Webtechnologien | 2015

Kapitel 3: JavaScript

JavaScript-Überblick | JavaScript-Grundlagen | Funktionen und Funktionale Aspekte | Objektorientierte Programmierung mit JavaScript | ...

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)

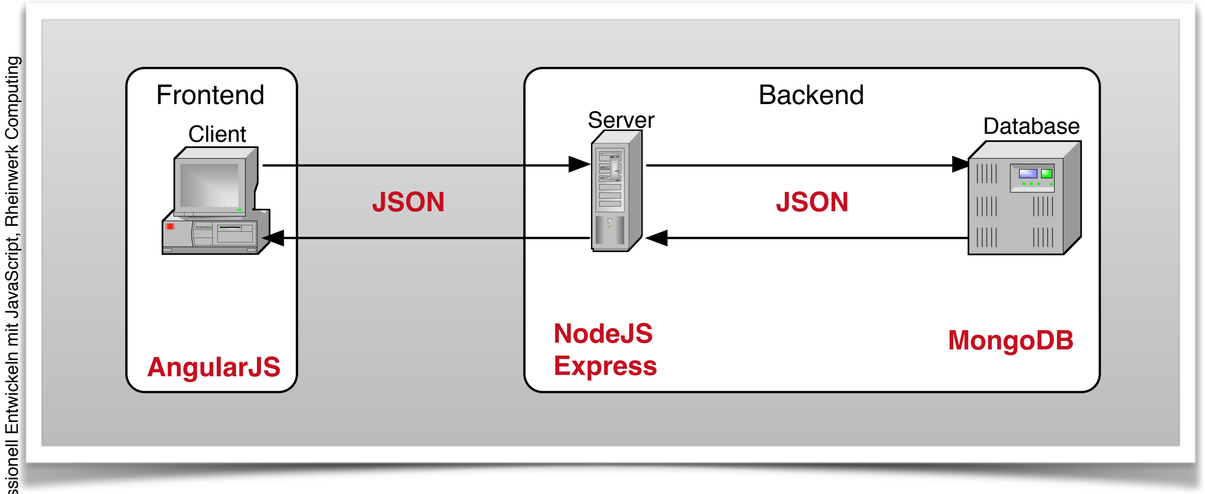
Merkmale

- 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*

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

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

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

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

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

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

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

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

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

3.2 JavaScript-Grundlagen
Datentypen und Werte - Arrays

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

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

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

```
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

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

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;
}
```

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

TELEKOM INNOVATION LABORATORIES
Service-centric NETWORKING

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`

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

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`

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

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

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

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

TELEKOM INNOVATION LABORATORIES
Service-centric NETWORKING

3.3 Funktionen und Funktionale Aspekte

to be continued