



## **) EcmaScript 6 und TypeScript )**

Version 1.0.8 (01.07.2017, Autor: Frank Bongers, Webdimensions.de)  
© 2017 by Orientation In Objects GmbH  
Weinheimer Straße 68  
68309 Mannheim  
<http://www.oio.de>

Das vorliegende Dokument ist durch den Urheberschutz geschützt. Alle Rechte vorbehalten. Kein Teil dieses Dokuments darf ohne Genehmigung von Orientation in Objects GmbH in irgendeiner Form durch Fotokopie, Mikrofilm oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere Datenverarbeitungsanlagen verwendbare Sprache übertragen werden. Auch die Rechte der Wiedergabe durch Vortrag sind vorbehalten.

Die in diesem Dokument erwähnten Soft- und Hardwarebezeichnungen sind in den meisten Fällen eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

## Inhaltsverzeichnis

1.	EINFÜHRUNG .....	8
1.1.	JAVASCRIPT – EIN BLICK AUF DIE ABSTAMMUNG .....	8
1.1.1.	JavaScript, die Grundsyntax .....	9
1.1.2.	JavaScript als objektorientierte Sprache .....	9
1.1.3.	JavaScript als funktionale Programmiersprache .....	9
1.1.4.	Der Standard – ECMA-Script .....	10
2.	ECMAScript 6 IMPLEMENTIERUNG UND TRANSPILER .....	12
2.1.	BABEL .....	13
2.2.	BROWSERIFY .....	13
2.3.	TRACEUR .....	14
2.4.	COFFEEScript, DART, OBJECTIVE J UND TYPESCRIPT .....	14
2.4.1.	CoffeeScript .....	15
2.4.2.	Dart .....	15
2.4.3.	Objective J .....	16
2.4.4.	TypeScript .....	16
2.4.5.	Weitere Sprachen und Tools .....	17
3.	TYPESCRIPT ALS ENTWICKLUNGSSPRACHE .....	18
3.1.	TYPESCRIPT-TRANSPILER ALS NODE-MODULE .....	19
3.1.1.	Test der Installation .....	20
3.1.2.	Kompilieren einer TypeScript-Source .....	20
3.2.	TYPESCRIPT IN WEBSTORM .....	21
3.2.1.	Automatische TypeScript-Kompilierung .....	22
3.2.2.	Händische TypeScript-Kompilierung .....	24
3.3.	TYPESCRIPT IN VISUAL STUDIO CODE .....	25
3.3.1.	tsconfig.json .....	26
3.3.2.	Linten für TypeScript-Quellen .....	28
3.3.3.	JavaScript-Build über Task-Runner .....	29
3.3.4.	JavaScript-Build über Terminal .....	31
4.	ÜBERBLICK ÜBER ECMAScript 6 .....	33
4.1.	SYNTAXERWEITERUNGEN .....	33
4.2.	CODE ORGANISATION .....	34
4.3.	ASYNCHRONE PROGRAMMIERUNG .....	34
4.4.	WERTE-COLLECTIONS .....	34
4.5.	MEHR API FÜR BASISOBJEKTTYPEN .....	35
4.6.	WEITERE NEUERUNGEN .....	35
5.	ERWEITERUNGEN DER GRUNDSYNTAX .....	35
5.1.	BLOCKSCOPE MIT LET .....	35
5.1.1.	Recap: var-Keyword zur Deklaration von Variablen .....	36
5.1.2.	let-keyword zur Deklaration von Variablen .....	37
5.2.	KONSTANTEN MIT CONST .....	37

5.2.1.	Das const-Keyword zur Deklaration von Konstanten .....	38
5.2.2.	Nicht überschreibbare Funktionen .....	39
5.3.	VERGLEICH VON WERTEN: „ULTRASTRIKT“ MIT OBJECT.IS() .....	40
5.3.1.	Object.is().....	40
5.4.	DER SPREAD- UND REST-OPERATOR .....	41
5.4.1.	Der Operator `...` als Spread-Operator .....	42
5.4.2.	Der Operator `...` als Rest-Operator.....	43
5.5.	DEKONSTRUKTION KOMPLEXER WERTE .....	44
5.5.1.	Händische Dekonstruktion komplexer Werte .....	45
5.5.2.	Echte Dekonstruktion von Arrays und Objekten.....	45
5.5.3.	Partielle Dekonstruktion von Objekten .....	46
5.6.	DEFAULTPARAMETER FÜR FUNKTIONEN .....	47
5.6.1.	Die Situation in ECMA5.....	47
5.6.2.	Echte Defaultparameter in ECMA6 .....	48
5.7.	ARROW-FUNKTIONEN .....	48
5.7.1.	Das function-Keyword und das thisObj .....	48
5.7.2.	Anonyme Funktionen .....	52
5.7.3.	Die Arrow Funktion.....	52
5.8.	STRINGS UND TEMPLATE-LITERALE .....	59
5.8.1.	Echte Multiline Strings mit Backticks.....	59
5.8.2.	Templating in Backtick-Strings .....	60
6.	OBJEKTORIENTIERUNG UND VERERBUNG .....	62
6.1.	DAS OBJECT-OBJEKT.....	62
6.2.	OBJEKT-LITERALE IN ECMA6.....	63
6.2.1.	Shorthand-Definitionen von Properties .....	63
6.2.2.	Shorthand-Definition von Methoden.....	64
6.2.3.	Shorthand-Definitionen mit „computed keys“ .....	66
6.3.	METHODEN DES OBJECT-OBJECTS .....	66
6.3.1.	Object.create() .....	66
6.3.2.	Object.defineProperty() und .defineProperties().....	68
6.3.3.	Object.preventExtensions(), .seal() und .freeze().....	68
6.3.4.	Object.isExtensible(),.isSealed() und .isFrozen() .....	69
6.3.5.	Object.keys() und Object.getOwnPropertyNames() .....	69
6.3.6.	Object.getOwnPropertySymbols().....	70
6.3.7.	Kopieren von Eigenschaften mit Object.assign() .....	71
7.	KONSTRUKTOREN MIT CLASS-KEYWORD.....	71
7.1.	GETTER UND SETTER IN KLASSEN .....	73
7.2.	VERWENDUNG VON SYMBOLEN .....	76
7.3.	EXTENDS UND SUPER – VERERBUNG MIT KLASSEN .....	78
8.	ERWEITERUNG DER ARRAYSYNTAX.....	79
8.1.	NEUE STATISCHE FUNKTIONEN FÜR DEN ARRAY-TYP .....	79
8.1.1.	Array.of() .....	80
8.1.2.	Array.from .....	81
8.2.	NEUE INSTANZMETHODEN FÜR DEN ARRAY-TYP .....	82
8.2.1.	[].copyWithin() .....	82
8.2.2.	[].fill() .....	82

8.2.3. [].find() .....	83
8.2.4. [].findIndex().....	83
9. DAS ITERABLE UND DIE FOR-OF-SCHLEIFE .....	83
9.1. DIE FOR-OF-SCHLEIFE .....	84
9.2. ITERARABLE - DIE METHODEN ENTRIES(), VALUES(), INDEX() .....	85
9.3. DAS ITERABLE-OBJECT.....	86
10. ASYNCHRONE PROGRAMMIERUNG .....	87
10.1. PROMISES .....	87
10.1.1. Die Methode .then().....	89
10.1.2. Die Methode .catch() .....	90
10.1.3. Promise-Beispiel mit setTimeout().....	91
10.1.4. Werte transformieren mit Promise-Chains.....	93
10.1.5. Promises und Ajax.....	96
10.1.6. Synchron/asynchroner Getter.....	98
10.2. GENERATOREN .....	102
10.2.1. Die Generatorfunktion function* („function star“) .....	102
10.2.2. Generator-Objekt und for-of Schleife .....	104
10.2.3. Boxenstopp im Code - „code suspension“ .....	105
10.2.4. Stopp für Input - Iteration Messaging.....	107
10.2.5. Werte erzeugen mit Generator und for-of .....	108
10.2.6. Ajax mit Generator.....	110
11. MODULARISIERUNG .....	114
11.1. IIFE-BASIERTE MODULPATTERN .....	115
11.1.1. Das „revealing“ und das „global Import“-Pattern .....	116
11.2. COMMONJS MODULE, AMD- UND UMD-MODULDEFINITIONEN .....	118
11.2.1. Common JS.....	119
11.2.2. AMD .....	120
11.2.3. Universal Module Definition .....	121
11.3. ECMA6 MODULE .....	123
11.3.1. Dateibezogene Module.....	123
11.3.2. Scoping von Modulen .....	124
11.3.3. Agnostische Module .....	124
11.3.4. Das Keyword „export“ .....	125
11.3.5. Import .....	128
11.3.6. Haupt- und Submodule mit ECMA6-Modulen.....	130
11.3.7. Laden von ECMA6-Modulen mit SystemJS.....	131
12. EINFÜHRUNG IN TYPESCRIPT .....	136
12.1. TYPISIERUNG IN TYPESCRIPT .....	137
12.2. TYPESCRIPT-TYPEN .....	137
12.3. TYPINFERENZ MIT „DIRECT“ UND „CONTEXTUAL“ TYPING .....	138
12.3.1. Der implizite Typ „any“ .....	138
12.3.2. Typing über Returnwert einer Funktion .....	139
12.3.3. Melden des impliziten Typs „any“ .....	140
12.4. STATISCHE TYPISIERUNG MIT TYPANNOTATION .....	141
12.4.1. Verhalten von null und undefined .....	142

12.4.2. Der explizite Typ „any“ .....	144
12.5. DER ENUM-TYP .....	144
12.6. UNION-TYPES.....	145
12.7. ARRAYS UND TYPISIERTE ARRAYS .....	146
12.7.1. Annotation von Array-Typen.....	147
12.8. TYPISIERUNG VON FUNKTIONEN .....	148
12.8.1. Typisierung von Funktionsparametern .....	148
12.8.2. Typisierung des Returnwertes einer Funktion .....	149
12.8.3. Funktionen ohne Returnwert .....	149
12.8.4. Arrow-Funktionen.....	150
12.8.5. Optionale Parameter .....	151
12.8.6. Defaultparameter .....	152
12.8.7. Der Function-Type .....	152
12.8.8. Function-Type und Callback-Parameter.....	153
12.8.9. Signatur-Overloading .....	154
12.8.10. Generische Funktionen.....	155
12.9. CLASSES IN TYPESCRIPT.....	157
12.9.1. Ausgangspunkt ECMA6 .....	157
12.9.2. Deklaration und Annotation von Properties .....	158
12.9.3. Modifier: public, private, protected, static .....	159
12.9.4. Typisierung von Konstruktor-Parametern.....	163
12.9.5. Modifier „public“ vor Konstruktor-Parametern.....	164
12.9.6. Weitere Modifier für Konstruktor-Parameter.....	166
12.9.7. Klassen als Annotation.....	168
12.9.8. Abstrakte Basisklassen .....	170
12.10. INTERFACES .....	171
12.10.1. Anonymes Interface .....	171
12.10.2. Benanntes Interface.....	172
12.10.3. Funktion implementiert Interface.....	173
12.10.4. Klasse implementiert Interface.....	173
12.10.5. Implementierung mehrerer Interfaces .....	174
12.10.6. Klassen als Interfaces.....	175
13. DECORATOR – ANNOTATION UND MODIFIKATION.....	176
13.1. CLASS-DECORATOR.....	177
13.1.1. Ausloggen der Instanzierung einer Klasse .....	178
13.1.2. Erweitern des Prototypes einer Klasse .....	179
13.2. METHOD-DECORATOR.....	180
13.3. PROPERTY-DECORATOR .....	182
13.4. PARAMETER-DECORATOR .....	184
13.4.1. Ausloggen des dekorierten Parameters .....	186
13.5. DECORATOR-FACTORY .....	188
14. LITERATUR.....	190
15. ONLINERESSOURCEN .....	191
15.1. LINKS ZU JAVASCRIPT .....	191
15.2. LINKS ZU TYPESCRIPT.....	191

# 1. Einführung

---

JavaScript ist unbestritten die **wichtigste Programmiersprache** des Internets. Sie läuft auf allen aktuellen Webplattformen vom Browser bis zum Smartphone, sowie in weiteren Umgebungen (z.B. mittels der Rhino-Laufzeit auch in Java-Applikationen).

Die Sprache wurde aufgrund der übersichtlichen Syntax als leicht erlernbar betrachtet, ihre Mächtigkeit hingegen galt wegen des „recht übersichtlichen“ Sprachumfangs als defizitär. Die aktuelle Version **ECMAScript 6.0**, seit Sommer 2015 offiziell standardisiert, erweitert diesen Sprachumfang nun wesentlich. Die in diesem Rahmen erfolgten **Erweiterungen** sind Gegenstand dieses Seminars. Zusätzlich werfen wir einen Blick auf **TypeScript** als Cross-Compiling Sprache zur Erstellung von ECMA6-Anwendungen.

## 1.1. JavaScript – Ein Blick auf die Abstammung

---

JavaScript wurde im Jahr 1995 von *Brendan Eich* für **Netscape** für die Erstellung interaktiver Webseiten entwickelt. Trotz des Namens besteht keine enge Verwandtschaft mit Java (dieser wurde aus reinen Marketingerwägungen gewählt).

JavaScript hat **drei Stammsprachen**, aus denen der Sprachumfang abgeleitet ist: C++/Java, Self und Scheme.

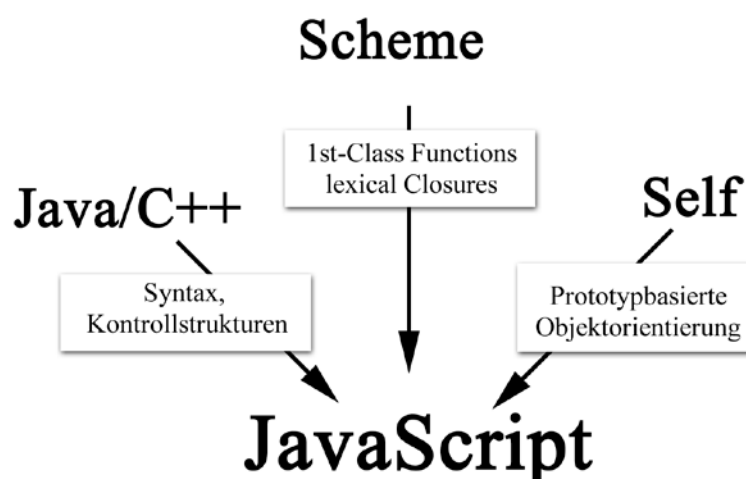


Abb.: Ableitung von JavaScript aus Scheme, Self und C++/Java

### 1.1.1. JavaScript, die Grundsyntax

---

Die **Syntax** der Sprache wurde aus dem Umfeld **C++/Java** abgeleitet. Die Schreibweise von Statements, Kontrollstrukturen und viele Keywords sind nahezu identisch mit denen der Stammsprachen, was den Einstieg für Programmierer aus den beiden genannten Sprachen (scheinbar) erleichtert. Gleichzeitig werden so aber auch, wegen diverser Unterschiede, Hürden aufgerichtet.

✓ **Besonderheit:** Aufgrund der Anforderungen an die Laufzeitdynamik entschied man sich gegen die Implementierung einer *festen* Typisierung und für ein sogenanntes „**loose Typing**“. JavaScript besitzt zwar Typen, doch sind diese an Werte und nicht an Variable gebunden.

### 1.1.2. JavaScript als objektorientierte Sprache

---

JavaScript war von Anfang an als **objektorientierte Sprache** konzipiert. *Brendan Eich* empfand die aus Java bekannte *klassenbasierte* Objektorientierung jedoch als „zu schwierig“ und entschied sich stattdessen für eine, von **Self** abgeleitete **prototypbasierte Objektorientierung**.

Anstatt Objekte von *Klassen* abzuleiten, sollte der Vererbungsmechanismus ebenfalls über *Objekte* laufen (Objekte erben von Objekten, nämlich dem „Prototyp“-Objekt).

Von Programmierern, die aus einer *klassenbasierten* Umgebung kommen, wird die *prototypbasierte* Objektorientierung oft als verwirrend empfunden. Sie ist jedoch lediglich „anders“. Um dem entgegenzuwirken und die ursprünglich sehr rudimentären Vererbungsmechanismen syntaktisch aufzuwerten, wurden dem Sprachumfang sukzessive weitere Syntaxmöglichkeiten hinzugefügt.

✓ **Fazit:** JavaScript ist eine vollwertige **objektorientierte Sprache**.

### 1.1.3. JavaScript als funktionale Programmiersprache

---

Eine oft verkannte Quelle des Sprachumfangs von JavaScript liegt in der (funktionalen) Sprache **Scheme**, von der die Arbeit mit Funktionen in JavaScript abgeleitet ist. In JavaScript handelt es sich bei Funktionen um



Objekten quasi gleichgestellte Typen (sog. „callable“ Objects), die somit als Eingabe- oder Rückgabewerte anderer Funktionen dienen können.

✓ In diesem Sinne sind Funktionen in JavaScript **1st-Class Citizens**.

Aus dieser Besonderheit (eine Funktion ist ein „Wert unter Werten“) leitet sich ab, dass Funktionen (als Wert) in Variablen oder Objekteigenschaften abgelegt werden können, als Parameter in Funktionen hineingereicht (ermöglicht **Higher-Order-Funktionen**) oder als Rückgabewert aus Funktionen hinausgereicht werden können (ermöglicht **lexical Closures**).

Da Funktionen somit aus ihrem Entstehungskontext *verschoben* werden können, muss der **aufrufende Kontext** einer Funktion bei ihrem Aufruf *dynamisch* übergeben werden. Dies geschieht in Form des **this-Objekts**, sorgt aber in Grenzfällen für scheinbar unberechenbares Verhalten.

Mittels dieser Syntaxstrukturen lassen sich in JavaScript alle wesentlichen *funktionalen* Programmierparadigmen nachvollziehen.

✓ Fazit: JavaScript ist eine vollwertige funktionale Programmiersprache.

#### 1.1.4. Der Standard – ECMA-Script

---

Die „European Computer Manufacturers Association“ ECMA spezifiziert JavaScript im **Standard ECMA-262**.

✓ Seit geraumer Zeit halten sich alle Browserhersteller an den ECMA-Standard. Alle aktuellen Browser können daher ECMA-kompatible Scripte ausführen.

✓ Bei der Weiterentwicklung des Standards wird auf **strikte Abwärtskompatibilität** geachtet. Ein aktueller Browser wird also stets ältere Scripte ausführen können, die einer früheren ECMAScript-Version entsprechen.

#### ECMAScript 1.0 und 2.0

Die *erste Version* des Standards (1997) war annähernd kompatibel mit JavaScript 1.1. Sie wurde unter der Bezeichnung **ISO/IEC 16262**

ebensovon der ISO übernommen und wurde seitdem mehrfach aktualisiert. Sie diente eher als Beschreibung des damals aktuellen Sprachumfangs der Netscape-Implementierung. Das gleiche gilt für ECMAScript 2.0.

### ECMAScript 3

Die dritte Version (1999) erweiterte erstmals den Sprachumfang. Neu hinzu kamen *reguläre Ausdrücke*, *Stringfunktionen*, *Fehlerbehandlung* mit try/catch und andere praktische Erweiterungen.

- ✓ Diese Version war bis 2009 aktueller Standard (ES3-Skripte sind in aktuellen Browsern noch immer lauffähig).

### ECMAScript 4

Anfangs auch als „JavaScript 2.0“ bezeichnet, war eine sehr umfassend Erweiterung der Sprachsyntax geplant, die in Richtung klassenbasierter Vererbung hätte gehen können. Aufgrund von Uneinigkeit im Entwicklerteam wurde der Entwurf fallengelassen.

- ✓ Die Version ECMAScript 4 wurde nie standardisiert.

### ECMAScript 5 und ECMAScript 5.1

Die „fünfte“ (praktisch die vierte) Version der Sprache wurde 2009 verabschiedet und 2011 als 5.1 erweitert. Als Erweiterung wurde ein *“strict mode“* der Runtime eingeführt, um Defizite der Vorgängerversion abwärtskompatibel zu beheben. Neu sind auch die native Unterstützung des *JSON*-Formats und zusätzliche Objekt- und Array-Methoden.

- ✓ Aktuell (2016) ist diese Version noch der meistimplementierte Standard.

### ECMAScript 6 / ECMAScript 2015

Die sechste Version der Sprache wurde Sommer 2015 standardisiert und wird derzeit in der aktuellen Browsergeneration implementiert. Der Sprachumfang wird wesentlich erweitert und bietet interessante Möglichkeiten zur asynchronen Programmierung (*Generatoren*), zur Modularisierung (*Module*, Import, Export), sowie zur Definition der Objektvererbung (Klassen), aber auch Erweiterungen der Typen (*Maps*, *Sets*) und der Grundsyntax (*Blockscope-Variablen*, *Arrow-Funktionen*).

- ✓ Der Grundcharakter der Sprache als prototypbasierte objektorientierte Sprache wird dennoch nicht angetastet. Die neuen Syntaxmöglichkeiten sind in diesem Sinne nur „syntactic sugar“.

### Bennennung der ECMA-Versionen in Zukunft

ECMAScript 6 war längere Zeit unter dem Codenamen „Harmony“ bekannt. Im Anschluss an die offizielle Verabschiedung als ECMAScript 6 (und wegen der gleichzeitigen Vorbereitung des Nachfolgers ECMAScript 7) entschied man sich zur Umbenennung in **ECMAScript 2015**.

Die Nachfolgeversion (ehemals ECMAScript 7) soll noch dieses Jahr als **ECMAScript 2016** erscheinen, was eine höhere Frequenz der Spracherweiterung in Form *jährlicher Updates* für die Folgejahre nahelegt.

- ✓ Wir verwenden weiterhin die Bezeichnung **ECMAScript 6**. (ES6)

## 2. ECMAScript 6 Implementierung und Transpiler

Die Implementierung von ECMAScript 6 (kurz ES6) in aktuellen Browsern ist noch nicht komplett, nähert sich jedoch langsam der Vollständigkeit. Einen Überblick erhalten Sie in der Matrix von *Juriy "kangax" Zaytsev* unter folgender Adresse:

- ✓ **ES6 Supportmatrix:** <http://kangax.github.io/compat-table/es6>

The screenshot shows the 'ECMAScript 6 compatibility' table. It lists various ES6 features on the left, such as 'let', 'const', 'destructuring', 'classes', and 'modules'. For each feature, it shows support status (Yes/No) across different platforms: Chrome, Firefox, Safari, Edge, IE, Opera, Node, and others. The table is color-coded: green for 'Yes', red for 'No', and yellow for 'Partial' or 'Experimental' support.

Abb.: ECMAScript 6 in Browsern ([kangax.github.io/compat-table/es6/](http://kangax.github.io/compat-table/es6/))

Aufgrund der geplanten jährlichen Weiterentwicklung von JavaScript, wird auf längere Sicht eine Asynchronizität zwischen der Sprache und den aktuellen Implementierungen in den Browsern vorliegen.

Um den aktuellen Sprachstand in einen implementierten Sprachlevel umzuwandeln wird daher der Einsatz von Transpilern („Transcompiling“ JavaScript nach JavaScript) unumgänglich bleiben.

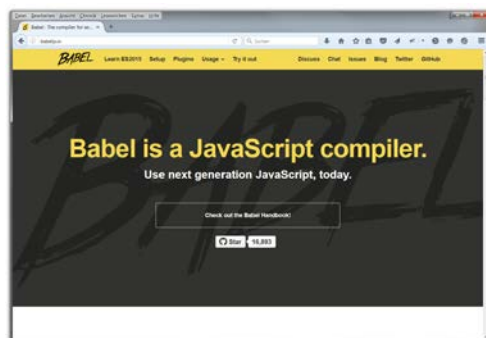
- ✓ Aktuell wird üblicherweise ECMA6 nach ECMA5 transpiliert. In Zukunft kann ECMA2017 nach ECMA6 gewandelt werden müssen und so fort.

Als gängige **Transpilertools** haben sich etabliert:

## 2.1. Babel

---

**Babel** eignet sich für statische Builds von ECMA5-ScripTEN aus ECMA6-Sourcen. Es kann über `babelify` (wobei indirekt `Browserify` verwendet wird, s.d.) in *Grunt* oder *Gulp*-Prozesse eingebunden werden und so der Transpile-Vorgang innerhalb eines Builds automatisiert werden. In Form von `babel-node` liegt auch ein NPM-Modul für ältere NodeJS-Versionen vor (in der aktuellen Version 6.x kann NodeJS ECMA6-Sourcen direkt ausführen).



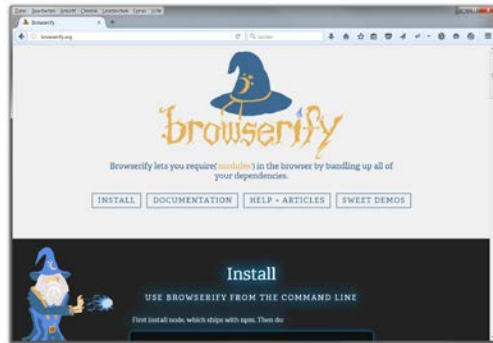
- ✓ Mehr: <http://babeljs.io/>

## 2.2. Browserify

---

Mit **Browserify** lassen sich client-seitige JavaScript-Projekte mit ES6-Modulen aufbauen, obwohl diese nicht nativ im Browser implementiert sind. Browserify transformiert hierfür alle Module und ihre Abhängigkeiten und erzeugt eine einzige, im Browser lauffähige Datei, das

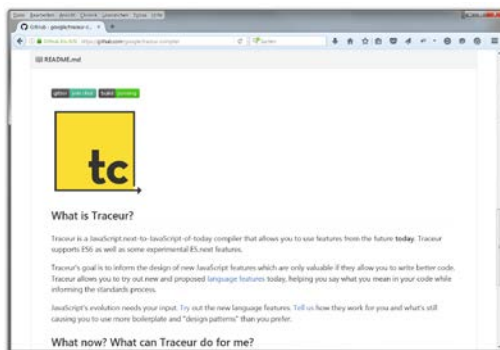
Browserify-Bundle. Obendrein können beim Einsatz von Browserify auch NPM-Module in ein Projekt integriert werden, was sonst nur für serverseitige JS-Projekte gilt.



✓ Mehr: <http://browserify.org/>

## 2.3. Traceur

**Traceur** (Google) ist ein Transpile-Tool, das ECMA6 nach ECMA5 rückübersetzt und zusätzlich einige, für noch aktuellere JavaScript-Versionen geplante Features berücksichtigt.



Anm.: Laut Kangax-Matrix werden einige relevante ES6-Sprachelemente von Traceur derzeit (noch) nicht unterstützt!

✓ Mehr: <https://github.com/google/traceur-compiler>

## 2.4. CoffeeScript, Dart, Objective J und TypeScript

Neben dem reinen JavaScript haben sich einige Sprachen (mehr oder weniger) etabliert, die entweder eine **alternative Syntax** bieten oder einen

**Superset** zu JavaScript bilden (dies dann meist auf JavaScript in der Version ES6).

### 2.4.1. CoffeeScript

CoffeeScript setzt eine, gegenüber JavaScript vereinfachte, an Ruby und Python angelehnte Syntax ein. So können beispielsweise Semikolons weggelassen und Blockklammern durch Einrückungen ersetzt werden. Das Hauptargument für die Sprache besteht in höherer Produktivität des Programmierers (weniger Code!), was allerdings mit dem Erlernen der CoffeeScript-Syntax erkauft wird. Es gibt einen Core-Compiler, der in jeder JS-Umgebung lauffähig ist, sowie ein Node-Modul.

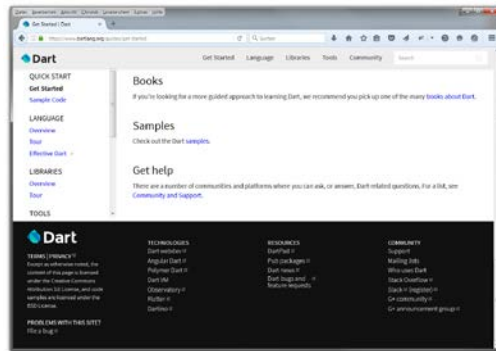


✓ Mehr: <http://coffeescript.org/>

### 2.4.2. Dart

Von Google wurde **Dart** (seit 2010) als **strukturierte, optional typisierte Alternative** zu JavaScript ins Feld geführt und eignet sich in der Tat sowohl für Webentwicklung als auch Entwicklungen im Embedded-Umfeld sowie für Mobile Applications.

Laut Google ist Dart skalierbar, also auch für „sehr große“ Anwendungen geeignet. Es existiert ein Transpiler **Dart2js**, der Dart in JavaScript übersetzt. Eine eigene Dart-VM für Browser ist derzeit nicht in Planung, soll aber für Mobilgeräte möglicherweise eingeführt werden.



✓ Mehr: <https://www.dartlang.org/>

### 2.4.3. Objective J

Bei **Objective J** handelt es sich, wie bei TypeScript, um einen *Superset* über JavaScript. Abgeleitet von Objective C wird dem JavaScript-Sprachschatz in Objective J eine **klassenbasierte Objektorientierung** hinzugefügt. Neben den nativen (prototypbasierten) JavaScript Objekten kennt Objective J auch (klassenbasierte) native Objective J-Objekte. Objective J wird im Rahmen des Entwicklungstools **Cappuccino** eingesetzt.



✓ Mehr: <http://www.cappuccino-project.org>

### 2.4.4. TypeScript

Da **TypeScript** in diesem Seminar behandelt wird, soll es nur kurz der Vollständigkeit halber erwähnt werden. Wo Objective J das Gewicht auf „echte“ Klassen legt, nimmt TypeScript eher den TypeFlow ins Visier und belässt der Sprache ihre eigene prototypbasierte Natur.

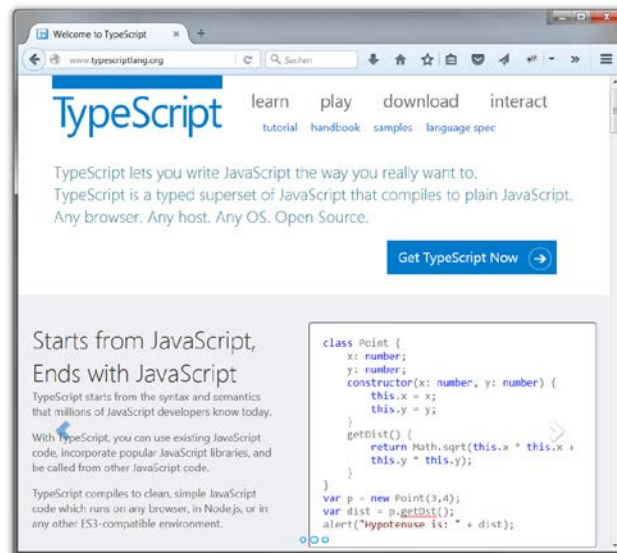


Abb.: Website [www.typescriptlang.org/](http://www.typescriptlang.org/)

## 2.4.5. Weitere Sprachen und Tools

Erwähnt werden können hier noch (ohne weitere Bewertung):

CoffeeScript II, Coco, IcedCoffeeScript, Parcec CoffeeScript, Uberscript, ToffeeScript, Caffeine, EmberScript, GorillaScript, Ham, ColaScript, CokeScript, MoonScript, Earl Grey, Spider, CirruScript, FutureScript, Mascara, PureScript, Flow, Mochiscript, Latte JS.

Desweiteren existieren für fast alle relevanten Programmiersprachen passende Tools, die diese jeweils nach JavaScript übersetzen.

Der bekannteste Vertreter ist vermutlich **GWT** (Java nach JavaScript), ähnlich gelagert ist jedoch auch *Pyjamas* (Python nach JavaScript), *Perlito* (Perl nach JavaScript) oder *Script#* (C# nach JavaScript).

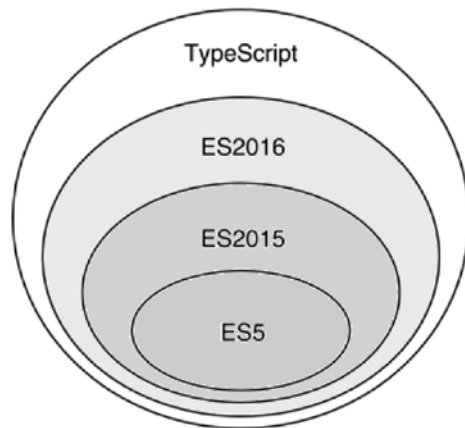
Die Liste ließe sich fortsetzen. Unter folgender Adresse finden Sie eine Linkliste zu den genannten und weiteren Sprachen und Tools.

✓ **Mehr:** <https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS>



### 3. TypeScript als Entwicklungssprache

TypeScript ist eine, von Microsoft supportete **Transpilersprache** zur Entwicklung von JavaScriptanwendungen. TypeScript umfasst in der aktuellen **Version 2.4.1** den *kompletten Sprachumfang* von ECMA6 (ES2015), sowie von Teilen des geplanten Nachfolgers ECMA7 (ES2016).



✓ Jedes JavaScript ist automatisch auch gültiges Typescript!

Zusätzlich setzt Typescript auf Typisierung (in Form von *Annotations*), was ein **statisches Typechecking** während der Entwicklung ermöglichen.

**Transpiler** von Typescript nach Javascript (wahlweise nach ECMA6, ECMA5 oder ECMA3) können als NPM-Module geladen werden oder sind in verschiedene IDEs bereits *integriert* oder *integrierbar* (Visual Studio, Visual Studio Code, Webstorm, Sublime und weitere).

Es muss erwähnt werden, dass trotz des Vollständigkeitsanspruches von TypeScript **nicht alle Syntaxelemente** von ECMA2015 und ECMA2016 in der aktuellen Version des TypeScript-Compilers unterstützt sind.

Der Compiler *belässt* betroffene Syntax-Strukturen in diesem Fall im *ursprünglichen Format* (sie werden also *nicht kompiliert*).

✓ Aktuell betrifft dies Generatoren und Proxies.

✓ **Who is who:** Von Interesse mag es sein, *wer* als maßgeblicher Kopf hinter der Entwicklung von TypeScript steht. Es handelt sich um niemand geringeren als **Anders Hejlsberg**, dem Erfinder von *Delphi*

und *Turbo Pascal*, gleichzeitig Chefarchitekt hinter der Entwicklung von *C#*. Dass entsprechend Anleihen aus diesen Richtungen in *TypeScript* einfließen, sollte daher nicht verwundern.

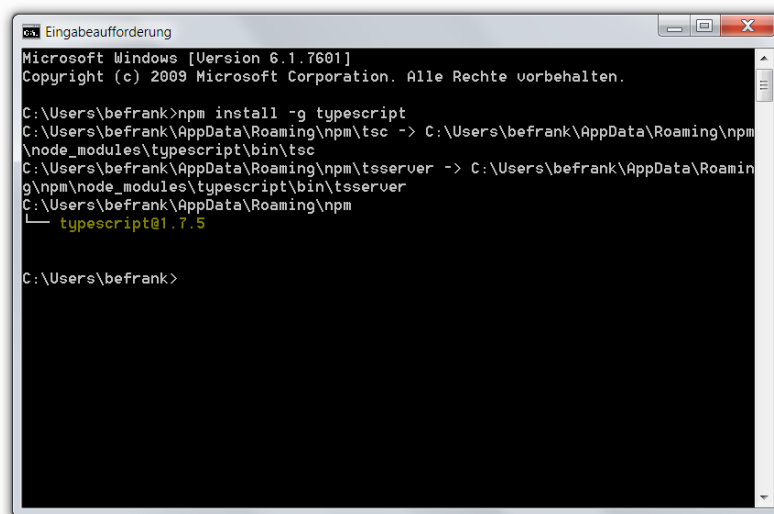
### 3.1. TypeScript-Transpiler als Node-Module

Da es keine Laufzeitumgebung gibt, die TypeScript unmittelbar ausführt, muss die Sprache mit Hilfe eines Programms, dem *Transpiler*, in eine lauffähige *Zielsprache* übersetzt werden, in diesem Fall in einen der gängigen Versionen von *JavaScript*.

- ✓ Der TypeScript-Transpiler wird von Microsoft als Open Source zur Verfügung gestellt. Am gängigsten ist die Variante als Node-Modul.

Die Installation eines solchen **TypeScript-Transpilers** ist im Prinzip einfach. Zunächst muss der Transpiler als Node-Modul global installiert werden. Dies setzt eine Installation von NodeJS voraus, um den Node Package Manager **NPM** zur Verfügung zu haben:

```
npm install -g typescript
```



```
Eingabeaufforderung
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\befrank>npm install -g typescript
C:\Users\befrank\AppData\Roaming\npm\tsc -> C:\Users\befrank\AppData\Roaming\npm\node_modules\typescript\bin\tsc
C:\Users\befrank\AppData\Roaming\npm\tsserver -> C:\Users\befrank\AppData\Roaming\npm\node_modules\typescript\bin\tsserver
C:\Users\befrank\AppData\Roaming\npm
└─ typescript@1.7.5

C:\Users\befrank>
```

Es gibt für den TypeScript-Transpiler keine explizite Update-Funktionalität. Für einen Upgrade installieren Sie einfach erneut, wie oben gezeigt.

### 3.1.1. Test der Installation

---

Ist der Transpiler installiert, so können Sie dies an der Kommandozeile testen, indem Sie seine *Version* abfragen:

```
tsc --version  
// -> Version 2.4.1
```

- ✓ Mit dieser Ausgabe meldet sich der TypeScript-Compiler der Version 2.4.1 (Stand Juli 2017),

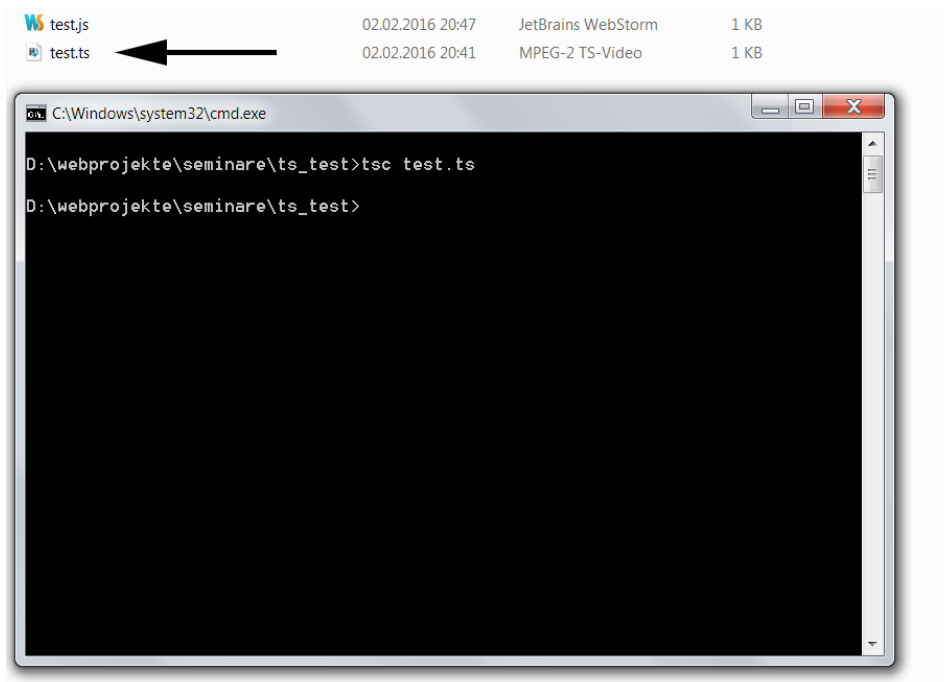
### 3.1.2. Kompilieren einer TypeScript-Source

---

Durch die **globale Installation** mit dem Flag `-g` kann anschließend in *jedem* Projektordner eine TypeScript-Datei angesprochen und nach JavaScript transpiliert werden:

```
tsc test.ts
```

Auch dies geschieht wieder über die **Kommandozeile**:



- ✓ Die Verarbeitung von *test.ts* erzeugt im gleichen Ordner eine Datei *test.js*.

Die TypeScript-Datei *test.ts* enthält in diesem Beispiel eine Funktionsdeklaration mit der neuen ECMA6-Syntax für Defaultparameter:

```
function addiere(a=0, b=0) {  
    return a + b;  
}
```

Der Compiler erzeugt eine Datei *test.js* (per Default in ECMA5), in welcher die ECMA6/TypeScript-Syntax auf folgende Weise umgesetzt ist:

```
function addiere(a, b) {  
    if (a === void 0) { a = 0; }  
    if (b === void 0) { b = 0; }  
    return a + b;  
}
```

- ✓ Da **Defaultwerte für Parameter** in ECMA5 nicht formulierbar sind, wird hier auf eine übliche (wenn auch unnötig komplizierte) Umschreibung gesetzt.

Grundsätzlich lassen sich (fast) alle in ECMA6 formulierbaren Strukturen auch in ECMA5 umsetzen (wenn dies auch gelegentlich sehr mühsam ist).

### 3.2. TypeScript in Webstorm

Die **Webstorm**-IDE besitzt einen eingebauten TypeScript-Compiler, der es ermöglicht, direkt beim Coden parallel ohne weiteres Zutun eine Javascript-Datei zu erzeugen, die dann auch gleich im Browser getestet werden kann.

- ✓ Die IDE besitzt ein TypeScript-Template, mit dem TypeScript-Dateien unmittelbar angelegt werden können.

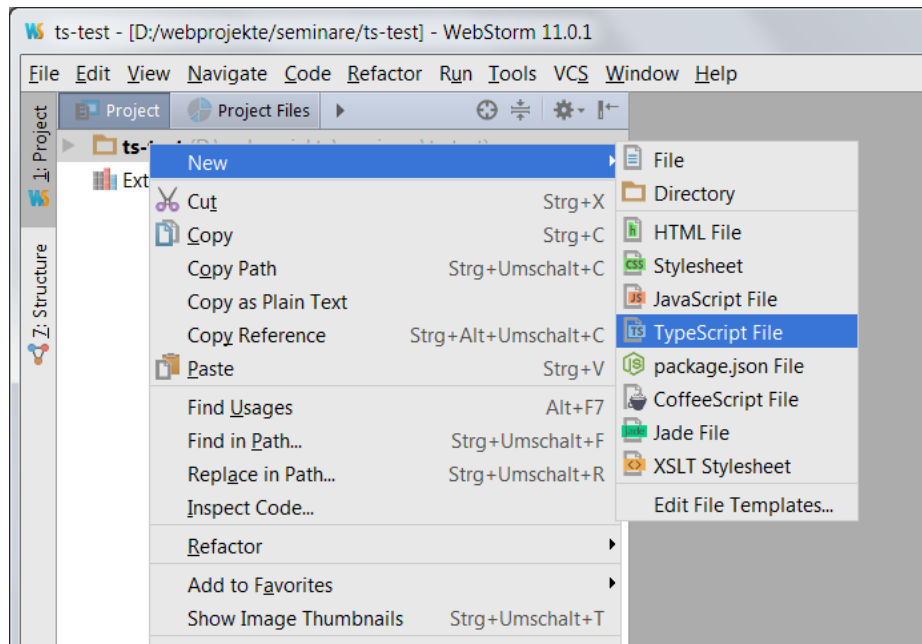
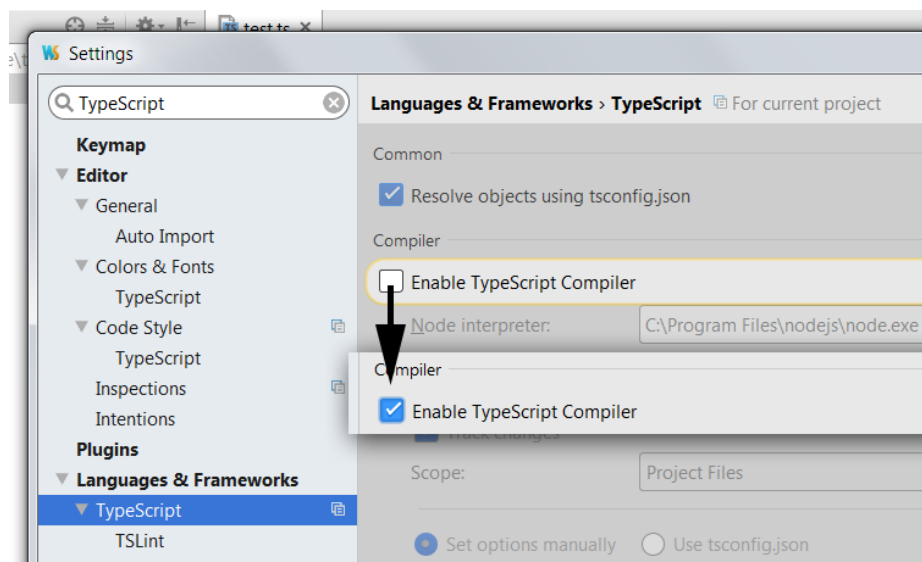


Abb.: Erzeugen einer TypeScript-Datei (Rechtsklick im Dateibaum)

### 3.2.1. Automatische TypeScript-Kompilierung

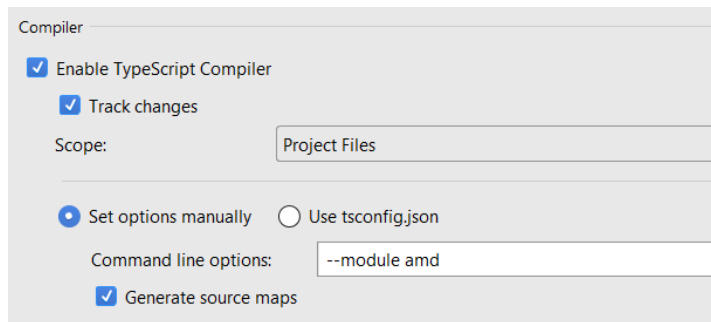
Um das **automatische** Kompilieren von TypeScript zu ermöglichen, muss die Option in den **Settings** zunächst aktiviert werden:



- ✓ Eine Installation von **NodeJS** ist zwingend erforderlich und *muss* Webstorm *bekannt* sein (Pfad angeben)!

Es ist sinnvoll, die **Option „Track Changes“** angewählt zu lassen. Der Compiler tritt dann automatisch in Aktion, wenn die TypeScript-Datei bearbeitet wird. Jede Änderung wird umgehend in der JS-Zieldatei nachvollzogen.

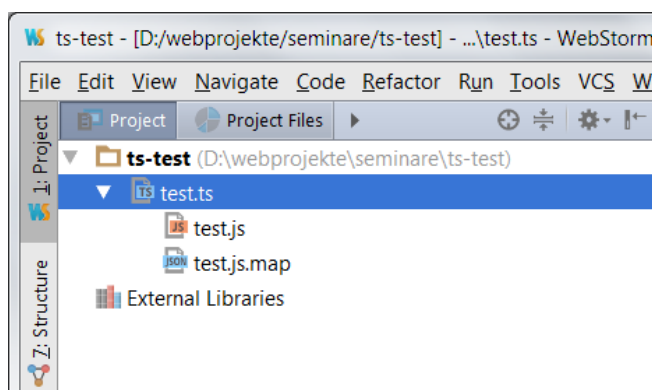
- ✓ Beachten Sie: Wenn Sie die Option „Track Changes“ (Checkbox) abwählen, muss der Compile-Vorgang händisch angestoßen werden.



Anschließend brauchen keine weiteren Maßnahmen getroffen zu werden; jede *.ts*-Datei wird *automatisch* nach JavaScript übersetzt, wobei jeweils zusätzlich eine **Sourcemap** *.map* angelegt wird.

- ✓ **Sourcemaps sind optional:** Wählen Sie die Option „Generate source maps“ in den Settings ab, wenn Sie keine Sourcemaps benötigen.

Die Compile-Dateien *test.js* und *test.js.map* sind der Source-Datei *test.ts* im Projektbaum optisch zugeordnet (sie liegen aber in Wirklichkeit parallel im Ordner; dies soll sie nur als von *test.ts* „abhängige Dateien“ kennzeichnen).



- ✓ Da Webstorm den regulären TypeScript-Compiler verwendet, ist das Ergebnis bei gleichem *test.ts* identisch zum Kommandozeilentool.

Hier wird aber im Kommentar zusätzlich auf die Map-Datei verwiesen:

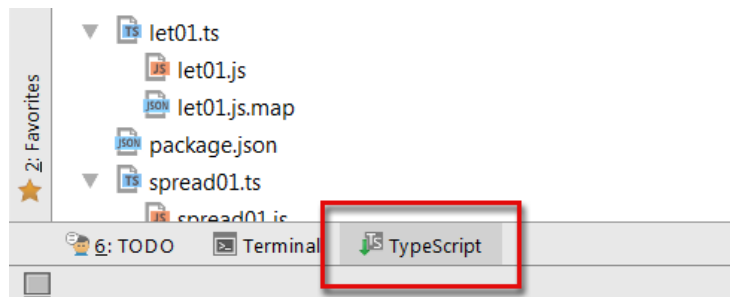
```
function addiere(a, b) {  
    if (a === void 0) { a = 0; }  
    if (b === void 0) { b = 0; }  
    return a + b;  
}  
//# sourceMappingURL=test.js.map
```

### 3.2.2. Händische TypeScript-Kompilierung

Um eine Kompilierung gezielt anzustoßen, steht einem geöffnetem `.ts`-File ein **TypeScript-Compiler-Fenster** zur Verfügung, in dem per Button `Compile All` das aktuelle `.ts`-File (oder alle des Projekts) neu kompiliert werden kann.

- ✓ Dies ist definitiv nötig, wenn Sie die „Track Changes“-Option in den Settings abgewählt haben sollte.

Um das Compiler-Fenster zu aktivieren, klicken Sie am unteren Rand der IDE auf den Button `TypeScript`.



Den Button `Compile All` finden Sie in der Icon-Leiste links im Fenster.

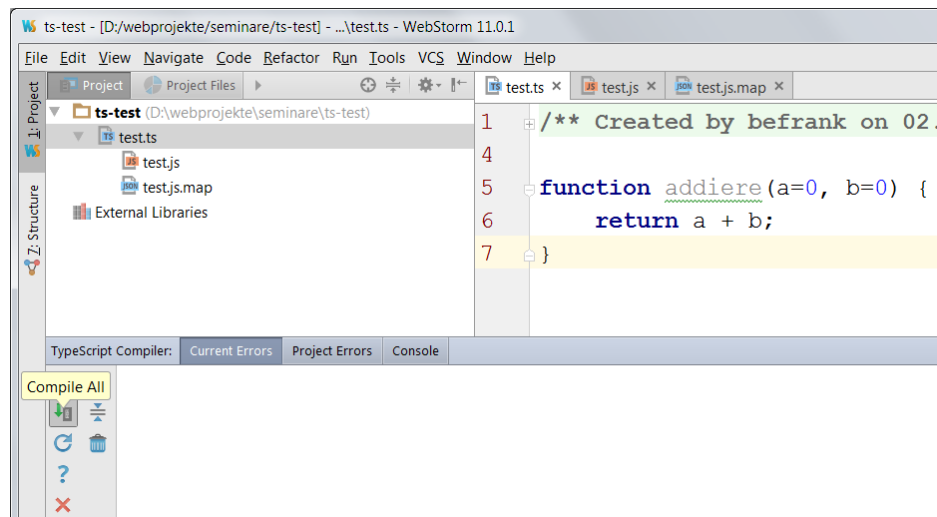


Abb.: TypeScript-Compiler-Panel in Webstorm

✓ **Achtung:** Wird eine erzeugte Javascript-Datei im Projektordner *verschoben*, verliert Webstorm sie aus dem Auge. **Änderungen im Sourcefile** werden *nicht* in der verschobenen Datei berücksichtigt. Stattdessen wird „vor Ort“ eine *neue* Javascript-Datei erstellt.

#### Wichtig:

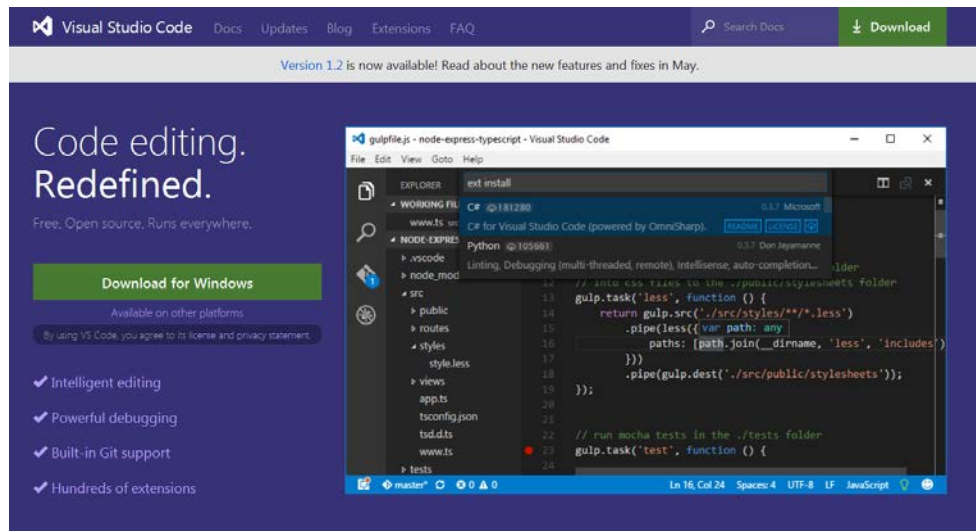
Selbstverständlich können *nur diejenigen* ECMA6-Features von TypeScript nach JavaScript übersetzt werden, die im jeweiligen TypeScript-Wortschatz enthalten sind. So konnten alle Syntaxfeatures, die mit **Generatoren** (also auch *Spreads* von *Iterables*, *for-of* auf *Iterables*) oder **Proxies** zusammenhängen, mit TypeScript 1.8 nicht umgesetzt werden.

### 3.3. TypeScript in Visual Studio Code

**Visual Studio Code** ist ein von Microsoft als Open Source in der Klasse von Sublime platzierter **Editor**, der, anders als die eher überfrachtete *Visual Studio IDE* nicht über eine Projektverwaltung verfügt, dafür aber sehr schlank und wenig ressourcenhungrig ist. Geboten wird jedoch **IntelliSense** sowie eine eingebaute **Git**-Unterstützung.

Eine Unterstützung für *JavaScript*, *TypeScript* und *NodeJS* ist von Haus aus vorhanden, weitere Features können als **Extension** nachgerüstet werden (beispielsweise Code-Linter für ECMA6 oder TypeScript).





✓ Download: <https://code.visualstudio.com/>

### 3.3.1. tsconfig.json

Zwar kann TypeScript in VC einzeln auf *Dateibasis* verarbeitet werden, doch wird empfohlen, dies **auf Projektbasis** zu tun, da dies die Möglichkeit bietet, innerhalb einer Datei namens **tsconfig.json** die Compiler-Optionen exakt festzulegen.

Am einfachsten ist es, diese Datei mittels des **Typescript-Compilers** anzulegen. Dies erledigen Sie über das Kommandozeilen-Tool, das Sie auf Ihr Projektverzeichnis richten.

Geben Sie dort folgenden Befehl ein, um eine *tsconfig.json* zu erzeugen:

```
tsc --init
```

Diese Datei wird in der Verzeichnismwurzel des Projekts angelegt, wo VC sie dann finden kann. Per Default hat die so generierte *tsconfig.json* folgende Form:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
```

```
        "sourceMap": false
      },
      "exclude": [
        "node_modules"
      ]
    }
  }
```

Das Property **"compilerOptions"** bezeichnet über die Option **"target"** ECMAScript 5 ("es5") per Default als die Zielversion des Compilervorgang. Alternativ sind die Angaben **'es3'** oder **'es6'** möglich.

Das **"exclude"**-Property unterbindet das das Kompilieren eventueller **.ts**-Dateien in den genannten Verzeichnissen. Alle anderen Verzeichnisse des Projekts werden verarbeitet.

```
{
  "compilerOptions": {
    "target": "es6",
    "outFile": "../build/app.js"
  },
  "exclude": [
    "node_modules"
  ]
}
```

Natürlich können beliebig andere Optionen gesetzt werden. Die folgende Datei nennt ECMA6 als Zielversion, lässt eine SourceMap-Datei erstellen und entfernt Kommentare aus den Ergebnisdateien. Zusätzlich wird mit **outFile** ein Dateiname und ein Pfad für die Ergebnisdatei angegeben.

Die zu kompilierenden Daten werden über das **"files"**-Property festgelegt (dieses ist alternativ zum **exclude**-Property einsetzbar). Fehlt dieses Property, so werden **alle** **.ts**-Dateien im Projektverzeichnis und dessen Unterverzeichnissen kompiliert, sofern kein **Exclude** ausgesprochen wird.

```
{
  "compilerOptions": {
    "target": "es6",
    "removeComments": true,
    "outFile": "../build/app.js",
    "sourceMap": true
  },
  "files": [
    "core.ts",
  ]
}
```

```

    "types.ts",
    "app.ts"
  ]
}

```

- ✓ Eine für *tsconfig.json* definierte Option "compileOnSave", die festlegt, dass beim Speichern einer *.ts*-Datei automatisch ein Compilevorgang angestoßen wird, wird gegenwärtig von Visual Studio Code **noch nicht unterstützt**.
- ✓ Alle **Optionen** für *tsconfig.json* finden Sie hier:  
<https://www.typescriptlang.org/docs/handbook/compiler-options.html>

### 3.3.2. Linter für TypeScript-Sourcen

Hier sehen Sie ein einfaches Beispiel einer TypeScript-Datei im Editor, und zwar eine TypeScript-Klasse (noch ohne Typannotationen für die Konstruktor-Parameter; wird später erweitert).

```

class Student {
    private name: string;
    constructor(public vorname, public nachname) {
        this.name = vorname + " " + nachname;
    }
}

var tom = new Student(5, 7); // nicht moniert!

```

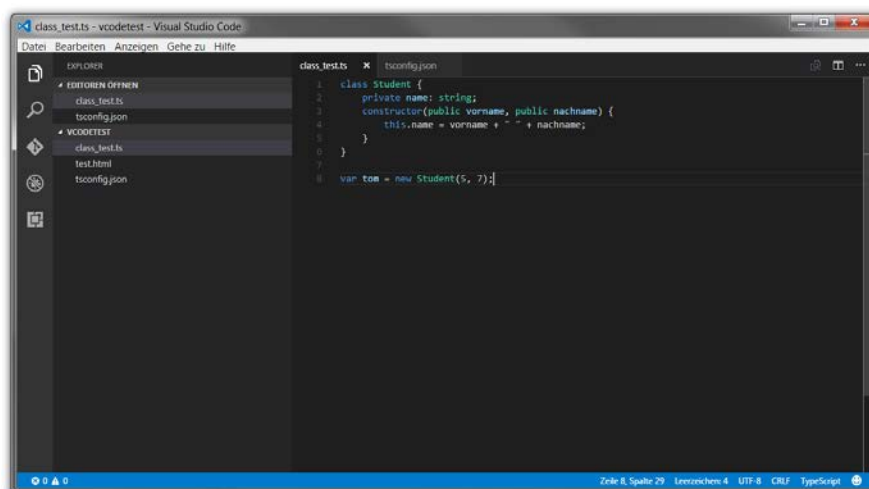


Abb.: Eine TS-Datei mit einer TypeScript-Klasse

Die Klasse wird nun mit *Typannotationen* für die Konstruktorparameter versehen, diese sollen stets Strings sein:

```
class Student {  
    private name: string;  
    constructor(public vorname:string, public nachname:string) {  
        this.name = vorname + " " + nachname;  
    }  
}
```

```
var tom = new Student(5, 7); // als Fehler erkannt!
```

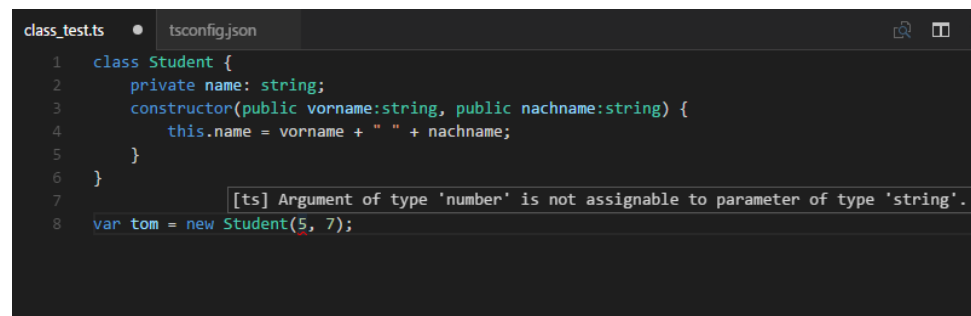


Abb.: TSLint moniert einen falschen Argument-Typ

Die falsch übergebenen Parameter werden nun *moniert*.

```
var tom = new Student("Tom", "Tester"); // korrigiert
```

### 3.3.3. JavaScript-Build über Task-Runner

Eine TypeScript-Datei kann in Visual Studio Code über einen **Build-Prozess** verarbeitet werden. Nachdem alle vom TypeScript-Linter monierten Fehler korrigiert sind, soll aus dem TypeScript eine JavaScript-Datei generiert werden.

Öffnen Sie, um dies einzuleiten **Anzeigen > Befehlspalette** und geben Sie ein: „**Configure Task Runner**“. Wählen Sie die Option **tsconfig.json**.

✓ VSC erstellt automatisch eine *tasks.json*.

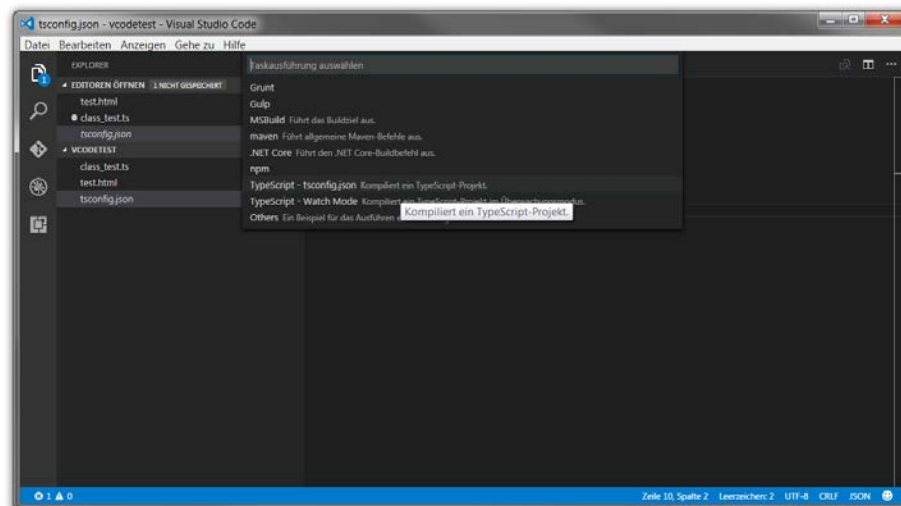


Abb.: Erstellen einer Build-Task anhand von tsconfig.json

### Shortcut für Build:

✓ Sie können einen Build nun mit **Strg+Shift+B** anstoßen.

Anhand der TypeScript-Datei wurde nun eine korrespondierende *JavaScript*-Datei generiert:

```
var Student = (function () {    function
Student(vorname, nachname) {
    this.vorname = vorname;
    this.nachname = nachname;
    this.name = vorname + " " + nachname;
}
    return Student;
})();
var tom = new Student("Tom", "Tester");
//# sourceMappingURL=class_test.js.map
```

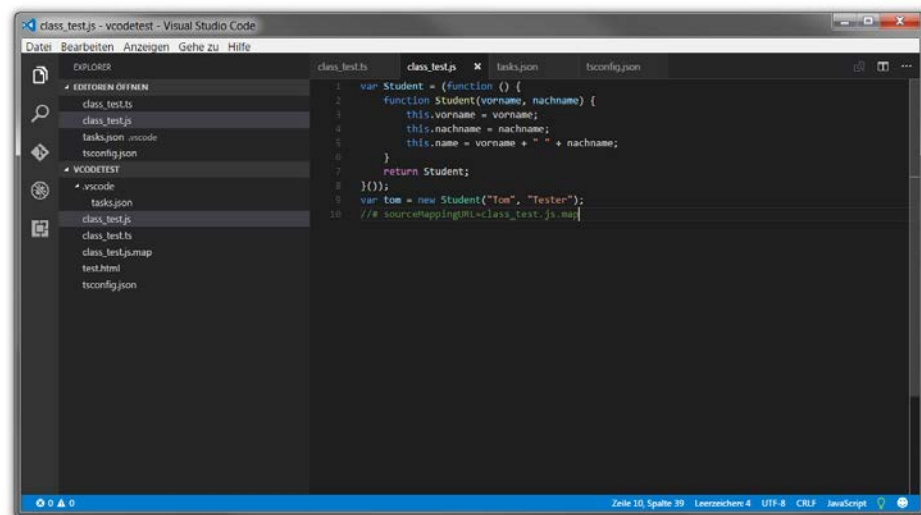


Abb.: Die generierte JavaScript-Datei im Editor.

### 3.3.4. JavaScript-Build über Terminal

Visual Studio Code verfügt (wie Webstorm) über eine eingebaute Eingabeaufforderung, die derjenigen des Betriebssystems äquivalent ist. VSC bezeichnet sie als **Terminal**.

- ✓ Sie öffnen das Terminal entweder über das Menü **Anzeigen > Integriertes Terminal** oder über das Tastenkürzel **Strg + ö**.

Das Terminal zeigt automatisch in das Projektverzeichnis, auf dem Visual Studio Code aktuell arbeitet. Sie können hier also den **TypeScript Compiler ansprechen**. Er arbeitet dann anhand der Konfiguration, die Sie (sinnvollerweise) im Projektverzeichnis als *tsconfig.json* angelegt haben.

Geben Sie einfach folgende Anweisung ein:

```
tsc
```

In diesem Falle wird der TSC alle TypeScript-Quellen im Ordner kompilieren und neben jeder *.ts*-Datei eine korrespondierende *.js*-Datei anlegen (bzw. diese aktualisieren).

- ✓ **Zur Erinnerung:** Über die *tsconfig.json* kann hierbei der **Ausschluss von Unterordnern** (wie *node\_modules*) vom Kompiliervorgang festgelegt, ein **Ausgabeordner** für die JavaScript-Ergebnisse angegeben (sofern diese nicht parallel zu den TS-Quellen abgelegt

werden sollen) oder die Erstellung von **SourceMaps** angewiesen werden.

Möchte man nicht nach *jedem* Edit-Vorgang die Neukompilierung der TS-Sourcen extra anweisen, so kann der TSC zum *Überwachen* seiner Sourcen veranlasst werden (dieser „**Watch-Prozess**“ wird durch die NodeJS-Umgebung übernommen). Geben Sie hierfür folgenden Befehl ein:

```
tsc --watch
```

- ✓ Ein Speichern einer geänderten TS-Source triggert nun die automatische Neuverarbeitung aller im Projekt befindlichen TS-Dateien (darunter natürlich auch die der geänderten).

## 4. Überblick über ECMAScript 6

ECMA6 (ECMA2015) setzt abwärtskompatibel auf ECMA5 auf, beinhaltet die ältere Sprachversion also (analog beinhaltet ECMA5 die Vorgängerversion ECMA3). Dasselbe gilt für ECMA7 (ECMA2016), das auf ECMA6 aufbaut.

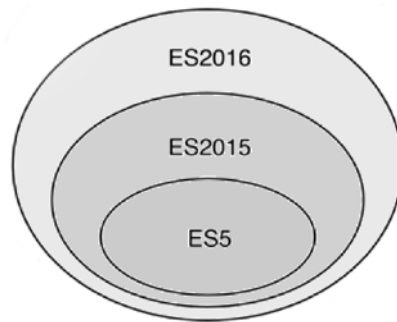


Abb.: Die „ECMA-Zwiebel“

Die in ECMAScript 6 gegenüber ECMAScript 5 erfolgten Sprach-erweiterungen sind sehr umfangreich und können im Rahmen des Seminars nicht in vollem Umfang vorgestellt werden. Hier soll deshalb ein allgemeiner Überblick versucht werden, bevor in die speziellen Abteilungen vertieft wird.

### 4.1. Syntaxerweiterungen

- Blockscope `let`
- Echte Konstanten `const`
- Spread-/ Restoperatoren
- Defaultparameter
- Destructuring
- Template Literale
- Symbol als "primitive value"
- Arrow-Funktionen ("Lambdas")
- Shorthand-Notation für Objektliterale
- `for-of`-Schleife



ECMA6 bietet desweiteren erstmals „echten“ **Blockscope** für Variable und "echte" **Konstante**, sowie **Defaultparameter** für Funktionen. **Spread-** und **Restoperatoren**, **Destrukturierung** sowie die neue `for-of`-Schleife erleichtern den Umgang mit komplexen Werten, **Fat-Arrow-Funktionen** adressieren das Kontextproblem verschachtelter Funktionen.

## 4.2. Code Organisation

---

- Iteratoren
- Generatoren
- Module
- Classes

Als wesentliche neue Bausteine für ES6-Programme seien hier **Generatoren** (für callbackfreie asynchrone Programmierung), **Module** (Modularisierung und Import/Export) und **Klassen** (einfachere Beschreibung von Konstruktoren) genannt.

## 4.3. Asynchrone Programmierung

---

- Promises
- Generatoren + Promises

Für das Gebiet der asynchronen Programmierung stehen **Promises** (Typ A) zur Verfügung. Auch die Generatoren werden hier einen Wirkungskreis finden.

## 4.4. Werte-Collections

---

- Typisierte Arrays
- Maps
- Weak Maps
- Sets
- Weak Sets

Als **neue komplexe Typen** werden Maps, Weak Maps, (sog.) typisierte Arrays und Sets eingeführt.

## 4.5. Mehr API für Basisobjekttypen

---

Erweiterungen der API gibt es für...

- Array
- Object
- Math
- Number
- String

Wir stoßen auf Neuerungen beim `Object`-Typ, wie die `assign`-Methode, mit der aufzählbare Instanzeigenschaften eines Objekts in ein anderes kopiert und somit in gewissem Rahmen **Cloning** oder **Merging** ermöglicht wird.

Zu `Object.keys()` gesellen sich die Methoden `getOwnPropertyNames()` und `getOwnPropertySymbols()`, die Einführung von **Symbolen** bietet Neuigkeiten bei der Erstellung von Properties. Alle Objekte besitzen nun `watch`- und `unwatch`-Methoden, die Callbacks an **Propertyänderungen** binden, bzw. dies lösen können.

## 4.6. Weitere Neuerungen

---

- Reflect-API
- Proxy

Im Bereich der Introspection sollte das **Reflect**-Objekt erwähnt werden, das **Proxy**-Objekt übernimmt Aufgaben, die dem (noch?) nicht verwirklichten `Object.observe()` zugedacht wurden.

# 5. Erweiterungen der Grundsyntax

---

Am Anfang soll es um einige der „kleineren“ Neuerungen gehen, die ECMA6/ECMA2015 dem Sprachumfang von ECMA 5 hinzufügt.

## 5.1. Blockscope mit `let`

---

In ECMA 5 kennt JavaScript lediglich zwei Arten von Scopes: den **globalen Scope** (lexikalischer Scope) und den **Funktionsscope** innerhalb des

Codeblocks einer Funktion. ECMA2015 schafft Abhilfe, indem der Deklaration von Variablen mit `var` eine alternative Deklaration mit dem `let`-Keyword zur Seite gestellt wird.

### 5.1.1. Recap: var-Keyword zur Deklaration von Variablen

Im Rahmen gewöhnlicher **Codeblocks** (alle Kontrollstrukturen) ist mit `var` kein eigener Scope (Blockscope) möglich, sondern es gilt der „current“ Scope (außerhalb von Funktionen stets der globale Scope).

So werden in Blöcken mit `var` deklarierte Variablen aus dem Block in den globalen Scope *gehoistet*. Innerhalb von Funktionen mit `var` deklarierte Variablen gelten dagegen dort lokal.

Dies hat unbequeme Folgen, was beispielsweise die Zählervariablen von Schleifen angeht:

```
for(var i = 0; i < 10; i++) {  
    // tu etwas zehn Mal  
}  
console.log(i); // 10
```

Die Zählervariable `i` ist, obwohl eigentlich nicht mehr benötigt, auch **nach** der Schleife noch existent. Sie wird sogar, obwohl im Schleifenkopf deklariert, aus dem Schleifenblock *gehoistet*, sodass der oben stehende Code von der Wirkung her eigentlich wie folgt lautet:

```
console.log(i); // undefined  
  
var i;  
for(i = 0; i < 10; i++) {  
    console.log("Tri tra trullala!");  
}  
console.log(i); // 10
```

**Hoisting:** Die Variable `i` ist *überall* im Scriptblock sichtbar. Allerdings ist sie *vor* ihrer Initialisierung im Schleifenblock noch `undefined`.

### 5.1.2. let-keyword zur Deklaration von Variablen

Da das `var`-Keyword aus Gründen der Abwärtskompatibilität nicht umgedeutet werden darf, wird in ECMA2015 ein neues Keyword `let` eingeführt, das Blockscope respektiert. Ein wie vor im Schleifenkopf deklarierter Zähler gilt dank `let` nur innerhalb des Schleifenblocks:

```
for(let j = 0; j < 10; j++) {  
    console.log("Obladi, oblada!");  
}  
// console.log(j); // ReferenceError: j is not defined
```

Die Variable wie vor *außerhalb* der Schleife zu deklarieren, lässt die die Schleife überleben. Achtung: dennoch ist sie *nicht* gleichermaßen global zugänglich, wie eine mit `var` deklarierte Variable:

```
// Temporal Dead Zone vor der Deklaration!!!  
// console.log(k); // ReferenceError:k is not defined
```

```
let k; // Initialisierung nicht erforderlich
```

```
for(k = 0; k < 10; k++) {  
    console.log("Yeah, yeah yeah!");  
}  
console.log(k); // 10
```

#### Kein Hoisting! Kein Zugriff vor Deklaration!

`let`-Variablen besitzen **Blockscope**. Darüberhinaus jedoch werden sie *nicht gehoistet*. Ihre Referenzierung in ihrem Scope *vor* der eigentlichen Deklaration verursacht daher einen **Referenzfehler**!! Dies wird als **TDZ** (Temporal Dead Zone) bezeichnet.

```
test = 'Fehler'  
// <- ReferenceError: test is not defined  
let test = 'Ein Test'
```

### 5.2. Konstanten mit const

Explizite Konstanten waren in ECMA5 nicht wirklich (bzw. nur auf Umwegen möglich). Man behalf sich mit kosmetischen Konventionen, wie

der **Großschreibung** von Symbolnamen, die eine Konstante benennen.  
Einen wirklichen Schutz gegen Manipulation bot dies jedoch nicht:

```
var MEINE_KONSTANTE = 42;  
console.log(MEINE_KONSTANTE); // 42;  
MEINE_KONSTANTE = 43; // Oops...  
console.log(MEINE_KONSTANTE); // 43;
```

Mit `var` deklarierte Symbole **können keine** echten Konstanten sein.

### 5.2.1. Das `const`-Keyword zur Deklaration von Konstanten

In ECMA2015 existiert nun ein `const`-Keyword, mit dem **echte Konstante** deklariert werden können, deren Wert nach Deklaration unveränderlich ist.

✓ Bei näherer Betrachtung ist `const` ein Cousin der `let`-Deklaration:

```
const echt_konstant = 17;  
console.log(echt_konstant); // 17  
  
echt_konstant = 21;           // TypeError:  
                               // invalid assignment  
                               // to const `echt_konstant`
```

Eine `const`-Konstante quittiert den Versuch, ihren Wert zu überschreiben, mit einem *Typfehler*.

Eine Konstante **muss** jedoch initialisiert werden, da sie *keinen impliziten Wert* (auch nicht `undefined`) besitzt:

```
const geht_nicht; // SyntaxError:  
                  // missing = in const declaration
```

Ebenso wie `let`-Variablen beachten `const`-Konstanten den Blockscope und werden nicht geholtet!

Wie von `let` bereits bekannt, beachtet auch `const` den Blockscope:

```
// eine globale Konstante
const myConst = 'aussen'
console.log(myConst)      // -> aussen
{
  // eine NEUE Konstante für diesen Block
  const myConst = 'innen'
  console.log(myConst)    // -> innen
}
// die globale Konstante von eben:
console.log(myConst)      // -> aussen
```

### 5.2.2. Nicht überschreibbare Funktionen

- ✓ Eine naheliegende Verwendung von Konstanten ist die Speicherung von Funktionen, die *nicht* neu definiert oder überschrieben werden sollen.

Überschreiben ist ansonsten ohne Widerstand möglich, wenn Funktionen (wie gewohnt) in `var`-Variablen (oder `let`-) abgelegt werden:

```
var ueberschreibbar = function() {
  // ich kann überschreiben werden
}

// Neu definieren? Kein Problem:
ueberschreibbar = function(){
  // ich habe die erste Funktion überschrieben
};
```

**Achtung:** Da eine *Function-Deklaration* einer `var`-Variablen entspricht, ist auch sie stets überschreibbar.

- ✓ Alternativ kann man aber eine **Konstante** mit einer *Function-Expression* füllen:

```
const nichtUeberschreibbar = function(){
  // ich bin nicht überschreibbar! Wirklich!
};

// Typfehler:
nichtueberschreibbar = function(){
```

```
    // Netter Versuch. Geht aber nicht. Typfehler!!  
};
```

Auf diesem Weg lässt sich sicherstellen, dass eine Funktion für die aktuelle Laufzeit unverändert bleibt.

### 5.3. Vergleich von Werten: „ultrastrikt“ mit Object.is()

---

Als neue statische Methode des Object-Typs kommt in ECMA6 eine Methode hinzu, die einen *noch strikteren* Vergleich zwischen Werten ermöglicht als der bereits geläufige Operator `===` (*triple equal*).

#### 5.3.1. Object.is()

---

Trotz des „strict equality“-Vergleiches gibt es Grauzonen, in denen Werte entweder als gleich betrachtet werden, ohne dass sie es tatsächlich sind, oder wo ein Vergleich prinzipiell gleicher Werte fehlschlägt, bzw. nur auf Umwegen vorzunehmen ist. Hier zwei typische Beispiele:

```
var plusNull = 0;  
var minusNull = -0;
```

Mathematisch betrachtet sind beide Werte *ungleich* (man betrachtet -0 als Annäherung vom negativen Zahlenstrahl her, +0 ist dasselbe von der anderen Richtung. Jedoch:

```
plusNull === minusNull; // fälschlich: true
```

Ähnliches ergibt sich, wenn zwei NaN-Werte miteinander verglichen werden, selbst wenn sie sich aus der gleichen Operation ergeben.

```
var nanEins = "dies"/"das";  
var nanZwei = "dies"/"das";
```

✓ Allerdings bewahrt der Wert NaN keine „Erinnerung“ an sein Zustandekommen auf, sodass die Operation unerheblich ist.

Nun aber gelten zwei NaN per Definition als „einander ungleich“, sodass der Vergleich auch beim besten Willen fehlschlagen wird.

```
nanEins === nanZwei; // fälschlich (?) false
```

Das gilt pikanterweise beim Vergleich eines NaN-Wertes mit sich selbst:

```
nanEins === nanEins; // höchst fälschlich false
```

Immerhin kann man mit Hilfe von `isNaN()` feststellen, ob es sich bei einem Wert um NaN handelt, sodass man mit etwas Mühe zum gewünschten Ergebnis kommt:

```
isNaN(nanEins) === isNaN(nanZwei); // true (?) ... naja  
isNaN(nanEins) === isNaN(nanEins); // true (o.k.)
```

Dies erfordert den Aufruf eines Wrappers, um den eigentlichen Vergleich durchzuführen.

✓ **Zwischenbemerkung:**

Die globale Utilitymethode `isNaN()` ist nun auch als *statische Methode* des `Number`-Objects verfügbar: `Number.isNaN()`. Dasselbe gilt für `parseInt()` und `parseFloat()`.

Mit Hilfe von `Object.is()` lassen sich beide Vergleiche erfolgreich und vor allem auf die gleiche Art und Weise durchführen:

```
Object.is(plusNull, minusNull); // false (!)  
  
Object.is(nanEins, nanZwei); // true  
Object.is(nanEins, nanEins); // true
```

✓ Hier ist anzumerken, dass `Object.is()` nicht als Ersatz (oder auch nur als Äquivalent) für *triple equal* gedacht ist, sondern nur für Grenzsituationen wie Negativ-Null oder NaN-Vergleich vorgesehen und angebracht ist.

## 5.4. Der Spread- und Rest-Operator

In ECMA2015 wird ein neuer Operator `...` eingeführt, der den Umgang mit Arraystrukturen vereinfacht und, gleichsam im Vorübergehen, auch das leidige Defizit des `argument`-Objekts (das ja bekanntlich kein Array ist) umgehen hilft.

✓ Der Operator `...` wird, je nach seinem Einsatz, als **Spread**- oder **Rest**-Operator bezeichnet.



### 5.4.1. Der Operator ``...`` als Spread-Operator

---

Vor ein Array gestellt, dient der Operator `...` als „vor Ort“-Dekonstruktor, also als **Spread-Operator**, der ein Array in Einzelwerte zergliedert („ausbreitet“):

Nehmen wir eine Funktion, die drei Werte addiert:

```
function addiere(a,b,c) {  
    console.log("Summe: ", a + b + c);  
}
```

Füttern wir diese Funktion mit drei **Zahlen**, geht alles gut:

```
addiere(3,4,5); // Summe: 12
```

Sobald man jedoch ein **Array** übergibt, kann keine Addition mehr stattfinden; `a` enthält das Array, die Parameter `b` und `c` sind *undefined*:

```
var werte = [1,2,3];  
addiere(werte); // Summe: 1,2,3undefinedundefined
```

Abhilfe in solchen Fällen gibt der **Spread-Operator**, der, vor ein Array gestellt, dieses in Einzelwerte auflöst:

```
addiere(...werte); // Summe: 6
```

Ein Spread ist auch in anderen Fällen nützlich. Ein bestehendes Array in ein neues Array zu integrieren, geht auf diese Weise nicht:

```
var mehrwerte;  
mehrwerte = [werte, 4, 5, 6]; // [[1,2,3],4,5,6]
```

Allenfalls eine Konkatenierung wäre ein gangbarer Weg:

```
mehrwerte = werte.concat([4,5,6]); // [1,2,3,4,5,6]
```

Um das zu integrierende Array als eine *Folge von Einzelwerten* einzufügen, ist der Spread-Operator durchaus nützlich:

```
mehrwerte = [...werte, 4, 5, 6]; // [1,2,3,4,5,6]
```

### 5.4.2. Der Operator `...` als Rest-Operator

Vor die Deklaration der letzten Parametervariable einer Funktion gestellt, wirkt der Operator `...` als **Rest-Operator**. Der so gekennzeichnete Parameter „nimmt“ sich alle Überschussargumente und stellt diese in der Funktion als **Array** zur Verfügung.

```
function addiere(a, b, ...mehr) {  
    console.log(typeof mehr); // object  
    console.log(mehr instanceof Array); // true  
}
```

✓ Dies kann als das Gegenteil von „Ausbreiten“ (*spread*) verstanden werden, da Einzelwerte in ein Array *zusammengefasst* werden

Das Array `mehr` bleibt leer, falls keine weiteren Werte übergeben werden.

In ECMA 5 muss man bei unbestimmter Parameterzahl in vielen Fällen zu Trickereien mit dem `arguments`-Objekt greifen. Da es sich bei `arguments` nicht um ein Array handelt, bietet es keine Array-Methoden:

```
function universaladdierer(){  
    // reduce-Methode von Array "ausborgen":  
    var summe = Array.prototype.reduce.call(arguments,  
        function(a,b){  
            return a + b;  
        });  
}
```

```
universaladdierer(1,2,3); // 6
```

*Direkt* geht das nämlich nicht. Dies hier ist **falsch**:

```
function universaladdierer2(){  
    // Fehler:  
    var summe = arguments.reduce(  
        function(a,b){  
            return a + b;  
        });  
    console.log(summe);  
}  
  
universaladdierer2(1,2,3);  
// TypeError: arguments.reduce is not a function
```

Auch Spread hilft bei `arguments` nicht unmittelbar weiter. Stattdessen erzeugt man mit dem **Rest-Operator** eine Alternative zum `arguments`-Objekt, die als Array in der Funktion verfügbar gemacht wird:

```
function universaladdierer3(...args){
    var summe = args.reduce(
        function(a,b){
            return a + b;
        });
    console.log(summe);
}
```

```
universaladdierer3(1,2,3); // 6
```

Der Spread-Operator funktioniert nur bei **iterierbaren Typen**, zu denen **Arrays** sehr wohl, *Objekte* jedoch *nicht* gehören (wie das `arguments`-Objekt; deshalb hilft `...arguments` nicht, um dies in ein Array umzuwandeln oder in Einzelwerte zu zergliedern).

**Alternative:** Eine Lösung des eben vorgestellten Problems ist auch mit der neuen statischen Array-Methode `Array.from()` möglich:

```
var echtesArray = Array.from(arguments);
```

Die Methode `Array.from()` konsumiert „arrayähnliche Objekte“ und gibt ein echtes Array zurück. Ob dieser Ansatz als einfacher zu betrachten ist, sei an dieser Stelle nicht entschieden.

**Seitenblick:** *Strings* gehören zu den iterierbaren Typen:

```
console.log([..."Hallo"]); // ["H", "a", "l", "l", "o"]
```

## 5.5. Dekonstruktion komplexer Werte

Häufig müssen aus **Objekten** oder **Arrays** einzelne Propertywerte oder Felder in Variablen ausgelagert werden, wobei im Falle von Objekten die Variablennamen gewöhnlich den Propertynamen gleichen sollen.

### 5.5.1. Händische Dekonstruktion komplexer Werte

---

Dies ist in ECMA 5 nur händisch machbar, also recht umständlich. Hier ein Beispiel für die Extraktion von Arrayfeldern in Einzelvariablen:

```
var arr = ["X", "Y", "Z"];

var x = arr[0],    // X
    y = arr[1],    // Y
    z = arr[2];    // Z
```

Analog gilt dies für Extraktion von Properties:

```
var obj = {
  a: 1,
  b: 2,
  c: 3
};

var a = obj.a,
    b = obj.b,
    c = obj.c;
```

### 5.5.2. Echte Dekonstruktion von Arrays und Objekten

---

In ECMA2015 ist nun eine *echte Dekonstruktion* („Zerlegung in Einzelwerte“) von Arrays und Objekten möglich. Die syntaktische Grundlage hierfür ähnelt der Literalschreibweise von Arrays und Objekten, nur dass sie auf der *LHS* angewendet wird.

Für Arrays:

```
var arr = ["X", "Y", "Z"];
var [x, y, z] = arr;
```

Für Objekte gibt es zwei Möglichkeiten der Dekonstruktion. Betrachten wir ein simples Objekt, dessen Properties als Variable extrahiert werden sollen:

```
var obj = {
  a: 1,
  b: 2,
  c: 3
};
```

Sollen die Variablennamen den Propertynamen **gleich**, so kann man ganz kurz schreiben:

```
// erzeugt die Variablen a, b, c:  
var {a, b, c} = obj;
```

Dies ist jedoch nur eine Verkürzung von

```
var {a: a, b: b, c: c} = obj;
```

**Achtung:** Bei der Dekonstruktion gilt `objProp:varName`.  
Der Bezeichner *rechts* vom Doppelpunkt ist also der *Targetbezeichner*!

Mit der vollständigen Schreibweise können Objektproperties a, b, c folglich auch unter **frei wählbaren Bezeichnern**, wie `meinA`, `meinB`, `meinC`, nach draussen abgebildet werden.

```
// erzeugt die Variablen meinA, meinB, meinC  
var {a: meinA, b: meinB, c: meinC} = obj;
```

### 5.5.3. Partielle Dekonstruktion von Objekten

---

Man ist dabei nicht darauf angewiesen, ein Objekt **komplett** zu zerlegen, indem für *jedes* Property eine Variable angelegt wird. Möchte man nur *bestimmte* Properties herausziehen, so genügt:

```
var obj = {  
  a: 1,  
  b: 2,  
  c: 3  
};  
  
// a -> y, c -> z:  
var {a:y, c:z} = obj;  
console.log(y, z); // 1, 3
```

oder, ohne Name-Mapping:

```
var {a, c} = obj  
console.log(a, c); // 1, 3
```

## 5.6. Defaultparameter für Funktionen

In ECMA2015 erhält man als neue Syntaxmöglichkeit das Setzen von **Defaultwerten** für Funktionsparameter, was in ECMA 5 noch nicht möglich ist, bzw. nur mit kleinen syntaktischen Tricks „gezaubert“ wurde

### 5.6.1. Die Situation in ECMA5

Die Abwesenheit von Defaultwerten kann Verdruss verursachen, wenn zu wenige Argumente übergeben werden, deren Parametervariablen in der Funktion dann *undefined* sind.

```
function addiere(a, b) {  
    console.log(a + b);  
}  
  
addiere(2,3);           // 5  
addiere(4);            // NaN (b ist undefined -> NaN)  
addiere();             // NaN (a und b sind undefined)  
addiere(null, 5);      // 5 (null wird nach 0 konvertiert)
```

Dies kann man beheben, indem man dem Parameter einen Wert zuweist, falls der Eingabewert „falsy“ ist (z.B. *undefined*):

```
function addiere1(a, b) {  
    a = a || 0; // ist etwas in a oder nicht??  
    b = b || 0; // ist etwas in b oder nicht??  
    console.log(a + b);  
}  
  
addiere1(2,3);          // 5  
addiere1(4);            // 4 (b auf 0 gesetzt)  
addiere1();             // 0 (a und b auf 0 gesetzt)  
addiere1(null, 5);      // 5 (a auf 0 gesetzt)
```

Es ist zu beachten, dass bei diesem Konstrukt eine sinnvolle Substitution eines fehlenden (*undefined*, daher „falsy“) Werts nur durch einen ebenfalls „falsy“ Wert wie 0 erfolgen kann.

### 5.6.2. Echte Defaultparameter in ECMA6

---

In ECMA2015 existiert nun die Möglichkeit, einen echten Defaultwert für einen Parameter zu definieren. Dieser wird dem entsprechenden Parameter einfach in der Funktionsdeklaration **zugewiesen**.

- ✓ Man nennt dies eine *Default Value Expression*. Es darf **jeder beliebige Ausdruck** verwendet werden, der einen Wert darstellt oder zurückgibt, also ein *Literal* (wie hier), aber auch ein *Funktionsaufruf*. Die Expression wird „lazy“ ausgewertet, also nur „bei Bedarf“.

```
function addiere2(a=0, b=0) {  
    console.log(a + b);  
}  
  
addiere2(2,3);      // 5  
addiere2(4);        // 4  
addiere2();         // 0  
addiere2(null, 5);  // 5
```

Beachten Sie, dass der übergebene Wert `null` **nicht**, wie im vorangegangenen Beispiel durch den Defaultwert 0 **substituiert** wird. Da er einen *gültigen Wert* darstellt, kann er „passieren“. Er wird dann jedoch im Lauf der Addition in die Zahl 0 umgewandelt (wie im ersten Beispiel).

## 5.7. Arrow-Funktionen

---

Bevor wir zu **Arrow-Funktionen** kommen, soll zunächst ein kurzer Rückblick auf „normale“ Funktionen geworfen werden, wie sie mit dem `function`-Keyword gebildet werden.

### 5.7.1. Das `function`-Keyword und das `thisObj`

---

In JavaScript werden **Funktionen** mit dem `function`-Keyword erzeugt bzw. deklariert (die Verwendung des `Function`-Konstruktors ist möglich, aber unüblich).

```
// Deklaration:  
function beispiel(a, b) {
```

```
    return a + b;
  }

  // Expression (RHS):
  var beispiel = function(a, b) {
    return a + b;
  };
```

Beide obigen Konstrukte sind gleichwertig. Der in Prinzip einzige Unterschied zwischen einer Deklaration und einer Expression ist:

- ✓ Eine function-**Deklaration** *muss* einen Namen besitzen.
- ✓ Eine function-**Expression** *darf* einen Namen besitzen (*muss* es aber nicht).

Eine *Expression* ist üblicherweise ein **RHS-Ausdruck** und wird in Zusammenhang mit Zuweisungen verwendet, oder wenn anonyme Funktionen übergeben werden. Namen sind jedoch gestattet:

```
// Expression (RHS):
var beispiel = function ganzAndererName(a, b) {
  return a + b;
};
```

- ✓ *Aufgerufen* werden kann die obige Funktion nur mit `beispiel()`. Intern „kennt“ sie sich jedoch als „*ganzAndererName*“ (dies ist der `function.name`, der jedoch nicht über diesen Bezeichner lesbar ist). Dies ist für rekursiven Aufruf nutzbar.

**Seitenblick:** Ein bekannter Einsatz der Function-Expression ist die **IIFE**:

```
// im Prinzip ein RHS-Ausdruck in der Klammer:
(function(){
  console.log("Ich werde immediately invoked.");
})();
```

Ebenfalls ein RHS-Ausdruck ist die direkte Übergabe einer anonymen Funktion, wie in einer Higher-Order-Method eines Arrays:



```
[1,2,3].forEach(function(val){  
    console.log(val)  
});
```

Auch hier *darf* die Funktion benannt werden (im Debugger sichtbar):

```
[1,2,3].forEach(function iterator(val){  
    console.log(val)  
});
```

Ob Expression oder Deklaration der Funktion zugrundeliegen, ist bei ihrem Gebrauch gleichgültig:

**Kontext:** Beim Aufruf der Funktion wird stets ein *thisObj* übergeben, das den Kontext der Funktion bestimmt. Das *thisObj* ist über den *impliziten Parameter* *this* im Inneren der Funktion zugänglich. Was als *thisObj* übergeben wird, hängt davon ab, ob das Script (oder die betreffende Funktion) im „strict Mode“ oder im „normal Mode“ ausgeführt wird.

**Argumente:** Alle der Funktion übergebenen Werte sind über eventuell vereinbarte Parametervariablen (im Beispiel a und b), stets aber über den *impliziten Parameter* *arguments* (ein arrayähnliches Objekt) zugänglich.

✓ Man nennt *this* und *arguments* die beiden „impliziten“ Parameter einer JavaScript-Funktion, weil ihre Bezeichner im Funktionsblock ohne Vereinbarung definiert und mit Werten belegt sind.

```
function beispiel(a, b) {  
    console.log(this, arguments);  
    return a + b;  
}
```

Ein Aufruf im **normalen Modus** loggt *window* als *thisObj* und *[4,5]* als *arguments*-Objekt aus:

```
beispiel(4, 5); // window, [4,5]
```

```
function beispiel(a, b) {  
    "use strict";  
    console.log(this, arguments);  
    return a + b;  
}
```

Ein Aufruf im **strict Mode** loggt *undefined* als *thisObj* und *[4,5]* als *arguments*-Objekt aus.

```
beispiel(4, 5); // undefined, [4,5]
```

Wird die Funktion (auch in der strikten Laufzeit!) jedoch **explizit** als *Methode* des window-Objekts aufgerufen, so ist das *thisObj* stets window:

```
window.beispiel(4, 5); // window, [4,5]
```

Auch in anderen (nicht strikten) Fällen ist das *thisObj* (vielleicht unvermutet) window, beispielsweise für „innere“ Funktionen einer Objektmethode:

```
var obj = {
  x: "X-Prop",
  meth: function() {
    console.log("Gibt x von ", this, " aus!");
    function xAusgeben() {
      console.log("x: ", this.x); // klappt nicht!
    }
    xAusgeben(); // x: undefined
  }
}
```

In diesem (etwas konstruierten) Beispiel schlägt der Zugriff auf das Objekt-Property `x` fehl, weil die innere Funktion **eben nicht** auf das `this` ihrer **lexikalischen Umgebung** zugreift (was `obj` wäre), sondern ein eigenes *thisObj* erhält (per Definition ist dies `windows`).

Um das *thisObj* einer Funktion **explizit** bestimmen zu können, existieren die Methoden des Function-Objekts `call`, `apply` und `bind`. Die Methode `call` löst das Problem durch „Hineinreichen“ eines expliziten *thisObj* in die Funktion:

```
var obj = {
  x: "X-Prop",
  meth: function() {
    console.log("Gibt x von ", this, " aus!");
    function xAusgeben() {
      console.log("x: ", this.x); // klappt nicht!
    }
    xAusgeben.call(this); // x: X-Prop
  }
}
```

### 5.7.2. Anonyme Funktionen

---

Anonyme Funktionen werden gerne verwendet, wo Funktionen als Literal übergeben werden müssen. Dies ist in **Higher-Order Functions** der Fall, wie sie beispielsweise in Form einiger Array-Methoden vorliegen.

```
[1, 2, 3].map(function (num) {  
    return num * 2  
});  
// -> [2, 4, 6]
```

Prinzipiell sind anonyme Funktionen auch als RHS-Statements bei Zuweisungen von Funktionen an Variablen oder Properties gebräuchlich:

```
var hallo = function(){  
    console.log("Hallo!");  
}
```

**Anmerkung:** Für Debuggingzwecke ist es oftmals günstiger, einer Funktion einen Namen zu geben, um sie auf dem Callstack identifizieren zu können:

```
[1, 2, 3].map(function quadrieren(num) {  
    return num * 2  
});
```

Hier erhält die ursprünglich anonyme Callbackfunktion des Map-Prozesses einen Namen. Für den Aufruf wird er nicht benötigt.

### 5.7.3. Die Arrow Funktion

---

Die in ECMA6 eingeführten **Fat-Arrow**-Funktionen verfolgen zwei Ziele. Zum einen ist die Syntax kürzer.

```
x => x*2 // multipliziert ein übergebenes x mit 2
```

...anstelle von

```
function mal2(x) {  
    return x*2  
}
```

Es wird kein `function`-Statement zur Formulierung benötigt und für einzeilige Anweisungsblöcke können zusätzlich die Blockklammern entfallen. Ein `Return`-Statement ist dann ebenfalls obsolet.

Betrachten wir das vorhin gegebene Beispiel eines Arraymappings erneut:

```
// Anm.: Die Funktion dürfte einen Namen erhalten
[1, 2, 3].map(function (num) {
    return num * 2
});
// -> [2, 4, 6]
```

...so kann dies mit Arrow-Funktion kürzer geschrieben werden:

```
// Anm: die Arrow-Funktion KANN KEINEN Namen erhalten
[1, 2, 3].map( num => num * 2 );
// -> [2, 4, 6]
```

- ✓ Arrow-Funktionen sind jedoch grundsätzlich **anonym** und es existiert, da es sich nicht um Deklarationen handelt, auch kein Weg, ihnen *direkt* einen Namen zu geben. Das Mapping mit `function` böte die Möglichkeit einer Benennung für Debuggingzwecke.

Man kann jedoch eine Arrow-Funktion als **RHS-Ausdruck** einsetzen:

```
var mal2 = x => x*2;
```

## Die Arrow-Funktion und this

Die andere Besonderheit gegenüber mit `function` formulierten Funktionen ist, dass einer Fat-Arrow-Funktion beim Aufruf kein *thisObj* übergeben wird, sondern diese sich (lexikalisch) das *this* der aktuellen Umgebung nimmt!

Betrachten wir das im Vorfeld gegebene **Kontext-Problem** mit der „inneren Funktion“ einer Objektmethode, das mittels `call()` gelöst wurde, so lässt sich dasselbe **noch einfacher** mit einer Arrow-Funktion erreichen:

```
var obj = {
    x: "X-Prop",
    meth: function() {
        console.log("Gibt x von ", this, " aus!");
        var xAusgeben = () => {
            // dasselbe this, wie aussen!
            console.log("x: ", this.x); // klappt!
        }
    }
}
```

```
    }  
    // Sensation:  
    xAusgeben(); // x: X-Prop  
  }  
}
```

Bei Arrow-Funktionen kann auf das Speichern des Kontexts oder eine explizite Übergabe mit `call()` oder `apply()` verzichtet werden.

Das ist aber kein dynamischer, sondern noch immer ein *lexikalischer* Scope. Dies bedeutet, dass eine Arrow-Funktion im Rahmen ihrer Entstehungsumgebung arbeitet, wie eine „herkömmliche Funktion“ auch.

Betrachten wir hierfür kurz das Beispiel zu „Beweis“ des lexikalischen Scopings:

```
var a = "Ein A";  
  
function aLesen() {  
  console.log("function:", a);  
}  
  
aLesen();           // -> Ein A (der Trivialfall)  
  
var aLesenArrow = () => console.log("Arrow:", a);  
  
aLesenArrow();      // -> Ein A (der Trivialfall)  
  
function aLesenVerwenden() {  
  // Verdeckung des globalen a durch lokales a erwartet  
  var a = "Lokales A";  
  
  console.log("Einsatz beider Funktion:");  
  
  aLesen(); // KEINE Verdeckung  
  aLesenArrow(); // KEINE Verdeckung  
}  
aLesenVerwenden();
```

- ✓ Auch eine Fat-Arrow-Funktion ist **lexikalisch** an den Scope ihrer Erstellung gebunden. *Ebenfalls* lexikalisch gebunden ist aber auch das zu diesem Zeitpunkt gültige *thisObj*.

Im **DOM-Umfeld** kann die Arrow-Funktion daher eine Erleichterung sein, da man sich das *Binden* eines *thisObj* mit `bind()` oder eine *Kontextverschiebung* mit `call()` oder `apply()` beim Aufruf sparen kann:

```
var p1 = document.getElementById('p1');

p1.addEventListener('click', function(){

    function werBinIch() {
        console.log("Function:", this);
    }

    var boundWerBinIch = (function(){
        console.log("Bound Function:", this);
    }).bind(this);

    var arrowWerBinIch = () => console.log("Arrow:", this);

    werBinIch();                // window (Ooops!!)
    werBinIch.call(this);       // p1
    boundWerBinIch();           // p1
    arrowWerBinIch();           // p1

});
```

Ein anderes Beispiel ist ein Callback, der aus dem Kontext eines Objekt hinaus an ein anderes Objekt gebunden wird. Nehmen wir als Vorstellung ein Objekt `controller` mit einer Methode `makeRequest()`, die durch einen Button getriggert werden soll.

Es existiere eine Setup-Funktion `setupRequest()`, die ein Buttonelement entgegennimmt und die Eventbindung vornimmt:

```
var controller = {
    makeRequest: function() {
        // Request absetzen...
    },
    setupRequest:function(btn) {
        // Kontexthack nötig:
        var self = this;
        btn.addEventListener("click", function() {
            // ..
            self.makeRequest(..);
        }, false);
    }
};
```

Zwar arbeitet `setupRequest()` auf dem `controller`-Objekt, jedoch nicht mehr die **Callbackfunktion** zum Zeitpunkt des Buttonklicks! Da sie ein neues *thisObj* übergeben bekommt, ist *this* dort nicht verwendbar, sondern muss durch ein gespeichertes `self` ersetzt werden.

Mit einer **Arrow-Funktion** als Callback ist dies nicht mehr erforderlich:

```
var controller = {
  makeRequest: function() {
    // Request absetzen...
  },
  setupRequest: function(btn) {
    // Kein Kontexthack nötig:
    btn.addEventListener("click", () => {
      // ..
      this.makeRequest(..);
    }, false);
  }
};
```

## Argumente

Nicht notwendigerweise benötigt eine Arrow-Funktion einen Eingabewert.

```
var f1 = () => 12;
```

Diese Arrow-Funktion erwartet *kein Argument*. In diesem Fall müssen *leere Parameterklammern* geschrieben werden.

- ✓ Die Arrow-Funktion `f1` gibt konstant 12 als Wert zurück. Wir brauchen dafür kein `return`-Statement. Einzeilige Codeblöcke können ohne Blockklammern geschrieben werden.

Der Fall eines einzelnen Eingabeparameter ist ebenso einfach:

```
var f2 = x => x * 2;
```

Diese Arrow-Funktion erwartet ein Argument, das sie intern als `x` kennen wird. Statt eines konstanten Rückgabewerts wird ein Ausdruck definiert.

- ✓ Ein **Einzelparameter** wie in `f2` braucht nicht geklammert zu werden. Im Codeblock wird er als lokale Variable behandelt. Noch immer einzeilig.

Es sind aber auch mehrere Argumente und Codeblöcke möglich:

```
var f3 = (x, y) => {  
    var z = x * 2 + y;  
    y++;  
    x *= 3;  
    return (x + y + z) / 2;  
};
```

Die Arrow-Funktion `f3` erwartet zwei Parameter `x` und `y`, die deshalb geklammert werden müssen. Da hier zudem ein Codeblock aus mehreren Zeilen benötigt wird *und* die Funktion einen Wert zurückgeben soll, werden jetzt auch die Codeklammern und ein `return`-Statement benötigt.



### Bad practise: Arrow um jeden Preis

Da der „Gewinn“ allein durch die kürzere Syntax oftmals gering ist, und die Arrow-Schreibweise nicht per se übersichtlich ist, sollte man erwägen, ein **gewöhnliches Function-Statement** einzusetzen, solange der Aspekt des lexikalischen *thisObj* irrelevant ist.

## Rückgabewert

Der Ausdruck *rechts* des Fat Arrow wird automatisch als **Rückgabewert** der Funktion verstanden:

```
[1, 2, 3].map( num => num * 2 );  
// -> [2, 4, 6]
```

Der Ausdruck kann ohne Weiteres komplexer werden, ohne dass dies missverständlich wird:

```
[1, 2, 3, 4].map((num, index) => num * 2 + index)  
// -> [2, 5, 8, 11]
```

Soll der Rückgabewert jedoch ein **Objekt** sein, so wird die Lage etwas schwieriger, was daran liegt dass die geschweiften Klammern sowohl als Objektbegrenzer wie auch als Blockklammern verwendet werden.

Nehmen wir an, aus einer Zahlenliste `[1, 2, 3]` soll per Mapping eine Liste aus **Objekten** der Form `[{zahl:1},{zahl:2},{zahl:3}]` erzeugt werden.

Ein erster, naiver, Ansatz führt *nicht* zur erwarteten Ausgabe:

```
[1, 2, 3].map(n => { zahl: n })  
// -> [undefined, undefined, undefined]
```



Dies liegt daran, dass die als Objektbegrenzer *gemeinten* Klammern als Blockklammern *interpretiert* werden. Die simple Lösung besteht im **Klammern** des als Rückgabewert gewünschten Objekts:

```
[1, 2, 3].map(n => ({ zahl: n }))  
// -> [{zahl:1},{zahl:2},{zahl:3}]
```

## Arguments-Objekt

Einer „herkömmlichen“ Funktion wird neben einem *thisObj* auch stets ein `arguments`-Objekt übergeben. Im Falle einer Arrow-Funktion wird **beides** dem lexikalischen Kontext entnommen.

- ✓ Achtung: Das bedeutet jedoch, dass in einer Arrow-Funktion *kein* `arguments`-Keyword existiert, *solange* die Arrow-Funktion nicht im *Kontext* eines Function-Scopes erzeugt wird!

Dies ist einfach zu zeigen:

```
var argTest = () => console.log(arguments);  
argTest(); // Referenzfehler
```

...wohingegen:

```
var testAussen = function(){  
  
    var argTest = () => console.log(arguments);  
    argTest(); // -> [2, 3, 4]  
  
}  
testAussen(2, 3, 4)
```

Eine Arrow-Funktion hat direkten Zugriff auf `arguments` einer umgebenden Funktion (braucht also keine Parameternamen zu kennen!).

- ✓ **Folgerung:**  
Eine Arrow-Funktion erspart einem den Aufruf von `apply()`, bei dem *this*-Kontext (für Arrow nicht erforderlich) und `arguments` (für Arrow ebenfalls nicht erforderlich) explizit übergeben werden.

## 5.8. Strings und Template-Literale

---

In Javascript sind Zeichenketten (Strings) primitive Werte. Obwohl es einen Konstruktor `String()` gibt, werden Strings üblicherweise als Literale verwendet. Stringlitterale werden durch Begrenzersymbole bestimmt, bei denen es sich (gleichwertig) um das einfache und das doppelte Anführungszeichen handelt.

Bisher galt:

- ✓ Strings sind atomar, d.h. nicht unterteilbar. Sie können daher keine Variablen enthalten, da diese dort nicht erkannt werden. In JavaScript existiert (anders als beispielsweise das `$`-Zeichen in PHP) kein Erkennungssymbol für Variablen, um diese innerhalb von Strings hervorheben zu können.
- ✓ Strings sind „per se“ einzeilig, können sich also nicht (als *multiline strings*) über mehrere Codezeilen erstrecken. Auszugebende Umbruchzeichen (und auch andere, „problematische“ Zeichen) können aber durch einen vorangestellten Backslash maskiert werden.

```
var beispiel1 = "Dies ist ein String";
```

```
var beispiel2 = 'Dies ist auch ein String';
```

In ECMA6 wird nun auch der **Backtick** ``` als Stringbegrenzersymbol definiert.

```
var beispiel3 = `Ich bin ein String in ECMA6`;
```

### 5.8.1. Echte Multiline Strings mit Backticks

---

Das Neue an Backtick-Strings ist, dass sie sich über **mehrere Codezeilen** erstrecken können:

```
var langesBeispiel = `  
  Das ist ein String,  
  der über mehrere Zeilen  
  geht.  
`;
```

Dies erleichtert die bisherige Praxis im Umgang mit HTML-Code, der in Variablen gespeichert werden musste. Bislang musste ein solches Gebilde in eine Reihe von verketteten Einzelstrings aufgelöst werden, um eine sinnvolle Zeilenlänge zu erreichen:

```
var myForm = '<form action="#">' +  
            '<input type="text" name=\'test\'>' +  
            '<input type="submit" ' +  
            ' value="Abschicken">' +  
            '</form>';
```

Dies ist unpraktisch und bedeutet, dass nicht einfach kopierter HTML-Quellcode einer Variablen zugewiesen werden kann. Zudem müssen Anführungszeichen im Inneren des Strings eventuell maskiert werden, wenn sie den Begrenzerzeichen entsprechen.

Mit Backticks geht dies nun einfacher:

```
var myBacktickForm = `  
  <form action="#">  
    <input type="text" name='test'>  
    <input type="submit" value="Abschicken">  
  </form>  
`;
```

- ✓ Da nun auch beliebige Anführungszeichen unmittelbar im String gestattet sind, braucht man sich keine weiteren Gedanken zu machen, in welcher Form Attributwerte gequotet werden.

### 5.8.2. Templating in Backtick-Strings

---

Außerdem können Backtick-Strings auch **Ausdrücke** enthalten, die bei Verwendung des Strings ausgeführt (interpoliert) werden. Um den Ausdruck zu markieren, muss dieser mit der Zeichenfolge `${ ... }` eingefasst werden:

```
var vorname = "Peter";  
var halloString = `Hallo, mein Name ist ${vorname}!`;   
console.log(halloString);  
// -> Hallo, mein Name ist Peter!
```

Im einfachsten Fall handelt es sich bei dem Ausdruck (wie oben) um eine Variablenreferenz. Es kann jedoch *jeder beliebige* Ausdruck übergeben werden:

```
var addiere = `1 + 1 = ${1 + 1}`;  
console.log(addiere); // -> 1 + 1 = 2  
  
var heute = `  
    Heute ist der  
    ${new Date().toLocaleString()}  
`;  
console.log(heute);  
//-> Heute ist der 21.8.2016, 23:24:40.
```

Dies ist auch nützlich bei der Instanziierung von HTML-Templates:

```
var article = {  
    titel: 'Template Literale sind toll!',  
    teaser: 'Mit Backtick-Strings ist vieles möglich...',  
    text: 'Multiline-Strings, Interpolation und mehr!',  
}  
  
var {title,teaser,body} = article;  
  
var htmlString = `  
<article>  
    <header>  
        <h1>${titel}</h1>  
    </header>  
    <section>  
        <div>${teaser}</div>  
        <div>${text}</div>  
    </section>  
</article>  
`;
```

Dies ergibt wie erwartet:

```
<article>  
    <header>  
        <h1>Template Literale sind toll!</h1>  
    </header>  
    <section>  
        <div>Mit Backtick-Strings ist vieles möglich...</div>  
        <div>Multiline-Strings, Interpolation und mehr!</div>
```

```
</section>
</article>
```

## 6. Objektorientierung und Vererbung

---

Wo die im Rahmen von ECMA5 (im Jahr 2009) für Objekte und den `Object`-Konstruktor hinzugekommenen Erweiterungen nun „gefühl“ die Programmierer endlich weitgehend erreicht haben, legt ECMA6 wiederum mit ein paar Neuerungen nach.

Betrachten wir vielleicht kurz das `Object`-Objekt, wie es sich bisher darstellt.

### 6.1. Das `Object`-Objekt

---

Mit `new Object()` aufgerufen, erstellt der **Object-Konstruktor** ein leeres Objekt, das von seinen Eigenschaften dem Literal `{ }` jedoch gleichkommt, sodass seine explizite Verwendung überflüssig ist. Der Basiskonstruktor übernimmt nämlich *auch* die Instanziierung eines Literals, sodass auch dieses stets ein „`Object`-Objekt“ ist.

In ECMA 3 gab es zur Erzeugung eines nicht-leeren Objekts zwei Wege: Man konnte das Objekt als ein **Literal** beschreiben, oder einen eigenen **Konstruktor** schreiben. Auch im zweiten Fall wird der `Object`-Konstruktor eingesetzt, nämlich für die Erzeugung des im Konstruktor verwendeten *thisObjects*:

```
// Literal (wird unmittelbar instanziiert)
var peter = {
  vorname: "Peter",
  hallo: function(){
    return "Hallo, ich bin " + this.vorname + "!";
  }
};

// Konstruktor mit Instanziierung:
var Person = function(vorname){
  // this ist ein neues Object-Objekt!!
  this.vorname = vorname;
};
```

```
};  
Person.prototype.hallo = function(){  
    return "Hallo, ich bin " + this.vorname + "!";  
}  
var klaus = new Person("Klaus");
```

## 6.2. Objektliterale in ECMA6

---

Bereits bei der Erstellung von Objektliteralen bietet ECMA6 einige Neuigkeiten. Vereinfacht wird die Definition von Properties und Methoden mittels sogenannter **Shorthand-Definitionen**.

### 6.2.1. Shorthand-Definitionen von Properties

---

Bisher müssen Objekteigenschaften im Literal mit *key* und *value* definiert werden:

```
var obj = {  
    test: "Ein Test"  
};
```

Liegt der *value* in einer Variable vor, so kann man alternativ schreiben:

```
var test = "Ein Test";
```

```
var obj = {  
    test:test  
};
```

In ECMA6 kann man innerhalb eines Objektliterals eine **Shorthand-Definition** (*concise property*) verwenden. Im Grunde wird die Variable *direkt* als Property eingesetzt:

```
var test = "Ein Test";  
var obj = {  
    test    // entspricht: test:test  
};  
console.log(obj.test); // -> Ein Test
```

## 6.2.2. Shorthand-Definition von Methoden

---

Hier ein Beispiel, das illustriert, wie ein Objektliteral erzeugt werden könnte, das *Methoden* enthält.

Ein als Speicher dienendes (anfänglich leeres) Objekt `speicher` wird über eine API bedient, die in einem anderen Objekt `speicherApi` liegt, das als Literal erstellt wird. Innerhalb des Literals werden drei Properties als Methoden mit Function-Objekten belegt:

```
var speicher = {};  
  
var speicherApi = {  
  getItem: function(key) {  
    return key in speicher ? speicher[key] : null;  
  },  
  setItem: function(key, value) {  
    speicher[key] = value  
  },  
  clear: function() {  
    speicher = {};  
  }  
};
```

Soweit, so gut. Das Objekt kann nun wie folgt verwendet werden:

```
speicherApi.setItem('test', 'Ein Test');  
console.log(speicherApi.getItem('test'));  
// -> Ein Test
```

Alles neu in ECMA6?

Für die ECMA6-Schreibweise des Literals gibt es nun einige Varianten. Zunächst kann die Methode einfach „inline“ im Literal definiert werden (*concise method*). Hierbei kann das `function`-Keyword entfallen. Dies sieht wie folgt aus:

```
var speicherApi = {  
  getItem(key) {  
    return key in speicher ? speicher[key] : null;  
  },  
  setItem(key, value) {  
    speicher[key] = value  
  },  
  clear() {
```

```
        speicher = {};  
    }  
};  
  
speicherApi.setItem('test', 'Anderer Test');  
console.log(speicherApi.getItem('test'));  
// -> Anderer Test
```

Ein anderer Ansatz soll das eigentliche Objektliteral vereinfachen, indem die Methoden außerhalb definiert werden (was beispielsweise im Rahmen eines Moduls Sinn macht. Das Beispiel könnte für einen Export der API wie folgt umgebaut werden:

```
var speicher = {};  
  
function getItem(key) {  
    return key in speicher ? speicher[key] : null;  
}  
  
function setItem(key, value) {  
    speicher[key] = value;  
}  
  
function clear() {  
    speicher = {}  
}  
  
// nur die API exportieren  
module.exports = {  
    getItem: getItem,  
    setItem: setItem,  
    clear: clear  
};
```

Das Auslagern der Methodendefinitionen aus dem Literal sorgt für Übersicht. Mittels der vorhin erläuterten Shorthand-Definitionen für *concise properties* lässt sich die exportierte API noch ein wenig mehr verkürzen:

```
// API mit Shorthand-Definitionen:  
module.exports = {  
    getItem,  
    setItem,  
    clear  
};
```



### 6.2.3. Shorthand-Definitionen mit „computed keys“

---

Ein ähnliches Prinzip ist bei sogenannten „computed keys“ möglich. Dies ist von der bekannten **Arrayschreibweise** abgeleitet, die alternativ zur Dot-Syntax zum Lesen und Erzeugen von Objekteigenschaften verwendet werden kann.

```
var testKey = 'test';

var obj = {};
obj[testKey] = 'Ein Test';

console.log(obj.test); // -> Ein Test
```

Analog hierzu kann man bei der Erstellung eines Objektliterals auch einen Stringkey durch eine Variable ersetzen, indem man diese in eckige Klammern setzt:

```
var testKey = 'test';
var obj = {
  [testKey]: "Ein Test"
};
console.log(obj.test); // -> Ein Test
```

## 6.3. Methoden des Object-Objects

---

In ECMA5 wurden dem Object-Objekt eigene direkte (d.h. „statische“) Methoden hinzugefügt, die alternative Wege zum Erzeugen und Absichern von Objekten sowie zur expliziten Definition von Properties bieten. Zusätzlich werden Methoden zur Introspektion hinzugefügt.

### 6.3.1. Object.create()

---

Die ECMA5-Methode `Object.create()` bietet einen alternativen Weg zur Erzeugung von Objektinstanzen, der ohne Einsatz eines Konstruktors oder des *new*-Keywords funktioniert. In der Regel werden wir dies im Rahmen einer Factory einsetzen:

```
var PersonFactory = function(vorname) {
  // Prototypeigenschaften
  var proto = {
```

```
        hallo: function() {
            return "Hallo, ich bin " + this.vorname + "!";
        }
    };
    // die Instanz:
    var person = Object.create(proto);
    // Instanzeigenschaften
    person.vorname = vorname;
    // Instanz zurückgeben:
    return person;
};

var peter = PersonFactory("Peter"); // ohne new!
```

- ✓ Ernsthaft etabliert hat sich dieses Verfahren bislang nicht. Interessant ist, dass es weder den Einsatz des Keywords *this* (bei der Erstellung der Instanzeigenschaften) noch die explizite Nennung des `prototype`-Properties (zur Definition des Prototyp-Objekts) bedingt.

Da hier eine **Factoryfunktion** ein Objekt erstellt und (explizit) zurückgibt, ist das `new`-keyword überflüssig. Es schadet jedoch nicht:

```
var Tom = new PersonFactory("Tom"); // na ja..
```

Eine interessante Randbemerkung mag sein, dass so auch Objekte erzeugt werden können, die *nicht* vom `Object`-Objekt abstammen.

Ein „gewöhnliches“ Objekt erhalten wir wie folgt:

```
// Object-Objekt als Prototype-Objekt:
var obj = Object.create({});
```

Der „Witz“ besteht darin, dass auch ein `null`-Objekt als Prototyp-Objekt übergeben werden darf (`null` ist ja „`typeof`“ `'object'` ...):

```
// null-Objekt als Prototyp:
var objOhneEigenschaften = Object.create(null);
```

- ✓ Diesem „nullbasierten“ Objekt fehlen alle gewöhnlichen Basisproperties, wie die `toString`-Methode und andere.

### 6.3.2. Object.defineProperty() und .defineProperties()

---

Diese beiden Methoden zur „granularen“ Beschreibung einzelner Objekteigenschaften oder einer Liste von Eigenschaften über Descriptoren werden im Seminar zu ECMA5 vorgestellt. Daher hier nur kurz als Wiederholung:

Wir unterscheiden **Accessor-** und **Value-Descriptoren**. Erstere definieren den Zugang zum Property über *Getter* und *Setter*, letztere setzen einen *Wert* und bestimmen dessen *Schreibbarkeit*.

Gemeinsam ist beiden Typen von Descriptoren, dass man die *Sichtbarkeit* (Aufzählbarkeit bzw. *enumerability*) und die *Löschbarkeit* (*configurability*) der definierten Eigenschaft festlegen kann.

Nicht festgelegte Descriptor-Eigenschaften defaulten zu *false* (nicht schreibbar, nicht sichtbar, nicht löschbar).

### 6.3.3. Object.preventExtensions(), .seal() und .freeze()

---

Auch diese drei Methoden wurden im ECMA5-Seminar vorgestellt. Sie sichern ein ihnen übergebenes Objekt `obj` gegenüber Manipulationen ab.

Man kann in dieser Staffelung

- `Object.preventExtensions(obj)`:  
Nicht-Erweiterbarkeit der Objektstruktur festlegen
- `Object.seal(obj)`:  
Strukturintegrität gegenüber Erweiterung *und* Reduktion festlegen
- `Object.freeze(obj)`:  
Wertintegrität (gegen Überschreiben von Values) zusätzlich zur Strukturintegrität festlegen.

**Freeze:** Zu beachten ist, dass Wertintegrität **nicht innerhalb komplexer Property-Values** gilt. So können Items innerhalb von *Array-Values* überschrieben, entfernt oder hinzugefügt werden, dasselbe gilt analog für die Properties innerhalb von Object-Values.  
Im Zweifel muss ein **separater Freeze** für das betreffende Property ausgesprochen werden!

- ✓ Hinweis: Die Sicherungsmethoden können auch innerhalb von Konstruktoren auf das *thisObject* angewendet werden.
- ✓ Die Sicherung eines *bereits gesicherten* Objekts kann verstärkt (z.B. von *Seal* zu *Freeze*), aber nicht abgeschwächt oder entfernt werden.

#### 6.3.4. Object.isExtensible(),.isSealed() und .isFrozen()

---

Eine *ExtensionPrevention*, ein *Seal* oder ein *Freeze* kann nicht rückgängig gemacht werden. Um den Zustand eines Objekts *obj* in Erfahrung zu bringen, existieren die drei Testmethoden `Object.isExtensible(obj)`, `Object.isSealed(obj)` und `Object.isFrozen(obj)`, die mit einem Booleschen Wert antworten.

Die drei Methoden wurden im ECMA5-Seminar vorgestellt.

#### 6.3.5. Object.keys() und Object.getOwnPropertyNames()

---

Die Methode `Object.keys()` gibt für ein ihr übergebenes Objekt ein *Array* zurück, dass eine **Liste aller aufzählbaren String-Keys** enthält, die als **Instanzeigenschaften** definiert sind. Nicht erfasst, werden Symbol-Keys, nicht aufzählbare String-Keys und Eigenschaften, die im Prototyp liegen:

```
var c = Symbol('c');

function F(){
  // String-Key
  this.a = "A";
  // String-Key, nicht enumerable
  Object.defineProperty(this, "b", {value:"B"});
  this[c] = "C"; // Symbol-Key
}
// String-Key im Prototyp
F.prototype.d = "D";

var o = new F();
Object.keys(o); // -> ["a"] (nur "a" ist aufzählbar)
```

Anders als `Object.keys()` gibt `Object.getOwnPropertyNames()` sowohl *aufzählbare* als auch *nicht aufzählbare* Eigenschaften zurück:

```
Object.getOwnPropertyNames(o); // ["a", "b"]
```

### 6.3.6. Object.getOwnPropertySymbols()

---

Die eben wahrgenommene Lücke, die `Object.keys()` und `Object.getOwnPropertyNames()` hinterlassen haben, wird durch die statische Methode `Object.getOwnPropertySymbols()` geschlossen, die eben das tut, was ihr Name nahelegt: sie gibt **Symbol-Keys** zurück.

Als Änderung gegenüber dem vorigen Object legen wir nun auch noch einen Symbol-Key in den Prototyp.

```
var c = Symbol('c');
var e = Symbol('e');

function F(){
  // String-Key
  this.a = "A";
  // String-Key, nicht enumerable
  Object.defineProperty(this, "b", {value:"B"});
  this[c] = "C"; // Symbol-Key
}
// String-Key im Prototyp
F.prototype.d = "D";
// Symbol-Key im Prototyp
F.prototype[e] = "E";

var o = new F();

Object.getOwnPropertySymbols(o); // -> [Symbol {}]
```

Mit dem so erhaltenen Symbol kann der Wert ausgelesen werden:

```
var [s] = Object.getOwnPropertySymbols(o);
console.log(o[s]); // -> C
```

- ✓ Der Symbol-Key `e` im Prototyp wird nicht erfasst, da es sich nicht im einen „eigenen“ Key der Instanz handelt.

### 6.3.7. Kopieren von Eigenschaften mit Object.assign()

---

✓ **Achtung:** Bisher in Internet Explorer nicht unterstützt!

Die ES6-Methode `Object.assign()` kopiert **sichtbare Instanzeigenschaften** eines oder mehrerer Objekte (*sources*) in ein Zielobjekt (*target*) und gibt das so erweiterte Zielobjekt zurück.

✓ Kopiert werden hierbei nicht nur für Properties mit String-Keys, sondern auch durch Symbole gekennzeichnete Properties.

```
Object.assign(target, ...sources)
```

So kann beispielsweise ein Objekt **geklont** werden, indem es einem leeren Target-Objekt zugewiesen wird:

```
var objX = {x: "X"};
var cloneX = Object.assign({}, objX);
console.log(cloneX);           // {x: "X"}
console.log(objX === cloneX);  // false
```

Es können *mehrere Objekte* verschmolzen (merged) werden. Hierbei werden die Values *gleichnamiger* Properties am Zielobjekt überschrieben:

```
var cloneXY = Object.assign({x: "ABC"}, {x: "X"}, {y: "Y"});
console.log(cloneXY);           // {x: "X", y: "Y"}
```

✓ Ist das Target nicht leer, so wird es um die Eigenschaften der Sources *erweitert*.

## 7. Konstruktoren mit class-Keyword

---

JavaScript ist eine **prototypbasierte** objektorientierte Sprache. Dies gilt weiterhin für ECMA6, obwohl hier eine alternative (und vielleicht einfachere) Syntax zur Erstellung von Konstruktoren mittels des `class`-Keywords eingeführt wird.

Um es unmissverständlich klar zu machen: Die in ECMA6 eingeführten „Klassen“ sind lediglich „syntactic sugar“ über dem herkömmlichen prototypischen Ansatz in Javascript. Wir sind mit dem `class`-keyword keinen Schritt näher an „echter“ klassenbasierter Objektorientierung.

Hier ein Beispiel für die „herkömmliche“ Erstellung eines **Konstruktors** (wobei dieser hier mit einer IIFE bereits „fortschrittlich“ verkapselt wurde):

```
var Person = (function () {
    // die Konstruktorfunktion:
    function Person(vorname) {
        this.vorname = vorname;
    }
    // Methoden werden an den Prototyp gebunden:
    Person.prototype.hallo = function () {
        console.log("Hallo, ich bin " + this.vorname, "!");
    };
    // Auch Properties können hier abgelegt werden:
    Person.lieblingsessen = "Pizza";

    return Person;
})();

var peter = new Person("Peter");
peter.hallo();
// -> Hallo, ich bin Peter
console.log(peter.lieblingsessen);
// -> Pizza
```

In ECMA6 kann analoges mit dem `class`-Keyword beschrieben werden. Das `prototype`-Objekt wird zwar nicht erwähnt, findet aber hinter den Kulissen ebenfalls Verwendung:

```
class Person {
    // auch hier: ein Konstruktor
    constructor(vorname) {
        this.vorname = vorname;
    }

    // Shorthand-Deklaration einer Prototype-Methode:
    hallo() {
        console.log("Hallo, ich bin " + this.vorname, "!");
    }
}
```

```
var hans = new Person("Hans");  
hans.hallo();  
//-> Hallo, ich bin Hans!
```

Es besteht eine gewisse (gefährliche) *Ähnlichkeit* zur Syntax der Objektliterale. Analog zu dieser wird die „Method-Shorthand“-Notation verwendet, bei der das `function`-Keyword unterbleibt.

✓ **Achtung:** Es stehen im `class`-Block jedoch weder *Kommas* noch *Semikolons* zwischen den Propertydefinitionen!

Eine Sonderstellung hat in diesem Zusammenhang die Funktion `constructor`, die dieselbe Aufgabe erfüllt wie der **Konstruktor** in der „prototypischen“ Schreibweise: Eben diese Funktion wird bei der Erzeugung von Instanzen aufgerufen und setzt mittels `this` dann die Instanzeigenschaften.

✓ Alle *anderen* Methoden werden automatisch dem **Prototyp** zugeordnet, ohne dass dies eigens erwähnt werden muss.

#### Nochmals zur Syntax:

Eine Klasse ist *kein Objektliteral*. Da es sich hier *nicht* um eine Aufzählung von Properties handelt, steht zwischen zwei Eigenschaften **kein Komma**!

## 7.1. Getter und Setter in Klassen

Nicht vorgesehen ist das einfache Definieren einfacher Properties im Prototype, wie es früher geschehen konnte. Dies wird als „unsicher“ (*unsafe*) betrachtet und soll nun durch Getter und Setter ersetzt werden.

Beim Einsatz von **Gettern** und **Settern** für ein Prototyp-Property `prop` ist es vorgesehen (sic), dass gleichzeitig eine „getarte“ Eigenschaft `_prop` angelegt wird, etwas so:

```
class Person {  
  constructor(vorname) {  
    this.vorname = vorname;  
    // hier die „getarte“ Eigenschaft:  
    this._lieblingessen = "Pizza";  
  }  
}
```



```
// Getter analog zu Prototype-Methode deklariert:
get lieblingsessen() {
    return this._lieblingsessen;
}

// ... ebenso der Setter, der die Zuweisung vornimmt
set lieblingsessen(essen) {
    console.log("Esse jetzt ", essen, "!");
    this._lieblingsessen = essen;
}

hallo() {
    console.log("Hallo, ich bin " + this.vorname);
}
}
```

Das beabsichtigte Prototype-„Property“ wird also durch einen Getter repräsentiert; quasi einer Funktion mit dem *Namen des Properties* mit davorgesetzten Keyword `get`.

- ✓ Wie von Accessor-Deskriptoren her bekannt, ist der Zugriff auf die Eigenschaft jedoch *direkt* möglich, ohne zu „wissen“, dass über Getter/Setter gearbeitet wird, z.B.:

```
console.log(gustav.lieblingsessen); //-> Pizza
```

- ✓ Analoges gilt für den Setter, der benötigt wird, falls das Property auch schreibbar sein soll. Er wird einfach durch Zuweisung an das Property getriggert (s.u.).

Machen wir eine Objektinstanz anhand der Klasse. Dies erfolgt, wie gewohnt durch Aufruf mit `new` (wie bei einem Konstruktor):

```
var gustav = new Person("Gustav");
gustav.hallo();
```

Oops, bei der Betrachtung des Objekts kommt heraus, dass die „getarnte“ Eigenschaft `this._lieblingsessen` gar nicht so geheim ist. Im Log des Objekts ist sie sichtbar und entpuppt sich auch als direkt schreibbar.

```
console.log(gustav);

// Änderung über den Setter per einfacher Zuweisung:
gustav.lieblingsessen = "Pasta"; // triggert Setter
console.log(gustav.lieblingsessen); // -> Pasta

// Änderung am Setter vorbei:
gustav._lieblingsessen = "Sauerkraut"; // Oops...
console.log(gustav.lieblingsessen); // -> Sauerkraut
```

✓ Interessant ist es, dass nicht *zwangsweise* ein Setter vereinbart werden muss.

Es ist denkbar, den **Speicher** für den Wert auch **außerhalb** der Klasse zu definieren. Hier liegt er in einer Variable `_lieblingsessen`.

```
"use strict";
// außerhalb liegende Variable
var _lieblingsessen = "Pizza";

class Person {
  constructor(vorname, alter) {
    this.vorname = vorname;
    this.alter = alter;
  }

  hallo() {
    console.log("Hallo, ich bin ",this.vorname);
  }

  // NUR ein Getter, kein Setter!!
  get lieblingsessen(){
    return _lieblingsessen;
  }
}

var peter = new Person("Peter",42);
console.log(peter.lieblingsessen); // Pizza

peter.lieblingsessen = "Pasta";
// Typfehler wegen strict Mode !!!
```

**Anmerkung:** Die außerhalb des `class`-Blocks liegende Variable ist unproblematisch, wenn die Klasse aus einem Modul als Export zur Verfügung gestellt wird, da sie dann in dessen Scope „versteckt“ ist.

Ansonsten lässt sich die Kapselung auch wieder über eine **IIFE** realisieren.

```
var Person = (function(){
    var _lieblingessen = "Pizza";
    return class {
        ///
    }
})();
```

✓ Natürlich würde diese Vorrichtung für ein Getter/Setter-Property zu Problemen führen, da sich alle Instanzen einen Speicher teilen!

## 7.2. Verwendung von Symbolen

Die Erweiterung von Objekten ist prinzipiell unvorhersehbar. Werden neue Properties mit String-Keys versehen, ist eine versehentliche Duplizierung nicht undenkbar. Außerdem sind String-Key-Properties von außen (meist) sichtbar, wie im Fall vermeintlich „getarnter“ Speicher für Getter/Setter.

Die Einführung von **Symbolen** („symbol values“) schiebt hier einen Riegel vor, indem sie die Erzeugung *primitiver* (also für Object-Keys tauglicher) Werte mit einer *Identität* (also Unverwechselbarkeit) ermöglicht.

Hierbei wird die `Symbol`-Methode verwendet, die bei Aufruf ein Symbol zurückgibt:

```
var meinSymbol = Symbol();
```

Hierbei gilt stets:

```
console.log(Symbol() === Symbol()); // -> false
```

**Achtung**, das `Symbol`-Objekt wird stets nur als **Funktion** verwendet und **nicht als Konstruktor**! Ein Aufruf `new Symbol()` hat einen *Typfehler* zur Folge: „*TypeError: Symbol is not a constructor*“.

Die `Symbol`-Funktion nimmt optional einen Stringwert entgegen, der als „Beschreibung“ (*description*) des Symbols gilt. Die Übergabe gleicher Descriptions erzeugt wiederum *nicht* zwei gleiche Symbole:

```
var testSymbol = Symbol('test');

console.log(Symbol('x') === Symbol('x')); // -> false
```

Die Beschreibung wird mittels der `toString`-Methode des Symbols sichtbar, wobei der String *ohne eigene Begrenzer* ausgegeben wird:

```
testSymbol.toString(); // -> "Symbol(test)"
```

Die Verwendung an einem Objekt erfolgt analog zur Array-Schreibweise. Die Dot-Syntax ist nur mit String-Keys möglich:

```
var testSymbol = Symbol('test');
var testObj = {};
testObj[testSymbol] = "Test für einen Symbol-Key.";
```

✓ Zudem sind Symbol-Properties prinzipiell nicht „enumerable“, also von außen nicht sichtbar.

Die Symbole können in Literalen auch *direkt* eingesetzt werden:

```
var sym1 = Symbol('s');
var sym2 = Symbol('s'); // dies ist ein ANDERES Symbol

var o = {
  [sym1]:42,
  [sym2]:17
};

// Symbole sind nicht „enumerable“:
console.log(o); // -> Object {} (erscheint leer!)

// Zugriff auf Values ist über die Symbole möglich:
console.log(o[sym1]); // -> 42
console.log(o[sym2]); // -> 17

// Symbol-Keys können extrahiert werden:
var symKeys = Object.getOwnPropertySymbols(o);

console.log(symKeys); // -> [Symbol {}, Symbol {}]
console.log(symKeys[0]===sym1); // -> true
```

Im Rahmen der Erzeugung von Objekten werde Symbole alternativ zu Stringkeys eingesetzt. Hierbei wird das Symbol im gleichen Scope deklariert, wie der Konstruktor/die Klasse:

```
var lieblingsessenSymbol = Symbol();

class Person {
  constructor(vorname) {
    this.gattung = vorname;
    this[lieblingsessenSymbol] = "Pizza";
  }

  get lieblingsessen() {
    return this[lieblingsessenSymbol];
  }

  set lieblingsessen(essen) {
    this[lieblingsessenSymbol] = essen;
  }

  hallo() {
    console.log("Hallo, ich bin " + this.vorname, "!");
  }
}

var tina = new Person("Tina");
tina.hallo();
tina.lieblingsessen = "Pfannkuchen";
```

- ✓ Der **Vorteil** der Verwendung eines Symbols ist, neben der Vermeidung von Kollisionen, ein besseres „Verstecken“ des eigentlichen Speicherproperties. Im Gegensatz zur vorher verwendeten Stringkey-Eigenschaft `this._lieblingsessen` wird die Symboleigenschaft `[lieblingsessenSymbol]` nicht von *for-in* erfasst.

Natürlich existiert die Methode `getOwnPropertySymbols()` die einem hier wieder einen Strich durch die Rechnung machen kann.

### 7.3. extends und super – Vererbung mit Klassen

Da nun zumindest der Anschein von Klassen verfügbar ist, erwartet man auch einfachere Möglichkeiten der Vererbung. Auch bisher war es schon möglich, Vererbung in ES5 zu simulieren, dies war jedoch eine recht komplexe Angelegenheit, da nur Objektinstanzen vererben konnten.

- ✓ In ECMA6 gibt es nun endlich ein `extends`-Keyword, das die Vererbung von einem Konstruktor ermöglicht, der dann als `super` (quasi die „Superklasse“) bezeichnet wird.

```
class Fahrer extends Person {
  constructor(klasse, vorname) {
    super(vorname); // ruft Konstruktor Person auf
    this.klasse = klasse;
  }

  hallo() {
    super.hallo(); // ruft gleichnamige Supermethode
    console.log("Habe Führerschein ", this.klasse,
"!");
  }
}

var viktor = new Fahrer ("Klasse 1", "Viktor");
viktor.hallo();
// -> Hallo, ich bin Viktor!
// -> Habe Führerschein Klasse 1!
```

## 8. Erweiterung der Arraysyntax

---

Der Array-Typ wurde in ECMA6 durch einige Methoden erweitert, von denen zwei direkt dem `Array`-Konstruktor zugeordnet werden, die also als „**statische Methoden**“ gewertet werden können. Die restlichen neuen Methoden liegen an `Array.prototype`, stehen also als **Instanz-Methoden** zur Verfügung.

- ✓ Der Array-Typ wird auch zu den **Iterables** gezählt, womit ein bestimmtes Verhalten bezeichnet wird. Mehr dazu später.

### 8.1. Neue statische Funktionen für den Array-Typ

---

Bisher gab es lediglich *eine* statische Methode für den Array-Typ, nämlich `Array.isArray()`, die dazu dient, festzustellen, ob es sich bei einem, ihr *übergebenen*, Wert in der Tat um ein Array handelt.

```
Array.isArray(["Rosen","Tulpen","Nelken"]); // -> true
```

Statische Methoden stehen **nicht** an den Instanzen zur Verfügung.

```
[1,2,3].isArray(); // -> TypeError ... not a function
```

In ECMA6 kommen zwei weitere Methoden hinzu.

### 8.1.1. Array.of()

Soll ein Array aus Zahlen konstruiert werden und wird hierfür der **Array-Konstruktor** verwendet, so gibt es ein Problem, wenn nur *eine* Zahl übergeben wird. Bei mehreren Zahlen geht alles klar...

```
var zahlenArray = new Array(3,2,1);  
// -> [3, 2, 1]
```

Aber:

```
var zahlenArray = new Array(3);  
// -> [undefined, undefined, undefined] ...oops!
```

Diese Konsistenzlücke füllt `Array.of()` wie folgt:

```
var zahlenArray = Array.of(3);  
// -> [3]
```

Die Methode nimmt allerdings nicht nur Zahlen, sondern *jegliche Art* von Aufzählungen entgegen, kann also den Konstruktor vollwertig ersetzen:

```
var arr = Array.of("Rosen",5,"Nelken",false);  
// -> ["Rosen", 5, "Nelken", false]
```

Aber Vorsicht, es handelt sich um einen **Methodenaufruf** und *nicht* um eine *Variante* des Konstruktoraufrufs. Dies hier ist **falsch**:

```
var zahlenArray = new Array.of(3);  
// -> TypeError: Array.of is not a constructor
```

### 8.1.2. Array.from

Die Methode `Array.from()` geht die Herausforderung an, aus einem „arrayähnlichen“ Objekt ein „echtes“ Array abzuleiten. Hierbei wird nicht das übergebene Objekt umgeformt, sondern ein **neues Array** erstellt.

- ✓ Die Methode ist zufrieden, wenn das übergebene Objekt ein „Iterable“ ist, wofür ein `length`-Property und *numerische* Keys genügen.

Nehmen wir ein entsprechend passendes, sehr wenig „arrayähnliches“ Objekt als Ausgangspunkt:

```
var keinArray = {  
  length: 3,  
  2: "Egal"  
};
```

Dies übergeben wir nun an `Array.from()`:

```
var jetztEinArray = Array.from(keinArray);  
console.log(keinArray);  
// -> Object { 2="Egal", length=3}  
console.log(jetztEinArray);  
// -> [undefined, undefined, "Egal"]
```

- ✓ **Achtung:** Die Fächer mit `undefined` enthalten *real* den Value *undefined*, existieren also!

```
// "wirklich" leere Fächer würden übersprungen:  
jetztEinArray.forEach(function(item){  
  console.log(item);  
});  
// -> undefined (!)  
// -> undefined (!)  
// -> Egal
```

Wird als *zweites Argument* eine Funktion übergeben, so dient diese als **Mapping-Funktion**. Hier ein Beispiel:

```
var upperStringArray = Array.from(keinArray,  
  function mapper(val, ind){  
    if(typeof val == "string") {  
      return val.toUpperCase();  
    }  
  })
```



```
    }  
  });  
  console.log(upperStringArray);  
  // -> [undefined, undefined, "EGAL"]
```

## 8.2. Neue Instanzmethoden für den Array-Typ

---

Andere Neuerungen in diesem Bereich stehen als Instanzmethoden allen Arrays zur Verfügung.

### 8.2.1. [].copyWithin()

---

Diese Methode kopiert Items innerhalb des Arrays an eine neue Position, wobei die ursprünglich dort befindlichen Items überschrieben werden. Es wrt der Zielindex übergeben, der Startindex des zu kopierenden Bereichs sowie optional die Länge des zu kopierenden Bereichs.

```
[ "A", "A", "A", "B", "B", "B", "B" ].copyWithin(4,0);  
// -> [ "A", "A", "A", "B", "A", "A", "A" ]  
  
[ "A", "A", "A", "B", "B", "B", "B" ].copyWithin(4,0,2);  
// -> [ "A", "A", "A", "B", "A", "A", "B" ]
```

✓ Die Länge des Arrays wird durch `copyWithin()` nicht verändert!

### 8.2.2. [].fill()

---

Die Methode **füllt** ein Array mit einem ihr übergebenen Wert ab einem Startindex (2. Argument) bis zum Ende oder zu einem *exklusiven*(!) Endindex (3., optionales Argument).

```
var arr = new Array(4).fill("Hey ho");  
// -> ["Hey ho", "Hey ho", "Hey ho", "Hey ho"]  
  
var arr = new Array(4).fill("Hey ho", 1);  
// -> [undefined, "Hey ho", "Hey ho", "Hey ho"]  
  
var arr = new Array(4).fill("Hey ho", 1, 3);  
// -> [undefined, "Hey ho", "Hey ho", undefined]
```

- ✓ **Achtung:** Befinden sich im Array bereits Items, so werden diese überschrieben:

```
var arr = [1,2,3,4,5].fill("Hey ho",1,3);  
// -> [1, "Hey ho", "Hey ho", 4, 5]
```

### 8.2.3. [].find()

---

Verhält sich wie [].some(), gibt allerdings für einen Match nicht *true* sondern den gematchten **Value** zurück:

Wir wollen die erste 5 oder 7 eines Ziffernarrays haben:

```
[9,3,4,2,7,6,1,1,5,7].find(function(val) {  
    return val == 5 || val == 7;  
});  
// -> 7
```

### 8.2.4. [].findIndex()

---

Ergänzend zu [].find() gibt [].findIndex() den **Index** des Matches zurück:

```
[9,3,4,2,7,6,1,1,5,7].findIndex(function(val) {  
    return val == 5 || val == 7;  
});  
// -> 4
```

## 9. Das Iterable und die for-of-Schleife

---

ECMA6 führt den Begriff des **Iterable** ein, was bedeutet, dass eine iterierbare Struktur (*sequence of values*) mit Hilfe einer Schleife traversiert werden kann. Diese Struktur muss dafür *linear*, d.h. „geordnet“ sein (beispielsweise über einen numerischen Index).

- ✓ Die Objekttypen **String**, **Array**, **TypedArray**, **Map** und **Set** sind *Iterables*.

Diese Anordnung trifft also „per se“ auf **Arrays** zu, weshalb Arrays quasi der Standardfall eines Iterables sind.

- ✓ Iterables werden im Zusammenhang mit der neuen *for-of* Schleife, dem Spread-Operator, dem `yield*`-Keyword (*yield star*, siehe Generatoren) und für *Destructuring*-Zuweisungen verwendet.

## 9.1. Die for-of-Schleife

---

Um ein Iterable *direkt* zu verarbeiten stellt ECMA6 die neue Schleifenform `for ... of` zur Verfügung

```
var blumenArray = ["Rosen", "Tulpen", "Nelken"];

for(let val of blumenArray) {
  console.log(val);
}
// -> Rosen
// -> Tulpen
// -> Nelken
```

Das geht analog auch mit einem **String**, da dieser ebenfalls ein *Iterable* darstellt:

```
var testString = "Hey ho!";

for(let char of testString){
  console.log(char);
}
// -> H
// -> e
// -> y
// -> 
// -> h
// -> o
// -> !
```

- ✓ **Hintergrundinfo 1:** Hinter den Kulissen bewirkt die *for-of* Schleife, dass das *Iterator-Protokoll* des so angesprochenen Iterable aktiv wird und ein sogenanntes **Iterator-Objekt** zurückgibt, welches die Werte-Sequenz repräsentiert und das dann verarbeitet wird.

- ✓ **Hintergrundinfo 2:** In einem Iterator-Objekt finden wir eine *lineare Anordnung* von Werten, wobei durch den Aufruf einer implementierten **Methode** `next()` zum *jeweils nächsten* weitergegangen wird. Im Grunde ist `for...of` also ein Trigger dieser `next()`-Methode.

**Seitenblick:** Wir werden dieses Prinzip später bei *Generatoren* wiederfinden, weshalb Generatoren auch zu den *Iterables* gezählt werden.

## 9.2. Iterable - die Methoden `entries()`, `values()`, `index()`

Man kann aus einem Iterable auf direktem Weg ein *Iterable-Object* erhalten, indem man seine Methode `entries()` aufruft. Das Iterable-Objekt enthält eine Folge von *Value-Arrays*, die jeweils *Index* und *Wert* verpackeln. Es verfügt über eine `next()`-Methode.

Desweiteren stehen die Methoden `index()` und `values()` zur Verfügung, um an die *Indexreihenfolge* bzw. die gespeicherten *Werte* zu kommen.

```
var blumenArray = ["Rosen", "Tulpen", "Nelken"];
```

- ✓ Der Rückgabewert der Methoden ist jedoch nur ein *arrayähnliches Objekt*, sodass wir zur Illustration auf den Rückgabewert mit `...` einen *Spread* machen und es so in ein Array zurückwandeln.

Folgendes finden wir heraus:

```
[...blumenArray.keys()];  
// -> [0,1,2] // die Indexfolge  
  
[...blumenArray.entries()];  
// -> [ [0,"Rosen"], [1,"Tulpen"], [2,"Nelken"] ]
```

**Anmerkung:** Die Methode `values()` ist derzeit aus Kompatibilitätsgründen aus dem Implementierungen entfernt worden.

```
[...blumenArray.values()]; // Spread wandelt in Array!!  
// -> ["Rosen", "Tulpen", "Nelken"] // die Werte
```

Der oben gezeigte Code funktioniert daher momentan nicht.

### 9.3. Das Iterable-Object

---

Die `entries()`-Methode gibt, wie gesagt, ein **Iterable-Object** zurück und verfügt daher über eine `next()`-Methode. Diese gibt ein Objekt mit `value`-Property zurück, welches das jeweils „anstehende“ *Index-Value*-Pärchen enthält.

```
// dieses iterable Objekt tut zunächst nichts...
var blumenIterableObject = blumenArray.entries();

var blumenObjekt;

// wir holen den ersten Value-Container:
blumenObjekt = blumenIterableObject.next();

// und betrachten dessen value-Property::
console.log(blumenObjekt.value);
// -> [0, "Rosen"]
// und betrachten dessen done-Property:
console.log(blumenObjekt.done);
// -> false
```

Das wiederholen wir dreimal:

```
blumenObjekt = blumenIterableObject.next();
console.log(blumenObjekt.value);
// -> [1, "Tulpen"]
console.log(blumenObjekt.done);
// -> false

blumenObjekt = blumenIterableObject.next();
console.log(blumenObjekt.value);
// -> [2, "Nelken"]
console.log(blumenObjekt.done);
// -> false

blumenObjekt = blumenIterableObject.next();
console.log(blumenObjekt.value);
// -> undefined
console.log(blumenObjekt.done);
// -> true
```

- ✓ Sobald die Sequenz durchschritten ist, nimmt das Property `done` des Iterator-Objekt den Wert `true` an. Ein weiterer Aufruf von `next()` gibt dann `undefined` zurück.

## 10. Asynchrone Programmierung

---

Asynchrone Programmierung wird allgemein mittels **Callbacks** realisiert, wobei man die „*continuation*“ (die Fortsetzung des Programms nach dem asynchron erfolgenden Schritt) an die Instanz weitergibt („*passing*“), die für diesen asynchronen Schritt zuständig ist. Man nennt dieses Prinzip auch „*Inversion of Control*“.

Potentielle Probleme ergeben sich bei mehreren asynchronen Stufen in **Unübersichtlichkeit** der sich ergebenden *Programmstruktur* („*Callbacktrichter*“), der **Unvorhersehbarkeit** der *Reihenfolge* in der Callbacks am Ende aufgerufen werden und der **Unsicherheit**, ob der Rückruf auch tatsächlich zuverlässig erfolgen wird.

### 10.1. Promises

---

Mittels sogenannter Promises kann die *Inversion of Control* nun erneut umgedreht werden, weshalb sich das Prinzip inzwischen allgemein durchgesetzt hat. In Javascript konnte es auf einfachem Wege bislang nur über spezielle Bibliotheken (q.js u.ä.) realisiert werden.

Ein ECMA6 stehen nun (endlich?) **Promises** als native Objekte zur Verfügung und können per Konstruktoraufbau erstellt werden. Hierfür wird dem Konstruktor `Promise()` eine Funktion übergeben, die als **Handlerfunktion** bezeichnet wird.

```
var p = new Promise(handlerFunktion);
```

Die Handlerfunktion definiert den **Prozess**, den das Promise überwachen soll und besitzt die Mittel das Promise so zu steuern, dass es seine Aufgabe erfüllen kann, nämlich Callbacks zu feuern.

Hierfür wird der Handler mit zwei Argumenten versorgt, nämlich den Funktionen `resolve` und `reject` (diese Parameternamen haben sich etabliert), die zum Auflösen bzw. Zurückweisen des Promises dienen.

Folgende vier Zustände kennt ein Promise:

✓ **fulfilled**

Der asynchrone Prozess, der das Promise steuert, ist erfolgreich beendet und hat die `resolve`-Funktion gefeuert.

✓ **rejected**

Der asynchrone Prozess, der das Promise steuert, ist fehlgeschlagen und hat die `reject`-Funktion gefeuert.

✓ **pending**

Der asynchrone Prozess, der das Promise steuert, dauert an. Das Promise ist weder *fulfilled* noch *rejected* und keine der beiden übergebenen Funktionen wurde bislang gefeuert.

✓ **settled**

Dies ist kein offizieller Zustand, sondern bedeutet nur, dass der Prozess beendet ist. Das Promise ist also entweder *fulfilled* oder *rejected*.

**Der springende Punkt:** Das Promise verkapselt einen asynchronen Vorgang und erlaubt Zugriff auf dessen Ergebnis. Der Zugriff wird gewährt, sobald der Vorgang abgeschlossen ist. Da das Promise anschließend *immutable* ist, steht das Ergebnis auch später zuverlässig zur Verfügung.

Üblicherweise verwendet man eine *anonyme Funktion* als Handler, womit wir beim folgenden Bild sind:

```
var p = new Promise( function(resolve,reject){  
    // einen asynchronen Prozess starten  
} );
```

- ✓ Ist der gestartete Prozess erfolgreich, so müssen in der Handlerfunktion Maßnahmen zur Behandlung des Erfolgs getroffen werden. Dies besteht darin, dass im **Erfolgsfall** die übergebene `resolve`-Funktion gestartet wird.

Die `resolve`-Funktion schaltet das Promise nicht reversibel in den Zustand **fulfilled**, wenn sie ohne Argument aufgerufen wird. Sie nimmt optional ein Argument entgegen.

```
var p = new Promise( function(resolve,reject){
    // einen asynchronen Prozess starten

    if(prozessErfolgreich == true) {
        resolve(ergebnis)
    }
});
```

- ✓ Falls es sich bei dem **resolve-Argument** um ein **Promise** handelt, so steuert das übergebene Promise ab diesem Moment das Verhalten des Promises, d.h. dieses schaltet auf *fulfilled*, wenn das übergebene Promise *fulfilled* ist (oder sein wird) bzw. auf *rejected*, wenn das übergebene Promise *rejected* ist (oder sein wird).
- ✓ Handelt es sich bei dem Argument *nicht* um ein Promise, sondern einen beliebigen anderen Wert, so schaltet das Promise endgültig in den Zustand *fulfilled*.

Schlägt der Prozess fehl, so wird analog die *reject*-Funktion gestartet.

Die *reject*-Funktion nimmt ein Argument entgegen, das als *reason* für die Zurückweisung des Promises verwendet wird und schaltet das Promise (nicht reversibel!) in den Zustand **rejected**.

```
var p = new Promise( function(resolve,reject){
    // einen asynchronen Prozess starten

    resolve(ergebnis)

    reject(reason)
} );
```

### 10.1.1. Die Methode .then()

---

Nehmen wir exemplarisch eine Funktion `get()` an, die ein Promise-Objekt zurückgibt. Sie verkapselt (möglicherweise) einen Ajax-Request, der eine als URL übergebene JSON-Datei holt.

Das Promise-Objekt bietet eine Methode `then()`, mit der Callbacks gebunden werden können:



```
get('story.json').then();
```

Die Methode nimmt ein bis zwei Funktionen entgegen. Die erste übergebene Funktion dient als **Success-Callback**. Sie feuert, sobald das Promise in den *fulfilled*-Zustand umspringt:

```
get('story.json').then(function(response) {  
    console.log("Erfolg!", response);  
});
```

Wird eine zweite Funktion übergeben, so dient sie als **Error-Callback** und feuert, sobald das Promise in den *rejected*-Zustand springt.

```
get('story.json').then(function(response) {  
    console.log("Erfolg!", response);  
}, function(error) {  
    console.log("Fehler!", error);  
});
```

✓ Es kann also immer nur eine der beiden Funktionen feuern!

Möchte man lediglich einen Callback für die Fehlerbehandlung binden, so ist dies mit `then()` möglich, indem der Wert *undefined* anstelle der ersten Funktion übergeben wird.

```
get('story.json').then(undefined, function(error) {  
    console.log("Fehler!", error);  
});
```

Anstelle dieses Konstrukts kann alternativ die Methode `catch()` eingesetzt werden, die nur *eine* Funktion, den Fehler-Callback, entgegennimmt.

### 10.1.2. Die Methode `.catch()`

Eine andere Methode der Fehlerbehandlung ist über die Methode `catch()` möglich, die einen Error-Callback entgegennimmt. Hier wird sie mit dem Aufruf der `then()`-Methode verkettet:

```
get('story.json')  
    .then(function(response) {
```

```
        console.log("Erfolg!", response);
    })
    .catch(function(error) {
        console.log("Fehler!", error);
    });
```

- ✓ Hierbei ist zu beachten, dass die `then()`-Methode ein **neues Promise** zurückgibt, das aber an das Originalpromise gekoppelt ist (also durch dessen Zustand gesteuert wird).

Streng genommen ist der obige Code daher *nicht* dasselbe wie...

```
var prom = get('story.json');

prom.then(function(response) {
    console.log("Erfolg!", response);
});

prom.catch(function(error) {
    console.log("Fehler!", error);
});
```

### 10.1.3. Promise-Beispiel mit `setTimeout()`

Ein Promise bewahrt seinen Zustand auf, weshalb auch später noch Callbacks mit der `then()`-Methode gebunden werden können. Zur Illustration wird hier eine Funktion vorgestellt, die ein Promise zurückgibt, das per Timeout nach einer Zufallszeit aufgelöst wird (wobei die Zufallszeit zurückgegeben wird).

```
function machTimerPromise() {

    return new Promise(function(resolve, reject) {
        var randomTime = Math.floor(Math.random()*50)*50;
        setTimeout(function() {
            resolve('Fertig nach ' + randomTime + 'ms.');
```

Nun wird ein Promise erzeugt:

```
var timerPromise = machTimerPromise();

// sofort Callbacks binden:
timerPromise.then(function(result) {
    console.log(result);
});
```

An das bestehende Promise können auch *nach* dessen Auflösung noch weitere Callbacks gebunden werden (auf einem Promise kann unbegrenzt oft dessen `then()`-Methode ausgeführt werden). Dies wird durch einen weiteren, großzügig gesetzten Timeout erreicht:

```
// und viel später geht auch noch
setTimeout(function(){

    // Promise ist jetzt bereits „settled“:
    timerPromise.then(function(result) {
        console.log("Wie war das? ", result);
    });

}, 5000)
```

Der Promisegenerator kann so erweitert werden, dass das Promise auch `rejected` werden kann. Dies soll geschehen, wenn die Zufallszeit einen Schwellenwert überschreitet.

```
function machTimerPromise() {
    return new Promise(function(resolve, reject) {
        var randomTime = Math.floor(Math.random()*50)*50;
        setTimeout(function() {
            if(randomTime < 1000){
                resolve(randomTime);
            } else {
                reject(randomTime);
            }
        }, randomTime);
    })
}
```

Die zweite an `then()` übergebene Funktion feuert für den *rejected*-Zustand.

```
var timerPromise = machTimerPromise();

// sofort Callbacks binden:
timerPromise.then(function(result) {
```

```
    console.log('Fertig nach ' + result + 'ms.');
```

```
  }, function(error){
```

```
    console.log(error + 'ms dauert zu lange!');
```

```
  });
```

Dies gilt auch für die später gebundenen Callbacks.

```
// und viel später geht auch noch
```

```
setTimeout(function(){
```

```
  timerPromise.then(function(result) {
```

```
    console.log("Wie lang war's? ", result + 'ms!!');
```

```
  }, function(){
```

```
    console.log("Oops, Zeitüberschreitung!");
```

```
  });
```

```
}, 5000);
```

---

#### 10.1.4. Werte transformieren mit Promise-Chains

---

Das Interessante an Promises und ihrer `then()`-Methode ist, dass sich damit **Promise-Chains** bilden lassen.

Man kann beliebig viele Callbacks über *parallele* `then()`-Aufrufe bilden. Die Callbacks erhalten in diesem Fall alle denselben Resolve- bzw. Reject-Wert:

```
// paralele Callbackchains
```

```
var prom = new Promise(function(resolve, reject){ ...})
```

```
prom.then(fn, fn);
```

```
prom.then(fn, fn);
```

```
prom.then(fn, fn);
```

Da bei jeder Anwendung von `then()` ein neues Promise zurückgegeben wird, kann alternativ auch eine **Kette** aus *voneinander abhängigen* Promises gebildet werden:

```
var prom1 = new Promise(function(resolve, reject){
```

```
  ...})
```

```
var prom2 = prom1.then(fn, fn);
```

```
var prom3 = prom2.then(fn, fn);
```

```
var prom4 = prom3.then(fn, fn);
```

Die Kette ist in dem Sinne *voneinander abhängig*, als dass, sobald das primäre Promise „settled“ ist, alle anderen ebenfalls ihre Callback-Chains feuern. Jedes Promise der Kette wird durch sein Vorgänger-Promise

getriggert und veranlasst, entweder seine *Success*- oder seine *Error*-Callbacks zu feuern.

Hier wird das Promise *fulfilled*:

```
var prom1 = new Promise(function(resolve, reject) {
    // resolve triggert Success Stufe 1
    resolve(42);
});

// Success Stufe 1 wird getriggert
var prom2 = prom1.then(function(resp) {
    console.log("Success Stufe 1:", resp);
    // hier kann eine Verarbeitung stattfinden:
    return resp * 2;
}, function(err){
    console.log("Fehler Stufe 1", err);
});

// Success Stufe 2 wird getriggert
prom2.then(function(resp) {
    // hier kommen verarbeitete Daten an:
    console.log("Success Stufe 2:", resp); // 84
}, function(err){
    console.log("Fehler Stufe 2", err);
});
```

- ✓ Es erscheint zwar naheliegend, dass bei einem *fulfilled* Promise automatisch die Success-Callbacks aller nachfolgenden Promises getriggert werden. Dies ist jedoch nicht zwangsläufig so!

Jetzt wird das Promise rejected:

```
var prom1 = new Promise(function(resolve, reject) {
    // resolve triggert Success Stufe 1
    reject("Fehler");
});

// Error Stufe 1 wird getriggert:
var prom2 = prom1.then(
function(resp) {
    // Success (nicht getriggert)
    console.log("Success Stufe 1:", resp);
}, function(err){
    console.log("Fehler Stufe 1", err);
});
```

```
        return "Fehler";
    });

    // Success Stufe 2 wird getriggert (!)
    prom2.then(function(resp) {
        console.log("Success Stufe 2:", resp); // Fehler
    },
    function(err){
        console.log("Fehler Stufe 2", err);
    });
});
```

- ✓ Hier triggert der *Return* im *Error-Callback* der Stufe 1  
(unerwarteterweise?) tatsächlich den *Success-Callback* der Stufe 2.

Um auch dort den *Error-Callback* zu triggern, muss anstelle eines *Returns* ein **Fehler** geworfen werden. Also nochmal:

```
var prom1 = new Promise(function(resolve, reject) {
    // resolve triggert Success Stufe 1
    reject("Fehler");
});

// Error Stufe 1 wird getriggert:
var prom2 = prom1.then(
function(resp) {
    // Success (nicht getriggert)
    console.log("Success Stufe 1:", resp);
}, function(err){
    console.log("Fehler Stufe 1", err);
    throw new Error("Fehler");
});

// Error Stufe 2 wird getriggert
prom2.then(function(resp) {
    console.log("Success Stufe 2:", resp);
}, function(err){
    console.log("Fehler Stufe 2", err);
});
```

**Fazit:** Die Wahl der Success- oder Errorverarbeitung der jeweils folgenden Stufe wird allein durch (kontrollierten) Einsatz von *Return* oder *Throw* gesteuert.

- ✓ Durch **kontrollierten Return** ist eine Datenweitergabe und -verarbeitung in einer Promise-Chain möglich.
- ✓ Durch **kontrollierten Throw** lässt sich, egal ob aus Success- oder Error-Callback der Error-Callback der Folgestrufe triggern.

### 10.1.5. Promises und Ajax

---

Einen XMLHttpRequest zu bilden ist keine Kunst, allerdings muss die Ergebnisbehandlung bereits beim Formulieren des Requests feststehen, da die Bindung der Callbacks erfolgen muss, *bevor* der Request abgeschlossen ist.

An dieser Stelle kommt das Promise ins Spiel. Die Überlegung ist, die Definition des Requests in eine Funktion zu verkapseln und von außen zu konfigurieren (indem einfach der *URL der Ressource* übergeben wird). Die Funktion soll ein Promise zurückgeben, dessen Settling durch den Request kontrolliert wird.

```
function ajaxMitPromise(url) {  
    // Promise erzeugen und zurückgeben:  
    return new Promise(function(resolve, reject) {  
        // hier XHR-Prozess definieren  
        // Callbacks binden für Resolve/Reject  
        // und Request absetzen  
    });  
}
```

An das zurückgegebene Promise können mittels der `then()`-Methode beliebig Callbacks gebunden werden.

```
var ajaxPromise = ajaxMitPromise('beispiel.json');  
ajaxPromise.then(fnSuccess, fnError);
```

Im Inneren wird ein normaler Ajaxvorgang definiert. Der `onload`-Handler feuert nach Abschluss des Requests. Anhand des **Status** wird entschieden, ob ein Resolve oder ein Reject vorgenommen wird. Für den Fall, dass ein **Netzwerkfehler** gemeldet wird (das ist *nicht* dasselbe wie ein 404!), wird noch ein `onerror`-Handler definiert, er ebenfalls einen Reject vornimmt.

Hier ist die fertige Funktion:

```
function ajaxMitPromise(url) {
  // Promise erzeugen und zurückgeben:
  return new Promise(function(resolve, reject) {
    // hier ajaxen
    var req = new XMLHttpRequest();
    req.open('GET', url, true);

    req.onload = function() {
      // Status muss 200 sein!!
      if (req.status == 200) {
        // Promise mit response auflösen:
        resolve(req.response);
      } else {
        // ...ansonsten Fehler melden
        reject(Error(req.statusText));
      }
    };

    // schlechtes Netzwerkwetter behandeln:
    req.onerror = function() {
      reject(Error("Blöder Network Error!"));
    };

    // ... und abschicken
    req.send();

  }); // Ende Promise-Konstruktor
} // Ende Funktion
```

Die Funktion kann nun eingesetzt werden. Der Vorteil ist, dass das Binden der Callbacks für die Verarbeitung nun vom eigentlichen Erzeugen des Requests **abgekoppelt** ist.

Wie jetzt klar ist, ist es gleichgültig, ob zu dem Zeitpunkt, zu dem die Bindung erfolgt, der Prozess schon abgeschlossen ist, oder noch andauert.

```
// Dieser Prozess wird erfolgreich abgeschlossen:
var userPromise = ajaxMitPromise('ajax/users.json');

userPromise.then(function(response) {
  console.log("Erfolg!", response);
  return response;
}, function(error) {
  console.error("War wohl nix!", error);
});
```



Folgenden Log sehen wir:

```
Erfolg!
{
  "users": {
    "peter": {
      "vorname": "Peter",
      "nachname": "Panter"
    },
    ...
  }
}
```

Schlägt der Request fehl, weil *keine Datei* gefunden wird (404), so passiert folgendes:

```
var userPromise = ajaxMitPromise('gibtsnicht.json');
// etc. ...

"NetworkError: 404 Not Found - ..."
War wohl nix!
Error: Not Found
```

Es greift also der **Error-Callback** von Stufe 1, der durch seinen *Throw* den Error-Callback am `userPromise` triggert.

Nehmen wir jedoch einen **unerlaubten Request** vor, beispielsweise „cross-domain“, so greift der `onerror`-Callback von Stufe 1:

```
var userPromise = ajaxMitPromise('http://sonstwo.de');
// etc. ...

"NetworkError: 404 Not Found -
http://www.sonstwo.de/...."
War wohl nix!
Error: Blöder Network Error!
```

### 10.1.6. Synchron/asynchroner Getter

---

Manchmal ist es nicht im Vorfeld klar, ob eine Funktion, die einen Wert liefern soll, hierfür einen *asynchronen* Prozess starten muss, oder nicht.

- ✓ In diesem Fall gibt man grundsätzlich, also auch, wenn ein Wert unmittelbar (d.h. synchron) verfügbar ist, ein `Promise` zurück, um eine

**Gleichbehandlung** des synchronen und des asynchronen Falles zu erreichen:

Nehmen wir an, wir besäßen ein Cache-Objekt mit Userdaten:

```
var userCache = {  
  "peter": {  
    "vorname": "Peter",  
    "nachname": "Panter"  
  },  
  // etc.  
};
```

Auf dem Server gibt es ein Verzeichnis `ajax/`, das Datensätze als JSON enthält, wie beispielsweise:

*heiner.json*

```
{ "vorname": "Heiner", "nachname": "Hecht" }
```

Eine Funktion soll die Userdaten zurückgeben. Entweder direkt aus dem Cache, oder vom Server, aus dem JSON-Repository:

```
function getUserDetail(user) {  
  
  // es wird IMMER ein Promise zurückgegeben  
  if (userCache[user]) {  
    console.log("User im Cache.");  
  
    // Promise mit Cachedaten zurückgeben:  
    return Promise.resolve(userCache[user]);  
  
  } else {  
  
    // nicht im Cache? Server fragen:  
    console.log("Nicht im Cache. Ajaxe Daten.");  
  
    // Promise für Ajaxdaten zurückgeben:  
    return ajaxMitPromise('ajax/' + user + '.json')  
  
    // hier schon Promise Stufe 1 definieren:  
    .then(function (result) {  
      // Daten in den Cache legen:  
      userCache[user] = result;  
    })  
  }  
}
```

```
        console.log("Success Stufe 1");
        // und an Stufe 2 weiterreichen:
        return JSON.parse(result);

    }).catch(function () {
        console.log("Error Stufe 1");

        // Error Callback Stufe 2 triggern:
        throw new Error('Kein User "' + user + '"!');
    });
}
}
```

Wir probieren das mal aus. Der erste Datensatz wird aus dem Cache geholt:

```
var peterPromise = getUserDetail('peter');

peterPromise.then(function(resp){
    console.log(resp);
    console.log("Cache:",JSON.stringify(userCache));
    // Cache unverändert
},function(err){
    console.log(err);
}); // im Cache
```

Der zweite Datensatz ist nicht im Cache. Er sollte per Ajax geholt werden und dann in den Cache geschrieben, was wir im Success-Callback kontrollieren:

```
var heinerPromise = getUserDetail('heiner');

heinerPromise.then(function(resp){
    console.log(resp)
    console.log("Cache:",JSON.stringify(userCache));
    // Heiner sollte jetzt drin sein!
},function(err){
    console.log(err);
}); // nicht im Cache
```

Die dritte Abfrage gilt einem User, für den kein Datensatz existiert. Der Ajaxprozess führt daher zu einem Fehler, der direkt den Errorcallback der Stufe 1 triggert, der einen Error zu Stufe 2 (die wir bilden) weiterreicht. Es wird also der Fehlercallback des Promises getriggert.

```
var josefPromise = getUserDetail('josef');

josefPromise.then(function(resp){
    // Success wird hier nicht getriggert...
},function(err){
    console.log(err);
}); // nicht im Cache
```

Der Ablauf führt zu folgender Logreihenfolge. Zuerst werden die drei Promises für *Peter*, *Heiner* und *Josef* gebildet:

```
User im Cache.
User nicht im Cache. Kontaktiere Server.
User nicht im Cache. Kontaktiere Server.
```

Das Auflösen erfolgt nun asynchron.

```
Object { vorname="Peter", nachname="Panter"}
```

```
Cache: {"peter": {"vorname": "Peter", "nachname":
"Panter"}, ... , "leo": {"vorname": "Leo", "nachname":
"Löwe"}}
```

*Peter* wird aus dem Cache geholt und zurückgegeben. Nun meldet der Ajaxcall für *Heiner* „Success“ und löst sein Promise auf. *Heiner* wird in den Cache geschrieben:

```
Success Stufe 1
Object { vorname="Heiner", nachname="Hecht"}
```

```
Cache: {"peter": {"vorname": "Peter", "nachname":
"Panter"}, ..., "heiner": {"vorname": "Heiner",
"nachname": "Hecht"}}
```

Nun wird der Fehlschlag beim Holen des *Josef*-Datensatzes bemerkt und die Fehlerbehandlung durchgeführt:

```
"NetworkError: 404 Not Found - ..."
```

```
Error Stufe 1
Error: User "josef" nicht gefunden.
```

## 10.2. Generatoren

Auf's Elementare heruntergebrochen, repräsentiert ein **Generator** (genauer, ein „Generator-Objekt“) eine Folge von Werten (eine Sequenz). In diesem Sinne ist er ein **Iterator-Objekt**, ähnlich dem, das man aus einem „Iterable“, wie einem Array (ebenfalls eine Folge von Werten) oder einem String (eine Folge von Characters) ableitet. Ein *Generator-Objekt* muss jedoch nicht eine „finite“ Sequenz darstellen.

- ✓ Ein **Generator-Objekt** ist einem *Iterator-Objekt* sehr ähnlich. Beide besitzen eine Methode `next()`, die den nächsten anstehenden *Wert* der Sequenz zurückgibt (in einem Array zusammen mit seinem *Index*) und ein `done`-Property verwaltet, das nach Durchlaufen der Sequenz auf `true` gesetzt wird.

Ein Generator-Objekt (kurz als Generator bezeichnet) muss ebenfalls *erzeugt* werden, analog zu dem Iterator-Objekt, das aus einem Iterable abgeleitet wird.

### 10.2.1. Die Generatorfunktion `function*` („function star“)

Zum Erzeugen von Generatoren verwenden wir eine **Generatorfunktion**, die deinen Generator zurückgibt (ohne dabei aber ein Konstruktor zu sein!).

Die `function*`-Deklaration („*function star*“, das Keyword `function` gefolgt von einem Stern `*`) definiert eine solche *Generatorfunktion*, welche ein *Generator-Objekt* zurückgibt.

```
function* meineGeneratorfunktion() {  
    // tut was generatormäßiges  
}
```

Eine Generatorfunktion könnte auch mittels einer `function*`-Expression definiert werden:

```
var meineGeneratorfunktion = function* () {  
    // tut was generatormäßiges  
};
```

Es existiert im Prinzip auch ein *Konstruktor* `GeneratorFunction()`, welcher allerdings *nicht* global als Objekt verfügbar ist, sondern (bei Bedarf) von einem `function*`-Literal abgeleitet werden muss:

```
var GeneratorFunction =  
  Object.getPrototypeOf(function*({})).constructor;
```

Dies ist eher eine prinzipielle Möglichkeit, als eine gebräuchliche Herangehensweise.

Hier ist eine Generatorfunktion, die einen Generator zurückgibt, der nacheinander die Reihe der Ganzzahlen zur Verfügung stellt.

```
function* ganzeZahlen () {  
  var n = 1;  
  while (true){  
    yield n++;  
  }  
}
```

✓ Beim `yield`-Keyword (*yield* = „etwas ergeben“) **stoppt** der Generator, um einen *Wert* zurückzugeben.

Es handelt sich bei der Generatorfunktion zwar nicht um eine *Funktion im eigentlichen Sinne*, sondern um einen *eigenen Typ*. Ähnlich einer Funktion ist die Generatorfunktion allerdings „**callable**“, kann also aufgerufen werden.

Tun wir dies, so gibt die Generatorfunktion einen Generator zurück:

```
// beim Aufruf kein Unterschied zur Funktion:  
var zahlenReihe = ganzeZahlen();  
console.log(zahlenReihe.next());  
//-> { value=1, done=false }  
  
console.log(zahlenReihe.next());  
//-> { value=2, done=false }  
  
console.log(zahlenReihe.next());  
//-> { value=3, done=false }
```

✓ Die Aufrufe von `next()` könnten ewig fortgesetzt werden: Dieser Generator repräsentiert eine **unendliche Reihe** (*infinite sequence*).

### 10.2.2. Generator-Objekt und for-of Schleife

Da ein Generator, wie oben gesagt, ein Iterable-Objekt ist, kann auf ihn auch *direkt* eine **for-of-Schleife** ausgeführt werden. Mit einer *infiniten* Sequenz ist dies keine gute Idee. Lassen wir den Generator also nur die Zahlen bis 5 ausgeben und modifizieren hierfür die Generatorfunktion.

```
function* ganzeZahlenBis5() {  
  var n = 1;  
  while (n <= 5){  
    yield n++;  
  }  
}
```

#### Zwischenbemerkung:

Die Syntax der *Starfunktion* verlangt lediglich den **Stern** zwischen function-Keyword und function.name. Seine **Position ist beliebig**.

Wir können also nach Wahl wie folgt schreiben:

```
function* ganzeZahlenBis5() { ... }  
function * ganzeZahlenBis5() { ... }  
function *ganzeZahlenBis5() { ... }
```

... oder sogar (sic!):

```
function*ganzeZahlenBis5() { ... }
```

Unser Ganzzahlengenerator arbeitet mit entsprechend handlicherer Sequenz nun so

```
var zahlenReiheBis5 = ganzeZahlenBis5();  
  
for(let value of zahlenReiheBis5) {  
  console.log(value);  
}  
// -> 1  
// -> 2  
// -> 3  
// -> 4  
// -> 5
```

... und stoppt dann automatisch.

**Trick: Direkter Aufruf der Generatorfunktion als Iterable:**

Soll die Sequenz einer Generatorfunktion *direkt* ausgegeben werden, so ist dies mit der *for-of* Schleife auch möglich, *ohne* im Vorfeld einen Generator zu erzeugen.

```
for(let value of ganzeZahlenBis5()) {  
    console.log(value);  
}
```

Hierfür wird die Generatorfunktion als *Operand* von *for-of* **aufgerufen**.

### 10.2.3. Boxenstopp im Code - „code suspension“

Man kann ein Generator-Objekt auch als eine „Art von Codeblock“ betrachten, in dem **Pausen** (*yield*) eingebaut sind, und in zu einem beliebigen Zeitpunkt wieder ein **Einsprung** (*resume*) möglich ist, nämlich in der auf den *yield*-Keyword folgenden Zeile.

- ✓ Im vorigen Beispiel ist dieser Fakt durch die Schleife verschleiert worden. Das Prinzip gilt dort jedoch bereits, jedoch handelt es sich um ein „wiederholtes“ Yield, nicht um eine Folge von Yields.

Schreiben wir uns eine Generatorfunktion, die einen Generator macht, der Zeile für Zeile einen Haiku ausgibt:

```
function * haikuZitieren() {  
    console.log(' Der alte Weiher:');  
    yield null;  
  
    // 1. Resume: Hier wieder einsteigen:  
    console.log('Ein Frosch springt hinein.');
```

**yield null;**

```
    // 2. Resume: Hier wieder einsteigen:  
    console.log('Oh! Das Geräusch des Wassers.');
```

**yield null;**

```
    // 3. Resume: Hier wieder einsteigen:  
    console.log('- Matsuo Bashō');
```

// wir sind durch :-)

```
}
```



- ✓ In diesem Fall ist der *Rückgabewert* des Yields unerheblich und wird auf `null` gesetzt. Wir interessieren uns nur für die *Seitenwirkung* des Generators, nämlich den Log.

```
var bashoHaiku = haikuZitieren();
```

Um das Zitat zu starten, muss `next()` aufgerufen werden. So bewegt man sich im Anschluss Zeile für Zeile durch das Gedicht:

```
bashoHaiku.next();  
bashoHaiku.next();  
bashoHaiku.next();  
bashoHaiku.next();
```

```
// -> Der alte Weiher:  
// -> Ein Frosch springt hinein.  
// -> Oh! Das Geräusch des Wassers.  
// -> - Matsuo Bashō
```

- ✓ Es wird deutlich, dass der Generator von sich aus seine Arbeit *weder aufnimmt* noch nach einer Pause *fortsetzt*. Stets brauchen wir `next()`.
- ✓ Auch deutlich ist, dass die Resumepunkte durch die Position der Yields festgelegt sind. Die **Reihenfolge** der Schritte ist daher **determiniert**.

**Frage:**

Warum sollte so ein Arbeitsschritt also nicht einen **asynchronen Prozess** starten, und mit dem Aufruf von `next()` *nach dessen Abschluss* der *nächste* Schritt aufgerufen werden?

- ✓ **Bingo**, genau darum geht es!

**Nebenbemerkung:**

In diesem speziellen Fall, wo der *Yield-Payload* keine Rolle spielt, kann der Generator dennoch auch über die *for-of* Schleife betrieben werden, wobei wieder die Generatorfunktion direkt gerufen wird. Der Schleifenblock kann leer bleiben:

```
for(let value of haikuZitieren()) {  
    // Wir verwenden den value nicht!!  
}  
  
// -> Der alte Weiher:  
// -> Ein Frosch springt hinein.  
// -> Oh! Das Geräusch des Wassers.  
// -> - Matsuo BashM
```

#### 10.2.4. Stopp für Input - Iteration Messaging

Zuvor muss geklärt werden, wie in einen laufenden Generatorprozess Werte hineingelangen können. Zum Glück hält der Generator bei jedem *Yield* an, um auf die Anweisung zum nächsten Schritt zu warten.

Folgendes gilt:

- ✓ Platziert man einen *Yield* in eine laufende **Zuweisung**, so wird diese **exakt** an diesem Punkt unterbrochen. Der folgende Aufruf von `next()` setzt **exakt** an diesem Punkt ein. Übergibt man an `next()` einen **Wert** (*iteration message*), so wird dieser an **exakt** dieser Stelle in den Code eingefügt.

Hierzu ein kurzes Beispiel:

```
function * partialMultiply(x) {  
    var y = x * (yield "Gib mir ein y!");  
    return y;  
}  
  
var mult6 = partialMultiply(6);  
  
// Start des Generators:  
var msg = mult6.next().value;  
console.log(msg);           // -> Gib mir ein y!  
  
// Eingabe des zweiten Wertes:  
var ergebnis = mult6.next(7).value;  
console.log(ergebnis); // -> 42
```

Die Vorrichtung funktioniert. Aber warum? Der erste *Yield* wird beim *ersten* `next()` erreicht, also gleich beim Start des Programms. Der Wert 6, der beim Erzeugen des Generators hineingereicht wurde, ist bereits an den Platz der Variablenreferenz auf den Parameter getreten:

```
function * partialMultiply(x) {  
    var y = 6 * (yield "Gib mir ein y!");  
    return y;  
}
```

Der *Yield* gibt ein Objekt mit dem String als `value` zurück und stoppt den Generator während der Zuweisung. Beim folgenden `next()` wird der Wert 7 übergeben. Er erscheint **exakt** am Punkt der Unterbrechung, sodass quasi folgende Situation entsteht:

```
function * partialMultiply(x) {  
    var y = 6 * (7);  
    return y;  
}
```

Die Multiplikation wird nun durchgeführt und das Ergebnis 42 zurückgegeben. Fertig.

### 10.2.5. Werte erzeugen mit Generator und for-of

---

Mit einem Generator können Werte erzeugt und mit jedem Schritt zurückgegeben werden. Hier ist eine Generatorfunktion, die eine im Prinzip unendliche Zahlenreihe erzeugen kann:

```
function * gibMirEineZahl() {  
    var nextVal;  
    while (true) {  
        if (nextVal === undefined) {  
            nextVal = 1;  
        } else {  
            nextVal = (2 * nextVal) + 4;  
        }  
        yield nextVal;  
    }  
}
```

Den von der Generatorfunktion abgeleiteten *Iterator* können wir mit *for-of* auslesen. Die `while(true)`-Konstruktion hält den Generator am Laufen.

Da die Schleife daher ebenfalls endlos lief, bauen wir eine **Break-Bedingung** ein, die einen Schwellenwert prüft und bei dessen Überschreitung die Schleife verlässt:

```
for (var z of gibMirEineZahl()) {  
    console.log(z); // wir loggen nur aus  
    // Raus, sobald die Zahl größer als 300 wird  
    if (z > 300) {  
        break;  
    }  
}  
// -> 1, 6, 16, 36, 76, 156, 316
```

Interessant ist hier, dass der Iterator seinen **Zustand speichert**, solange er „suspended“ ist (sich also zwischen zwei Yields befindet), vergleichbar mit einer Closure.

Erinnern wir uns, dass dies ein Generator ist, der eine **infinite Sequenz** erzeugt. Nach dem Break könnte der Iterator daher brachliegen und bis ans Ende aller Tage **Speicherplatz** blockieren. Dies ist zum Glück nicht der Fall.

- ✓ Zwischen der *for-of*-Schleife und „ihrem“ Iterable findet eine Kommunikation statt. Die Schleife selbst *benachrichtigt* bei ihrem Ende (also auch bei einem Break) den **Iterator**, dass er sich **beenden** soll. Die Schleife „räumt also hinter sich auf“.

Bei seinem **Abschluss** kann der Iterator noch eine Aktion vornehmen. Voraussetzung hierfür ist, dass wir den „aktiven“ Teil des Iterators in einen **try-Block** einschließen, und diesem einen **finally-Block** folgen lassen. Der Generator wird wie folgt erweitert:

```
function * gibMirEineZahl() {  
    try {  
        var nextVal;  
        while (true) {  
            if (nextVal === undefined) {  
                nextVal = 1;  
            } else {  
                nextVal = (2*nextVal) + 4;  
            }  
            yield nextVal;  
        }  
    }  
}
```

```
// wenn der Iterator beendet wird:
finally {
    console.log( "Feierabend. Letzte Zahl:", nextVal);
}
}
```

Um den *finally*-Block zu triggern muss der Iterator allerdings durch seine `return()`-Methode beendet werden. Hierfür müssen wir ihn in der Schleife *referenzieren* können, weshalb er sinnvollerweise *vor* der Schleife erzeugt wird. Die Schleife sieht jetzt etwas anders aus:

```
var iterator = gibMirEineZahl();

for (var z of iterator) {
    console.log(z);
    // Raus, sobald Zahl größer als 300:
    if (z > 300) {
        // beenden:
        iterator.return("Tschau!");
    }
}

// -> 1, 6, 16, 36, 76, 156, 316
// -> Feierabend. Letzte Zahl war: 316
// -> { value="Tschau!", done=true}
```

- ✓ Es muss **kein Break** in der Schleife gesetzt werden, da `return()` den Iterator in den `done:true`-Zustand versetzt. Die *for-of*-Schleife stoppt dann automatisch.

### 10.2.6. Ajax mit Generator

Wir wollen nun mit einem Generator eine Reihe von **Ajax-Requests** steuern. Zunächst benötigen wir eine Funktion, die einen Request erzeugt und absetzt. Sie soll den URL und ein Callback entgegennehmen, das den Request behandelt.

```
function ajax(url, cb) {
    var req = new XMLHttpRequest();
    req.open('GET', url);
    req.onload = cb;
    req.send();
}
```

Desweiteren brauchen wir eine Funktion, die zwischen dem Generator und der Ajax-Funktion vermittelt. Nennen wir sie den **Ajax-Runner**:

```
function ajaxRunner(url) {  
    ajax(url, function () {  
        if (this.status == 200) {  
            // den Iterator triggern:  
            iterator.next(this.responseText);  
        }  
        else {  
            // ...ansonsten Fehler melden  
            iterator.throw(new Error(this.statusText));  
        }  
    });  
}
```

- ✓ Der Ajax-Runner muss natürlich den Iterator kennen, der die Arbeit macht. Er setzt voraus, dass ein Objekt namens `iterator` existiert. Dieses Objekt wird in den Callback eingewoben, der der Ajax-Funktion übergeben wird. Der Ajax-Request „bedient“ also den Iterator!

Nun brauchen wir noch die Generatorfunktion, die wir `main` nennen. Wir versichten auf eine Fehlerbehandlung (hierfür könnten wir noch einen *try-catch*-Block einfügen):

```
function * main() {  
    // ajaxen...  
    var res = yield ajaxRunner("ajax/benno.json");  
    // ... und das Ergebnis ausgeben:  
    console.log(res);  
}
```

Es bleibt nur noch, die ganze Vorrichtung zu starten:

```
// Hier kommt der Iterator:  
var iterator = main();  
  
// ... und los geht's:  
it.next();  
  
// -> { "vorname": "Benno", "nachname": "Bär" }
```

Es klappt.

- ✓ Es fällt auf, dass die Anweisung in der `main`-Funktion wunderbar **synchron** aussieht, zumindest auf den ersten Blick:

```
// ajaxen...
var res = yield ajaxRunner("ajax/benno.json");
// ... und das Ergebnis ausgeben:
console.log(res);
```

- ✓ Der springende Punkt ist der *Yield*, der die weitere Ausführung des Iterators an **exakt** dieser Stelle stoppt.

Wie wir inzwischen wissen, wird der *Suspension-Point* nach dem folgenden `next()` durch den **Input** gefüllt, der an `next()` übergeben wird.

Dies geschieht aber **aus dem Ajax-Callback**:

```
// den Iterator triggern:
iterator.next(this.responseText);
```

An `next()` übergeben wird das **Ergebnis des Ajax-Requests**, sodass nun quasi folgendes am Suspension-Point erscheint:

```
// ajaxen...
var res = '{"vorname": "Benno", "nachname": "Bär"}';
// ... und das Ergebnis ausgeben:
console.log(res);
```

Q.E.D. Es ist gelungen, im Generator (indirekt) einen Ajax-Call zu formulieren und in quasi synchroner Weise auf dessen Resultat zurückzugreifen.

Das Schöne ist, dass wir das ebenso gut mit **mehreren Ajax-Calls** machen könnten. Nehmen wir an, wir würden per Ajax Userdatensätze einsammeln und diese in einen Array-Cache ablegen wollen.

In die `main`-Generatorfunktion müssten lediglich weitere Ajax-Calls über den Ajax-Runner vereinbart werden. Der Ajax-Callback startet **automatisch** den nächsten Request, sobald der Anstehende beendet ist. Wir brauchen nach wie vor den **Prozess nur zu starten**, er läuft dann allein bis zum Ende.

```
var users = [];

function * main() {
```

```
var res;

// 1. Request:
res = yield ajaxRunner("ajax/benno.json");
users.push(JSON.parse(res));

// 2. Request:
res = yield ajaxRunner("ajax/emil.json");
users.push(JSON.parse(res));

// 3. Request:
res = yield ajaxRunner("ajax/kuno.json");
users.push(JSON.parse(res));

// u.s.w. ... :-)

console.log(users);
}

// -> [{ vorname="Benno", nachname="Bär"},
      { vorname="Emil", nachname="Elefant"},
      { vorname="Kuno", nachname="Karpfen"}]
```

- ✓ Das eigentlich Wichtige ist, dass die **Reihenfolge der Requests** nun **determiniert** ist.

Das bedeutet, dass (nicht nur theoretisch) das *Ergebnis* von Request 1 vorliegt, *bevor* Request 2 abgesetzt wird (man könnte also seine Daten in den folgenden Request einbeziehen).

Hüllt man die Einzelrequests nun noch in *try-catch*-Blöcke, so kann jeder der Requests für sich fehlschlagen, ohne dass die `main`-Funktion insgesamt gestoppt wird:

```
var users = [];

function * main() {
  var res;
  try {
    res = yield ajaxRunner("ajax/benno.json");
    users.push(JSON.parse(res));
  } catch(err) {
    console.error( err );
  }
}
```



```
    try {
      // hier wird ein Fehler provoziert:
      res = yield ajaxRunner("XXX/XXXX.json");
      users.push(JSON.parse(res));
    } catch(err) {
      console.error( err );
    }
    try {
      res = yield ajaxRunner("ajax/kuno.json");
      users.push(JSON.parse(res));
    } catch(err) {
      console.error( err );
    }
    console.log(users);
  }
// -> Error: Not Found
// -> [{ vorname="Benno",  nachname="Bär"},
      { vorname="Kuno",  nachname="Karpfen" } ]
```

Im Ergebnis-Array fehlt lediglich der Datensatz, dessen Request fehlschlug. Wegen des `try`-Blocks springt der Generator zum folgenden *Yield* und setzt entsprechend seine Arbeit mit dem folgenden Request fort.

## 11. Modularisierung

---

Der Ursprung des Modulgedankens stammt aus der sog. „**modularen Programmierung**“, in der eine Applikation in eine Anzahl austauschbarer, voneinander unabhängiger **Komponenten** untergliedert wird. Jede dieser Komponenten ist dabei autark in dem Sinne, als dass sie alles enthält, was ihr die Arbeit bezüglich ihrer jeweiligen Aufgabe ermöglicht. Eine solche Komponente bezeichnet man als **Modul**.

Folgende Vorteile ergeben sich aus dem modularen Ansatz:

- **Übersichtlichkeit**  
Anstelle einer potentiell unübersichtlichen, monolithischen Anwendung treten Einzelkomponenten mit genau beschriebenem Aufgabenbereich.
- **Austauschbarkeit**  
Sofern bei einer Komponente die Schnittstelle erhalten bleibt, kann sie einfach durch eine verbesserte Variante ersetzt werden.

- **Testbarkeit**

Da eine Komponente nur eine Aufgabe erfüllt, ist sie separierbar und damit einfacher zu testen als eine Gesamtanwendung

- **Wiederverwendbarkeit**

Eine Komponente mit beschriebener Schnittstelle kann in anderen Anwendungen für dieselbe Aufgabe wiederverwendet werden.

- **Verkapselung**

Das Innenleben eines Moduls wird nur über seine Schnittstellen nach außen gegeben. Ein Eingriff von außen und unerwünschte Wechselwirkungen werden vermieden.

In ECMA6 wird erstmalig (aus JavaScript-Perspektive) eine **Syntax zur Formulierung von Modulen** sowie ein **Modulladersystem** in den Sprachkern einbezogen.

- ✓ In ECMA5 gab es noch keine spracheigene Schreibweise für Module. Das Bedürfnis nach Modularisierung existierte gleichwohl, weshalb allerhand Ansätze existieren, dieses Manko zu umschiffen.

### 11.1. IIFE-basierte Modulpattern

---

Bereits in ECMA5 gab es Pattern und Mechanismen, die auf die Modularisierung von Anwendungen hinarbeiteten. Man muss hier unterscheiden zwischen einer Modulbildung auf *Dateiebene* (Dateimodule) und *logischen Modulen* (auf Anwendungsebene).

Um Module auf Dateiebene zu nutzen, muss eine Vorrichtung zum **Laden** eben dieser Dateien herangezogen werden. Diese existiert in ECMA5 nicht nativ, kann aber einfach mittels generierter Scriptblöcke(per `createElement()`) realisiert werden. Alternativ existieren Loader-Frameworks wie **RequireJS**, die erhöhten Komfort bieten, aber auch Ansprüche in Bezug auf die Ausgestaltung der Module (AMD-Pattern).

- ✓ Um Module lediglich *laden* und sich nicht um die *Ausführung* kümmern zu müssen, bietet sich die Formulierung der Module mittels IIFE-Wrappern an.

### 11.1.1. Das „revealing“ und das „global Import“-Pattern

Die bekanntesten verwendeten diesbezüglichen Pattern sind das „**revealing Module**“-Pattern und das „**global Import**“-Pattern. Beide werden im Rahmen des Quellcodes üblicher JavaScript-Frameworks eingesetzt.

Hier ein einfaches „revealing Module“. Der IIFE-Scope dient dazu, lokale („geheime“) Modulaspekte zu bilden, die dann über ein *Return* kontrolliert in Teilen nach außen gegeben werden:

```
var revMod = (function(name){  
  
    // lokale, also "private" Variablen  
    var name = name || "User";  
    function hallo() {  
        console.log( "Hallo " + name + "!" );  
    }  
  
    // public API (die Modulaspekte "enthüllt")  
    return {  
        // lokale Funktion nach außen geben:  
        hallo: hallo  
    };  
})( "Peter" );
```

- ✓ Das Essentielle an diesem Pattern ist die so mögliche **Kapselung** und die klar definierte öffentliche **Schnittstelle**: Nur die zu verwendenden Teile des Moduls werden gezeigt (*revealing*-Prinzip).

Hier ein einfaches „Global Import“ Module. Auch in diesem Fall wird mit Hilfe der IIFE ein *Scope* gebildet und verwendet. Im Inneren des Moduls wird jedoch auf die „Außenwelt“, sprich, die *Umgebung* zugegriffen, hier durch den Zugriff auf das `windows`-Objekt.

- ✓ Um den Zugriff zu **entkoppeln** wird beim „global Import“ der Teil der Umgebung, den das Modul benötigt, gezielt **übergeben**.

Wir finden hier am „Eingang“ der IIFE daher das `windows`-Objekt, das hiermit im Funktionsblock des Moduls „nur noch“ eine lokale Variable darstellt. Im Code muss daher nicht mehr in die Umgebung „gegriffen“ werden, was einer Entkopplung gleichkommt (außerdem könnte ein abweichender lokaler Bezeichner gewählt werden):

```
(function(window){
  // lokale Funktionen definieren:
  var plus = function(x, y){
    return x + y;
  }

  var minus = function(x, y){
    return x - y;
  }
  // API-Objekt formulieren:
  var berechne = {
    summe: function(a, b){
      return plus(a,b);
    },
    differenz: function(a, b){
      return minus(a, b);
    }
  }
  // Objekt nach außen binden
  window.berechne = berechne;
})(window);
```

- ✓ Eine Kombination aus „revealing Module“ und „Importmodule“ ist das „Erweiterungsmodul“-Pattern, wo sich mehrere Module unabhängig von einer Ladereihenfolge ineinander integrieren können.

Die drei Pattern „revealing Module“, „global Import“ und „Erweiterungsmodul“ werden im Basisseminar zu JavaScript behandelt.

Wir verlassen nun das Gebiet der „mit Bordmitteln“ von ECMA5 darstellbaren Modulpattern. Das eigentliche Laden der Moduldateien würde über ein **generiertes Script-Element** erfolgen. Anschließend müsste die Einsatzbereitschaft (spricht, der Ladezustand) der Module getestet werden, um auf die Module zuzugreifen. Dies erfordert das Schreiben weiterer Logik.

- ✓ **Nicht ohne weiteres geregelt** ist bei all diesen Pattern die Frage der eventuellen **Abhängigkeit** eines Moduls von einem anderen Modul und dessen *automatische* Einbeziehung in den Ladevorgang (*Dependency-Management*).

## 11.2. CommonJS Module, AMD- und UMD-Moduldefinitionen

---

Wie eben gezeigt, ist es möglich, wenn auch unbequem, in reinem JavaScript Module und einen Modullader zu schreiben.

Aus diesem Gründen hat sich ein Ökosystem aus etablierten **Praxisansätzen** herauskristallisiert, in dem gleichermaßen die Schreibweise der Module wie auch deren Laden definiert ist. Wir unterscheiden drei Ansätze:



### **Common JS Module:**

Arbeitet unter NodeJS. Module werden **synchron** importiert. Dies ist möglich, weil NodeJS Zugriff auf das Filesystem des Hostrechners besitzt und das Laden von Dateien auch synchron vornehmen kann.

**Siehe:** [www.commonjs.org/](http://www.commonjs.org/)



### **AMD:**

Die „Asynchronous Module Definition“ ist, im Gegensatz zu *Common JS* für **Frontend-Anwendungen** geeignet. Module liegen im speziellen AMD-Format vor, die auch Abhängigkeiten aussprechen kann und werden stats **asynchron** (nicht blockierend) geladen.

**Siehe:** <http://requirejs.org/docs/whyamd.html>

Die Module, die beide Systeme benötigen, unterscheiden sich jedoch. Um dies zu umgehen, existiert ein Pattern zur Formulierung der Module, sodass diese sowohl in einer IIFE-Umgebung (ohne Modullader), als auch mittels eines Common JS-Laders oder eines AMD-Laders verwendet werden können. Die Module werden hierdurch jedoch komplexer.



### **Universal Module Definition (UMD):**

Das Formulieren eines Moduls gemäß der UMD-API erlaubt dessen Verwendung in **beliebigen Modulumgebungen**, sei es auf dem *Server* (als Common JS Modul) und *synchron* oder auf den *Client* (als AMD) und *asynchron*. Der vielseitigste Ansatz ist das **returnExports-Pattern**.

**Siehe:** <https://github.com/umdjs/umd>

### 11.2.1. Common JS

---

Jedes Common JS Modul muss stets in einer separaten Datei formuliert werden, wobei die nach außen zu gebenden Aspekte des Moduls an das `exports`-Objekt gebunden werden. Über dieses Objekt wird die Modulschnittstelle in der Importumgebung zugänglich gemacht.

Wir formulieren das Rechen-Modul in Common JS wie folgt:

*berechne.js*

```
var plus = function(x, y){
    return x + y;
}
var minus = function(x, y){
    return x - y;
}

// API-Objekt formulieren:
var berechne = {
    summe: function(a, b){
        return sum(a,b);
    },
    differenz: function(a, b){
        return sub(a, b);
    }
}
// Objekt nach außen binden
exports.berechne = berechne;
```

Der **Import** eines solchen Moduls geschieht einfach über den Aufruf der globalen Funktion `require()`, die in der Umgebung vordefiniert ist. Hier muss nochmals darauf hingewiesen werden, dass der Import **synchron** geschieht. Er erfolgt aus einer Datei *index.js*, die von Node aufgerufen wird:

*index.js*

```
// Import und Extraktion der Schnittstelle:
var berechne = require("./berechne").berechne;
// Verwendung unmittelbar möglich:
console.log(berechne.summe(1, 2));      //-> 3
console.log(berechne.differenz(1, 2)); //-> -1
```

✓ Die Dateierweiterung `.js` wird beim Import automatisch ergänzt.

### 11.2.2. AMD

---

Eine clientseitig im Browser verwendete JavaScript-Datei wird üblicherweise synchron mittels Scriptblock geladen. Bei modularen Anwendungen führt dies zu einer Verzögerung des Ladens, weshalb man in diesem Fall gerne auf asynchrones Laden zurückgreift.

Eine Umgebung, die asynchrones Laden spezieller Module ermöglicht, ist **RequireJS** (Dojo-Foundation), das dort globale Funktionen `require()` und `define()` zur Verfügung stellt. Module müssen zur Verwendung durch RequireJS im **AMD-Format** geschrieben sein.

**Siehe:** <http://requirejs.org/>

- ✓ Die Funktion `require()` von RequireJS ist nicht zu verwechseln mit der Funktion `require()` von CommonJS. Beide arbeiten verschieden!

Die Aufgabe von `define()` besteht in der **Formulierung eines Moduls**. Sie wird also bei der eigentlichen Definition des AMD-Moduls herangezogen.

- ✓ Die *hier* demonstrierte Signatur nimmt als einzigen Parameter eine **Callbackfunktion** entgegen.

Unser Rechenmodul sähe in AMD-Schreibweise wie folgt aus.

berechne.js

```
define(function(){
    // lokale Funktionen definieren:
    var plus = function(x, y){
        return x + y;
    }
    var minus = function(x, y){
        return x - y;
    }

    // API-Objekt formulieren:
    var berechne = {
        summe: function(a, b){
            return sum(a,b);
        },
        differenz: function(a, b){
```

```
        return sub(a, b);
    }
}
// Objekt nach außen geben
return berechne;
});
```

Wie man sieht, verwendet dieses AMD-Pattern eine Funktion, die ihren Scope dem Modul zur Verfügung stellt. Es ist in dieser Form im Prinzip eine Abwandlung des „revealing Module“-Pattern.

In der Anwendung wird das Modul über die Funktion `require()` geladen, die **RequireJS** zur Verfügung stellt (das Framework wird daher benötigt). Als ersten Parameter übergibt man ein Array, das Strings enthält, aus denen *RequireJS* den **Pfad zur Moduldatei** erschließt. Der zweite Parameter ist eine **Callbackfunktion** die nach erfolgreichem Laden der Datei feuert. Ihr wird da Modul übergeben, wozu der Parameter `berechne` dient.

*index.js:*

```
// holt berechne.js aus Basisverzeichnis
require(["./berechne"], function(berechne){

    // berechne ist der lokale Name der Modul-API:
    console.log(berechne.summe(1, 2));      //-> 3
    console.log(berechne.differenz(1, 2)); //-> -1

});
```

- ✓ Besitzt ein Modul Abhängigkeiten, so werden diese in der Moduldatei als Array vor der Callback-Funktion des Moduls übergeben und analog zum Require-Aufruf aufgelöst.

### 11.2.3. Universal Module Definition

---

Um gleichermaßen Ladesituationen, die den „klassischen“ IIFE-Ansatz verfolgen, zu bedienen, wie auch solche, die mit Common JS (synchron) oder RequireJS (AMD, asynchron) arbeiten, werden „intelligente“ Module benötigt, sie sich an die in der Ladesituation vorgefundene Umgebung anpassen.



Die „Universal Module Definition“ (UMD) ist dabei weniger ein Pattern, als ein Ansatz, der je nach Ziel mit verschiedenen Pattern arbeitet. Hier wird nur eines davon, nämlich das *returnExport*-Pattern vorgestellt.

- ✓ Das Pattern **returnExport** ist das beliebteste aus dem UMD-Portfolio. Ein so geschriebenes Modul funktioniert gleichermaßen in einer AMD-Umgebung wie in einer Common JS Situation.

Unser bereits bekanntes Rechner-Modul sieht, gemäß UMD formuliert, nun so aus:

berechne.js

```
// Achtung, dies ist eine IIFE !!!
// Global Import: root -> this !!
(function (root, factory) {
  // Prüfen: AMD oder Common JS?
  if (typeof define === 'function' && define.amd) {

    // Yup, das ist AMD! Factory wird übergeben.
    define([], factory);
  } else if (typeof exports === 'object') {

    // Yup, das ist Common JS! Factory wird aufgerufen!
    module.exports = factory();
  } else {
    // ... ansonsten global binden.
    root.returnExports = factory();
  }
})(this, function() {
  // Factory: die eigentliche Modul-Definition
  var plus = function(x, y){
    return x + y;
  }
  var minus = function(x, y){
    return x - y;
  }
  var berechne = {
    summe: function(a, b){
      return sum(a,b);
    },
    differenz: function(a, b){
      return sub(a, b);
    }
  }
```

```
    }  
    return berechne;  
  
  }); // Ende IIFE
```

Genug von der Historie. Ab sofort besitzt JavaScript eine *eigene*, native Art und Weise, wie Module zu formulieren und zu benutzen sind. Es bestehen gewisse Ähnlichkeiten, aber auch Unterschiede zu den soeben vorgestellten Systemen.

### 11.3. ECMA6 Module

Aktuell werden sowohl die **Syntax**, in der Module zu schreiben sind, als auch das dazugehörige **Ladesystem** in Browsern noch nicht unterstützt.

- ✓ Dennoch kann der Einsatz von Modulen in der Syntax von ECMA6 (bzw. ECMA2015) bereits als „*best practice*“ bezeichnet werden, da über kurz oder lang diesbezüglich eine stabile Basis zu erwarten ist.

#### Noch nicht ganz da:

Aktuell wird zum *praktischen Einsatz* dieser Modulsyntax sowohl ein **Transpiler** (wie *TypeScript*, *Babel*, *Traceur*) als auch ein **Bundler/Loader** (wie *Browserify*, *Webpack*, *SystemJS* oder *Rollup.js*) benötigt.

#### 11.3.1. Dateibezogene Module

ECMA6-Module sind dateibezogen, was bedeutet, dass jedes definierende Modul als **eigene Datei** angelegt werden muss.

- ✓ Eine **Moduldatei** wird per Definition stets im „strict Mode“ ausgeführt, *ohne dass* dies eigens deklariert werden muss. Es ist dennoch anzuraten, den „strict Mode“ explizit zu setzen, wenn auch eher aus Gründen der Klarheit und Übersichtlichkeit.

Der „strict Mode“ und seine Implikationen wurden im Basis-Seminar zu JavaScript behandelt.

Folge des „strict Modes“ für Moduldateien ist, dass dort (u.a.) folgendes gilt:

- Eine Variable muss ordnungsgemäß deklariert werden.
- Funktionsparameter benötigen eigenständige Bezeichner
- Das `with`-Statement kann nicht verwendet werden
- Wertzuweisungen an „read-only“-Properties von Objekten verursachen einen Fehler

### 11.3.2. Scoping von Modulen

---

Ein ECMA6-Modul stellt eine Kapsel dar.

In einem Modul deklarierte Variable sind an den **Scope** dieses Moduls gebunden (dies entspricht dem Verhalten von Common JS Modulen).

- ✓ **Variable** aus einem ES6-Modul sind grundsätzlich *lokal* und daher weder für andere Module erreichbar noch für die Umgebung, in die das Modul importiert wurde, es sei denn, die Variablen wurden jeweils *explizit* exportiert.

### 11.3.3. Agnostische Module

---

Die ECMA6-Module selbst sind **statisch**, machen also keine Annahme darüber, auf welche Art und Weise sie faktisch geladen werden. Hierfür ist eine **Loader-API** zuständig, die separat, passend zur Umgebung, konfiguriert werden muss.

Zuständig für das Laden der Module ist die **Umgebung**, in der sie verwendet werden. ECMA6-Module sollen gleichermaßen im Browser wie auf dem Server (also asynchron und synchron) verwendbar sein.

- ✓ In diesem Sinne sind ECMA6-Module „agnostisch“, indem sie *keine Vorannahmen* über die zu erwartende Umgebung treffen.

### 11.3.4. Das Keyword „export“

In einem ES6-Modul wird das Keyword `export` verwendet, um einen Teil des Modulscoopes öffentlich erreichbar zu machen. Alles was nicht explizit exportiert wird, bleibt privat.

Das Keyword `export` darf dabei nur im **Toplevel** des Moduls erscheinen, also *nicht* in Blöcken von Kontrollstrukturen oder Funktionen eingesetzt werden.

Es ist legal, *mehrere* `export`-Statements in einem Modul einzusetzen.

✓ **Sehr wichtig** ist es, zu verstehen, dass stets eine **Bindung** (ein *Pointer*) exportiert wird, nicht etwa ein *Value*:

Sollte sich der Wert (auch ein primitiver) einer exportierten Variable innerhalb eines importierten Moduls **zur Laufzeit** ändern, so tritt diese **Änderung** in diesem Augenblick auch überall dort in Kraft, wo die gebundene, exportierte Variable verwendet wird.

#### Benannter („named“) Export

Man kann aus einem Modul mittels eines `export`-Statements beispielsweise eine Variable exportieren:

```
var variableName = true;
export { variableName };
```

✓ Achtung, diese Schreibweise ähnelt zwar dem „Destructring“, dennoch handelt es sich nicht um dasselbe Prinzip.

Eine andere (kürzere) Schreibweise hierfür ist:

```
export var variableName = true;
```

Hier wird praktisch die Deklaration der Variablen mit dem `export`-Keyword geprefixt. Dasselbe geht auch für Funktionen:

```
export function tolleModulMethode() {
    // ... tut tolle Sachen
};
```

- ✓ Der Vorteil ist, dass man einen Export *gleichzeitig* zur Deklaration festlegen kann. Die Objektschreibweise hätte wiederum den Vorteil, dass man die Exporte ans Ende des Modulcodes verlagern kann.

Möchte man mehrere Aspekte (Variablen) eines Moduls exportieren, so kann man (anstelle mehrerer einzelner `export`-Statements) auch eine sogenannte Liste exportieren.

```
var variableName1 = true;
var variableName2 = false;
var variableName5 = "egal";

export { variableName1, variableName2, variableName3 };
```

- ✓ Eine **Exportliste** ist kein Array, sondern entspricht eher einem Objekt in Shortcut-Schreibweise!

Es besteht kein Zwang, nach außen denselben Bezeichner zu verwenden, wie im Modul. In diesem Fall kann man ein **Alias** exportieren:

```
var bloederName = true;
export { bloederName as tollerName };
```

- ✓ Der Bezeichner `bloederName` steht anschließend beim Import nicht zur Verfügung, sondern nur sein Alias.

Dies ist auch im Rahmen einer Liste möglich:

```
var variableName1 = true;
var variableName2 = false;

export {
  variableName1 as myVariableName1,
  variableName2 as myVariableName2
};
```

## Unbenannter Default-Export

Es gibt, neben den benannten Exporten auch einen sogenannten „Default-Export“, bei dem eine Bindung nach außen gegeben wird, *ohne* sie zu benennen.

- ✓ Es darf allerdings pro Modul nur *einen einzigen* derartigen, umbenannten **Defaultexport** geben. Zusätzlich sind jedoch auch beliebig viele *benannte* Exporte möglich.

Okay, *eigentlich* ist ein Defaultexport der Export einer Bindung mit dem *Namen* default. Dies nur als Randbemerkung...

Es gibt zwei Möglichkeiten, einen Defaultexport zu definieren:

```
var variableName = true;  
// Vorsicht hier! Kein Update bei Änderungen!  
export default variableName;
```

Etwas länger zu schreiben, aber expliziter ist die Aliasschreibweise:

```
var variableName = true;  
// Bindung verhält sich wie gewohnt  
export {variableName as default};
```

**Achtung:** Diese, zweite Schreibweise ist zu bevorzugen, da das Bindungsverhalten sonst von dem der benannten Exporte abweicht!

Ein Defaultexport ist so auch im Rahmen einer Exportliste zusammen mit benannten Exporten möglich.

```
var variableName = true;  
var variableName1 = true;  
var variableName2 = false;  
  
export {  
    variableName as default,  
    variableName1 as myVariableName1,  
    variableName2  
};
```

- ✓ Wozu dienen Defaultexporte nun?

Im Prinzip muss zur Verwendung eines benannten Exports dieser beim Import explizit namentlich eingebunden werden. Dies ist bei Defaultexporten nicht nötig. Mehr dazu beim Keyword `import`.

### 11.3.5. Import

---

Wenig überraschend, wird der Import eines Moduls (bzw. der in ihm definierten Exportbindungen) mit dem `import`-Keyword vorgenommen.

Ein Import besteht immer aus zwei Teilen:

- Nennung der Variablen, die importiert werden, wobei bezeichnet wird, wie diese im importierenden Skript zu heißen haben
- Das zu importierende **Modul** bzw. der Pfad zur Moduldatei **als String**. Dieser Teil wird als *module specifier* bezeichnet und er *muss* ein String sein (d.h. er darf *keine Variable* sein, die einen String enthält; es geht um statische Auswertbarkeit!!).

Beim Export wurde gesagt, dass nicht *Values*, sondern *Bindungen* (Pointer) exportiert werden. Quasi werden durch den Import die im Modul definierten Variablen im Scope des importierenden Scriptes verfügbar gemacht.

✓ **Achtung:** Diese Bindung ist *keine Zweiwegebindung*! Eine importierte Bindung ist **read-only**. Ein Versuch, den Wert einer importierten Variable zu *ändern*, scheitert mit einem *Fehler*!

Beispielsweise kann ein Import wie folgt lauten:

```
import x from "pfad/zum/modul";  
// x verwenden
```

In der Tat wird hier *nicht* ein benannter Export angesprochen, sondern der **Defaultexport**. Das `x` ist ein *frei wählbarer* Bezeichner.

Sehr ähnlich sieht folgendes Statement aus:

```
import { x } from "pfad/zum/modul";  
// x verwenden
```

Hier wird ein **benannter Export** namens `x` geladen. Der Bezeichner `x` ist damit *nicht frei wählbar*. Vielmehr wird die als `x` für den Export gebundene Modulvariable im Scope des Importskriptes zur Verfügung gestellt.

Auch ein **Alias** ist beim Import möglich:

```
import { x as y } from "pfad/zum/modul";  
// y verwenden
```

Die Exportbindung namens `x` ist im Importsript unter dem Namen `y` verfügbar. Der Name `x` ist im Importsript hingegen nicht zugänglich (obwohl es sich um den Exportnamen handelt).

An dieser Stelle darf offenbart werden, dass die erste, sehr kurze Schreibweise für einen Import des Defaultexports lediglich eine Abkürzung war für:

```
// offizieller Name des Defaultexports ist 'default':  
import { default as x } from "pfad/zum/modul";  
// x verwenden
```

Importiert man mehrere benannte Exporte, so geschieht dies im Rahmen einer **Importliste**:

```
import { x1, x2 } from "pfad/zum/modul";  
// x1 und x2 verwenden
```

✓ Dies ist, obwohl die Syntax *sehr* ähnlich aussieht, *kein* Destructuring!

Auch im Rahmen einer Importliste können Aliase ausgesprochen werden:

```
import {x1, x2 as x3} from "pfad/zum/modul";  
// x1 und x3 verwenden (x2 existiert hier nicht)
```

Soll gleichzeitig der Defaultexport genutzt werden, so muss dieser außerhalb der Importliste erscheinen:

```
// x (frei zu bezeichnen) ist der Defaultexport  
import x, {x1, x2} from "pfad/zum/modul";  
// x (Default), x1 und x2 verwenden
```

Möchte man ohnehin die gesamte API eines Moduls importieren und nicht, was mit einer explizit zusammengestellten Importliste möglich wäre, nur einen Teil dieser API, so greift man zu einem Pattern, das als **„Namespace-Import“** bekannt ist:

```
// Mal angenommen, es werden zwei Funktionen meth1 und  
// meth2 als benannte Exporte exportiert...
```

```
import * as myModule from "pfad/zum/modul";  
// dann stehen sie jetzt wie folgt zur Verfügung:  
myModule.meth1(); // meth1 des Moduls verwenden  
myModule.meth2(); // meth2 des Moduls verwenden
```



### 11.3.6. Haupt- und Submodule mit ECMA6-Modulen

---

Um ein konkretes Beispiel für den Einsatz von ECMA6-Modulen vorzustellen, soll nun ein **Hauptmodul** mit trigonometrischen und logarithmischen Funktionen versorgt werden, die in **Submodule** ausgelagert sind.

Physisch benötigen wir eine Basisdatei *mathES6.js* und ein Verzeichnis *math\_modulesES6*, in der zwei Dateien *logarithmES6.js* and *trigonometryES6.js* angelegt werden:

```
math_modulesES6/  
  logarithmES6.js  
  trigonometryES6.js  
mathES6.js
```

Die Datei *mathES6.js* fungiert als Root-Modul, die beiden anderen Dateien im Ordner bezeichnen wir als **Submodule**. Betrachten wir diese zuerst:

*logarithmES6.js*:

```
var LN2 = Math.LN2;  
var LN10 = Math.LN10;  
  
function getLN2(){  
  return LN2;  
}  
  
function getLN10(){  
  return LN10;  
}  
  
export { getLN2, getLN10 };
```

Die beiden Funktionen des Submoduls werden als Liste exportiert.

*trigonometryES6.js*:

```
var cos = Math.cos;  
var sin = Math.sin;  
  
function getSin(value){  
  return sin(value);  
}  
function getCos(value){  
  return cos(value);  
}
```

```
}  
export {getCos, getSin};
```

Das Modul ist analog zu dem trigonometrischen Modul aufgebaut.  
Betrachten wir nun das Rootmodul:

*mathES6.js*

```
import * as logarithm from "math_modulesES6/logarithmES6";  
import * as trigonometry from  
"math_modulesES6/trigonometryES6";  
  
export { logarithm, trigonometry }
```

Das Hauptmodul enthält selbst also keine Funktionen, sondern dient nur dem Import von Submodulen. Eine Anwendung braucht daher nur das Hauptmodul zu importieren und kann hierdurch auch die Funktionen der Submodule nutzen, ohne von deren Existenz zu wissen.

In dieser Weise *könnte* das Rootmodul in einer Anwendung genutzt werden, die ECMA6-Module importiert:

*indexES6.js*

```
import math from "math";  
console.log(math.trigonometry.getSin(3));  
console.log(math.logarithm.getLN2(3));
```

Hier gibt es in der Praxis ein kleines Problem, da wir noch keine Umgebung besitzen, die das Laden dieser Module direkt unterstützt. Es muss daher ein spezieller **Loader** für ECMA6-Module oder ein *Transpiler* eingesetzt werden, der die ES6-Syntax in ECMA5 und dort in Common JS oder AMD-Module übersetzt (die dann mit RequireJS geladen würden).

---

### 11.3.7. Laden von ECMA6-Modulen mit SystemJS

---

Das *unmittelbare* Laden und Verwenden von ECMA6-Modulen gestaltet sich derzeit noch etwas schwierig, weshalb wir auf den Einsatz von diversen Tools angewiesen sind. Jetzt benötigen wir eine Installation von **NodeJS**, um den Node Package Manager **NPM** benutzen zu können.

- ✓ Als Loader soll **SystemsJS** fungieren. SystemJS stellt ein globales Objekt **system** zur Verfügung, das den Loader implementiert. Es verfügt hierzu über eine Methode `import()`.

Leider ist der Weg dorthin sehr indirekt, weshalb wir zunächst eine *globale Installation* des **JSPM** (JavaScript Package Manager, derzeit noch Beta) benötigen:

```
npm install -g jspm@beta
```

Im Projekt, in dem wir ECMA6-Module einsetzen wollen, installieren wir das Tool lokal (eine `package.json` wurde im Vorfeld erzeugt). Dies stellt und letztlich auch den Loader zur Verfügung:

```
npm install --save-dev jspm@beta
```

Um das Tool für das aktuelle Projekt zu konfigurieren, benötigen wir eine Datei `jspm.config.js`, die wir über die Initialisierung erhalten:

```
jspm init
```

- ✓ Die Abfragen ähneln denen, die wir beim Erstellen eines `package.json` durch `npm init` durcharbeiten.

Im Grunde kann man alle Defaultwerte übernehmen, bis auf den, der den Transpiler festlegt. Anstelle von Traceur, der den Defaulttranspiler in SystemJS darstellt, soll der **TypeScript-Compiler** eingesetzt werden.

- ✓ Dies bedeutet hier keinen besonderen Unterschied, da keine statische, sondern eine dynamische Übersetzung zur Laufzeit vorgenommen wird.

Die betreffende Abfrage beantworten Sie mit dem String "TypeScript":

```
SystemJS.config transpiler (Babel, Traceur, TypeScript,  
None) [babel]: TypeScript
```

Die Dateien liegen an der vorgegebenen Position:

```
math_modulesES6/  
  logarithmES6.js  
  trigonometryES6.js  
mathES6.js
```

Wir fügen nun eine HTML-Seite hinzu, die das Modul *mathES6.js* verwenden soll: *systemJStest.html*

```
math_modulesES6/  
    logarithmES6.js  
    trigonometryES6.js  
mathES6.js  
systemJStest.html
```

Die Datei lädt *system.js*, also den Loader, sowie dessen Konfiguration *jspm.config.js*:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>Title</title>  
    <script src="../jspm_packages/system.js"></script>  
    <script src="../jspm.config.js"></script>  
    <script>  
        // Hier wird das Hauptmodul geladen und  
eingesetzt  
    </script>  
</head>  
<body>  
  
</body>  
</html>
```

Beim Testlauf ergibt sich, dass Teile des Tools nicht geladen werden. Dies kommt daher, dass die Anwendung davon ausgeht, direkt auf einem Rootverzeichnis zu arbeiten. Wenn dies (wie häufig) nicht der Fall ist, so muss in *jspm.config.js* der Eintrag für den `baseURL` geändert werden.

```
browserConfig: {  
    "baseURL": ""  
},
```

Im aktuellen Fall befindet sich der Modultest in einem Unterverzeichnis (dessen Name egal ist) des Verzeichnisses, in dem das Tool installiert ist. Der `baseURL`-Eintrag muss wie folgt geändert werden:

```
browserConfig: {  
    "baseURL": "../.."   
},
```

Das anschließende Laden eines Moduls mit dem **Loader** ist im Grunde einfach. Es wird lediglich der Pfad an `System.import()` übergeben:

```
System.import('./mathES6')
```

Beachten Sie die Pfadangabe, die im lokalen Verzeichnis beginnt. Die **fehlende Dateiendung** `.js` wird jedoch mit einem Ladefehler quittiert. Man könnte sie nun beim Import zwar ergänzen, müsste dies jedoch auch beim Laden der Submodule im Hauptmodul berücksichtigen.

Besser ist es, dem Laden den Befehl vorausgehen zu lassen, die Endung selbstständig zu ergänzen:

```
System.defaultJSExtensions = true;
```

- ✓ Dieses Flag besteht aus Gründen der Abwärtskompatibilität. Es wird den Nutzern dieses Tools empfohlen zum expliziten Einsatz von Dateiendungen überzugehen.

Das Laden des Moduls erfolgt asynchron. Die Methode `import()` gibt daher ein **Promise** zurück, das eine `then`-Methode besitzt:

```
System.import('./mathES6.js').then(function(){  
    console.log("Math-Modul geladen!");  
});
```

Die Callbackfunktion, die wir hier übergeben, feuert, sobald das Module geladen ist. Um das Modul zur Verwendung bereitzustellen, wird es als Objekt an die Funktion übergeben. Wir fangen es in einem Parameter `math` auf:

```
System.import('./mathES6.js').then(function(math){  
    console.log("Math-Modul geladen!", math);  
});
```

- ✓ Die Ähnlichkeiten mit dem RequireJS-Ansatz sind nicht rein zufällig.

Die beiden Submodule werden im Hintergrund geladen. Ihre Funktionen stehen, wie es sein soll, im `math`-Objekt zur Verfügung:

```
System.defaultJSExtensions = true;  
System.import('./mathES6.js').then(function(math){  
    console.log("Math-Modul geladen!", math);  
});
```

```
    console.log("Sinus von 3:",  
math.trigonometry.getSin(3));  
    console.log("Nat Log von 2:", math.logarithm.getLN2(3));  
});
```

Q.E.D. Besonders performant ist dies jedoch nicht und sollte noch nicht für Produktionszwecke eingesetzt werden. Es müssen durch den Browser neben den Modulen eine Vielzahl von Tools geladen werden, außerdem wird ihm eine *Just-In-Time*-Kompilation der ECMA6-Sourcen übergebürdet.

Der Vorteil ist, dass man den Code im originalen ECMA6 belassen kann. In der Zukunft mögen Loader im Browser implementiert sein, was das Laden beschleunigt. Zudem sollte der Code dann auch direkt lauffähig sein.

## 12. Einführung in TypeScript

An dieser Stelle soll ein Einstieg in die wichtigsten Features erfolgen, die in TypeScript über die ECMA-Syntax (ECMA6 und Teile von ECMA7) hinaus zur Verfügung stehen, und die unter anderem **statisches Typechecking** während der Entwicklungsphase ermöglichen, nämlich die **Typisierung**.

- ✓ Typisierung ist ein altes, auf **Bertrand Russels „Type Theory“** (1910) zurückzuführendes Prinzip, nach dem die Verarbeitung von Werten vereinfacht und sicherer gemacht wird, indem jedes Symbol eine **Typannotation** im Sinne eines „Kontraktes“ erhält.

Derartige Typen beziehen sich auf *Variablen, Ausdrücke, Objekte, Funktionen, Klassen und Module* und bilden in ihrer Gesamtheit ein **Typsystem**. Das Ziel besteht in der Vermeidung von Fehlern während der Verarbeitung. Man unterscheidet:

- **Statisches Typechecking**  
während der Code-Erstellung, also zum Zeitpunkt des Kompilierens der Anwendung, daher „Compile-Time“ oder „Design-Time“.
- **Dynamisches Typeshecking**  
während der Verarbeitung, also zur Laufzeit, daher „Run-Time“.

**JavaScript** ist eine nicht-typisierte Sprache („loose typed“). Die Typen werden während der Laufzeit dynamisch aufgrund der Zuweisung eines Wertes vergeben. Typinkonsistenzen werden durch Konversion behoben. Ist dies nicht möglich, so wird ein Typfehler gemeldet („Runtime-Error“).

Da JavaScript die *Zielsprache* jeder TypeScript-Applikation ist, spielen Überlegungen zum dynamischen Typechecking hier **keine Rolle**. Interessant ist für TypeScript daher lediglich *statisches* Typechecking („static typing“), das in Folge behandelt wird.

Hierfür ermöglicht es der in eine TypeScript-Entwicklungsumgebung eingebaute **Compiler**, Typkollisionen bereits im Editor aufzuspüren und nicht, wie ansonsten, aufgrund der (sprachbedingten) *dynamischen* Arbeitsweise von JavaScript, erst zur Laufzeit einer Anwendung.

Der Compiler besitzt also stets eine Doppelfunktion. Zum einen arbeitet er während der **Compile-Time** (auch „Design-Time“), d.h. während des

Erstellens des Codes in der IDE, als Type-Checker, der den Entwickler auf bemerkte Fehler wie Typkollisionen aufmerksam macht. Zum anderen ist er für die Übersetzung des Codes in ein JavaScript-Zielformat zuständig.

- ✓ Diese zweite Phase kann jedoch nur erfolgen, wenn alle während der Compile-Time aufgefundenen Fehler beseitigt sind. Der Compiler verweigert ansonsten eine Übersetzung.

## 12.1. Typisierung in TypeScript

---

Ähnlich wie C# arbeitet TypeScript grundsätzlich mit **Typinferenz**, trifft also „Annahmen“ über den Typ, den eine Variable oder eine Objekteigenschaft besitzt. Es handelt sich also um ein **strukturelles Typsystem** (im Gegensatz zu *nominellen* Typsystemen, wie beispielsweise bei C).

- ✓ Aufgrund der Typinferenz ist eine explizite Typannotation in TypeScript sehr häufig **nicht erforderlich**.

Nur in bestimmten Fällen, beispielsweise wenn der Typ eines Ausdrucks oder der gewünschte Typ einer Variablen nicht einer **Wertzuweisung** („*direct typing*“) oder **aus dem Kontext** erschlossen werden kann („*contextual typing*“), ist eine **explizite Typannotation** sinnvoll.

### Nochmals:

Variablen sind in TypeScript **immer** typisiert, entweder durch *Typinferenz* (über *Zuweisung* oder *Kontext*) oder durch eine konkrete *Typzuweisung*!

## 12.2. TypeScript-Typen

---

Alle TypeScript-Typen (mit Ausnahme von `any`) fallen in eine der folgenden Kategorien:

### Primitive Types:

Hierunter versteht man die Typen `void`, `undefined`, `null`, `boolean`, `number`, und `string` sowie den Typ `enum`.



### Union Types:

Zusammengesetzte Typen, die für eine Variable mehrere, alternativ erlaubte Typen nennen (Alternativen durch Pipe-Symbol getrennt)

### Object Types:

Dies umfasst Array-, Function-, Constructor-, Class- und Interface-Typen.

### Type Parameter:

In diese Gruppe fallen die sogenannten „Generics“.

## 12.3. Typinferenz mit „direct“ und „contextual“ Typing

---

Die explizite Nennung eines Typs ist für eine **Variable** in Typescript *nicht erforderlich*, sofern die Variable bei der Deklaration **initialisiert** wird, oder sich ein Typ aus dem Kontext erschließen lässt.

### ✓ Direct Typing:

Typescript „nimmt an“, dass es sich um eine Variable vom Typ des zugewiesenen Wertes handelt.

```
var implicitNumber = 42; // implizit Typ "number"
implicitNumber = '42'; // Fehler!

var implicitBoolean = true; // implizit Typ "boolean"
implicitBoolean = '42'; // Fehler!

var implicitString = "Hallo"; // implizit Typ "string"
implicitString = 42; // Fehler!
```

### 12.3.1. Der implizite Typ “any”

---

Wird eine Variable nur deklariert, *ohne dass* gleichzeitig ein Wert zugewiesen wird, über den TypeScript eine Typinferenz vornehmen könnte, so weist TypeScript der Variable implizit den Typ `any` zu:

```
var implicitAny; // implizit any !!

implicitAny = {}; // geht
implicitAny = 'toast '; // geht
implicitAny += 42; // geht
```

```
console.log(implicitAny); // "toast 42"
```

- ✓ In diesem Falle erhält die Variable „nebenbei“ den Typ `any`. Es ist sicherlich besser, dies im Zweifelsfall bewusst vorzunehmen.

Eine Variable über einer Zuweisung mit den Werten `null` oder `undefined` zu initialisieren, führt ebenfalls zum impliziten Typ „`any`“.

```
var implicitAnyNull = null;

var implicitAnyUndefined = undefined;
```

### 12.3.2. Typing über Returnwert einer Funktion

---

Mit Funktionen und den dort auftretenden bzw. durch den Compiler erschlossenen Typen werden wir und nachfolgend noch genauer befassen. Hier soll es weiterhin um den Typ einer Variablen gehen, die einen Wert erhält.

Erfolgt die Wertzuweisung nicht wie in den vorangegangenen Beispielen direkt, sondern über den **Rückgabewert einer Funktion**, so ändert sich das Verhalten des Compilers gegenüber dem an die Variable zugewiesenen Typ (es ist ja ein „contextual type“).

Die Variable erhält auf verschiedene Weise keinen Wert (oder einen „falsy“ Wert) zugewiesen. Wir verwenden für die Return-Zuweisungen eine IIFE:

```
// Typ ist „any“:
var implicitAny;

// Typ ist „void“:
var implicitVoid = (function(){
    // kein return-Statement!
})();

// Typ ist „void“:
var auchImplicitVoid = (function(){
    return; // leeres return-Statement
})();
```

Geschieht beim Deklarieren einer Variable *gar nichts*, so erhält diese den **„any“-Typ**. Eine Funktion ohne oder mit leerem Return (die also „nichts“ zurückgibt) generiert hingegen einen **“void“-Typ**.

```
// Typ ist „any“:  
var implicitAnyUndefined = undefined;  
  
// Typ ist „undefined“:  
var implicitUndefined = (function(){  
    return undefined; // explizites return von undefined  
})();
```

Ein direkt zugewiesenes `undefined` generiert einen **“any“-Typ**, wohingegen ein über `return` *explizit* zurückgegebenes `undefined` einen **“undefined“-Typ** erzeugt (nicht etwa einen “void“-Typ!).

```
// Typ ist „any“:  
var implicitAnyNull = null;  
  
// Typ ist „null“:  
var implicitNull = (function(){  
    return null;  
})();
```

Die direkte Zuweisung von `null` generiert den **„any“-Typ**. geschieht Ähnliches über `return`, so erhält man einen **„null“-Type**.

- ✓ Wie man sieht, nimmt das „contextual typing“ die Umstände sehr **wörtlich**, während das „direct typing“ die Weiterverwendbarkeit der Variablen durch erneute Zuweisung in den Vordergrund rückt.

Eine Variable, die den Typ “void”, “null” oder “undefined” besitzt ist nicht sonderlich nützlich, da ihr kein praktisch verwendbarer Wert zugewiesen werden kann. Es handelt sich dennoch um real existierende Typen, die den „primitive“ Types (siehe „TypeScript-Typen“) zugerechnet werden. Zum Glück sind sie im Rahmen einer direkten Zuweisung nicht darstellbar.

### 12.3.3. Melden des impliziten Typs „any“

Der implizite Typ „any“ ist meist unerwünscht, wird aber vom TypeScript-Compiler nicht als Fehler gewertet.

- ✓ Möchte man eine Vergabe des impliziten „any“-Typs sichtbar machen, so kann dies in der *tsconfig.json* geregelt werden.

```
{
  "compilerOptions": {
    ...
    "noImplicitAny": true
  }
}
```

Sobald dies geschehen ist, wird beim Compile-Vorgang ein **Fehler** gemeldet:

```
var implicitAny;
// Error: TS7005:Variable 'implicitAny'
  implicitly has an 'any' type.
```

- ✓ Hierbei wird nicht die *Zuweisung* des Typs „any“ vermieden, sondern die *Weiterverarbeitung* der Datei!

**Anmerkung:** Das Setzen von `"noImplicitAny": true` in *tsconfig.json* vermeidet auch, dass versehentlich Funktionsparameter untypisiert bleiben.

**Tipp:** Setzen Sie (vorsichtshalber) zusätzlich auch noch folgendes Flag:

```
{
  "compilerOptions": {
    ...
    "noImplicitAny": true
    "suppressImplicitAnyIndexErrors": true
  }
}
```

- ✓ Dieser Hinweis findet sich in der Angular2-Dokumentation.

## 12.4. Statische Typisierung mit Typannotation

In TypeScript lässt sich einer Variable, einem Parameter oder dem Rückgabewert einer Funktion **explizit** ein Typ zuordnen. Dies geschieht (abstrakt) über das Annotieren des Typs nach dem Variablenbezeichner:

```
var meineVariable: meinTyp;
```

Beim Kompilieren nach ECMAScript werden die Annotationen entfernt („type erasure“)!

Hier wird eine *numerische* Variable deklariert *und* initialisiert:

```
var test: number = 42;
```

Versucht man, dieser Variable einen **String** anstelle einer Zahl zuzuweisen, so meldet die IDE einen Fehler.

```
var test: number = 42;
test = '42'; // Fehler!
```

```
error TS2322:
Type 'string' is not assignable to type 'number'.
```

Dies ist eine Stringvariable:

```
var myString: string = "hello";
```

Und dies eine vom Typ *boolean*:

```
var mybool: bool = true;
```

Die Zuweisung eines falschen Typs verursacht einen Fehler:

```
var auchBool: bool = 123; // Fehler: "wrong type"
```

Beim *statischen Typechecking* erfolgt keine implizite Typkonvertierung,

```
var mybool: bool = true;
mybool = "false"; // falscher Typ in der Neuzuweisung
```

### 12.4.1. Verhalten von null und undefined

---

Ungeachtet des annotierten (oder erschlossenen) Typs ist bei einer Variablen **jederzeit** die Zuweisung der Werte `null` und `undefined` möglich:

```
var mybool: bool = true;
```

```
mybool = null;           // geht
mybool = undefined;      // geht ebenfalls
```

- ✓ In der Regel ist dieses Verhalten praktisch, da es eine Vorbelegung von Variablen oder Properties mit `null` oder ein Überschreiben mit `undefined` erlaubt. (Eine Vorbelegung mit `null` muss jedoch mit einer Typannotation einhergehen, da sonst „any“ als Typ vergeben wird.)

```
var sollBoolWerden: bool = null;
```

- ✓ Möchte man jedoch für die Zuweisung von `null` oder `undefined` eine Fehlermeldung, so kann man dies in der `tsconfig.json` regeln

```
{
  "compilerOptions": {
    ...
    "strictNullChecks": true
  }
}
```

In Folge meldet der Editor die Zuweisung von `null` oder `undefined` als unzulässig:

```
var myBool: boolean;

[ts] Type 'undefined' is not assignable to type 'boolean'.
var myBool: boolean
myBool = undefined;
```

Wie an den Fehlermeldungen erkennbar, handelt es sich bei `null` und `undefined` um echte eigene Typen.

Nützlich ist der `strictNullCheck`, wenn beispielsweise eine Funktion einen Eingabewert erhalten soll, bei dem es sich um einen String handeln muss.

```
function stringLaenge(str: string) {
  console.log(str.length); // Fehler bei null!
}
```

Unter normalen Umständen würde der Compiler nicht monieren, wenn statt eines Eingabestrings `null` (oder gar nichts) übergeben wird, was zur Run-Time einen Fehler verursachen würde.

Möchte man bei `strictNullCheck` *dennoch* erlauben, dass eine Variable neben einem festgelegten Typ auch den Wert `null` aufnehmen kann, so muss dies über einen **Union-Type** definiert werden:

```
// "strictNullChecks": true
var stringOderNull: string | null; // akzeptiert null

var stringAberNotNull: string;    // akzeptiert null nicht!
```

### 12.4.2. Der explizite Typ „any“

---

Will man für eine Variable *absichtlich* zulassen, dass nach der Deklaration Werte eines beliebigen Typs zugewiesen werden können, so kann sie auch *explizit* mit dem **Typ any** deklariert werden:

```
var explicitAny: any;
explicitAny = {};
explicitAny = 'toast ';
explicitAny += 42;
console.log(explicitAny); // "toast 42"
```

- ✓ Allzu großzügiger Einsatz des Typs `any` kommt einem Rückfall auf das “dynamic Typing”, also das Grundverhalten von JavaScript gleich. Die Vorteile von Typescript zur Design-Time würden dadurch verschenkt.

### 12.5. Der enum-Typ

---

Gemäß der Spezifikation handelt es sich bei einem **Enum** (Aufzählung) um einen Untertyp von `Number`. Er besteht aus einem Set von Elementen, deren Stringbezeichner jeweils einer Zahl zugeordnet sind. Das Konzept der TypeScript-Enums gleicht dem in Java, C++ und C#.

- ✓ Der **enum-Typ** zählt zu den primitiven Typen.

In Typescript wird der *enum*-Typ als **Liste aus Elementen**, ähnlich einem Array, definiert, wobei jedoch die *geschweiften* Klammern als Begrenzer eingesetzt werden:

```
// dies ist Typescript:
enum ZUSTAND {
    CONNECTING,
    CONNECTED,
    DISCONNECTING,
    WAITING,
    DISCONNECTED
};
```

Beim Kompilieren werden im JavaScript die *Zahlenwerte* zugewiesen, mit denen im Hintergrund verglichen werden soll. Sofern nicht explizit anders angeordnet, werden den Elementen in Anordnungsreihenfolge die Zahlen 0, 1, 2 ... zugeordnet:

```
// dies ist das generierte Javascript:
var ZUSTAND;
(function (ZUSTAND) {
    ZUSTAND[ZUSTAND["CONNECTING"]] = 0;
    ZUSTAND[ZUSTAND["CONNECTED"]] = 1;
    ZUSTAND[ZUSTAND["DISCONNECTING"]] = 2;
    ZUSTAND[ZUSTAND["WAITING"]] = 3;
    ZUSTAND[ZUSTAND["DISCONNECTED"]] = 4;
})(ZUSTAND || (ZUSTAND = {}));
```

Der *enum*-Typ kann nun wie folgt eingesetzt werden:

```
if (this.zustand === ZUSTAND.CONNECTING) {
    console.log('System wird verbunden');
}
```

## 12.6. Union-Types

Mittels sogenannter **Union-Types** ist die Angabe mehrerer, *alternativ erlaubter* Typen zulässig. Dies geschieht durch, mittels Pipe-Symbolen getrennte, Liste bei der Angabe der Annotation:

```
var myStringBool: string | boolean;
myStringBool = "Hallo";
myStringBool = false;
```

- ✓ Interessant ist die Möglichkeit, nicht nur primitive Typen als Alternativen angeben zu müssen, sondern beliebig mischen zu dürfen.



Hier als Beispiel eine Variable, die Pfadinformationen in Form eines Strings oder in Form eines Arrays, bestehend aus Einzelstrings, enthalten darf:

```
var path: string[] | string;  
path = '/temp/log.xml';  
path = ['/temp/log.xml', '/temp/errors.xml'];  
path = 1; // Error
```

✓ Auch drei, oder mehr alternative Typangaben sind möglich.

Auch ein Union-Type kann **kontextuell**, über Function-Return erzeugt werden. In folgendem Beispiel erhalten wir den Union-Type „string|null“:

```
// erhält Typ string | null:  
var implizitUnion = (function(sinnDesLebens){  
    if(sinnDesLebens !== 42){  
        return null;  
    } else {  
        return "Danke für den Fisch!";  
    }  
})(42);
```

Aus dem konkreten Programmfluss geht zwar hervor, dass ein **String** zugewiesen wird. Dennoch geht der Compiler davon aus, dass `null` als Returnwert erfolgen *könnte* und generiert einen Union-Type.

## 12.7. Arrays und typisierte Arrays

Eine Variable erhält durch Zuweisung eines Arrays einen **Array-Typ**. Der Compiler schließt aus den Typen der Items des zugewiesenen Arrays auf den Typ, den das Array besitzt. Für ein leeres Array ist dies der Typ „any[]“:

```
// erhält den Typ any[]:  
var einArray = [];
```

Ist das Array *nicht* leer und haben die Items *denselben* Typ, so wird dieser als Grundlage des Array-Typs verwendet:

```
// erhält den Typ number[]:  
var zahlen = [1,2,3];
```

Das Array kann anschließend in diesem Sinne verwendet werden:

```
zahlen.push(5); // geht
zahlen.push("Hopp"); // Fehler!
```

Ist das zugewiesene Array *heterogen* belegt, so erhält man automatisch ein auf einen **Union-Type** typisiertes Array:

```
// erhält den Typ (number | boolean)[]:
var boolZahl = [true, 42];
```

- ✓ Typinferenz genügt, solange bei der Initialisierung ein Array vorliegt, das alle faktisch gewünschten Typen enthält.

### 12.7.1. Annotation von Array-Typen

Arrays können natürlich auch explizit **annotiert** werden. Als Typen können alle gängigen TypeScript-Typen, sowie selbst definierte Custom-Typen verwendet werden.

Ein numerisches Array wird folgendermaßen annotiert:

```
var zahlenArray: number[] = [];
zahlenArray.push(2);
zahlenArray.push(3);
```

Eine andere, *gleichwertige* Schreibweise, um ein numerisch typisiertes Array zu beschreiben, sieht wie folgt aus:

```
var meineZahlen: Array<number> = [1, 2, 3];
```

Analog können String-Arrays oder Boolean-Arrays definiert werden. Möchte man Items unterschiedlicher Typen speichern können, so kann ein Array aus Union-Typen definiert werden:

```
var zahlenOderBools: (number|boolean)[] = [];
zahlenOderBools.push(3);
zahlenOderBools.push(true);
```

Auch dies kann in der *Angle-Brackets-Notation* geschrieben werden:

```
var auchBoolZahl: Array<number|boolean> = [true, 42];
```

Für die Fälle, wo ein heterogenes Array gewünscht ist, kann man den Typ `any[]` einsetzen, wonach sich das betreffende Array wie ein "gewöhnliches" JavaScript-Array verhält:

```
var egalwas: any[] = [];  
egalwas.push(1);  
egalwas.push("Hopp");  
egalwas.push([]);  
egalwas.push({});
```

## 12.8. Typisierung von Funktionen

---

Die Möglichkeit der Annotation besteht in TypeScript auch für **Funktionen**. Hierbei können sowohl die Parameter, als auch der Rückgabewert annotiert werden. Solange man nicht annotiert, tritt, wie gehabt, die übliche Typinferenz auf den Plan

Die folgende Funktion ist nicht annotiert. Als Argumente lässt sie daher (per Typinferenz) alle Typen zu („any“). Auch ihr Rückgabewert besitzt aus diesem Grund implizit den **Typ „any“**:

```
function addiereReturnAny(a, b) {  
    return a + b;  
}
```

Da TypeScript sich Typen auch aus dem Kontext erschließt („contextual typing“) besitzt eine neue Variable, welche mit dem Rückgabewert der Funktion initialisiert wird, folglich *ebenfalls* den **Typ „any“**:

```
var erg = addiereReturnAny(5,5);
```

Obwohl hier faktisch eine Zahl zugewiesen wird, erschließt der Compiler den Typ von `erg` allein über den Rückgabebetyp der Funktion. (Der Typ wird aus der Funktionsdeklaration erschlossen, nicht aus dem Ergebnis des Ausdrucks zur Runtime.)

Ein „any“-Typ ist nicht eingeschränkt:

```
erg = addiereReturnAny("zehn", "zwanzig");
```

... eine erneute Belegung mit einem String verursacht daher keinen Fehler.

### 12.8.1. Typisierung von Funktionsparametern

---

Sinnvoll ist *mindestens* eine Annotation der **Parameter**:

```
function addiereReturnNumber(a:number, b:number) {  
    return a + b;  
}
```

Da nun die Typen der Parameter *bekannt* sind, erschließt der Compiler, dass die Funktion für jeden erlaubten Einsatz den Typ "number" zurückgibt.

```
// erg2 erhält Typ "number":  
var erg2 = addReturnNumber(5,5);
```

Argumente mit falschem Typ werden moniert:

```
erg2 = addiereReturnNumber("zehn","zwanzig");  
// Argument of type 'string' is not assignable  
   to parameter of type 'number'
```

### 12.8.2. Typisierung des Returnwertes einer Funktion

---

Eine eigene Annotation des **Rückgabewertes** ist möglich, kann jedoch unterbleiben, sofern *contextual typing* erfolgen kann (ist nicht immer so!):

```
// Return-Typ kann aus Param-Typen erschlossen werden:  
function addRetNumExpl (a:number, b:number) :number {  
    return a + b;  
}  
  
// erg3 erhält (selbstverständlich) den Typ „number“:  
var erg3 = addRetNumExpl(5,7)
```

### 12.8.3. Funktionen ohne Returnwert

---

Für Funktionen ohne Rückgabewert gilt automatisch der Return-Typ "void".

```
// Kein Return: void  
function sagWas() {  
    console.log("Was soll ich sagen?");  
}
```

Dieser Typ gilt, wie bereits gezeigt, dann auch für eine Variable, welcher der Rückgabewert (vielleicht versehentlich) zugewiesen wird.

```
// Variable erhält Typ "void":  
var gesagt = sagWas();
```

Dies ist zwar kein Fehler, macht die Variable jedoch im Wesentlichen unbrauchbar, da ihr danach legal nur noch die Werte `undefined` und `null` zugewiesen werden können.

```
gesagt = null;           // geht meistens (s.u.)
gesagt = undefined;     // geht
gesagt = 42;            // Fehler
// Type 'number' is not assignable to type 'void'
```

✓ **Anmerkung:** Ist in `tsconfig.json` die Option `strictNullChecks` auf `true` gesetzt, ist im obigen Fall die Zuweisung von `null` nicht möglich!

Achtung, die Initialisierung einer Variable mit `undefined` gibt dieser den impliziten Typ `"any"`, ist also *nicht dasselbe*! Das gleiche gilt für die Initialisierung mit `null`.

```
// beide erhalten Typ "any":
var myUndefined = undefined;
var myNull = null;
```

✓ Grundsätzlich empfiehlt es sich, den Rückgabetyt einer Void-Funktion explizit zu setzen, wobei hier nur `"void"` oder `"any"` Sinn machen.

```
// Explizites void
function sagWasVoid() :void {
    console.log("Kombiniere, ich bin Void!");
}
```

## 12.8.4. Arrow-Funktionen

---

Die **Arrow-Notation** von Funktionen kann in TypeScript ebenso verwendet und bei Bedarf auch annotiert werden:

```
// nicht annotiert:
var add = (a,b) => a + b;

// erhält den Typ „any“:
var erg = add(17, 4);
```

✓ Da in der Arrow-Funktion weder Parameter noch Return annotiert wurden, hat `erg` den Typ `"any"`.

Die Arrow-Funktion wird genauso annotiert, wie eine gewöhnliche Funktion:

```
var add2 = (a:number, b:number) :number => a + b;
```

### 12.8.5. Optionale Parameter

---

JavaScript verlangt nicht zwingend die Übergabe eines Arguments für jeden definierten Parameter: Wird kein Argument übergeben, so wird der entsprechende Parameter einfach auf den Wert `undefined` gesetzt.

```
function add(a, b, c) {  
    if(typeof c !== "undefined") {  
        return a + b + c;  
    } else { // kein c übergeben  
        return a + b;  
    }  
}
```

```
var erg = add(17, 4); // -> 21 (TSC nörgelt!!)
```

In der TypeScript Design-Time ist die **Nichtübergabe eines Parameters** jedoch ein Fehler. Der Compiler moniert im vorausgehenden Fall, dass die Funktion „nicht ihrer Signatur entsprechend“ eingesetzt wird und verlangt einen *dritten* Parameter. Dies ist natürlich „lösbar“:

```
var erg = add(17, 4, undefined); // 21
```

Der dritte Parameter wäre damit aber *nicht wirklich* optional. Die Kenntlichmachung eines **optionalen Parameters** für den TS-Compiler erfolgt, indem man der Parametervariable ein **Fragezeichen** anfügt:

```
function addOpt(a, b, c?) {  
    if(c) {  
        return a + b + c;  
    } else {  
        return a + b;  
    }  
}
```

```
var erg = add(17, 4); // kein Problem mehr!
```

Dies lässt sich auch mit Annotation kombinieren:

```
function addOptNum(a:number, b:number, c?:number) {  
    if(c) {
```

```
        return a + b + c;
    } else {
        return a + b;
    }
}
```

- ✓ Wichtig ist es, dass als **optional** annotierte Parameter nicht *zwischen* obligatorischen Parametern stehen dürfen (wäre unsinnig!). Sie müssen sich stets am Ende der Parameterliste befinden.

### 12.8.6. Defaultparameter

---

Wie in ECMA6 üblich, können für Parameter auch **Defaultwerte** gesetzt werden. Sie können dann aber nicht gleichzeitig als optional gekennzeichnet werden.

```
function addDefNum(a:number,b:number,c:number=0) {
    return a + b + c;
}
```

### 12.8.7. Der Function-Type

---

Eine Variable, in der eine Funktion gespeichert wird, besitzt den **Function-Type**. Dieser Typ legt **Anzahl und Typ der Parameter** fest, sowie den Typ des Rückgabewertes. Nochmal die oben gezeigte, annotierte Funktion:

```
var add2 = (a:number, b:number) :number => a + b;
```

Die Variable `add2` erhält den per Typinferenz den Function-Typ **(a:number, b:number) => number**.

Einen **Function-Type** kann man, wenn man möchte, auch explizit auf der LHS annotieren:

```
// Deklaration mit Function-Type:
var calc: (a:number, b:number) => number;

// Zuweisung der eigentlichen Funktion:
calc = (a:number, b:number) :number => a + b;
```

- ✓ Der TypeScript-Compiler berücksichtigt den Function-Type beim Aufruf der Funktion und beurteilt Anzahl und Typ der übergebenen Argumente. Er moniert **falsche Typen**, sowie sowohl **Unter-** als auch **Überzahl von Argumenten** (auch Überschussargumente sind nicht zulässig!).

### 12.8.8. Function-Type und Callback-Parameter

Eine Annotation der LHS-Variable, die die Funktion aufnimmt mit einem Function-Type kann getrost unterbleiben. Anders ist dies, wenn eine Funktion **als Parameter** übergeben werden soll (also eine Callbackfunktion zur Verfügung gestellt werden soll):

```
function useCalc(a, b, calc) {  
    return calc(a, b);  
}  
  
// es wird eine Arrow-Function übergeben:  
var ergAny = useCalc(17, 4, (a,b)=>a+b )
```

Hier hätte der Compiler keine Möglichkeit, auf den Returntyp von `useCalc()` zu schließen. Das Ergebnis hätte den Typ „any“.

Auch könnte der Compiler nicht monieren, wenn als dritter Parameter keine Funktion übergeben wird (das *contextual typing* hat keine Run-Time Plausibilitätsprüfung).

Folgendes würde zur Design-Time nicht auffallen:

```
var err = useCalc(5, 7, "quatsch");
```

Man kann jedoch den Callback-Parameter als **Function-Type** annotieren:

```
function useCalcTyped(  
    a:number,  
    b:number,  
    calc: (a:number, b:number) => number  
    ) :number {  
    return calc(a, b);  
}
```

Die Fehlzuzuweisung (analog zu vorhin) fällt nun auf:

```
var err = useCalcTyped(5, 7, "quatsch")  
// Argument of Type "quatsch" not assignable ...
```



### 12.8.9. Signatur-Overloading

Mit implizitem Function-Type bei annotierten Parametern lässt der TSC nur eine Verwendung der Funktion anhand der ermittelten Signatur zu. Soll die Funktion auf verschiedene Art (Zahl, Name oder Typ der Parameter) aufrufbar sein, so ist dies über **Signatur-Overloading** definierbar.

- ✓ Hierfür bekommt eine Funktion mehrere alternativ gültige **Overload-Signaturen**, vergleichbar einem Function-Type, zugewiesen. Diese müssen vor der eigentlichen **Implementierungs-Signatur** stehen:

```
// 1. Signatur Overload:
function test(name: string) : string;

// 2. Signatur Overload:
function test(alter: number) : string;

// 3. Signatur Overload:
function test(single: boolean) : string;

// Implementierungs-Signatur:
function test(val: (string|number|boolean)) :string {
    switch(typeof val){
        case "string":
            return `Ich bin ${val}.`;
        case "number":
            return `Ich bin ${val} Jahre alt.`;
        case "boolean":
            return val ? "Bin Single." : "Kein Single.";
        default:
            return "Fehleingabe!";
    }
}
```

- ✓ Die Implementierungs-Signatur muss **allen** definierten Overload-Signaturen Genüge tun. Üblicherweise muss der/die Parameter daher mit Union-Type oder den Typ „any“ annotiert werden.

## 12.8.10. Generische Funktionen

Soll eine Funktion mit verschiedenen Datentypen arbeiten können, so erlauben **Generics**, diese „allgemein“ (als *generisch*) zu deklarieren.

Eine Funktion `getItems()`, die mit einem, später zu konkretisierenden, „allgemeinen“ Typ **T** arbeiten soll, wird wie folgt deklariert (hier sollen Arrays mit Items des allgemeinen Typs **T** per Ajax über Angabe eines URL `url` bezogen werden):

```
function getItems<T> (url:string, cb: (liste: []T) => void):void{
    $.ajax({
        url: url,
        method: "GET",
        success: function(data) {
            cb(data.items);
        },
        error : function(error) {
            cb(null);
        }
    });
}
```

Um die Items zu verarbeiten, wird der Funktion ein Callback als Argument übergeben, dessen **Function-Type** jeweils an den Typ der Daten-Arrays angepasst werden *müsste*, wenn nicht mit Generics gearbeitet werden würde. (Die Funktion wurde der Einfachheit halber mit jQuery formuliert, was für das Beispiel ohne Belang ist.)

✓ **Deklaration:** Die Funktion wird mit Angle-Brackets **<T>** hinter dem Funktionsnamen gekennzeichnet. Der Bezeichner **T** dient als Platzhalter für den eigentlichen (beim Aufruf festzulegenden) Typ.

Die spätere (tatsächliche) Verwendung der generischen Funktion **beim Aufruf** erfolgt unter Angabe des jeweiligen **konkreten** Typs:

```
getItems<User>("/api/users",function(users : Users[]) {
    for(var i; users.length; i++) {
        console.log(users[i].name);
    }
});

getItems<Order>("/api/orders", function(orders : Orders[]) {
    for(var i; orders.length; i++) {
        console.log(orders[i].total);
    }
});
```

```
    }  
  });
```

- ✓ Für den Aufruf wird der Platzhalter **T** (in den Angle-Brackets und sonst) also einfach durch den gewünschten Typ (hier **User** bzw. **Order**) ersetzt. De facto gleichen die Angle-Brackets somit Parameterklammern, mittels derer der konkrete Typ T übergeben wird, der dann überall in der Signatur den Platzhalter T ersetzt.

## 12.9. Classes in TypeScript

Im Bereich der Definition von Klassen mittels des `class`-Keywords hat TypeScript über das reine ECMA6 hinaus wiederum das Feature der **Typannotation** zu bieten. Zusätzlich zur Typannotation können für Properties mittels Modifier-Ausdrücken wie `public` oder `private` die Zugänglichkeit der Eigenschaft beschrieben werden.

### 12.9.1. Ausgangspunkt ECMA6

Betrachten wir als Ausgangspunkt eine ECMA6-Klasse `Student`, innerhalb derer ein Konstruktor definiert ist, der implizit ein Property `name` erzeugt, indem er dieses dem bei der Instanzierung modifizierten *thisObj* hinzufügt.

Diese Klasse hat noch keine TypeScript-Spezifika:

```
// ECMA6-Klasse
class Student {
  constructor(vorname, nachname) {
    this.name = vorname + " " + nachname;
  }
}
```

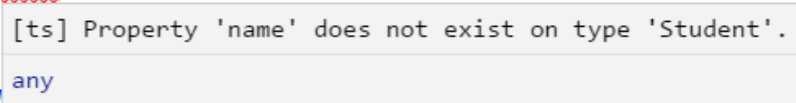
In ECMA6 erfolgt ein Aufruf wie folgt:

```
var tom = new Student("Tom", "Tomate");
```

Erzeugt wird ein Objekt der Form:

```
{
  name: "Tom Tomate"
}
```

```
class Student {
  constructor(vorname, nachname) {
    this.name = vorname + " " + nachname;
  }
}
var joe = new
```



Der TSC ist mit dieser ECMA6-Klassendefinition jedoch **nicht zufrieden**. Er stellt sich auf den Standpunkt, ein Property namens `name` „existiere an der Klasse `Student` nicht“.

- ✓ Dies liegt daran, dass TypeScript eine **Deklaration** der verwendeten Properties in der Klasse benötigt.

### 12.9.2. Deklaration und Annotation von Properties

---

Spezifisch für TypeScript ist die Notwendigkeit, ein Property mit Typannotation im Rahmen der Klasse zu **deklarieren**.

- ✓ Eine Propertydeklaration nennt **Namen** und **Typ** eines Properties, das an den Objektinstanzen erscheinen soll. Zusätzlich kann ein *Modifier* gesetzt werden (dazu später).

Hier geschieht das für das Property `name`, in das die Verkettung der übergebenen Strings für `vorname` und `nachname` eingegeben wird:

```
class Student {  
    // Deklaration eines Properties:  
    name: string;  
  
    constructor(vorname, nachname) {  
        this.name = vorname + " " + nachname;  
    }  
}  
  
var tom = new Student("Tom", "Tomate");
```

- ✓ Beachten Sie, dass hinter der Typeannotation ein **Semikolon** steht. Dies ist in der ECMA6-Syntax prinzipiell gestattet, dennoch ist der Block einer Klasse kein herkömmlicher Codeblock. In ECMA6 existiert keine Möglichkeit, ein Property zu deklarieren (auch keine Notwendigkeit).

Erzeugt wird auch hier ein Objekt der Form:

```
{  
    name: "Tom Tomate"  
}
```

...wobei TypeScript jedoch einen *Typfehler* meldet, falls sich für `name` kein Stringvalue ergibt.

Würde ein Property nicht unter Angabe des Typs deklariert werden (müssen), so könnte der TSC seinen Typ nur schwer über Typinferenz erschließen (solange Eingangsparameter nicht annotiert sind). Dies würde in vielen Fällen zum Typ „any“ führen, was nicht wünschenswert ist.

### 12.9.3. Modifier: public, private, protected, static

Die Typ-Annotation eines Properties kann mit dem Hinweis auf die Zugänglichkeit des Properties versehen werden. Hierfür wird der Annotation ein **Modifier-Keyword** (*access modifier*) vorangestellt.

- ✓ TypeScript unterscheidet bei den Properties zwischen `public`, `private`, `protected` und `static`. Grundsätzlich gilt ein Property als `public`, solange nichts anderes vermerkt ist.

#### Der Modifier public

Die Klasse könnte, mit unveränderter Wirkung, ein wenig ausführlicher also auch so geschrieben werden:

```
class Student {  
  
    // Property soll allgemein zugänglich sein:  
    public name: string;  
  
    constructor(vorname, nachname) {  
        this.name = vorname + " " + nachname;  
    }  
}
```

Das `public` Keyword vor der Annotation ändert das Verhalten der Klasse im Grunde nicht; das Property `name` wäre ohnehin *public*.

#### Der Modifier private

Setzt man das `name`-Property hingegen auf **private**, so reglementiert der TSC den Zugang zum Property. Von außen (im direkten Zugriff) gilt das Property als *nicht* erreichbar, wohl aber für Methoden des Objekts selbst.

Wir fügen der Klasse daher eine Getter-Funktion `getName()` im Prototype hinzu, die auf den Wert des `name`-Property zugreift:

```
class Student {  
  
    // Property soll nur für Klasse zugänglich sein:  
    private name: string;  
  
    constructor(vorname, nachname) {  
        this.name = vorname + " " + nachname;  
    }  
    // Getter für Zugriff auf name-Property:  
    getName() {  
        return this.name;  
    }  
}
```

**Wichtig:** Zu beachten ist, dass die kompilierte Klasse in der Runtime ein Objekt mit *öffentlichem* Property `name` erzeugt. Die Aussage „**private**“ hat also **nur während der Design-Time** des TSC eine Bedeutung.

Während der *Design-Time* (und nur dann) kann jedoch unerwünschter Direktzugriff auf das Property nun moniert werden:

```
var tom = new Student("Tom", "Tester");  
  
console.log(tom.name); // Fehler! Zugriff verboten.
```

✓ Der TSC meldet, das Property `name` sei „privat“ und nur „im Rahmen der Klasse zugänglich“.

Über den *Getter* ist das Property problemlos erreichbar, da dieser *privilegierten Zugriff* auf die Eigenschaft hat:

```
var tomName = tom.getName(); // korrekt
```

Die Kennzeichnung `private` erlaubt den **internen Gebrauch** des Properties durch *Objektmethode*n (z.B. für die Ausgabe des Werts), verhindert jedoch *unkontrollierten Direktzugriff* von aussen.

## Der Modifizier protected

Der Modifizier `protected` ist quasi die zugängliche Variante von `private`. Von Bedeutung ist dies, sobald von einer Klasse eine Subklasse abgeleitet wird. Hier wird als Beispiel von der Klasse `Student` die Klasse `Assistent` abgeleitet.

Vorerst soll das `name`-Property in `Student` als „private“ angelegt bleiben:

```
class Student {
  private name: string;
  ...
}
```

Es wird nun die abgeleitete Klasse `Assistent` gebildet, die eine eigene Methode `getName()` implementiert, die auf `name` (aus `Student`) zugreift:

```
class Assistent extends Student {
  private titel: string;
  constructor(vorname, nachname, titel){
    super(vorname, nachname);
    this.titel = titel;
  }
  getName(){
    // this.titel ist privat in Superklasse:
    return this.titel + " " + this.name; // Problem!
  }
}
```

```
class Assistent extends Student {
  private titel: string;
  constructor(vorname, nachname, titel){
    super(vorname, nachname);
    this.titel = titel;
  }
  getName(){
    // this.titel ist privat in Sup
    return this.titel + " " + this.name; // Problem!
  }
}
```

[ts] Property 'name' is private and only accessible within class 'Student'.

(property) Student.name: string

Der TSC moniert den Zugriff auf `this.name` im Getter, da dieser (das ist die Bedeutung von „private“) nur in der Klasse `Student` *selbst* zugänglich sei. Ein Zugriff aus der abgeleiteten Klasse `Assistent` ist verboten.

Dies ändert sich, sobald in der Basisklasse `Student` das Property als „protected“ gekennzeichnet wird:



```
class Student {
    protected name: string;
    ...
}
```

Der Getter der abgeleiteten Klasse `Assistent` funktioniert nun:

```
class Assistent extends Student {
    private titel: string;
    constructor(vn:string, nn:string, titel:string){
        super(vn, nn);
        this.titel = titel;
    }
    getName(){
        // this.titel ist protected in Superklasse:
        return this.titel + " " + this.name; // o.k!
    }
}
```

Im Gegensatz zu **private** erlaubt **protected** den internen Zugriff auf ein Property auch durch Objektmethoden (über die Klasse hinaus), die als Teil einer Subklasse definiert wurden.

## Der Modifier **static**

Ein als **static** bezeichnetes Property wird nicht einer *Instanz* der Klasse, sondern der *Klasse selbst* hinzugefügt.

- ✓ Ein **statisches Property** muss daher bei der Deklaration der Klasse mit einem **Wert** belegt werden.

Zugänglich gemacht wird es allein über die Klasse.

```
class Student {

    // Property existiert nur an Klasse:
    static gattung:string = "Mensch";

    private name: string;

    constructor(vorname:string, nachname:string) {
        this.name = vorname + " " + nachname;
    }
    // Getter für Zugriff auf name-Property:
```

```
        wasBinIch() {
            console.log(this.name, ", ", " ", Student.gattung)
        }
    }

    var tom = new Student("Tom", "Tester");
    tom.wasBinIch(); // Tom, Mensch
```

#### 12.9.4. Typisierung von Konstruktor-Parametern

TypeScript erlaubt die Typisierung von Parametern des Konstruktors. Dies geschieht auf die gewöhnliche Art, indem der Typ dem Parameternamen mit Doppelpunkt angefügt wird.

```
class Student {
    name: string;
    // Parameter mit Typ:
    constructor(vorname:string, nachname:string) {
        this.name = vorname + " " + nachname;
    }
}
```

- ✓ Eine Typannotation der Parameter ist optional, aber im Allgemeinen sinnvoll.

Erhält ein Property bei der Deklaration einen falschen Typ (z.B. `number`), wird *ohne* Parameter-Annotation möglicherweise *nicht* moniert, dass bei der Instanzierung ein falscher Typ zugewiesen werden könnte. Dies liegt daran, dass die Parameter in diesem Fall den impliziten Typ „any“ besitzen.

In unseren Fall greift jedoch die Typerschließung bereits über den Ausdruck der Zuweisung (dort ist ein String enthalten):

```
class Student {
  name:number;
  constructor(vorname, nachname) {
    [ts] Type 'string' is not assignable to type 'number'.
    (property) Student.name: number
    this.name = vorname + " " + nachname;
  }
}
```

### 12.9.5. Modifier „public“ vor Konstruktor-Parametern

Der `public`-Modifier *vor einem Parameter des Konstruktors*, bewirkt, dass für den Parameter automatisch ein gleich benanntes **öffentliches Property** angelegt wird.

```
class Student {
  // Deklaration eines Properties:
  name: string;

  constructor(public vorname, public nachname) {
    this.name = vorname + " " + nachname;
  }
}
```

```
var tom = new Student("Tom", "Tomate");
```

Erzeugt wird ein Objekt folgender Form:

```
{
  name: "Tom Tomate",
  vorname: "Tom",
  nachname: "Tomate"
}
```

Nach ECMA5 übersetzt passiert folgendes (zunächst *unter Weglassen* des Keywords):

```
class Student {
  name: string;
  // public-Modifier für ARGs weggelassen:
  constructor(vorname, nachname) {
    this.name = vorname + " " + nachname;
  }
}
```

```
    }  
  }
```

... was vom TS-Compiler in folgenden ECMA5-Konstruktor übersetzt wird:

```
var Student = (function () {  
  function Student(vorname, nachname) {  
    this.name = vorname + " " + nachname;  
  }  
  return Student;  
})();
```

Es wird in der Instanz also nur das name-Property erzeugt.

Mit dem Modifier `public` vor den **Parametern** ...

```
class Student {  
  public name: string;  
  constructor(public vorname, public nachname) {  
    this.name = vorname + " " + nachname;  
  }  
}
```

... erhalten wir folgende Ausgabe:

```
var Student = (function () {  
  function Student(vorname, nachname) {  
    // Properties existieren jetzt:  
    this.vorname = vorname;  
    this.nachname = nachname;  
    this.name = vorname + " " + nachname;  
  }  
  return Student;  
})();
```

In diesem Fall werden neben dem name-Property auch Properties für `vorname` und `nachname` definiert.

Zu beachten ist, dass für die durch `public` Modifier generierten Properties *keine explizite Typdeklaration* vorliegt (sie ist, wie gezeigt, nicht obligatorisch). Die hat jedoch den Nachteil, dass diese Properties, solange für den Parameter keine Typdeklaration vorliegt, den **Typ „any“** besitzen.

```
class Student {
    name:string;
    constructor(public vorname, public nachname) {
        this.name = vorname + " " + nachname;
    }
}
var tom = new Student("Tom ", "Tomate");
```

(property) Student.vorname: any

```
console.log(tom.vorname)
```

Es ist daher sinnvoll, implizite Properties mit einem Typ zu belegen.

```
class Student {
    name:string;
    constructor(public vorname:string,
        public nachname:string ) {
        this.name = vorname + " " + nachname;
    }
}
var tom = new Student("Tom ", "Tomate");
```

(property) Student.vorname: string

```
console.log(tom.vorname)
```

### 12.9.6. Weitere Modifier für Konstruktor-Parameter

Auch Modifier wie `private` oder `protected` können einem Parameter vorangestellt werden. In beiden Fällen wird ebenfalls eine Eigenschaft mit dem Namen des Parameters erzeugt, für die jedoch während der Design-Time eine Zugriffseinschränkung besteht (nicht in der Runtime!).

Hier ist die Eigenschaft `nachname` nur über die Getter-Funktion `getNachname()`, nicht aber *direkt* greifbar.

```
class Student {
    name:string;
    constructor(public vorname:string,
        private nachname:string ) {
        this.name = vorname + " " + nachname;
    }
}
```

```
    getNachname() {  
        return this.nachname;  
    }  
}  
  
var tom = new Student("Tom ", "Tomate");
```

```
[ts] Property 'nachname' is private and only accessible  
within class 'Student'.  
(property) Student.nachname: string  
console.log(tom.nachname);
```

**Achtung:** Nach dem Kompilieren kann in der Run-Time eine als `private` deklarierte Eigenschaft, die durch einen Konstruktorparameter erzeugt wurde, nicht mehr als solche erkannt werden.

Um zumindest einen Hinweis darauf zu liefern, kann man auch hier der Konvention folgen, sie mit einem **Unterstrich** zu präfixen:

```
constructor(public vorname:string,  
             private nachname:string ) {  
    this.name = vorname + " " + _nachname;  
}
```

- ✓ Setzt man den Modifier statt auf `private` auf `protected`, so kann auf `nachname` auch innerhalb **abgeleiteter Klassen** durch deren Methoden zugegriffen werden

Der Modifier **static** kann auf Parameter *nicht* angewendet werden!

Eine solche Eigenschaft wird *der Klasse selbst* zugewiesen. Er kann nicht bei der Erstellung einer Instanz dynamisch mit einem Wert belegt werden, sondern wird im Rahmen der Klassendeklaration mit einem Wert initialisiert.

### 12.9.7. Klassen als Annotation

---

Ebenso wie Interfaces (siehe dort) können auch **Klassen** als *Typannotationen* eingesetzt werden.

```
class Auto {  
    constructor(public info:string) {  
        // info ist ein implizites Property  
    }  
    hupen(){  
        return "Tuuuut!";  
    }  
}  
  
var kaefer: Auto = new Auto("1970 VW Käfer, rot");  
  
kaefer.hupen(); // Tuuuut!
```

Eine Annotation mit einer Klasse macht Sinn, wenn eine Objektinstanz einer Klasse als **Parameter** einer *Funktion* oder als **Wert** eines *Properties* eines Objekttyps eingesetzt wird.

```
class Student {  
    name:string;  
    constructor(public vorname:string,  
                public nachname:string,  
                public fahrzeug:Auto) {  
        this.name = vorname + " " + nachname;  
    }  
}  
  
var tom = new Student("Tom ", "Tomate", kaefer);
```

**Achtung:** Es muss daran erinnert werden, dass TypeScript kein *nominales*, sondern ein **strukturelles Typsystem** besitzt. Besitzen wir eine Klasse **Lkw**, das über alle *Properties* von **Auto** verfügt ...

```
class Lkw {  
    zuladung:string;  
    constructor(public info:string){  
        // Property zuladung nur an Lkw:  
        this.zuladung = "5 Tonnen";  
    }  
    hupen(){
```

```

        return "Tröööö!!";
    }
}

```

```
var meinMan:Lkw = new Lkw("1965 MAN 5 Tonner, blau");
```

... so kann eine Instanz von Lkw problemlos einem Student zugewiesen werden, auch wenn das Property `zuladung`, das hier existiert, bei Auto nicht definiert ist:

```
var tom = new Student("Tom ", "Tomate", meinMan);
```

```

var meinMan: Lkw

var tom = new Student("Tom ", "Tomate", meinMan);

(property) Student.fahrzeug: Auto

console.log(tom.fahrzeug)

```

Dies liegt daran, dass die Klasse Lkw dem *Interface* von Auto gerecht wird, indem es *all seine Properties* implementiert. Fehlt jedoch eine Eigenschaft, bzw. ist sie anders benannt (eine anderer Typname ist nicht ausschlaggebend), so meldet der TSC einen Interface-Fehler:

```

class Lkw {
    constructor(public info:string){
    }
    // Methode hupen() fehlt!
    bremsen(){
        return "Quietsch!!";
    }
}

```

```

[ts]
Argument of type 'Lkw' is not assignable to parameter o
f type 'Auto'.
  Property 'hupen' is missing in type 'Lkw'.
var meinMan: Lkw
var tom = new Student("Tom ", "Tomate", meinMan);

```



### 12.9.8. Abstrakte Basisklassen

---

Bei einer **abstrakten Klasse** handelt es sich um eine Klasse, die zwar zwar eigene Implementierungsdetails enthält, aber selbst nicht instanzierbar ist. Ableitende Klassen müssen die abstrakten Bestandteile zunächst implementieren, um eine Instanzierung zu ermöglichen.

```
abstract class Tier {  
    constructor(public anzahlBeine: number) {  
    }  
    abstract rufen(): string;  
    hallo() {  
        console.log(`  
            ${this.rufen()}! Habe ${this.anzahlBeine} Beine!  
        `);  
    }  
}
```

Diese abstrakte Klasse besitzt eine *abstrakte* Methode `rufen()`, die hier nicht näher definiert wird (sieht man von Rückgabebetyp). Da eine Klasse, die auf der abstrakten Klasse aufbaut aber gezwungen ist, diese Methode zu implementieren, ist dies auch nicht nötig.

Hier zwei Klassen, die auf der abstrakten Klasse Tier aufbauen:

```
class Hund extends Tier {  
    constructor() {  
        super(4);  
    }  
    rufen() {  
        return "Wuff!";  
    }  
}  
  
class Ente extends Tier {  
    constructor() {  
        super(2);  
    }  
    rufen() {  
        return "Quack!";  
    }  
}
```

```
var waldi = new Hund();  
waldi.rufen(); // Wuff! Habe 4 Beine!
```

## 12.10.Interfaces

---

Die Definition von **Interfaces** im Sinne der „Beschreibung einer Objektstruktur“ ist auch losgelöst vom Konzept der Klassen verfügbar.

- ✓ In ECMA6 fehlt das Konzept von Interfaces. Aus diesem Grund können Interfaces nicht im kompilierten JS-Code sichtbar werden!

Der Sinn eines Interfaces ist die Beschreibung der *Schnittstelle*, die ein Objekt bieten muss, um als „einer Klasse“ zugehörig erkannt zu werden (eher im Sinne von „Ducktyping“) oder generell als „verwendbar in einem Kontext“ gewertet zu werden.

TypeScript sieht **zwei Möglichkeiten** vor, wie ein Interface definiert werden kann. Die gebräuchliche Variante verwendet das Keyword `interface`, um ein „**benanntes**“ **Interface** zu erstellen, das (wie gezeigt werden wird) ähnlich einer Klasse eingesetzt wird. Für einfache Sachverhalte kann aber auch eine „**anonyme**“ **Interface-Definition** in Form einer *inline annotation* erfolgen, beispielsweise im Rahmen der Deklaration einer Variablen.

### 12.10.1. Anonymes Interface

---

Ein „anonymes“ Interface gleicht einer komplexen Typdefinition. Auf diesem Weg kann eine Variable typisiert werden, die (später) ein **Objekt** aufnehmen soll (als Literal oder als instanzierte Klasse).

Das anonyme Interface stellt sicher, dass dieses Objekt die festgelegten Mindesteigenschaften besitzt.

```
// anonyme Interface-Deklaration (inline)
var person: {vn: string; nn: string; alter: number};
```

Später wird dieser Variable ein Objekt zugewiesen:

```
person = {
  vn: "Peter",
  nn: "Panter",
  alter: 42,
  haustier: "Dackel"
};
```

Der TSC akzeptiert das übergebene Objekt, da dieses die Eigenschaften `vn`, `nn` und `alter` aufweist und diese jeweils den korrekten Typ besitzen.

Das zusätzliche Property `haustier` ist für ihn irrelevant (ein übergebenes Objekt darf *jederzeit* über das Interface *hinausgehende* Eigenschaften besitzen).

## 12.10.2. Benanntes Interface

Benannte Interfaces werden mittels des Keywords `interface` deklariert.

Das Keyword `interface` existiert nicht in ECMA6!

Ein Interface beschreibt stets **ausschließlich sichtbare**, bzw. zugängliche Teile einer Objektstruktur.

- ✓ **Folgerung:** Es können also nicht als „private“ oder „protected“ ausgezeichnete Properties Teil eines Interfaces sein. Dasselbe gilt für „static“-Properties, die ohnehin nicht an Instanzen in Erscheinung treten.
- ✓ Es ist hingegen *erlaubt*, Properties des Interfaces als „**optional**“ zu kennzeichnen. Dies geschieht wie bei Funktionsparametern mit dem Fragezeichen, das dem Bezeichner der optionalen Eigenschaft folgt.

Hier wird ein **Interface** `IPerson` deklariert:

```
// Das Interface bestimmt einen „Objekttyp“
interface IPerson {
    // Objekt muss diese Properties/Typen besitzen
    vn: string;
    nn: string;
    alter: number;
}
```

Ein anschließend erzeugtes Objektliteral `torsten` tut diesem Interface Genüge, ohne dass dem Objekt ein Konstruktor oder eine Klasse zugrunde liegt:

```
// Literal, also keine "echte" Person
var torsten = {vn: "Torsten", nn: "Tester", alter: 42};
```

- ✓ **Anmerkung:** Es ist nicht vorgeschrieben, *wie* Interfaces benannt werden. Jedoch ist es praktisch, sie von Klassen (die ebenfalls als Typpannotation verwendet werden können) unterscheiden zu können.

In diesem Beispiel wurde den Bezeichner daher ein „I“ vorangestellt.

### 12.10.3. Funktion implementiert Interface

---

Im Rahmen von Funktionen werden Interfaces verwendet, um den Typ (vielmehr: die Struktur) eines komplexen **Eingabewertes** festzulegen.

- ✓ Der TSC kann daraufhin erkennen, ob ein der Funktion übergebener Wert zulässig ist.

Hier wird eine Funktion `hallo()` definiert, die als Eingabe ein Objekt benötigt, das dem **Person-Interface** entspricht. Übergibt man das Objektliteral `torsten` der Funktion, so wird es akzeptiert.

```
// Interface als Parameterannotation:
function hallo( person: IPerson ) {
    return "Hallo, " + person.vn + " " + person.nn;
}

// Objekt wird akzeptiert, da es dem Interface genügt:
document.body.innerHTML = hallo(torsten);
```

- ✓ Ein Interface kann auch zur Beschreibung des *Rückgabewerts* einer Funktion verwendet werden.

### 12.10.4. Klasse implementiert Interface

---

In TypeScript kann eine **Klasse** ein Interface „implementieren“, was mit dem Keyword `implements` beschrieben wird, gefolgt von einem oder mehreren Interface-Bezeichnern (darüber mehr in Folge).

- ✓ Der TSC moniert, sofern bei einer Klasse, die „behauptet“, ein Interface zu implementieren, ein Interface-Property **fehlt**.

Hier ein Beispiel:

```
class Mensch implements IPerson {
    vn:string;           // vorgeschrieben!
    nn:string;           // vorgeschrieben!
    alter:number;        // vorgeschrieben!

    constructor(vn:string, nn:string){
    }
}
```

Es ist allerdings gestattet, über die im Interface hinaus beschriebenen Eigenschaften *weitere* Properties zu definieren:

```
class MenschMitFon implements IPerson {
    vn:string;
    nn:string;
    alter:number;
    fon:string; // nicht vorgeschr., daher erlaubt

    constructor(vn:string, nn:string,
                alter:number, fon:string){
    }
}
```

### 12.10.5. Implementierung mehrerer Interfaces

---

Es ist für eine Klasse jederzeit möglich, *mehrere Interfaces* gleichzeitig zu implementieren. Schreiben wir uns eines:

```
interface IFon {
    fon:string;
}
```

... sodass die Klasse von vorhin exakt beschrieben werden kann:

```
class MenschMitFon implements IPerson, IFon {
    vn:string;
    nn:string;
    alter:number;
```

```

        fon:string;    // jetzt vorgeschrieben!!

        constructor(vn:string, nn:string, fon:string){
        }
    }

```

- ✓ **Merke:** Eine Klasse kann zwar nur von *einer* anderen Klasse (mittels `extends`) *erben*, aber **beliebig viele Interfaces** *implementieren*.

### 12.10.6. Klassen als Interfaces

Außer den „offiziell“ über das Keyword `interface` generierten Interfaces, kann auch eine existierende *Klasse* wie ein Interface verwendet werden.

- ✓ Dies gilt analog auch für Annotation von Funktionsparametern oder Returnwerten!

Greifen wir auf die vorhin definierte Klasse `MenschMitFon` zurück:

```

class MenschMitFon implements IPerson {
    vn:string;
    nn:string;
    alter:number;
    fon:string;    // nicht vorgeschr., daher erlaubt

    constructor(vn:string, nn:string, fon:string){
    }
}

```

... und erzeugen ein neues Interface `IHaustier`:

```

interface IHaustier {
    haustier:string;
}

```

... so können wir eine neue Klasse `MenschMitHaustier` wie folgt ableiten:

```

class MenschMitHaustier
    implements MenschMitFon, IHaustier {
    vn:string;
    nn:string;
}

```

```
    alter:number;  
    fon:string;  
    haustier:string;  
  
    constructor(vn:string,  
                nn:string,  
                fon:string,  
                haustier:string){  
    }  
}
```

- ✓ **Merke:** Auch Klassen können mit dem `implements`-Keyword wie *Interfaces* verwendet werden.

## 13. Decorator – Annotation und Modifikation

- ✓ In ECMA2016 (ECMAScript 7) wird das Konzept des „Decorators“ (Dekorierer) in JavaScript eingeführt. Der Vorschlag stammt aus dem Entwicklerteam von Googles *AtScript* (dessen Entwicklung mit Mocoosofts *TypeScript* zusammengelegt wurde).

**Achtung:** Aktuell werden Dekoratoren noch in keiner JavaScript-Runtime unterstützt! Alle folgenden Beispiele müssen daher in TypeScript geschrieben und kompiliert werden.

Bei einem **Decorator** handelt es sich (salopp ausgedrückt) um eine *Funktion*, die verwendet wird, um ein „Ziel“ (bei dem es sich um eine **Klasse**, eine **Methode**, ein **Property** oder einen **Parameter** handeln kann), zu *modifizieren*.

- ✓ Eine solche Modifikation kann in einer *Annotation*, einer *Ergänzung* oder Erweiterung bestehen, kann aber auch die komplette *Ersetzung* des Targets durch ein mittels der Decorator-Funktion festgelegtes Objekt bedeuten.

Der „Decorator“ ist eines der grundlegenden *GoF-Entwurfsmuster* des Softwaredesigns und zählt zu den **Strukturmustern** („*structural pattern*“). Es findet sich in vergleichbarer Form auch bei *Python* oder *C#*.

Beim Verarbeiten von Dekoratoren durch den TSC ist es sinnvoll, in der **tsconfig.json** das Flag für `experimentalDecorators` bei den `compilerOptions` zu setzen, um Fehlermeldungen zu vermeiden:

```
{
  "compilerOptions": {
    ...
    "experimentalDecorators": true
  },
  ...
}
```

Wir werfen einen Blick auf die *vier Typen* von Dekoratoren und anschließend auf das Konzept der Decorator-Factory.

### 13.1. Class-Decorator

Beginnen wir mit dem meistverwendeten Typ, dem Class-Decorator. In TypeScript wird der **Class-Decorator** wie folgt definiert:

```
declare type ClassDecorator =
  <TFunction extends Function>(target: TFunction)
=> TFunction | void;
```

- ✓ Ein *Class-Decorator* wird **einmal** ausgeführt, unmittelbar bevor eine Instanz dieser Klasse erzeugt wird. Die Klasse kann zu diesem Zeitpunkt dann erweitert oder modifiziert werden.

Hier ein exemplarischer Class-Decorator:

```
function meinClassDecorator(target) {
  // tut was mit target
}
```

Der Decorator erhält beim Aufruf **ein Argument**, nämlich die Klasse als *target*.



- ✓ Es existiert entweder *kein Rückgabewert* (der Konstruktor der dekorierten Klasse bleibt unverändert) oder es wird eine *Konstruktorfunktion* zurückgegeben, die den Konstruktor der dekorierten Klasse *ersetzt*.

Der Decorator wird auf eine **Klasse** (ein Konstruktor in Schreibweise mit `class`-Keyword) angewandt, indem der Klassendeklaration der *Decoratorname* (mit Präfix `@`) vorangestellt wird.

**@meinClassDecorator**

```
class ichWerdeDekortiert {  
    constructor() {  
        // ich bin der Konstruktor der Klasse  
    }  
}
```

- ✓ Der Einsatz der Klammeroperatoren (als Funktionsaufruf!) ist *nicht* erforderlich, genau wie eine Übergabe des Parameters *nicht* erfolgt. Beides ergibt sich implizit aus der Position des Decoratoraufrufs.

Eine Klasse kann auch *mehrfach* dekoriert werden:

```
@ersterClassDecorator  
@zweiterClassDekorator  
class doppeltDekoriert { ... }
```

### 13.1.1. Ausloggen der Instanzierung einer Klasse

Wir möchten jeweils über einen Log von Instanzierung eines Objekts der Klasse `Person` erfahren, weshalb diese mit einem **Class-Decorator** `logClass` versehen wird:

**@logClass**

```
class Person {  
    public vorname: string;  
    public nachname: string;  
  
    constructor(vorname: string, nachname: string) {  
        this.vorname = vorname;  
        this.nachname = nachname;  
    }  
}
```

- ✓ In diesem Fall wird die Klasse durch eine neue Klasse, welche auf der alten aufbaut (*extends*), **ersetzt!** Wir geben daher eine neue Klasse zurück.

Die Class-Decorator-Funktion muss hierfür wie folgt aussehen:

```
function logClass(target: any) {  
  
    return class extends target {  
        test:string;  
  
        constructor(...args){  
            super(...args);  
            console.log('Neue Instanz erstellt: ' + target.name);  
            this.test = "Test Class Decorator";  
        }  
    }  
}
```

- ✓ Entscheidend ist, dass die **ursprüngliche Klasse überschrieben** wird, wobei das Original als Superklasse in die neue Klasse *einbezogen* wird.

Die Instanziierung einer Person wird nun ausgelagt:

```
// -> Neue Instanz erstellt: Person  
var peter = new Person("Peter", "Panter");
```

### 13.1.2. Erweitern des Prototypes einer Klasse

---

Einfacher ist es, wenn lediglich das **Prototype-Objekt erweitert** werden muss und der Konstruktor belassen werden kann. Auch in diesem Falle geben wir eine neue Klasse zurück, in der die zusätzliche Methode definiert wird.

- ✓ Der Konstruktor soll unverändert bleiben und wird daher in der Erweiterungsklasse nicht definiert.

Hier soll der Klasse `Person` über einen Decorator `addHallo` eine `hallo`-Methode hinzugefügt werden.

```
@addHallo
class Person {
    public vorname: string;
    public nachname: string;

    constructor(vorname : string, nachname : string) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
}
```

Die Decorator-Funktion ist diesmal überschaubar:

```
function addHallo(target: any) {
    console.log("Fügt der Klasse eine hallo-Methode hinzu.");
    return class extends target {
        // keine Definition eines Konstruktors hier!
        hallo() {
            console.log("Hallo, ich bin ", this.vorname, " ",
                        this.nachname, "!");
        }
    }
}
```

Eine instanzierte Person besitzt nun automatisch eine hallo-Methode:

```
var peter = new Person("Peter", "Panter");
console.log(peter);
peter.hallo(); //-> Hallo, ich bin Peter Panter!
```

## 13.2. Method-Decorator

Der **Method-Decorator** dekoriert, wie der Name bereits sagt, eine Methode im Rahmen einer *Klassendeklaration*. In TypeScript wird der Method-Decorator wie folgt definiert:

```
declare type MethodDecorator =
    <T>(target: Object, propertyKey: string | symbol,
    descriptor: TypedPropertyDescriptor<T>)
    => TypedPropertyDescriptor<T> | void;
```

Der Decorator erhält **drei Argumente**.

1. Das erste Argument *target* ist ein Zeiger auf das **Prototype-Objekt** der Klasse, welcher die dekorierte Methode gehört (Achtung, im Fall, dass eine statische Methode dekoriert wird, wird ein Zeiger auf den Konstruktor selbst übergeben!).
2. Das zweite Argument nennt den **Bezeichner** der Methode als *key* (in Form eines Strings bzw. ggfs eines Symbols).
3. Das dritte Argument ist der für die Methode definierten **Property-Descriptor** als *descriptor*.

Betrachten wir ein einfaches Beispiel:

```
class Beispiel {  
    @log  
    verdoppeln(n: number) {  
        return n * 2;  
    }  
}
```

Der Decorator der Methode `verdoppeln` soll das an die Methode übergebene Argument `n` ausloggen, wenn die Methode aufgerufen wird.

Der Decorator wird als Funktion deklariert. Wie oben in der Signatur beschrieben, müssen *drei Parameter* definiert werden. Die Funktion besitzt (laut Signatur) entweder *keinen* Returnwert (*void*) oder gibt einen *Property-Descriptor* zurück.

Ohne Returnwert wird die Methode nicht modifiziert (es können aber Seitenwirkungen definiert werden):

```
function log(target: Function, key: string, descriptor: any) {  
    console.log("Method-Decorator aufgerufen.");  
    return;  
}
```

Interessanter ist es natürlich, wenn die Methode *modifiziert* oder *ergänzt* wird.

**Ziel:** Hier soll neben der eigentlichen Arbeit der Methode, nämlich dem Verdoppeln einer eingegebenen Zahl, ein Log des Funktionsaufrufs hinzugefügt werden.

- ✓ Der Decorator muss daher *laut TS-Doku* einen **Property-Descriptor** zurückgeben, der dann die dekorierte Methode ersetzt.

```

function log(target: Function, key: string, descriptor: any) {

    console.log("Method-Decorator aufgerufen.");

    // Referenz auf dekorierte Methode speichern:
    var originalMethod = descriptor.value;

    return {
        value: function (...args: any[]) {
            // Argumentliste für Ausgabe in String verwandeln
            var a = args.map(a => JSON.stringify(a)).join();

            // Originalmethode aufrufen
            var erg = originalMethod.apply(this, args);

            // Ergebnis für Ausgabe in String verwandeln
            var r = JSON.stringify(erg);

            // Log an der Konsole
            console.log(`Log: ${key}(${a}) => ${r}`);

            // das Ergebnis, wie
            return erg;
        }
    };
}

```

Das Ergebnis ist, wie gewünscht, eine dekorierte Methode.

```

var beispiel = new Beispiel();
// -> Log: verdoppeln(5) => 10
var erg = beispiel.verdoppeln(5);

```

### 13.3. Property-Decorator

---

Auch einzelne Properties können im Rahmen einer Klasse dekoriert werden. In TypeScript wird der **Property-Decorator** wie folgt definiert:

```

declare type PropertyDecorator =
(target: Object, propertyKey: string | symbol)
=> void;

```

Der Decorator erhält **zwei Argumente**, nämlich das Prototype-Objekt der Klasse innerhalb derer das Property dekoriert wird als *target*, sowie den Bezeichner des Ziel-Properties als *propertyKey*. Die Funktion gibt *keinen Wert* zurück (*void*).

- ✓ Die Funktion gibt keinen Wert zurück, weil sie grundsätzlich das dekorierte Property durch eine Neudefinition *ersetzt*.

Der Decorator wird auf die Typdefinition eines Properties angewendet (hier auf `vorname`):

```
class Person {  
  
    @logProperty test:string;  
    public vorname: string;  
    public nachname: string;  
  
    constructor(vorname : string, nachname : string) {  
        this.vorname = vorname;  
        this.nachname = nachname;  
    }  
}
```

Die Decorator-Funktion definiert hier einen **Getter** und einen **Setter**, die soll das Auslesen und das Setzen eines Propertywertes ausloggen. Hierfür wird das Zielproperty in der Funktion erst *entfernt* (`delete`) und dann durch eine Neudefinition *ersetzt*:

```
function logProperty(target: any, key: string) {  
  
    // aktueller Wert der Eigenschaft:  
    let _val;  
  
    // Gettermethode mit Log  
    let getter = function () {  
        console.log(`Get: ${key} => ${_val}`);  
        return _val;  
    };  
  
    // Settermethode mit Log  
    let setter = function (newVal) {  
        console.log(`Set: ${key} => ${newVal}`);  
        _val = newVal;  
    };  
  
    // Löschen, wenn lösbar:  
    if (delete this[key]) {  
  
        // Property mit Getter und Setter neu machen:
```

```
    Object.defineProperty(target, key, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true
    });
  }
}
```

Der Getter und der Setter arbeiten (dies ist üblich für Accessor-Descriptors) auf einer Closure mit der lokalen Variable `_val`, die jeweils den aktuellen Wert des Properties enthält.

```
var panter = new Person("Peter", "Panter");

// -> Set: test => Ein Test!
// -> Get: test => Ein Test!
panter.test = "Ein Test!";

console.log(panter.test); // -> Ein Test!
```

Beachten Sie, dass Instanzeigenschaften auf diese Art nicht sinnvoll dekoriert werden, da die neu erstellten Properties im Prototype-Objekt der Klasse zu liegen kommen. Dies ist nicht anders möglich, da die Dekoration statisch erfolgt. Zu diesem Zeitpunkt existieren jedoch noch keine Instanzen.

## 13.4. Parameter-Decorator

Auch Parameter, die einer Methode der Objektinstanz übergeben werden, können dekoriert werden. In TypeScript wird der **Parameter-Decorator** wie folgt definiert:

```
declare type ParameterDecorator =
(target: Object, propertyKey: string | symbol,
parameterIndex: number)
=> void;
```

Als erstes Argument `target` wird das Prototype-Objekt der Klasse übergeben, das zweite Argument `propertyKey` ist der Stringname der Methode, zu der der dekorierte Parameter gehört. Als letztes wird der Index

`parameterIndex` des Parameters innerhalb der Argumente seiner Methode übergeben.

- ✓ Der Parameter-Decorator gibt keinen Wert zurück (bzw. der Returnwert wird ggfs. ignoriert!).

Erweitern wir die Klasse `Person` um eine `hallo`-Methode, die einen Parameter entgegennimmt:

```
class Person {
  public vorname: string;
  public nachname: string;

  constructor(vorname : string, nachname : string) {
    this.vorname = vorname;
    this.nachname = nachname;
  }
  public hallo(msg: string): void {
    console.log(this.vorname + " " +
      this.nachname + " sagt " + msg);
  }
}
```

Der Parameter `msg` der `hallo`-Methode soll dekoriert werden, um (dies geschieht über einen weiteren Decorator) das übergebene Argument auszuloggen. Hierzu definieren wir einen Decorator wie folgt:

```
function logParam(target: any, key: string, ind: number){

  // Prop anhand des Namens der Methode erzeugen
  var metadataKey = `log_${key}_parameters`;

  if (Array.isArray(target[metadataKey])) {

    // ParamIndex hinzufügen falls key bereits existiert
    target[metadataKey].push(ind);

  } else {

    // ... ansonsten ParamIndex dem key zuweisen
    target[metadataKey] = [ind];
  }
}
```



```

    }
  }

```

- ✓ Es wird ein *key* am Objekt (genauer, in dessen Prototype-Objekt) erzeugt, der den *Index* des zu betrachtenden Parameters als Wert und den *Namen* der hierfür zu beobachtenden Methode im Namen führt.

Der Decorator wird wie folgt angewendet:

```

class Person {
  public vorname: string;
  public nachname: string;

  constructor(vorname : string, nachname : string) {
    this.vorname = vorname;
    this.nachname = nachname;
  }
  public hallo(@logParam msg: string): void {
    console.log(this.vorname + " " +
      this.nachname + " sagt " + msg);
  }
}

```

### 13.4.1. Ausloggen des dekorierten Parameters

Wie eben gezeigt, bewirkt das Dekorieren des Parameters bislang nichts als das Hinzufügen eines Properties am Objekt, das den Key der zu überwachenden Methode und den Index (die Indices) des/der zu überwachenden Parameter enthält.

Dieses Property kann von einem Method-Decorator eingesetzt werden, um festzustellen, welches oder welche Argumente einer Methode ausgeloggt werden sollen (statt sie z.B: pauschal alle zu loggen).

Der Method-Decorator von vorhin wird dafür leicht umgebaut.

```

function logMethod(target: any, key: string, descr: any) {

  // Originalmethode speichern
  var originalMethod = descr.value;

  descr.value = function(...args: any[]) {

```

```

// sind Indices hinterlegt, dann hier:
var metadataKey = `log_${key}_parameters`;
var indices = target[metadataKey];

// Indices definiert? Dann ist dies ein Array...
if (Array.isArray(indices)) {

    // okay, alle gekennzeichneten Args ausloggen:
    console.log("Logparams erkannt!");
    for (var i = 0; i < args.length; i++) {
        if (indices.indexOf(i) !== -1) {
            var arg = args[i];
            var argStr = JSON.stringify(arg) || arg.toString();
            console.log(`Log: ${key} arg[${i}]: ${argStr}`);
        }
    }

    // falls Methode Rückgabewert besitzt (hier unnötig):
    var result = originalMethod.apply(this, args);
    return result;

} else {

    // ansonsten: alle ausloggen
    console.log("Keine Logparams erkannt!");
    var a = args.map(a => (JSON.stringify(a) ||
        a.toString())).join();
    var result = originalMethod.apply(this, args);
    var r = JSON.stringify(result);
    console.log(`Log: ${key} (${a}) => ${r}`);
    return result;
}
}
return descr;
}

```

Nun werden beide Dekoratoren auf die Klasse angewendet:

```

class Person {
    public vorname: string;
    public nachname: string;

    constructor(vorname : string, nachname : string) {
        this.vorname = vorname;
        this.nachname = nachname;
    }

    @logMethod

```

```

    public hallo(@logParam msg: string): void {
        console.log(this.vorname + " " +
            this.nachname + " sagt " + msg);
    }
}

```

Dies ergibt folgendes Verhalten:

```

var peter = new Person("Peter", "Panter");
console.log(peter.log_hallo_parameters); // -> [0]

// -> hallo arg[0]: "Na toll!"
peter.hallo("Na toll!");
// -> Peter Panter sagt Na toll!

// -> hallo arg[0]: "Na sowas..."
peter.hallo("Na sowas...", "Test 1", "Test 2");
// -> Peter Panter sagt Na sowas...

```

## 13.5. Decorator-Factory

Wie bislang gezeigt, werden Dekoratoren *ohne* explizite Übergabe von Argumenten aufgerufen. Es ist auch nicht möglich, im Rahmen des Aufrufs einen Wert zu übergeben! Um dies zu umgehen, setzt man eine **Decorator-Factory** ein, die einen Parameter entgegennehmen kann.

- ✓ Eine *Decorator-Factory* ist eine Funktion, die wie ein Decorator eingesetzt wird, dabei Argumente übernimmt und eine *Decorator-Funktion* zurückgibt, die dann unmittelbar angewendet wird.

**Angular 2** macht sich das Prinzip der *Decorator-Factory* zunutze, um seine @Component-Dekoratoren zu formulieren, bei deren Aufruf ein *Konfigurationsobjekt* übergeben werden kann.

Eine solche Factory könnte beispielsweise wie folgt implementiert werden:

```

function classDecoratorFactory(confObj: Object) {

    // gibt eine Decorator-Funktion zurück:
    return function (target: any) {

```

```

        console.log("Eigentlicher Decorator:", confObj);
    }
}

```

...was auf die Klasse Person wie folgt angewendet werden könnte:

```

@classDecoratorFactory({})
class Person {
    // ...
}

```

Der letztlich verwendete Decorator hat (über eine Closure) Zugang zum übergebenen Argument und kann es verwenden.

- ✓ Man kann beispielsweise als Objekt eine **Propertyliste** übergeben, die dem Prototyp-Objekt der Klasse hinzugefügt werden soll.

Dafür müsste die Factory etwa so aufgebaut sein (wir verwenden `Object.assign()`):

```

function classDecoratorFactory(propsObj: Object) {

    // gibt eine Decorator-Function zurück:
    return function (target: any) {
        console.log("Eigentlicher Decorator:", propsObj);

        // Mixin des propsObj in Prototype der Klasse:
        Object.assign(target.prototype, propsObj);
    }
}

```

Achtung, `Object.assign` lässt sich nur anwenden, wenn das Zielformat ES2015 (ECMA6) ist!

Nun wird beim Dekorieren der Klasse ein Konfigurationsobjekt übergeben:

```

@classDecoratorFactory({
    haustier:"Dackel",
    auto:"BMW",
    fahren:function(){
        console.log(this.vorname, "fährt", this.auto);
    }
})

```

```
class Person {
    public vorname: string;
    public nachname: string;

    constructor(vorname : string, nachname : string) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
}
```

Die dekorierte Klasse besitzt nun eine Methode `fahren()` und ein `haustier`-, sowie ein `auto`-Property.

```
var peter = new Person("Peter", "Panter");

peter.fahren();
// -> Peter fährt BMW
console.log(peter.haustier);
// -> Dackel
```

## 14. Literatur

---

- **Learning ECMAScript 6**  
Narayan Prusty  
(Packt, 2015)
- **Learning TypeScript**  
Remo H. Jansen  
(Packt, 2015)
- **TypeScript Blueprints**  
Ivo Gabe de Wolff  
(Packt, 2016)

„All-Time Favorites“:

- **JavaScript**. Das umfassende Referenzwerk, 6. Aufl.  
David Flanagan  
(O'Reilly, 2012)

- **JavaScript: The Good Parts**  
Douglas Crockford  
(O'Reilly, 2008)

## 15. Onlinere Ressourcen

---

### 15.1. Links zu JavaScript

---

#### **JavaScript Dokumentation (engl.):**

- [developer.mozilla.org/en/docs/JavaScript](https://developer.mozilla.org/en/docs/JavaScript)

#### **ECMA Spezifikation (engl.):**

- [www.ecma-international.org](http://www.ecma-international.org)

### 15.2. Links zu TypeScript

---

#### **TypeScript-Website:**

- [www.typescriptlang.org/](http://www.typescriptlang.org/)

#### **Dokumentation** mit Tutorials und Handbuch (engl.):

- [www.typescriptlang.org/docs/tutorial.html](http://www.typescriptlang.org/docs/tutorial.html)

#### **Playground** zum Ausprobieren:

- [www.typescriptlang.org/play/index.html](http://www.typescriptlang.org/play/index.html)

#### Typescript bei **Stackoverflow**:

- <http://stackoverflow.com/questions/tagged/typescript>