Webtechnologien | 2015

Kapitel 3: JavaScript

JavaScript-Überblick | JavaScript-Grundlagen | Funktionen und Funktionale Aspekte | Objektorientierung | JavaScript im Browser | ...

Axel Küpper | Fachgebiet *Service-centric Networking* | TU Berlin & Telekom Innovation Laboratories

3.1 JS-Überblick
Geschichte und Merkmale


(inoffizielles Logo)

Geschichte

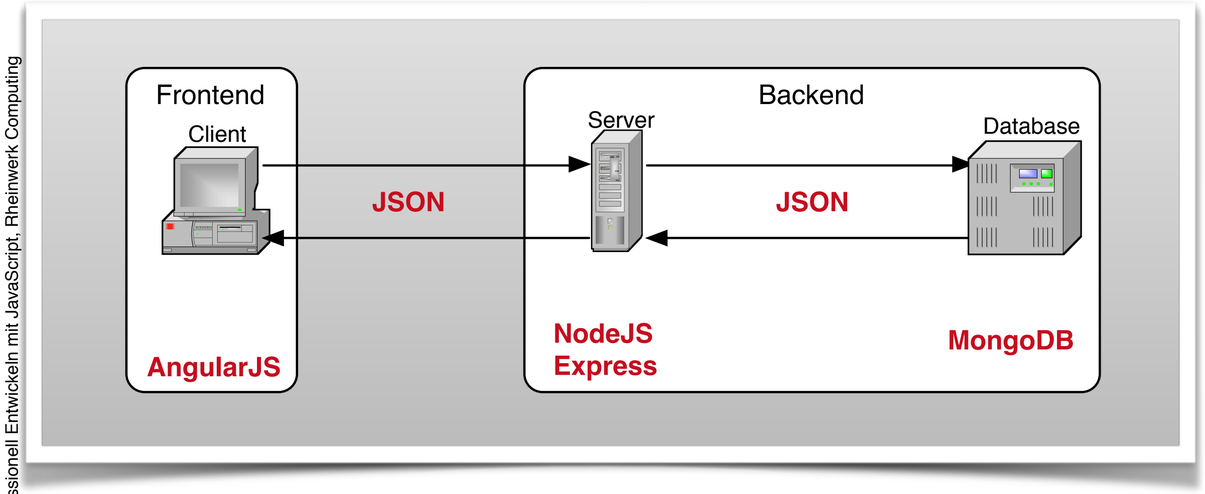
- Entwicklung der ersten Version unter dem Namen *LiveScript* 1995 in 12 Tagen von Brendan Eich für den *Netscape Navigator*
- Namensänderung zu *JavaScript* 1996, basierend auf einer Kooperation zwischen dem Java-Entwickler *Sun* und Netscape
- Zeitgleicher Entwicklung einer ähnlichen Sprache unter dem Namen *JScript* durch Microsoft für den Internet Explorer 3.0
- *ECMAScript*: Standardisierung von JavaScript durch die *European Computer Manufacturer Association* (*ECMA*) mit der Zielsetzung eines einheitlichen Standards
- Verabschiedung von ECMAScript Version 6 in 2015
- JavaScript ist lediglich eine Implementierung von ECMAScript
- Andere Implementierungen von ECMAScript: *QtScript*, *ActionScript* (Flash) und *ExtendScript* (Verwendung in anderen Adobe-Produkten)

Merkmale

- Java und JavaScript haben bis auf einen ähnlichen Namen und eine ähnliche Syntax nicht viel gemeinsam
- Wesentliche Konzepte von JavaScript sind *funktionale Programmierung* und *prototypische Objektorientierung*

3.1 JS-Überblick

Anwendungen: Clientseitiges versus Serverseitiges JS



mit Änderungen übernommen von: Ackermann, P. (2015) Professionell Entwickeln mit JavaScript, Rheinwerk Computing

Clientseitige JavaScript-Webanwendungen

- Verwendung war lange Zeit auf das User Interface einer Webseite beschränkt
- *DHTML* (*Dynamisches HTML*): Manipulation des *DOM* (*Document Object Model*) durch JavaScript
- Spätere Erweiterungen zur asynchronen Kommunikation zwischen Browser und Webserver durch *AJAX* (*Asynchronous JavaScript and XML*)
- Stand heute: Realisierung von *Rich Internet Applications* (*RIA*) und *Single Page Applications* (*SPA*)
- Idee von SPA: Website besteht nicht mehr aus mehreren, sondern einer einzelnen Webseite, die je nach Nutzerinteraktion dynamisch aktualisiert wird
- Austausch von Inhalten erfolgt über die *JavaScript Object Notation* (*JSON*) die "nahtlos" durch JavaScript verarbeitet werden kann
- Serverseitige Speicherung von Inhalten im JSON-Format durch *MongoDB*
- Grundlage für weitere Frameworks wie *AngularJS*

Serverseitige JavaScript-Anwendungen

- *Node.js*: Plattform die es ermöglicht serverseitige Aufgaben mit JavaScript umzusetzen
- Vielfältige Node.js-Module zur Realisierung von Webservern, REST-basierten Web Services, Datenbankzugriffen, mehrsprachige Anwendungen
- Höchst skalierbare Architektur mit Echtzeitfähigkeiten
- Grundlage für weitere Frameworks wie *Express*

3.2 JavaScript-Grundlagen
Wichtige Merkmale

Keywords

Reserved keywords as of ECMAScript 6

- break
- case
- class
- catch
- const
- continue
- debugger
- default
- delete
- do
- else
- export
- extends
- finally
- for
- function
- if
- import
- in
- instanceof
- let
- new
- return
- super
- switch
- this
- throw
- try
- typeof
- var
- void
- while
- with
- yield

http://developer.mozilla.org/

Skript-Sprachen

- Programmiersprachen die nicht vor der Ausführung durch einen *Compiler* übersetzt werden, sondern während der Ausführung durch einen *Interpreter*
- Einfacher in der Umsetzung als Compiler-Sprachen, da Compile-Zeit entfällt (zum Beispiel nach kleinen Änderungen, keine Einbindung komplexer Bibliotheken, ...)
- Interpretierte Sprachen benötigen längere Ausführungszeit da die Übersetzung während der Ausführung erfolgt

Dynamische Typisierung

- Datentypen werden dynamisch zur Laufzeit ermittelt
- Keine Möglichkeiten eine Variable mit einem Typ zu deklarieren
- Typ einer Variablen kann sich zur Laufzeit ändern
- Automatische Konvertierung von Typen, beispielsweise bei Vergleichen mit dem ==-Operator

Funktionale Programmierung

- Funktionen als *erstklassige Objekte*, d.h. sie können Variablen zugewiesen und als Parameter anderer Funktionen verwendet werden
- Deklarativ: Man bestimmt was ein Programm macht, nicht wie es etwas macht

Prototypische Objektorientierung

- Umsetzung des objektorientierten Paradigmas basierend auf *Prototypen*, nicht Klassen

3.2 JavaScript-Grundlagen

Datentypen und Werte - Zahlen und Zeichenketten

```
var ganzZahl = 5;
var fliesskommazahl=5.4;
console.log(typeof ganzZahl);        // number
console.oog(typeof fliesskommazahl);  // number
```

```
var interpret = 'DJ Shadow';
var titel = "Endtroducing";
var meldung1 = "Der Titel lautet \"Endtroducing\"";// mit Escape-Sequenz
var meldung2 = 'Der Titel lautet "Endtroducing"';  // ohne Escape-Sequenz
```

Zahlen

- Keine Unterscheidung zwischen Ganzzahlen und Fließkommazahlen
- Ale Zahlen werden als 64-Bit-Fließkommazahlen dargestellt
- Dezimalschreibweise (ohne Präfix), Hexadezimalschreibweise (mit Präfix 0x) oder Oktalschreibweise (mit Präfix O)
- Keine Unterstützung der Binärschreibweise
- Liegt ein Wert außerhalb des Wertebereichs, wird `Infinity` als Wert gesetzt
- Entspricht ein Wert nicht einem Zahlenwert, wird NaN (not a number) als Wert gesetzt (zum Beispiel bei einer Division durch 0)

Zeichenketten

- Bestehen aus 16-Bit-Zeichen nach UCS-2-Kodierung
- Definiert durch einfache oder doppelte Anführungszeichen
- Kein Datentyp `char` in JavaScript für einzelne Zeichen)
- Zugriff auf einzelne Zeichen einer Zeichenkette mit `charAt()`
- Methoden die auf einer Zeichenkette ausgeführt werden verändern diese nicht, sondern geben eine neue Zeichenkette zurück
- Zeichenketten können mit den Operatoren < und > verglichen werden

3.2 JavaScript-Grundlagen

Datentypen und Werte - Boolean, Undefined und Null

`false`, `0` und leere Zeichenketten sind gleich

```
console.log(false == 0);                    // true
console.log(false == "");                   // true
console.log(0 == "");                       // true
```

`null` und `undefined` sind nur untereinander gleich

```
console.log(null == false);                 // false
console.log(null == true);                  // false
console.log(null == null);                  // true
console.log(undefined == undefined);        // true
console.log(undefined == null);             // true
```

NaN ist zu nichts gleich, nicht einmal zu sich selbst

```
console.log(NaN == false);                  // false
console.log(NaN == null);                   // false
console.log(NaN == NaN);                    // false
```

Innerhalb von booleschen Bedingungen evaluieren *truthy*-Werte zu `true` und *falsy*-Werte zu `false`

```
console.log(false == null);                 // false
if(null) {
  console.log("null");
} else if (!null) {
  console.log("!null");                     // Ausgabe
}
console.log(true=={});                      // false
if({}) {
  console.log("{}");                        // Ausgabe
} else if(!{}) {
  console.log("!{}");
}
```

Boolean

- Neben den booleschen Werten `false` und `true` interpretiert JavaScript auch nicht-boolesche Werte entweder als *falsy* oder *truthy*
- `null`, `undefined`, leere Strings, `0` und NaN zählen zu den Werten die als *falsy* interpretiert werden; alle anderen Werte werden als *truthy* interpretiert
- Aber: Vergleichsalgorithmus von JavaScript liefert für `null==false` und `null==true` in beiden Fällen `false` (ebenso für `undefined`) weil sich `null` und `undefined` nicht auf `0` oder `1` abbilden lassen, d.h. `null` und `undefined` sind nur untereinander gleich

`undefined`

- Globale Variable
- Variablen die nicht initialisiert wurden, nicht existente Objekteigenschaften sowie nicht vorhandene Funktionsparameter haben den Wert `undefined`, zeigen also auf die globale Variable

`null`

- Schlüsselwort (Literal)
- Wenn oftmals verwendet wenn ein Objekt optional genutzt werden kann

3.2 JavaScript-Grundlagen

Datentypen und Werte - Objekte

```
var person = {
  name : 'Max',
  nachname : 'Mustermann';
  sprechen : function() {
    console.log('Hallo');
  }
}
```

```
console.log(person.name);          // Max
console.log(person['nachname']);   // Mustermann
```

```
delete person.name;
console.log(person.name);      // Ausgabe: undefined
delete person['nachname'];
console.log(person.nachname); // Ausgabe: undefined
```

Objekte

- Container für Schlüssel-Wert-Paare
- Über einen Schlüssel kann auf den dahinter liegenden Wert zugegriffen werden
- Wert kann entweder ein Literal, eine Funktion oder ein anderes Objekt sein
- Möglichkeiten der Erstellung von Objekten
  - *Konstruktorfunktion*
  - *Objekt-Literal-Schreibweise*
  - Mittels der Funktion `Object.create()`
- Zugriff auf Eigenschaften eines Objektes erfolgt entweder über die Punktschreibweise oder über die `[]`-Notation
- Hinweis: Punktschreibweise funktioniert nur für Eigenschaften mit gültigen Variablennamen - für Eigenschaften ohne gültigen Namen (zum Beispiel solche die Bindestrich enthalten) muss die `[]`-Notation verwendet werden
- Objekte sind im Gegensatz zu primitiven Datentypen veränderbar, d.h. Eigenschaften können nachträglich geändert werden
- Mittels delete können Objekteigenschaften gelöscht werden

3.2 JavaScript-Grundlagen
Datentypen und Werte - Arrays

Arrays

- Konstruktorenfunktion new `Array()` oder über die Literal-Kurzschreibweise deklariert werden

Erzeugung mittels Konstruktorenfunktion

```
var interpreten = new Array();
interpreten[0] = 'Kyuss';
interpreten[1] = 'Baby Woodrose';
interpreten[2] = 'Hermano';
interpreten[3] = 'Monster Magnet';
interpreten[4] = "Queens of the Stone Age';
```

Erzeugung mittels Literal-Kurzschreibweise

```
var interpreten = [
  'Kyuss',
  'Baby Woodrose',
  'Hermano',
  'Monster Magnet',
  'Queens of the Stone Age'
];
```

```
new Array(10);   // erzeugt ein Array der Länge, wobei alle Werte undefined sind
new Array(10,11); // erzeugt ein Array der Länge 2 mit den werten 10 und 11
```

Element	Funktion
`concat()`	Hängt Elemente oder Arrays an ein bestehendes Array an.
`filter()`	Filtert Elemente aus dem Array auf Basis eines in Form einer Funktion übergebenen Filterkriteriums.
`forEach()`	Wendet eine übergebene Funktion auf jedes Element im Array an.
`join()`	Wandelt ein Array in eine Zeichenkette um.
`map()`	Bildet die Elemente eines Arrays auf Basis einer übergebenen Umwandlungsfunktion auf neue Elemente ab.
`pop()`	Entfernt das letzte Element eines Array.
`push()`	Fügt ein neues Element am Ende des Arrays ein.
`reduce()`	Fasst die Elemente eines Arrays auf der Basis einer übergebenen Funktion zu einem Wert zusammen.
`reverse()`	Kehrt die Reihenfolge der Elemente im Array um.
`shift()`	Entfernt das erste Element eines Arrays.
`slice()`	Schneidet einzelne Elemente aus einem Array heraus.
`splice()`	Fügt neue Elemente an beliebiger Position im Array hinzu.
`sort()`	Sortiert das Array, optional auf Basis einer übergebenen Vergleichsfunktion.

3.2 JavaScript-Grundlagen

Datentypen und Werte - Variablen


```
var v = 5;
console.log(typeof v); // "number"
var v = 'Hallo';
console.log(typeof v); // "string"
let w = 5;
```

Variablen

- Deklaration über die Schlüsselwörter `var` und `let` (ab ECMAScript6)
- Deklaration erfolgt ohne Typangabe, Bestimmung des Datentyps dynamisch zur Laufzeit bei Wertzuweisung der Variablen
- Mit `let` angelegte Variablen sind nur im aktuellen Codeblock sichtbar, mit `var` angelegte Variablen innerhalb der gesamten Funktion innerhalb der sie definiert wurden oder global wenn sie nicht innerhalb einer Funktion definiert wurden

Globale Variablen

- Nicht mittels `var` oder `let` angelegte Variablen sind global
- Globale Variablen werden als Eigenschaften des globalen Objekts definiert (zum Beispiel im Browser das Objekt `window`)
- Variablen die ohne `var` angelegt werden können Eigenschaften des globalen Objektes überschreiben (was man vermeiden sollte)

```
const LOG_LEVEL_DEBUG = 'debug';
console.log(LOG_LEVEL_DEBUG); // Ausgabe: debug
LOG_LEVEL_DEBUG = 'info';
console.log(LOG_LEVEL_DEBUG); // Ausgabe: debug
```

Konstanten

- Ab ECMAScript6 können Konstanten mit dem Schlüsselwort `const` definiert werden
- Wert einer Konstanten kann nach Initialisierung nicht mehr verändert werden

Namenswahl

- Variablennamen müssen mit einem Buchstaben, einem Unterstrich oder dem Dollarzeichen beginnen
- Darauf folgende Zeichen sind Buchstaben, Ziffern oder der Unterstrich

Datentypen und Werte - Funktionen (I)

Deklaration mittels Funktionsanweisung

```
function addition(zahl1, zahl2) {
  return zahl1 + zahl2;
};
```

Überprüfung des Typs von Funktionsparametern

```
function addition(zahl1, zahl2) {
  if((typeof zahl1 !== "number") || (typeof zahl2 !== "number")) {
    throw new TypeError("Parameter müssen Zahlen sein.");
    }
    return zahl1 + zahl2;
};
```

Deklaration mittels Funktionsausdruck

```
var addition = function additionsFunction(zahl1, zahl2) {
  return zahl1 + zahl2;
};
```

Deklaration einer anonymen Funktion und Zuweisung an Variable

```
var addition = function(zahl1, zahl2) {
  return zahl1 + zahl2;
};
```

Deklaration mittels `Function`-Konstruktor

```
var addition = new Function("zahl1", "zahl2", "return zahl1 +
zahl2");
```

Funktionen

- "First class", d.h. können als Parameter anderer Funktionen verwendet, Variablen zugewiesen oder als Rückgabewert einer Funktion verwendet werden
- Verschiedene Möglichkeiten der Deklaration von Funktionen
  - über eine *Funktionsanweisung* (*function statement*)
  - über einen *Funktionsausdruck* (*function expression*)
  - über einen Konstruktor des `Function`-Objekts
- Keine Typangabe bei Eingabewerten
- Weder Parameter noch Typ des Rückgabewertes werden explizit angegeben

Datentypen und Werte - Funktionen (II)

Technische Universität Berlin

Funktionsaufruf

```
var ergebnis1 = addition(2,2);
console.log(ergebnis1); // Ausgabe: 4
var ergebnis2 = addition('Hallo ', 'Welt');
console.log(ergebnis2); // Ausgabe: Hallo Welt
```

Funktionen aufrufen

Dynamische Anzahl von Funktionsparametern mit `arguments`

```
function addiereAlle1() {
  var ergebnis = 0;
  for(var i=0; i<arguments.length; i++) {
    ergebnis += arguments[i];
  }
  return ergebnis;
}
```

Dynamische Anzahl an Funktionsparametern

- Beim Aufruf einer Funktion steht innerhalb der Funktion ein Objekt `arguments` zur Verfügung, welches sämtliche Funktionsparameter enthält
- `arguments` wird verwendet wenn eine Funktion mit beliebig vielen oder einer variablen Anzahl von Parametern aufgerufen werden können soll
- `arguments` ähnlich zu einem Array mit Eigenschaften wie `length`
- Aber: keine Unterstützung typischer Array-Methoden wie `concat()`, `slice()` oder `forEach()`
- Ab ECMAScript6: `rest`-Parameter als Alternative zu `arguments`

Dynamische Anzahl von Funktionsparametern mit `rest`-Parameter

```
function addiereAlle2(...zahl) {
  var ergebnis = 0;
  for(var i=0; i<zahl.length; i++) {
    ergebnis += zahl[i];
  }
  return ergebnis;
}
```

3.2 JavaScript-Grundlagen
Kontrollstrukturen und Schleifen (I)

Verwendung von `if … else`

```
if(i>8) {
  console.log("i ist größer als 8");
} else {
  console.error("i ist kleiner oder gleich 8");
}
```

Klassische Verwendung boolescher Funktionen …

```
function beispiel(parameter) {
  if(parameter !== undefined && parameter !== null) {
    console.log("Definiert und nicht null");
  }
}
```

… und vereinfachte Version

```
function beispiel(parameter) {
  if(parameter) {
    console.log("Definiert und nicht null");
  }
}
```

if/else

- Analog zur Verwendung in Java und anderen Programmiersprachen
- Innerhalb der `if`-Klausel lassen sich nicht nur boolesche Werte, sondern Werte beliebigen Typs verwenden - jeder Wert in JavaScript evaluiert innerhalb boolescher Bedingungen entweder zu `true` oder `false`
- Remember: `undefined` und `null` evaluieren zu `false`

3.2 JavaScript-Grundlagen
Kontrollstrukturen und Schleifen (II)

```
function gibVier() {
  return 4;
}
function gibAuchVier() {
  return 4;
}
var s=4;
switch(s) {
  case gibVier(): console.log("gibVier"); break;
  case gibAuchVier(): console.log("gibAuchVier"); break;
  default: console.log("nichts");
}
// Ausgabe des Programms: gibVier
```

switch

- Anweisung für Mehrfachverzweigungen
- Unterstützt Werter beliebigen Typs
- Werte der einzelnen `case`-Ausdrücke lassen sich alternativ dynamisch über Funktionsaufrufe ermitteln (Vergleich: in Java müssen die Werte Konstanten sein)
- Ergeben mehrere Funktionsaufrufe den gleichen Wert, wird der `case`-Ausdruck ausgewählt, der als zuerst eintritt

```
var i = 10;
while (i > 0) {
  console.log(i);
  i--;
}
```

```
var i = 10;
do {
  console.log(i);
  i--;
} while (i > 0);
```

```
for (var i = 10; i > 0; i--) {
  console.log(i);
}
```

Schleifen

Unterstützung von `while`-, `do`- und `for`-Schleifen

Weitere Schleifenarten: `for…in` und `for…of`

3.2 JavaScript-Grundlagen
Fehlerbehandlung

```
function holePerson(id) {
  if (id<0) {
    throw new Error('ID darf keinen negativen Wert haben: '+id); }
  return {id : id}; // hier normalerweise holen der Personendaten aus der
  Datenbank
}

function holePersonen(ids) {
  var result=[];
  ids.forEach(function(id) {
    try {
      var person = holePerson(id);
      result.push(person);
    } catch (exception) {
      console.log(exception);
    }
  });
  return result;
}
```

```
>holePersonen([2, -5, 137])
[Error: ID darf keinen negativen Wert haben: -5]
[{id: 2}, {id: 137}]
```

mit Änderungen übernommen von: Ackermann, P. (2015) Professionell Entwickeln mit JavaScript, Rheinwerk Computing

Exceptions

- Ähnlich wie bei Java und C# mittels `try-catch-finally`
- `throw` wird nicht wie bei Java in der Methodendeklaration aufgeführt um die Art des Fehlers zu spezifizieren den die Methode werfen kann
- Mittels `throw` können beliebige Objekte geworfen werden, man sollte allerdings Objekte vom Typ `Error` oder davon abgeleitete Objekte bevorzugen
- Im Gegensatz zu anderen Programmiersprachen gibt es nur ein einziges `catch` pro Anweisung
- Argument der `catch`-Anweisung enthält eine Instanz des Error-Objektes mit den Basisattributen `name` (für die Angabe des Fehlertyps) und `message` (für die Übergabe des Fehlertextes)

3.2 JavaScript-Grundlagen
Operatoren (I)

- Neben den Standardoperatoren (+, -, *, /) bietet JavaScript eine Reihe weiterer Operatoren

Vergleichsoperatoren

Operation	Operator	Beschreibung
Gleichheit	==	Liefert `true` wenn die Operanden gleich sind.
Ungleichheit	!=	Liefert `true` wenn die Operanden nicht gleich sind.
strikte Gleichheit	===	Liefert `true` wenn die Operanden gleich sind und außerdem den gleichen Datentyp haben.
strikte Ungleichheit	!==	Liefert `true` wenn die Operanden nicht gleich sind und/ oder nicht den gleichen Datentyp haben.
größer als	>	Liefert `true` wenn der linke Operand größer als der rechte ist.
größer oder gleich	>=	Liefert `true` wenn der linke Operand größer als oder gleich dem rechten Operand ist.
kleiner als	<	Liefert `true` wenn der linke Operand kleiner als der rechte ist.
kleiner oder gleich	<=	Liefert `true` wenn der linke Operand kleiner als oder gleich dem rechten Operand ist.

Arithmetische Operatoren

Operation	Operator	Beschreibung
Modulo	%	Liefert den ganzzahligen Rest der Division der beiden Operanden.
Inkrement	++	Unärer Operator der den Operanden um eins erhöht. Kann sowohl als Präfix- als auch als Postfix-Operator verwendet werden.
Dekrement	--	Unärer Operator der eins vom Operanden subtrahiert. Kann sowohl als Präfix- als auch als Postfix-Operator verwendet werden.
unäre Negation	!	Unärer Operator der die Negation des Operanden liefert.

Logische Operatoren

Operation	Operator	Beschreibung
logisches UND	&&	Binärer Operator der den ersten Operanden zurückgibt falls dieser false ergibt. Ansonsten wird der zweite Operand zurückgegeben.
logisches ODER	||	Binärer Operator der den ersten Operanden zurückgibt falls dieser true ergibt. Ansonsten wird der zweite Operand zurückgegeben.
logisches NICHT	!	Unärer Operator den den Operanden negiert

3.2 JavaScript-Grundlagen
Operatoren (II)

Technische Universität Berlin

Bitweise Operatoren

Operation	Operator	Beschreibung
Bitweises UND	&	Überprüft für jede Bitposition ob der jeweilige Wert bei beiden Operanden ist. Liefert 1 zurück wenn ja, andernfalls 0.
Bitweises ODER	|	Überprüft für jede Bitosition ob der jeweilige Wert bei einem der beiden Operanden 1 ist. Liefert 1 zurück wenn ja, andernfalls 0.
Bitweises XOR	^	Überprüft für jede Bitposition ob der jeweilige Wert bei genau einem der beiden Operanden 1 ist. Liefert 1 zurück wenn ja, andernfalls 0.
Bitweises NICHT	~	Unärer Operator der die einzelnen Bits des Operanden invertiert.
Bitweise Linksverschiebung	<<	Bitweise Linksverschiebung des linken Operanden um die Anzahl der Stellen die durch den rechten Operator definiert wird.
Bitweise Rechtsverschiebung unter Beachtung des Vorzeichens	>>	Bitweise Rechtsverschiebung des linken Operanden um die Anzahl der Stellen die durch den rechten Operanden definiert wird.
Bitweise Rechtsverschiebung ohne Beachtung des Vorzeichens	>>>	Bitweise Rechtsverschiebung des linken Operanden um die Anzahl der Stellen die durch den rechten Operanden definiert wird ohne Beachtung des Vorzeichens.

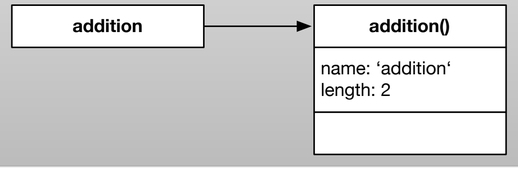
Spezielle Operatoren

Operation	Operator	Beschreibung
Konditionaler Operator	`<Bedingung> ? <Wert1>:<Wert2>`	Tertiärer Operator der abhängig von einer Bedingung (erster Operand) einen von zwei Werten zurückgibt (die durch den zweiten und dritten Operanden definiert werden).
Löschen von Objekten, Objekteigenschaften oder elementen eines Arrays	`delete`	Erlaubt das Löschen von Elementen in einem Array, das Löschen von Objekten sowie das Löschen von Objekteigenschaften.
Existenz einer Eigenschaft in einem Objekt	`<eigenschaft> in <objekt>`	Überprüft ob eine Eigenschaft in einem Objekt vorhanden ist.
Typüberprüfung	`<objekt> instanceof <typ>`	Binärer Operator der überprüft ob ein Objekt von einem Typ ist.
Typbestimmung	`typeof <operand>`	Ermittelt den Datentyp des Operanden. Der Operand kann ein Objekt, ein String, eine Variable oder ein Schlüsselwort sein. Optional kann der Operand in Klammern angegeben werden.

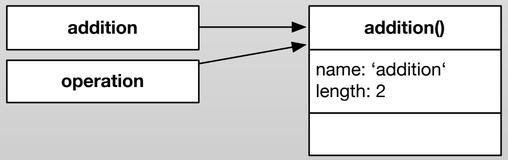
mit Änderungen übernommen von: Ackermann, P. (2015) Professionell Entwickeln mit JavaScript, Rheinwerk Computing

3.3 Funktionen und Funktionale Aspekte
Methoden und Funktionen

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




```
var operation = addition;
```




```
var ergebnis1 = addition(2,2);
var ergebnis2 = operation(2,2);

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

Methoden versus Funktionen

- *Methoden*: Funktionen die als Eigenschaft eines Objektes oder als Eigenschaft einer anderen Funktion definiert werden
- *Funktionen*: Funktionen die für sich stehen

Funktionen

- Funktionen werden durch Objekte repräsentiert, d.h. durch Instanzen des `function`-Objekts
- Bei Erzeugung einer Funktion wird ein Funktionsobjekt mit dem Namen der Funktion erzeugt sowie eine gleichnamige Variable die auf das Funktionsobjekt zeigt

Funktionen sind erstklassig

- Funktionen haben den gleichen Stellenwert wie Objekte oder primitive Datentypen, d.h. sie sind erstklassig (first class)
- Funktionen können
  - Variablen zugewiesen,
  - als Werte innerhalb von Arrays verwendet,
  - innerhalb von Objekten oder innerhalb anderer Funktionen definiert oder
  - als Parameter oder Rückgabewert von Funktionen verwendet werden.
- Wenn eine Funktion einer Variablen zugewiesen wird, zeigt die Variable anschließend auf das Funktionsobjekt
- Die Eigenschaften der Ursprungsfunktion, z.B. `name`, bleiben erhalten

3.3 Funktionen und Funktionale Aspekte

Funktionen in Arrays und als Funktionsparameter

Funktionen in Arrays

- Variablen die auf Funktionsobjekte zeigen können an allen Stellen verwendet werden an denen auch "normale" Variablen verwendet werden dürfen
- Beispiel zeigt die Definition von vier Funktionen und deren anschließende Wertübergabe an ein Array

```
function addition(x,y) {
  return x+y;
}
function subtraktion(x,y) {
  return x-y;
}
function multiplikation(x,y) {
 return x*y;
}
function division(x,y) {
return x/y;
}
var operationen=[
    addition,
    subtraktion,
    multiplikation,
    division];
```

```
var operation;
for(var i=0; i<operationen.length; i++) {
  operation = operationen[i];
  console.log(operation(2,2));
}
```

Funktionen als Funktionsparameter

- Funktionen können als Parameter einer anderen Funktion verwendet werden
- Findet Anwendung bei *asynchronen Funktionsaufrufen*, bei denen die Dauer der Bearbeitung unklar ist (zum Beispiel beim Download einer Datei)
- Übergebene Funktion (*Callback-Funktion* oder *Callback-Handler*) wird aufgerufen wenn das Ergebnis der asynchronen Funktion bereitsteht

```
function metaOperation(operation,x,y) {
  return operation(x,y);
}
```

```
function asynchroneFunktion(callback) {
  var ergebnis=0;
  /* Hier erfolgt die Berechnung von ergebnis */
  callback(ergebnis);
}
```

Funktionen als Rückgabewert

```
function operationenFabrik(name) {
  switch(name) {
    case: 'addition': return function(x,y) {
      return x+y;
    }
    case: 'subtraktion': return function(x,y) {
      return x-y;
    }
    case: 'multiplikation': return function(x,y) {
      return x*y;
    }
    case: 'division': return function(x,y) {
      return x/y;
    }
    default: return function() {
      return NaN;
    }
  }
}
```

```
var addition=operationenFabrik('addition');
console.log(addition(2,2));
var subtraktion=operationenFabrik('subtraktion');
console.log(subtraktion(2,2));
var multiplikation=operationenFabrik('multiplikation');
console.log(multplikation(2,2));
var division=operationenFabrik('division');
console.log(division(2,2));
var nichts=operationenFabrik('nichts');
console.log(nichts(2,2));
```

Funktionen liefern andere Funktionen als Rückgabewert

Zurückgegebene Funktionen können direkt aufgerufen werden

```
console.log(operationenFabrik('addition')(2,2));
console.log(operationenFabrik('subtraktion')(2,2));
console.log(operationenFabrik('multiplikation')(2,2));
console.log(operationenFabrik('division')(2,2));
```

Funktionen als Rückgabewert

- Funktionen können andere Funktionen als Rückgabewert liefern
- Funktionen können anonym direkt innerhalb eines Ausdrucks definiert werden, d.h. ohne Zuweisung zu einer Variablen
- Zurückgegebene Funktionen können ihrerseits ebenfalls eine Funktion als rückgabewert liefern und diese wiederum usw.

3.3 Funktionen und Funktionale Aspekte

Funktionen innerhalb von Funktionen und innerhalb von Objekten

```
function operationenContainer(x,y) {
  var addition=function(x,y) {
    return x+y;
  }
  var subtraktion=function(x,y) {
    return x-y;
  }
  var multiplikation=function(x,y) {
    return x*y;
  }
  var division=function(x,y) {
    return x/y;
  }
  console.log(addition(x,y));
  console.log(subtraktion(x,y));
  console.log(multiplikation(x,y));
  console.log(division(x,y));
}
operationenContainer(2,2);
```

Funktionen innerhalb von Funktionen

- Funktionen können innerhalb anderer Funktionen definiert werden
- Beachte: Funktionen im Beispiel sind keine Methoden
- Funktionen sind außerhalb der umgebenden Funktion nicht sichtbar und können von dort nicht direkt aufgerufen werden

```
var operationen = {
  addition: function(x,y) {
    return x+y;
  },
  subtraktion: function(x,y) {
    return x-y;
  },
  multiplikation: function(x,y) {
    return x*y;
  },
  division: function(x,y) {
    return x/y;
  }
}
console.log(operationen.addition(2,2));
console.log(operationen.subtraktion(2,2));
console.log(operationen.multiplikation(2,2));
console.log(operationen.division(2,2));
```

Methoden

- Definition von Funktionen innerhalb eines Objekts
- Aufruf der Methode über die jeweilige Objektreferenz

```
var operationen = {
  addition(x,y) {
    return x+y;
  },...
```

Alternative Schreibweise ohne `function`-Schlüsselwort ab ECMAScript 6

3.3 Funktionen und Funktionale Aspekte

Ausführungskontext einer Funktion (I)

`this` im Kontext eines Objekts bezieht sich auf das Objekt

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

Globale Funktion in der `this` verwendet wird

```
function getNameGlobal() {
  return this.name;
}
console.log(getNameGlobal()); // undefined
```

`this` im globalen Kontext bezieht sich auf das globale Objekt

```
var name="globaler Name";
function getNameGlobal() {
  return this.name;
}
console.log(getNameGlobal()); // Ausgabe: globaler Name
```

Im strikten Modus ist `this` im globalen Kontext nicht definiert

```
"use strict";
var name="globaler Name";
function getNameGlobal() {
  return this.name;
}
console.log(getNameGlobal()); // Fehler: this nicht
definiert
```

Ausführungskontext

- C# oder Java: Schlüsselwort `this` spricht innerhalb einer Objektmethode die jeweilige Objektinstanz an für welche die Methode definiert wurde
- Grundsätzlich andere Bedeutung von `this` in JavaScript
- *Ausführungskontext*: `this` innerhalb einer Funktion bezieht sich nicht auf das Objekt in dem die Funktion *definiert* wurde, sondern auf das Objekt in dem die Funktion *ausgeführt* wird
- Je nachdem ob eine Funktion als globale Funktion oder als Methode eines Objekts aufgerufen wird, hat `this` einen anderen Wert

Das globale Objekt

- Hängt von der Laufzeitumgebung ab
- Im Webbrowser ist `window` das globale Objekt, in NodeJS hängt es vom verwendeten Modul ab
- Wird eine Funktion im globalen Scope aufgerufen, bezieht sich `this` auf das globale Objekt
- Ausnahme: strikter Modus, hier hat `this` innerhalb einer globalen Funktion den Wert `undefined`

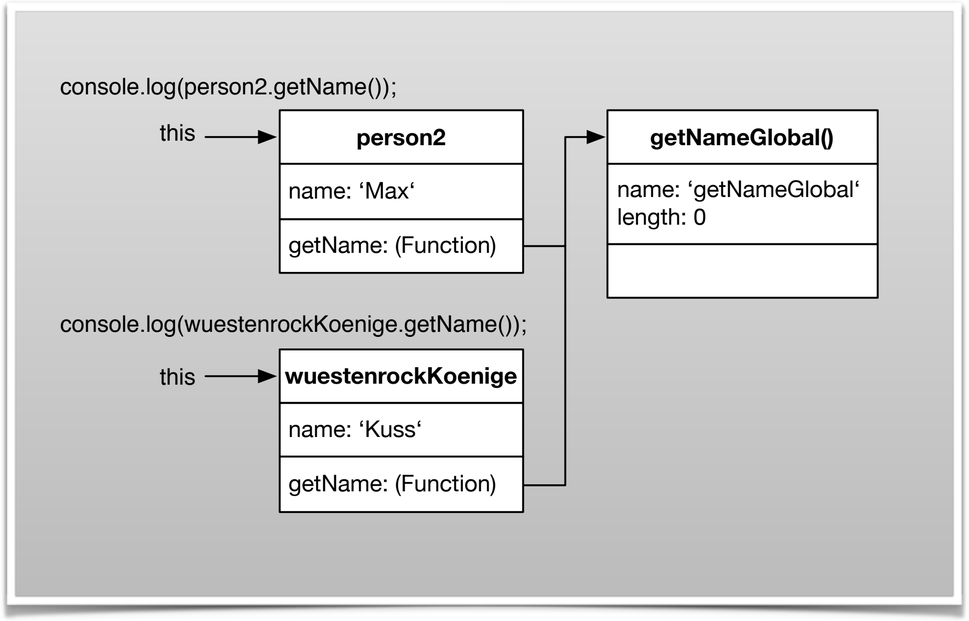
3.3 Funktionen und Funktionale Aspekte
Ausführungskontext einer Funktion (II)

`this` bezieht sich auf den Kontext der Funktion

```
var name="globaler Name";
function getNameGlobal() {
  return this.name;
}
var person2={
name: 'Moritz',
getName: getNameGlobal
}
var wuestenrockKoenige={
name: 'Kyuss',
getName: getNameGlobal
}
console.log(person2.getName()); //Ausgabe: Moritz
console.log(wuestenrockKoenige.getName()); //Ausgabe: Kyuss
```

Zusammenfassung Ausführungskontext

- Bei Aufruf einer globalen Funktion bezieht sich `this` auf das globale Objekt (bzw. ist im strikten Modus nicht definiert)
- Wird eine Funktion als Objektmethode aufgerufen, bezieht sich `this` auf das Objekt
- Wird eine Funktion als Konstruktorfunktion aufgerufen, bezieht sich `this` auf das Objekt das durch den Funktionsaufruf erzeugt wird



3.3 Funktionen und Funktionale Aspekte
Sichtbarkeitsbereich von Funktionen (I)

Zugriff auf Variablen die deklariert aber nicht initialisiert sind

```
function beispiel() {
  var y;
  console.log(y);
}
beispiel(); // Ausgabe: undefined
```

Zugriff auf Variablen die weder deklariert noch initialisiert sind

```
function beispiel() {
  console.log(y);
}
beispiel(); // ReferenceError
```

Zugriff auf Variablen die deklariert und initialisiert sind

```
function beispiel() {
  var y=4711;
  console.log(y);
}
beispiel(); // Ausgabe 4711
```

Sichtbarkeitsbereich von Funktionen

- Kein *Block-Scope* für Variablen, d.h. { und } spannen keinen Gültigkeits- oder Sichtbarkeitsbereich für Variablen auf
- Gültigkeitsbereich von Variablen wird durch die umgebende Funktion begrenzt
- *Function-Level-Scope*: Variablen die innerhalb einer Funktion definiert werden sind innerhalb der gesamten Funktion sichtbar sowie innerhalb anderer (innerer) Funktionen, die in der (äußeren) Funktion definiert sind

Hoisting

- Variablendeklarationen werden vom JavaScript-Interpreter immer an den Beginn der jeweiligen Funktion *gehoben* (von engl. to hoist), siehe nächste Folie
- Zur Vermeidung von Fehlern (z.B. Doppelbelegungen von Variablennamen) sollte alle Variablen bereits zu Beginn einer Funktion mittels `var` deklariert werden

3.3 Funktionen und Funktionale Aspekte

Sichtbarkeitsbereich von Funktionen (II)

```
function beispiel() {
  if(x) {
    var y=4711;
  }
  for (var i=0; i<4711; i++) {
    /* Irgendwas machen */
  }
  console.log(y);
  console.log(i);
}
beispiel(true);
```

- Trotz der Deklaration und Initialisierung der Variablen `y` und `i` in der `if`-Anweisung, wird außerhalb der Codeblöcke auf beider Variablen zugegriffen
- Ausgegeben wird zweimal der Wert 4711

```
function beispiel() {
  console.log(y);
  console.log(i);
  if(x) {
    var y=4711;
  }
  for (var i=0; i<4711; i++) {
    /* Irgendwas machen */
  }
}
beispiel(true);
```

- Vermutung: es kommt zu einem `ReferenceError` (weil beide Variablen zum Zeitpunkt des Aufrufs weder deklariert noch initialisiert)
- Vermutung ist falsch wegen Hoisting
- Ausgabe ist `undefined`

```
function beispiel() {
  var y;
  var i;
  console.log(y);
  console.log(i);
  if(x) {
    y=4711;
  }
  for (i=0; i<4711; i++) {
    /* Irgendwas machen */
  }
}
beispiel(true);
```

- Hoisting: der Interpreter verschiebt intern alle Deklaration innerhalb einer Funktion an den Anfang
- Variablen werden durch Hoisting deklariert, aber nicht initialisiert

```
function beispiel() {
  var y, i;
  console.log(y);
  console.log(i);
  if(x) {
    y=4711;
  }
  for (var i=0; i<4711; i++) {
    /* Irgendwas machen */
  }
}
beispiel(true);
```

- Gute Programmierpraxis: explizite Deklaration aller Variablen am Anfang der Funktion

3.4 Objektorientierung

Überblick und Objekt-Literal-Schreibweise zur Erzeugung von Objekten

Arten von Objekten

- *Native Objekte*: vordefinierte Objekte die bereits durch die Sprach selbst zur Verfügung gestellt werden (Beispiele `String`, `Number`, `Array`, `Image`, `Date`, `Math`, `Object`, ...)
- *Host-Objekte*: Objekte die von der Laufzeitumgebung zur Verfügung gestellt werden (Beispiele für Webbrowser sind `window` (repräsentiert das Browser-Fenster) und `document` (repräsentiert das Webdokument))
- *Benutzerdefinierte Objekte*: Objekte die der Entwickler selbst erstellt

Objekte erstellen

- Erzeugung durch die *Objekt-Literal-Schreibweise*
- Erzeugung über eine *Konstruktorfunktion*
- Erzeugung über die Helfermethode `Object.create()`

Objekt-Literal-Schreibweise

- Datenstrukturen aus Schlüssel-Wert-Paaren
- *Schlüssel* repräsentieren die Eigenschaften eines Objektes
- *Werte* sind entweder andere Objekte, primitive Datentypen oder Funktionen
- Verfügt ein Objekt über eine Funktion, heißt diese *Methode* oder *Objektmethode*
- Beliebige Schachtelung von Objekten möglich
- Alternativ zur Schachtelung können Objekte getrennt voneinander definiert und dann mit Referenzen eingebunden werden
- Bevorzugte Lösung wenn einfache Objekte benötigt werden, von denen es nur eine Instanz geben soll (*Singletons*)

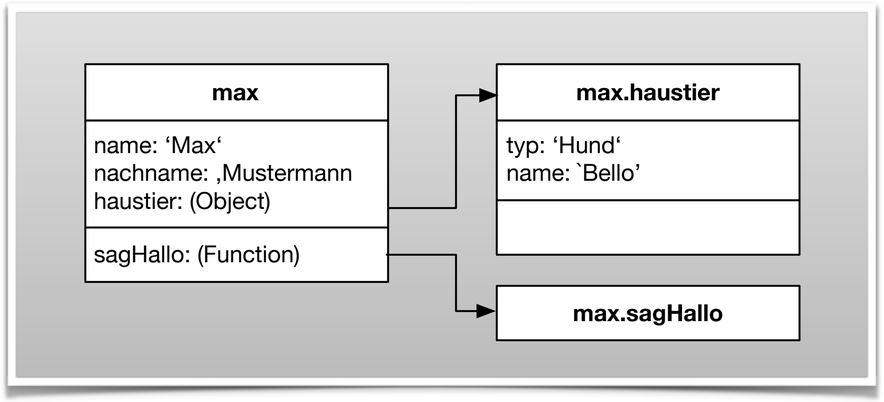
```
var max={
  name: 'Max',
  nachname: 'Mustermann',
  sagHallo: function() {
    console.log('Hallo');
  }
}
```

3.4 Objektorientierung

Schachtelung und Referenzierung von Objekten

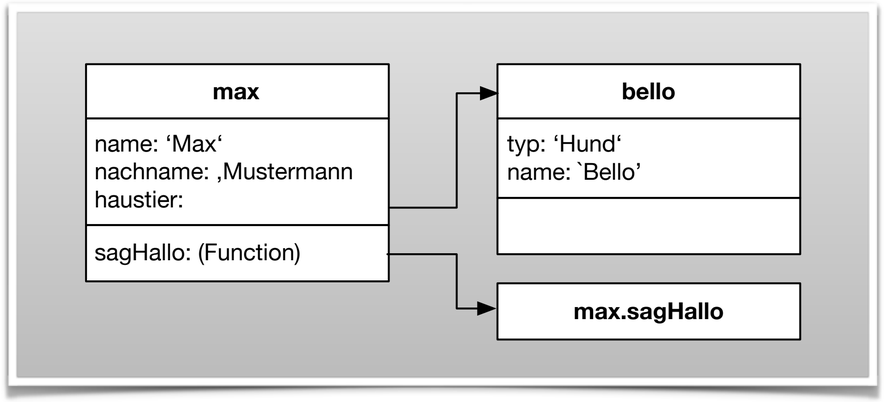
Schachtelung von Objekten in der Objekt-Literal-Schreibweise

```
var max={
  name: 'Max',
  nachname: 'Mustermann',
  sagHallo: function() {
    console.log('Hallo');
  },
  haustier: {
    name: 'Bello',
    typ: 'Hund'
  }
}
```



Einbindung/Referenzierung externer Objekte in der Objekt-Literal-Schreibweise

```
var bello={
  name: 'Bello',
  typ: 'Hund'
}

var max={
  name: 'Max',
  nachname: 'Mustermann',
  sagHallo: function() {
    console.log('Hallo');
  },
  haustier: bello
}
```



3.4 Objektorientierung
Konstruktorfunktionen (I)

```
function Konstruktor () {
    // Zugriff auf das neue Objekt über this,
    // Hinzufügen der Eigenschaften und Methoden
    this.eigenschaft = "wert";
    this.methode = function () {
        // In den Methoden wird ebenfalls über this auf das Objekt zugegriffen
        alert("methode wurde aufgerufen\n" +
              "Instanz-Eigenschaft: " + this.eigenschaft);
    };
}

// Erzeuge Instanzen
var instanz1 = new Konstruktor();
instanz1.methode();
var instanz2 = new Konstruktor();
instanz2.methode();
// usw.
```

Konstruktorfunktionen

- *Erzeuger* (engl. *construct*=erzeugen) neuer Objekte
- Aufruf mit `new Function`
- Definition einer Konstruktorfunktion analog zu der normaler Funktionen
- Ausnahme: Konstruktorfunktionen werden nicht mit einer `return`-Anweisung beendet
- Zur Abgrenzung zu normalen Funktionen wird für Konstruktorfunktionen i.d.R. die Upper-Camel-Case-Schreibweise (erster Buchstabe groß, Rest klein) verwendet
- Wenn eine Funktion mit `new` aufgerufen wird, wird intern zunächst ein neues, leeres `Object`-Objekt (Instanzobjekt) angelegt und die Funktion im Kontext dieses Objektes ausgeführt
- Im Konstruktor kann das neu angelegt Objekt über `this` angesprochen werden
- Hauptunterschied zur Objekt-Literal-Schreibweise: es lassen sich unzählige gleich ausgestattete Objekte erzeugen (Instanzen)

3.4 Objektorientierung
Konstruktorfunktionen (II)

```
function Katze (name, rasse) {
    this.name = name;
    this.rasse = rasse;
    this.pfoten = 4;
}
var maunzi = new Katze('Maunzi', 'Perserkatze');
```

```
var maunzi = new Katze('Maunzi', 'Perserkatze');
console.log(maunzi.name + ' ist eine ' + maunzi.rasse);
maunzi.rasse = 'Siamkatze';
console.log(maunzi.name + ' ist neuerdings eine ' + maunzi.rasse);
```

- Konstruktorfunktionen können Parameter zur Erzeugung von Instanzen mit unterschiedlichen Eigenschaften entgegennehmen
- Dem Konstruktor übergebene Parameter werden zu Eigenschaften des Instanzobjektes
- Verschiedene Instanzen können abweichende Eigenschaftswerte haben, aber sie verfügen über dieselben Eigenschaften
- Eigenschaften können nach der Erzeugung der Instanz durch `instanzobjekt.member` neue Werte bekommen

```
var maunzi = Katze('Maunzi', 'Perserkatze);
console.log(maunzi.name);
//TypeError: Cannot read property 'name' of undefined
```

Gefahr von Laufzeitfehlern

- Jede Konstruktorfunktion kann durch Weglassen von new auch als normale Funktion aufgerufen werden
- Gefahr von Laufzeitfehlern: es gibt keinen Rückgabewert und `this` wird im Aufrufkontext interpretiert

3.4 Objektorientierung
Object.create()

```
var max=Object.create(Object.prototype);
max.name='Max';
max.nachname='Mustermann';
console.dir(max);
```

```
var max=Object.create(Object.prototype, {
  name: {
    value: 'Max',
    writable: false,
    configurable: true,
    enumerable: true
    },
  nachname: {
    value: 'Mustermann',
    writable: true,
    configurable: true,
    enumerable: true
  }
});
console.dir(max);
```

- Erzeugung von Objekten mittels `Object.create()` (ab ECMAScript 5)
- Erster Übergabeparameter bezeichnet den Prototyp des zu erstellenden Objektes (eine Art Vorlage für das Objekt)
- Nach Erzeugung des Objektes können Objekteigenschaften definiert werden
- Alternativ können Eigenschaften (und die Eigenschaften der Eigenschaften, sogenannte Eigenschaftsdeskriptoren) als zweiter Parameter beim Aufruf von `Object.create()` festgelegt werden

Attribute bei Property-Deskriptoren

- `value`: bezeichnet den Wert der Eigenschaft
- `writable`: kennzeichnet ob die Eigenschaft verändert werden darf oder nicht (standardmäßig true)
- `configurable`: kennzeichnet ob die Attribute einer Eigenschaft verändert werden dürfen (standardmäßig true)
- `enumerable`: kennzeichnet ob die Eigenschaft bei der Iteration über dieObjekteigenschaften berücksichtigt werden soll oder nicht (standardmäßig true)
- `get`: bezeichnet die Funktion die aufgerufen werden soll wenn lesend auf die Eigenschaft zugegriffen wird
- `set`: bezeichnet die Funktion die aufgerufen werden soll wenn schreibend auf die Eigenschaft zugegriffen wird

3.4 Objektorientierung
Prototypen (I)

```
var max={
  name: 'Max',
  nachname: 'Mustermann'
  };
console.log(max.__proto__);              // Object {}
console.log(Object.getPrototypeOf(max)); // Object {}
```

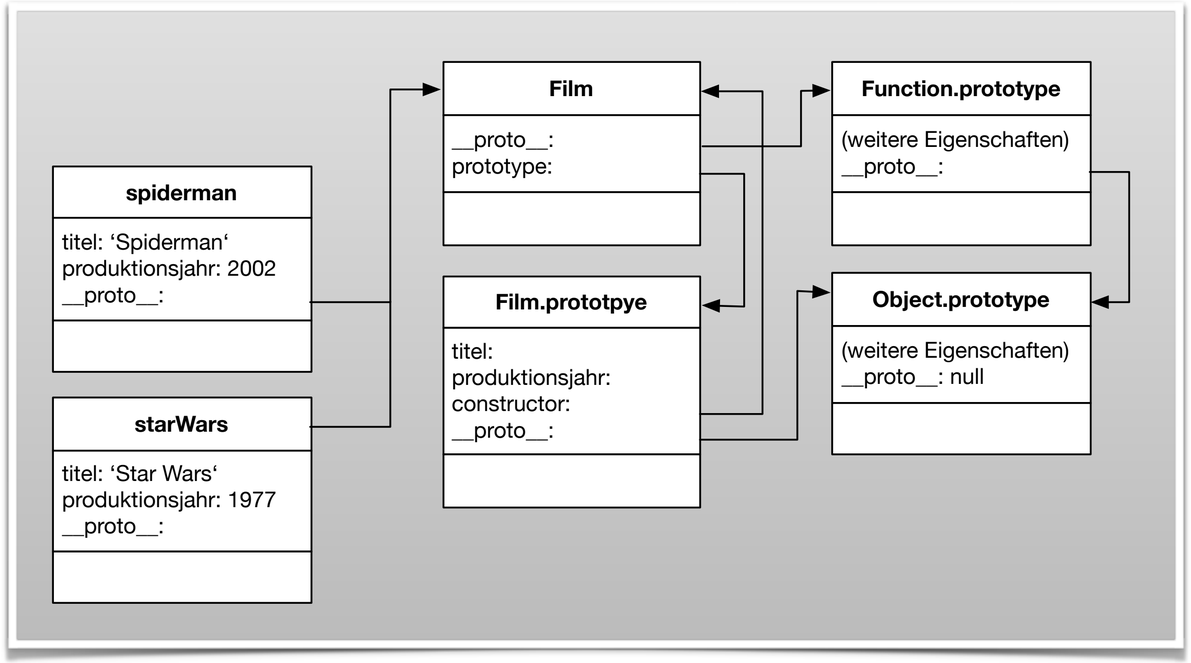
```
var maexchen=Object.create(max);
maexchen.name='Maexchen';
console.log(maexchen.__proto__);
// Object {name: 'Max', nachname: 'Mustermann'}
console.log(Object.getPrototypeOf(maexchen));
// Object {name: 'Max', nachname: 'Mustermann'}
console.log(maexchen.name);     // Maexchen
console.log(maexchen.nachname); // Mustermann
```

Prototyp

- Vorlage oder Blaupause (Blueprint) für ein Objekt
- Bei Verwendung einer Konstruktorfunktion oder der Objekt-Literal-Schreibweise ist standardmäßig das allgemeine Objekt `Object` der Prototyp für das anzulegende Objekte
- Eine Konstruktorfunktion `name` erzeugt ein Objekt `name.prototype`, welches
  - als Vorlage für die Objektinstanzen dient
  - als Vorlage das Objekt `Object.prototype` hat
- Über die (versteckte) Eigenschaft `[[prototype]]` (oder `__proto__`) eines Objektes kann auf die Prototypen eines Objektes zugegriffen werden
- Über die Eigenschaft `prototype` einer Konstruktorfunktion kann auf den erzeugten Prototyp zugegriffen werden
- Bei Erzeugung mittels `Object.create()` gibt man den Prototypen explizit als Parameter an (Grundlage für die *Prototypische Vererbung*)

3.4 Objektorientierung
Prototypen (II)


```
function Film(titel, produktionsjahr) {
  this.titel=titel;
  this.produktionsjahr=produktionsjahr;
};

var spiderman=new Film('Spiderman', 2002);
var starWars=new Film('starWars', 1977);

console.log(spiderman.__proto__);     // Film {}
console.log(starWars.__proto__);      // Film {}
console.log(spiderman.constructor);   // function Film() {...}
console.log(starWars.constructor);    // function Film() {...}
```

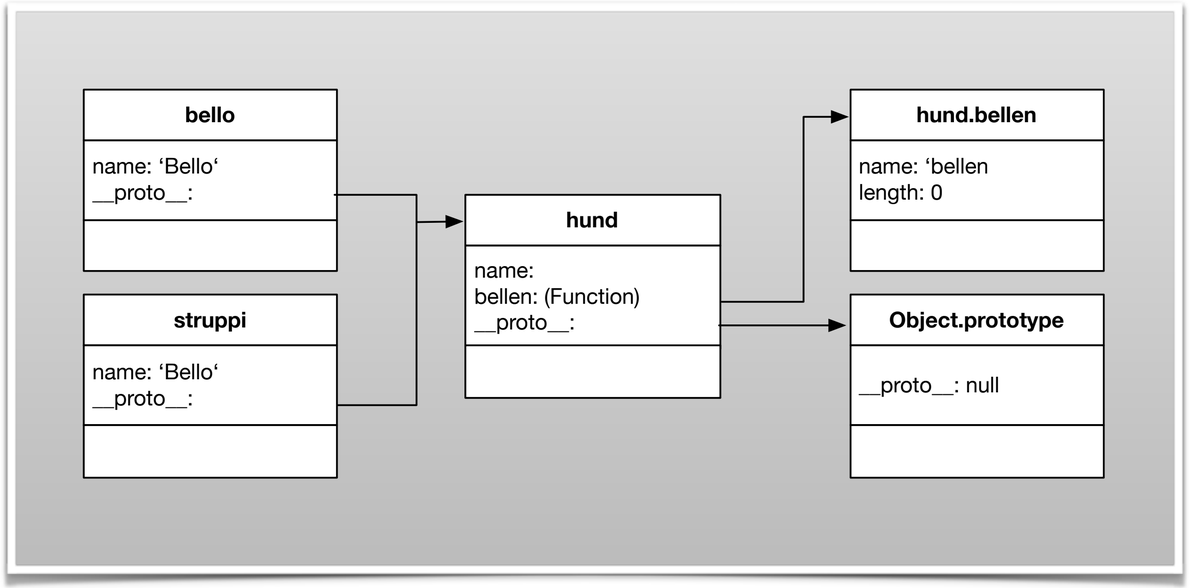


3.4 Objektorientierung
Prototypische Vererbung (I)

Prototypische Vererbung

- Basiert ausschließlich auf Objekten (keinen Klassen wie bei Java)
- Eigenschaften und Methoden eines Objektes kann ein anderes Objekt erben, indem es das Objekt als Prototyp verwendet
- `Object.create()` ist der einfachste Weg ein neues Objekt basierend auf einem Prototyp zu erzeugen

```
var hund={
  name: undefined,
  bellen: function() {
    console.log('Wau');
  }
}
var bello=Object.create(hund);
bello.name='Bello';
var struppi=Object.create(hund);
struppi.name='Struppi';
hund.bellen();    // Wau
bello.bellen();   // Wau
struppi.bellen(); // Wau
```

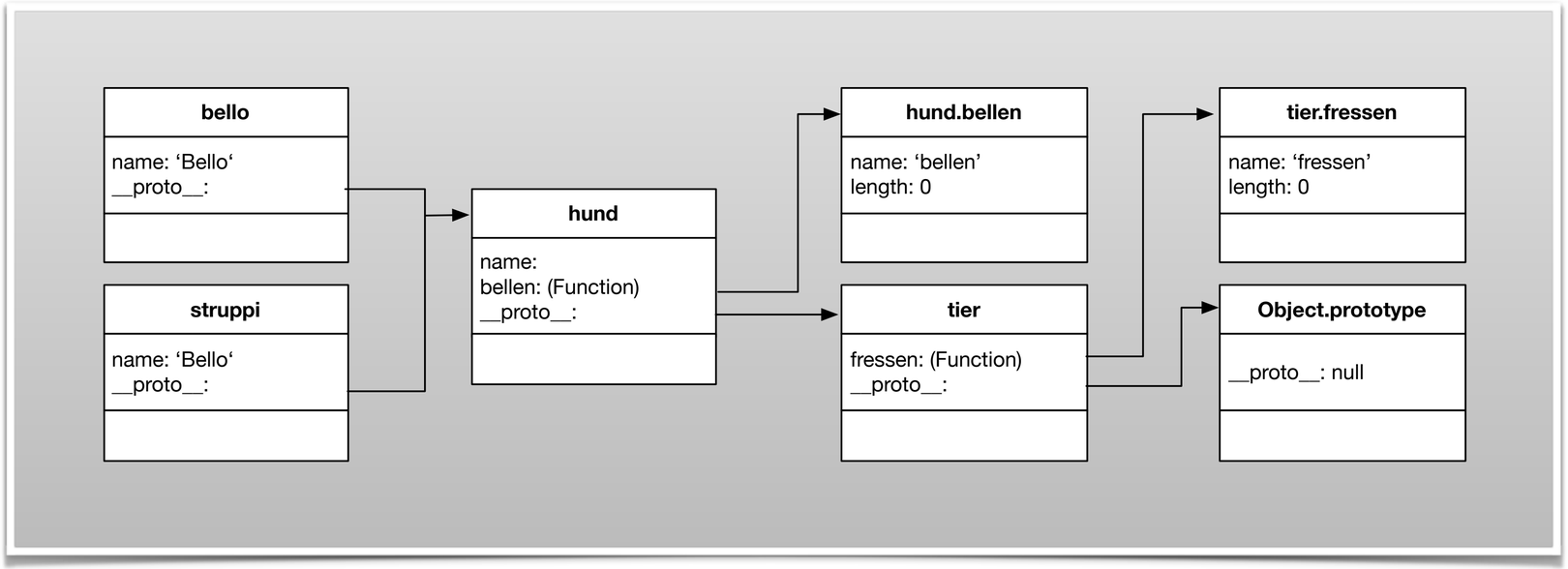


Prototypenkette (I)


```
var tier={
  name: undefined,
  fressen: function() {
    console.log('fressen');
  }
}

var hund=Object.create(tier);
hund.bellen=function() {
  console.log('Wau');
}

var bello=Object.create(hund);
bello.name='Bello';
var struppi=Object.create(hund);
struppi.name='Struppi';
bello.fressen();    // fressen
bello.bellen();     // Wau
```

Prototypenkette (Prototype Chain)

- Vererbte Eigenschaften sind nicht zwangsläufig in dem vererbten Objekt enthalten, sondern werden standardmäßig referenziert durch den Verweis auf die Eigenschaften des Prototypen
- Eigenschaften eines Prototypen können jedoch im erbenden Objekt überschrieben werden und sind dann dort eingebettet
- Analog bei Vererbung von Referenzen auf Funktionsobjekte
- Ist der Prototyp selber von einem anderen Prototyp abgeleitet, entsteht eine Verkettung von Prototypen
- Bei Zugriff auf eine bestimmte Eigenschaft eines Objektes wird ausgehend von dem Objekt die Prototypkette durchlaufen, bis die gewünschte Eigenschaft gefunden wurde



3.4 Objektorientierung
Prototypenkette (II)

```
var tier={
  fressen: function() {
    console.log('fressen');
  }
}

var hund=Object.create(tier);
hund.bellen=function() {
  console.log(this.name +': Wau');
}

var bello=Object.create(hund);
bello.name='Bello';
var struppi=Object.create(hund);
struppi.name='Struppi';
struppi.bellen=function() {
  hund.bellen();
}

struppi.bellen(); // undefined: Wau
```

```
var tier={
  fressen: function() {
    console.log('fressen');
  }
}

var hund=Object.create(tier);
hund.bellen=function() {
  console.log(this.name +': Wau');
}

var bello=Object.create(hund);
bello.name='Bello';
var struppi=Object.create(hund);
struppi.name='Struppi';
struppi.bellen=function() {
  hund.bellen.call(this);
}

struppi.bellen(); // Struppi: Wau
```

call()-Methode

- Szenario: Aufruf einer Methode des Prototypobjektes aus dem abgeleiteten Objekt heraus
- Wichtig: Berücksichtigung des `this`-Kontextes
- Linkes Beispiel: Ausgabe führt zu einem unerwünschten Resultat da bei Aufruf von `hund.bellen()` sich der `this`-Kontext auf hund bezieht, nicht auf `struppi`
- Abhilfe durch `call()`-Methode im rechten Beispiel: Übergabe des Kontextes des aufrufenden Objektes

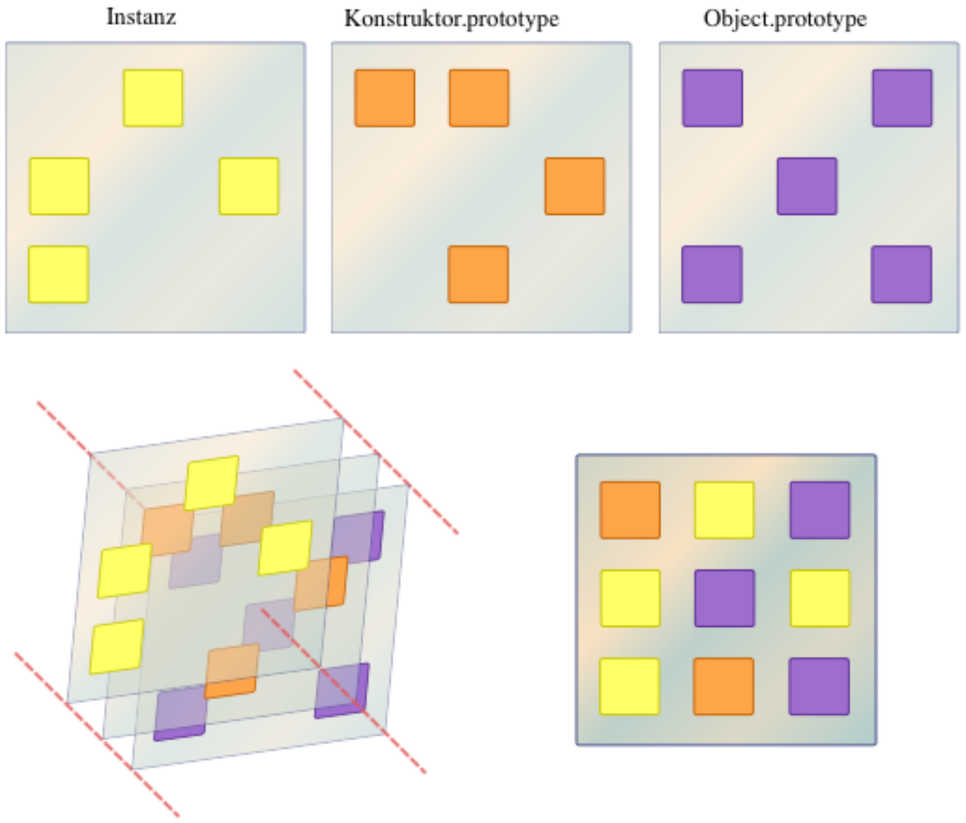
3.4 Objektorientierung
Prototypenkette (III)


```
var tier={
  fressen: function() {
    console.log('fressen');
  }
}

var hund=Object.create(tier);
hund.bellen=function() {
  console.log(this.name +': Wau');
}

var bello=Object.create(hund);
bello.name='Bello';
var struppi=Object.create(hund);
struppi.name='Struppi';
struppi.bellen=function() {
  console.log('Miau');
}

bello.bellen(): // Bello: Wau
struppi.bellen(); // Miau
```

Überschreiben von Methoden

- Einfaches Überschreiben von Methoden des Prototypen durch Hinzufügen einer Methode mit dem gleichen Namen beim erbenden Objekt
- Analog für Objekteigenschaften
- Beim Durchlaufen der Prototypenkette wird die überschreibende Methode des erbenden Objektes vor der überschriebenen Methode des Prototyps gefunden und dementsprechend ausgeführt

3.4 Objektorientierung
Prototypenkette (IV)



mit Änderungen übernommen von: http://molily.de/

Glasplatten-Analogie

- Objekte der Prototypenkette entsprechen beklebten Glasplatten
- Farbige Zettel entsprechen Eigenschaftswerten, Zettelpositionen entsprechen Eigenschaftsnamen
- Instanz verfügt über einige Zettel, Prototyp ebenso, einige Zettel kleben an den gleichen Positionen
- Funktionalität über die eine Instanz verfügt entspricht der Summe der Zettel aller hintereinander gelegter Glasplatten der Prototypenkette
- Reihenfolge der Glasplatten: unterste Objektinstanz vorne, `Object` als letztes
- Überschriebene Eigenschaften werden durch mehrere Zettel auf unterschiedlichen Glasplatten an der gleichen Position repräsentiert
- Sichtbarer Zettel an einer Position entspricht dem Eigenschaftswert der für die abgeleitete Objektinstanz gültig ist, d.h. hinter diesem Zettel befindliche andere Zettel wurden überschrieben

3.5 JavaScript im Browser

to be continued...