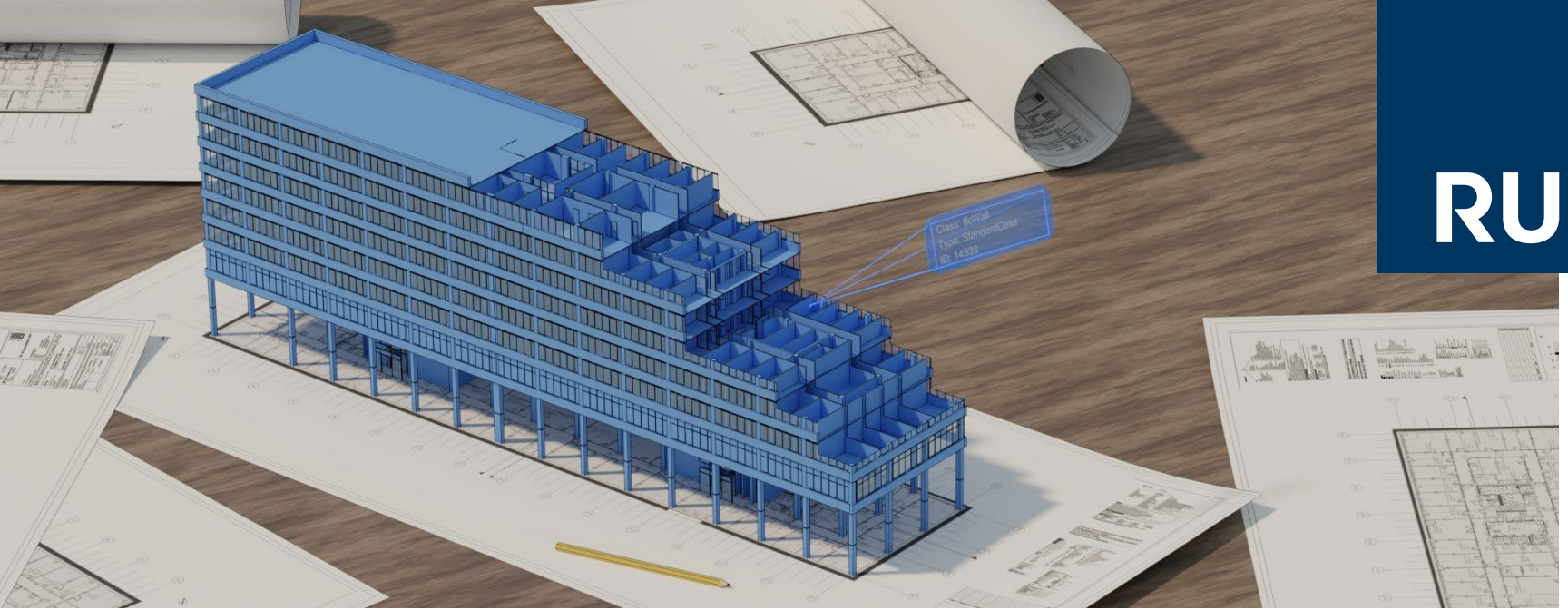


RUB

RUHR-UNIVERSITÄT BOCHUM

WEB-ENGINEERING

Informatik
im Bauwesen

Sommersemester 2023

JavaScript (JS)
Eventhandling

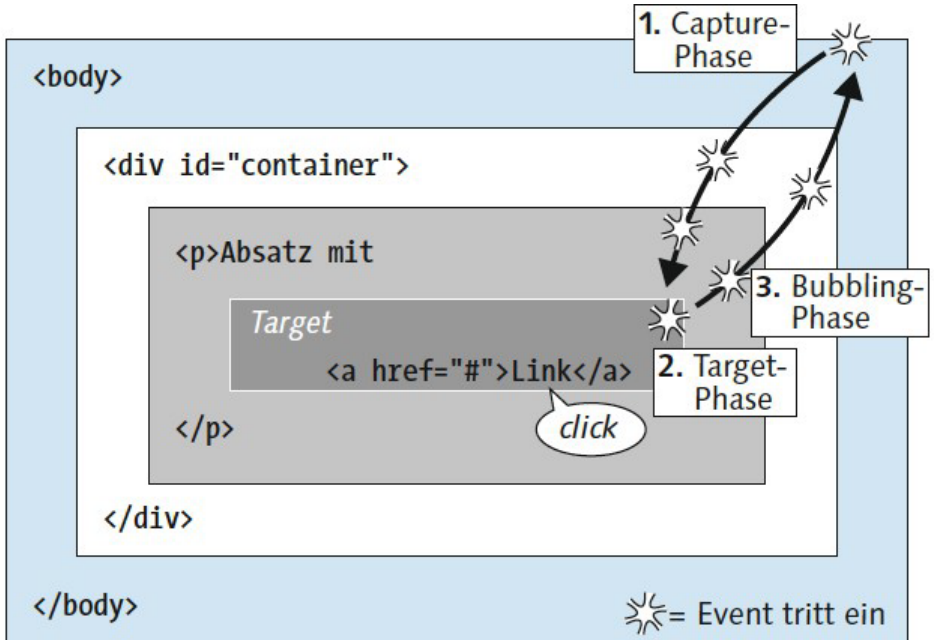
Eventhandling

Ereignisse in einer HTML-Seite treten auf, wenn etwas mit beliebigen Elementen der Seite geschieht. Damit ein Event behandelt werden kann, sind drei Dinge von Bedeutung:

- Das Ereignis muss bemerkt werden
- Es muss bekannt sein, um was für ein Ereignis es sich handelt, an welchem Element und bei welchen Koordinaten es auftritt
- Eine Funktion muss mit dem Auftreten des Events verknüpft sein

Eventhandling

- Die Erfassung eines Ereignisses beginnt stets an der DOM-Wurzel
- Es wandert dann bis zum Target – dem eigentlichen Ort
- An jedem Element, das dabei passiert wird, kann ein Listener das Ereignis erfassen
- Anschließend feuert das Event am Target selbst und wandert dann von dort den Baum wieder nach oben
- Auch hier kann ein Ereignis an jedem Punkt des Weges über einen Listener erkannt und mit einem Handler verarbeitet werden



Eventhandling

Zur Erfassung von Ereignissen wird ein Event-Listener benötigt.

- Event-Objekte speichern alle wesentlichen Informationen.
- Die Verarbeitung eines Ereignisses erfolgt mit Hilfe eines Event-Handlers bzw. einer Callback-Funktion.

```
// Referenz auf das zu beobachtende DOM-Element:
var meinP = document.getElementById('p1');
// den Listener an die DOM-Referenz binden:
meinP.addEventListener('click', function() {
                    alert('Ein Ereignis trat auf.');
}, false);
```

Eventhandling

Es gibt eine Vielzahl unterschiedlicher Event-Typen, welche bei unterschiedlichen Ereignissen ausgelöst werden:

```
target.addEventListener(type, listener[, options]);
```

Interface Events	Mouse Events	Keyboard Events	Form Events	Mutation Events	CSS Events
load / unload	click / dblclick	input	submit	DOMNode Inserted	transitionend
error	mousedown / mouseup	keydown / keyup	change	DOMNode Removed	animation start
resize	mouseover / mouseout	keypress	input	DOMSub treeModified	animation end
scroll	mousemove				animation iteration

Vollständige Auflistung unter: https://developer.mozilla.org/en-US/docs/Web/Events

Eventhandling

- Das Event-Objekt besitzt abhängig vom eingetretenen Ereignis unterschiedliche Eigenschaften
- Das Event-Objekt wird immer dem Event-Handler übergeben und kann auch dort verarbeitet werden
- In welcher der Phasen ein Ereignis erkannt wird, hängt davon ab, wie der Listener definiert worden ist

```
meinP.addEventListener('click', function(e) {
                    alert('Ein ' + e.type + '-Ereignis trat auf.');
}, false);
```

Soll ein Eventlistener für die Capture-Phase definiert werden, bekommt der dritte Parameter der `addEventListener`-Methode den Wert true

Eventhandling

- In der Capture-Phase wird das Ereignis am Body erfasst, bevor es das eigentliche Target erreicht hat. Es ist auch möglich, die Weitergabe zu unterbinden und sogleich die Bubbling-Phase zu starten

```
document.body.addEventListener('click', function(e){
                    if(confirm('Capture anhaltent?')) e.stopPropagation();
}, true);
```

Verschiedene Methoden zur Event-Propagation

Methode	Erläuterung
event.isImmediatePropagationStopped()	Speichert den Aufruf von event.stopImmediatePropagation().
event.isPropagationStopped()	Speichert den Aufruf von event.stopPropagation().
event.stopImmediatePropagation()	Verhindert das Feuern weiterer Event-Handler der gleichen Registrierungsgruppe.
event.stopPropagation()	Verhindert Weitergabe des Ereignisses an folgende Observer.

Eventhandling

- Manche HTML-Elemente besitzen ein Defaultverhalten gegenüber bestimmten Events. Der über einen Listener an das Element gebundene Handler wird stets ausgelöst, bevor das Defaultverhalten in Aktion tritt

```
meinP.addEventListener('click', function(e) {
          e.preventDefault();
          alert('Link wird überprüft und ist derzeit deaktiviert');
    }, false);
```

Property/Methode	Erläuterung
`event.isDefaultPrevented()`	Gibt den Zustand von `event.preventDefault()` als `true` oder `false` wieder.
`event.preventDefault()`	Verhindert die Defaultaktion, die mit dem Ereignis verbunden ist.

JavaScript (JS)
ECMAScript 6 (ES6)

ES6 Einführung

Bei ECMAScript 6 (ES6) handelt es sich um eine Spezifikation zur Standardisierung von JavaScript

- Seit ES6 werden die Versionen nach Ihrem Release-Jahr benannt
  - ECMAScript 6 → ECMAScript 2015
  - Aktuellster Release ECMAScript 2020
- Die Neuerung von ES6 waren gewaltig für die Syntax von JavaScript (Major Release)
  - Viele dieser Neuerungen gehören mittlerweile zum gängigen Gebrauch
  - Erst seit 2018 von allen gängigen Browsern unterstützt
  - Die Neuerungen von 2015 – 2020 ergänzen die ES6 Grundlage sinnvoll

ES6 Einführung

Viele Neuerungen von ES6 wurden bereits vermittelt:

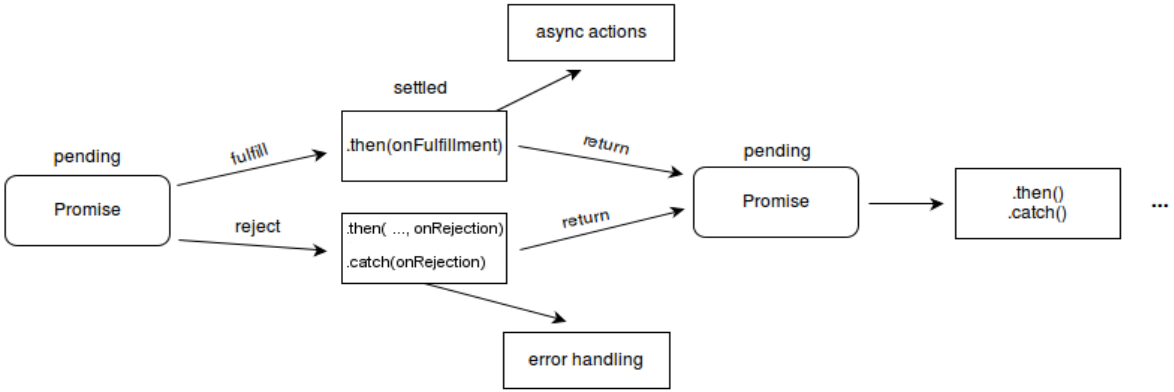
- Deklaration von Variablen mit `let` und Konstanten mit `const`
- Definition von `Arrow Functions`
- Verbesserter `Block-Scope` (Gültigkeitsbereiche)
- Rest Parameter (`...variable`)

Was ist noch Wichtig zu Kennen?

- Verwendung von `Promises` bei Callbacks
- `async` und `await` (seit 2017)
- Klassen in ES6
- Vererbungskonzept
- Neue Built-In Methoden
  - Number Methoden
  - String Suche
  - Array Find und Find Index

Promises

- Ein Promise repräsentiert einen Inhalt, der noch nicht zur Verfügung steht. Dabei wird „versprochen“ den Inhalt nachzuliefern
- Ermöglicht es intelligent Eventhandler zu nutzen um asynchron zu arbeiten
- Ein Promise kann drei Zustände haben – pending, fulfilled und rejected
- Es können Timeouts genutzt werden um die Zeit zu begrenzen



Promises

`Promises` sind ein optimierter Weg um mit JavaScript asynchron zu arbeiten

```
let promise = new Promise(function(resolve, reject) {
        if(promise_erfuellt){
                resolve("erledigt");
        }else{
                reject(new Error("Fehler"));
        }
});
promise.then(function(val){ … }); //wenn status fullfilled dann …
```

- Der Exekutor akzeptiert zwei Parameter `resolve & reject` (`Callbacks`)
- Ein `Promise` kann einen der drei Zustände haben
  - `pending` → initialer Status
  - `fulfilled` (erfüllt) → Operation erfolgreich
  - `rejected` (zurückgewiesen) → Operation gescheitert

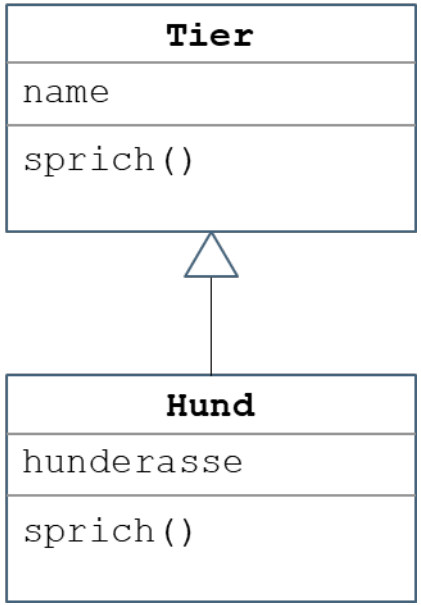
Async await

`Async` ist eine besondere Syntaxform der `Promises`

- Das `async` Keyword wird vor der Funktion angegeben, in der `await` genutzt werden soll
- Funktionen mit dem Keyword `async` geben ein `Promise` zurück
- `await` lässt JavaScript warten bis das `Promise` erfüllt ist

```
async function asyncCall() {
       console.log('calling');
       const result = await resolveAfter2Seconds();
       console.log(result);
}
```

Klassenkonzept



Was sind Klassen?
Klassen dienen als `Bauplan` für Objekte

- Methoden geben das Verhalten vor:

  `sprich()` → Führt Anweisungen aus, um das Verhalten zu bewirken.

- Attribute geben die Eigenschaften vor:

  `name` → Enthält Information des Objekts.

Vererbung

- Damit nicht jede Klasse komplett neu Konstruiert werden muss, wird im Umgang mit Klassen meist Vererbung genutzt
- Wenn Vererbung genutzt wird leitet eine spezifizierte Klasse das Verhalten und die Eigenschaften einer generalisierten Klasse ab

ES6 Klassen

Mit ES6 wurde in JavaScript das Konzept der Klassen eingeführt.

- Klassen werden in JavaScript mit dem Schlüsselwort class definiert:

```
class Testklasse{
  constructor(){
    this.name = "Test";
  }
}
```

- Die Konstruktor-Methode `constructor` ist eine besondere Methode zum instanziieren eines Objektes
- Die Konstruktor-Methode einer Klasse wird mit dem Schlüsselwort new aufgerufen:

```
let erstesObjekt = new Testklasse();
```

ES6 Klassen

Beispiel einer Klasse in ES6

```
class User{
    constructor(name, alter, email){
        this._name = name;
        this._alter = alter;
        this._email = email;
    }
    get name() {
        return this._name;
    }
    set name(neuerName){
        this._name = neuerName;
    }
}
```

Mit `this` können Variablen referenziert werden, die sich im Scope innerhalb eines Objekts befinden.

`Setter-` und `Getter-`Methoden werden genutzt, um das Verhalten beim setzen und zurückgeben von Objektinhalt zu steuern.

```
const alex = new User('Alex', 25, 'Alex@email.com');
console.log(alex.name); //Output Alex

alex.name = 'Hans';
console.log(alex.name); //Output Hans
```

ES6 Klassen

Vererbung und die Prototypenkette

- Die in JavaScript genutzt Form der Vererbung ist nicht wie in Java (OOP)
- Klassen sind in ES6 als syntaktischer Zucker zur prototypenbasierten Vererbung zu verstehen
  - In Java kann ein Objekt über seine Herkunft definiert werden
- Das Verhalten einer Instanz ist aus diesen Informationen abzuleiten:
  - Von welchen Klassen wird geerbt?
  - Welche Interfaces implementiert die Klasse?
- In JavaScript definiert sich ein Objekt über sein Verhalten.
  - Ein Objekt ist nicht das Ergebnis seiner Vererbungshierarchie
  - Mit dem Konzept der prototypenbasierten Vererbung, wird das Erstellen eines Objekts erweitert

ES6 Klassen

Vererbung und die Prototypenkette

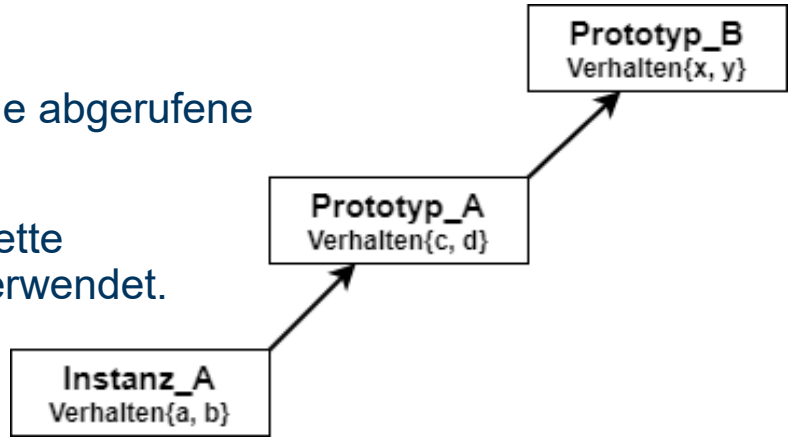
- In JavaScript kann man einem Konstruktor ein Prototyp übergeben.
  - Als Prototyp kann jeder gültige Konstruktor mit einer Prototypeigenschaft eingesetzt werden (andere Objekte, Funktionen, Funktionsaufrufe, etc…)
- Einem Prototyp kann ebenfalls in seinem Konstruktor ein weiterer Prototyp übergeben werden
  - Dieser Vorgang kann beliebig oft wiederholt werden, jedoch wird er immer mit dem letzten Prototypen null abgeschlossen
  - So wie null immer das Ende der Prototypenkette bildet, bildet Object immer den Start
  - Das Ergebnis ist eine Kette aus Prototypen → Die Prototypenkette
    - eines Objekts erweitert

ES6 Klassen

Vererbung und die Prototypenkette

Ablauf der Prototypenkette:

1. Beispielsweise wird `Instanz_A` nach der Methode y gefragt
2. `Instanz_A` weist diese Verhalten nicht auf, also wird das nächste Objekt in der Prototypenkette gefragt
3. Das `Prototyp_A`-Objekt hat ebenfalls nicht die abgerufene Funktionalität
4. Das dritte und letzte Objekt in der Prototypenkette `Prototyp_B` hat die Methode `y`, diese wird verwendet.



ES6 Klassen

Vererbung

In diesem Beispiel erbt die Klasse Administrator von der Klasse User

```
class Administrator extends User {
    constructor(name, alter, email, role) {
        super(name, alter, email);
        this._role = role;
    }
    get role() {
        return this._role;
    }
    set role(newRole){
        this._role = newRole;
    }
}
const lea = new Administrator('Lea', 32, 'lea@email.com', 'Admin');

console.log(lea.name); //Lea
console.log(lea.role); //Admin
```

`extends` lässt Administrator von User erben

`super` greift auf den Konstruktor der Elternklasse (User) zu

ES6 Klassen

Vererbung

- Diese Form der Vererbung hat einige Nachteile.
  - Auch wenn es die aufgerufene Methode nicht gibt, wird dennoch einmal die ganze Kette durchlaufen, da eine Instanz nicht die Informationen all seiner verketteten Objekte kennt
  - Durch die mit unter langen Prototypenkette kann es zu einer Vergleichsweisen schlechten Performance kommen
- Um sicherzustellen, dass der Aufruf in unserem Objekt erfolgen kann, gibt es folgende Methode in JS:

```
console.log(Instanz_A.hasOwnProperty('property1'));
// expected output: true
```

ES6 Klassen

Private Attribute und Methoden zur Kapselung des Programms

- Die Kapselung der Programmbestandteile führt zu einem kontrollierten Zugriff auf Methoden und Attribute einzelner Klassen
- Der Zugriff auf Methoden und Attribute einer Klasse erfolgt über ein dafür vorgesehenes Interface z.B. Getter- & Setter-Methoden
  - Durch diese Vorgehensweise werden unerwartete Lese- und Schreiboperationen unterbunden (Geheimnisprinzip)
- In JavaScript gab es bis einschließlich ES6 kein spezielles Keyword um Variablen und Methoden `private` zu setzen!
- Es gibt verschiedene Techniken, mit derer Lese- und Schreibzugriff eingestellt werden kann

ES6 Klassen

Private Attribute und Methoden zur Kapselung des Programms

- Kein Geheimnisprinzip angewandt, da der Lese- und Schreibzugriff immer noch gestattet ist
- Zeigt dem Entwickler, dass die Variable `private` sein soll (Notation mit `_name` fungiert als Warnung)

```
class User {
    constructor(name, alter, email) {
        this._name = name;
        this._alter = alter;
        this._email = email;
    }

    getName = () => this._name;

    set name(neuerName) {
        this._name = neuerName;
    }
}
```

ES6 Klassen

Private Attribute und Methoden zur Kapselung des Programms

- Eine Funktionale Kapselung, ermöglicht die Abkapselung nach außen durch den `function scope` mit var–Deklaration
- Getter & Setter-Methoden → Ermöglichen den Zugriff auf die Attribute des Objekts

```
class User {
    constructor(name, alter, email) {
        var _name = name;
        var _alter = alter;
        var _email = email;
    }

    getName = () => _name;

    set name(neuerName) {
        this._name = neuerName;
    }
}
```

ES6 Klassen

Private Attribute und Methoden zur Kapselung des Programms

- In ES6 wurde noch kein Keyword für eine Deklaration einer Variable als `private` eingeführt, jedoch ist dies in ES2019 (ES10) geschehen
- ES2019 wird von den meisten Plattformen unterstützt
- Das `# (Hashtag)` Kennzeichnet eine Variable oder Methode als `private`

```
class User {
    //private Variable
    #name;
    #alter;

    //private Methode
    #doSomething = () => {}

    constructor(name, alter){
        this.#name = name;
        this.#alter = alter;
    }
}
```

ES6 Klassen

Über statische Methoden

- Seit ES6 ist es möglich statische Methoden zu erstellen
- Statische Methoden werden in JavaScript direkt über die Klasse oder ihren Konstruktor aufgerufen und nicht über eine Instanz der Klasse
- Statische Methoden haben keinen Zugriff auf Daten spezifischer Objekte
- Statische Methoden sind mit *this* innerhalb einer nicht statischen Methode nicht direkt erreichbar

```
class TestMe{
      constructor(){
            console.log(this.constructor.staticMethod());
      }

      static staticMethod(){
            return 'Hello World';
      }
  }
```

ES6 Klassen

Über statische Methoden

- Bei der Vererbung einer Klasse mittels `extend` an eine weitere Klasse, werden statische Methoden mit vererbet
- Statische Methode bieten sich als `utility methods` an, welche häufig wiederverwendet werden und keine Instanz benötigen um aufgerufen zu werden (bekanntes Beispiel: `Math`)

```
class Calc{
    static modus = 'deg';

    static add(num_1, num_2){
        return num_1 + num_2;
    }

    static get zufall(){
        return Math.random();
    }
}
```

```
Calc.add(15, 9);  // 24

Calc.modus;       // deg
Calc.modus = 'rad';
Calc.modus;       // rad

class Calc_2 extends Calc{};
Calc_2.add(7, 4); // 11
```

Built-In Methoden → `Math`

Das `Math` Standardobjekt

- Wurde bereits vor ES6 eingeführt
- Bietet mathematische Methoden
  - Errechnen des Cosinus einer Zahl
  - Finden eines bestimmten Wertes in einer Sequenz (maximal und minimal Wert)
- Konstanten der Math-Objekts sind z.B.
  - Die Kreiszahl PI
  - Die Eulersche Zahl e
  - Die Quadratwurzel aus 2

Nachkommastellen entfernen

```
Math.trunc(-42.7); // -42
```

Vorzeichen einer Zahl bestimmen

```
Math.sign(7) // 1
```

Werte zur Potenz berechnen

```
Math.pow(2, 4); // 16
```

Kreiszahl Konstante PI

```
const kreisU = 2 * Math.PI * rad;
```

Built-In Methoden → `Number`

Das `Number` Wrapper-Objekt

- Wurde mit ES6 eingeführt
- Bietet Number-Methoden zum prüfen, ob ein Objekt vom Typ Number ist (Integer, Float, …)
- Konstanten des Number Objektes sind beispielsweise:
  - `MIN_VALUE` → Der kleinste repräsentierbare Zahlenwert
  - `MAX_VALUE` → Der größte repräsentierbare Zahlenwert
  - `NaN` → Steht für `Not a Number`

Variablen auf Typ `Number` prüfen:

```
Number.isNaN(" hallo ");// true
Number.isNaN("123");// false
```

Auf Endlichkeit prüfen:

```
Number.isFinite(Infinity);//false
Number.isFinite(123);// true
```

Die größte repräsentierbare Zahle:

```
if (x * y < Number.MAX_VALUE)
{...}
```

Built-In Methoden → `Math & Number`

Ein Vergleich von `float` Werten ist aufgrund von Rundungsfehlern fehleranfällig

- `Math.abs` → berechnet den Absolutwert einer Zahl
- `Number.EPSILON` → ein sehr kleiner positive Wert
- Berechnet sich aus: `1 - (kleinste Gleitpunktzahl > 1)`

```
Math.abs((0.1 + 0.2) - 0.3) < Number.EPSILON  // true
```

Die Differenz der beiden Argumente (`0.1 + 0.2`) und `0.3` wird berechnet. Die Funktion `Math.abs` bildet den Absolutwert der Subtraktion.

Das Ergebnis ist `true` wenn `Absolutwert` < `Number.EPSILON`. Die `Differenz` der vermeintlich gleichen Werte ist nahe `0` → die Werte sind gleich groß

Built-In Methoden → `Array`

Die globale Klasse `Array`

- Bietet aber seit ES6 neue Methoden
- Bekannte ES Array-Methoden:
  - `sort` → Sortieren der Elemente
  - `push/pop` → Hinzufügen oder Entfernen des obersten Elementes
  - `slice` → Zerteilen von Sequenzen
  - `concat` → Zusammenfügen von Sequenzen

Sortiert eine Abfolge von Daten durch String vergleiche in UTF-16:

```
const zahlen = [1, 5, 4, 41];
zahlen.sort(); // [1, 4, 41, 5]
```

Erstellt eine flache Kopie eines ausgewählten Teilabschnitts:

```
const zahlen =  ['A', 'B', 'C','D'];
zahlen.slice(1, 3); // ['B', 'C']
```

Built-In Methoden → `Array`

Die globale Klasse `Array`

- Neue ES6 Array-Methoden:
  - `find` → Gibt den Wert im Array an, der die Bedingung erfüllt:

```
const array1 = [5, 12, 8, 130, 4];
  array1.find(x => x > 12); // 130
```

- `findIndex` → Gibt den Index des Elements, welches die Bedingung erfüllt hat:

```
const array1 = [5,  12,  8,  130,  4];
      array1.findIndex(x => x > 12); // 3
```

Built-In Methoden → `String`

Das globale `String` Objekt

- Bekannte String-Methoden:
  - Gibt den Index-Anfang eines Teilabschnitts zurück:

```
'Test ABC Me'.indexOf('ABC'); //5
```

  - Gibt den Buchstaben an einer bestimmten Position zurück:

```
'Test ABC Me'.charAt(3); //t
```

  - Legt eine kleingeschriebene Kopie des Strings an:

```
'Test ABC Me'.toLowerCase(); //test abc me
```

  - Entfernt Leerzeichen und Zeilenumbrüche:

```
'   Hallo   '.trim(); //Hallo
```

  - Bildet einen Teilstring:

```
'Hallo Welt'.substr(1, 6); // allo W
```

Built-In Methoden → `String`

Das globale `String` Objekt

- Neue ES6 `String`-Methoden:
  - `repeat(x)` → erzeugt eine sich wiederholende Kopie des Strings

```
'abc'.repeat(2); //abcabc
```

  - `startsWith()/endswith()` → Prüft den Anfang oder das Ende einer Zeichenkette

```
'Hallo, wie gehts? '.startsWith('Hal'); //true
'Hallo, wie gehts? '.endsWith('gut'); //false
```

  - `includes()` → Prüft, ob eine Teilabschnitt im String vorhanden ist

```
'Hallo, wie gehts? '.includes("geht"); // true
```

Built-In Methoden → `String`

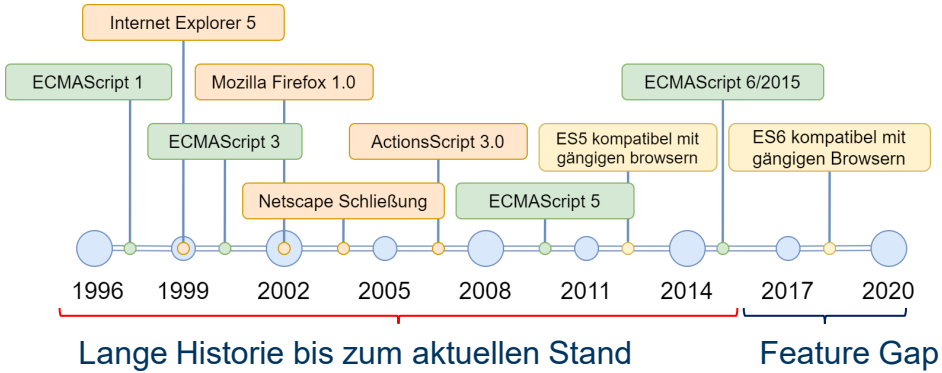
- Mit ES6 wurden Template literals eingeführt.
  - Ermöglichen Ausdrücke innerhalb von Strings
  - Sie werden mit `backticks` und nicht mit Anführungszeichen umschlossen → (` `)

```
var a = 21;
console.log(`Die Antwort ist, ${a*2}`);
// Die Antwort ist, 42
```

ES5 Kompatibilität

ES6 ist Abwärtskompatibel

- Heißt, Web-Anwendungen die in ES5 geschrieben wurden sind noch immer anwendbar
- Viele Nutzer verwenden weiterhin veralterte Systeme oder Web-Anwendungen
  - Die Wartung erfordert eine auseinandersetzen mit alten Notationen
- Feature Gap → Verzögerung der Kompatibilität von Programmversionen



Web-Engineering
TypeScript

TypeScript

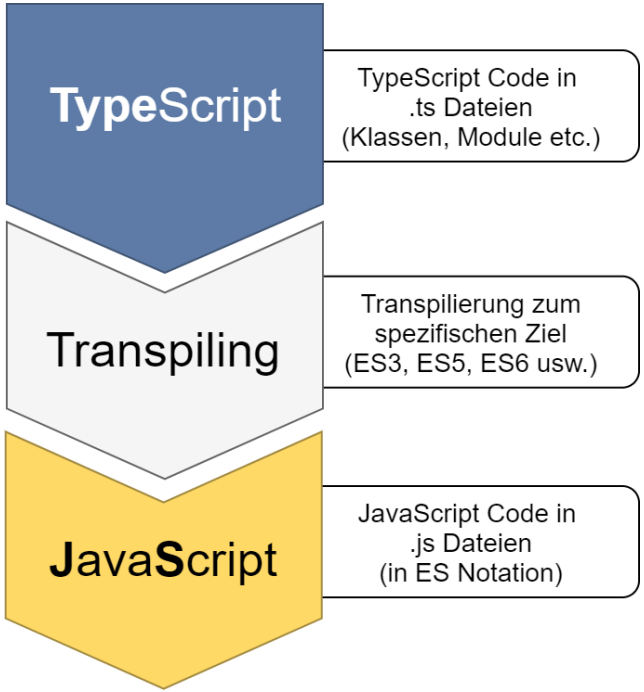
Übersicht Typescript

- Entwickelt von Microsoft
- Superset von Javascript mit optionaler Typisierung und Modulen
- Trans-Compiling zu Javascript, alle Javascript Anwendungen sind auch in Typescript gültig und es können JS Bibliotheken genutzt werden
- Lesbarer und gut strukturierter Code macht es einfacher Fehler früher zu erkennen, dazu trägt besonders die Typisierung bei, auch wenn sie nicht strikt ist.
- Unter anderem durch die Typisierung bieten viele IDEs weiterreichende Hilfen und Möglichkeiten für den Programmierer (im vergleich zu normalen JS IDEs)
- 2012 wurde Typescript in Version 0.8 veröffentlicht und seit dem ständig weiterentwickelt. Inzwischen ist Typescript bei der Version 4.2+

TypeScript

Motto von TypeScript: „JavaScript that scales“

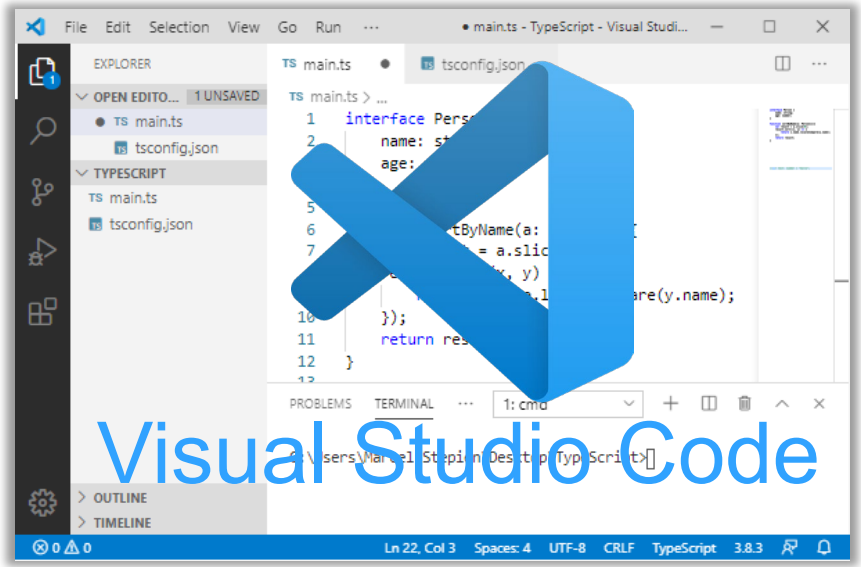
- JavaScript war nur als funktionale Programmiersprache gedacht
- Programmstrukturen wie Klassen, Module und Interfaces wurden ursprünglich nicht betrachtet
- TypeScript ist ein in JavaScript geschriebenes Open-Source Superset von JavaScript (Alle Javascript Elemente sind enthalten in TypeScript)
  - TypeScript ist typisiert
  - Wird von gängigen Tools stark unterstützt und bietet daher codetechnische Funktionen (intelligent code-completion, snippets, Typechecking usw.



TypeScript

Voraussetzungen für TypeScript

- TypeScript wird über Node.js eingebunden (Node Package Manager, npm)
- Nach der Installation kann die Prozesskette vom schreiben des TS Codes bis zu der Transpilierung in JavaScript Code genutzt werden
- Zum Entwickeln von TypeScript bietet sich Visual Studio Code an.
  - VS Code ist ein Open Source Editor, in TypeScript/JavaScript geschrieben
  - Es können auch eine ganze Reihe andere Editoren und IDE`s genutzt werden (Sublime, Vim, Eclipse, Atom)



Quelle: https://code.visualstudio.com/

TypeScript

Transpilierung Konfigurieren

- Die TypeScript Konfigurations-Datei (`tsconfig.json`) wird über die Kommandozeile in einem ausgewählten Ordner erzeugt

Kommandozeile

```
C:\Users\...> tsc --init
```

- Bei der Transpilierung von TS wird die Konfigurationsdatei eingebunden
- Neben zahlreichen Optionen, ermöglicht die Konfiguration es die Zielversion von ECMAScript einzustellen

Inhalte von `tsconfig.json`:

```
{
  "compilerOptions": {
  // "incremental": true,
     "target": "es6",
     "module": "commonjs",
     ...
}
```

Mögliche Zielversionen:

```
target:   'ES3' | 'ES5'(default) | 'ES2015' | 'ES2016' | 'ES2017' |
          'ES2018'| 'ES2019' | 'ES2020' | 'ESNEXT',
```

TypeScript

TypeScript ermöglicht die Angabe von typisierten Variablen!

- TypeScript verfügt über eine Reihe von vordefinierten primitiven Typen

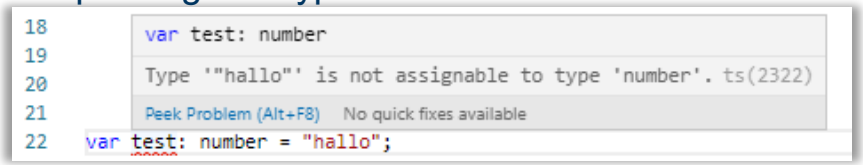
```
var valueA: number          = 31;
var valueB: string          = 'Max Mustermann';
var listA:  Array<number>   = [1, 2, 3];
var listB:  any[]           = [false, 5, 'hallo'];
```

Der Typ der Variable wird mit Doppelpunkte angegeben. Auch Listen können so angegeben werden. Der Typ `any` ist hierbei ein generischer Supertyp aller Typen.

- TypeScript überprüft Typen und signalisiert Fehler
- Mit der Listenschreibweise können komplexe Datensätze recht einfach definiert werden
- Beispiel für Listenschreibweise:

```
var value : [number, string, any];
value = [12, 'Jasmin', true];
```

Überprüfung der Typen:



TypeScript

Die Definition von eigenen Typen wird über `interfaces` ermöglicht

- Das Interface beschreibt die Inhaltliche Struktur, den ein Wert eines Types haben muss
- Dieses Vorgehen wird oft als `duck typing` oder `structural subtyping` bezeichnet
- Optionale Parameter werden mit Fragezeichen gekennzeichnet (bsp. `value?`)
- Nachträgliche Änderungen werden mit dem Schlüsselwort `readonly` unterbunden

```
interface Car {
  name:  string;
  year? :  number;
  readonly inspected :  boolean;
}
function sell(obj: Car) {
  if (!obj.inspected) {
    console.log("Werkstatt aufsuchen.");
  } else {
    console.log(obj.name  +
       ", Baujahr: " + obj.year  +
       ", im Top Zustand!");
  }
}
//Funktionsaufruf
sell({name: "Ford Fiesta", year: 2002,
inspected: false});
sell({name: "Ford KA", inspected: true });
```

TypeScript

- Mit TypeScript können auch semantische Enumerationen definiert werden
- Der Typ der Enumerationswerte kann dabei explizit gesetzt werden
- Wird in nativen JavaScript durch Objekte als Platzhalter gelöst (als Dictionary)

Numerisch aufsteigende Aufzählung von Werten:

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}
//Up = 1, Down = 2,
//Left = 3, Right = 4
```

Numerisches Aufzählung von Werten:

```
enum Direction {
    Up = 5,
    Down = 12,
    Left = 20,
    Right = 14
}
```

String Aufzählung von Werten:

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT"
}
```

TypeScript

Transcompiler Code Gegenüberstellung → ES5

```ts
interface Person {
  name: string;
  age: number;
}
class Firma {
  personal: Person[];
  constructor(p: Person[]) {
    this.personal = p;
  }
  sortByName() {
    var result =
         this.personal.slice(0);
    result.sort((x, y) => {
      return x.name.localeCompare(y.name);
    });
    return result;
  }
}
```

Typisierung nur im TS Code

Generierter JavaScript ES5 Code:

```js
var Firma = /** @class */ (function () {
  function Firma(p) {
    this.personal = p;
  }
Firma.prototype.sortByName = function () {
    var result = this.personal.slice(0);
    result.sort(function (x, y) {
      return x.name.localeCompare(y.name);
    });
    return result;
  };
  return Firma;
}());
```

TypeScript

Transcompiler Code Gegenüberstellung → ES5

```
interface Person {
  name: string;
  age: number;
}
class Firma {
  personal: Person[];
  constructor(p: Person[]) {
    this.personal = p;
  }
  sortByName() {
    var result =
        this.personal.slice(0);
    result.sort((x, y) => {
      return x.name.localeCompare(y.name);
    });
    return result;
  }
}
```

Klassen existieren nicht im ES5

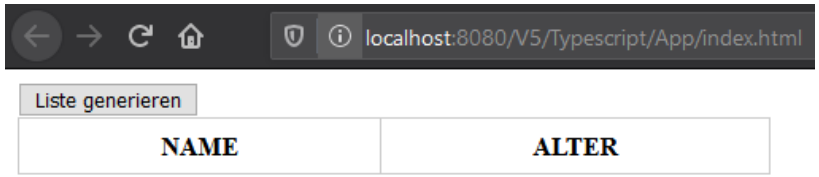
Arrow Functions werden durch `function` ersetzt

Generierter JavaScript ES5 Code:

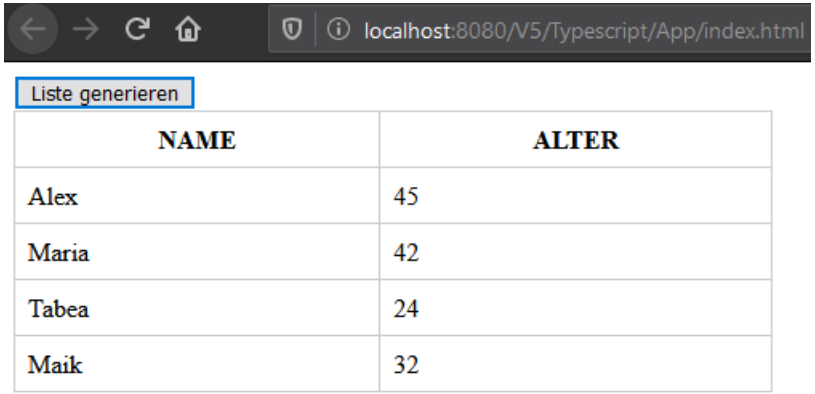
```
var Firma = /** @class */ (function () {
  function Firma(p) {
    this.personal = p;
  }
Firma.prototype.sortByName = function () {
    var result = this.personal.slice(0);
    result.sort(function (x, y) {
      return x.name.localeCompare(y.name);
    });
    return result;
  };
  return Firma;
}());
```

TypeScript

Darstellung im Browser



Der Tabellenkopf und die erste Zeile der Tabelle sind bereits erstellt.



Weitere Zeilen werden dynamisch, auf Knopfdruck, nach folgendem Muster ergänzt.

<tr> <td> Name </td> <td> Alter </td> </tr>
x4

TypeScript → Beispiel

```
var alex: Person = { name: 'Alex', age: 45 };
var maria: Person = { name: 'Maria', age: 42 };
…
var angestellte: Person[] = [alex, maria, …];
var beispiel = new Firma(angestellte);
```

Instanziierung der Personenobjekte mit `Typisierten Variablen`

Instanziierung des Angestellten Arrays vom `Typ Person`

```
function loadTable(firma: Firma) {
    const tableBody = document.getElementById('tableData');
    var dataHTML = '';
    for (let [key, value] of Object.entries(firma.personal)) {
        dataHTML += `<tr><td>${value.name}</td>`;
        dataHTML += `<td>${value.age}</td></tr>`;
    }
    var loadBtn = document.getElementById("button");
    if(loadBtn != null && tableBody != null){
        loadBtn.addEventListener('click', () => {
            tableBody.innerHTML = dataHTML; });
    }
}
```

Innerhalb der Funktion wird der Inhalt einer Tabelle dynamisch erzeugt.

Dem `button` wird seine Funktionalität beim Knopfdruck zugeordnet. Die null Prüfung muss stattfinden in TS! (Lösung für The Billion Dollar Mistake)!