HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Teil 2: Javascript

Prof. Dr.-Ing. Thorsten Weiss

Version 1.0

Javascript

WEBENTWICKLUNG 2
one step ahead

Javascript:

- **Javascript / Typescript** sind die **führenden Skriptsprachen für Frontends (im Browser)** von Webanwendungen.
- Auch viele **Backends bzw. Teile davon** werden inzwischen in Javascript programmiert.
- **Eine Historie ist z.B. unter https://w3schoolsua.github.io/js/js_history_en.html#gsc.tab=0 zu finden.**
- **Typescript** wird mit **Transpiling** zu Javascript häufig genutzt.
- Ursprünglich wurde Javascript zur **DOM-Manipulation** genutzt (z.B. Eigenschaften ändern, Funktionen ausführen, wenn auf Buttons geklickt wird). Rückständige Projekte machen das noch heute, moderne Systeme bauen auf intelligente Komponenten, weswegen die händische DOM-Manipulation hier nicht behandelt wird.
- Durch **Paketmanager** kann in Javascript mit modernen Programmierfeatures entwickelt werden und zu Paketen zusammen gefügt und in allgemein kompatiblen Code umgewandelt werden.
- Moderne Systeme mit Paketmanagern und **intelligenten Komponenten** machen **das lästige Einbinden von script-Tags glücklicherweise obsolet.**

ECMA Script

- Seit der „Version" ECMA Script 6 wurde Javascript massiv erweitert und weist nun einige wichtige Programmierfeatures auf, die eine sehr saubere (aber auch eine sehr unsaubere) Architektur auf.
- Beispiele für Neuerungen sind eine klare Klassenunterstützung aber auch bessere Werkzeuge, die aus der funktionalen Programmierung inspiriert sind.

Einführung von Javascript

- Um mit Javascript zu üben wird eine Umgebung, die Javascript interpretieren kann, benötigt.
- Falls Visual Studio Code genutzt wird, kann hier beispielsweise eine Erweiterung Javascript RECL gentzt werden (siehe Anleitung und Video in Moodle)

Grundlegende Sprachfeatures

Dieses Kapitel setzt Programmierkenntnisse voraus.

Dynamische Typisierung

- **Java ist eine typisierte** Programmiersprache. Bevor das Programm übersetzt wird, muss der Typ einer Variablen explizit angegeben werden, sonst kann das Programm nicht übersetzt werden.
  Beispiele:

```
Integer noCounts = 0;
String myString = "Hallo";
```

- Javascript unterstützt die dynamische Typisierung. Alle Variablen sind Objekte und werden mit dem Schlüsselwort **let** gekennzeichnet. Der Typ wird aus dem Kontext ermittelt:

```
let noCounts = 10;
```
Javascript erkennt, dass es ein Integer sein soll, weil 10 eine Ganzzahl ist.

```
let myString = "Hallo";
```
Javascript macht myString zu einer Variablen vom Typ String.

```
let temp = 23.3;
```
Da die Zahl ein Dezimalpunkt hat, wird der Datentyp Double.

- Interessant ist, dass sich der Typ zur Laufzeit ändern kann.

```
let myVar = "Hallo";
myVar = 10;
```

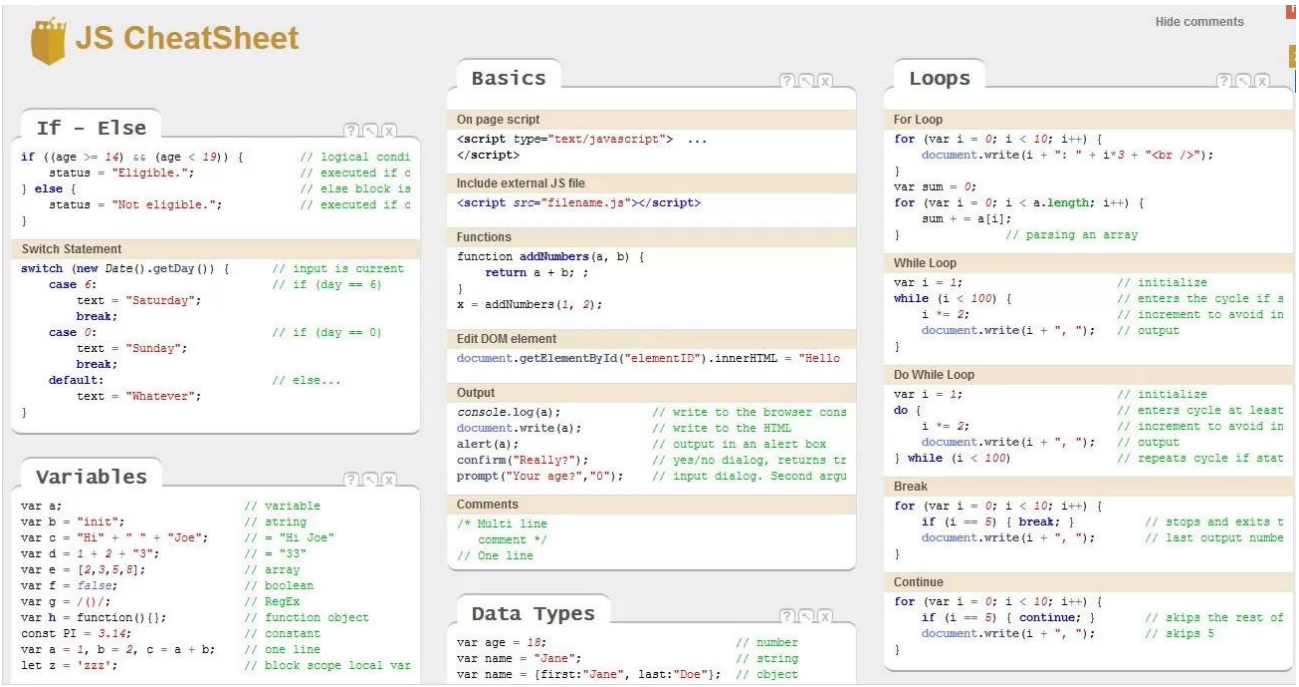
Ausprobieren:

```
let myVar = "Hallo";
console.log(myVar);
myVar = 10;
console.log(myVar);
```

Javascript – Strings

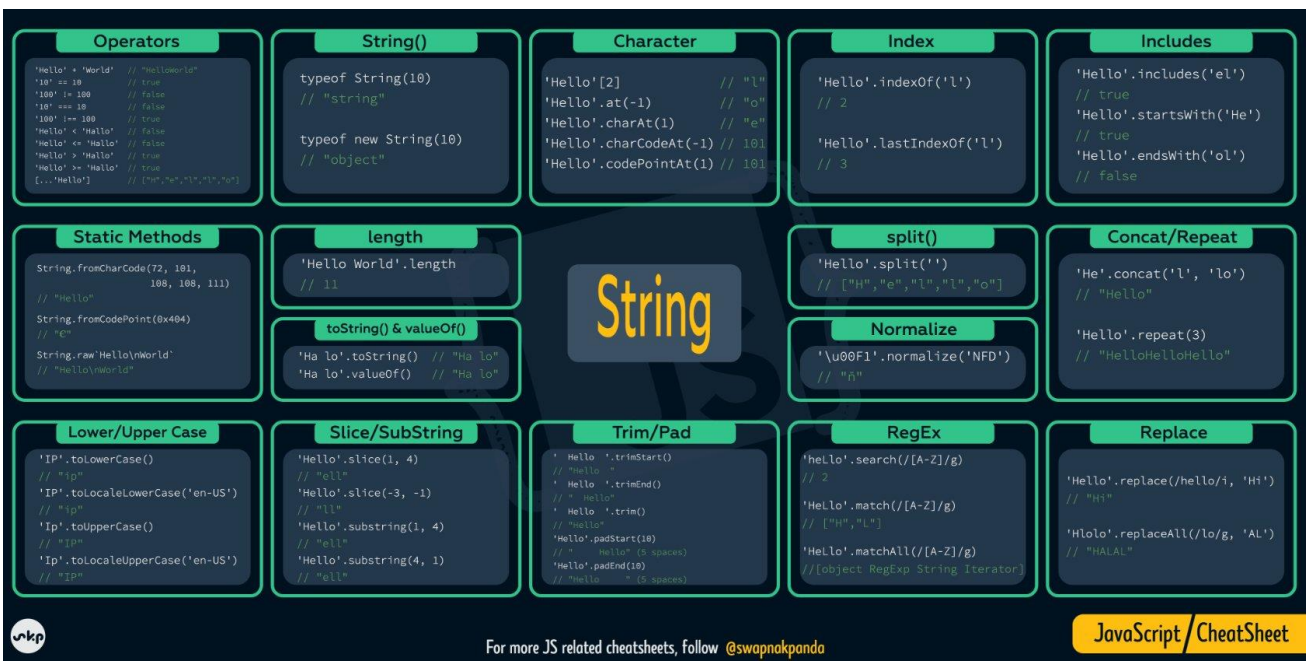
Javascript Primitiven

- Folgender Cheatsheet enthält die wichtigsten Funktionen
- https://htmlcheatsheet.com/js/



Strings

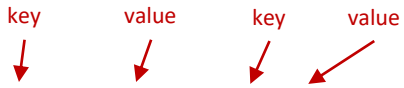
- In der Webentwicklung wird viel mit Strings gearbeitet. Ein Grund ist, dass viele Daten als Strings übertragen werden und auch zahlreiche Texteingaben, die verarbeitet werden müssen.
- Daher wurden zur Verarbeitung von Strings zahlreiche Funktionen in Javascript integriert.
- Folgender Cheatsheet kann als gute Übersicht dienen. Quelle: https://twitter.com/swapnakpanda/status/1450798415164555266/photo/1



Javascript – Objekte und Arrays

Objekte

- Javascript erlaubt den einfachen **Umgang mit Objekten**. Ein Objekt kann aus eindeutigen **Keys und Values** bestehen.




```
let myObj = { name: "Karl", age: 30 } ;
```

- Der Zugriff auf die Attribute erfolgt über Punkt-Notation:

```
let myName = myObj.name;   ← "Karl"
```

Arrays

- Aufgrund der dynamischen Typisierung können Arrays verschiedene Datentypen bzw. Objekte enthalten. (vgl. bei Collections ist nur ein Objekt Typ möglich)

```
let myArr = [0,3,5,6,7,8];
```

Objekt mit key age

Objekt mit key zip

```
let myObjArr = [{ name: "Karl", age: 30 } , { name: "Fritz", zip: 88250}];
```

- Dies ermöglicht sehr flexible Datenstrukturen.
- Der **Zugriff kann über den Index erfolgen**. Indizes fangen immer bei 0 an:

```
let names = ["Fritz", "Lea", "Robert"];

let aName = names[0]; ← Fritz
aName = names[1]; ← Lea
```

- **Auch die Änderung einzelner Elemente** kann über den Index erfolgen:

```
names[0] = "Karl" ← ["Karl", "Lea", "Robert"];
```

- Der Zugriff auf Objekte des Arrays kann über den Index und Punktnotation erfolgen.

```
let myObjArr = [{ name: "Karl", age: 30 } , { name: "Fritz", zip: 88250}];

let karlName = myObjArr[0].name; ← Karl
```

- Falls es das Objekt oder den Key nicht gibt:

```
let invalidField = myObjArr[0].aField; ← gibt undefined zurück.
```

- Diese Abfrage oder ermittelt somit, ob das Feld korrekt angegeben wurde und existiert.

```
if (invalidField == undefined)  {

}
```

- Alternativ: hasOwnProperty

```
if (myObjArr[0].hasOwnProperty('name' == undefined)  {

}
```

Javascript - Arrays

WEBENTWICKLUNG 2
one step ahead

Gängige Funktionen für Arrays

Funktion	Nutzen
pop()	**Entfernt** das **letzte Element** im Array
push()	Fügt **ein Element am Ende hinzu** ['Fritz', 'Karl'].push('Rita');
shift()	**Shifted** die Elemente nach links und gibt das gelöschte Element zurück `let fruits = ["Banana", "Orange", "Apple", "Mango"];` `fruits.shift()` → Rückgabe: Banana → fruits ["Orange", "Apple", "Mango"];
unshift(new)	**Shifted** die Elemente und fügt neues Element hinzu. `let fruits = ["Orange", "Apple", "Mango"];` `fruits.shift("Peach")` → Rückgabe: ArrayLänge 4 → fruits → ["Peach","Orange", "Apple", "Mango"];
delete arr[i]	**Löscht** das i-te Elemente `let fruits = ["Orange", "Apple", "Mango"];` `delete fruits[1]` → fruits → ["Orange", "Mango"];

concat	Arrays **verbinden** und als neues Objekt zurück geben
	`const myGirls = ["Cecilie", "Lone"];`
	`const myBoys = ["Emil", "Tobias", "Linus"];`
	`const myChildren = myGirls.concat(myBoys);`

splice	Neue Elemente im Array einfügen
	`const fruits = ["Banana", "Orange", "Apple", "Mango"];`
	`fruits.splice(2, 0, "Lemon", "Kiwi");`
	→ `["Banana", "Orange", "Lemon", "Kiwi", "Apple", "Mango"]`

slice	Elemente aus einem Array ausschneiden
	`const fruits = ["Banana", "Orange", "Apple", "Mango"];`
	`fruits.splice(0, 1);` → `["Orange", "Apple", "Mango"]`

toString	Zu String umwandeln (Prüfen, was bei Objekten passiert!) `const fruits = ["Banana", "Orange", "Apple", "Mango"];`

Quelle: W3schools

Arrays iterieren

- Das Iterieren von Arrays wird an vielen Stellen benötigt. Oftmals kommen Daten in Form von Arrays (z.B. Tabelleninhalte), die beispielsweise formatiert, umgewandelt oder weitere Operationen ausgeführt werden muss.
- Folgendes Konstrukt ist aus vielen Programmiersprachen bekannt:

```
let persons = [{ name: 'Karl', age: 30 } , { name: 'Fritz', age: 52}];
for (let i=0 ; i < persons.length ; i++) {
    console.log(persons[i].name + " is " + persons[i].age + " years old")
}
```

- Javascript erlaubt eine **elegantere Formulierung – bei jedem Durchgang wird ein Element aus dem Array entnommen, bis alle Elemente durchlaufen sind:**

```
let persons = [{ name: 'Karl', age: 30 } , { name: 'Fritz', age: 52}];
for (let person of persons) {
    console.log(person.name + " is " + person.age + " years old")
}
```

Javascript – Kopien/Referenzen

WEBENTWICKLUNG 2
one step ahead

Mutable / Referenzverhalten

- Javascript arbeitet normalerweise mit Referenzen – dies ist bei Objekten und Arrays zu beachten.
- Beispiele:

```
let persons = [{ name: 'Karl', age: 30 } , { name: 'Fritz', age: 52}];
for (let person of persons) {
    person.name = person.name + " is his name";
    console.log(person.name + " is " + person.age + " years old")
}

for (let i=0 ; i < persons.length ; i++) {
    console.log(persons[i].name + " is " + persons[i].age + " years old")
}
```

Wenn hier ein Wert geändert wird, wird dies auf dem „Original“ gemacht. Das heißt die Änderungen in dem Objekt bleiben bestehen.

Output:

```
'Karl is his name is 30 years old'
'Fritz is his name is 52 years old'
'Karl is his name is 30 years old'
'Fritz is his name is 52 years old
```

- **Beispiel Objekt:**

```
let aPerson = { name: 'Karl', age: 30 };
let refPerson = aPerson;
refPerson.name="Fritz";
console.log(aPerson.name);
```

Dies ist eine Referenz (wie in Java)
Der Inhalt des Objekts wird **NICHT** kopiert.

Output:

```
Fritz
```

- Das **Kopieren eines Arrays ist oftmals** nötig. Dies kann über den folgenden Operator gemacht werden. (Dies geht erst ab ECMA Script 6)

```
let fruits = ["apple", "peach", "cherry"];
let copyFruits = [...fruits];
copyFruits[0] = "peach";
for (let fruit of fruits) {
    console.log(fruit);
}
```

Hier wird eine Kopie angefertigt und die Änderung des Attributes wird nicht auf dem Original ausgeführt.
… heisst Spread-Operator

```
Output
'apple'
'peach'
'cherry'
```

- **Achtung – beim Kopieren von Arrays von Objekten** werden **nur die Referenzen** kopiert, nicht deren Inhalt:

```
let persons = [{ name: 'Karl', age: 30 } , { name: 'Fritz', age: 52}];
let copyPersons = [...persons];
copyPersons[0].name = 'Rita';
for (let person of persons) {
    console.log(person.name + " is " + person.age + " years old")
}
```

Da nur Referenzen kopiert werden, werden die Attribute auch in persons geändert.

```
Output:
'Rita is 30 years old'
'Fritz is 52 years old'
```

- Soll auch der Inhalt der Objekte kopiert werden, kann dies über die map-Funktion geschehen (siehe unten):

```
persons = [{ name: 'Karl', age: 30 } , { name: 'Fritz', age: 52}];
let realCopyPersons = persons.map(person => person);
copyPersons[0].name = 'Rita';
for (let person of persons) {
    console.log(person.name + " is " + person.age + " years old")
}
```

Map gibt immer ein neues Array von Objekten und nicht deren Refernz zurück. Daher wird persons nicht verändert

Output:

```
'Karl is his name is 30 years old'
'Fritz is his name is 52 years old'
```

Javascript - Functions

WEBENTWICKLUNG 2
one step ahead

Funktionsparameter

- Javascript kann Variablennamen auch Funktionen zuordnen, die als Parameter aufgerufen werden können.

```
let mySum = function sum(a,b) { return a+b };

console.log(mySum(3,5));
```

- Arrow-Functions stellen eine Kurzschreibweise für Funktionen dar. Diese werden häufig genutzt:

```
() => expression
```
→
```
function() {
    return expression;
}
```

```
(param1, param2) => expression
```
→
```
function(param1, param2) {
    return expression;
}
```

```
(param1, param2) => { statements }
```
→
```
function(param1, param2) {
    // statements
    return expression;
}
```

- Arrow Functions werden als anonyme Funktionen bezeichnet, da sie keinen Namen haben.
- Arrow Functions können **auch wie eine Variable zugwiesen werden** und haben dann einen Bezeichnet (sind streng genommen nicht mehr anonym). Funktionen können als Parameter „injiziert“ werden.

```
let myAlgo = (aCustomFunc) => {
    let a = 4;
    let b = 5;
    let result = aCustomFunc(a,b);
    console.log("custom function result: " + result);
    return result;
}

myAlgo(mySum);
```

- Objekte können somit auch Attribute enthalten, die Funktionen sind.

```
let myAlgoObject = {
    myVar: 1,
    myString: "Hallo",
    myFunc: () => {console.log("Hallo")}
}

myAlgoObject.myFunc();
```

Javascript – map, filter, find

WEBENTWICKLUNG 2
one step ahead

Filtern von Arrays

- Javascript bietet eine vereinfachte Syntax zum Filtern von Objekten.
- Klassisch könnte die Formulierung so lauten:

```
let somePersons = [{ name: 'Karl', age: 31 } , { name: 'Fritz', age: 52}, { name: 'Lea', age: 23}];
let olderThan30 = [];
for (let person of somePersons) {
    console.log(person)
    if (person.age >= 30) {
       olderThan30.push({name: person.name, age: person.age})
    }
}
```

Kopieren der Inhalte und anfügen an das Array olderThan30

- Die Filter-Funktion kann dies in Kurzschreibweise erledigen. Hierzu werden anonyme Funktionen (siehe unten) genutzt. Filter iteriert das Array durch und übernimmt jene Elemente für die ein true zurück gegeben wird.

```
let somePersons = [{ name: 'Karl', age: 31 } , { name: 'Fritz', age: 52}, { name: 'Lea', age: 23}];
let olderThan30 = somePersons.filter(person => person.age >= 30);
```

Filter legt immer „neue Arrays an“, keine Referenzen.

Filter übernimmt jene Arrayelemente, deren Attribut age >= 30 ist und damit true liefert.

Suchen eines Elements:

- Die Find Function gibt das erste gefundene Element wieder, das der Bedingung genügt.

```
somePersons = [{ name: 'Karl', age: 31 } , { name: 'Fritz', age: 52}, { name: 'Lea', age: 23}];
let findKarl = somePersons.find(person => person.name === 'Karl');
```

enthält das erste Element, für das die Bedingung erfüllt ist.

- Die Funktion liefert undefined, falls kein Element gefunden wurde.

```
let somePersons = [{ name: 'Karl', age: 31 } , { name: 'Fritz', age: 52}, { name: 'Lea', age: 23}];
let findAnna = somePersons.find(person => person.name === 'Anna');
if (findAnna === undefined) {
    console.log("Anna nicht enthalten")
}
```

Abprüfen, ob Objekt gefunden wurde

Komfortables Verändern und Verarbeiten von Elementen

- Die map-Funktion iteriert durch ein Array. Der Return-Wert wird dem neuen Array zugewiesen.
- map erstellt Kopien. Es lässt das Original unberührt:

```
somePersons = [{ name: 'Karl', age: 31 } , { name: 'Fritz', age: 52}, { name: 'Lea', age: 23}];
let personStrings = somePersons.map(person => "Name: " + person.name + " is " + person.age + " years old");
console.log(personStrings)
```

Das ursprüngliche Element und das zurückgegebene Element personString ist ein Array aus 3 Strings

```
Output:
Name: Karl is 31 years old
Name: Fritz is 52 years old
Name: Lea is 23 years old
```

const, let, var

```
// Konstante erstellen
const MAX = 10;

// Konstante kann nicht überschrieben werden
MAX = 5;→ Fehler

// eine Variable erstellen, die innerhalb des Blocks gültig ist.
let value = 8;
value = 10;→ ok

// Konstantes Objekt
const obj = {};

// Werte der Attribute können geändert werden
obj.a = 10; → ok

// der Typ kann nicht verändert werden.
obj = 100; → Fehler

// var gilt auch außerhalb der {} – Blöcke und ist damit fehleranfällig für Doppelnennungen.
var myVar = 100;
```

Empfohlen: Schlüsselwort let statt var verwenden, let-Variablen existieren nur innerhalb der {} Blöcke (wie lokale Variablen in Java)

Template Strings

```
var name = "Karl";
console.log(`Mein Name ist ${name}.`);
```

Platzhalter für Strings erlauben schnelle, übersichtliche Stringgeneration (vgl. Swift)

```
var multiline = `
<div>
<p>String über mehrere Zeilen</p>
</div>
`;
```

Mehrzeilige Strings

Wichtige neue Features in ES6 (2)

Spread Operator

Alle Elemente aus parts hier einfügen.

```
var parts = ['shoulders', 'knees'];
var lyrics = ['head', ...parts, 'and', 'toes']; → 'head', 'shoulders', 'knees', 'and',
'toes,
```

Klassen und Vererbung

```
// ES5 Methoden

var Vehicle = function(name) {
        this.name = name;
}
Vehicle.prototype.drive = function()
{
        console.log("Driving ",this.name);
};
```

Basisklasse

Methode

```
class Vehicle {
  constructor(name) {
     this.name = name;
  }
  drive() {
     console.log("Driving ", this.name);
  }
}
```

Abgeleitete Klasse

```
class Car extends Vehicle {
    constructor(name, brand) {
        super(name);
        this.brand = brand;
    }
}
```

Default Argumente in Funktionen Vererbung

```
// ES5
function Person(name, age) {
   this.name = name || " John";
   this.age = age || 25;
}
```

```
// ES6
function Person(name = "John", age = 25) {
   this.name = name;
   this.age = age;
}
```

Objektdestrukturierung

Erweiterung des Prototypes um die Funktion drive

```
// ES5
const name = this.obj.name;
const age = this.obj.age;


const name = "Karl";
obj = { name: name }
```

Dies wird in in modernen SPA sehr oft verwendet!

Auch mehrere Variablen in einer Zeile

```
//ES6
const {name} = this.obj;
const {name, age} = this.obj;


const name = "Karl";
const obj = { name }
```

Hoisting

```
{...
   myInt = 4;
   ...
   var myInt;
}
```

Alle Variablendeklarationen werden an den Beginn des Blocks { … } „geschoben“
Wenn also eine Variable weiter unten im Code deklariert wird, so ist diese im ganzen Block deklariert. (myInt ist also keine globale Variable)

Geht, aber schlecher Stil