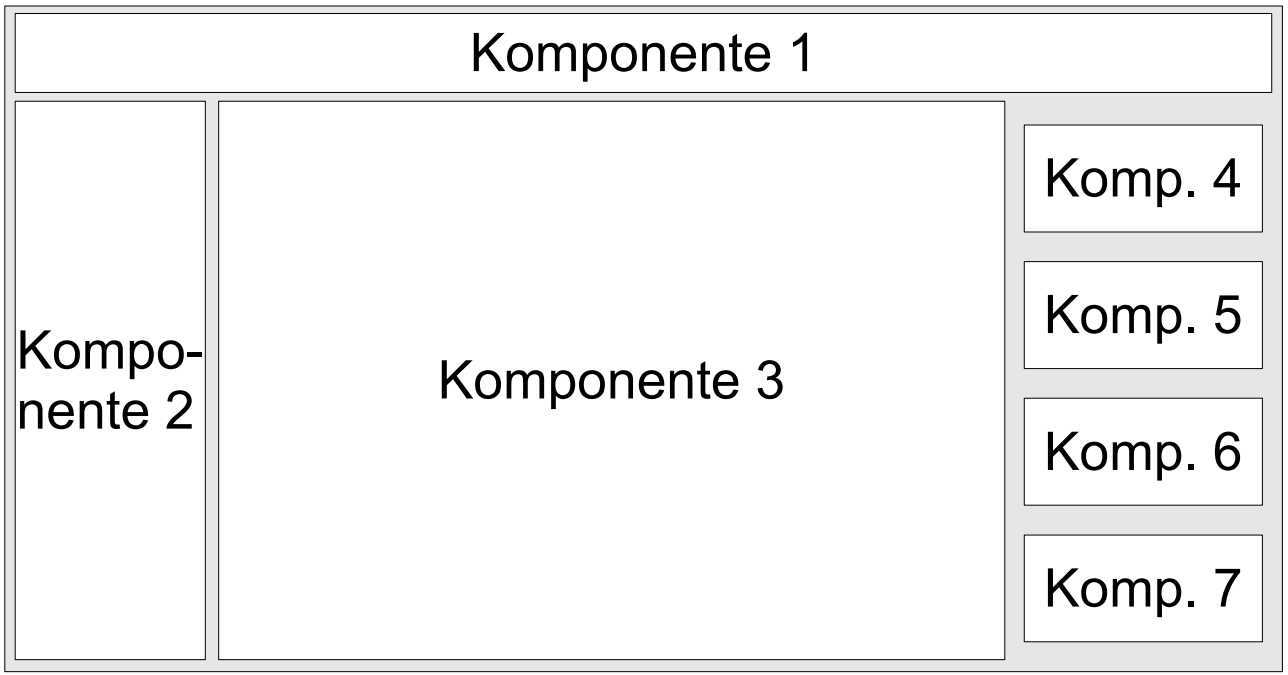
Übergabe der Daten beim Montieren

- Die darzustellenden Daten müssen nicht statisch in die HTML-Seite eingebettet werden
- Beispiel: Element **<user-name>** mit Übergabe der Daten mittels JavaScript

```
<body>
  <user-name></user-name>
  <script>
    riot.mount('user-name',
      {"firstname":"Mark", "lastname":"Muster"});
  </script>
</body>
```

Komponententechnologien: React (Facebook)

- Verwendet z. B. für Facebook, Instagram
- Webseite ist Komposition unterschiedlicher Komponenten
- Komponenten mit Hilfe von HTML + JavaScript + CSS umgesetzt
- Komponenten innerhalb eines Webauftritts beliebig einsetzbar und einfach zu verteilen
- Im Betrieb erfolgt eine dynamische Anpassung der Inhalte mittels JavaScript



React und MVC

- Konzipiert als JavaScript-Bibliothek für die Entwicklung von Benutzerschnittstellen
  - Das View in Model-View-Controller (MVC)
  - Insbesondere für komplexe dynamische Bedienoberflächen
  - Erlaubt die
    - Kapselung von Inhalten als Komponenten
    - sowie eine effiziente Präsentation dynamischer Daten mit Hilfe von Virtual DOM
  - **Ziel: Bestmögliche Performance**
    - Das Browser-DOM wurde nicht für laufende dynamische Anpassungen konzipiert: Schlechte Performance

Virtual DOM

- Kernkonzept von React
- Erlaubt Verbesserungen der Performance bei Seitenupdates
  - Erforderliche Änderungen an der Schnittstelle werden in einem eigens dafür angelegten **virtuellen DOM** umgesetzt
  - In einem nachgelagerten Schritt werden die Unterschiede zwischen dem virtuellen und dem echten DOM identifiziert
  - Die Abweichungen werden in eine Warteschlange gepuffert und zyklisch gesammelt auf das echte DOM übertragen

Beispiel: Status-Anzeige für Freundeliste (1)

- Anfangszustand des DOM

```
<ul>
  <li class="pc active">
    Albert
  </li>
  <li class="pc active">
    Barbara
  </li>
  <li class="pc active">
    Claudia
  </li>
</ul>
```

- "Schnappschuss"-Zustand

```
<ul>
  <li class="pc active">
    Stefan
  </li>
  <li class="pc active">
    Barbara
  </li>
  <li class="mobile idle">
    Claudia
  </li>
</ul>
```

- Die Liste auf der rechten Seite ändert sich durch Statusänderungen laufend: DOM muss ständig angepasst werden

Beispiel: Status-Anzeige für Freundeliste (2)

- Erforderliche Änderungen:

```
anzeige.childNodes[0].textContent = "Stefan";
anzeige.childNodes[2].className = "mobile idle"
```

- Die Inhalte der Anzeige ändern sich laufend und erfordern entsprechende Anpassungen im DOM
  - Anpassungen bremsen die Bedienschnittstelle aus

Entwickeln mit React

- Entwicklung geschieht auf der Grundlage von JavaScript
  - Zur Vereinfachung der Entwicklung kann der JavaScript-Programmcode teilweise durch deskriptive Code-Anteile ersetzt werden
  - JSX: Mischung aus JavaScript und XML

JSX

- JavaScript + XML
- JSX muss vor der Ausführung in JavaScript übersetzt werden
  - Methode 1: Unter Node.js Quellcodedatei mit jsx-Compiler übersetzen (für produktiven Einsatz)
  - Methode 2: On-the-fly bei der Ausführung der Datei durch den Online-Übersetzer ("Transpiler") Babel (für Entwicklung)
    - Fertige Seiten übersetzen, um Übersetzungsaufwand beim Seitenaufruf zu sparen
    - Offline-Version von Babel verfügbar
- Vermeiden von JSX spart den Übersetzungsvorgang

Beispiel: Klasse für Kundendaten mit JSX

```
<script type="text/babel">
  var Kunde = React.createClass({
    propTypes: {
      vorname: React.PropTypes.string.isRequired,
      nachname: React.PropTypes.string.isRequired,
      alter: React.PropTypes.number.isRequired
    },
    render: function() {
      return <li className="Kunde">
             <p className="Kundenname">
               {this.props.vorname + " " +
                this.props.nachname +
                ", " + this.props.alter}
             </p></li>;
    }});
    :
    :
</script>
```

Allgemein

- Vue und React erlauben die Entwicklung auf niedriger Ebene in reinem JavaScript: Sehr aufwendig
- Auf einer höheren Ebene lässt sich der Anteil von JavaScript-Codeanteilen mit Hilfe deskriptiver Sprachelemente verringern (ähnlich zu Riot)
  - Code-Komplexität nimmt wegen des erforderlichen Zusammenspiels von HTML-Templates und JavaScript zu
  - Zusätzlicher Übersetzungsvorgang auf dem Server
  - Komplexe Entwicklungsumgebung
  - Hoher Installationsaufwand (NodeJS, Package Manager (NPM, Bower), Transpiler, etc.)
  - Regelmäßige Änderungen der zugrundeliegenden Syntax

Datenspeicherung

Motivation

- Die Möglichkeiten zur Datenspeicherung aus dem Browser wurden von Beginn an stark eingeschränkt
- Kein direkter Zugriff aus JavaScript auf die Festplatte des Rechners möglich
  - Schutz der Daten und des Rechners vor Schadsoftware
- In dem Maße, indem sich e-Commerce-Konzepte im Web etablierten, stieg der Bedarf an Speichermöglichkeiten
- Das Speichern von Daten auf dem Server erfordert eine Benutzerverwaltung
- Es ist einfacher, die Daten dort zu speichern, wo sie anfallen und benötigt werden: Auf dem Clientrechner

Beispiele

- Daten mehrseitiger Formulare zwischenspeichern
- Warenkorbinhalte zwischen Seitenaufrufen und Browsersessions erhalten
- Personalisierte Webseiten und Werbung, Zugriffsstatistiken
- Webanwendungen
  - Wie jede andere Anwendung, benötigen diese eine Möglichkeit, anfallende Daten zu speichern
  - Z. B. Zustandsdaten, Anwendungseinstellungen
  - Diese Daten müssen beim nächsten Aufruf wieder eingelesen werden können
- Paywalls
  - Login-Daten zwischenspeichern

Personenbezogene Daten

- Der Einsatz von Speichertechnologien erfordert die vorherige Zustimmung des Benutzers
  - Benutzer müssen darüber informiert werden, dass Daten gespeichert werden sollen
  - Die Gründe hierfür müssen erläutert werden
  - Die Zustimmung muss eingeholt werden und die Möglichkeit gegeben werden, diese jederzeit zu widerrufen
    - Technische Umsetzung der Zustimmung z. B. durch Einsatz einer Check Box
- siehe auch:
  - https://de.wikipedia.org/wiki/Datenschutzrichtlinie_f%C3%BCr_elektronische_Kommunikation
  - http://eur-lex.europa.eu/homepage.html?locale=de (nach Datenschutzgrundverordnung bzw. DSGVO suchen)

Cookies

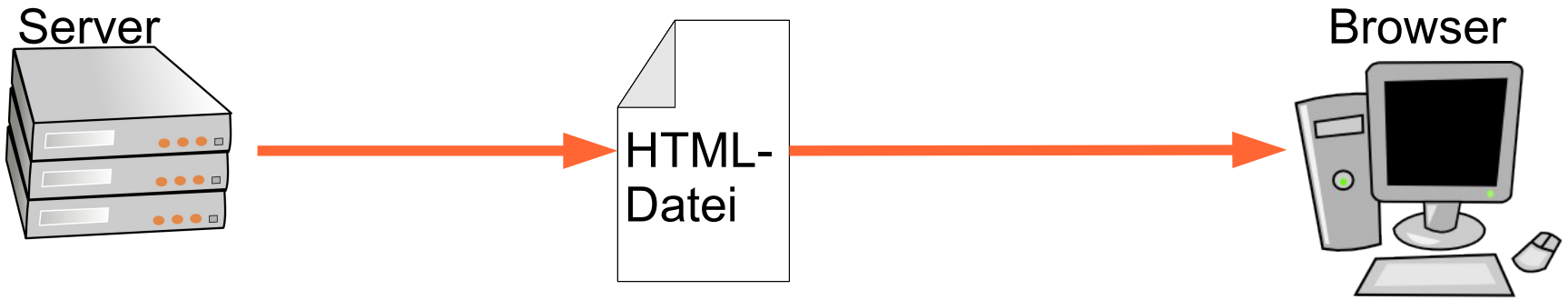
- Netscape (1994)
- Cookies erlauben es, kleine Datenmengen auf dem Clientrechner zu speichern
  - Z. B. eine Benutzerkennung, die dazu dient, einen zum Benutzer gehörenden Datensatz in einer Datenbank zu adressieren
- Dies erfolgt mit Hilfe von JavaScript als Zeichenkette in der `cookie`-Eigenschaft des Dokumentobjekts
- `document.cookie`

Cookies: Speicherung

- Der Cookie wird in einer speziellen Datenbank des Browsers abgelegt
  - Datenbankanwendung oder auch lediglich eine Liste einfacher Textdateien in einem dafür bestimmten Verzeichnis

Cookies: Einlesen

- Beim Aufruf einer Webseite prüft der Browser, ob für diese Seite ein Cookie vorliegt



- Ist dies der Fall, wird dessen Wert in `document.cookie` gespeichert und kann mittels JavaScript verarbeitet werden
  - Der Cookie ist mit der Webseite assoziiert, die den Cookie gesetzt hat
  - Andere Websites können nicht auf die Daten des Cookies zugreifen

Cookie: Genereller Aufbau (1)

- **`key-value; expiration_date; path; domain; secure`**
  - **`key-value`**: Ein Variablenname und ggf. ein oder mehrere Werte. Werte dürfen keine Leerzeichen, Semikolons und Kommas enthalten
    - Cookie-Werte darum ggf. mit der Funktion **`encodeURIComponent()`** encodieren
  - Beispiele: **`kunde=wert`**
    `kunde=wert1#wert2#wert3#...`
  - JavaScript "sieht" beim Auswerten des Cookies nur den **`key-value`**-Teil
  - Der Teil ab "**`; expiration_date...`**" dient zum Speichern von Verwaltungsdaten und für die interne Browserverwaltung

Cookie: Genereller Aufbau (2)

- **`expiration-date`**: Definiert Lebensdauer des Cookies in sec.
  - Beispiel: **`; max-age=60`**
- **`path`**: Gültigkeit mit einem bestimmten Pfad assoziieren
- **`domain`**: Per Voreinstellung gehört ein Cookie zu der Seite, die ihn gesetzt hat. Mit Hilfe von einer Domainangabe kann der Gültigkeitsbereich erweitert werden

Beispiel: Zugriffszähler

```
<html lang="de"><head><!-- … --></head>
<body>
  <output id="output"></output>
  <script>
    if (document.cookie.indexOf("zaehler") > -1) {
      var cparts = document.cookie.split("=");
      var z = cparts[1]; // z: Zählwert
      z = parseInt(z) + 1; // Zeichenkette -> int
      document.getElementById("output").innerHTML =
        "Guten Tag! Dies ist Ihr " + z + ". Besuch!";
      document.cookie = "zaehler=" + z + "; max-age=60";
    }
    else { // Kein Cookie => Erstbegrüßung
      document.getElementById("output").innerHTML =
        "Guten Tag! Wir begrüßen Sie als Neukunde!";
     // Cookie anlegen durch Beschreiben der Variable
     document.cookie = "zaehler=1; max-age=60";
    }
</script></body></html>
```

Cookie-Lebensdauer

- Cookies werden am Ende ihrer Lebenszeit vom Browser gelöscht
- Um die Lebenszeit des Cookies zu verlängern, muss dieser beim erneuten Seitenaufruf wieder angelegt werden

```
document.getElementById("output").innerHTML =
   "Guten Tag! Dies ist Ihr " + z + ". Besuch!";
document.cookie = "zaehler=" + z + "; max-age=60";
```

- Cookie löschen:

```
document.cookie =
  "zaehler=; expires=Thu, 01 Jan 1970 00:00:00 GMT";
```

Nachteile (1)

- Cookie-Spezifikation RFC 2965 (2000):
  - Cookies dürfen 4 KByte groß werden
  - Jede Domain darf bis zu 20 Cookies setzen
  - Browser müssen 300 Cookies Platz bieten
- Cookies also in ihrer Größe und in ihrem Umfang beschränkt
- Alte Cookies können willkürlich gelöscht werden, sobald eine der angegebenen Maximalzahlen überschritten wird
  - Websites mit hohem Cookie-Aufkommen können Cookies anderer Sites aus dem Speicher verdrängen
- Gegenseitige Beeinflussung mehrerer Anwendungsinstanzen in unterschiedlichen Browserfenstern: Alle schreiben in das gleiche Cookie

Nachteile (2)

- Server und Client können auf den Cookie zugreifen
  - Cookies werden bei Seitenaufruf auch an den Server übertragen
  - Bremst Seitenaufruf
- Potentielles Sicherheitsrisiko
  - Daten werden nicht automatisch verschlüsselt gespeichert
  - Cookies können durch Hacking entwendet werden
- Tracking-Cookies:
  - Rechner eines Surfers wird mit einer ID versehen, die auch bei dynamisch zugeteilten IP-Adressen funktioniert
- Benutzer schalten Cookies ab
- **Cookies stellen folglich keine sichere Speichermethode dar**

Web Storage

- Auch: DOM Storage, Supercookies
- Mit HTML5 eingeführt
- Deutlich größere Speicherkapazität: ca. 10MB
- Keine Datenübertragung zum Web Server beim Seitenaufruf
- Kein Ablegen von Daten in den Web Storage durch den Web Server
- Benutzung gegenüber Cookies vereinfacht
  - Daten werden in einem assoziativen Feld als Zeichenketten gespeichert (ggf. Typumwandlung erforderlich)
- Es gibt zwei Varianten:
  - Lokale Speicherung: Local storage
  - Sessionspezifische Speicherung: Session storage

Lokale Speicherung

- Daten werden in Form von Local Shared Objects (LSO) gespeichert
- Diese sind mit einer Domain und dem lokalen Benutzerprofil des Zugriffsrechners verknüpft
- Alle Scripte von Webseiten dieser Domain können in dem betreffenden Benutzerprofil auf die Daten zugreifen
- Im Gegensatz zu Cookies bleiben die Daten ohne weitere Vorkehrung auch nach Beenden einer Browsersitzung bestehen
  - Nicht mehr erforderliche Daten müssen explizit gelöscht werden
- Speicherung durch Schreiben in die Eigenschaft `localStorage` des **`Window`**-Objekts

Session-spezifische Speicherung

- Gespeicherte Daten sind mit dem Browser-Fenster verknüpft
- Der Zugriff ist auf dieses Fenster beschränkt
  - Auf diese Weise können mehrere Instanzen derselben Webanwendung in verschiedenen Fenstern laufen, ohne dass diese gegenseitig ihre Daten überschreiben
  - Erweiterte Funktionalität gegenüber Cookies
- Die Daten werden beim Schließen des Fensters gelöscht
- Speicherung durch Schreiben in die Eigenschaft `sessionStorage` des `Window`-Objekts

Zugriff auf den Web Storage

- Ist für beide Speichervarianten gleich
- Direktes Schreiben in und Lesen aus Variablen (entsprechend für **`sessionStorage`**):
  - ```
    localStorage.name =
    document.getElementById("name").value;
    ```
  - ```
    localStorage["name"] =
    document.getElementById("name").value;
    ```
  - ```
    document.getElementById("output").innerHTML =
    "Guten Tag " + localStorage.name;
    ```
    ```

Zugriffsmethoden

- **`setItem('key', 'value')`, `getItem('key')`, `removeItem('key')`**
  - **`key`**: Name der Eigenschaft des Objekts bzw. des Schlüssels des assoziativen Felds
  - **`value`**: Wert der Eigenschaft bzw. des Feldelements
- Beispiele:
  - `localStorage.setItem("name", document.getElementById("name").value);`
  - `document.getElementById("output").innerHTML = "Guten Tag " + localStorage.getItem("name");`
  - `localStorage.removeItem("name");`

Datentypen

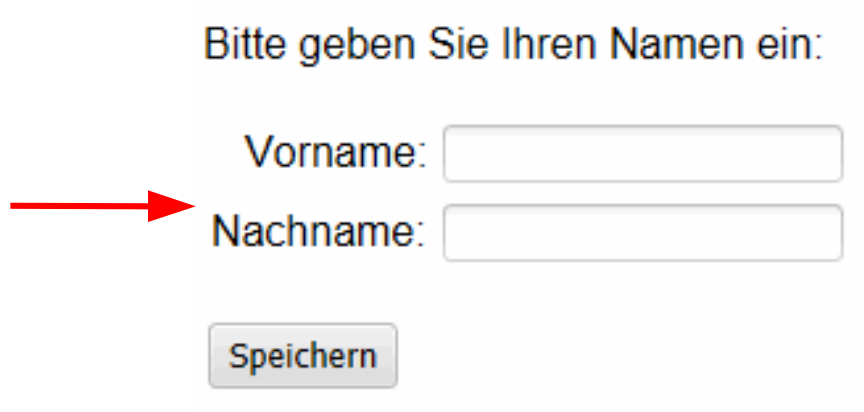
- Die Speicherung der Daten erfolgt als Strings
- Bei anderen Typen ist eine Typumwandlung erforderlich
- Beispiel: Speicherung eines Zahlenwerts

```
localStorage.val = 3.57; // Umwandlung in String
var val = 10 + parseFloat(localStorage.val);
```

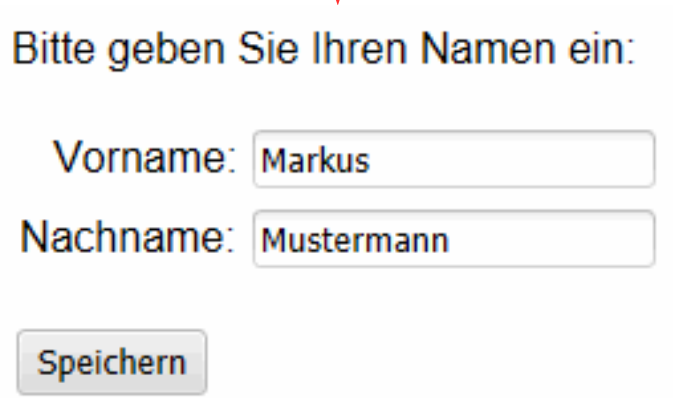
  - Die Zahl wird bei der Speicherung implizit in eine Zeichenkette umgewandelt
  - Die eingelesene Zeichenkette muss vor der Rechnung wieder in einen Zahlenwert umgewandelt werden
  - Sonst würden Strings konkateniert
    - Das Ergebnis wäre dann `103.57` und nicht `13.57`

Beispiel: Besuchernamen erfassen

- Neukunden wird eine Eingabemaske angezeigt, in der sie ihren Namen eingeben können



- Nach dem Speichern der Daten oder bei erneutem Aufruf der Seite wird die Eingabemaske nicht mehr angezeigt



- Anstatt dessen wird der/die Kund/in mit Namen angesprochen

→ Guten Tag Markus Mustermann

HTML-Kopf

- Für die dynamische Anzeige der Eingabemaske werden CSS-Klassen für das Ein- und Ausblenden definiert
-
```
<style>
  .default {
    display: none;
  }
  .show {
    display: block;
  }
</style>
```
- JavaScript einbinden:

```
<script src="MyLocalStore.js" defer></script>
```

HTML-Körper

```
<body>
  <div id="output"></div>
  <div id="input" class="default">
    <p>Bitte geben Sie Ihren Namen ein:</p>
    <table>
      <tr>
        <td align="right">Vorname:</td>
        <td><input type="text" id="vorname"></td>
      </tr>
      <tr>
        <td align="right">Nachname:</td>
        <td><input type="text" id="nachname"></td>
      </tr>
    </table>
    <p><button id="saveBtn">Speichern</button></p>
  </div>
</body>
```

Objekte speichern

- Mittels Web Storage können auch Objekte gespeichert werden
- Diese müssen vor der Speicherung mit der Funktion `JSON.stringify()` in einen JSON-String umgewandelt werden
- Dieser Vorgang wird als **Serialisierung** bezeichnet
- Wir nutzen dies, um die Formulardaten als Objekt zu speichern:

```
document.getElementById("saveBtn").onclick=function(){
  var person = {
    vorname: document.getElementById("vorname").value,
    nachname: document.getElementById("nachname").value
  }
  localStorage.person = JSON.stringify(person);
  doLocalStorage(); // Gespeicherte Daten anzeigen
};
```

Objekte einlesen

- Als String gespeicherte Daten werden mit der Funktion `JSON.parse()` aus dem String ausgelesen
- Beispiel:

```
if(localStorage.person) {
  var person = JSON.parse(localStorage.person);
  // ...
}
else {
  /* Kein Supercookie vorhanden
     => Formular anzeigen */
}
```

Daten einlesen: Vollständiger Code

```
function doLocalStorage () {
  if (localStorage.person) {
    document.getElementById("input").classList.
      remove("show");
    var person = JSON.parse(localStorage.person);
    document.getElementById("output").innerHTML =
      "Guten Tag " +
       person.vorname  + " " + person.nachname +"<br>";
  }
  else { // Kein Supercookie => Formular anzeigen
   document.getElementById("input").classList.
     add("show");
  }
}
// Zu Beginn vorhandene Daten anzeigen, sonst Maske
doLocalStorage();
```

Objekte löschen

- Erfolgt wie bei einfachen Variablen durch **`removeItem(object)`**
- Beispiel:

```
localStorage.removeItem("person");
```

- Alle Einträge (Objekte, Variablen) löschen:

```
localStorage.clear();
```

Feld speichern

- Felder lassen sich ebenfalls serialisieren
- Beispiel:

```
var person = {
  vorname: document.getElementById("vorname").value,
  nachname: document.getElementById("nachname").value
}
person.val = []; // Im Objekt ein Feld anlegen
for (var i=0; i<10; i++) {
  person.val.push(i); // Feldelemente schreiben
}
localStorage.person = JSON.stringify(person);
```

Feld einlesen

```
var person = JSON.parse(localStorage.person);

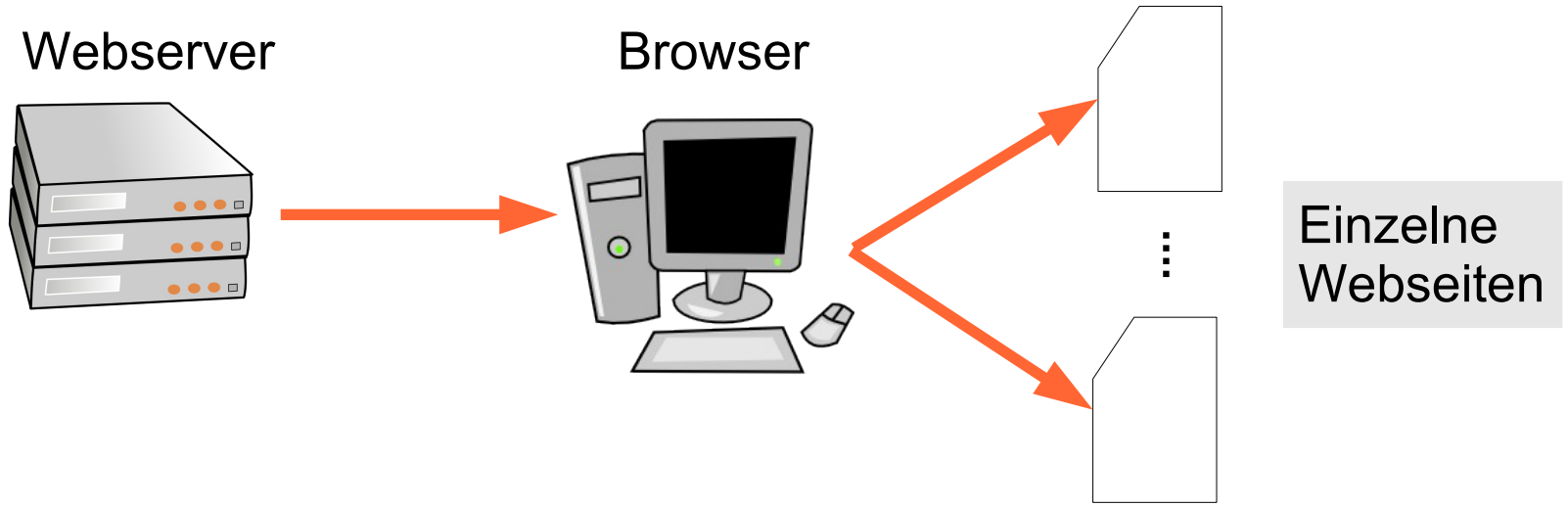
var str = "Guten Tag " +
  person.vorname + " " + person.nachname + "<br>";

  for(var i=0; i < person.val.length; i++)
    str += person.val[i] + ", ";

  document.getElementById("output").innerHTML = str;
```

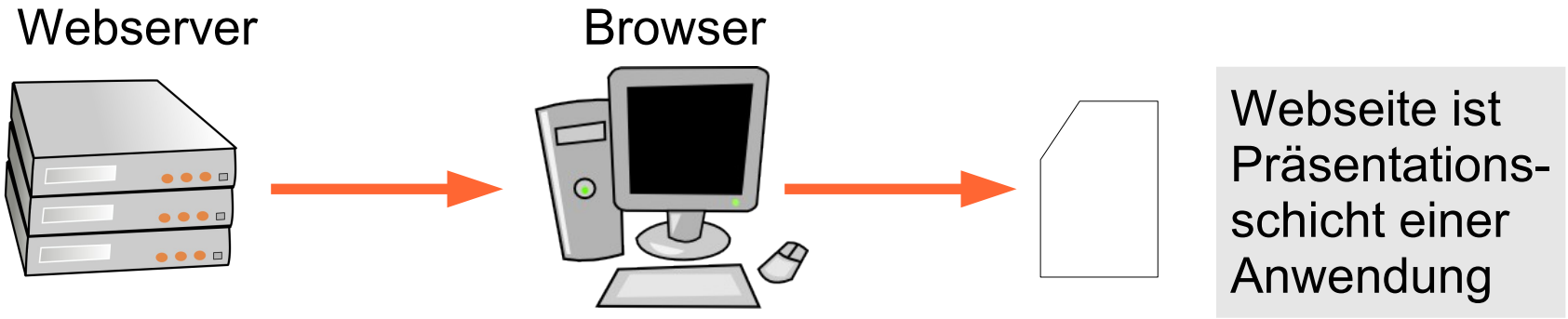
Verlaufsverwaltung

Traditioneller Webauftritt



- Webserver liefert einzelne Webseiten aus
- Webseiten sind über URLs adressierbar
- URLs bieten einen beliebigen Einsprungpunkt in den Webauftritt
- Einsprungpunkte lassen sich über Bookmarks sichern und durch Weitergeben der URL an Dritte weiterreichen
- Im Client kann durch Nutzen des Vor- und Zurück-Buttons zwischen den Seiten gewechselt werden

Webanwendung



- Webserver liefert Webanwendung aus
- Anwendung lädt dynamisch Daten nach, um Seiteninhalte aufzufrischen: Es erfolgt kein Seitenwechsel ("data on the wire")
- An die Stelle wechselnder Seiten treten **Anwendungszustände**
- Anwendungszustände sind nicht über URLs adressierbar
- Keine Bookmarks, kein Weiterreichen
- Im Client kann nicht mit Hilfe der Vor- und Zurück-Buttons zwischen den Zuständen gewechselt werden

Client-Side Routing

- Erlaubt es, die Benutzeroberfläche einer Webanwendung mit Hilfe künstlicher URLs zu steuern
- Die URL-basierte Steuerung traditioneller Webauftritte wird mit Hilfe von JavaScript nachgerüstet
- Zu diesem Zweck werden die relevanten Parameter eines Anwendungszustands in einem Objekt gekapselt und einer künstlichen URL zugeordnet
- Anwendungszustände werden auf einem Stapel gespeichert
- Der Aufruf der URL eines Zustands führt zur Restauration des Anwendungszustands
- Für das Routing auf der Seite des Clients existieren eine Vielzahl von Lösungen

HTML5-History-API

- Anwendungszustände werden mit Hilfe der Methode `pushState()` im Browserverlauf gespeichert
  - Hierfür werden alle relevanten Zustandsparameter in einem Zustandsobjekt gespeichert und an die Methode übergeben
- Benutzung der Vor- und Zurück-Buttons im Browser löst einen `popstate`-Event aus
- Für diesen Event muss ein Ereignisbehandler registriert werden
  - Die Aufgabe des Ereignisbehandlers besteht darin, an Hand der Daten im Zustandsobjekt den Anwendungszustand wiederherzustellen

Zustand speichern (1)

- `history.pushState(stateObject, title, URL);`
  - `stateObject`: Enthält die Daten der Zustandsbeschreibung. Obergrenze 640kB. Größere Zustandsobjekte müssen im Web Storage gespeichert werden. Zustandsobjekt dann lediglich Verweis auf das Objekt im Web Storage
  - `title`: Titel werden bislang ignoriert. Für Aufwärtskompatibilität trotzdem besser angeben oder eine leere Zeichenkette
    - Titel kann bei Bedarf mit Daten aus dem Zustandsobjekt durch Schreiben in `document.title` gesetzt werden

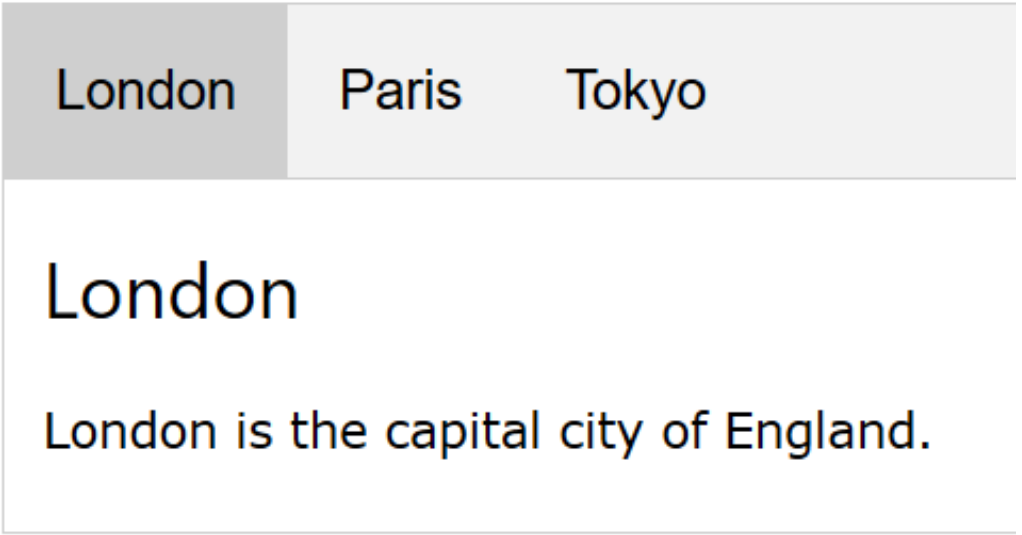
Zustand speichern (2)

- `history.pushState(stateObject, title, URL);`
  - `URL`: Der in der Adresszeile anzuzeigende Eintrag
    - Ermöglicht Bookmarking und Browserneustart für einen bestimmten Anwendungszustand
    - URL so konstruieren, dass auf ihrer Grundlage ein bestimmter Anwendungszustand restauriert werden kann
    - URL kann absolut oder relativ sein. Wenn relativ, wird URL der aktuellen Seite ergänzt
    - Eintrag optional: Fehlt dieser, wird die originale URL angezeigt.
    - Browser versucht in jedem Fall nicht, diese Seite zu laden
- siehe auch:

  https://developer.mozilla.org/en-US/docs/Web/API/History_API

Beispiel: Single Page App

- Abgewandeltes Reiter-Beispiel von W3Schools
- Informationen zu unterschiedlichen Städten mit Reitern auswählen
- Alle Informationen werden unter der gleichen Webseite angezeigt
  - Keine Verzweigung über URL möglich
  - Vor- und Zurück-Buttons des Browsers funktionieren nicht



HTML (1)

```
<html>
  <head>
    <link rel="stylesheet" href="SinglePageApp.css">
    <script src="SinglePageApp.js" defer></script>
  </head>
  <body>
    <ul class="tab">
      <!-- <span> an Stelle von Verweis <a>, da Verweis
           Browser-History beeinflusst -->
      <li><span id="sLondon" class="tablinks"
        onclick="openCity('London')">London</span></li>
      <li><span id="sParis" class="tablinks"
        onclick="openCity('Paris')">Paris</span></li>
      <li><span id="sTokyo" class="tablinks"
        onclick="openCity('Tokyo')">Tokyo</span></li>
    </ul>
    <!-- Fortsetzung nächste Folie -->
```

HTML (2)

```
    <div id="London" class="tabcontent">
      <h3>London</h3>
      <p>London is the capital city of England.</p>
    </div>

     <div id="Paris" class="tabcontent">
       <h3>Paris</h3>
       <p>Paris is the capital of France.</p>
     </div>

     <div id="Tokyo" class="tabcontent">
       <h3>Tokyo</h3>
       <p>Tokyo is the capital of Japan.</p>
     </div>
   </body>
</html>
```

- CSS: Analog zum Tabs-Beispiel mit **span** an Stelle von **a**-Element, siehe http://www.w3schools.com/howto/howto_js_tabs.asp

JavaScript: Zustand speichern

```
function openCity(cityName) {
    hideCities();
    showCurrentCity(cityName);

    // Save state in state object
    var stateObj = {city: cityName};
    history.pushState(stateObj, cityName,
      "SinglePageApp.html?" + cityName);
}
```

- Aus dem Namen der Stadt lassen sich die relevanten IDs ableiten
- Name der Stadt wird ebenfalls URL-encodiert für das Bookmarking

JavaScript: Zustand wiederherstellen

```
window.onpopstate = function (event) {
  if (event.state) {
    // Wenn das Ereignis ein state-Objekt hat
    // stelle den Zustand wieder her
    var state = event.state;
    hideCities();
    showCurrentCity(state.city);
  }
}
```

- Benutzung des Vor- und Zurück-Buttons löst auf dem Window-Objekt einen popstate-Event aus
- Das damit verknüpfte Ereignisobjekt besitzt in der State-Eigenschaft eine Kopie des Zustandsobjekts

JavaScript: Bookmarks ermöglichen

- Beim Laden einer Webseite wird ein **onload**-Event ausgelöst
- Hierfür Ereignisbehandler registrieren und in diesem die URL der aufgerufenen Seite decodieren, um die gewünschte Stadt auszulesen. Gewünschte Stadt schließlich anzeigen
-
```
window.onload = function (event) {
  if (window.location.href.indexOf("?") > -1) {
    var cityName = window.location.href.substr(
      window.location.href.indexOf("?")+1);
    showCurrentCity(cityName);
  }
}
```

JavaScript: Angezeigte Seite ausblenden

```
function hideCities() {
  // Hide all elements with class="tabcontent"
  var tabcontent =
    document.getElementsByClassName("tabcontent");
  for(var i=0; i<tabcontent.length; i++)
    tabcontent[i].style.display = "none";
  // Deactivate all elements with class="tablinks"
  var tablinks =
    document.getElementsByClassName("tablinks");
  for(var i=0; i<tablinks.length; i++)
    tablinks[i].classList.remove("active");
}
```

- Die Klasse **`active`** liefert den grauen Hintergrund für den aktiven Reiter-Zustand

JavaScript: Gewünschte Seite anzeigen

```
function showCurrentCity(cityName) {
  // Show current tab, and add "active" class to the
  // link or button, respectively, that opened the tab
  document.getElementById(cityName).style.display =
    "block";
  document.getElementById("s"+cityName).
    classList.add("active");
}
```

- ID des Reiters wird durch Konkatenation von `"s"` mit `cityName` erzeugt

Audio

HTML5-Audio

- Realisiert mit Hilfe des **<audio>**-Elements
  - Je nach installierter Codebasis werden in **<source>**-Elementen unterschiedliche Dateiformate angeboten
- Beispiel:

```
<audio controls>
  <source src="http://adress1/title1.mp3"
    type='audio/mpeg' />
  <source src="http://adress1/title1.oga"
    type='audio/ogg; codecs=vorbis' />
  <p>Audio element nicht unterstützt.</p>
</audio>
```

- **controls**: Audioplayer mit GUI. Z. B. Play-Button
- Die erste unterstützte Quelle wird abgespielt

In JavaScript

- Audio-Element erzeugen: `var a = new Audio();`
- Audio laden:

```
if (a.canPlayType("audio/mp3"))
    a.src = "audio/myaudiofile.mp3";
else if (…)
  // Ggf. andere Dateiformate prüfen
```

- Abspielen: `a.play();`
- Pausieren: `a.pause();`

Literatur

[1] D. Flanagan: JavaScript – Das umfassende Referenzwerk, O'Reilly, 2012

[2] Gallardo, R., et al.: The Java Tutorial: A Short Course on the Basics, https://docs.oracle.com/javase/tutorial/, 21.6.2018

[3] Polyfills: http://webcomponents.org/polyfills/

[4] https://www.w3schools.com/