



FH Salzburg

VO Web-Technologien

Einheit 5, Oliver Jung

Technik
Gesundheit
Medien

Clientseitige Web-Programmierung

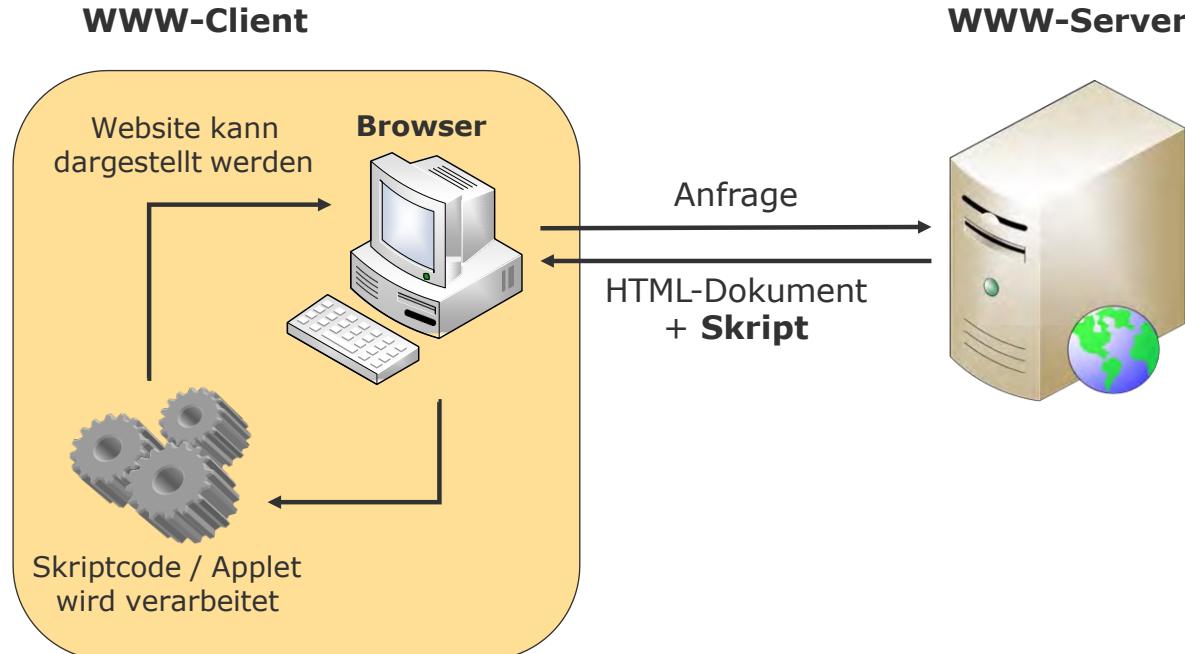


- Moderne Webseiten sind sehr lebendig, sie
 - **gehen interaktiv auf Nutzerwünsche ein,**
 - **verknüpfen** inhaltlich passend **unterschiedliche Webseiten** (Mashups / Widgets) oder
 - werden erst **auf Nutzerwunsch generiert**
- **Web-Programmierung** umfasst die Techniken zur Entwicklung solcher **dynamischen Webseiten und Webanwendungen** bereit
- Bei der **clientseitigen Webprogrammierung** geht es vornehmlich um die mit **JavaScript** realisierte Web-Programmierung, die **im Browser** (Web-Client) **zur Ausführung kommt**
- Unter **serverseitiger Webprogrammierung** werden alle Techniken zusammengefasst, die auf dem Server ausgeführt werden

Clientseitige Web-Programmierung



Ausführung im Browser :



Clientseitige Web-Programmierung



Laden von Skripten

- Skript-Dateien werden im HTML-Dokument referenziert
- Lösen zusätzlichen HTTP-Request zum Server aus, um das Skript zu laden
- Verarbeitung der Skripte:
erst parsen, dann ausführen

Achtung

- Laden und Ausführen von Skripten blockiert Anzeige von restlichem HTML Dokument
 - ➔ Skripte am Ende einbinden
- Anzahl und Größe der Skripte haben Auswirkung auf Geschwindigkeit der Webseite

```
<html>
...
<body>
...
<script
  type="text/javascript"
  src="script.js"
>
</script>
</body>
</html>
```

JavaScript



- **JavaScript:** Einzige weit verbreitete und **standardisierte** Technologie für clientseitige Skripte in Webseiten
- Features:
 - Zugriff & Manipulation der HTML-Struktur:
Document Object Model (DOM)
 - Reaktion auf vom Nutzer ausgelöste **Events**
 - Asynchrone Server-Kommunikation im Hintergrund: **AJAX, Websockets**
 - Animationen, 3D-Grafiken, Musik
 - Zugriff auf Standort, Batteriestatus, Webcam, Mikrofon, Bewegungssensoren des Clients...
 - inspiriert vom Funktionsumfang mobiler Apps
 - **Service Worker:** ermöglicht Offline-Zugriff auf Webseiten
 - ...

Anwendungsbeispiele



Online-Karten

The screenshot shows a map interface with a search bar at the top left containing "FH Salzburg". Below the search bar is a large photograph of the university's modern glass building. The map itself displays a detailed view of the Salzburg region, specifically the area around Urstein. Key locations marked include "FH Salzburg" (red dot), "Schloss Urstein" (pink dot), "Pension Dorrerbauer" (pink dot), "Hotel & Gasthof zum Kirchenwirt" (orange dot), "Arnika Apotheke Kanitz" (red dot), "FC Puch" (blue dot), "Bergspezial Puch" (blue dot), and "Gasthaus Bischof" (pink dot). The map also features several rivers, including the Salzach and Aubach, and various roads like the A10 and E45. On the left side of the map, there is a sidebar for "FH Salzburg" with the following information:

- FH Salzburg**
- 4,3 ★★★★★ (100) ⓘ
- Universität
- Übersicht** (selected)
- Rezensionen
- Info
- Routenplaner
- Speichern
- In der Nähe
- An Smartphone senden
- Teilen
- Urstein S 1, 5412 Salzburg, Österreich
- Geschlossen · Öffnet Mi um 08:00
- fh-salzburg.ac.at
- fh-salzburg.ac.at
- +43 50 22110
- P3FP+6V Salzburg, Österreich
- An mein Smartphone senden
- Änderung vorschlagen

At the bottom left of the sidebar, there is a "Fotos" section with a small thumbnail of a photo labeled "Ebenen".

Anwendungsbeispiele



Kollaborations-Tools

Screenshot of a code editor interface showing two versions of a factorial function. The sidebar on the left includes icons for file operations (New, Open, Save, Undo, Redo) and communication (Comment, Chat).

| Title | Language | Viewing |
|-----------------------------|----------|----------|
| Evolution of a Python pr... | Python | 2 others |

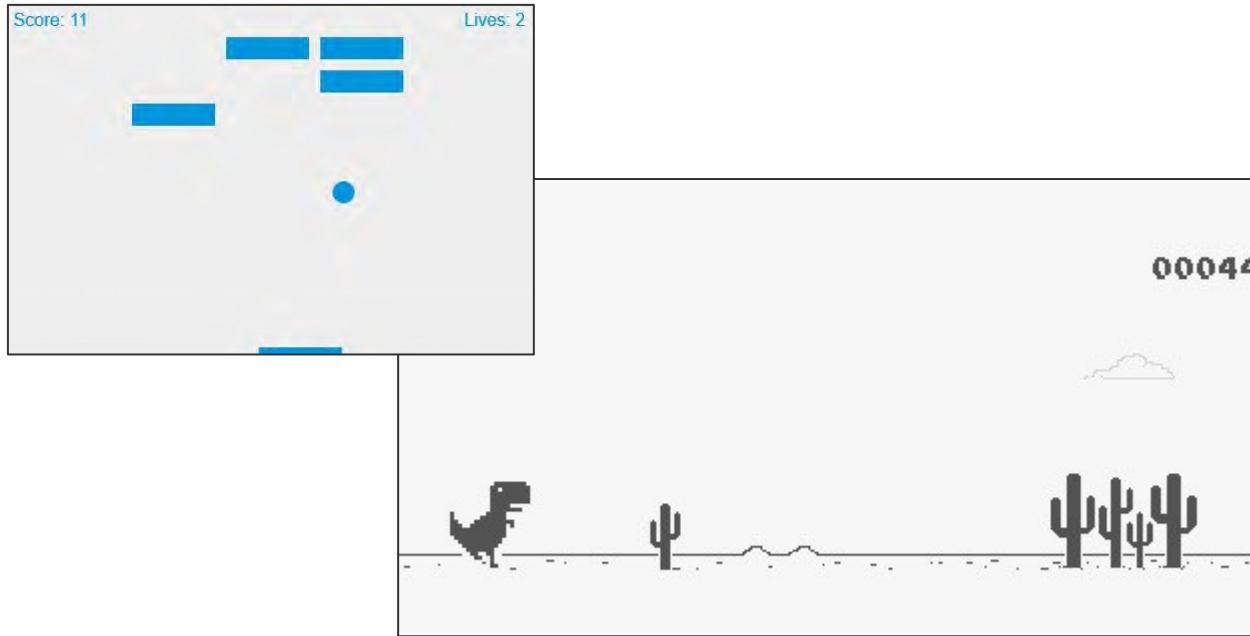
```
9 #First year programmer, studied Pascal
10 def factorial(x):
11     result = 1
12     i = 2
13     while i <= x:
14         result = result * i
15         i = i + 1
16     return result
17
18 print(factorial(6))
19
20
21 #First year programmer, studied C
22 def fact(x): #{
23     result = i = 1;
24     while (i <= x): #{
25         result *= i;
26         i += 1;
27     //}
28     return result;
29 //}
30 print(fact(6))
31
32
33 #First year programmer, SICP
34 @tailcall
35 def fact(x, acc=1):
36     if (x > 1): return (fact((x - 1), (acc * x)))
37     <
```

Indent Mode: Soft Tab Size: 4 Show Invisibles: No Word Wrap: Off Ln: 33 Col: 1

Anwendungsbeispiele



Browser-Spiele



Früher: Flash, Silverlight, Java-Applets



- Haupttreiber für die **ursprüngliche** Verbreitung dieser Technologien:
 - fehlende Unterstützung von Audio und Video im Browser
 - schlechte Leistung und Geschwindigkeit von JavaScript
 - fehlende Funktionen in den Standard-Technologien des Webs
- Heutzutage nicht mehr unterstützt
 - Java-Applets und Flash in Vergangenheit durch Sicherheitsprobleme aufgefallen
 - Adobe Flash seit dem 12.01.2021 nirgendwo mehr unterstützt
- Und nun?
 - Audio und Video seit HTML5 nativ unterstützt
 - verbesserte Geschwindigkeit und Funktionsumfang von JavaScript
 - Animationen, interaktive Inhalte: mit CSS und JavaScript umsetzbar
 - alle Funktionen direkt in JavaScript und CSS umsetzbar

DOM – Document Object Model



Wie kann mit einer Programmiersprache auf HTML-Dokumente und deren Inhalt zugegriffen werden?

Document Object Model – DOM

- Grundlage für das Auslesen aus und Manipulieren von HTML-Dokumenten auf Client-Seite
- Ursprünglich plattform- & sprachunabhängige Schnittstelle zum Zugriff auf Dokumente(teile)
- Vom W3C standardisiert
 - bildet HTML-Baumstruktur in Klassen und Objekte ab
 - Manipulation der Bäume und Navigation in Bäumen
 - Ereignismodell
 - Unterstützung von XML-Namespaces

DOM: Objekt-Modell



Aufbau

- DOM modelliert Dokument entsprechend seiner Struktur als Strukturbbaum
- DOM definiert Objekte mit zugehörigen Attributen und Methoden zur Manipulation der Objekte
- DOM legt fest:
 - Schnittstellen und Objekte zur Darstellung/Manipulation von Dokumenten
 - Semantik der definierten Objekte/Schnittstellen zur Beschreibung von Attributen und Verhalten
 - Beziehungen und Kompatibilität der einzelnen Objekte/Schnittstellen

DOM: Objekt-Modell



HTML-Code und seine Darstellung

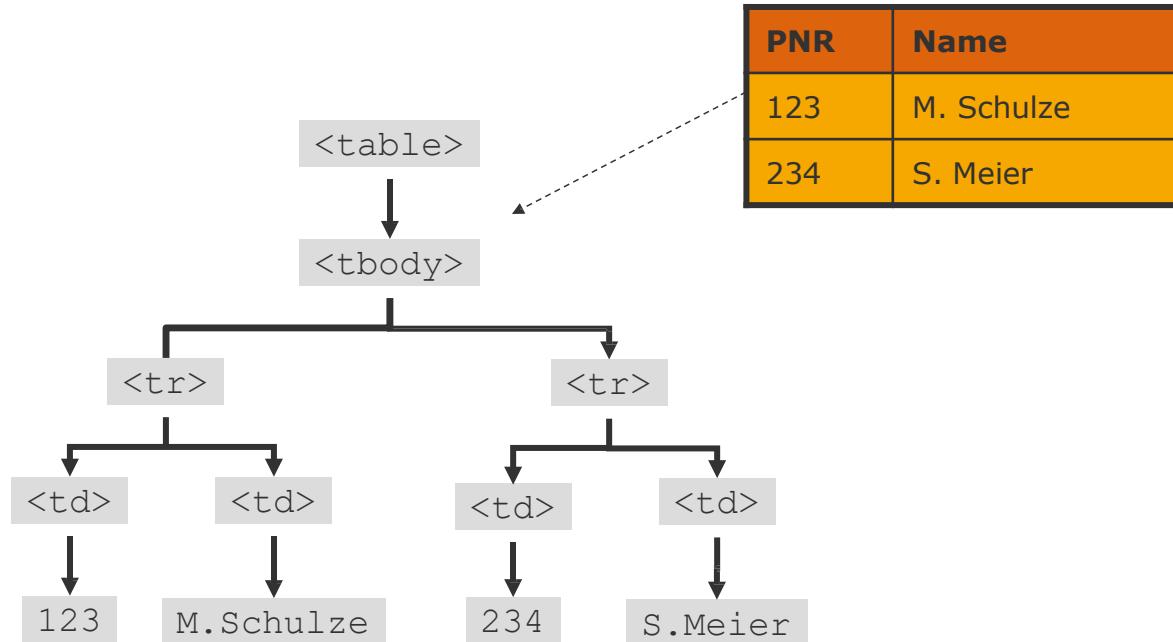
```
<table>
...
<tbody>
  <tr>
    <td>123</td>
    <td>M. Schulze</td>
  </tr>
  <tr>
    <td>234</td>
    <td>S. Meier</td>
  </tr>
</tbody>
</table>
```

| PNR | Name |
|-----|------------|
| 123 | M. Schulze |
| 234 | S. Meier |

DOM: Objekt-Modell



Darstellung als Strukturbau



DOM: Objekt-Modell



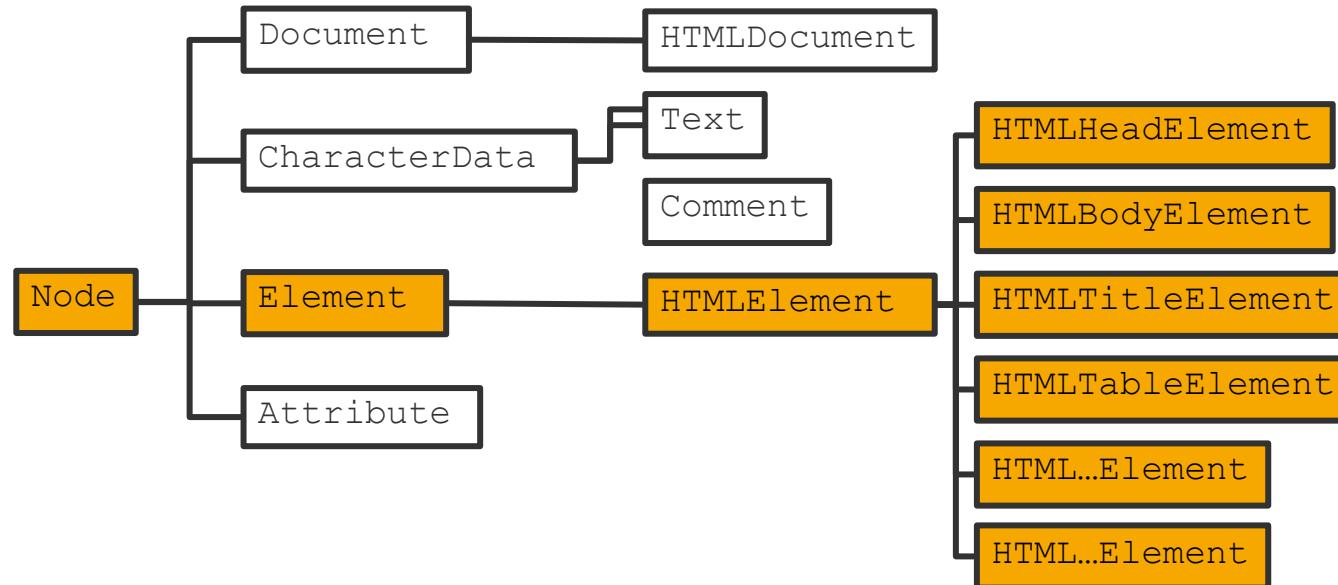
Klassenhierarchie

- Zu jedem Dokument gehören:
 - Wurzelknoten
 - Elementknoten
 - Attributknoten
 - Textknoten
 - (Kommentarknoten)
- DOM ist eine Spezifikation, definiert Schnittstellen
- Programmiersprachen (v.a. JavaScript) implementieren diese
- Implementierungen in anderen Sprachen möglich, z.B.
 - für Java: Dom4J, JDOM, ...

DOM: Objekt-Modell



Klassenhierarchie



DOM: Objekt-Modell



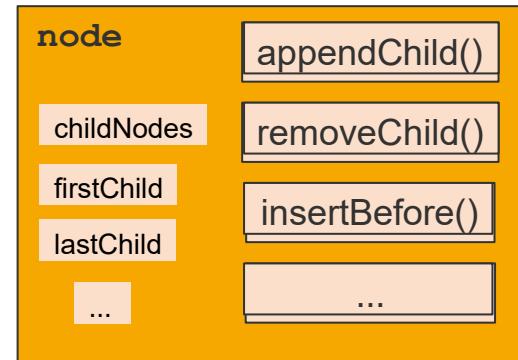
Zentrale Klasse: Node

■ Attribute:

- node.childNodes
- node.firstChild
- node.lastChild
- node.parentNode
- ...

■ Methoden:

- node.appendChild()
- node.removeChild()
- node.insertBefore()
- node.cloneNode()
- ...



JavaScript – Begriffsklärung



- **Java:** Plattformunabhängige Programmiersprache
- **JavaScript:** Script-Sprache zum manipulieren von HTML-Elementen
 - erstmals in Netscape Navigator verwendet
 - Name aus Marketinggründen gewählt, keine Gemeinsamkeiten mit Java
- **ECMAScript:** Spezifizierung für Script-Sprachen – Standard: ECMA-262
 - basiert ursprünglich auf JavaScript
 - heute implementiert JavaScript ECMAScript
- **JScript:** Microsofts JavaScript-Dialekt
 - nahezu identisch mit JavaScript, anderer Name aufgrund von Lizenzrechten
 - implementiert ECMAScript

Wer ist ECMA?

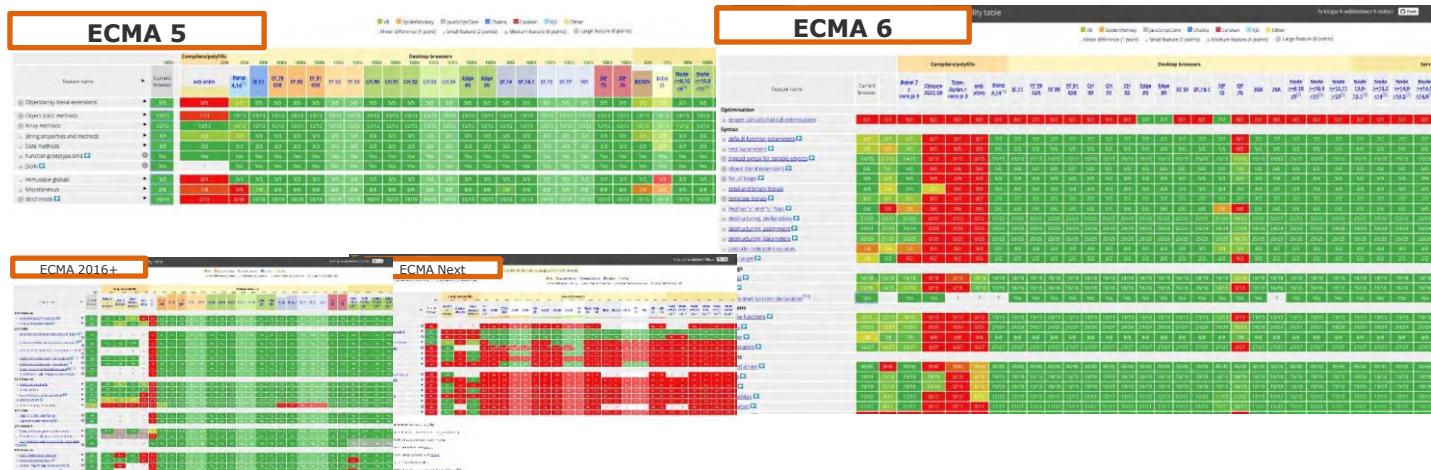


- **Ehemals:** European Computer Manufacturing Association
 - Sitz in Genf
- Mitgliedschaftsbasierte Standardisierungsorganisation
 - Adobe, Google, IBM, HP, Minolta, Intel, Hitachi, PayPal, Yahoo, Microsoft, AirBNB, AMD, Apple, Canon, Facebook, Netflix, Toshiba, Twitter, ...
 - ETH Zürich, EPFL Lausanne, Bundesanstalt für Materialforschung (BAM), Mozilla Foundation, ...
- Mehrere Standards
 - ECMA-262 → ECMAScript
 - ECMA-107 → FAT12/FAT16 Dateisystem
 - ECMA-404 → JSON
 - ...
 - <http://www.ecma-international.org/publications/standards/Standard.htm>

Browserkompatibilität



- Nicht alle Browser unterstützen immer alle Features
- Kompatibilität hat sich deutlich verbessert
- Übersicht, welcher Browser was unterstützt
 - <http://kangax.github.io/compat-table/>



JavaScript - Sicherheit



Früher: Schlechter Ruf aufgrund verschiedener Sicherheitsprobleme

Lösungsansätze:

- JavaScript-Interpreter in einer **Sandbox**
 - ➔ JavaScript hat nur Zugriff auf Objekte des Browsers, nicht auf lokale Hardware oder das Filesystem
- Zugriff nur auf Ressourcen der selben Domäne
 - **Same Origin Policy (SOP)**
 - ➔ Erschwert **Cross-Site-Scripting (XSS)**

Heute: Weit verbreitet und in nahezu jeder Webanwendung anzutreffen

- Sicherheit verbessert, Angriffsmöglichkeiten können allerdings nicht grundsätzlich ausgeschlossen werden
- derzeit keine bessere Alternative für dynamische Webseiten
- Frameworks und Bibliotheken erleichtern Einsatz von JavaScript

Exkurs: Same Origin Policy – Praxisbeispiel



Von FH Salzburg auf google.de: Ausführen eines GET-Requests

→ Nicht möglich, weil Namensraum (Domain) nicht
vom gleichen Ursprung (=origin) stammt

1. HTTP-Requests auf **google.de** in der Entwicklerkonsole vorbereiten:

```
» x = new XMLHttpRequest()
← ▶ XMLHttpRequest { readyState: 0, timeout: 0, withCredentials: false, upload: XMLHttpRequestUpload,
» x.open("GET", "https://google.de")
...
```

2. Abschicken

```
» x.send()
```

3. Fehlermeldung browserseitig:

! Quellübergreifende (Cross-Origin) Anfrage blockiert: Die Gleiche-Quelle-Regel verbietet das Lesen der externen Ressource auf <https://google.de/>.

ECMAScript Versionen



- 1995: JavaScript wird erstmals angekündigt
- 1996: Netscape Navigator 2 mit JavaScript
- 1996: Internet Explorer 3 mit JScript
- 1996: Netscape übergibt JavaScript an ECMA zur Standardisierung
- 1997: ECMAScript 1
- 2007/8: ECMAScript 4 scheitert an Unstimmigkeiten der beteiligten Parteien
- 2009 – ECMAScript 5
- 2015 – 2024: ECMAScript 2015 - 2024
- 2025 – ECMAScript 2025
 - wurde im Juni 2025 fertiggestellt
- ES.next
 - Platzhalter, bezieht sich auf die nächste Version zum Zeitpunkt des Schreibens

JavaScript – Variablen und Funktionen



JavaScript – Moderne dynamische Sprache → keine feste Typisierung

- Ein **Wert** in JavaScript ist immer von einem speziellen **Typ**
 - **Primitive Datentypen:**
String, Number, BigInt, Boolean, Null, Undefined,
 - **Objects** (für komplexe Datentypen)
 - **Symbols** (kreieren Identifier für Objects)
- Die wichtigsten Bestandteile von JavaScript-Programmen (Skripten) sind **Funktionen** → aufrufbarer Code
 - JavaScript enthält Funktionen, können aber auch selbst kreiert werden
- JavaScript **Referenzen im Netz:**
 - <http://www.w3schools.com/jsref/default.asp> (Englisch)
 - <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (Englisch)
 - <https://wiki.selfhtml.org/wiki/JavaScript> (Deutsch)

JavaScript – Variablen und Typen



- **Variablen** ermöglichen es, zur späteren Verwendung Werte zu speichern

```
        Schlüsselwort          Bezeichner          Wert  
        ↓  
let hello = "Hallo Welt";
```

- **Dynamische Sprachen** (z.B. JavaScript) erlauben es Variablen, Werte unterschiedlicher Typen zuzuweisen

```
let hello = "Hallo Welt";  
hello = 39;  
hello = true;
```



- **Statische Sprachen** (z.B. Java) erlauben dies nicht

```
String hello = "Hallo Welt";  
hello = 39;  
hello = true;
```

JavaScript – Variablen und Typen



- Modernes JavaScript besitzt verschiedene Schlüsselwörter zur **Definition von Variablen**:

1. **const** – beinhaltet konstante Werte
2. **let** – beinhaltet Werte die sich ändern dürfen
 - Beispiele:

```
let hello = "Hallo Welt";  
hello = 39;  
hello = true;
```



```
const hello = "Hallo Welt";  
hello = 39;  
hello = true;
```

- Mit **let**, oder **const** deklarierte Variable dürfen nicht im selben Block redeklariert werden:

```
let hello = "Hallo Welt";  
let hello = 39;  
let hello = true;
```

```
const hello = "Hallo Welt";  
const hello = 39;  
const hello = true;
```

JavaScript – null und undefined



null ist ein eindeutiger Wert und steht für ein leeres Objekt

```
typeof null === "object" // ergibt true
```

undefined ist ein Typ und undefinierte¹ Variablen sind vom Typ „undefined“.

```
typeof undefined === "undefined" // ergibt true
```

¹ Undefinierte Variablen, sind Variablen die zwar deklariert, aber nirgends definiert worden sind

JavaScript – Funktionen



- **Funktionen:** Wiederverwertbare Code-Bausteine zum Speichern von Verhalten
 - ... können parametrisiert werden
 - ... können einen Wert zurückgeben
 - ... sollen genau eine Aufgabe übernehmen
 - ... sollen einen sprechenden Bezeichner haben
 - ... sollten möglichst keine Seiteneffekte haben

```
1  function add(a, b) {  
2      return a + b;  
3  }
```

add(3, 4)

Definition

Aufruf

JavaScript – Funktionen



- **Funktionen:** Wiederverwertbare Code-Bausteine zum Speichern von Verhalten
 - ... können in Variablen gespeichert werden

```
1  function add(a, b) {  
2      return a + b;  
3 }
```

```
add(3, 4)
```

vs

```
1  const add = function(a, b) {  
2      return a + b;  
3 }
```

```
add(3, 4)
```

JavaScript – setInterval und setTimeout



- **setTimeout** ermöglicht die einmalige Ausführung einer Funktion **nach einem bestimmten Zeitintervall**.
- **setInterval** ermöglicht die wiederholte Ausführung einer Funktion, die nach dem **Zeitintervall** beginnt und dann in diesem Intervall **kontinuierlich wiederholt** wird.

```
let timerId = setTimeout (func|code, [delay], [arg1], [arg2], ...)  
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

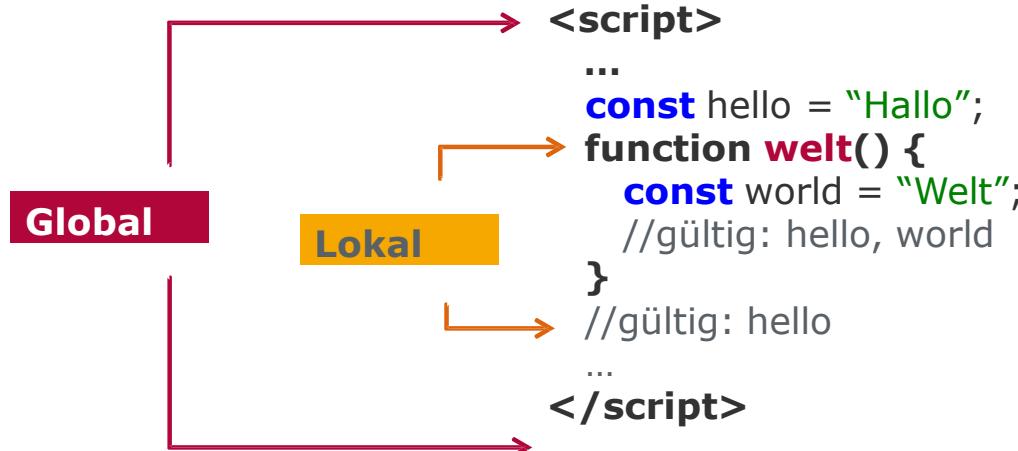
```
// sag "Hallo" nach 2 Sekunden  
function sayHi() { alert('Hello'); }  
setTimeout(sayHi, 2000);
```

```
// schreib "tick" jede Sekunde auf die Konsole  
function ticke() { console.log('tick'); }  
let timerId = setInterval(ticke, 1000);  
  
// nach einer Minute den "tick" stoppen  
setTimeout(() => { clearInterval(timerId); }, 60000);
```

JavaScript – Variablen und Scope



- Gültigkeitsbereich einer Variablen → **Scope**
- JavaScript kennt zwei Scopes
 - **Globaler Scope**
 - **Lokaler Scope**
- Gültigkeitsbereich ist abhängig vom Ort der Definition



JavaScript – Variablen und Scope



- Variablen deklariert mit **let** und **const** sind nur im jeweiligen Block gültig
 - ein Block ist ein Bereich, der mit geschweiften Klammern {} abgegrenzt ist
- **Beispiele:**

```
<script>
```

```
...
let hello = "Hallo";
x = function() {
    let hello = "Welt";
    console.log(hello);
    // gibt „Welt“ aus
}
console.log(hello);
// gibt "Hallo" aus
...
```

```
</script>
```

```
<script>
```

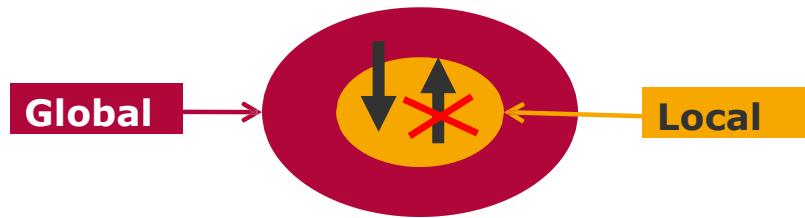
```
...
const hello = "Hallo";
x = function() {
    const hello = "Welt";
    console.log(hello);
    // gibt „Welt“ aus
}
console.log(hello);
// gibt "Hallo" aus
...
```

```
</script>
```

JavaScript – Variablen und Scope



- Gültigkeit einer Variable „diffundiert“ nach innen/unten aber nicht umgekehrt



- globale Variable ist auch innerhalb eines Blocks nutzbar
- lokale Variable ist ausschließlich innerhalb des Blocks nutzbar, in der sie definiert wurde

JavaScript – Arrays



Arrays

- Ermöglichen das Speichern mehrerer Elemente in einer Variable

```
1 const birds = ["Adler", "Papagei", "Spatz"];
```

- Ermöglichen es, über die Elemente zu iterieren

```
1 for (const bird of birds) {  
2   console.log(bird);  
3 }
```

Adler
Papagei
Spatz

- Ermöglichen beliebigen Zugriff auf die Elemente

```
1 console.log(birds[2]);  
2 console.log(birds[0]);
```

Spatz
Adler

- Ermöglichen das Hinzufügen neuer Elemente

```
1 birds.push("Amsel");  
2 console.log(birds);
```

(4) ["Adler", "Papagei", "Spatz", "Amsel"]

JavaScript – Array



Arrays

- Können als eine Art assoziatives Array genutzt werden

```
1 const sparrow = {weightGrams: 25, maxLifetimeYears: 9};
```

- Können um neue Werte erweitert werden

```
1 sparrow.color = "green";
2 sparrow.size = 15;
3 console.log(sparrow);
```

```
{
  weightGrams: 25,
  maxLifetimeYears: 9,
  color: "green",
  size: 15
}
```

- Können Funktionen speichern

```
1 sparrow.chirp = function() {
2   console.log("chirp chirp");
3 };
4 sparrow.chirp();
```

```
chirp chirp
```

JavaScript – Array Funktionen (Auszug)



.push(), .pop(): Elemente einem Array hinzufügen bzw. entfernen

.join(): verbindet die Array Inhalte mit beliebigen Trennzeichen

- `let s = myArray.join(", ")`

.sort(): sortieren der Inhalte eines Arrays

- `let s = myArray.sort();`

.toString(): liefert eine komma-separierte String-Repräsentation des Arrays

.map(): erstellt ein neues Array indem auf jedes Element eine Funktion angewendet wird

- `let map1 = myArray.map(x => x * 2); // seit ES2015`

.filter(): erstellt ein neues gefiltertes Array

- `let map1 = myArray.filter(x => { return x < 10; }); // nur Werte < 10 (seit ES5)`

.forEach(): for-Schleife für Arrays

- `myArray.forEach(x => console.log(x)); // seit ES5`

.splice(): löscht oder ersetzt Elemente im Array

- `myArray.splice(2, 3); // lösche ab index 2, 3 Elemente aus dem Array`

... und viele mehr: siehe developer.mozilla.org

JavaScript – Strings



Strings

- Speichern Zeichenketten

```
1 const helloWorld = "Hello World!";
```

- Können aus mehreren Teilen zusammengesetzt werden

```
1 const hello = "Hello";
2 const world = "World";
3 console.log(hello + " " + world + "!");
```

Hello World!

- Können als Formatstrings einfacher zusammengesetzt werden

```
1 console.log(` ${hello} ${world} !`);
```

Achtung Backtick!

Hello World!

- In Formatstrings können auch direkt Berechnungen ausgeführt werden

```
1 console.log(` ${5 + 5}`);
```

10

JavaScript – DOM Elemente



JavaScript kann DOM Elemente manipulieren

- Wurzelement des DOM ist `document`
- Ausgehend hiervon, können andere Elemente über ...
 - ... ihre ID `querySelector('#foo')`
 - ... ihre Klasse `querySelector('.bar')`
- ... adressiert werden
- Dabei gibt `querySelector` nur ein Element zurück, während `querySelectorAll` alle Elemente zurückgibt, auf die der Selektor passt
- Auf manche Elemente ist auch direkter Zugriff möglich:
 - `document.forms` → gibt alle Formulare in einem Dokument zurück
 - `document.images` → gibt alle Bilder in einem Dokument zurück

JavaScript – DOM Elemente



document.getElementsByTagName(<Name als String>)

Finden von Elementen über den Tagnamen (z.B. "table" für "<table>") - returniert eine Liste von Elementen

document.getElementsByClassName(<Name als String>)

Finden von Elementen über den Klassennamen (z.B. "content" für "<div class='content'>") - returniert eine Liste von Elementen

document.getElementById(<ID als String>)

Finden einen Elements über die id (z.B. 'main-content' für "<div id='main-content'>") – returniert ein einzelnes Element

JavaScript – DOM Elemente



JavaScript kann DOM Elemente manipulieren

- Änderung der Klasse

```
1 const div = document.querySelector(".green");
2 div.classList.replace("green", "red");
```

→ Klasse grün durch rot austauschen

- Änderung des Inhalts

```
1 document.querySelector("#extra").innerHTML = "Gurken";
→ Inhalt des DOM Elements mit der ID extra wird
```

1. Äpfel
2. Kirschen
3. Mangos



1. Äpfel
2. Kirschen
3. Mangos



1. Äpfel
2. Kirschen
3. Gurken

Beispiel (HTML)

```
1 <div class="list green">
2   <ol>
3     <li>Äpfel</li>
4     <li>Kirschen</li>
5     <li id="extra">Mangos</li>
6   </ol>
7 </div>
```

CSS

```
1 .green {
2   color: white;
3   background-color: green;
4 }
5 .red {
6   color: white;
7   background-color: red;
8 }
```

JavaScript – DOM Elemente - window



- In client-seitigen JS stellt das **Window-Objekt** das Fenster in dem das (HTML)-Dokument angezeigt wird virtuell dar
- **window** ist ein globales Objekt
- Es ermöglicht über Eigenschaften und Methoden **Zugriff auf das Fenster**
- **window** ist das obere Ende der Gültigkeitsbereichskette eines Fensters

JavaScript – DOM Elemente - window



```
var antwort = 42;  
window.antwort === 42 // => true
```

Wichtige Objekte von **window**

- document
- history
- location
- ...

Wichtige Methoden und Eigenschaften von **window**

- alert()
- **window** hat ein Attribut „**console**“ mit der Methode „**log**“
console.log()
- print()
- setInterval() / setTimeout()
- close()
- ...

JavaScript – EventHandler



HTML Elemente können direkt mit JavaScript-Funktionalität angereichert werden

- Interaktion des Nutzers mit Element löst dann die entsprechende Aktion aus

```
1  <h1 id="hello">Hello</h1>
2  <script>
3  | document.querySelector("#hello").addEventListener("click", function() {
4  |   alert("World!");
5  });
6  </script>
```



JavaScript – if-Bedingung

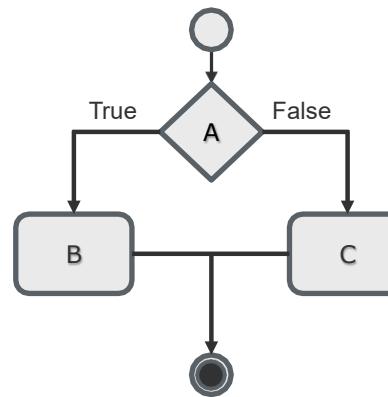


Sollen Skript-Inhalte miteinander verglichen werden, wird **if-Bedingung** verwendet

- Je nachdem, wie Bedingung erfüllt ist, wird eine Aktion ausgeführt
- Es werden Vergleichsoperatoren verwendet

```
Falls Bedingung A dann  
    Anweisung B  
Sonst  
    Anweisung C  
Ende
```

```
IF (A =True)  
    B  
ELSE  
    C  
END IF
```



JavaScript – if-Bedingung



```
1  x = 3;
2  if (x == 6) {
3    alert('Zahlen sind identisch');
4  }
5  alert (x);
```

```
1  x = 3;
2  if (x == 6) {
3    alert('Zahlen sind identisch');
4  } else {
5    alert('Zahlen sind nicht identisch');
6  }
7  alert (x);
```

```
1  x = 3;
2  if (x == 6) {
3    alert('Die Zahlen haben den Wert 6');
4  } else if (x == 3) {
5    alert('Die Zahlen haben den Wert 3');
6  } else {
7    alert('Die Zahlen haben einen anderen Wert');
8  }
9  alert (x);
```

JavaScript – if-Bedingung

Vergleichsoperatoren



| Operator | Bedeutung | Ergebnis der Operation |
|--------------------|---------------|---|
| <code>==</code> | ist gleich | Wahr, wenn die Werte gleich sind |
| <code>!=</code> | ungleich | Wahr, wenn die Werte ungleich sind |
| <code>></code> | größer | Wahr, wenn der linke Wert größer als der rechte ist |
| <code>>=</code> | größergleich | Wahr, wenn der linke Wert größer oder gleich dem rechten ist |
| <code><</code> | kleiner | Wahr, wenn der linke Wert kleiner als der rechte ist |
| <code><=</code> | kleinergleich | Wahr wenn der linke Wert kleiner oder gleich dem rechten Wert ist |

JavaScript – if-Bedingung

Typgenauer Vergleich



2 Gleichsetzungszeichen vs. 3 Gleichsetzungszeichen

```
1  x = 3;
2  if (x == 6) {
3    |   alert('Die Zahlen haben den Wert 6');
4  }
5  alert (x);
```

```
1  x = 3;
2  if (x === 6) {
3    |   alert('Die Zahlen haben den Wert 6');
4  }
5  alert (x);
```

| Operator | Bedeutung | Ergebnis der Operation |
|----------|------------|--|
| == | ist gleich | Wahr, wenn die Werte gleich sind und außerdem auch die Typen gleich sind |
| != | ungleich | Wahr, wenn die Werte ungleich sind oder nicht den gleichen Typ haben |

JavaScript – if-Bedingung



Vergleichsoperatoren können miteinander in einer Bedingung verbunden werden

- Mit "und" (`&&`)
- Mit "oder" (`||`)

```
1  x = 1;
2  if(x > 3 && x < 5) {
3    alert('Die Zahl ist 4');
4 }
```

```
1  x = 3;
2  if (x == 6 || x < 5) {
3    alert('Die Zahl ist 6 oder kleiner als 5');
4 }
5 alert (x);
```

JavaScript – switch case Bedingung



Anstelle von if-Abfragen können **Switch-case Bedingungen** verwendet werden

- Je nach Szenario können switch-case Bedingungen übersichtlicher sein

```
1  x = 3;
2  if (x == 1) {
3    alert('Die Zahl ist 1');
4  } else if (x == 2) {
5    alert('Die Zahl ist 2');
6  } else if (x == 3) {
7    alert('Die Zahl ist 3');
8  } else if (x == 4) {
9    alert('Die Zahl ist 4');
10 } else if (x == 5) {
11   alert('Die Zahl ist 5');
12 } else {
13   alert('Die Zahl ist größer als 5');
14 }
```

```
1  x = 3;
2  switch(x) {
3    case 1:
4      alert("Die Zahl ist 1");
5      break;
6    case 2:
7      alert("Die Zahl ist 2");
8      break;
9    case 3:
10   alert("Die Zahl ist 3");
11   break;
12   case 4:
13     alert("Die Zahl ist 4");
14     break;
15   case 5:
16     alert("Die Zahl ist 5");
17     break;
18   default:
19     alert("Die Zahl ist größer als 5");
20     break;
21 }
```

JavaScript – switch case Bedingung



- Ein Ausdruck (*x*) wird an die Anweisung (*switch*) übergeben
- Der Ausdruck wird dann in den einzelnen Fällen (*case*) überprüft
- Trifft ein Fall zu, so wird die Anweisung des Falles ausgeführt
 - danach wird die Bedingung gebrochen/beendet (*break*)
- Passt der Ausdruck in keinen Fall, so wird die Anweisung des Standard-Falles (*default*) ausgeführt

```
1  x = 3;
2  switch(x) {
3      case 1:
4          alert("Die Zahl ist 1");
5          break;
6      case 2:
7          alert("Die Zahl ist 2");
8          break;
9      case 3:
10         alert("Die Zahl ist 3");
11         break;
12     case 4:
13         alert("Die Zahl ist 4");
14         break;
15     case 5:
16         alert("Die Zahl ist 5");
17         break;
18     default:
19         alert("Die Zahl ist größer als 5");
20         break;
21 }
```

JavaScript – Schleifen



Schleifen gibt es in jeder Programmiersprache

- Entweder werden eine festgelegte Anzahl an Durchgängen durchlaufen oder gestoppt, wenn eine Bedingung nicht mehr zutrifft
- Grundlegender Aufbau:
 - Schleifenbefehl mit einer Bedingung
 - Anweisung / Schleifen-Code
- In JavaScript gibt es **drei Arten** von Schleifen
 - for
 - while
 - do ... while
- **Achtung:** Ein Anfängerfehler sind **Endlosschleifen!**

```
Schleifenbefehl (Bedingung)
{
    Anweisung / Code
}
```

JavaScript – Schleifen

for-Schleife



for-Schleife ist Schleife, bei der Anzahl der Durchläufe bekannt ist

- Alle wichtigen Daten und Anweisungen sind bereits im Schleifenkopf vorhanden
 - Initialisierung der Variablen
 - Schleifenbedingungen
 - Abschließende Anweisung

```
for (
    Variablen Initialisierung;
    Schleifenbedingung;
    abschließende Anweisung;
)
{
    Code
}
```

```
1  for (zahl = 0; zahl < 5; zahl++) {
2      alert(zahl);
3  }
```

JavaScript – Schleifen

for-Schleife



■ Initialisierung der Variablen

- Variable wird initialisiert und kann dann in den Schleifenanweisungen verwendet werden
- im Beispiel: `zahl = 0`

```
1  for (zahl = 0; zahl < 5; zahl++) {  
2      |     alert(zahl);  
3  }
```

■ Schleifenbedingungen

- gibt an, wie lange die Schleife durchlaufen wird
- im Beispiel: so lange die Variable `zahl` kleiner als 5 ist

■ Abschließende Anweisung

- jedes Mal am Ende eines Schleifendurchlaufes wird diese Anweisung ausgeführt
- ist eine Art Zähler
- im Beispiel: Variable `zahl` wird erhöht
 - `zahl++` ist eine vereinfachte Schreibweise für `zahl = zahl + 1`

JavaScript – Schleifen

while-Schleife



while-Schleife ist einfachste Form einer Schleife

- Wiederholt ("iteriert") solange die Bedingung „wahr“ ist
- Stoppt, wenn die Bedingung „falsch“ ist
- Zuerst wird Bedingung geprüft und dann, bei „wahr“, der Schleifen-Code ausgeführt

```
while(Bedingung) {  
    Code  
}
```

```
1  zahl = 0  
2  while(zahl < 5) {  
3      zahl = zahl + 1  
4  }
```

- Im Beispiel:
 - Variable *zahl* wird vor der Schleife auf 0 gesetzt
 - Die Schleifenbedingung gibt an, dass die Schleife so lange durchlaufen wird, bis *zahl* nicht mehr kleiner als 5 ist
 - Im Schleifen-Code wird die Variable *zahl* in jedem Durchgang um 1 erhöht

JavaScript – Schleifen

do-while-Schleife



do-while-Schleife ist eine Anpassung der while-Schleife

- Hier wird zuerst der Schleifen-Code ausgeführt und dann die Bedingung geprüft
- Schleifen-Code wird also in jedem Fall mindestens einmal ausgeführt

```
do {  
    Code  
} while(Bedingung)
```

```
1  zahl = 0  
2  do {  
3      alert(zahl)  
4      zahl = zahl + 1  
5  } while(zahl < 5)
```

JavaScript – Schleifen

while- vs. do-while Schleife



```
1  zahl = 5
2  do {
3      alert(zahl)
4      zahl = zahl + 1
5  } while(zahl < 5)
6
```

```
1  zahl = 5
2  while(zahl < 5) {
3      alert(zahl)
4      zahl = zahl + 1
5  }
```

- Ergebnis der Variable *zahl* ist nach Schleifendurchlauf 6
- Da: Auch wenn die Bedingung nicht zutrifft (Variable *zahl* mit Wert 5 ist nicht kleiner als 5), wird der Schleifen-Code zuvor einmal ausgeführt
- Die Variable *zahl* wird um 1 erhöht, somit ist das Ergebnis am Ende der Schleife 6
- Das Ergebnis der Variable *zahl* ist nach Schleifendurchlauf noch immer 5
- Da: Die Bedingung zuerst geprüft wird und die Variable *zahl* mit dem Wert 5 nicht kleiner als 5 ist
- Somit wird der Schleifencode nicht einmal ausgeführt und *zahl* hat am Ende das Ergebnis 5

Weiterentwicklung von JavaScript



JavaScript wird stetig weiterentwickelt und Neuerungen in Form von ECMAScript-Standards veröffentlicht

- Größtenteils Vereinfachungen der Syntax
(sogenannter „Syntactic Sugar“)
- **Aber:** Browser-Hersteller benötigen Zeit zur Umsetzung
 - Nutzer verwenden möglicherweise alte Browser-Version
- **Transpiler** wandeln neuartigen JavaScript-Code in Browser-kompatiblen JavaScript-Code um
 - ermöglicht Entwicklern, die Nutzung neuer JavaScript-Features bevor Browser diese unterstützen
 - prominentes Beispiel: JavaScript Compiler **Babel**
- ECMAScript 2015 (ES6) kann heutzutage bedenkenlos bei empfohlenen Browsern als „vollständig unterstützt“ betrachtet werden

Objektorientiertes JavaScript mit ECMA 6



Schlüsselwort **class** dient der Definition einer Klasse

- Setzen der Attribute (Zustand) im Konstruktor – Spezielle Methode:

```
1  class Ball { ←  
2    constructor(x, y) {  
3      this.x = x;  
4      this.y = y;  
5    } ←
```



- Hinzufügen von Methoden (Verhalten) für alle Objekte der Klasse:

```
6    moveX() {  
7      this.x += 10;  
8    } ←  
9  } ←
```

- **Wichtig:** Methoden werden *innerhalb* der Klasse definiert

Objektorientiertes JavaScript mit ECMA 6



- Erzeugen von Objekte der Klasse Ball:

```
1 let ball1 = new Ball(10, 10);
2 let ball2 = new Ball(10, 50);
3
4 ball1.moveX();    → ball1.x: 20
5 ball2.moveX();    → ball2.x: 20
```

- Hinzufügen/Überschreiben von Methoden (Verhalten) *einzelner* Objekte:

```
1 ball1.moveY = function () {
2     this.y += 10;
3 }
4
5 ball1.moveY();    → ball1.y: 30
6 ball2.moveY();    → Error - Die Funktion moveY wurde nur
                           auf ball1 definiert
```

Objektorientiertes JavaScript mit ECMA 6



Vererbungen

- Bereits in ECMA 5 möglich (mittels Prototype.chain, Object.create)
- Eleganter in ECMA 6:
 - Eltern-Klasse

```
1  class Ball {  
2    constructor(x, y) {  
3      this.x = x;  
4      this.y = y;  
5    }  
6 }
```



- Kind-Klasse

```
1  class RedBall extends Ball {  
2    constructor(x, y) {  
3      super(x, y);  
4      this.c = "red";  
5    }  
6 }
```



Weitere Neuerungen in ECMA 6



■ Arrow-Funktionen

```
1 // Function Expression
2 const multiplyFunc = function(a, b) { return a * b; }
3 console.log(multiplyFunc(3, 5)); // 15
4 // Arrow function
5 const multiplyArrowFunc = (a, b) => a * b; ←
6 console.log(multiplyArrowFunc(3, 5)); // 15
```

■ optionale Standard-Parameter für Funktionen

```
1 function add(a = 1, b = 2) { return a + b; }
2 add(3, 4); // 7
3 add(2); // 4
4 add(); // 3
```

Weitere Neuerungen in ECMA 6



- **for-of-Schleife** zum Iterieren über alle Werte eines Objekts:

```
1 const prime = [2, 3, 5, 7, 11, 13, 17, 19]
2
3 for (let n of prime) { // Ausgabe jeder Zahl in einer Zeile
4   console.log(n)
5 }
```

- **Template literals** für verbesserte Textkomposition;

```
1 const firstName = 'Joe';
2 console.log(`Hello ${firstName}! Good morning!`);
```



- **Promise** für vereinfachte asynchrone Programmierung

- ...

Neuerungen seit ECMA 7 (Auszug)



■ Asynchrone Funktionen

- erleichtert Lesbarkeit
- mehrere asynchrone Funktionen können gleichzeitig ausgeführt werden

```
1  async function getCharacters() {  
2      // wartet/blockiert bis die Daten geladen wurden  
3      let data = await fetch('https://swapi.dev/api/people');  
4      console.log(data)  
5  }  
6  getCharacters();
```

- Alle Neuerungen in ECMA: <https://github.com/sudheerj/ECMAScript-features>
- Browserkompatibilität von aktuellen ECMA-Standards:
 - <https://kangax.github.io/compat-table/>
 - <https://caniuse.com/>



FH Salzburg

VO Web-Technologien

Einheit 5, Oliver Jung