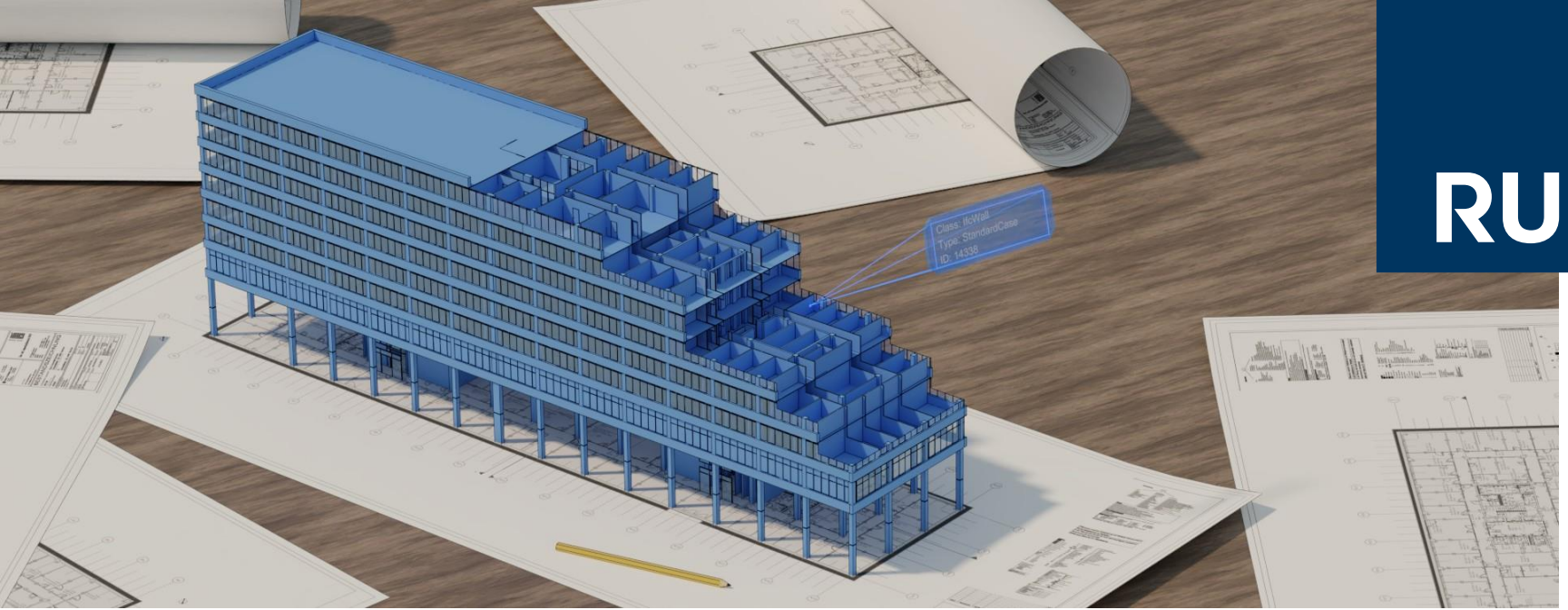


RUB

RUHR-UNIVERSITÄT BOCHUM

WEB-ENGINEERING

Informatik
im Bauwesen

Sommersemester 2023

Web-Engineering
JavaScript
Weiterführende Themen

JavaScript Weiterführende Themen

Variablendeklaration mit let und var

- Es macht einen Unterschied, ob eine Variable mittels var oder let definiert wurde
  - Geltungsbereich (Scope)

    `var`: Scope ist durch die umgebende Funktion definiert
    `let`: Variable ist nur innerhalb des nächstgelegenen Blocks sichtbar
  - Wiederverwendung

    `var`: Variable kann erneut deklariert werden
    `let`: Variable kann nicht erneut deklariert werden

JavaScript Weiterführende Themen

Variablendeklaration mit let und var

```
11  function demoLet(){
12
13      //index ist hier nicht sichtbar
14
15      for (let index = 0; index < 5; index++) {
16          //index ist hier sichtbar
17
18      }
19
20      //index ist hier nicht sichtbar
21
22  }
```

```
24  function demoVar(){
25
26      //index ist hier sichtbar
27
28      for (var index = 0; index < 5; index++) {
29          //index ist hier sichtbar
30      }
31
32      //index ist hier sichtbar
33
34  }
```

JavaScript Weiterführende Themen

Variablendeklaration mit let und var

Beispiel: `let`

```
function bsp_0 (){
  let x = 42;
  if(true){
    console.log(x); // 42
  }
};
```

```
function bsp_1(){
  let x = 42;
  if(true){
    console.log(x); // ReferenceError
    let x = 5; // x wird neu deklariert
  }
};
```

Beispiel: `var`

```
function bsp_3 (){
  var x = 42;
  if(true){
    console.log(x); // 42
  }
};
```

```
function bsp_4(){
  var x = 42;
  if(true){
    console.log(x); // 42
    var x = 5; // x wird nicht
  }            //   neu deklariert
};
```

JavaScript Weiterführende Themen

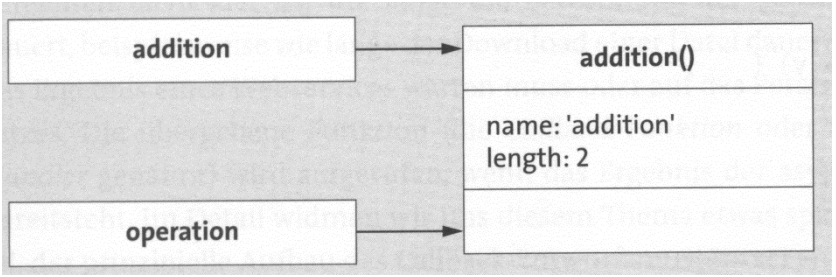
Besonderheiten von Funktionen

- Funktionen werden in JavaScript durch Objekte repräsentiert, somit können die Funktionen auch Variablen zugewiesen werden

```
function addition(x, y) {
        return x + y;
}

var operation = addition;

console.log(operation.name) // Ausgabe: addition
```



JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Funktionen können somit auch in Arrays verwendet werden

```
function addition(x, y) {
        return x + y;
}
function substraktion (x, y) {
        return x - y;
}
var operationen = [addition, substraktion];
for (var i=0; i<operationen.length; i++) {
        console.log(operationen[i](2,2));
}
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Funktionen können in JavaScript als Parameter einer anderen Funktion verwendet werden

```
function metaOperation(operation, x, y) {
        return operation(x,y);
}
console.log(metaOperation(addition, x ,y));
```

- Dementsprechend ist es auch möglich eine Funktion als Rückgabewert zu verwenden

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Wenn eine Funktion innerhalb eines Objekts definiert wird, wird von einer Objektmethode gesprochen

```
var operationen = {

        addition: function(x, y) { //Objektmethode: addition
                return x + y;
        }

        subtraktion(x, y) { //Objektmethode: subtraktion
                return x - y;
        }
}

console.log(operationen.addition(2,2));
console.log(operationen.subtraktion(2,2));
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Funktionen besitzen in JavaScript einen bestimmten Kontext (`this`), d.h. sie beziehen sich auf das Objekt, in dem die Funktion ausgeführt wird.

```
var person = {
        name : 'Max‘,
        getName : function() {
        this.name;
        }
}
console.log(person.getName());      // Ausgabe: Max
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Funktionen besitzen in JavaScript einen bestimmten Kontext (`this`), d.h. sie beziehen sich auf das Objekt, in dem die Funktion ausgeführt wird.

```
var name = 'Max';
function getNameGlobal() {
            return this.name;
}
console.log(getNameGlobal());       // Ausgabe: Max
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Funktionen besitzen in JavaScript einen bestimmten Kontext (`this`), d.h. sie beziehen sich auf das Objekt, in dem die Funktion ausgeführt wird.

```
var name = 'Max';
function getNameGlobal() {
            return this.name;
}
var person = {
            name : 'Moritz';
            getName : getNameGlobal
}
console.log(person.getName());     // Ausgabe: Moritz
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Im Gegensatz zu Java kennt JavaScript keinen Block-Scope für Variablen bzw. {} spannen keine Gültigkeitsbereich auf

```
function beispiel(x) {
        if (x) {
            var y = 2015;
        }
        for (var i=0; i<2015; i++) {}
        console.log(y);
        console.log(i);
}
beispiel(true);     // Ausgabe: 2015 2014
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Alle Variablendeklaration sind bereits zu Beginn einer Funktion bekannt. Dies wird durch den JavaScript-Interpreter übernommen (Hoisting). Zur besseren Übersicht sollten alle Variablen zu Beginn der Funktion deklariert werden

```
function beispiel(x) {
        var y, i; // Macht der Interpreter automatisch
        if (x) {
            y = 2015;
        }
        for (i=0; i<2015; i++) {}
}
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- In Javascript können durch Hoisting Variablen benutzt werden, bevor sie deklariert wurden.
- `let` und `const` können jedoch nicht benutzt werden, bevor sie initialisiert wurden.
- Für Initialisierungen gibt es kein Hoisting
- Generell sollten Variablen immer oben, vor der Nutzung deklariert werden

Funktioniert:

```
x = 5;
var x;
```

Fehler:

```
x = „BMW“;
let x;
```

```
var x = 5;
console.log(x + " " + y);
var y = 7;
```

→ 5 undefined

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

使用相同的名称，但是不同的参数，定义不同的函数

- In JavaScript ist das Überladen von Funktionen **nicht möglich**, da die Funktionen nur anhand des Namens unterschieden werden
  - In TypeScript können jedoch Funktionen Überladen werden
- Somit überschreibt eine später definierte Funktion immer die zuvor definierte Funktion mit dem gleichen Namen
- Um in JavaScript überladene Funktionen nachzubilden, muss die Funktionalität auf verschiedene Funktionen verteilt werden. Es kann so eine Funktion erstellt werden, die anhand eines Attributs intern die richtige Funktion aufruft. Somit würde eine Funktion mit unterschiedlicher Funktionalität entstehen

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Funktionen werden auch überschrieben, wenn im gleichen Kontext eine gleich benannte Variable deklariert wird

```
function beispiel(x) {}

// Überschreibt die erste Funktion
function beispiel(x, y) {}

// Überschreibt die zweite Funktion
var beispiel = true;
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Es könnte eine Funktion erstellt werden, die anhand eines Attributs intern die richtige Funktion aufruft

```
function personenSuchen(a) {
  if (typeof a === 'string') {
    Namenssuche(a);
  }
  if (typeof a === 'number') {
    Geburtsdatumssuche(a);
  }
};
personenSuchen(5); //Geburtsdatumssuche
personenSuchen('Herbert');//Namenssuche
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Arrow functions oder fat arrow functions wurden mit ES6 eingeführt
- Das Schlüsselzeichen für diese Funktionen ist das => Zeichen
- arrow functions sind eine bestimmte Art von anonymen Funktionen mit ihrer eigenen Syntax

```
(argument1, argument2, ... argumentN) => {
        // Funktionsrumpf
}
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Varianten der arrow function:

```
(param1, pram2, param3) => { Anweisung_1, Anweisung_2 ... }
```

  - Bei nur einer Anweisung sind die geschweiften Klammern des Anweisungsblockes nicht notwendig

```
(param1, pram2, param3) =>  Anweisung //"{}" sind optional
```

  - Bei nur einem Parameter sind die runden Klammern nicht notwendig.

```
einParam => { Anweisungen } //"()" sind optional
```

  - Bei keinem Parameter müssen nur die runden Klammern angegeben werden.

```
() => { Anweisungen } //eine Funktion ohne Parameter
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- In Javascript gilt LexicalScope
  - Innere Funktionen haben Zugriff auf Variablen der äußeren Funktionen
  - Ob eine Variable verfügbar ist hängt davon ab, wo sie deklariert wird

```
function init() {
    var name = ‘RUB’;
    function displayName() {
            alert(name);}
    displayName();}
init();
```

JavaScript Weiterführende Themen

Besonderheiten von Funktionen

- Ein wichtiger Punkt im Bezug auf arrow functions ist der Scope innerhalb dieser Funktionen
  - Ein `this` in einer Arrow `function` bezieht sich nicht auf den Scope der Funktion sondern auf den des Eltern-Objekts
  - Arrow `functions` haben somit keinen eigenen Scope
  - Der Lexical Scope macht arrow functions zu einer guten bzw. schlechten Wahl in bestimmten Szenarien

JavaScript Weiterführende Themen

Funktionale Konzepte

- Bei der funktionalen Programmierung liegt der Fokus auf Funktionen.
- Ein typisches Beispiel ist die `forEach()` Methode für Arrays. Als Parameter wird eine Funktion, die jedes Element im Array mit drei Parametern aufgerufen wird: Element, Index und das Array selbst.

```
var namen = ['Max', 'Moritz', 'Werner'];
namen.forEach(function(name, index, namen) {
        console.log(name);
});
```

JavaScript Weiterführende Themen

Funktionale Konzepte

- Bei der funktionalen Programmierung liegt der Fokus auf Funktionen.
- Ein typisches Beispiel ist die `forEach()` Methode für Arrays.

```
var namen = ['Max', 'Moritz', 'Werner'];
namen.forEach(function(name, index, namen) {
        console.log(name);
});
```

- Sobald Funktionen wie Daten weitergegeben werden, wie hier als Parameter, handelt es sich um Lambda Funktionen

JavaScript Weiterführende Themen

Funktionale Konzepte

- Eine weitere, häufig anzutreffende Problemstellung ist es, aus einem bestehenden Array eine gewisse Anzahl an Elementen anhand bestimmter Kriterien herauszufiltern.
- Hierzu kann die `filter()` Methode für Arrays verwendet werden, welche prinzipiell wie die `forEach()` Methode aufgebaut ist.

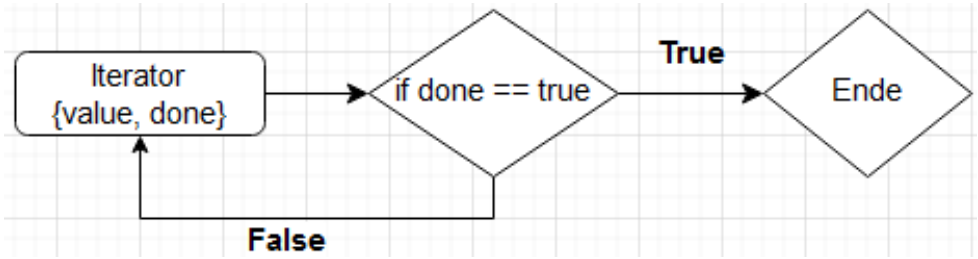
```
var alben = [
    {
        titel: "Push the Sky Away",
        interpret: "Nick Cave",
        erscheinungsjahr: 2013
    },
    {
        titel: "No more shall we part",
        interpret: "Nick Cave",
        erscheinungsjahr: 2001
    },
    {
        titel: "Live from Mars",
        interpret: "Ben Harper",
        erscheinungsjahr: 2003
    },
    {
        titel: "The Will to Live",
        interpret: "Ben Harper",
        erscheinungsjahr: 1997
    }
];
```

```
var vor2000 = alben.filter(function(album, index, alben) {
    return album.erscheinungsjahr < 2000;
});
```

JavaScript Weiterführende Themen

Iteratoren

- Iteratoren (iterator) werden genutzt um Sequenzen von Daten effizienter zu durchlaufen
- In JavaScript ist ein iterable ein Objekt, welches über einen iterator verfügt
- Der iterator stellt eine next() Methode zur Verfügung
- Die Methode next() hat zwei Rückgabewerte:
  - value → Wert zum Key (momentanes Element)
  - done → ein boolscher Wert



JavaScript Weiterführende Themen

Symbol

- Symbol ist ein primitiver Datentyp, die Symbol()-Funktion gibt Werte vom Typ Symbol zurück
  - Jeder Symbol Wert von Symbol() ist einzigartig und eignet sich daher auch als Identifier für Objekteigenschaften
- Symbol.iterator gibt den Standarditerator für ein Objekt an
  - Soll eine for...of Schleife genutzt werden, wird die Iterator-Methode ohne Argumente aufgerufen und die zurückgegebenen Iteratoren werden zum finden der zu iterierenden Werte genutzt

JavaScript Weiterführende Themen

Iteratoren

- Integrierte Typen wie `array` oder `map` weisen ein Standarditerationsverhalten auf
  - Andere Typen, wie beispielsweise Objekte, nicht
- Um iterierbar zu sein, muss ein Objekt die Iterator-Methode implementieren, welche über die Konstante *Symbol.iterator* erreichbar ist
  - Wenn ein Objekt iteriert werden soll, wird die Iterator-Methode aufgerufen
  - Der zurückgegebene Iterator wird benutzt, um die zu iterierenden Werte zu erhalten
- Sobald ein Datentyp einen Iterator enthält kann die for…of-Schleife darauf angewendet werden

JavaScript Weiterführende Themen

Iteratoren

- Eigens erstellte Objekte zu iterables machen
- In diesem Bsp. werden mittles der Methode `makeStud()` eine zufällige Anzahl (bis max. 8) Studenten erzeugt und in einem dementsprechend großen Array abgelegt.

```
const jahrgang = {
   [Symbol.iterator]: () => {
      return{
         next: () =>{
            const jahrgangGroesse = Math.random()>0.80
            if (!jahrgangGroesse){
               return{
                  value: makeStud(),
                  done: false
               }
            }
            return{ done: true }
         }
      }
   }
}

// Nun kann die for...of-Schleife angewendet werden
for (const students of jahrgang){
    console.log(students);
}
```

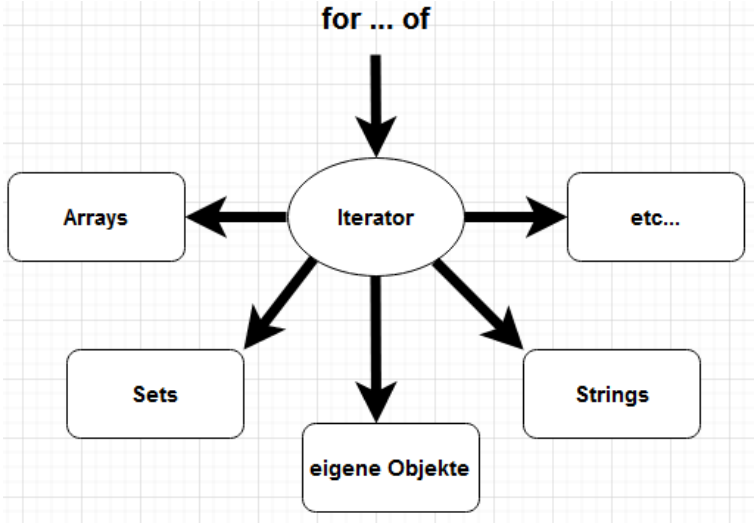
Hier werden die Studenten Objekte erzeugt. Ein Wert und der angepasste Status

Wenn die korrekte Anz. Erzeugt wurde, wird `done` auf `true` gesetzt

JavaScript Weiterführende Themen

Iteratoren

- Sobald ein Datentyp ein `iterator` enthält kann die `for...of`-Schleife darauf angewendet werden.



JavaScript Weiterführende Themen

Callback-Entwurfsmuster

- Beim Callback-Entwurfsmuster wird einer Funktion eine andere Funktion als Parameter übergeben. Die übergebene Funktion wird im Laufe der Ausführung der aufgerufenen Funktion von dieser aufgerufen.

```
1  function summePerCallback(x, y, callback) {
2      var ergebnis = x + y;
3      if(typeof callback === "function") {
4          callback(ergebnis);
5      }
6  }
7  summePerCallback(2, 2, function(ergebnis) {
8      console.log("Das Ergebnis lautet: " + ergebnis);
9  });
```

JavaScript Weiterführende Themen

Callback-Entwurfsmuster

- Funktionsdefinition
- Funktionsaufrufe
- Ausgabe: 1 2 3 8 16

```
const ausgabe = (akkumulator, callback) => {
    console.log(akkumulator);
    callback(akkumulator);
};
const duplicate = (akkumulator) =>{
    ausgabe(akkumulator, (akkumulator) =>{
        akkumulator += akkumulator;
        ausgabe(akkumulator, (akkumulator) =>{
            akkumulator += akkumulator;
            ausgabe(akkumulator, () =>{
                akkumulator += akkumulator;
                ausgabe(akkumulator, () => {
                    akkumulator += akkumulator;
                    ausgabe(akkumulator, () => {});
                });
            });
        });
    });
};
duplicate(1);
```

Web-Engineering
JavaScript
Prototypen

JavaScript Weiterführende Themen

Prototypen in ES5

- Die Objektorientierung in JavaScript 5 (ES5) basiert nicht auf Klassen sondern sogenannten Prototypen. Objekte in JavaScript sind dynamische, veränderbare Datenstrukturen aus Schlüssel-Wert-Paaren. Prinzipiell existieren in JavaScript 5 (ES5) drei Arten von Objekten:
  - Native Objekte: Dies sind vordefinierte Objekte, die durch die Sprache zur Verfügung gestellt werden: `String, Number, Array, Math` etc.
  - Host-Objekte: Diese Objekte werden von der Laufzeitumgebung bereitgestellt. Innerhalb eines Browsers sind dies unteranderem `window` und `document`.
  - Benutzer-Objekte: Diese teilweise recht komplexen Objekte werden durch den Entwickler erstellt.

JavaScript Weiterführende Themen

Objekte erstellen

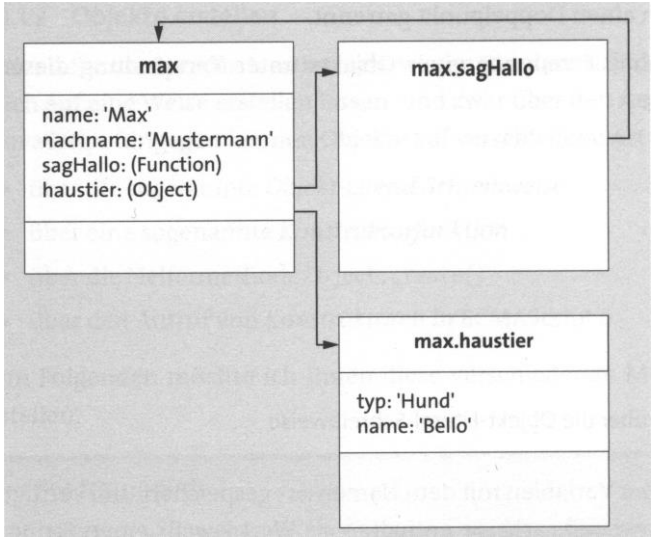
- In JavaScript 5 (ES5) werden Objekte nicht ausschließlich über Konstruktoren erstellt. Es können folgende Erzeugungsarten verwendet werden
  - Objekt-Literal-Schreibweise
  - Konstruktorfunktionen
  - Helfermethode `create()`
  - Konstruktoren (in ECMAScript 6)

JavaScript Weiterführende Themen

Objekte erstellen

- Jedes Objekt, welches über die Literal-Schreibweise erzeugt wird, ist implizit ein Singleton

```
1  var max = {
2      name : 'Max',
3      nachname : 'Mustermann',
4      sagHallo : function() {
5          console.log('Hallo');
6      },
7      haustier : {
8          name : 'Bello',
9          typ: 'Hund'
10     }
11 };
```



JavaScript Weiterführende Themen

Singletons

- Jedes Objekt, welches über die Literal-Schreibweise erzeugt wird, ist implizit ein Singleton
- Es wird sichergestellt, dass es nur eine Objektinstanz des Objekts geben kann

JavaScript Weiterführende Themen

Objekte erstellen

- Jedes Objekt, welches über die Literal-Schreibweise erzeugt wird, ist implizit ein Singleton

```
1  var max = {
2      name : 'Max',
3      nachname : 'Mustermann',
4      sagHallo : function() {
5          console.log('Hallo');
6      },
7      haustier : {
8          name : 'Bello',
9          typ: 'Hund'
10     }
11 };
```

```
1  var bello = {
2      name : 'Bello',
3      typ: 'Hund'
4  };
5  var max = {
6      name : 'Max',
7      nachname : 'Mustermann',
8      sagHallo : function() {
9          console.log('Hallo');
10     },
11     haustier : bello
12 };
```

JavaScript Weiterführende Themen

Objekte erstellen

- Funktionen können nicht nur als „normale“ Funktionen, sondern auch als Konstruktorfunktion aufgerufen werden. Bei der Deklaration ändert sich dabei nichts. Es hat sich jedoch als Konvention etabliert, dass der Funktionsname in sogenannter Upper-Camel-Case-Schreibweise definiert wird

```
1  function Album(titel) {
2      this.titel = titel;
3  }
4  var album = new Album('Sky Valley');
5  console.log(album.titel); // Sky Valley
```

JavaScript Weiterführende Themen

Objekte erstellen

- Jedes Objekt kann in JavaScript als Prototyp (Vorlage) für ein anderes Objekt verwendet werden. Das Prototyp-Objekt stellt dem neuen Objekt seine Eigenschaften und Methoden zur Verfügung
- Der Prototyp eines Objektes ist in der Eigenschaft `__proto__` hinterlegt. Mit Hilfe der Methode `Object.isPrototypeOf()` kann überprüft werden, ob ein Objekt der Prototyp eines anderen Objektes ist

```
1  var max = {
2      name: 'Max',
3      nachname: 'Mustermann'
4  };
5  console.log(max.__proto__); // Object {}
6  console.log(Object.getPrototypeOf(max)); // Object {}
```

JavaScript Weiterführende Themen

Objekte erstellen

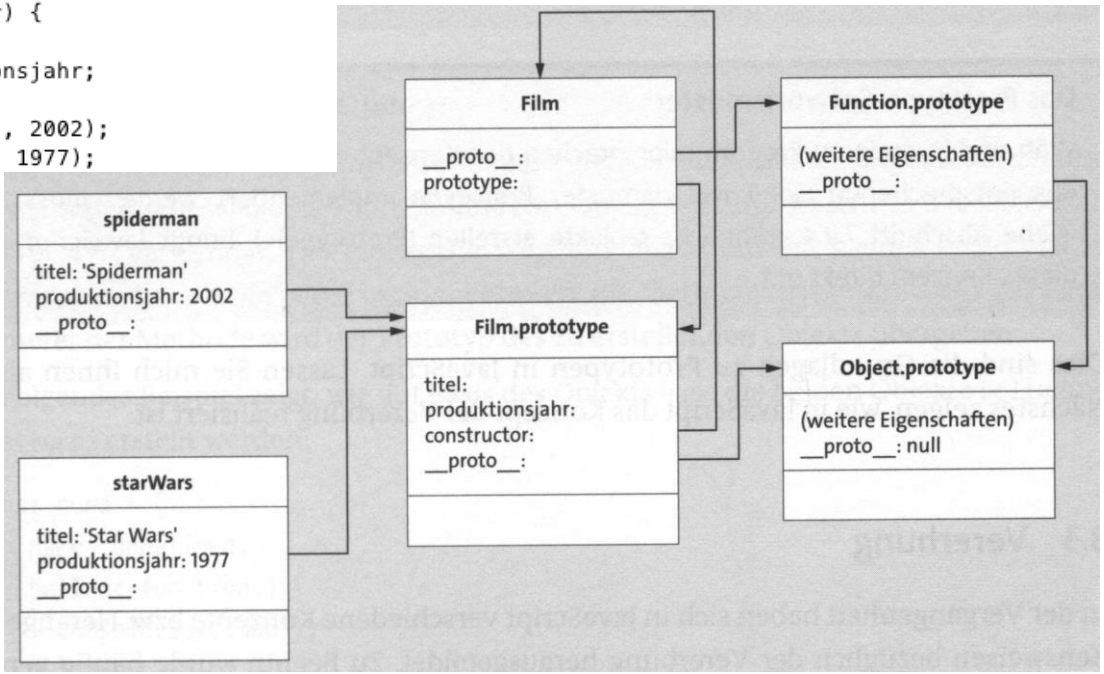
- Bei Objekten, die mit einer Konstruktorfunktion erzeugt wurden, zeigt die Eigenschaft `__proto__` auf den Prototypen, der von der Konstruktorfunktion intern dazu verwendet wird
- Auch Konstruktorfunktionen haben eine Eigenschaft `__proto__`, die auf den Prototypen der Funktion verweist

```
1  function Film(titel, produktionsjahr) {
2      this.titel = titel;
3      this.produktionsjahr = produktionsjahr;
4  };
5  var spiderman = new Film('Spiderman', 2002);
6  var starWars = new Film('Star Wars', 1977);
7  console.log(spiderman.__proto__); // Film {}
8  console.log(starWars.__proto__); // Film {}
9  console.log(Object.getPrototypeOf(spiderman)); // Film {}
10 console.log(Object.getPrototypeOf(starWars)); // Film {}
11 console.log(spiderman.constructor); // function Film() {...}
12 console.log(starWars.constructor); // function Film() {...}
```

JavaScript Weiterführende Themen

Objekte erstellen

```
1  function Film(titel, produktionsjahr) {
2      this.titel = titel;
3      this.produktionsjahr = produktionsjahr;
4  };
5  var spiderman = new Film('Spiderman', 2002);
6  var starWars = new Film('Star Wars', 1977);
```



JavaScript Weiterführende Themen

Objekte erstellen

- Jedes Objekt kann in JavaScript als Prototyp (Vorlage) für ein anderes Objekt verwendet werden. Das Prototyp-Objekt stellt dem neuen Objekt seine Eigenschaften und Methoden zur Verfügung
- Der Prototyp eines Objektes ist in der Eigenschaft `__proto__` hinterlegt. Mit Hilfe der Methode `Object.isPrototypeOf()` kann überprüft werden, ob ein Objekt der Prototyp eines anderen Objektes ist

```
1  var max = {
2      name: 'Max',
3      nachname: 'Mustermann'
4  };
5  console.log(max.__proto__); // Object {}
6  console.log(Object.getPrototypeOf(max)); // Object {}
```

JavaScript Weiterführende Themen

Objekte erstellen

- Es kann auch der Prototyp von `Object` verwendet werden, dem während der Erzeugung durch *Property-Deskriptoren* oder anschließend Eigenschaften zugewiesen werden kann

```
1  var max = Object.create(Object.prototype, {
2      name: {
3          value: 'Max',
4          writable: false,
5          configurable: true,
6          enumerable: true
7      },
8      nachname: {
9          value: 'Mustermann',
10         writable: true,
11         configurable: true,
12         enumerable: true
13     }
14 });
15 console.dir(max);
```

JavaScript Weiterführende Themen

Objekte erstellen

- Der Name des Property-Deskriptors ist gleichzeitig der Name der Eigenschaft, die definiert werden soll. Für das Property-Objekt existieren verschiedene Attribute:
  - `writable:` wenn `false`, kann der Wert der betroffene Objekt-Eigenschaft nicht mehr verändert werden
  - `enumerable:` wenn `false`, taucht die betroffene Objekt-Eigenschaft nicht mehr in den Auflistungen aller Eigenschaften des Objekts (wie z.B. bei for-in-Schleifen) auf
  - `configurable:` wenn `false`, kann die Eigenschaft nicht gelöscht werden und auch seine writeable- enumerable- und configurable-Werte können nicht mehr verändert werden

JavaScript Weiterführende Themen

Objekte erstellen

- Seit ECMAScript6 (Juni 2015) lassen sich auch Klassen mit Hilfe des Schlüsselwort `class` definieren. Auch die Definition von Funktionen wurde deutlich vereinfacht.

```
1  /* Funktioniert nur in ES6 */
2  class Person {
3      constructor(vorname, nachname) {
4          this.vorname = vorname;
5          this.nachname = nachname;
6      }
7      toString() {
8          return this.vorname + ' ' + this.nachname;
9      }
10 }
```