

IB3 - WEB

Ausarbeitung JavaScript

Maximilian Broszio - 1823675
30.11.2019

Inhalt

Inhalt.....	1
1 Übersicht und Geschichte von JavaScript	2
1.1 Entstehung und Standardisierung von JavaScript	2
1.2 Verwendung von JavaScript.....	2
1.3 Eigenschaften von JavaScript.....	2
1.3.1 Objektorientierung.....	3
1.3.2 Typisierung.....	3
2 JavaScript Grundstrukturen	4
2.1 Datentypen, Variablen und Datenstrukturen	4
2.1.1 Datentypen.....	4
2.1.2 Truthiness	5
2.1.3 Scope von Variablen.....	5
2.1.4 Hoisting	5
2.1.5 Arrays	6
2.2 Kontrollstrukturen	8
2.2.1 Verzweigungen.....	8
2.2.2 Schleifen.....	9
2.3 Funktionen.....	11
2.4 Objekte	11
3 Event-Handling	15
3.1 Verarbeitung eines Events	15
3.2 Asynchrone Verarbeitung (Ajax)	16
4 Zugriff auf den DOM-Baum	19
5 Debugging über Browsertools	22
6 Ausblick auf das jQuery Framework.....	25
A. Lösungen zu den Aufgaben	27
B. Literaturverzeichnis.....	29
C. Abbildungsverzeichnis	30
D. Tabellenverzeichnis.....	30
E. Codeverzeichnis	30

1 Übersicht und Geschichte von JavaScript

Dieses Kapitel ist eine Einführung in JavaScript und soll einen Überblick über diese Programmiersprache sowie deren Entstehung geben. Anschließend wird aufgezeigt wie man JavaScript in HTML-Dokumente einbinden kann und welche Variante zu bevorzugen ist. Weiterhing wird auf die Objektorientiertheit von JavaScript und die Unterschiede zu Java eingegangen.

Nach dem durcharbeiten dieses Kapitels, sollten Sie in der Lage sein, folgende Fragestellungen zu beantworten:

- Was ist ECMAScript und wie steht es mit JavaScript im Zusammenhang?
- Wie bindet man JavaScript in eine Website ein?
- Wofür benutzt man JavaScript?
- Wie ist JavaScript typisiert?
- Wie unterscheidet sich JavaScript von Java in Sachen Objektorientiertheit?

1.1 Entstehung und Standardisierung von JavaScript

JavaScript ist eine Turing-vollständige und somit vollwertige Programmiersprache die 1995 von Brendan Eich, der damals bei dem amerikanischen Softwareunternehmen „Netscape Communications“ gearbeitet hatte, entwickelt wurde. Dort wurde Ihm aufgetragen eine neue Programmiersprache innerhalb von zehn Tagen zu entwickeln, die in Netscape Navigator, dem Browser des Unternehmens, lauffähig ist. Diese neue Sprache sollte von der Syntax an Java angelehnt, aber bei weitem nicht so umfangreich sein (vgl. [Sev12, S. 7-8]). Das heutige JavaScript ist eine Implementation des ECMA-262 Standards, welcher die Skriptsprache ECMAScript spezifiziert. ECMAScript selbst wurde 1996 basierend auf den damals vorherrschenden Technologien JavaScript und JScript von Microsoft, entworfen. (vgl. [Ecm19a])

1.2 Verwendung von JavaScript

Der damalige Hauptgrund für die Entwicklung von JavaScript war es die bestehenden Möglichkeiten von HTML und CSS durch dynamisches HTML zu erweitern. Es ermöglicht durch Manipulation des DOM-Baumes HTML- und CSS-Code dynamisch zu generieren, bestehenden Code zu verändern und so auf Benutzerinteraktionen zu reagieren. Weiterhin ist JavaScript hardwareunabhängig und daher vielseitig einsetzbar. Heutzutage kann man mit entsprechenden Frameworks und Libraries, wie zum Beispiel Node.js, JavaScript auch serverseitig zum Einsatz zu bringen.

JavaScript kann über verschiedene Wege in ein HTML-Dokument eingebunden werden. Der JavaScript Code kann direkt innerhalb des HTML-Dokumentes, oder in einer externen JavaScript Datei eingefügt werden. Beides geschieht über den HTML `<script>`-Tag. Manche JavaScript Funktionen können auch direkt inline als Attribut zu HTML-Elementen hinzugefügt werden. Das direkte einfügen von JavaScript Code in das HTML-Dokument ist schlechter Stil, da es schnell zu einem unübersichtlichen Dokument führt und HTML nur der Strukturierung von Texten, Bildern und Links dient. Deshalb ist es zu bevorzugen den JavaScript Code in eine separate Datei auszulagern und diese im `<script>`-Tag mit dem `src`-Attribut zu referenzieren.

1.3 Eigenschaften von JavaScript

Dieses Unterkapitel stellt wesentliche Eigenschaften von JavaScript dar und vergleicht diese mit Java.

1.3.1 Objektorientierung

JavaScript unterscheidet sich von klassischen objektorientierten Programmiersprachen wie Java, da Klassen zwar existieren, aber die Sprache nicht auf ihnen basiert. In Java sind Klassen zwingend erforderlich, um Objekte zu erzeugen. Java Objekte bilden Instanzen der vorher definierten Klassen. In den Java Klassen werden die Eigenschaften und das Verhalten, sprich die Methoden, der Objekte fest definiert. In JavaScript sind Objekte flexibler. Ihnen können weitere Eigenschaften und Methoden auch nach der Erstellung zugewiesen, oder bereits vorhandene Eigenschaften und Methoden entfernt werden. Des Weiteren hat jedes Objekt ein sogenanntes Prototyp-Objekt, von dem es erbt. Dieses Prototyp-Objekt stellt Eigenschaften und Methoden für die davon erbenden, weiteren Objekte, zu Verfügung. Bei einem neu erstellten Objekt-Literal ist dies der Prototyp des globalen Objektes `Object`, welches zum Beispiel die `toString`-Methode bereitstellt. In Java ist das Äquivalent zu diesem Objekt die `Object`-Klasse, von der jede Java Klasse erbt und auf deren Methoden jedes Objekt Zugriff hat. Erstellt man Objekte in JavaScript über eine Konstruktor-Funktion, kann man den Prototypen der Konstruktor-Funktion modifizieren und so, Eigenschaften und Methoden für alle Objekte dieses Typs festlegen. Möchte man auf Eigenschaften oder Methoden eines Objektes zugreifen welche aber nicht für das Objekt definiert sind, wird im Prototyp danach gesucht.

1.3.2 Typisierung

JavaScript verwendet das Schema der dynamischen Typisierung. Dies bedeutet, dass die Typsicherheit von Variablen erst während der Laufzeit, anstatt dem Kompiliervorgang überprüft wird. Deswegen werden Variablen in JavaScript ohne Datentyp mit den Schlüsselwörtern `var`, `let` oder `const` deklariert. Aufgrund der dynamischen Typisierung kann man einer Variablen im späteren Programmverlauf auch Werte verschiedener Datentypen zuweisen. Einer Variable, der zuvor ein String zugewiesen war, kann später einen Wert des Typen `Number` beinhalten. Im Gegensatz zu statisch typisierten Sprachen wie Java, ermöglicht die dynamische Typisierung größere Flexibilität, kann aber auch zu Fehlern führen, die erst während dem Programmablauf erkannt werden. Bei statisch typisierten Sprachen überprüft der Compiler die Datentypen der Variablen ständig. So können Fehler frühzeitig erkannt werden.

Zusammenfassung:

Nach diesem Kapitel sollte man einen groben Überblick über die Entstehungsgeschichte von JavaScript und ECMAScript erlangt haben. Außerdem wurde darauf eingegangen wie man JavaScript im Zusammenhang mit HTML-Dokumenten verwendet und wie sich die Objektorientiertheit von JavaScript von Java unterscheidet.

Aufgaben:

Aufgabe 1.1:

Wie kann man JavaScript in ein HTML-Dokument einbinden und welche Variante ist zu bevorzugen?

Aufgabe 1.2:

Wie ist JavaScript typisiert und welche Vorteile/Nachteile hat dieses Schema?

Aufgabe 1.3:

Was sind Prototypen in JavaScript?

2 JavaScript Grundstrukturen

In diesem Kapitel werden die verschiedenen Sprachelemente von JavaScript erläutert und mit anderen Programmiersprachen verglichen.

Nach dem durcharbeiten dieses Kapitels, sollten Sie in der Lage sein, folgende Fragestellungen zu beantworten:

- Was für Datentypen existieren in JavaScript und wie läuft die Typumwandlung ab?
- Wie ist der Scope von Variablen?
- Was versteht man unter Hoisting?
- Was für Kontrollstrukturen existieren in JavaScript?
- Wie werden Funktionen definiert und was ist mit ihnen möglich?
- Wie funktionieren Vererbung und Attribut-Zugriffschutz im Vergleich zu Java?

2.1 Datentypen, Variablen und Datenstrukturen

In diesem Unterkapitel werden die verschiedenen JavaScript Datentypen erläutert und dargestellt wie Typumwandlungen gehandhabt werden. Des Weiteren wird der Scope der Variablen, deren Truthiness und Hoisting thematisiert.

2.1.1 Datentypen

Es existieren sechs primitive Datentypen in JavaScript: `String`, `Boolean`, `Number`, `Null`, `Undefined` und `Symbol`. Des Weiteren gibt es noch den abstrakten Datentyp `Object`. Von all diesen Datentypen, außer `Null` und `Undefined`, können Instanzen mit einem Konstruktor erstellt werden. Im Gegensatz zu anderen Programmiersprachen wird in JavaScript nicht zwischen `Integer`, `Float` und anderen Zahlen-Datentypen unterschieden. `Number` repräsentiert immer einen 8-Byte Wert entsprechend dem Format einer Zahl in doppelter Genauigkeit nach IEEE 754. Neben Ganzzahlen und Gleitkommazahlen kann `Number` auch symbolische Werte wie positive und negative Infinity, sowie NaN (Not a Number) annehmen (vgl. [Ecm19b]). Die Typen `Undefined` und `Null` haben jeweils nur einen Wert, `undefined` und `null`. Der Unterschied zwischen ihnen besteht darin, dass jede `var`-Variable den Wert `undefined` besitzt, bis sie initialisiert wird. Der Wert repräsentiert etwas nicht definiertes. Der Wert `null` hingegen steht für eine gewollte Absenz eines Wertes.

Der `Symbol`-Datentyp ist aus anderen Skriptsprachen wie Ruby bekannt und verhält sich dort wie in JavaScript. Eine `Symbol`-Instanz ist ähnlich wie ein `String`, allerdings ist ein `Symbol` einzigartig und somit niemals gleich. Bei der Erstellung einer `Symbol`-Instanz kann dem Konstruktor eine optionale Beschreibung für das Symbol übergeben werden.

Typumwandlungen in JavaScript werden, wenn benötigt, implizit und automatisch durchgeführt. Allerdings existieren Funktionen, um Datentypen manuell zu konvertieren. So wird auch bei den Vergleichsoperatoren zwischen einem strikten und einem Typ konvertierenden Operator unterschieden. Der `==` Operator wandelt die Operanden erst in den gleichen Typ um, bevor sie verglichen werden, während der `===` Operator den Wert und den Typ ohne vorherige Konvertierung vergleicht. Deshalb ergeben die Ausdrücke in der folgenden Abbildung unterschiedliche Ergebnisse:

Code:	Konsolenausgabe:
<pre><script> console.log("Vergleich mit == : " + ('5' == 5)); console.log("Vergleich mit === : " + ('5' === 5)); </script></pre>	<pre>Vergleich mit == : true Vergleich mit === : false</pre>

Abbildung 2.1: Vergleichsoperator Test

Der `==` Operator konvertiert den String `'5'` erst zu einer Number und vergleicht diese dann mit dem Operanden, deshalb ist das Resultat `true`. Der `===` Operator vergleicht den Wert und den Typ der Operanden. In diesem Fall sind die Typen String und Number, daher wird `false` zurückgegeben.

2.1.2 Truthiness

Operatoren und Kontrollstrukturen, die mit booleschen Werten arbeiten, wie zum Beispiel `if`-Anweisungen, können in JavaScript auch Werte, die nicht vom Typ Boolean sind, evaluieren, da eine automatische Typumwandlung zu Boolean durchgeführt wird. In JavaScript entsprechen `false`, `NaN`, `null`, `undefined`, `0`, `-0` und Leere Strings dem booleschen Wert `false`. Alle anderen Werte entsprechen dem booleschen Wert `true`.

2.1.3 Scope von Variablen

Definition 2.1:

In JavaScript bezeichnet der Scope den Bereich, in dem Ausdrücke sichtbar sind und auf sie zugegriffen werden kann.

Für Variablen, die mit dem Schlüsselwort `var` deklariert werden, unterscheidet man zwischen dem globalen und lokalen Scope. Auf Variablen, die innerhalb des globalen Scopes deklariert werden, kann von überall im Programm aus zugegriffen werden. Auf Variablen, die innerhalb einer Funktion deklariert werden, kann man nur lokal, also innerhalb der Funktion, zugreifen. Jede Funktion bildet ihren eigenen lokalen Scope, während der globale Scope alle Funktionen umschließt. In anderen Programmiersprachen wie zum Beispiel Java, bilden Kontrollstrukturen wie `if`-Anweisung oder Schleifen einen Block Scope. Die darin deklarierten Variablen sind außerhalb des Blockes nicht adressierbar, auch nicht innerhalb des Funktionseigenen lokalen Scopes. In JavaScript existiert dieser Scope nur für Variablen, die mit dem Schlüsselwort `let` oder `const` deklariert wurden. Lokale Variablen, die mit `let` deklariert werden, existieren ebenso in der funktionalen Programmiersprache LISP. Dort sind sie ebenfalls dazu gedacht Variablen innerhalb eines lokalen Scopes zu erzeugen.

2.1.4 Hoisting

In JavaScript kann auf Variablen und Funktionen zugegriffen werden bevor diese deklariert wurden. Dies funktioniert, da der JavaScript Interpreter Hoisting (engl., 'hochziehen') anwendet. Der Interpreter durchsucht die Funktion auf lokal deklarierte Variablen und 'zieht' diese unter den Funktionskopf. Ebenso 'zieht' er Funktionsdeklarationen nach oben, auch innerhalb des globalen Scopes. Existiert eine Variable, die im globalen Scope deklariert wurde und eine gleichnamige lokale Variable, wird die lokale Variable 'gehohstet' und überdeckt die globale Variable, auch wenn diese noch nicht initialisiert und somit `undefined` ist. Dieses Verhalten kann leicht zu Fehlern führen.

Code:	Konsolenausgabe:
<pre> <script> if (true) { console.log(letTest); let letTest; letTest = 4; } </script> <script> if (true) { console.log(varTest) var varTest; varTest = 4; } </script> </pre>	<pre> undefined ReferenceError: can't access lexical declaration `letTest' before initialization </pre>

Abbildung 2.2: Hoisting anhand von var und let Variablen

Wie die obige Abbildung zeigt kann auf `let` Variablen nicht vor der Deklaration zugegriffen werden. Beim Zugriff wird ein `ReferenceError` ausgegeben. Weiterhin kann man erkennen, dass derselbe Code mit einer `var` Variablen, `undefined` ausgibt. Dies liegt daran, dass die Deklaration von `varTest` unter den Funktionskopf gezogen wurde und somit der Zugriff darauf möglich ist, da `var` Variablen immer mit `undefined` initialisiert sind. Die eigentliche Initialisierung mit 4 wird nicht „gehoisted“, somit wird `undefined` ausgegeben. Anders als `var`-Variablen werden `let`- und `const`-Variablen zwar „gehoisted“, sind aber noch nicht initialisiert (vgl. [Moz19a]).

2.1.5 Arrays

In einem Array können viele verschiedene Werte gespeichert und über eine einzige Variable adressiert werden. Auf die einzelnen Werte kann man dann über einen Index oder Schlüssel zugreifen. Indexierte Arrays starten mit dem Index 0. Dieser Index erhöht sich mit jedem weiteren Datenfeld. Anstatt eines Indizes kann man auch einen String als Schlüsselwert nutzen und so den einzelnen Werten Namen geben, anstatt sie durczunummerieren. Diese Art von Arrays werden auch **assoziative Arrays** genannt und „[...] bilden in JavaScript zugleich die Basis des Datentyps `Object`“ [Bew18, S.91]. Ein gemischtes Array, das Schlüssel und Indizes für den Zugriff auf Werte nutzt ist auch möglich. Aufgrund der dynamischen Typisierung können JavaScript Arrays auch Werte mit unterschiedlichen Datentypen aufnehmen. In statisch typisierten Programmiersprachen wie Java, wird der Datentyp des Arrays bei der Deklaration festgelegt. So kann in Java ein Array des Typen `String` auch nur Strings enthalten. In JavaScript kann ein Array `Strings`, `Numbers` und Werte anderer Typen beinhalten. Arrays in JavaScript haben auch keine vorher definierte Länge, was in Java wiederum der Fall ist, sondern die Größe verändert sich dynamisch.

In JavaScript gibt es verschiedene Möglichkeiten ein Array zu erzeugen. Die erste Möglichkeit ist, das Array als Literal zu erzeugen. Die zweite Möglichkeit ist das Array über eine Konstruktor-Funktion zu erstellen. Hierfür wird das Schlüsselwort `new` verwendet. Um ein leeres Array zu erzeugen, können bei beiden Möglichkeiten die Klammern leer gelassen werden. Folgend ist die Syntax der beiden Möglichkeiten abgebildet:

```

var array = [Wert, Wert, ...];
var array = new Array( Werte );

```

Syntax Beispiel 2.1: Erstellung von Arrays

Bei der Erstellung über den Konstruktor muss beachtet werden, dass bei Übergabe eines einzigen Number-Wertes, nicht ein Array mit genau diesem Wert als Inhalt, sondern ein Array mit genau dieser Länge erzeugt wird. Dies wird in folgender Abbildung aufgezeigt:

Code:

```
<script>
var array = new Array(1, 2, 'a');
var array2 = new Array(5);
var array3 = new Array('a');

console.log("Ausgabe Länge von array: ")
console.log(array.length)

console.log("Ausgabe Länge von array2: ")
console.log(array2.length)

console.log("Ausgabe Länge von array3: ")
console.log(array3.length)

console.log("Ausgabe des Inhalts von array2")
console.log(array2)
</script>
```

Konsolen-Ausgabe:

```
Ausgabe Länge von array:
3
Ausgabe Länge von array2:
5
Ausgabe Länge von array3:
1
Ausgabe des Inhalts von array2
▶ Array(5) [ undefined, undefined, undefined, undefined, undefined ]
```

Abbildung 2.3: Beispiel der Arrayerzeugung über den Konstruktor

`array2` wird durch einen Konstruktor-Aufruf mit der Zahl 5 erzeugt. Dadurch entsteht ein Array mit fünf leeren Feldern, also Felder vom Typ `undefined`. Die anderen beiden Arrays werden ebenfalls über den Konstruktor erzeugt, allerdings werden im Falle von `array1` mehrere Werte und im Falle von `array3` ein String übergeben.

Ein assoziatives Array erzeugt man über die Objekt-Notation oder mit dem oben aufgeführten Konstruktor, wobei die Klammern leer und die Schlüssel/Wert-Paare einzeln aufgeführt werden müssen:

```
var array = {
    "name1" : Wert;
    "name2" : Wert;
    ...
}

var array = new Array();
array["name"] = Wert;
...
```

Syntax Beispiel 2.2: Erstellung von assoziativen Arrays

2.2 Kontrollstrukturen

Dieses Unterkapitel wird näher auf verschiedene in JavaScript existierende Kontrollstrukturen, wie Verzweigungen und Schleifen eingehen.

2.2.1 Verzweigungen

If-Anweisung

Mit der `if`-Anweisung können Code Segmente abgegrenzt und nur unter einer Bedingung, die durch einen booleschen Wert repräsentiert wird, ausgeführt werden. Der boolesche Wert kann das Ergebnis eines Vergleichs, eine Variable oder aber auch ein Literal, das durch eine automatische Typumwandlung zu einem booleschen Wert konvertiert wird, sein.

Die Syntax der If-Anweisung ist wie folgt festgelegt:

```
if (Bedingung) { Anweisungen } else { Anweisungen }
```

Syntax Beispiel 2.3: If-Anweisung

Die nach dem `if`-Block folgende `else`-Anweisung ist optional. Sie wird immer dann ausgeführt, wenn die Bedingung der `if`-Anweisung `false` ergibt. Die Anweisungssegmente bilden einen eigenen Block innerhalb der geschweiften Klammern und sind somit für `let` Variablen ein eigener Block Scope. Das `if`-Anweisungskonstrukt kann auch durch den ternären Operator, gekennzeichnet durch ein Fragezeichen, abgekürzt werden. Dessen Syntax sieht wie folgt aus:

```
(Bedingung) ? Ausdruck1 : Ausdruck2
```

Syntax Beispiel 2.4: Ternärer Operator

Ausdruck 1 wird ausgeführt falls die Bedingung `true` ergibt und Ausdruck 2 falls sie `false` liefert. Dieser Operator existiert genauso in anderen Programmiersprachen wie Ruby und Java. Im Sinne der Lesbarkeit des Codes, sollte man den ternären Operator für komplexer und längere Ausdrücke vermeiden, da der Code so schnell unübersichtlich werden kann.

Switch-Case-Anweisung

Die `switch-case`-Anweisung ist eine Alternative zu der `if`-Anweisung und ist vor allem bei einer Unterscheidung zwischen mehreren Fällen nützlich. Die Syntax der `switch-case`-Anweisung sieht wie folgt aus:

```
switch(Ausdruck) {  
    case Wert1: Anweisungen break;  
    case Wert2: Anweisungen break;  
    case Wert3: Anweisungen break;  
    ...  
    default:    Anweisungen  
}
```

Syntax Beispiel 2.5: Switch-Case

Der nach dem `switch` Schlüsselwort einzufügende Ausdruck, ist der, der mit den verschiedenen Werten der einzelnen `cases` (engl. „Fälle“) verglichen wird. Die Fälle werden mit dem Schlüsselwort `case` eingeleitet und bestehen aus einem Testwert, der dann mit der obigen Anweisung verglichen wird. Stimmen diese überein, wird der Anweisungsblock des `case` ausgeführt. Die einzelnen Vergleiche finden über den `===` Operator statt. Es werden also nicht nur der Wert, sondern auch die Typgleichheit überprüft. Gibt es kein `case`, welches mit dem Ausdruck übereinstimmt, wird der

optionale `default` case ausgeführt. Geschweifte Klammern nach den einzelnen `case`-Anweisungen sind nicht zwingend erforderlich. Möchte man aber einen Block Scope innerhalb des `case`, um so `let` Variablen auf das eine `case` einzuschränken, kann man die geschweiften Klammern nach dem Doppelpunkt setzen. Die `break`-Anweisung ist ebenfalls optional. Durch `break` wird nach dem Aufruf des `case` die `switch`-Anweisung beendet und die nachfolgenden `cases` ignoriert, auch wenn diese ebenfalls mit dem Ausdruck übereinstimmen. Lässt man es weg, werden alle nachfolgenden `cases` ausgeführt, egal ob der Wert mit dem Ausdruck übereinstimmt oder nicht. Dies geschieht solange bis das Programm auf eine `break`-Anweisung stößt oder die `switch`-Anweisung beendet ist. Aufgrund dieses Verhaltens handelt es sich „[...] um eine **Fall-Through-Anweisung**“ [Ste14, S. 80].

2.2.2 Schleifen

While und do-while-Schleifen

Im Schleifenkopf der `while`-Schleife befindet sich eine Bedingung, die vor dem ersten Durchlauf und auch vor allen weiteren Durchläufen überprüft wird. Nur wenn die Bedingung den booleschen Wert `true` liefert wird die Schleife durchlaufen, also läuft die Schleife solange, bis die Bedingung `false` ergibt. Der Unterschied zwischen der `while`-Schleife und der `do-while`-Schleife ist, dass bei der `do-while`-Schleife die Überprüfung der Bedingung erst nach dem ersten Durchlauf stattfindet. Daher ist es garantiert, dass die Schleife mindestens einmal durchlaufen wird, während es bei der `while`-Schleife zu gar keinem Durchlauf kommen kann. Deswegen wird die `do-while`-Schleife auch **fußgesteuert** und die `while`-Schleife **kopfgesteuert** genannt. Die Syntax der `do-while`-Schleife unterscheidet sich von der `while`-Schleife wie folgt:

```
while(Bedingung) { Anweisungen }  
do { Anweisungen } while(Bedingung);
```

Syntax Beispiel 2.6: `while`- und `do-while`-Schleife

for, for-in und for-of-Schleifen

Die `for`-Schleife ist eine weitere kopfgesteuerte Schleife, bei der die Anzahl der Durchläufe im Schleifenkopf definiert ist. Daher ist es sinnvoll diese Schleife zu benutzen, falls die Anzahl der Durchläufe, die abgearbeitet werden sollen, bekannt ist. Der Initialisierungsausdruck wird genau einmal bei Beginn der Schleife ausgeführt. Er ist dafür da eine Zählervariable oder ähnliches zu deklarieren und zu initialisieren. Für solche Zählervariablen ist es sinnvoll `let` zu verwenden und sie so Block lokal zu halten. Der Bedingungsausdruck wird auch hier mit einer Typumwandlung zu einem booleschen Wert konvertiert und führt sobald dieser `false` liefert zu einem Schleifenabbruch. Der dritte Ausdruck wird nach jedem Schleifendurchlauf ausgeführt. Mit ihm kann man die vorher deklarierte Zählervariable manipulieren, oder die Bedingung beeinflussen, um so die Schleife nach der gewünschten Anzahl an Durchläufen zu beenden.

Die `for-in`-Schleife erfüllt einen bestimmten Zweck in JavaScript. Sie ist dafür gedacht über die Eigenschaften von Objekten zu iterieren. Es wird über die Namen der Eigenschaften iteriert und nicht über die Werte. Die Iteration geschieht in zufälliger Reihenfolge, daher ist es nicht sinnvoll die `for-in`-Schleife zum Iterieren über Arrays, bei denen die Reihenfolge der Daten wichtig ist, oder ähnliche Objekte zu benutzen. Für Arrays die String-Schlüssel anstatt Indizes benutzen, ist die `for-in`-Schleife jedoch nützlich. Die Variable nimmt in jedem Schleifendurchlauf den Namen einer Eigenschaft des Objektes an. So kann im Anweisungsblock über die Variable auf die Werte der Eigenschaft zugegriffen werden.

Die `for-of`-Schleife ist der `for-in`-Schleife sehr ähnlich. Allerdings ist die `for-of`-Schleife für spezielle Objekte, wie Arrays oder Maps gedacht. Hier wird über die Werte, anstatt über die Namen der Eigenschaften iteriert. Der Variable wird in jedem Schleifendurchlauf ein Wert des Objektes zugewiesen. Auf den Wert kann dann im Anweisungsblock zugegriffen werden.

In der folgenden Abbildung wird der Unterschied zwischen der for-in- und for-of-Schleife anhand eines Beispiels verdeutlicht:

Code:	Konsolen-Ausgabe:
<pre><script> var array = ['a','b','c'] console.log("Starte for-in-Schleife: ") for (let i in array) { console.log(i) } console.log("Starte for-of-Schleife: ") for (let i of array) { console.log(i) } </script></pre>	<pre>starte for-in-Schleife: 0 1 2 starte for-of-Schleife: a b c</pre>

Abbildung 2.4: Vergleich zwischen der for-in- und for-of-Schleife anhand eines Beispiels

In diesem Beispiel wird über das vorher definierte Array einmal mit der for-in- und for-of-Schleife iteriert. Jeder Durchlauf der Schleife gibt den aktuellen Wert der innerhalb des Schleifenkopfes deklarierten Variable aus. Dabei gibt die for-in-Schleife die Namen der Eigenschaften aus, hier die Indizes des Arrays, während die for-of-Schleife die eigentlichen Werte der Eigenschaften des Arrays ausgibt.

Folgend ist die jeweilige Syntax der verschiedenen for-Schleifen dargestellt:

```
for(Initialisierung; Bedingung; Ausdruck) { Anweisungen }
for(Variable in Objekt) { Anweisungen }
for(Variable of iterierbaresObjekt) { Anweisungen }
```

Syntax Beispiel 2.7: for-Schleifen

2.3 Funktionen

Funktionen in JavaScript sind Objekte. Sie können Variablen zugewiesen und auch anderen Funktionen übergeben werden. Sie können mit dem Schlüsselwort `function` oder einem Konstruktor erzeugt werden:

```
function f( formelle Parameter ) { Anweisungen }
var f = new Function( formelle Parameter ) { Anweisungen };
```

Syntax Beispiel 2.8: Deklaration einer Funktion

Die formalen Parameter einer Funktion befinden sich im lokalen Scope und können daher nur innerhalb der Funktion referenziert werden. Wie im Unterkapitel ‚Hoisting‘ schon erwähnt werden Funktionsdeklarationen ebenfalls gehoisted. Dies gilt nicht für anonyme Funktionen. Anonyme Funktionen werden erstellt in dem man die Funktion ohne Namen deklariert. Funktionen können über ihren Namen oder über die Variable, in der sie gespeichert sind, aufgerufen werden. Außerdem hat jede Funktion eine `apply`-Methode, die ebenfalls dazu gedacht ist, die Funktion aufzurufen. Es ist auch möglich Funktionen innerhalb des Anweisungsblocks einer Funktion zu deklarieren und so, mehrere Funktionen ineinander zu verschachteln. Sind Funktionen ineinander verschachtelt, kann die innere Funktion auf den Scope der äußeren zugreifen, aber nicht umgekehrt. Funktionen mit einer

Variablen Anzahl an Parametern können über das `arguments`-Objekt jeder Funktion realisiert werden. Innerhalb dieses Objektes werden die übergebenen Parameter gespeichert, auch wenn mehr Werte übergeben werden, als in der Deklaration spezifiziert wurden. Über dieses Objekt kann dann innerhalb der Funktion iteriert werden (vgl. [Moz19c]). Alternativ kann die Funktion wie in Java mit einem Rest-Parameter deklariert werden. Dieser Parameter sieht wie folgt aus:

```
function f(a, b, ...restParameter) { Anweisungen }
```

Syntax Beispiel 2.9: Funktion mit Rest-Parameter

Der Rest-Parameter, gekennzeichnet durch drei Punkte, sammelt alle übergebenen Parameter, die keinem der vorherigen formellen Parameter zugewiesen werden konnten.

2.4 Objekte

Wie im obigen Abschnitt über Arrays schon erwähnt bilden assoziative Arrays die Basis für Objekte. Daher ist es auch möglich Objekte als Literal wie ein assoziatives Array zu erzeugen. Weiterhin kann man Objekte auch mit Hilfe einer Konstruktor Funktion nach folgendem Schema erzeugen:

```
function Objektname(Formale Parameter) {  
    Eigenschaften und Methoden deklarieren  
}  
  
var objekt = new Objektname(Aktuelle Parameter);
```

Syntax Beispiel 2.10: Erstellung eines Objektes über eine Konstruktor-Funktion

Der Konstruktor wird über das Schlüsselwort `new` aufgerufen und innerhalb der Klammern werden die erforderlichen Parameter übergeben. Auf die Eigenschaften und Methoden des Objektes kann man wie in Java über `Objektname.Attribut` zugreifen. So kann man einem Objekt auch eine neue Eigenschaft oder eine Methode hinzufügen. In JavaScript gibt es keine Möglichkeiten den Zugriff auf Attribute mit Schlüsselwörtern wie `private` oder `protected` zu begrenzen. Anstatt dessen kann man die Attribute eines Objektes in einem innerhalb des Konstruktors erzeugten Objekt „verstecken“. Darauf zugreifen kann man dann über ein weiteres in dem Konstruktor erzeugtes Objekt, welches `setter`- und `getter`-Methoden zu Verfügung stellt. Das zweite Objekt mit dem auf die „privaten“ Attribute zugegriffen wird, ist das Objekt, welches am Ende des Konstruktors zurückgegeben wird. So kann man über dieses Objekt auf die „privaten“ Attribute `getter`- und `setter`-Methoden zugreifen.

In Java wird Vererbung durch das Schlüsselwort `extends` und der Methode `super()` realisiert. Mit `extends` wird eine Unterklasse definiert, die eine Oberklasse „erweitert“. Innerhalb des Konstruktors kann man mit der `super()`-Methode den Konstruktor der Oberklasse aufrufen und diesem Parameter zur Initialisierung von Attributen übergeben. Da es in JavaScript keine „richtigen“ Klassen gibt, wird die Vererbung, über die im ersten Kapitel erwähnten Prototypen realisiert. Es ist auch möglich Konstruktor-Funktion mit der `call()`-Methode zu verketteten und so das Verhalten der `super()`-Methode in Java nachzubilden. Die `call()`-Methode wird mit einer Referenz auf das aufrufende Objekt via `this` und den verschiedenen zu übergebenden Parametern aufgerufen.

Die folgende Abbildung veranschaulicht die Vererbung und den Zugriffsschutz anhand eines Beispiels:

Code:	Konsolenausgabe:						
<pre><!DOCTYPE html> <html lang="de"> <head> <meta charset="UTF-8"> </head> <body> <script> function fahrzeug(name, preis) { var privat = {}; privat.name = name; privat.preis = preis; this.getName = function() { return privat.name; } this.getPreis = function() { return privat.preis; } } function lkw(name, preis, anzahlAnhänger) { fahrzeug.call(this, name, preis); var privat = {}; privat.anzahlAnhänger = anzahlAnhänger; this.getAnzahlAhänger = function() { return privat.anzahlAnhänger; } } var lkwObjekt = new lkw("lastwagen", 80000, 2); console.log(lkwObjekt); console.log("Direkte Zugriffe: "); console.log(lkwObjekt.anzahlAnhänger); console.log(lkwObjekt.name); console.log("Zugriff über getter: "); console.log("Name des Farhzeugs: " + lkwObjekt.getName()); console.log("Anzahl an Anhängern: " + lkwObjekt.getAnzahlAhänger()); </script> </body> </html></pre>	<pre>{-} ▶ "getAnzahlAhänger": function nger() {≡ ▶ getName: function getName() {≡ ▶ getPreis: function getPreis() {≡ ▶ <prototype>: Object { - }</pre> <table border="1"><thead><tr><th>Direkte Zugriffe:</th></tr></thead><tbody><tr><td>undefined</td></tr><tr><td>undefined</td></tr><tr><td>Zugriff über getter:</td></tr><tr><td>Name des Farhzeugs: lastwagen</td></tr><tr><td>Anzahl an Anhängern: 2</td></tr></tbody></table>	Direkte Zugriffe:	undefined	undefined	Zugriff über getter:	Name des Farhzeugs: lastwagen	Anzahl an Anhängern: 2
Direkte Zugriffe:							
undefined							
undefined							
Zugriff über getter:							
Name des Farhzeugs: lastwagen							
Anzahl an Anhängern: 2							

Abbildung 2.5: Beispiel Zugriffsschutz und Vererbung

In diesem Beispiel sind zwei Konstruktoren deklariert. Der lkw-Konstruktor übergibt die Parameter `name` und `preis` an den `fahrzeug`-Konstruktor mittels der `call()`-Methode. Somit entsteht eine Vererbungshierarchie. In beiden Konstruktoren werden die Attribute mit Hilfe eines `privat`-Objektes „versteckt“ und nur über die `getter`-Methoden verfügbar gemacht. Anhand der Konsolenausgabe kann man erkennen, dass das erstellte `lkw`-Objekt die `getName`- und `getPreis`-Methoden geerbt hat und die direkten Zugriffe auf Attribute `undefined` ausgeben, während die `getter`-Methoden das gewünschte Ergebnis liefern.

Zusammenfassung:

In diesem Kapitel wurde ein Überblick über die verschiedenen in JavaScript vorhandenen Datentypen und sprachlichen Grundelemente gegeben. Außerdem wurde die Typisierung von JavaScript und Besonderheiten wie Hoisting erläutert. Des Weiteren wurde genauer auf die verschiedenen objektorientierten Eigenschaften von JavaScript wie Vererbung und Zugriffsschutz eingegangen.

Aufgaben:

Aufgabe 2.1:

Was ist das Besondere an dem JavaScript Datentyp Number?

Aufgabe 2.2:

Was ist der Unterschied zwischen dem == und dem === Operator?

Aufgabe 2.3:

Wie unterscheiden sich Java Arrays von den JavaScript Arrays?

Aufgabe 2.4:

Warum ist es oft nicht sinnvoll mit einer `for-in`-Schleife über Arrays zu iterieren?

Aufgabe 2.5:

Wie ist Vererbung in JavaScript realisiert?

Aufgabe 2.6:

Wie lassen sich Attribute vor dem Zugriff von außen schützen?

3 Event-Handling

In diesem Kapitel wird das Event-Handling in Java erläutert. Es wird vor allem auf den Ablauf und die Verarbeitung eines Events durch EventHandler eingegangen. Weiterhin wird ein kurzer Überblick über das Ajax Konzept gegeben und dessen funktionsweise dargestellt.

Nachdem dieses Kapitel durchgearbeitet wurde sollten Sie in der Lage sein,

- EventHandler an HTML-Elemente anzubringen,
- die Phasen des Event-Ablaufs zu nennen,
- das Prinzip des Bubbling und des Capturing erläutern zu können,
- die Funktionsweise des Ajax Konzepts verstanden haben sowie
- die Vor- und Nachteile von Ajax wiedergeben zu können.

In JavaScript existieren verschiedene Events (engl. Ereignisse), die durch unterschiedliche Geschehnisse ausgelöst werden können. Ein Beispiel für eines dieser Events wäre das `click`-Event, das durch den Anwender bei einem Mausklick auf ein HTML-Element ausgelöst wird. Diese Events können mit einem sogenannten Event-Handler abgefangen und verarbeitet werden. Event-Handler können direkt im HTML-Dokument mit Attributen wie `onload` oder `onclick` eingefügt werden, oder in einem Script über die `addEventListener`-Methode. Die zweite Variante, bei der ein externes Skript den Elementen automatisch die Event-Handler zuweist, ist stets zu bevorzugen, da so HTML und JavaScript getrennt bleibt [Ste14, S. 266]. Jedes HTML-Element kann mehrere Event-Handler besitzen.

3.1 Verarbeitung eines Events

Beim Auslösen eines Events auf einem Element wird standardmäßig nicht nur der EventHandler dieses Elements ausgelöst, sondern auch die entsprechenden EventHandler der Eltern-Elemente. Dies liegt daran, dass das Event-Handling drei verschiedene Phasen durchläuft. Die erste Phase ist die Event-Capturing-Phase. Hier „wandert“ das Event durch die Eltern-Kind-Elementstruktur im DOM-Baum. Es startet im obersten Element und wandert dann bis hin zum tiefsten Kind-Element, welches das ursprüngliche Event ausgelöst hat. Dieses Element wird das `target`-Element genannt. Beim Hinzufügen von Event-Handlern durch die `addEventListener`-Methode, kann optional ein boolescher Wert übergeben werden, der das Verhalten während der Event-Capturing-Phase bestimmt. Standardmäßig wird hier ein `false` angenommen, was dazu führt das während der Event-Capturing-Phase nichts Besonderes passiert. Setzt man den Wert jedoch auf `true`, wird der Event-Handler in der Capturing Phase registriert und reagiert auf Events die auf einem Element tiefer im DOM-baum ausgelöst wurden frühzeitig. Ist das Event am `target`-Element angekommen, beginnt die zweite, sogenannte „Target-Phase“. Hier wird der Event-Handler des `target`-Elements ausgeführt. Anschließend beginnt die dritte Phase, welche die Event-Bubbling-Phase ist. Diese Phase ist das Gegenteil der Event-Capturing-Phase. Das Event „wandert“ den DOM-baum wieder nach oben, bis zum obersten Element. Hier liegt hier der Ursprung des am Anfang des Unterkapitels erwähnten Verhaltens. Alle Event-Handler, die nicht in der Capturing-Phase registriert wurden, werden jetzt ausgehend vom `target`-Element, bis hin zum obersten Element ausgelöst. Dieses Verhalten wird durch das folgende Beispiel veranschaulicht:

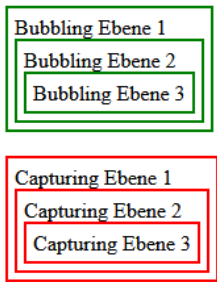
<p>Code:</p> <pre> <!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <style> .bubble { border: 2px solid green; margin: 5px; padding: 5px; display: table-cell; } .capture { border: 2px solid red; margin: 5px; padding: 5px; display: table-cell; } </style> </head> <body> <div id="bubble-ebene1" class="bubble">Bubbling Ebene 1 <div id="bubble-ebene2" class="bubble">Bubbling Ebene 2 <div id="bubble-ebene3" class="bubble">Bubbling Ebene 3</div> </div>
 <div id="capture-ebene1" class="capture">Capturing Ebene 1 <div id="capture-ebene2" class="capture">Capturing Ebene 2 <div id="capture-ebene3" class="capture">Capturing Ebene 3</div> </div> </div> </body> <script> var bubbleElements = document.getElementsByClassName("bubble"); var captureElements = document.getElementsByClassName("capture"); for(let element of bubbleElements) { element.addEventListener("click", function() { console.log("Bubbling: " + this.id + " wurde ausgelöst"); }); } for(let element of captureElements) { element.addEventListener("click", function() { console.log("Capturing: " + this.id + " wurde ausgelöst"); }, true); } </script> </html> </pre>	<p>Website:</p>  <p>Konsolen-Ausgabe nach Klick auf das jeweilige Ebene 3 <div> Element:</p> <pre> Bubbling: bubble-ebene3 wurde ausgelöst Bubbling: bubble-ebene2 wurde ausgelöst Bubbling: bubble-ebene1 wurde ausgelöst Capturing: capture-ebene1 wurde ausgelöst Capturing: capture-ebene2 wurde ausgelöst Capturing: capture-ebene3 wurde ausgelöst </pre>
---	--

Abbildung 3.1: Vergleich Bubbling und Capturing

In diesem Beispiel wurden die jeweiligen geschachtelten `<div>`-Elemente mit einem Click-Event-Handler versehen. Die grün-umrahmten `<div>`-Elemente haben einen Event-Handler der „bubbert“ und die roten einen der „captured“. Gelb markiert ist der boolesche Wert in der zweiten for-of-Schleife, der das Capturing aktiviert. Anhand der Konsolen-Ausgabe kann man erkennen wie sich die Reihenfolge der Verarbeitung des Events bei einem Klick auf die jeweiligen „Ebene 3“ `target`-Elemente unterscheidet. Bei einem Klick auf „Bubbling Ebene 3“ wird zuerst dessen Event-Handler und dann die der übergeordneten Elemente ausgelöst. Der Klick auf „Capturing Ebene 3“ löst erst die Event-Handler der übergeordneten Elemente und dann den des `target`-Elements aus.

3.2 Asynchrone Verarbeitung (Ajax)

Ajax ist ein Akronym und steht für „Asynchronous JavaScript and XML“. Es ermöglicht das nebenläufige Senden von HTTP-Requests an den Server, sodass die aktuelle HTML-Seite nicht verlassen werden muss und so passive Wartezeit für den Benutzer vermieden werden kann. Ein Beispiel für eine Verwendung von Ajax ist die Überprüfung der Verfügbarkeit eines Benutzernamens bei der Registration auf verschiedenen Webseiten. Während der Benutzername eingegeben wird, kontrolliert ein Skript auf dem Server ob der Name bereits vergeben ist und sendet das Ergebnis zurück an die Anwendung, welche das Ergebnis dann über die Manipulation des DOM-Baumes

anzeigt. Für den Austausch der Daten wird ein Kommunikationsobjekt benötigt. In den heutigen Browsern ist die `XMLHttpRequest` Schnittstelle dafür verantwortlich. Um diese zu benutzen muss ein Objekt davon mit einem Konstruktor erzeugt werden. Dieses Objekt stellt die für die asynchrone Kommunikation benötigten Methoden und Attribute zu Verfügung. Mit den Methoden `open()` und `send()` kann dann ein HTTP-Request gestellt werden. Mit der Methode `open()` wird angegeben welche URL geöffnet und welche HTTP-Methode verwendet werden soll. Außerdem kann man optional angeben, ob die Anfrage asynchron geschehen soll und einen Benutzernamen und Passwort für eine HTTP-Authentifizierung hinterlegen (vgl. [Jäg08, S. 189]). Die `send()`-Methode übermittelt, die zuvor spezifiziert Anfrage. Verwendet man die HTTP-POST-Methode müssen die zu versendenden Daten der `send()`-Methode als Parameter übergeben werden. Der HTTP-Request wird vom Server verarbeitet und das Ergebnis per HTTP-Response zurückgeschickt.

Um auf Zustandswechsel des Request-Ablaufs zu reagieren wird eine Call-Back-Funktion benötigt. Das `XMLHttpRequest`-Objekt hat ein Attribut namens `onreadystatechange` welches einen Event-Handler beschreibt, der immer dann aufgerufen wird, wenn das `readyState`-Attribut sich verändert. Dieses Attribut kann 5 verschiedene Werte annehmen:

Wert	Bedeutung
0	Die <code>open</code> -Methode wurde nicht aufgerufen
1	Die <code>open</code> -Methode wurde aufgerufen, <code>send</code> -Methode noch nicht
2	Die <code>send</code> -Methode wurde aufgerufen
3	Daten wurden teilweise empfangen
4	Daten wurden vollständig empfangen

Tabelle 3.1: Werte des `readyState`-Attributs [Jäg08, S.190]

Da Funktionen in JavaScript Objekte sind, kann man die Call-Back-Funktion, oder eine anonyme Funktion dem `onreadystatechange`-Attribut zuweisen und somit auf die Änderungen des Zustands reagieren. In der Call-Back-Funktion sollte überprüft werden ob das `readyState`-Attribut den Wert **4** erreicht hat und somit alle Daten vorhanden sind. Ebenfalls sollte überprüft werden, dass der HTTP-Request ohne Fehler abgelaufen ist. Dazu kann das `status`-Attribut des `XMLHttpRequest`-Objektes auf die gängigen HTTP-Response Statuscodes geprüft werden. Der Antworttext des HTTP-Request wird im `responseText`-Attribut gespeichert und kann darüber ausgegeben, oder anderweitig verwendet werden.

Neben den Vorteilen von Ajax für den Benutzer, wie der Vermeidung von Wartezeiten und ständigem neu laden der Website, gibt es jedoch auch einige Nachteile. Beispielsweise gibt es Probleme mit der Browsernavigation, da sich die einzelnen Zustände nicht als Lesezeichen speichern lassen und die Vorwärts- und Rückwärts-Buttons nicht richtig funktionieren, bzw. sich nur über komplizierte Umwege realisieren lassen.

Zusammenfassung:

Dieses Kapitel erläuterte verschiedenen Arten Event-Handler an HTML-Elemente anzubringen und wie diese Events bearbeiten. Weiterhin wurden die einzelnen Phasen des Event-Ablaufs erläutert und deren Auswirkungen betrachtet. Außerdem wurde ein kurzer Überblick über Ajax verschafft und dessen Funktionsweise erläutert.

Aufgaben:

Aufgabe 3.1:

Warum sollte man Event-Handler nicht direkt im HTML-Dokument an die Elemente anbringen?

Aufgabe 3.2:

Nennen Sie die drei Phasen des Event-Ablaufs.

Aufgabe 3.3:

Erklären Sie den Unterschied zwischen „Bubbling“ und „Capturing“.

Aufgabe 3.4:

Was versteht man unter Ajax und welche Vor- und Nachteile hat dieses Konzept?

4 Zugriff auf den DOM-Baum

Dieses Kapitel beschäftigt sich mit dem Zugriff und der Manipulation von HTML-Elementen über den DOM-Baum. Dabei wird auf die Struktur des DOM-Baumes eingegangen und einzelne Methoden für die Manipulation des Baumes erläutert.

Nach durcharbeiten dieses Kapitels sollten Sie in der Lage sein,

- die Struktur des DOM-Baumes darzustellen,
- die verschiedenen Arten des Zugriffs auf HTML-Elemente und deren Attribute zu erklären sowie
- verschiedene Methoden zum Erstellen und Entfernen von HTML-Elementen nennen zu können.

Während dem Laden einer Website erstellt der Browser ein Modell der Website in Baumstruktur. Deshalb ist es sinnvoll ein Skript, das den DOM-Baum manipuliert erst am Ende des HTML-Bodys einzufügen, da man so sichergehen kann das der DOM-Baum schon vollständig geladen ist. Alternativ kann man den Teil des Skripts der den DOM-Baum manipuliert in einem DOMContentLoaded-EventHandler verpacken. „Das DOMContentLoaded-Event wird ausgelöst, wenn das initiale HTML-Dokument vollständig geladen und geparkt ist. Es wird dabei nicht auf Stylesheets, Bilder und Frames gewartet“ [Moz19b]. Der DOM-Baum ist eine einheitliche Schnittstelle die Plattform und Programmiersprachen unabhängig ist. Die Baumstruktur stellt das HTML-Dokument und all seine Elemente dar. Jedes HTML-Element wird als Element-Node (engl. Element-Knoten) innerhalb des Baumes dargestellt. Neben den Element-Nodes gibt es noch Attribut-Nodes und Text-Nodes, welche die zu den Element-Nodes zugehörigen Attribute und Texte darstellen. Die drei soeben genannten Node-Typen und die Document-Node stellen die vier wichtigsten Node-Typen dar. Ein weiterer Node-Typ ist zum Beispiel ein Kommentar. Die Document-Node repräsentiert die komplette HTML-Seite und über diese Node werden alle weiteren Nodes adressiert. Es ist sozusagen der Startpunkt bei jedem Zugriff auf weitere Element-Nodes. Jede Node des Baumes wird als Objekt repräsentiert. Jedes Node-Objekt hat verschiedene Methoden und auch Attribute die Informationen über sich selbst, sowie die Beziehungen zu den benachbarten Knoten beinhalten. Die Attribute und Texte sind jeweils eine „child-Node“, also ein Unterknoten der Element-Node. Den DOM-Baum und seine Nodes kann man mit Hilfe von JavaScript dynamisch manipulieren. Man kann Elemente hinzufügen, entfernen oder verändern. Dafür werden verschiedene Methoden bereitgestellt.

Es gibt verschiedene Wege eine Element-Node zu referenzieren. Ein einzelnes Element kann man über dessen ID oder das name-Attribut ansprechen. Allerdings sollte die Referenzierung über das name-Attribut vermieden werden, da das name-Attribut im Gegensatz zu der ID nicht eindeutig ist und auch je nach HTML-Tag verschiedene Bedeutungen haben kann. Mehrere Elemente können über eine gemeinsame Klasse oder dem HTML-Tag referenziert werden. Bei den Referenzierungen über die Klasse, den Tag und dem name-Attribut wird ein NodeList-Objekt zurückgegeben, welches eine Sammlung an einzelnen Nodes repräsentiert, während über die ID eine Referenz auf das einzelne Node-Objekt zurückgegeben wird. Da wie oben beschrieben all diese Methodenaufrufe über das Document-Objekt getätigt werden, sehen sie sehr ähnlich aus:

```
var element = document.getElementById("ID");
var elements = document.getElementsByName("name");
var elements = document.getElementsByClassName("Class");
var elements = document.getElementsByTagName("Tag-Name");
```

Syntax Beispiel 4.1: Selektion von HTML-Elementen mittels DOM-Methoden

Die referenzierten Element-Objekte kann man dann wiederum über verschiedene Methoden oder deren Attribute modifizieren. Die Attribute der Element-Objekte haben `getter` und `setter`, können aber auch direkt wie bei anderen JavaScript Objekten angesprochen werden. Die Namen der Attribute entsprechen den HTML-namen. Nur das `class`-Attribut heißt `className`. Der Text einer Element-Node steht in dem Attribut `nodeValue`.

Das folgende Beispiel demonstriert einen Zugriff auf ein HTML-Element und die Manipulation der Attribute des Elementes.

Code:	Ergebnis:
<pre><!DOCTYPE html> <html lang="de"> <head> <meta charset="UTF-8"> </head> <body> <h1>Dies ist eine Testseite!</h1> <script> var element = document.getElementsByTagName("H1")[0]; element.style.color = "blue"; element.innerHTML = "Überschrift geändert!"; </script> </body> </html></pre>	<p>Überschrift geändert!</p>

Abbildung 4.1: Beispiel DOM-Baum Manipulation

Im obigen Beispiel wird ein `<h1>`-HTML-Heading-Element über dessen tag-Name einer Variablen zugewiesen. Da die `getElementsByTagName`-Methode eine Collection zurückgibt, muss das einzelne Node-Objekt mit einem Index selektiert werden. Anschließend wird das `style`-Attribut des Elements modifiziert, um so die Farbe der Überschrift zu ändern. Wichtig hierbei ist, dass es sich um das inline style-Attribut handelt und nicht das Stylesheet. Außerdem wird über das `innerHTML` Attribut der Text der Überschrift geändert. Für die Änderung des Textes gibt es mehrere Möglichkeiten. Da der Text in einer child-Node des Elementes steht, hätte man auch über das `nodeValue`-Attribut der child-Node den Text ändern können. Der Unterschied zwischen den beiden Methoden ist, dass `innerHTML` im Gegensatz zu `nodeValue` den Text als HTML parst.

Über das `document`-Objekt können auch neue Elemente dem Baum hinzugefügt oder entfernt werden. Für das Entfernen eines Elementes existiert die `removeChild(element)`-Methode. Mit dieser Methode kann man eine child-Node über dessen `parentNode` entfernen. Zum Erstellen von Elementen gibt es die `createElement(element)`-Methode. Das erstellte Element muss anschließend noch mit der `appendChild(element)`-Methode einer anderen Node hinzugefügt werden.

Zusammenfassung:

In diesem Kapitel ging es hauptsächlich um die Struktur des DOM-Baumes und der einzelnen node-Objekte. Ebenso wurden verschiedene Möglichkeiten für den Zugriff auf und die Manipulation von HTML-Elementen innerhalb des DOM-Baumes vorgestellt.

Aufgaben:

Aufgabe 4.1:

Was sind die vier wichtigsten node-Typen des DOM-Baumes?

Aufgabe 4.2:

Warum sollte man für den Zugriff auf ein einzelnes HTML-Element den Weg über dessen ID, anstatt dem Weg über das name-Attribut bevorzugen?

Aufgabe 4.3:

Was ist der Unterschied zwischen einer Änderung des Textes eines HTML-Elements über das `innerHTML`-Attribut und des `nodeValue`-Attributs des Text-Kind-Knotens?

5 Debugging über Browsertools

In den verschiedenen verfügbaren Browsern gibt es unterschiedliche Tools, um JavaScript und andere webbasierte Sprachen zu debuggen. Dieses Kapitel beschäftigt sich mit der Konsole und anderen Debugging-Tools des Firefox-Browsers.

Nach durcharbeiten dieses Kapitels sollten Sie in der Lage sein Ihren Code mit folgenden Methoden zu debuggen:

- Ausgaben auf der Konsole tätigen und damit Objekte analysieren,
- mit dem Debugger Breakpoints setzen und schrittweise den Programmablauf nachzuverfolgen sowie
- die verschiedenen Scopes und die dazugehörigen Variablen anzuzeigen.

Die meisten Firefox Entwicklertools können über das „Hamburger-Menü“ und dessen Web-Entwickler Tab, oder die F12 Taste aufgerufen werden, wobei es im Hamburgermenü zusätzliche Tools für Webentwickler gibt, die über die F12 Taste nicht erreichbar sind. Zwei der wichtigsten Tools für das Debuggen von JavaScript sind die Konsole und der Debugger. Über die Konsole werden Fehlermeldungen oder andere im Quellcode spezifizierte Ausgaben angezeigt. Eine Methode für die Konsolenausgabe, die auch in mehreren der Code-Beispiele in vorherigen Kapiteln verwendet wurde, ist die `console.log()`-Methode. Gibt man Objekte mit dieser Methode aus, kann man in der Konsole die Struktur des Objektes genauer untersuchen, da neben den Attributen auch der dazugehörige Prototyp angezeigt wird. Folgende Abbildung zeigt den Aufbau und die Funktion der Konsole anhand eines Beispiels:



Abbildung 5.1: Ausgabe eines Objektes auf der Konsole

Das in diesem Beispiel mit einem Konstruktor erzeugte `fahrzeug`-Objekt wird auf der Konsole ausgegeben. Die erzeugte Ausgabe ist im rot markierten Abschnitt (1) der Abbildung zu erkennen. Das Objekt mit seinen Attributen und dessen Prototyp wird ausgegeben. Die Details der `fahren()`-Funktion und des Prototypen kann bei Bedarf durch ausklappen des entsprechenden Reiters ebenfalls eingesehen werden. In Abschnitt (2) der Abbildung kann man einige Optionen zur Filterung der Konsolenausgabe sehen. So kann man sich bei Bedarf nur Fehler, Warnungen oder andere Ausgaben anzeigen lassen.

Der Debugger funktioniert ähnlich wie in anderen Programmiersprachen. Er verschafft einen guten Überblick über den Aufbau des Scripts und unterstützt so bei der Fehlersuche. Weiterhin ist es möglich „Breakpoints“ im Quellcode zu setzen und so nach erneutem ausführen des Skripts die Ausführung des Codes an der gewählten Stelle unterbrechen. Dann kann man schrittweise die Ausführung fortsetzen und das Verhalten beobachten. Währenddessen erlaubt der Debugger eine Einsicht in die aktuelle Wertebelegung der Variablen innerhalb der vorhandenen Scopes. Anhand der folgenden Abbildung wird der Debugger näher erläutert:

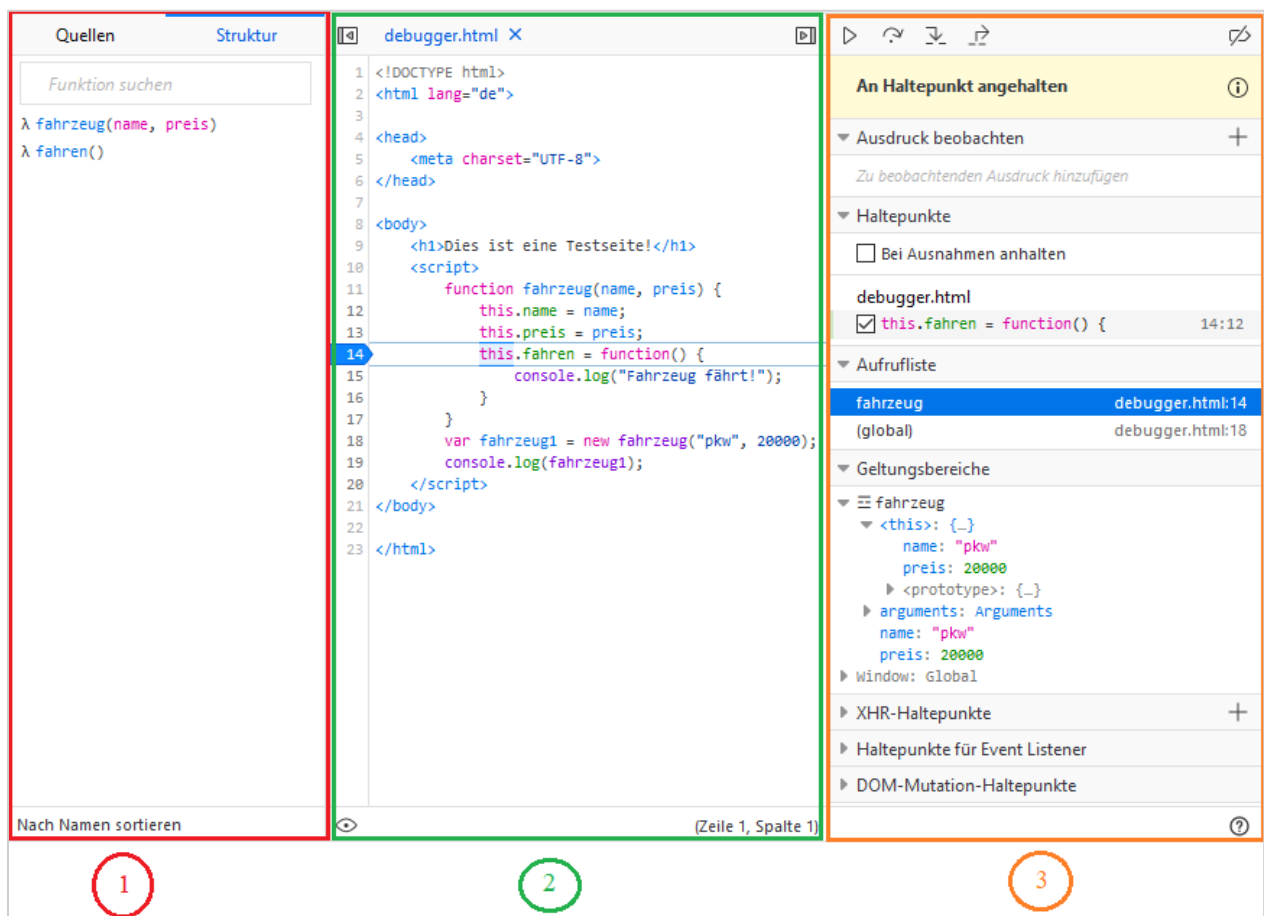


Abbildung 5.2: Funktion und Struktur des Firefox Debuggers

In Abschnitt (1) der Abbildung kann zwischen dem Struktur-Tab, in dem man zu der Position der verschiedenen Funktionen innerhalb des Quellcodes springen kann, und dem Quellen-Tab gewechselt werden. Im Quellen-Tab kann zwischen verschiedenen Dateien, die im Debugger behandelt werden sollen, gewechselt werden. Innerhalb von Abschnitt (2) wird der eigentliche Quellcode angezeigt. Hier ist es möglich Breakpoints durch einen Klick auf die Zeilennummerierung zu setzen. In diesem Beispiel wurde ein Breakpoint in Zeile 14, vor der Deklaration der `fahren`-Funktion gesetzt. Im obersten Teil von Abschnitt (3) kann bei gesetztem Breakpoint der weitere Verlauf des Programms gesteuert werden. Hier kann man das Programm normal weiterlaufen lassen, einen Schritt überspringen oder in einen Ausdruck hinein bzw. heraus springen. Innerhalb des Reiters

„Geltungsbereiche“, werden die einzelnen Scopes und die darin existierenden Variablen mit ihren Werten angezeigt.

Im Web-Entwickler Tab des „Hamburger-Menüs“ existiert auch eine JavaScript Entwicklungsumgebung, in der man JavaScript Code schreiben und direkt ausführen kann. Diese Entwicklungsumgebung ist allerdings seit der Firefox Version 70 veraltet und soll in Zukunft entfernt werden [vgl. Moz19d].

Zusammenfassung:

In diesem Kapitel wurden die Firefox Konsole und der Debugger vorgestellt und aufgezeigt wie man sich mit diesen Tools die Fehlersuche erleichtern kann.

Aufgaben:

Aufgabe 5.1:

Wie kann auf die Konsole und den Debugger zugegriffen werden?

Aufgabe 5.2:

Wie lautet eine Methode, um Nachrichten auf der Konsole auszugeben?

Aufgabe 5.3:

Was sind Breakpoints?

Aufgabe 5.4:

Welche Vorteile bietet der Debugger?

6 Ausblick auf das jQuery Framework

Da das Arbeiten mit dem DOM mühselig sein kann gibt es jQuery, eine JavaScript Bibliothek, welche neue Funktionen für verschiedene Anwendungsbereiche zur Verfügung stellt und so die Arbeit, insbesondere mit dem DOM erleichtert. jQuery kann direkt im `<script>`-Element in jedes HTML-Dokument eingebunden werden. Dazu fügt man im `src`-Attribut des Tags eine URI zu einer Onlinequelle von jQuery ein. Hat man die jQuery Bibliothek heruntergeladen, kann man diese auch lokal einbinden. Dazu muss der lokale Pfad zu der Bibliotheks-Datei angegeben werden. In vorherigen Kapiteln wurde auf die Problematik des Ladens von Skripten bevor der DOM-Baum vollständig aufgebaut ist hingewiesen. Hierfür bietet jQuery auch eine komfortablere Lösung an, als zum Beispiel einen `DOMContentLoaded`-EventHandler zu benutzen. In jQuery gibt es eine Funktion, die erst nach dem vollständigen Laden des DOM-Baumes aufgerufen wird. Diese Funktion sieht folgendermaßen aus:

```
$ (document).ready(function() { Anweisungen });  
$(function() { Anweisungen });
```

Syntax Beispiel 6.1: Ausführen von Anweisungen nach dem Laden des DOM-Baumes

Die zweite Variante ist eine abgekürzte Version der oberen Funktionsdeklaration.

Die `$(„Element“)`-Funktion erlaubt es DOM-Elemente über eine CSS ähnliche Syntax zu selektieren. Anstatt jedes Element mit `getElementById` oder ähnlichen Funktionen zu referenzieren wird die `$`-Funktion verwendet. „Die Funktion `$()` steht in jQuery für eine Kurzschreibweise, um ein Element der Webseite zu referenzieren“ [Ste18, S.31]. Mit dieser Funktion können Elemente zum Beispiel über ihre ID oder über Gruppenselektoren referenziert werden. Dies würde folgendermaßen aussehen:

```
var element = $("#elementID");  
var element = $("#id, .klasse");
```

Syntax Beispiel 6.2: jQuery Selektoren

Die `$`-Funktion gibt immer ein arrayartiges Objekt zurück, während die DOM-Methode `getElementById()` beispielsweise ein einzelnes Node-Objekt zurückgibt. Das Node-Objekt kann aber mittels eckiger Klammern und einem Index oder der `get(„Index“)`-Methode ausgelesen werden. jQuery bietet auch `getter`- und `setter`-Methoden, um auf Attribute zuzugreifen und diese zu modifizieren. Weiterhin bietet jQuery die `css()`-Methode an, die das manipulieren des Styles von HTML-Elementen vereinfacht. Mit dieser Methode kann über einen String-Parameter auf eine CSS-Eigenschaft zugegriffen werden und ein neuer Wert übergeben werden. In dem Kapitel ‚Zugriff auf den DOM-Baum‘ wurde die Manipulation des Styles und des Inhalts eines HTML-Elements mittels DOM-Methoden vorgestellt. In der folgenden Abbildung wird die gleiche Manipulation mit Hilfe von jQuery gezeigt:

<p>Code:</p> <pre> <!DOCTYPE html> <html lang="de"> <head> <meta charset="UTF-8"> <script src="https://ajax.googleapis.com /ajax/libs/jquery/3.4.1/jquery.min.js"></script> </head> <body> <h1>Dies ist eine Testseite!</h1> <script> \$(document).ready(function() { \$("h1").css("color", "blue"); \$("h1").html("Überschrift geändert!"); }); </script> </body> </html> </pre>	<p>Ergebnis:</p> <p>Überschrift geändert!</p>
---	---

Abbildung 6.1: DOM-Baum Manipulation mit jQuery

Innerhalb der `ready()`-Funktion wird das `<h1>`-Element mit der `$`-Funktion selektiert und anschließend mittels der `css()`- und `html()`-Funktion manipuliert. Die `html()`-Funktion ist das jQuery Äquivalent zu der DOM `innerHTML`-Methode (vgl. [Ste18, S 151]).

jQuery bietet auch Methoden für das Event-Handling oder um Effekte und Animationen zu erzeugen. Informationen zu diesen und weiteren Themen sind in dem im Literaturverzeichnis angegebenen Buch „jQuery - Das universelle JavaScript-Framework für das interaktive Web und mobile Apps“ von Ralph Steyer vorzufinden.

Zusammenfassung

Dieses Kapitel gab einen kleinen Ausblick auf das jQuery Framework und dessen Vorteilen gegenüber der Arbeit mit den DOM-Methoden. Es wurde auf die Element-Selektion und die Manipulation dieser Elemente eingegangen.

Aufgaben

Aufgabe 6.1:

Wie wird das jQuery Framework in ein Skript eingebunden?

Aufgabe 6.2:

Wie werden HTML-Elemente mit jQuery selektiert?

Aufgabe 6.3:

Wie kann man in jQuery sicherstellen, dass das Skript erst nach dem Laden des DOM-Baumes ausgeführt wird?

A. Lösungen zu den Aufgaben

1.1) Man kann JavaScript Code direkt im HTML-Dokument einbringen oder als externe Datei die mit dem `src`-Attribut innerhalb des `<script>`-Tags verlinkt wird. Letzteres ist zu bevorzugen, da so HTML und JavaScript strikt getrennt bleiben.

1.2) JavaScript ist dynamisch typisiert, was höhere Flexibilität im Gegenzug für höhere Fehleranfälligkeit tauscht, da die Typen der Variablen nicht vor dem Programmablauf durch den Compiler überprüft wird.

1.3) Prototypen sind eine Art Blaupause für Objekte. Jedes Objekt hat ein Prototyp, von dem es erbt und somit dessen Attribute und Methoden übernimmt. Auf die Prototypen kann zugegriffen werden, um diese und die davon erbbenden Objekte zu verändern.

2.1) In JavaScript gibt es keine Unterscheidung zwischen `int`, `short`, `float` oder `double`. Der Datentyp `Number` fasst all diese Elemente zusammen.

2.2) Der `==` Operator führt eine Typkonversion vor dem Vergleich durch, während der `===` Operator auf Wert- und Typ-Gleichheit prüft.

2.3) JavaScript Arrays sind dynamischer. Sie haben keine statische Länge und können als assoziative Arrays mit String-Indizes benutzt werden.

2.4) Die `for-in`-Schleife iteriert über die Namen der Eigenschaften eines Objektes. In Falle von Arrays sind dies die Indizes. Weiterhin ist bei `for-in`-Schleifen keine bestimmte Reihenfolge garantiert.

2.5) Vererbt wird über Prototypen oder durch den Aufruf der `call()`-Methode innerhalb des Konstruktors des untergeordneten Objektes.

2.6) Attribute können in einem zusätzlichen, innerhalb des Konstruktors deklarierten Objekt „versteckt“ werden. Der Zugriff muss dann über `getter`- und `setter`-Methoden erfolgen.

3.1) Genau wie bei der generellen Einbindung von Skripten in HTML-Dokumente soll das Ziel einer Trennung von HTML und JavaScript gewahrt bleiben.

3.2) Capturing Phase -> target-Phase -> Bubbling-Phase

3.3) Capturing arbeitet sich vom äußersten Element bis zum innersten Element durch, während Bubbling genau andersherum arbeitet.

3.4) Ajax ist ein Konzept für einen asynchronen Austausch von Daten zwischen dem Server und einer Anwendung. Es ermöglicht Datenübertragung, ohne ein neu laden der Webseite zu erfordern. Dadurch wird die Wartezeit für den Benutzer vermindert. Allerdings führt es auch dazu das einige Browserfunktionen, wie der Rückwärts-Button nicht mehr richtig funktionieren.

4.1) Document-Node, Element-Nodes, Text-Nodes, Attribut-Nodes

4.2) Das `name`-Attribut ist nicht eindeutig. Außerdem hat das `name`-Attribut für manche HTML-Elemente unterschiedliche Bedeutungen.

4.3) Bei Verwendung von `innerHTML` wird der Text auch als HTML geparkt.

5.1) Mit der F12 Taste oder dem „Hamburger-Menü“.

5.2) `console.log()`

5.3) Breakpoints sind vom Entwickler gesetzte Punkte im Code an denen die Programmausführung unterbrochen wird und dienen dem Debugging.

5.4) Der Debugger bietet eine gut strukturierte Übersicht über das Programm. Außerdem ermöglicht er es Variablen und deren Werte während des Programmablaufs zu analysieren und das Programm Schritt für Schritt durchzulaufen.

6.1) Es kann online über eine URI oder bei heruntergeladener jQuery Bibliothek lokal eingebunden werden. Beide geschieht über das `src`-Attribut innerhalb des `<script>`-Tags.

6.2) Über die `$`-Funktion und einem CSS-ähnlichen Selektor.

6.3) Mit Hilfe der `$(document).ready()`-Funktion oder deren verkürzte Schreibweise.

B. Literaturverzeichnis

[Sev12] C. Severance: „JavaScript: Designing a Language in 10 Days“, in: *Computer Band 45 Ausgabe 2*; IEEE, Februar 2012, DOI: 10.1109/MC.2012.57, S. 7-8.

[Ecm19a] ECMA International, ECMA-262 ECMAScript Language Specification 2019, Introduction; <<https://www.ecma-international.org/ecma-262/10.0/index.html#Title>> (10.11.19)

[Ecm19b] ECMA International, ECMA-262 ECMAScript Language Specification 2019, 6.1.6 The Number Type; <<https://www.ecma-international.org/ecma-262/10.0/index.html#Title>> (10.11.19)

[Moz19a] Mozilla Web Docs, Grammar and Types, Variable hoisting; <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types> (15.11.19)

[Moz19b] Mozilla Web Docs, DOMContentLoaded; <<https://developer.mozilla.org/de/docs/Web/Events/DOMContentLoaded>> (22.11.19)

[Ste14] R. Steyer: JavaScript – Die universelle Sprache zur Web-Programmierung; Carl Hanser Verlag; München; 2014; ISBN 978-3-446-43942-9.

[Bew18] J. Bewersdorff: Objektorientierte Programmierung mit JavaScript: Direktstart für Einsteiger; Springer Vieweg; Wiesbaden, 2018; ISBN 978-3-658-21076-2.

[Jäg08] K. Jäger: Ajax in der Praxis; Springer Verlag; Berlin / Heidelberg, 2008; ISBN 978-3-540-69334-5.

[Ste18] R. Steyer: jQuery - Das universelle JavaScript-Framework für das interaktive Web und mobile Apps, 2. Auflage; Carl Hanser Verlag; München; 2018; ISBN: 978-3-446-45651-8.

[Moz19c] Mozilla Web Docs, Arguments-Object; <<https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/arguments>> (28.11.19)

[Moz19d] Mozilla Web Docs, Deprecated Tools; <https://developer.mozilla.org/de/docs/Tools/Deprecated_tools> (29.11.19)

C. Abbildungsverzeichnis

Abbildung 2.1: Vergleichsoperator Test	5
Abbildung 2.2: Hoisting anhand von var und let Variablen.....	6
Abbildung 2.3: Beispiel der Arrayerzeugung über den Konstruktor.....	7
Abbildung 2.4: Vergleich zwischen der for-in- und for-of-Schleife anhand eines Beispiels.....	10
Abbildung 2.5: Beispiel Zugriffsschutz und Vererbung	13
Abbildung 3.1: Vergleich Bubbling und Capturing	16
Abbildung 4.1: Beispiel DOM-Baum Manipulation	20
Abbildung 5.1: Ausgabe eines Objektes auf der Konsole	22
Abbildung 5.2: Funktion und Struktur des Firefox Debuggers	23
Abbildung 6.1: DOM-Baum Manipulation mit jQuery	26

D. Tabellenverzeichnis

Tabelle 3.1: Werte des readyState-Attributs [Jäg08, S.190]	17
--	----

E. Codeverzeichnis

Syntax Beispiel 2.1: Erstellung von Arrays	7
Syntax Beispiel 2.2: Erstellung von assoziativen Arrays	8
Syntax Beispiel 2.3: If-Anweisung	8
Syntax Beispiel 2.4: Ternärer Operator	8
Syntax Beispiel 2.5: Switch-Case	9
Syntax Beispiel 2.6: while- und do-while-Schleife	9
Syntax Beispiel 2.7: for-Schleifen.....	10
Syntax Beispiel 2.8: Deklaration einer Funktion.....	11
Syntax Beispiel 2.9: Funktion mit Rest-Parameter	11
Syntax Beispiel 2.10: Erstellung eines Objektes über eine Konstruktor-Funktion	11
Syntax Beispiel 4.1: Selektion von HTML-Elementen mittels DOM-Methoden	19
Syntax Beispiel 6.1: Ausführen von Anweisungen nach dem Laden des DOM-Baumes	25
Syntax Beispiel 6.2: jQuery Selektoren.....	25