Clean(er) Code mit ECMAScript 6

JavaScript ist nicht gerade für seine übersichtlichen Quellcodes bekannt. ECMAScript 6 versucht mit dieser Tatsache aufzuräumen. In diesem Artikel stelle ich einige der neuen Features auf den Prüfstand und versuche herauszufinden, wie man heutige JavaScript-Projekte übersichtlicher und damit einfacher lesbar gestalten kann.

Der Kern von JavaScript wurde innerhalb von drei Tagen entwickelt und war ursprünglich nur dazu gedacht, etwas Dynamik in die bis dahin statischen HTML-Seiten zu bringen. Heute erfreut sich JavaScript einer immer stärker werdenden Beliebtheit und wird zur Umsetzung ganzer Anwendungen eingesetzt.

Damit ist klar, dass sich die Anforderungen an die Sprache seit ihrer Schaffung drastisch geändert haben. Komplexere Anwendungen und größere Teams erfordern eine gute Organisation und Lesbarkeit des Quellcodes. Eine Eigenschaft, für die JavaScript heute nicht gerade bekannt ist.

ECMAScript 2015 ist die inzwischen sechste Spezifikation für JavaScript. Sie bringt Neuerungen, die bei der Erfüllung dieser Anforderungen helfen können. Nachfolgend stelle ich die meiner Meinung nach wichtigsten Neuerungen in ECMAScript 6 vor und zeige, wie diese den bisherigen Code (basierend auf ECMAScript 5) verbessern können.

Strict Mode

Der `Strict Mode` ist im Grunde Bestandteil des ECMAScript 5. Er wurde allerdings als Vorbereitung auf ECMAScript 6 konzipiert. Somit ist ein aktiver `Strict Mode` Voraussetzung für die Nutzung einiger der hier vorgestellten Features.

`Strict Mode` versetzt die JavaScript-Runtime in einen - wie der Name schon sagt - restriktiveren Ausführungsmodus. Hier eine Kurzfassung der wichtigsten Änderungen:

- Das `with`-Statement führt zu einem Syntax-Fehler.
- Doppelte Eigenschaftsnamen in Objekten führen zu einem Syntax-Fehler.
- Oktale Notation für Nummern führen zu einem Syntax Fehler.
- Nicht definierte Variablen führen zu einem Referenz-Fehler.
- Doppelte Funktionsparameter führen zu einem Syntax-Fehler.
- Die Schlüsselwörter `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` and `yield` dürfen nicht mehr als Bezeichner verwendet werden.

Aktiviert wird der Modus wahlweise für die gesammte Datei oder eine einzelne Funktion.

```
'use strict'; // Strict Mode global aktivieren

function strict() {
'use strict';
        // Strict Mode nur auf Funktionsebene
}
```

Der `Strict Mode` sorgt somit dafür, dass typische Probleme und eventuelle Kompatibilitätsprobleme mit nachfolgenden ECMA-Spezifikationen vermieden werden. Es ist also empfehlenswert, den `Strict Mode` so oft es geht, zu aktivieren und alte Scripte mit dessen Hilfe zu migrieren.

Variablen

Bisher konnte man Variablen nur mit dem Schlüsselwort `var` bestimmen. `var` definiert die Variable dabei untypisiert auf Funktionsebene. Das bedeutet, die Variable ist unabhängig von der Position der Definition in der gesammten Funktion verfügbar. Das ermöglicht zum Beispiel, dass das folgende Script eine "10" anstatt eines Fehlers ausgibt:

```
for (var i = 0; i < 10; i++) {
        console.log(i); // 0, ..., 9
}
console.log(i); // 10
```

Die Variable `i` ist zwar auf dem Blocklevel der For-Schleife definiert worden, ist aber auf Funktionsebene verfügbar, im Beispiel sogar global. Spinnt man dieses Verhalten etwas weiter, kann man sich wahrscheinlich gut vorstellen, welch skurrile Fehler dies hervorrufen kann.

Let

Abhilfe schafft das neue Schlüsselwort `let`. Es beschränkt die Sichtbarkeit der Variable auf das Blocklevel. Tauscht man im Beispiel das Schlüsselwort `var` durch `let` aus, so kommt es zu einem Referenz-Error. `let` trägt somit zu einem leichter verständlichen Quellcode bei, vermeidet schwer nachvollziehbare Fehler und ist wahrscheinlich die Verhaltensweise, die ein Neueinsteiger von `var` erwarten würde.

Const

Ein weiteres neues Schlüsseltwort zur Definition von Variablen ist `const`. Es ermöglicht das Anlegen von Konstanten. Konstanten sind wie in anderen Sprachen auch, schreibgeschützte Variablen. Die Gültigkeit der Konstante ist wie bei `let` lediglich auf Blocklevel.

```
'use strict';

const constObject = {
        prop: 'oldValue'
};

// Operation schlägt mit einem TypeError fehl.
constObject = {
        prop: 'newValue'
};

// Aber diese klappt!
constObject.prop = 'newValue';
```

Das Beispiel zeigt eine Besonderheit: das Schreiben von Properties innerhalb von konstanten Objekten ist zulässig. Das bedeutet, dass bei Objekten lediglich die Referenz auf das Objekt nicht neu geschrieben werden darf. Der Wert kann allerdings durch Operationen auf dem Objekt verändert werden. In dieser Eigenschaft gleicht es dem Java-Schlüsselwort `final`.

Werden Konstanten richtig angewendet, ermöglichen sie es, potenziellen Programmiererfehlern vorzugreifen. Insbesondere im Zusammenhang mit den ECMAScript 6 Modulen kann dieses neue Schlüsselwort die Fehleranfälligkeit des Quellcodes verringern.

Utility

Utility-Sammlungen wie `lodash` oder `underscore` sind sehr hilfreiche Werkzeuge beim Arbeiten mit Arrays und Objekten. Mit ECMAScript 6 erhält JavaScript die Funktion `Array.prototype.find` und es wird fraglich, ob diese Sammlungen nun noch benötigt werden, denn viele Funktionen der Sammlungen haben inzwischen ein passendes Gegenstück in der Spezifikation gefunden.

Die meines Erachtens drei wichtigsten Funktionen stelle ich nun kurz vor. Als Grundlage für die nachfolgenden Beispiele wird ein Array `users` angenommen:

```
let users = [
  { 'name': 'Mia', 'age': 14 },
  { 'name': 'Max', 'age': 40 },
  { 'name': 'Pablo','age': 18 }
];
```

find

Find (ECMAScript 6) findet das erste Element in einem Array, für welches die übergebene Filter-Funktion den Wert `true` zurückgibt.

```
let isMax = function(user) {
   return user.name == 'Max';
}

// Finde den Benutzer 'Max'.
let max = users.find(isMax);
```

forEach

ForEach (ECMAScript 5.1) iteriert über alle Elemente des Arrays und übergibt den jeweiligen Wert an eine Callback-Funktion, in der die Informationen verarbeitet werden können. Optional kann das Callback als zweiten Wert auch noch den Index des Elements im Array und als dritten Wert das Array selbst übergeben bekommen.

```
let logUser = function(user) {
   console.log(user);
}

// Gebe jeden Benutzer aus.
users.forEach(logUser);

// Alternativ auch direkt mittels console.log
// Es wird nur die Referenz auf die
// Funktion übergeben.
users.forEach(console.log);
```

filter

Die Filter-Funktion wurde zwar bereits in ECMAScript 5.1 spezifiziert, soll hier aber trotzdem erwähnt werden, da sie die beliebte gleichnamige Funktion aus den Utility-Sammlungen ersetzt. `Filter` filtert anhand einer Funktion die zurückgegebenen Einträge. Im Resultat wird ein neues Array ausgegeben, welches lediglich Einträge enthält, für die die Filter-Funktion `true` zurückgibt.

```
let isYoungerThan20 = function(user) {
   return user.age < 20;
}

// Nutzer, die jünger als 20 sind, werden
// gefiltert. Es bleiben nur Mia und Pablo
// im Ergebnis.
let teenager = users.filter(isYoungerThan20);
```

Hat man also Sammlungen wie `lodash` oder `underscore` im Einsatz oder plant man, diese in einem neuen Projekt einzusetzen, lohnt es sich vorher zu prüfen, welche Funktionen benötigt werden und ob diese bereits in JavaScript

vorhanden sind. So erspart man sich unnötige Diskussionen über die einzusetzende Hilfssammlung und ein paar Kilobyte in der Anwendungsgröße.

Funktionen

Arrow Functions

Die Definition einer Funktion kann in ECMAScript 6 nun mit der neuen Notation der `arrow functions` abgekürzt werden. Das folgende Codebeispiel definiert eine Funktion und speichert dessen Referenz in einer Variable `helloFunction`. Die Funktion wird einmal unter Anwendung von ECMAScript 5 und einmal mittels der `arrow functions` definiert.

```
'use strict';
let helloFunction;

// ECMAScript 5
helloFunction = function(name) {
   console.log('Hello ' + name);
};

// ECMAScript 6
helloFunction =
      name => console.log('Hello ' + name);
```

Hat eine Funktion keine oder mehrere Parameter ändert sich der Aufbau wie folgt:

```
'use strict';

// Mehrere Parameter
let greet = (greeting, name) =>
        console.log(greeting + ' ' + name);
greet('hola', 'Pablo');

// Ohne Parameter
let helloWorld = () => {
   console.log('hello world');
};
```

An dieser Stelle kann man sich streiten, ob die besagte Neuerung den Quellcode übersichtlicher oder kryptischer macht. Ähnlich wie bei den Lambda-Ausdrücken in Java sollte diese Notation nur dort eingesetzt werden, wo lediglich einfache und vor allem kurze Quellcode-Ausschnitte verwendet werden. Zum Beispiel zur Angabe einer speziellen Sortierfunktion oder eines Suchkriteriums.

Nachfolgend ein kurzes Beispiel für eine sinnvolle Anwendung der `arrow functions`. Dazu betrachten wir noch einmal das Beispiel, welches wir für die Vorstellung der Funktion `Array.prototype.find` verwendet haben.

```
'use strict';

let users = [
  { 'name': 'Mia', 'age': 14 },
  { 'name': 'Max', 'age': 40 },
  { 'name': 'Pablo','age': 18 }
];

// Finde den Benutzer 'Max'.
// Beachte, "return" muss nicht angegeben werden.
users.find(user => user.name == 'Max');
```

Das Beispiel kann nun also als Einzeiler umgesetzt werden, die Definition der Filter-Funktion fällt deutlich kürzer aus.

An den richtigen Stellen eingesetzt, kann diese Notation den Quellcode durchaus übersichtlicher gestalten. Es sollte nur nicht jede `function` durch die neue Notation ersetzt werden.

Rest-Parameter

Auch die Rest-Parameter sind in anderen Programmiersprachen bekannte Pattern. So kennt man diese in der Java-Welt unter dem Namen `varargs`. Rest-Parameter ermöglichen es, alle in einer Funktion verbleibenden Parameter in ein Array zu schreiben.

```
'use strict';

let format = (mask, ...params) => {
   if (!mask) {
       mask = '';
   }
   params.forEach((value, index) => mask =
           mask.replace('{'+index+'}',value));
   return mask;
};

console.log(format('{0} {1}', 'hola', 'Pablo'));
```

In ECMAScript 5 musste man hierfür noch `arguments` verwenden:

```
var format = function() {
   var mask = arguments[0];

   if (typeof mask == 'undefined') {
       mask = '';
   }

   for (var i = 1; i < arguments.length; i++) {
      mask = mask.replace('{'+(i-1)+'}', arguments[i]);
   }
   return mask;
};

console.log(format('{0} {1}', 'hola', 'Pablo'));
```

Das Arbeiten mit dem Schlüsselwort `arguments` war schon immer sehr mühevoll, scheint es doch ein Array zu sein und dann wieder nicht. Es fehlen viele der Funktionen des Array-Prototyps. Man musste derartige Aufrufe also immer manuell über den Prototypen `Array.prototype.XXX.call(arguments, ...)` ausführen. Es ist sehr erfreulich zu sehen, dass der Großteil dieser Anwendungsfälle mit Rest-Parametern einfacher gelöst werden kann.

Default-Patameter

Parameter können jetzt auch einen Standard-Wert erhalten. Diese vereinfachen unser Beispiel aus dem Abschnitt zu Rest-Parametern noch weiter.

```
'use strict';

let format = (mask = '', ...params) => {
   params.forEach((value, index) => mask =
          mask.replace('{'+index+'}', value));
   return mask;
};

console.log(format('{0} {1}', 'hola', 'Pablo'));
```

Default-Parameter befreien so den Quellcode von lästigen Werteprüfungen und verbessern die Übersichtlichkeit durch klare Standardbelegungen.

Verfügbarkeit der ECMAScript-6-Features auf ausgewählten Plattformen

Feature	IE	Edge	Firefox	Chrome	Safari	NodeJS
Strict Mode	11	12	4	13	6	4.3.2
let	11	14	50	49	10	4.3.2
const	11	12	36	49	10	4.3.2
Array.prototype.find	n/a	13	45	56	9	4.3.2
Array.prototype.forEach	10	12	21	23	6	4.3.2
Array.prototype.filter	10	12	21	23	6	4.3.2
Arrow Functions	n/a	12	22	45	10	5.11.0
Rest Parameter	n/a	12	15	47	10	5.11.0
Default Parameter	n/a	14	51	49	10	6.9.3

Quellen: http://caniuse.com/, http://node.green/ und https://developer.mozilla.org/de/ (Stand: Februar 2017)

Fazit

Es wird übersichtlicher, da bin ich mir sicher. ECMAScript 6 bringt viele Änderungen, die an sein namentliches Vorbild Java erinnern. Klassen, Importe, Sichtbarkeiten der Variablen, Rest-Parameter und Lambdas sind hier nur einige Beispiele. Das ist meiner Meinung nach ein gutes Zeichen, hat sich Java doch viele Jahre in großen, komplexen und verteilten Projekten bewährt.

Die hier aufgezeigten Features von ECMA-Script 6 sind noch lange nicht alles. Es bietet mit Promises, Klassen und Modulen noch sehr interessante Themen, welche aber umfangreich genug sind, um einen weiteren Artikel zu füllen.

Weiterführende Links:

Strict Mode:
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Strict_mode

Default Parameter Unterstützung:
https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Default_parameters#Browser_compatibi

ES6-Umsetzung:
http://kangax.github.io/compat-table/es6/

ES6-Umsetzung:
http://caniuse.com/

NodeJS ES6-Umsetzung:
http://node.green/

Michael Ruttka

Michael Ruttka ist Berater bei der buschmais GbR. Seine fachlichen Schwerpunkte liegen in der Entwicklung und Konzeption von Webanwendungen.

Impressum | **Herausgeber** buschmais GbR, Leipziger Straße 93, 01127 Dresden | **Telefon** +49 351 3209230 | **Telefax** +49 351 32092329 | **E-Mail** info@buschmais.com | **Internet** www.buschmais.de |