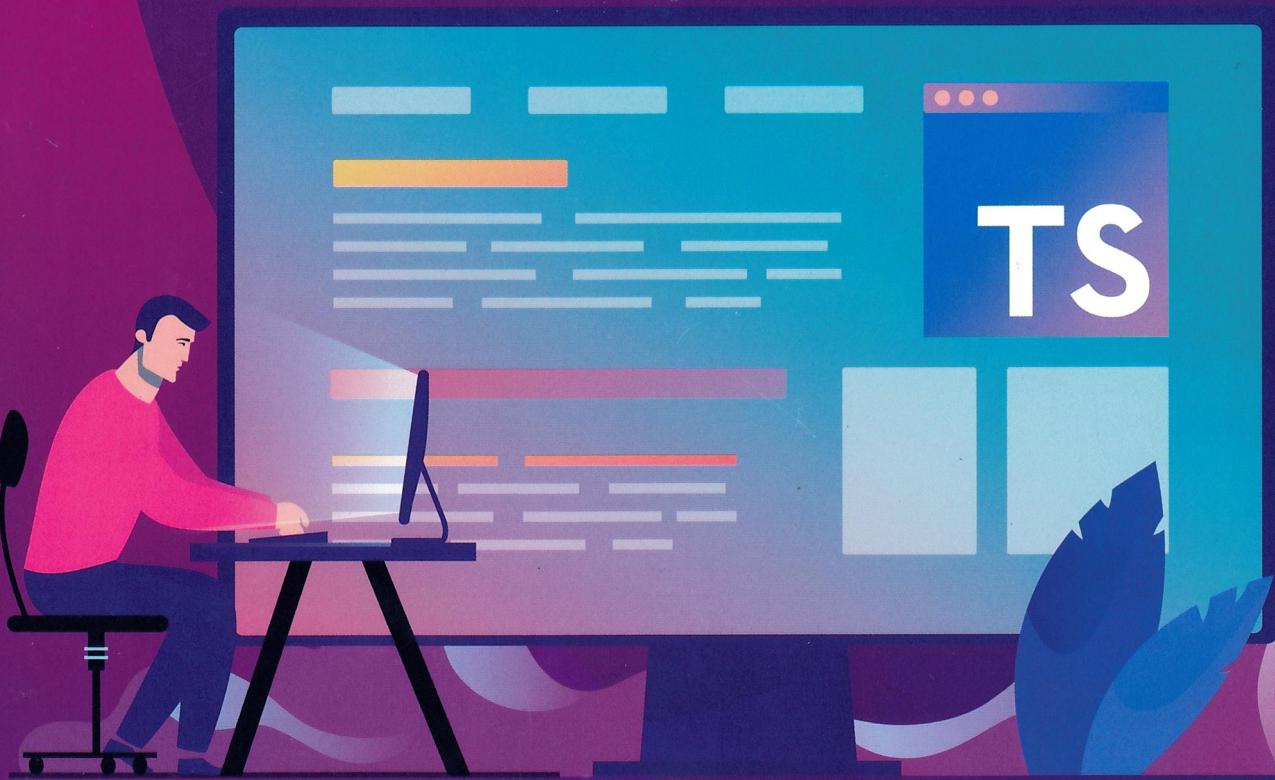


# web & mobile **DEVELOPER**

[webundmobile.de](http://webundmobile.de)

## TypeScript 3.6 im Detail

Microsofts neueste JavaScript-Version beseitigt zahlreiche  
Probleme, vor die die Standard-Sprachversion Entwickler stellt S. 28



### Next.js Version 9

Jochen Schmidt stellt die neueste Version des  
React-Frameworks im Detail vor S. 118

Ausgabe **11/19**

Deutschland CH: 29,90 CHF  
A, B, NL, L: 16,45 EUR



4 198255 014951 11

## TYPESCRIPT VERSION 3.6 IM DETAIL

# TypeScript reloaded

Der offene Sprachstandard von JavaScript sorgt für Probleme, weil Compiler viele Probleme nicht erkennen können. TypeScript Version 3.6 schafft Abhilfe.

**A**ls die in Verona ansässige GRUSP vor einigen Jahren das Kongressmotto »JavaScript Everywhere« ausgaben, grinsten viele über diese Aussage. In der Praxis hatte der Kongressveranstalter allerdings Recht. Die Kriege zwischen Opera, Mozilla und Microsoft hatten dazu geführt, dass JavaScript-VMs mit immer höherer Performance aufwarten konnten.

Ob der immensen Verbreitung von Browsern entstand auf diese Art und Weise eine quasi universelle Ausführungsumgebung, die durch Node.js eine serverseitige Komponente erhielt. Die immer größere Verbreitung sorgte allerdings auch für Probleme, da die Einführung neuer Syntaxfeatures immer schwieriger wurde.

Jeremy Ashkenas kam bei seinem gegenüber anderen JavaScript-Sprachen mittlerweile ins Hintertreffen geratenen CoffeeScript-Projekt auf die in Bild 1 gezeigte Struktur.

Der Rest ist im Großen und Ganzen Geschichte. Deshalb wollen wir die Gelegenheit nutzen, einen Blick auf die neueste Version von Microsofts JavaScript-Derivat zu werfen. Auch wenn der Fokus auf der Vorstellung neuer Sprachfeatures seit TypeScript 3.4 liegt, sollen auch Quereinsteiger nicht aussteigen. Microsoft richtet die Sprach-Syntax konsequent an C# und Java aus. Das Lesen der hier gezeigten Codebeispiele dürfte ausreichen, um Ihnen ein solides Grundverständnis von TypeScript zu verschaffen.

## Eine Frage der Arbeitsumgebung

Da TypeScript ein mehr oder weniger von Microsoft verwaltetes Projekt ist, möchten in den folgenden Schritten - ganz konventionell - auf Visual Studio 2017 setzen. Achten Sie im Installer darauf, die für die Web-Entwicklung notwendige Payload zu markieren (Bild 2). Leider ist Microsoft bei der Aktualisierung der Visual Studio-Paketquellen alles andere als

schnell. Wer nach der Selektion der Checkbox in das Einzelne Komponenten-Tab wechselt, sieht die in Bild 3 gezeigte Szenerie.

Zur Lösung des Problems besuchen wir die URL <https://docs.microsoft.com/en-us/visualstudio/javascript/javascript-in-vs-2019?view=vs-2019>, in der Microsoft über die verschiedenen Arten der TypeScript-Installation informiert. Im Interesse der Bequemlichkeit wollten wir hier auf den Visual Studio-Marketplace setzen, um uns das Hantieren mit NuGet- oder NPM-Paketen zu ersparen.

Besuchen Sie im nächsten Schritt die URL <https://marketplace.visualstudio.com/items?itemName=TypeScriptTeam.typescript-36rc>, um die Webseite des Pakets TypeScript 3.6 RC for Visual Studio zu öffnen. Klicken Sie dort auf den grünen Download-Button, und installieren Sie das rund 5 MByte große SDK wie gewohnt auf ihrer Workstation. TypeScript 3.6 liegt zum Zeitpunkt der Drucklegung dieses Artikels als Re-

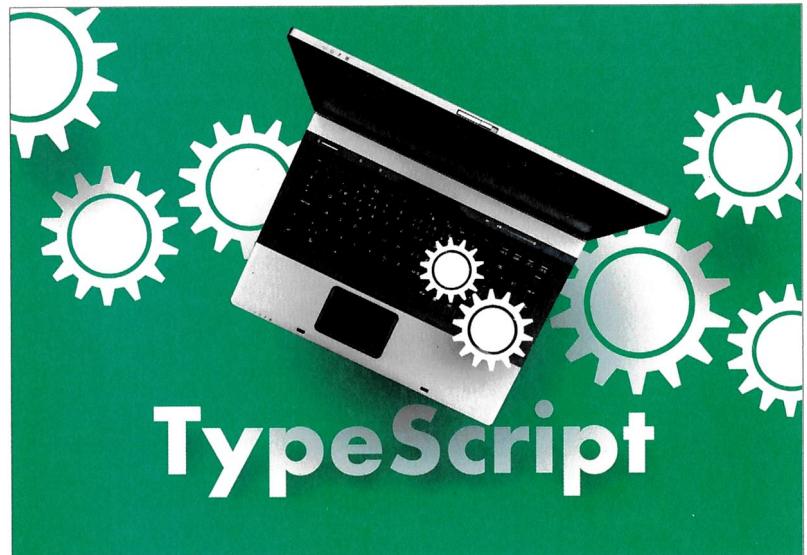
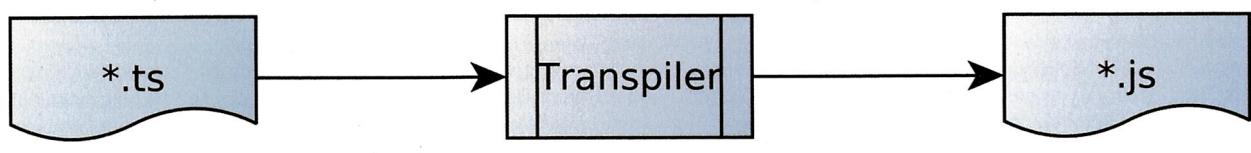
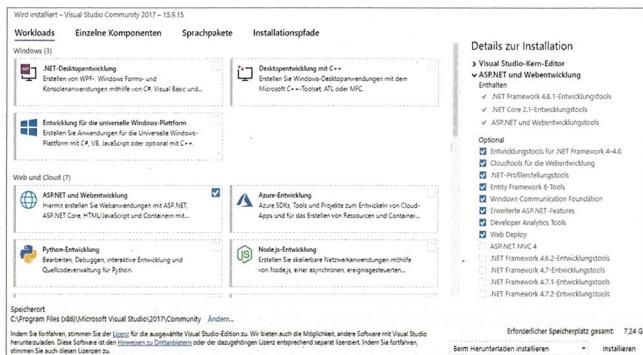


Foto: Shutterstock / GagoDesign



Ein Transpiler verwandelt beliebige Sprachen in Javascript (Bild 1)



Diese Payload stellt die für die Web-Entwicklung benötigten Komponenten bereit (Bild 2)

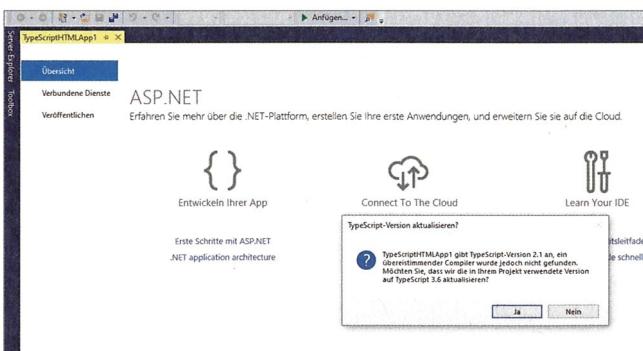
lease Candidate vor - nach dem Erscheinen dürfte das SDK seinen Namen ändern. Der in Visual Studio 2015 enthaltene Template-Typ für HTML-Applikationen steht sowohl in der Version 2017 als auch in der Version 2019 nicht mehr zur Verfügung. Als Abhilfe müssen wir abermals in den Visual Studio-Marketplace zurückkehren, wo wir die URL <https://marketplace.visualstudio.com/items?itemName=Rich-Newman.TypeScriptHTMLApplicationTemplate> aufrufen.

Danach steht ein als HTML Application with TypeScript bezeichnetes Paket zur Verfügung, auf dessen Basis sie ein neues Projekt mit mehr oder weniger beliebigen Dateinamen anlegen. Achten Sie darauf, dass manche Konfigurationen von Visual Studio an dieser Stelle auch die Generierung von Node.js-Applikationen erlauben.

Im Rahmen der erstmaligen Erzeugung einer Vorlage blendet Visual Studio die in Bild 4 gezeigte Warnung ein. Die im Template verwendete Version von TypeScript ist etwas veraltet. Klicken Sie an dieser Stelle auf Ja, um die Aktualisierung abzuschließen.

## Analyse der Projektstruktur

Nach dem Abnicken der Meldung und dem erfolgreichen Start von Visual Studio sehen Sie im Play-Bereich eine Ausführungsconfiguration, die den Aufruf von IIS Express erlaubt. Klicken Sie diesen an, um in der Ausgabe die Abarbeitung eines Kompilationsprozesses zu beobachten. Danach startet der Internet Explorer und ruft eine `localhost`-URL mit



Die Pflege des Templates erfolgt durch einen – manchmal etwas langsamen – Drittentwickler (Bild 4)



Das in Visual Studio integrierte Paket bringt eine stark veraltete TypeScript-Runtime mit (Bild 3)

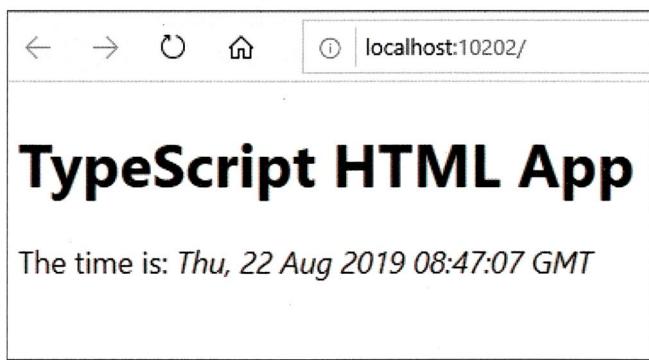
einem mehr oder weniger beliebigen Port auf (Bild 5). Sofern Sie dort die Lokalzeit ihres Systems sehen, ist die Toolchain in bester Ordnung.

Das aus der Vorlage entstandene Projekt besteht neben einer gewöhnlichen CSS-Datei auch aus einem `.html`-File und einer Datei mit der Endung `.ts`. Diese steht, wenig überraschend, für TypeScript.

Wir wollen unsere Betrachtungen hier aber mit der HTML-Datei beginnen, die einige Ressourcen lädt:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" />
<title>TypeScript HTML App</title>
<link rel="stylesheet" href="app.css" type="text/css" />
<script src="app.js"></script>
</head>
```

Besonders interessant ist hier die Inklusion der `.js`-Datei, die eine gewöhnliche JavaScript-Endung aufweist. Der TypeScript-Transpiler durchsucht in der vorliegenden Konfiguration unser gesamtes Projekt, um alle `.ts`-Dateien durch ihre transpilierten Versionen zu ersetzen. In unserem Fall laden wir den Code, der in der ebenfalls vom Projekt angelegten `.ts`-Datei unterkommt.



**Lokalzeit des Systems:** Scheint die Uhrzeit am Bildschirm auf, so ist alles gut (Bild 5)

Für die eigentliche Ausgabe ist dann ein `<div>` erforderlich, dass unser Code zur Laufzeit mit Informationen bevölkern wird:

```
<body>
<h1>TypeScript HTML App</h1>
<div id="content"></div>
</body>
</html>
```

Der Code beginnt mit der Deklaration einer Klasse. Eines der wichtigsten Design-Ziele von TypeScript war, Microsofts .net-Entwickler so gut wie möglich vom - zugegebenermaßen etwas wilden - JavaScript-Prototypen-Objektsystem zu isolieren. Aus diesem Grund ist die Klassensyntax so stark wie möglich an C# und Java angelegt, was im Fall unserer vorliegenden Klasse durch die Verwendung von Member-Variablen Ausdruck findet:

```
class Greeter {
  element: HTMLElement;
  span: HTMLElement;
  timerToken: number;
```

Neben der für erfahrene Programmierer selbsterklärenden Deklaration der Klasse findet sich hier ein wichtiges Syntax-Element. Der Doppelpunkt steht in TypeScript für die Zuweisung eines Datentyps. Fehlt er, so ist eine Variable gewöhnlich und wird vom Compiler nicht überprüft.

Im Bereich der primitiven Datentypen verhält sich TypeScript so, wie es der zugrunde liegende JavaScript-Interpreter vorschreibt. Das einzige Ärgernis für von C und Co. umsteigende Entwickler ist, dass es nur einen Zahlentyp gibt.

Es ist in JavaScript nicht möglich, natürliche und mit dezentalen Teilen ausgestattete Zahlen auf verschiedene Arten anzusprechen. TypeScript kann dieses fehlende Feature naturgemäß nicht nachrüsten, weil die Engine keinen neuen Datentyp einführen kann.

Als kleinen Ersatz dafür unterstützt TypeScript allerdings verschiedene Arten zum Anschreiben numerischer Konstanten:

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

Damit wollen wir allerdings wieder zu unserem Klassenelement zurückkehren. Konstruktoren von Klassen entstehen in TypeScript prinzipiell über das Schlüsselwort `constructor`. Im Fall unseres Projekts sieht der Code folgendermaßen aus:

```
constructor(element: HTMLElement)
{
  this.element = element;
  this.element.innerHTML += "The time is: ";
  this.span = document.createElement('span');
  this.element.appendChild(this.span);
```

```
  this.span.innerText = new Date().toUTCString();
}
```

Auch hier findet sich keine Wissenschaft. Wir beschaffen uns im ersten Schritt einen Verweis auf das Steuerelement und schreiben diesem danach Informationen ein. Das `this`-Schlüsselwort erlaubt im Code die Reflektion gegen sich selbst und die Attribute der Klasse anzusprechen.

TypeScript 3.6 führt im Bereich der Konstruktoren eine Besonderheit ein. Wer den Namen einer Funktion als String anliefert und als String `constructor` vergibt, erstellt nun – analog zu ECMAScript – ebenfalls eine Konstruktorfunktion:

```
class C {
  "constructor"() {
    console.log("I am the constructor now.");
  }
}
```

Die einzige Ausnahme von dieser Regel sind Member-Variablen, deren Namen durch eine Berechnung entsteht. Ein (zugegebenermaßen seltenes) Beispiel wäre die Verwendung des `[ ]`-Operators nach dem folgenden Schema:

```
class D {
  ["constructor"]() {
    console.log("I'm not a constructor - just a
plain method!");
  }
}
```

Microsoft löst die Deklaration von Member-Funktionen unterschwellig. Sie bestehen aus dem Funktionsnamen, einer eventuellen Parameterliste und dem in geschwungene Klammern zu setzen Code. Im Fall unseres Programms finden sich derer zwei, die für das Starten und das Anhalten des Programms verantwortlich sind:

```
start() {
  this.timerToken = setInterval(() =>
  this.span.innerHTML = new Date().toUTCString(), 500);
}

stop() {
  clearTimeout(this.timerToken);
}
```

Von C umsteigende Entwickler suchen verzweifelt nach dem Einsprungpunkt. Diesen gibt es in TypeScript-Applikationen nicht, der Sprachstandard sieht auch keine Funktion a la `main` vor. Stattdessen werden alle `.ts`-Dateien, die zur Laufzeit ja `.js`-Dateien sind, im Rahmen des Ladens wie gewohnt geparsst. Im öffentlichen Bereich liegender Code gelangt dabei zur Ausführung.

Im Fall unseres Programms erfolgt der Start, in dem `window.onload` ein Delegat eingeschrieben wird. Die JavaScript-

Runtime ruft diese Methode auf, wenn der Aufbau des DOM-Modells abgeschlossen ist:

```
window.onload = () => {
    var el = document.getElementById('content');
    var greeter = new Greeter(el);
    greeter.start();
};
```

Innerhalb des Delegaten erzeugen wir im ersten Schritt eine Instanz unserer Klasse, um sie danach durch Aufrufen ihrer start-Methode zur Timer-Registrierung zu animieren. Damit ist dieses Beispiel auch schon durchgesprochen.

## Experimente mit dem Typsystem

Die in der Einleitung gemachte Feststellung mit dem offenen Sprachstandard basiert darauf, dass während der Kompilation von normalem JavaScript nicht festgestellt werden kann, ob ein Methodenaufruf mit einem für diese Methode bekannten Objekt erfolgt.

Es gibt am Markt mittlerweile zwar statische Analysewerkzeuge; bei deren Verwendung besteht allerdings das Problem, dass die Entwickler aufgrund der zu vielen Warnungen irgendwann nicht mehr auf diese reagieren.

Zur Demonstration der Typisierungsmöglichkeiten von TypeScript wollen wir uns im nächsten Schritt eine Klasse anlegen, die eine Art TypeScript-Struct darstellt:

```
class Person {
    name: string;
    age: number;
    location: string;
}
```

Unsere Klasse besteht aus drei Member-Variablen, die Informationen über ein lebendes oder totes Individuum aufnehmen können. Ansonsten verhält sich *Person* nach dieser Deklaration wie ein beliebiger anderer Datentyp. Wir könnten die *Greeter*-Klasse also nach dem folgenden Schema erweitern, um die Verarbeitung von *Person*-Instanzen zu ermöglichen:

```
class Greeter {
    ...
processPerson(aPerson) {
    console.log("Person received!" + aPerson)
```

### Entwicklung von UWP-Applikationen

TypeScript lässt sich zur Entwicklung von UWP-Applikationen verwenden. Beachten Sie allerdings, dass *.jsproj*-Dateien mit Visual Studio 2019 nicht mehr geöffnet werden können. Als Workaround bietet sich die Verwendung von Visual Studio 2017 an, das zum Zeitpunkt der Drucklegung von Microsoft ebenfalls am aktuellen Stand gehalten wird.

```
}
```

TypeScript-Objekte zerfallen während der Ausführung zu einem gewöhnlichen JavaScript-Objekt. Ruft TypeScript-Code eine gewöhnliche JavaScript-Funktion auf, so deaktiviert der Transpiler die Typüberprüfung. Die aus der gewöhnlichen Webentwicklung bekannte Funktion *console.log* würde also statt einer *Person*-Instanz eine gewöhnliche Instanz von *Object* bekommen.

Zur Überprüfung des korrekten Programmverhaltens adaptieren wir den an *window.onload* übergebenen Handler:

```
window.onload = () => {
    ...
    greeter.start();
    let myPerson = new Person();
    greeter.processPerson(myPerson);
};
```

Das korrekte Funktionieren unseres Programms lässt sich unter anderem dadurch überprüfen, dass sie den Code starten und danach in die Konsole des Browsers wechseln. Wir sehen dort die Ausgabe *Person received![object Object]*, die die weiter oben angelegte Hypothese bezüglich des Objektseins von TypeScript-Klassen belegt

Leider kommt der Parameter der Methode *processPerson* derzeit noch ohne Typqualifikation aus, weshalb TypeScript keine Überprüfung vornehmen kann. Sie könnten in *window.onload* also auch einen String oder eine Zahl anliefern. Zur Aktivierung der Prüfung während der Transpilation müssen wir den Korpus der Methode im ersten Schritt um eine Deklaration ergänzen. Dabei setzen wir auf den von weiter oben bekannten Doppelpunkt-Operator:

```
processPerson(aPerson:Person) {
    console.log("Person received!" + aPerson)
}
```

Da wir derzeit im Rahmen der Initialisierung ja eine *Person*-Klasse übergeben, funktioniert vorerst weiter alles wie gewohnt. Ein interessanter Aspekt von JavaScript ist das Duck Typing. Dahinter steht der Gedanke, dass das englische Sprichwort von der Ähnlichkeit der Ente auch auf die Typenzuweisung anzuwenden ist.

Als nächste Aufgabe wollen wir überprüfen, ob TypeScript Duck Typing unterstützt. Hierzu erzeugen wir eine neue Klasse namens *SecondPerson*, die mit unserer schon angelegten Personen-Klasse im Bezug auf die Attribute identisch ist. Im nächsten Schritt übergeben wir statt einer Instanz der Person eine Instanz der neuen Klasse im Rahmen der Initialisierung des Projektskeletts:

```
class SecondPerson {
    name: string;
    age: number;
    location: string;
```

```

33 class SecondPerson {
34   name: string;
35
36   location: string;
37
38 }
39
40 type AgelessPerson = Omit<Person, "age">;
41
42
43
44
45 window.onload = () => {
46   var el = document.getElementById('content');
47   var greeter = new Greeter(el);
48   greeter.start();
49   let myPerson = new SecondPerson();
50   greeter.processPerson(myPerson);
51 };

```

Fehlerliste...      Gesamte Projektmappe      2 Fehler      1 Warnung      0 Mitteilungen      Erstellen + IntelliSense

Projekt	Datei	Ze...	Unterdrückun...
TypeScriptHTMLApp1	app.ts	50	Aktiv
TypeScriptHTMLApp1	app.ts	50	
TypeScriptHTMLApp1			

**Duck Typing:** auch in TypeScript mit von der Partie (Bild 6)

```

}
window.onload = () => {
  ...
  let myPerson = new SecondPerson();
  greeter.processPerson(myPerson);
};

```

Dieses Programm lässt sich ebenfalls problemlos ausführen, womit die Korrektheit der Annahme bewiesen ist. Zur Provokation eines Compilerfehlers editieren wir die Klasse nun nach dem folgenden Schema:

```

class SecondPerson {
  name: string;

  location: string;
}

```

Die neue Version unterscheidet sich von ihrer Vorgängerin dadurch, dass der Parameter `age` nun nicht mehr vorhanden ist. Eine Kompilation des Programms würde nun mit dem in Bild 6 gezeigten Fehler scheitern. Wer die englischsprachige Fehlermeldung näher ansieht, stellt fest, dass der TypeScript-Transpiler das Fehlen des soeben entfernten Attributs moniert.

### Vom adaptierten Typen

Unsere soeben durchgeführten Experimente mit dem Klassensystem waren kein Selbstzweck. Es ist offensichtlich, dass der TypeScript-Transpiler vergleichsweise tief gehende Überprüfungen und Reflektionen der Klassen durchführt. Microsoft nutzt dies zur Implementierung von Utility Types - ei-

nem vergleichsweise seltsamen syntaktischen Zucker, dessen Verhalten in Bild 7 gezeigt ist.

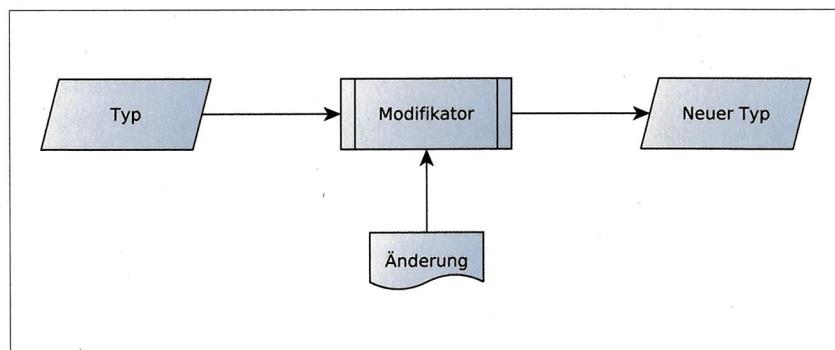
Die dahinter stehende Idee ist, dass ein Utility Type zwischen einem schon vorhandenen Typ und dem globalen Typespace lebt. Er führt Veränderungen am schon vorhandenen Typ durch, um das Ergebnis als neuen Typ zu exponieren.

Zur Illustration dieses Design Patterns wollen wir auf eines der in TypeScript 3.5 neu eingesetzten Kommandos setzen. Zur Nutzung von `Omit` ist folgende Deklaration erforderlich. Achten Sie darauf, die vorher angelegte Definition der Klasse zu ersetzen, bevor Sie die Datei abermals speichern:

```
class SecondPerson = Omit<Person, "age">;
```

`Omit` übernimmt - so zumindest die Theorie - einen Parameter und einen String, der den Namen des zu eliminierenden Member an liefert. Wir sollten hier also eine neue Klasse namens `SecondPerson` erhalten, der das `Age`-Attribut fehlt.

Leider ist dem in der Praxis nicht so. Wer den Code speichert und ausführt, bekommt stattdessen die Fehlermeldung `Omit bezieht sich nur auf einen Typ, wird aber hier als Wert`



**Utility Types** verändern schon vorhandene Typen (Bild 7)

verwendet. Ursache für dieses auf den ersten Blick unbefriedigende Verhalten ist, dass die meisten Utility Types nur mit Interfaces beziehungsweise Typen, nicht aber mit gewöhnlichen Klassen arbeiten. Anstatt wie bisher eine Klasse zu erzeugen, erzeugen wir nun ein Interface:

```
interface Person {
  name: string;
  age: number;
  location: string;
}
type SecondPerson = Omit<Person, "age">;
```

Interfaces verhalten sich in TypeScript genauso, wie sie es von Java oder C# erwarten würden. Die Nutzung ist nicht nur aus Gründen der Kommunität interessant. Mehrfachvererbung wirft in TypeScript interessante Fragen auf, die im Thread unter <https://stackoverflow.com/questions/40807808/typescript-multiple-inheritance> besprochen werden.

Die für die Ausgabe verantwortliche Funktion der Klasse Greeter wundert sich über diese Änderung nicht weiter. Ein Interface ist als Parameter einer Funktion genauso erlaubt wie eine Klasse. Legt man das Interface als Parametertyp fest, so müssen alle übergebenen Klasseninstanzen beziehungsweise Objekte das geforderte Interface implementieren.

Problematischer ist der Konstruktor, da sich ein Interface nicht unter Nutzung des `new`-Operators initialisieren lässt. Die Belebung erfolgt stattdessen in einer Klasse, die über das `implements`-Statement ihre Beziehung zum Interface benennt:

```
class PersonImpl implements SecondPerson {
  name: string;
  location: string;
}
```

Analog zu Java und C# gilt auch in TypeScript, dass die Verwendung des `implements`-Keywords nur für das Anzeigen der Beziehung zwischen Klasse und Interface verantwortlich ist. Die eigentliche Realisierung der geforderten Variablen

und/oder Methoden bleibt Verantwortung des Entwicklers. Wichtig ist an dieser Stelle, dass die in Visual Studio permanent laufende Überprüfung beim Erkennen von korrekten Implementierungen Zusatzzeit braucht. Wundern Sie sich also nicht, wenn die Klasse oder das `implements`-Statement nach der Eingabe noch für einige Sekunden rot unterlegt erscheinen. Im nächsten Schritt können wir `onload` anpassen:

```
window.onload = () => {
  ...
  let myPerson = new PersonImpl();
  greeter.processPerson(myPerson);
};
```

Der letzte Akt besteht dann darin, in `processPerson` die Verwendung des Interfaces zu befehligen:

```
processPerson(aPerson:SecondPerson) {
  console.log("Person received!" + aPerson)
}
```

Das korrekte Funktionieren des `omit`-Befehls lässt sich dadurch überprüfen, dass sie in `processPerson` versuchen, auf das `age`-Attribut des Parameters zurückzugreifen. Der Transpiler wird in diesem Zugriff nicht erlauben, weil das betreffende Attribut ja durch den Aufruf von `omit` eliminiert wurde.

Damit ist dabei nur eines von vielen möglichen Beispielen, die zwei Parameter aufnehmen können. Die unter <https://www.typescriptlang.org/docs/handbook/utility-types.html> bereitstehende Liste aller Utility-Typen listet die in [Tabelle 1](#) genannten zweiwertigen Types auf.

## Exkurs: Spiele mit Interfaces

An dieser Stelle wollen wir die Besprechung von Utility Types unterbrechen, um uns einer kleinen Besonderheit zuzuwenden. Das normalerweise für numerische Oder-Verknüpfungen verantwortliche Symbol lässt sich in TypeScript auch auf Typen anwenden und verhält sich dann im Großen und Ganzen analog.

Zum Verständnis der Motivation für das Problem wollen wir uns die folgende Methode ansehen, die die Ausrichtung von angelieferten Informationen in Abhängigkeit des Datentyps des zweiten Parameters verändert:

```
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error('Expected string or number,
  got '${padding}'.');
}
```

So lange die Funktion nur von kooperativen Entwicklern verwendet wird, ist alles in Ordnung. Murphys Gesetz be- ►

Tabelle 1: Zweiwertige Types

Typ	Kurzbeschreibung
Exclude<T,U>	Entfernt alle Elemente aus T, die auch in U vorhanden sind, und liefert das Resultat zurück
Extract<T,U>	Liefert einen Typen zurück, der alle sowohl in T als auch in U enthaltenen Elemente enthält
Omit<T,K>	Entfernt Attribut
Pick<T,K>	Erzeugt einen neuen Typen, der nur die in K angeführten Elemente aus T übernimmt
Record<K,T>	Kombiniert Typ K mit den in T angelieferten Werten, und liefert einen Datenbankeintrag zurück

sagt, dass über kurz oder lang eine Person findet, die statt eines Strings und einer Zahl beispielsweise ein Objekt anliefert. In diesem Fall würde die durch `throw` ausgelöste Fehlermeldung erst während der Programmausführung erscheinen, die in TypeScript während der Transpilation durchlaufenden Überprüfungen würden den Fehler nicht finden.

Unter Verwendung von Union Types könnte sich die Methode folgendermaßen präsentieren:

```
function padLeft(value: string, padding: string | number) {
    if (typeof padding === "number") {
        return Array(padding + 1).join(" ") + value;
    }
    if (typeof padding === "string") {
        return padding + value;
    }
}
```

In der neuen Version der Funktion darf die Variable nur noch den Typ String oder den Typ Number annehmen. Das Übergeben eines anderen Werts würde nun zu einem Fehler führen.

Neben dieser Stabilität erhöhenden Anwendung findet man Union Types auch zur Parametrisierung von Utilities. Ein Beispiel dafür ist die folgende Typdeklaration, die einen dreiwertigen Union Type in einen einwertigen Typ verwandelt:

```
type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"
```

Neben den mit einem zusätzlichen Parameter ausgestatteten Utility Types kennt TypeScript auch Hilfsmethoden, die eine von den Sprachentwicklern festgelegte Operation gegen den angelieferten Typ durchführen. Zur Demonstration der Möglichkeiten möchten wir uns das folgende Interface ansehen, das von Haus aus zwei Werte mitbringt:

```
interface Todo {
    stringA: string;
    stringB: string;
}
```

Sofern sich alle Entwickler über die damit einhergehenden Risiken im Klaren sind, ist es möglich beziehungsweise wünschenswert, ein Interface nicht immer komplett implementieren zu müssen. Um die Simulation der Situation realistisch zu

gestalten, wollen wir die folgende Funktion als Arbeiter annehmen:

```
processTodo(aField: Todo)
{
    console.log(aField);
}
```

TypeScript ermöglicht Entwicklern das schnelle Beleben von Interfaces, ohne dabei auf eine Klassendeklaration zurückgreifen zu müssen. Stattdessen nutzt man geschwungene Klammern, was zu folgenden Snippet führt:

```
window.onload = () => {
    var el = document.getElementById('content');
    var greeter = new Greeter(el);
    greeter.start();

    const todo1 = {
        stringA: 'organize desk',
        stringB: 'clear clutter',
    };
    greeter.processTodo(todo1);
};
```

Da die Variable beide Strings mitbringt, funktioniert die Arbeitung ohne Probleme. Dies lässt sich dadurch überprüfen, dass sie unser Programm im Edge-Browser laden und die Ausgabe in der Konsole überprüfen.

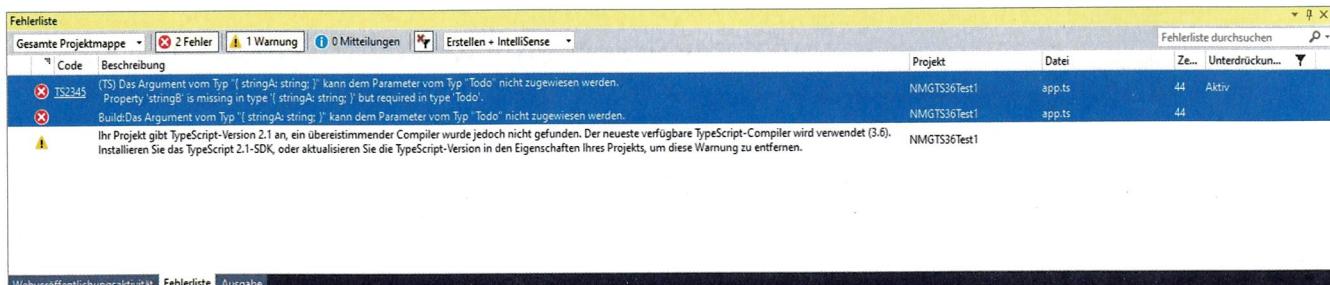
## Interface adaptieren

Als nächsten Versuch adaptieren wir die Deklaration der Variable, um nur noch einen der beiden Strings anzuliefern:

```
const todo1 = {
    stringA: 'organize desk'
};
greeter.processTodo(todo1);
```

Wer das Programm abermals zu Kompilation freigibt, bekommt nun den in Bild 8 gezeigten Fehler. Der TypeScript-Transpiler stellt fest, dass das Objekt die Bedingungen der Funktion nicht mehr erfüllt.

In der Theorie könnte man an dieser Stelle das Interface adaptieren, um die nicht unbedingt erforderlichen Elemente als



Die Kompilation scheitert aufgrund des fehlenden Members (Bild 8)

optional zu markieren. Hierzu reicht es aus, ein Fragezeichen vor die Variablen beziehungsweise Attribute zu stellen:

```
interface Props {
  a?: number;
  b?: string;
};
```

Die Verwendung der weiter oben erstmals vorgestellten Typen ermöglicht uns eine niedrigschwelligere Lösung, die weniger Veränderungen in der gesamten Programmstruktur verursacht.

Kehren Sie in die Deklaration der Klasse Greeter zurück und adaptieren Sie den an die Funktion übergebenen Parameter nach dem folgenden Schema:

```
class Greeter {
  element: HTMLElement;
  ...
  processTodo(aField: Partial<Todo>)
  {
    console.log(aField);
  }
}
```

Mit dem Utility Type Partial ausgestattete Parameter beziehungsweise Objekte werden von der TypeScript-Runtime als komplett optional betrachtet. Für Aufrufer bedeutet dies, dass sie an dieser Stelle mehr oder weniger beliebige Ele-

### GitHub ist wichtig

TypeScript-Entwickler müssen sich – wohl oder übel – mit der GitHub-Issue-Oberfläche anfreunden. Microsoft nutzt die im Rahmen der Implementierung abgehaltenen Diskussionen als Erweiterung der Dokumentation. In vielen Fällen finden sich, insbesondere kurz nach dem Erscheinen eines neuen Features, dort nicht in der Dokumentation besprochene Spezialaspekte.

mente übergeben dürfen. Die Verantwortung für die Überprüfung des Vorhandenseins fällt einzig und allein in den Aufgabenbereich des Entwicklers. Würde die Methode fortgeschrittene Operationen durchführen und dabei über Nullwerte stolpern, so käme es zur Laufzeit zu Fehlern. Aus diesem Grund sollten Sie die Verwendung von Partial gut überlegen - analog zu Goto ist auch dies ein Werkzeug, dass man mit Vorsicht einsetzen muss. Auch hier gilt, dass Partial nur ein Beispiel für die Möglichkeiten ist. **Tabelle 2** stellt die zum Zeitpunkt von TypeScript 3.6 verfügbaren Types vor.

### Steigerung der algorithmischen Stabilität

Über die beste Art, wie man in einer komplexen Applikation globale oder konstante Werte vorhält, lässt sich hervorragend streiten. Programmiersprachen wie Java oder C bieten seit langer Zeit die Möglichkeit an, Variablen als Konstante zu deklarieren. Diese lassen sich dann - je nach Programmierumgebung - nicht oder nur mit komplizierter Pointerarithmetik verändern. TypeScript 3.4 verbesserte das im Sprachstandard seit längerer Zeit vorhandene *readonly*-Attribut, das sich nun auf die folgenden Arten anwenden lässt:

```
function foo(arr: ReadonlyArray<string>) {
  arr.slice();           // okay
  arr.push("hello!");   // error!
}

function foo(arr: readonly string[]) {
  arr.slice();           // okay
  arr.push("hello!");   // error!
}

function foo(pair: readonly [string, string]) {
  console.log(pair[0]); // okay
  pair[1] = "hello!";   // error
}
```

Beachten Sie bei der Verwendung des Operators, dass er sich nur auf Felder und Tupel anwenden lässt. In der Dokumentation führt Microsoft die folgenden beiden Beispiele an, die *readonly* auf beliebige Klassen anzuwenden suchen und zum Zeitpunkt der Drucklegung nicht gültigen Code darstellen:

```
let err1: readonly Set<number>; // error!
let err2: readonly Array<boolean>; // error!
```

Weitere Informationen zu den Erweiterungen des *readonly*-Operators finden sich in GitHub unter der URL <https://github.com/microsoft/TypeScript/pull/3433>

► Tabelle 2: Verfügbare Types

Typ	Kurzbeschreibung
InstanceType<T>	Liefert den Typ einer Generatorfunktion zurück. Primär zur internen Verwendung
NonNullable<T>	Reduziert einen Union Type um die Werte null und undefined
Partial<T>	Liefert einen Typ zurück, in dem alle Attribute optional sind
Readonly<T>	Liefert einen Typ zurück, in dem alle Attribute <code>ReadOnly</code> sind und dementsprechend nicht verändert werden dürfen
Required<T>	Liefert einen Typ zurück, in dem alle Attribute vorhanden sind. Auf Deklarationsebene mit <code>? as optional</code> bezeichnete Elemente werden ebenfalls verpflichtend
ReturnType<T>	Analysiert die übergebene Funktion, und liefert ihren Rückgabetypr zurück
ThisType<T>	Liefert den Typ eines Typs zurück. Primär zur internen Verwendung, setzt Anpassungen des Kompliationsprozesses voraus

[github.com/Microsoft/TypeScript/pull/29435](https://github.com/Microsoft/TypeScript/pull/29435). Die Erweiterungen des `readonly`-Operators erlauben Microsoft die Umsetzung von Konstanten, die weder Arrays noch andere komplexe Datentypen sind. Zum Zeitpunkt der Drucklegung gibt es zwei Syntaxvarianten. In allen TypeScript-Anwendungen gleichermaßen ist die folgende Methode gültig:

```
let x = "hello" as const;
let y = [10, 20] as const;
let z = { text: "hello" } as const;
```

Wer nicht mit `.tsx`-Dateien arbeitet, darf stattdessen auch auf eckige Klammern setzen:

```
let x = <const>"hello";
let y = <const>[10, 20];
let z = <const>{ text: "hello" };
```

In beiden Fällen deklarieren wir im ersten Schritt eine gewöhnliche Konstante, um danach ein konstantes Array und ein konstantes Objekt zu erzeugen. Zusammengestellte Objekte - in unserem Fall die Variablen Y und Z - bekommen bei Verwendung des `const`-Kennworts eine Gruppe konstanter Member zugewiesen.

Eine als `const` bezeichnete Variable oder Konstante darf zur Laufzeit nicht mehr verändert werden. Die offizielle Beschreibung des Sprachstandards lautet dabei übrigens *A `const` assertion can only be applied to a to a string, number, boolean, array, or object literal*. Auch hier gilt, dass unter <https://github.com/Microsoft/TypeScript/pull/29510> eine detaillierte Diskussion zur Implementierung des Features bereitsteht.

Insbesondere in wissenschaftlichen Anwendungen wünscht man sich immer wieder eine Möglichkeit, Wert-

Quellen ansprechbar zu machen. Ein klassisches Beispiel dafür wäre die Erzeugung einer Fibonacci-Reihe, die von einem Algorithmus abgeerntet und weiterverarbeitet wird. Was in der Welt der Design Patterns seit längerer Zeit als Generator beziehungsweise Factory bekannt ist, wurde mittlerweile als JavaScript-Funktion deklariert. TypeScript-Entwickler können dieses Feature benutzen, wenn ihr Projekt ES6 als Baseline verwendet.

Die Anpassung derartiger Einstellungen erfolgt normalerweise durch Bearbeitung der Datei `tsconfig.json`. Visual Studio erstellt die TypeScript-Konfigurationsdatei im Rahmen jedes Kompilationslaufs aus den in der IDE eingestellten Parametern. Klicken Sie deshalb auf den *Properties*-Eintrag im Projektmappen-Explorer, um das in Bild 9 gezeigte Einstellungsfenster auf den Bildschirm zu holen.

Passen Sie die ECMAScript-Version an und speichern Sie die Einstellungen. Je nach Lust und Laune ihrer Visual Basic Studio-Installation ist dann noch eine Rekomplilation erforderlich, um die Toolchain auf den aktuellsten Stand zu bringen. Auf der Kommandozeile arbeitende Entwickler können unter Verwendung von `--downlevelIteration` auch Generatoren unter ES3 und ES5 avisieren.

Generatoren sind in TypeScript Funktionen, die eine (idealerweise) niemals endende Quelle von Werten zurückliefern. Die Deklaration erfolgt wie eine gewöhnliche Funktion, nach dem `function`-Schlüsselwort findet sich der aus C als Pointersymbol bekannte Stern.

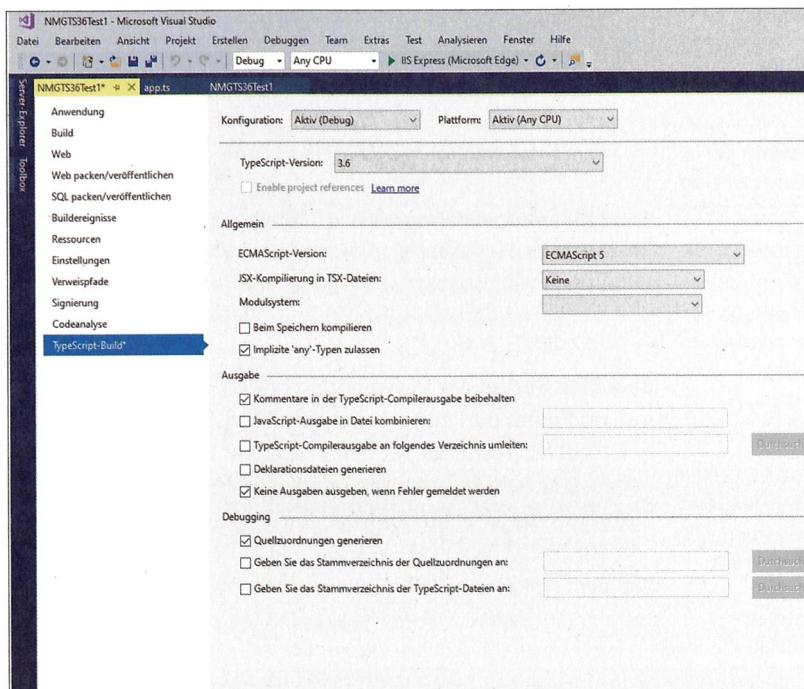
Für ein erstes kleines Beispiel möchte ich die folgende Sequenzfunktion realisieren, die ihren gehaltenen Zustand multipliziert und danach zurück liefert:

```
function* seqMalZwei() {
    var i = 1;
    while (true) {
        i = i * i;
        yield i;
    }
}
```

Von besonderer Bedeutung ist das `yield`-Kommando. Es weist die Runtime dazu an, denn als Parameter übergebenen Wert an den Aufrufer zurückzuliefern und die Ausführung der Methode danach zu pausieren. Der Ausführer hat seinerseits die Funktion `Next`, über die er die Runtime zur Fortsetzung der Funktion animiert. Sie rennt dann so lange weiter, bis sie abermals zum Retournieren eines Zustands aufgerufen wird.

Zur Illustration der Möglichkeiten benötigen Sie einen Button, mit dem sie unseren Generator anwerfen. Öffnen Sie hierzu die HTML-Datei, und fügen Sie den Knopf nach dem folgenden Schema hinzu:

```
<body>
<h1>TypeScript HTML App</h1><0x000A>
```



Visual Studio hilft bei der graphischen Bearbeitung der `tsconfig.json` (Bild 9)

```
<button onclick="worker();">Generate!
</button>
<div id="content"></div>
</body>
</html>
```

HTML-erfahrene Entwickler wundern sich, wo die Funktion `worker` am Ende unterkommt. Die Antwort darauf findet sich weiter oben. Eine TypeScript-Datei wird vom Transpiler in eine JavaScript-Datei umgewandelt. Enthält ein beliebiges `.ts`-File eine Methode, die nicht in einer Klasse ist, so wird diese nach der Transpilation zu einer gewöhnlichen JavaScript-Funktion im globalen Namespace. Als größeres Problem erweist sich die Frage, wo wir die Generator-Instanz zwischen den Aufrufen unterbringen. Das `window`-Objekt ist dafür geradezu ideal geeignet. Problematisch ist, dass TypeScript das Hinzufügen beliebige Attribute verbietet.

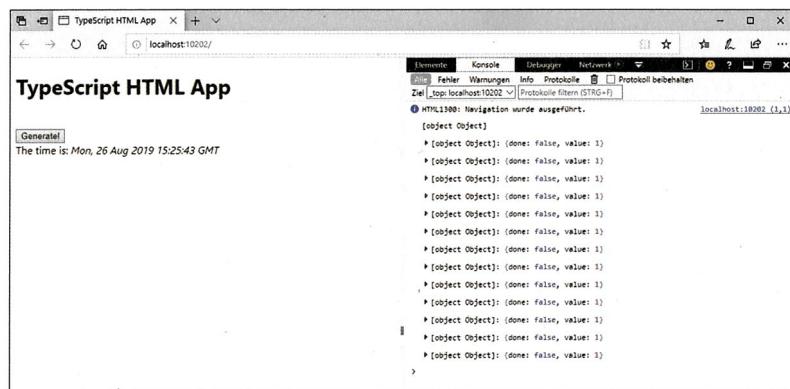
Als brutaler Workaround bietet sich die Verwendung eines Casts an. Wenn wir `window` im ersten Schritt in ein `any` umwandeln, informieren wir die TypeScript-Routine darüber, dass der vorliegende Wert als gewöhnliche JavaScript-Variablen zu behandeln ist. Danach lässt sich die Instanz unseres Generators unterbringen. Ihre Erzeugung erfolgt übrigens, indem wir die weiter oben angelegte Generator-Funktion wie eine gewöhnliche Methode aufrufen:

```
window.onload = () => {
  greeter.processTodo(todo1);
  (window as any).myIterator = seqMalZwei();
};
```

Damit fehlt nur noch die eigentliche Arbeiter-Methode. Sie beginnt mit der Wiederherstellung unserer Generator-Instanz, ruft danach die `Next`-Methode auf und gibt das Ergebnis in die Browserkonsole aus:

```
function worker() {
  let myIter = (window as any).myIterator;
  console.log(myIter.next());
}
```

An dieser Stelle ist unser Programm startbereit, Bild 10 zeigt das Resultat. Besonders interessant ist, dass TypeScript nicht



Unser Programm liefert permanent eins zurück (Bild 10)

nur den von `Yield` gelieferten Wert retourniert. Stattdessen finden wir auch ein Flag namens `done`, das uns darüber informiert, ob im Generator noch zusätzliche Werte verfügbar sind. Die Runtime bestimmt diese Informationen dadurch, dass sie prüft, ob die Ausführung der Methode schon beendet ist. Sobald die Generator-Funktion die Kontrolle an das Betriebssystem zurückgibt (also ausläuft), führen weitere Aufrufe der Methode `next` zu zur Rückgabe von `undefined`.

Unser Programm liefert permanent eins zurück, weil die Multiplikation von eins mit eins – naturgemäß – ebenfalls eins retourniert. Zur Behebung des Problems reicht es aus, den anfangs in `i` gespeicherten Wert anzupassen:

```
function* seqMalZwei() {
  var i = 2;
  while (true) {
    i = i * i;
    yield i;
  }
}
```

Ein weiterer Weg zur Vorführung der Zustandshaltigkeit ist die folgende Generatorfunktion. Sie hält sich gar nicht mit Schleifen auf, sondern besteht nur aus einer Gruppe von `yield`-Aufrufen:

```
function* generator(){
  console.log('0');
  yield 0;
  console.log('1');
  yield 1;
  console.log('2');
}
```

Zum Test dieses Generators bietet sich das folgende Snippet an. Die am Ende der jeweiligen Zeile befindlichen Kommentare zeigen, was in der Browserkonsole erscheinen würde:

```
var iterator = generator();
console.log(iterator.next());
// { value: 0, done: false }
console.log(iterator.next());
```

## Mehr über Prototypen

Wer sich intensiv mit Javascript auseinandersetzt, sollte sich zumindest einmal Gedanken zum Prototyp-System machen. Addy Osmanis Klassiker Learning JavaScript Design Patterns mag mittlerweile etwas veraltet sein, ist unter aber <https://addyosmani.com/resources/essentialjsdesignpatterns/book/> kostenlos lesbar und mehr oder weniger Pflichtlektüre für jeden Entwickler.

```
// { value: 1, done: false }
console.log(iterator.next());
// { value: undefined, done: true }
```

Ein weiterer Vorteil der Generatoren ist, dass man mehrere Generator-Objekte gleichzeitig anlegen kann. Im Fall unseres weiter oben abgedruckten Generators wäre es möglich, für jeden Client eines Servers eine neue Zahlenfolge anzulegen. Dazu müssten sie nur im für die jeweilige Verbindung verantwortlichen Objekt ein zusätzliches Feld unterbringen, in dem sie die Ergebnisse des Aufrufs unterbringen.

## Erhöhung der Typsicherheit

TypeScript 3.6 führt Erweiterungen ein, um die Typsicherheit der von Generatoren angelieferten Werten zu erhöhen. Microsoft schlägt zur Vorführung der Möglichkeiten die Verwendung des folgenden Generators vor:

```
function* foo() {
  if (Math.random() < 0.5) yield 100;
  return "Finished!"
}
```

Im Regelbetrieb liefert *foo* eine Zahl zurück, während er nach dem (zufällig ausgelösten) Ende der Schleifen-Abarbeitung einen String retourniert. Das endgültige Ergebnis eines Generators lässt sich über die *done*-Eigenschaft abrufen. Das folgende Snippet zeigt, dass ihr Wert ab 3.6 voll typisiert ist:

```
let iter = foo();
let curr = iter.next();
if (curr.done) {
  // TypeScript 3.5 and prior thought this was
  // a 'string | number'.
  // It should know it's 'string' since 'done' was
  // 'true'!
  curr.value
}
```

Ein weiteres Ärgernis war die Rückgabe von *yield*, die bisher immer als *any* angenommen wurde. TypeScript 3.6 bringt auch hier Änderungen:

```
function* bar() {
  let x: { hello(): void } = yield;
  x.hello();
}

let iter = bar();
iter.next();
iter.next(123); // oops!
```

Da *iter* nun nicht mehr vom Typ *any* ist, würde TypeScript 3.6 den Aufruf von *Next* mit einem Parameter verweigern. Ältere Versionen des Interpreters hätten diesen Fehler während der Transpilation nicht ausgeräuchert, was einen Laufzeitfehler provoziert hätte.

## Was ist mit der Ente?

Das Konzept vom Ententest ist nicht nur in der Informatik, sondern beispielsweise auch in den Politikwissenschaften weit verbreitet. Der Begriff stammt vom Zitat »When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck«, das auf ein Gedicht des Amerikaners James Whitcomb Riley zurückgeht.

Unsere bisher verwendeten Funktionen hatten zwar typisierte Parameter, informierten die Umgebung aber nicht darüber, welchen Wert das von ihnen zurückgegebene Resultat hat. Dieses Problem lässt sich mit wenig Aufwand aus der Welt schaffen:

```
function add(x: number, y: number): number {
  return x + y;
}
```

Analog zu allen anderen Variablen und Parameterdeklarationen kommt auch hier der Doppelpunkt-Parameter zum Einsatz. Der an ihm übergebene Typ bestimmt, was von der Funktion *F* in Richtung des Aufrufes zurückgegeben wird.

TypeScript erlaubt Entwicklern das Festlegen optionaler Parameter. Im Fall der folgenden Funktion muss der Aufrufer beispielsweise nur den Vornamen bereitstellen:

```
function buildName(firstName:string, lastName?:string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}
```

Alternativ dazu darf der Erzeuger der Funktion auch einen Standardparameter festlegen.

## Fazit

Wer .net- oder Java-Programmierer schnell zur Webentwicklung befähigen muss, wird TypeScript lieben. JavaScript-Großmeister mögen sich über den Verlust ihrer Freiheit beschweren: der durchschnittliche Entwickler freut sich, gewohnte syntaktische Elemente zu finden. Schon aus diesem Grund ist TypeScript eine Sprache, die Architekten und Projektmanager im Hinterkopf behalten sollten. ■



**Tam Hanna**

ist Autor, Trainer und Berater mit den Schwerpunkten Webentwicklung und Webtechnologien. Er lebt in der Slowakei und leitet dort die Firma Tamoggemon Holding k.s. Er bloggt sporadisch unter [www.tamoggemon.com](http://www.tamoggemon.com)



Foto: Shutterstock / StockEU

## OBJEKTE MIT CSS GRIDS OPTIMAL POSITIONIEREN

# Zweidimensional

Die Grundlagen von CSS Grids und wie man damit das Layout einer Website gestaltet.

In der aktuellen W3C Definition wird CSS Grids als zweidimensionales, rastergestütztes Layoutsystem beschrieben, das für das Design von Benutzeroberflächen optimiert ist. Dies klingt zwar beim ersten Lesen sehr technisch, umschreibt jedoch sehr präzise die Aufgaben des CSS-Moduls »CSS Grid Layout Module Level 1«. In der aktuellen Ausgabe vom Dezember 2017 hat es den Status des Candidate Recommendation erreicht.

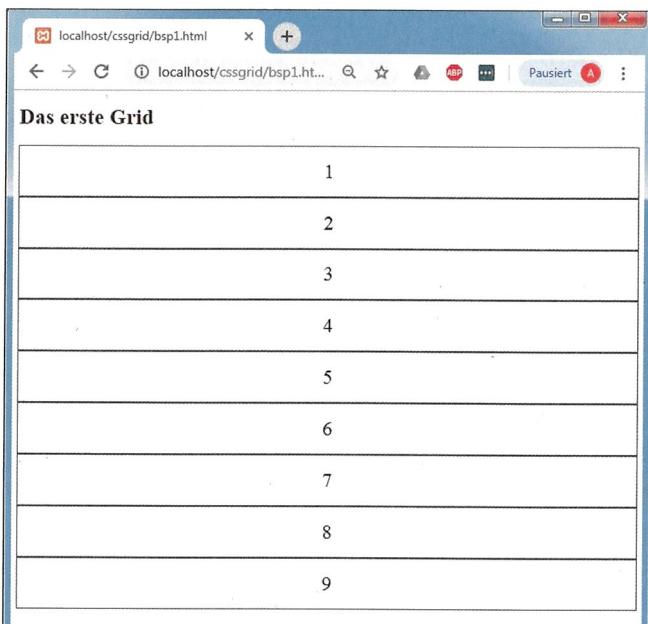
Viel wichtiger ist allerdings der Umsetzungsgrad bei den aktuellen Browsern: dort sind die CSS Grids inzwischen angekommen und als Technologie bereits seit 2017 etabliert. Die einzigen beiden Ausnahmen bilden die Browser von Microsoft. Der Internet Explorer 10 bietet nur eine eingeschränkte Unterstützung, beim Edge Browser werden CSS Grids erst seit Version 16 unterstützt.

Am besten steigen wir direkt in die Beschreibung von CSS Grids mithilfe eines konkreten Beispiels ein. In diesem bauen wir erst einmal mittels HTML ein Raster auf, das in den folgenden Schritten hinsichtlich Aussehen und Größe angepasst

werden kann. Hierfür wählen wir ein Raster mit 3x3 Feldern. Diese werden über eine Container-Klasse gruppiert und als `div`-Tags dargestellt:

```
<div class="grid-container">
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
  <div class="grid-item">7</div>
  <div class="grid-item">8</div>
  <div class="grid-item">9</div>
</div>
```

Wenn Sie dies in den `body`-Tag Ihrer Webseite einfügen und diese anschließend aufrufen, erhalten Sie lediglich eine Liste mit den Zahlen von eins bis neun die untereinanderstehen.



**Der Anfang:** Vor dem Einsatz der StyleSheets sieht das Grid lediglich wie eine Tabelle aus ([Bild 1](#))

Damit haben Sie die Grundlage für die nächsten Formatierungsschritte. Damit das Ganze erst einmal wie ein Gitter oder Raster aussieht, erhält die Container-Klasse eine entsprechende Formatierung:

```
.grid-container {
    display: grid;
}
```

Rufen Sie anschließend die Seite erneut auf, erhalten Sie eine erste Tabelle, allerdings nur mit einer Spalte und neun Zeilen ([Bild 1](#)). Als nächstes weisen wir unserem Container über die Eigenschaft *grid-template-columns* drei Spalten zu. Da wir die Anzahl der Spalten gleichmäßig über die Bildschirmbreite verteilen möchten, nutzen wir hierzu die Breite *auto*:

```
grid-template-columns: auto auto auto;
```

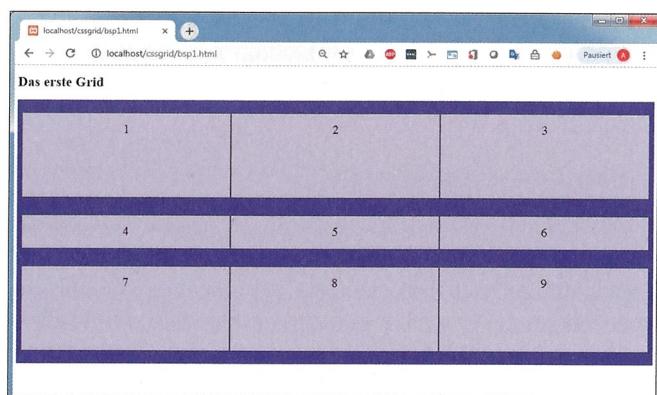
Alternativ dazu können Sie den einzelnen Spalten auch eine feste Breite zuordnen. In diesem Fall könnte die Definition beispielsweise wie folgt aussehen:

```
grid-template-columns: 80px 200px auto
```

CSS Grids bieten Ihnen nicht nur die Möglichkeit, Spalten zu formatieren, sondern auch die Zellen. Der Befehl *grid-template-rows()* funktioniert nach dem gleichen Schema wie das Gegenstück für die Spalten. Sie geben je Zeile einen Wert an. Im folgenden Beispiel weisen wir der ersten und dritten Zeile eine Breite von 200 Pixel zu und der mittleren von 80 Pixel:

```
grid-template-rows: 200px 80px 200px;
```

Bei der Formatierung der Zellen gibt es zahlreiche Eigenschaften, die Sie vom Umgang mit Tabellen auch kennen.



**Zwischenräume:** Mit den Befehlen *space-around* und *space-between* definieren Sie den Abstand zwischen den Zellen ([Bild 2](#))

Wenn Sie den Container größer machen als die Gesamthöhe der Zellen, dann können Sie die Anordnung der Zellen durch verschiedene Eigenschaften beeinflussen:

```
height: 600px;
background-color: blue;
padding: 10px;
```

Der Container erhält eine Gesamthöhe von 600 Pixel, damit sind also 120 Pixel an freier Fläche in der Höhe vorhanden. Damit Sie die Zellen und den Hintergrund besser voneinander unterscheiden können, wird dieser blau eingefärbt. Dank der Eigenschaft *grid* werden die Zellen automatisch hellblau. Mittels *padding* werden die Grid-Zellen 10 Pixel weg vom Container-Rand platziert.

Noch hängen Ihre neun Zellen jedoch am oberen Rand des Containers. Die genaue Positionierung lässt sich im nächsten Schritt über die Eigenschaft *align-content* durchführen. Für die Positionierung des Grids entlang der vertikalen Achse gibt es drei Werte: *flex-start* platziert es am oberen Rand, *center* in der Mitte und *flex-end* entsprechend am unteren Rand. Interessant sind auch noch die beiden Werte *space-around* und *space-between*. Beim ersten wird der noch verfügbare freie Platz um die einzelnen Reihen – auch oben und unten – gleichmäßig platziert. Im zweiten Fall wird oben und unten nur ein kleiner Rand gelassen, der freie Platz entsprechend zwischen den innen liegenden Reihen verteilt ([Bild 2](#)).

## Steuern der Abstände

Wenn Ihnen diese Vorgehensweise für die Definition der Abstände zu ungenau ist, stehen Ihnen weitere Eigenschaften zur Verfügung, die Sie auf den Container anwenden können. *grid-column-gap* legt die Abstände zwischen den Reihen des Grids fest, *grid-row-gap* entsprechend zwischen den Zeilen des Grids:

```
grid-row-gap: 30px;
grid-column-gap: 30px;
```

Wenn Ihnen dies zu viel Schreibaufwand ist, können Sie auch die Eigenschaft *grid-gap* verwenden. Diese kann ein oder ►

zwei Parameter besitzen. Im Falle von zwei Werten sind dies nacheinander der Abstand der Spalten und der Zeilen. Geben Sie nur einen Wert an, wird dieser für beide Abstände gleichermaßen verwendet (Bild 3).

## Arbeiten mit den Kindern

Bis jetzt haben wir uns um den Container gekümmert und die Formatierungen darauf angewendet. Innerhalb des Containers befinden sich jedoch auch die einzelnen Zellen, die Kind-Elemente. Wie Sie bereits gesehen haben, werden diese im Standard von links oben nach rechts unten angeordnet.

Diese Reihenfolge können Sie jedoch über die Container-Eigenschaft *grid-auto-flow* beeinflussen. Mit dem Wert *column* werden die Felder von oben nach unten sortiert (Bild 4):

```
grid-auto-flow: column;
```

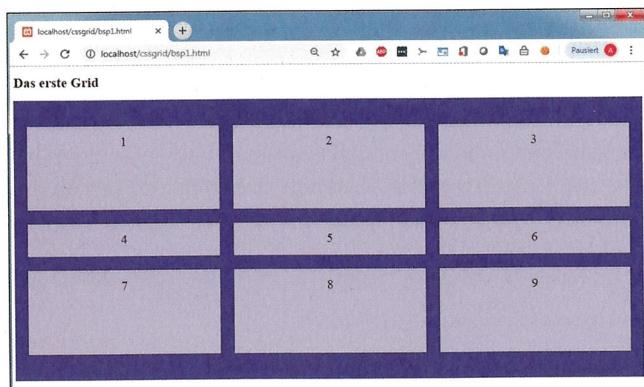
Damit haben Sie einige der zentralen Eigenschaften des Grids kennengelernt. Jetzt ist es an der Zeit, das Grid für die Positionierung von Elementen zu verwenden. In unserem nächsten Beispiel nehmen wir wieder das 3x3 Grid als Ausgangsbasis. Wir wollen damit eine Website mit Header, Footer und einem Inhaltsbereich aufbauen. Die Kopfzeile erstreckt sich über die obere Zeile, die Fußzeile entsprechend über die untere. Der verbleibende Rest wird für den Inhalt verwendet.

Daraus ergibt sich die folgende Struktur für den Container und die Kind-Elemente:

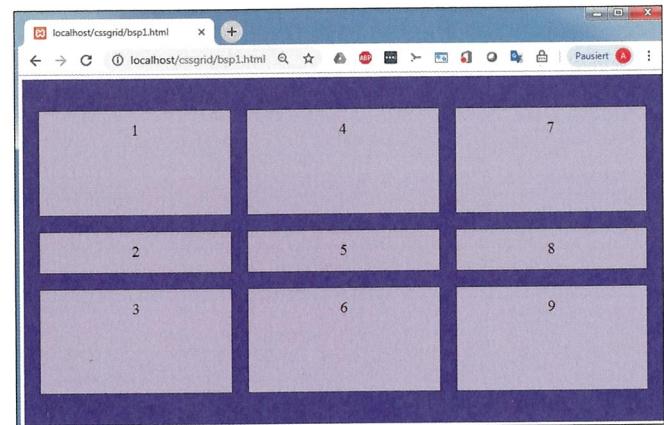
```
<div class="grid-container">
  <header>Kopfzeile</header>
  <content>Inhalt</content>
  <footer>Fusszeile</footer>
</div>
```

Die Formatierung des Containers haben wir die zentralen Elemente des vorherigen Beispiels beibehalten, allerdings die Größenangaben ein wenig angepasst:

```
.grid-container {
  display: grid;
```



**Individuelle Abstände:** Sie können die Abstände zwischen den Zeilen und Spalten bei den Grids individuell festlegen (Bild 3)



**Reihenfolge:** Die einzelnen Zellen ordnen Sie entweder von oben nach unten oder von links nach rechts an (Bild 4)

```
grid-template-columns: auto auto auto;
grid-template-rows: 100px 1fr 50px;
height: 600px;
background-color: blue;
padding: 10px;
align-content: center;
}
```

Wenn Sie die Webseite jetzt aufrufen, sehen Sie allerdings noch keine Grid-Struktur, sondern lediglich die drei Begriffe *Kopfzeile*, *Inhalt* und *Fusszeile*. Diese sind am oberen Rand des Containers auf drei Spalten verteilt. Die Adressierung der Spalten erfolgt anhand der Ränder der Spalten. Diese und die Zeilen sind durchnummert von 1 bis 4 (Bild 5).

## Vertikale und horizontale Achsen

Zum Adressieren der Bereiche wählen Sie entsprechend die Eigenschaften *grid-row-start* und *grid-row-end*, um einen Bereich auf der vertikalen Achse zu markieren. Für die horizontale Achse entsprechend *grid-column-start* und *grid-column-end*. Für unsere 3x3 Matrix bedeutet dies somit für die Markierung einer kompletten Zeile – hier der Kopfbereich – die folgende Beschreibung:

```
header {
  background:red;
  grid-column-start:1;
  grid-column-end:4;
  grid-row-start:1;
  grid-row-end:2;
}
```

Der Startpunkt für den Kopfbereich ist somit die linke äußere Begrenzungslinie. Diese trägt die Nummer 1. Der Endpunkt ist entsprechend die rechte äußere Begrenzungslinie mit der Nummer 4. Da sich der Abschnitt nur über eine Zeile erstreckt, ist der Start hier entsprechend die obere und untere Begrenzungslinie der ersten Zeile. Damit der Bereich sich anschließend optisch vom Rest abhebt, haben wir ihn noch rot eingefärbt.

Auf die gleiche Art und Weise werden auch die beiden Bereiche *content* und *footer* formatiert. Die Festlegung für die horizontale Ausrichtung ist gleich, lediglich der Zähler für die Zeilen ändert sich, da die zweite beziehungsweise dritte Zeile formatiert werden. Für die optische Unterscheidung kommen die Farben gelb und grün zum Einsatz:

```
content {
    background:yellow;
    grid-column-start:1;
    grid-column-end:4;
    grid-row-start:2;
    grid-row-end:3;
}

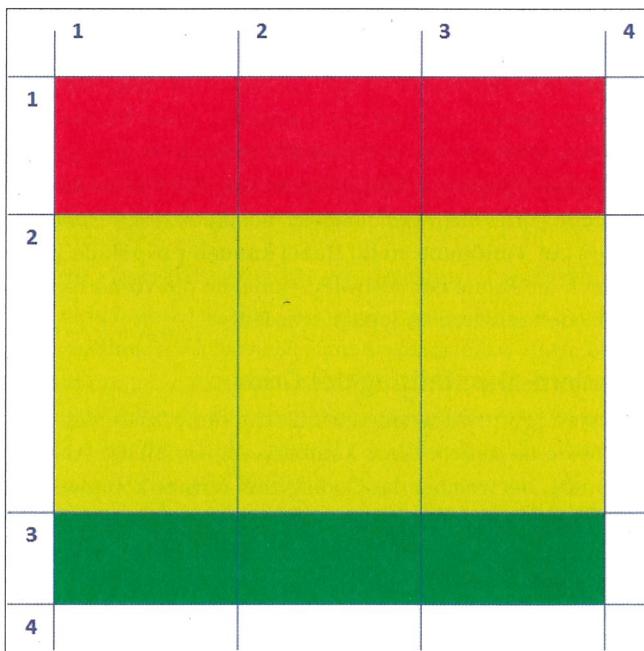
footer {
    background:green;
    grid-column-start:1;
    grid-column-end:4;
    grid-row-start:3;
    grid-row-end:4;
}
```

CSS bietet Ihnen auch an dieser Stelle die Möglichkeit, die Schreibweise noch ein wenig zu komprimieren. Dazu werden die beiden Befehle *grid-column-start* und *grid-column-end* zusammengefasst. Das Ergebnis in unserem Fall lautet dann:

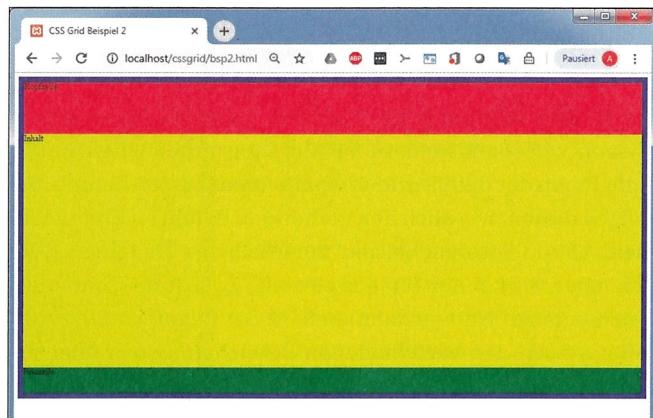
```
grid-column: 1 / 4;
```

Diese Kurzschreibweise entspricht exakt der Langform:

```
grid-column-start:1;
```



**Begrenzer:** Die Ränder zwischen den einzelnen Zellen werden durchnummiert und starten jeweils am linken beziehungsweise oberen Rand mit 1 (Bild 5)



**Benannte Bereiche:** Durch die Verknüpfung mehrerer Zellen entsteht ein benannter Bereich, etwa für Kopf- und Fußzeile sowie die Inhalte (Bild 6)

```
grid-column-end:4;
```

Analog verhält es sich mit den Beschreibungen für die Spalten für die es die Kurzform *grid-row* gibt. Damit haben Sie die Entwicklung des ersten Layouts auf Basis von CSS-Grids abgeschlossen (Bild 6).

Im vorherigen Beispiel sind die Rasterlinien des Grids mithilfe von Zahlen – beginnend mit 1 – durchnummiert worden. Dies kann, je nach Größe des Grids unter Umständen auch ein wenig unübersichtlich werden. Deswegen haben Sie an dieser Stelle auch die Möglichkeit, die einzelnen Linien mit Namen zu versehen. Hierfür verwenden Sie weiterhin die beiden Eigenschaften *grid-template-rows* und *grid-template-columns*. Allerdings kombinieren Sie die Breitenangaben der jeweiligen Zeile oder Spalte mit den Namen für die vorhandenen Rasterlinien. Diese werden in die Definition mit eckigen Klammern eingefügt.

Im folgenden Beispiel werden die horizontalen Linien wieder durchnummiert, erhalten jedoch ein vorangestelltes h. Bei den vertikalen Linien wird der Nummerierung entsprechend ein v vorangestellt. Daraus ergibt sich für das vorherige Beispiel die folgende Container-Definition mitsamt der vier horizontalen und vertikalen Linien:

```
grid-template-columns:[h1] auto [h2] auto [h3]
auto [h4];
grid-template-rows: [v1] 100px [v2] 1fr [v3] 50px [v4];
```

Die restlichen Konfigurationsinformationen für den Container bleiben entsprechend erhalten. Der Zugriff auf die Linien findet in der Definition der drei Kind-Elemente entsprechend über die Namen statt. Daraus ergibt sich exemplarisch für den Bereich *header* die folgende angepasste Definition:

```
header {
    background:red;
    grid-column-start: h1;
    grid-column-end:h4;
    grid-row-start: v1;
```

```
grid-row-end: v2;
}
```

Sie können allerdings nicht nur die Rasterlinien mit einem Namen versehen, sondern auch die Rasterbereiche. Hierfür steht Ihnen der Befehl *grid-template-areas* zur Verfügung. Sie nutzen diesen, wie auch den vorherigen Befehl im Eltern-Element, also in unserem Beispiel innerhalb des Containers *grid-container*. Sie können für jede einzelne Zelle Ihres Containers einen eigenen Namen definieren. In der Regel wird die Aufteilung nach den verschiedenen Bereichen jedoch über die gleichen Zellennamen realisiert. Das folgende Beispiel setzt unsere vorherigen Definitionen von *header*, *content* und *footer* um und gibt diesen die entsprechenden Zellen-Namen:

```
.grid-container {
    display: grid;
    grid-template-columns: auto auto auto;
    grid-template-rows: 100px 1fr 50px;
    height: 600px;
    background-color: blue;
    padding: 10px;
    align-content: center;
    grid-template-areas:
        "header header header"
        "content content content"
        "footer footer footer"
    ;
}
```

Wie Sie am Programmbeispiel sehen, werden die einzelnen Zellen-Namen innerhalb von *grid-template-areas* mittels Anführungszeichen definiert. Zwischen jedem Zellennamen steht ein Leerzeichen. Rufen Sie anschließend die Webseite auf, werden Sie allerdings außer der Aufteilung nach den drei Spalten noch nichts von der Umsetzung erkennen. Sie müssen die Namen der Zellen erst noch mit den Kindelementen verknüpfen. Hierfür steht Ihnen die Eigenschaft *grid-area* in Kombination mit dem Namen der Zelle zur Verfügung. Für die bessere optische Hervorhebung haben wir auch hier wieder die einzelnen Bereiche farblich hervorgehoben:

```
header {
    grid-area: header;
    background:red;
}
content {
    grid-area: content;
    background:yellow;
}
footer {
    grid-area: footer;
    background:green;
}
```

Damit sieht das Beispiel optisch wieder wie seine Vorgänger aus. Am Ende gibt es an dieser Stelle unterschiedliche Wege,



**Lücken im Layout:** Sie können bei der Definition des Layouts auch einzelne Bereiche frei lassen – entweder mit dem Wert *none* oder einem einfachen Punkt ([Bild 7](#))

wie Sie ans Ziel kommen können. Es ist stark davon abhängig, wie Ihr Layout aussieht. Neben der Zuweisung eines Namens erlaubt es die Eigenschaft *grid-template-areas* auch, bestimmte Zellen aus der Definition auszuschließen. Wenn Sie beispielsweise innerhalb unseres 3x3 Grids in der zweiten Zeile die dritte Zelle ganz rechts nicht als Inhalt definieren möchten, weisen Sie dieser einfach den Wert *none* zu. In diesem Fall wird sie nicht berücksichtigt und der Bereich ist in der Breite entsprechend kleiner:

```
grid-template-areas:
    "header header header"
    "content content none"
    "footer footer footer"
;
```

Alternativ zu dem Wert *none* können Sie als Abkürzung auch einfach einen Punkt verwenden ([Bild 7](#)).

In unseren bisherigen Beispielen hatten die Grids eine feste Größe, die sich anhand des Containers orientiert hat. Gerade bei Websites für den mobilen Bereich ist dies jedoch nicht immer gegeben. Deswegen nutzen Entwickler an dieser Stelle gerne die Möglichkeiten, die ihnen das Responsive Design zur Verfügung stellt. Dabei können sowohl die Höhe als auch die Breite der Website flexibel an die vorherrschenden Gegebenheiten angepasst werden.

## Standard-Darstellung des Grids

Im ersten Schritt verwenden wir hierfür den klassischen Weg über *@media screen*. Diese kombinieren wir mit der Angabe der Breite, bei welcher das Coding zum Einsatz kommen soll. In unserem Beispiel sind dies 37,5em, was in etwa 600 Pixeln entspricht. In diesem Fall soll das Layout wie zuvor dargestellt werden.

Für eine bessere optische Unterscheidung haben wir das Layout in ein 4x3 Layout (vier Spalten, drei Zeilen) geändert. Der Inhaltsbereich umfasst für den Hauptinhalt drei Spalten – Feld *content* – und für Zusatzinformationen eine Spalte. Diese wird durch das Feld *c1* repräsentiert. Daraus ergibt sich

für die Standard-Darstellung des Grids das folgende Coding:

```
@media screen and (min-width: 37.5em) {
  .grid-container {
    grid-template-rows: 100px 1fr 50px;
    grid-template-columns: auto auto auto;
    grid-template-areas:
      "header header header header"
      "content content content c1"
      "footer footer footer footer";
  }
}
```

Falls der Bildschirm beziehungsweise das Browser-Fenster kleiner als die 600 Pixel sein sollte, kommt ein alternatives Layout zum Einsatz. Dieses hat nur eine Spalte, aber dafür vier Zeilen:

```
.grid-container {
  height: 100vh;
  display: grid;
  grid-template-columns: 100%;
  grid-template-rows: 100px 4fr 1fr 50px;
  grid-template-areas:
    "header"
    "content"
    "c1"
    "footer";
}
```

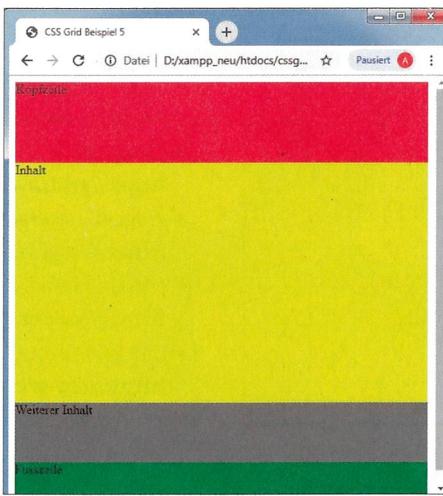
Wird die Webseite nun über einen kleineren Bildschirm mit weniger als 600 Pixel Breite aufgerufen, wird die Darstellung auf eine Spalte beschränkt ([Bild 8](#)).

## Alternative Umsetzung von Responsive Design

CSS Grids bieten Ihnen aber auch noch einen alternativen Weg um Ihre Responsive Designs zu realisieren. Diese basieren auf der `minmax()`-Funktion. Wenn Sie beispielsweise ein Design mit zwei Spalten benötigen, wobei die erste zwischen 200 und 500 Pixel breit sein soll und die zweite 1fr, dann lautet die Definition wie folgt:

```
display: grid;
grid-template-columns: minmax(200px, 500px) 1fr;
```

Diese beiden Zeilen reichen für ein Responsive Design bereits aus. Es gibt im Umgang mit der Funktion noch ein paar Einschränkungen: logischerweise kann der Wert, den Sie als Minimum angeben nicht größer sein als das Maximum. Der



**Responsive Design:** Ein flexibles Layout für unterschiedliche Display-Größen lässt sich mit einem CSS-Grid auf unterschiedliche Weise umsetzen ([Bild 8](#))

Einsatz der Einheit `fr` ist innerhalb der `minmax()`-Funktion möglich, allerdings nur für den maximalen Wert:

```
grid-template-columns:
  minmax(250px, 1fr) 1fr;
```

In diesem Beispiel hat das Design zwei Spalten. Die erste hat mindestens 250 Pixel und maximal die Hälfte des zur Verfügung stehenden Platzes. Die Breite der zweiten Spalte ist mindestens [Gesamtbreite des Containers – 250 Pixel] breit.

Wenn Sie den Befehl für das vorherige Beispiel einsetzen möchten, ergibt sich daraus – unter Berücksichtigung der minimalen Breite von 37.5em – die folgende Spaltendefinition:

```
grid-template-columns:
  minmax(150px, 1fr) minmax(150px, 1fr) minmax(150px, 1fr)
  minmax(150px, 1fr);
```

Die vier Spalten haben somit die gleiche Breite, die mindestens 150 Pixel entspricht und maximal ein Viertel der zur Verfügung stehenden Breite darstellt.

Sollen sich die vier Spalten nicht in ihrer Breite unterscheiden wiederholt sich die Definition zwangsläufig vier Mal. Aus Effizienzgründen wurde für einen solchen Fall die `repeat()`-Funktion implementiert. Sie können diese im Zusammenspiel mit den Eigenschaften `grid-template-columns` und `grid-template-rows` einsetzen. Die Funktion besitzt zwei Parameter: die Anzahl der Wiederholungen und die zu wiederholende Definition:

```
grid-template-columns: 1fr 2fr 1fr 2fr 1fr 2fr;
grid-template-rows: 100px auto 20% 100px auto 20%; ►
```

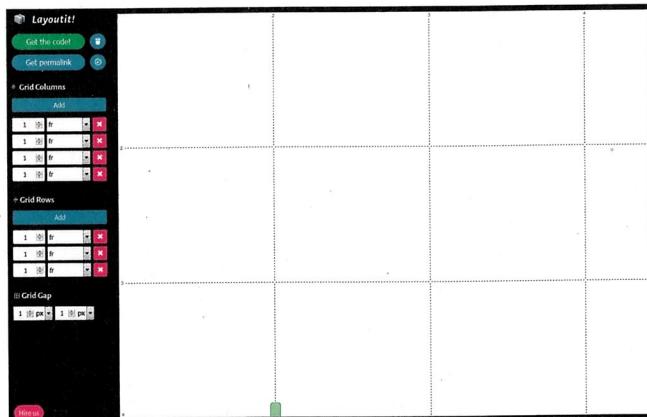
Dieses Beispiel lässt sich einfach optimieren und übersichtlicher gestalten. Bei der Definition der Spalten kommen die Angaben `1fr 2fr` drei Mal vor, bei den Zeilen sind `100px auto 20%` doppelt vorhanden. Umgesetzt mit der `repeat()`-Funktion ergibt dies die folgende alternative Definition:

```
grid-template-columns: repeat(3, 1fr 2fr);
grid-template-rows: repeat(2, 100px auto 20%);
```

Wenn wir dies auf unser obiges Beispiel anwenden, haben wir anstelle der sehr langen Definition für die Spalten eine sehr übersichtliche Darstellung erhalten.

## Weitere Optimierungen

Die `repeat()`-Funktion bringt allerdings auch ein Problem mit: sobald die Breite des Ansichtsfensters zu gering ist um die Anforderungen zu erfüllen – im obigen Beispiel kleiner als 600 Pixel – werden die Spalten nicht umgebrochen und das



**CSS Generator:** Es stehen Ihnen verschiedene kostenlose Online-Generatoren für die Gestaltung Ihres CSS Grids im Internet zur Verfügung (Bild 9).

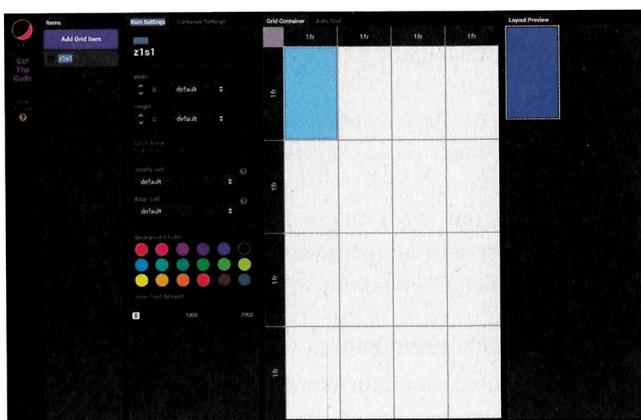
Layout funktioniert nicht. Aus diesem Grund wird anstelle einer exakten Anzahl an Wiederholungen oftmals die Funktion *auto-fit()* verwendet.

Damit übernimmt der Browser den Zeilenumbruch für Sie und ermittelt dafür die optimale Anzahl an Spalten. Dafür wird die maximal mögliche Anzahl ermittelt und der restliche Platz auf diese Spalten aufgeteilt:

```
grid-template-columns: repeat(auto-fit, minmax(300px, 500px));
```

Wenn Sie beispielsweise einen Viewport vorfinden, der 1400 Pixel breit ist, dann passen dort nach der obigen Definition genau vier Spalten rein. Diese haben dann jeweils eine Breite von 350 Pixel. Ist der Viewport hingegen 1600 Pixel breit, dann sind dies fünf Spalten mit einer Breite von 520 Pixel. Diese Berechnung übernimmt beim Einsatz der Funktion *auto-fit()* der Browser für Sie.

Damit kennen Sie jetzt die notwendigen Grundlagen um in Zukunft das Layout Ihrer Webseiten Grid-basiert zu gestalten. Gerade die Grunddefinition eines Grids kann jedoch ein wenig aufwändiger sein, abhängig von Ihrem Layout. Dies



**Leistungsfähig:** Einer der Generatoren mit dem größten Funktionsumfang stammt von dem Entwickler Dmitrii Bykov (Bild 10)

## Links zum Thema

- CSS Grid Layout Module Level 1 – W3C  
<https://www.w3.org/TR/2017/CR-css-grid-1-20171214/>
- Layoutit! CSS Grid Generator  
<https://grid.layoutit.com/>
- cssgr.id – Interactive CSS Grid Tool  
<https://cssgr.id/>
- CSS Grid Template Builder  
<https://codepen.io/anthonydugois/full/RpYBmy/>
- CSS Grid Layout Generator  
<https://css-grid-layout-generator.pw/>

hat zu zahlreichen Websites im Internet geführt, mit denen Sie das grundlegende Layout des Grids gestalten können. Dies beinhaltet sogar die Verwendung von Bezeichnern für die einzelnen Zellen und Spalten.

Die beiden Angebote cssgr.id und Layoutit! bieten Generatoren mit einem vergleichbaren Leistungsumfang, allerdings mit einem etwas unterschiedlichen Layout. Während cssgr.id sehr minimalistisch daherkommt, bietet Ihnen Layoutit! eine deutlich intuitivere Oberfläche (Bild 9).

Einen anderen Ansatz wählt der Entwickler Anthony Dugois mit seinem Codepen-Projekt. Über die Oberfläche können Sie über die Anzahl von Zeilen und Spalten das Layout bestimmen, die Bereiche benennen und anschließend per Maus auf dem Bildschirm verteilen. Das Ergebnis übernehmen Sie mittels Copy & Paste in Ihren eigenen Programmcode. Die meisten Funktionen bietet der CSS Grid Layout Generator von Dmitrii Bykov. Damit Sie nicht den Überblick verlieren, hat der Entwickler noch ein Video gedreht, in dem er die Vorgehensweise bei der Anlage Ihres persönlichen Layouts erläutert. Kurz zusammengefasst legen Sie als erstes das grundlegende Aussehen des Grids fest und können anschließend zu jeder Zeile, Spalte und Zelle die Eigenschaften festlegen. Den resultierenden Programmcode übernehmen Sie – unterteilt nach CSS und HTML – zum Abschluss per Copy & Paste (Bild 10).

## Fazit

CSS Grids bieten eine leistungsstarke Möglichkeit, Layouts flexibel und einfach zu gestalten. Dank der durchgängigen Unterstützung aller aktuell gängiger Browser – außer dem Internet Explorer – werden diese auf allen Plattformen gleichermaßen zuverlässig angezeigt. ■



**Andreas Hitzig**

arbeitet seit mehr als zwei Jahrzehnten als freiberuflicher IT Autor. Neben dem Thema Web Development sind seine weiteren Schwerpunkte Android und IT Security.



Foto: Shutterstock / Kentoh

## SCHNELLEINSTIEG IN JAVASCRIPT (TEIL 2)

# Objektorientiert

In JavaScript kann man auch objektorientiert programmieren.

In folgenden Artikel wird eine Einführung in die objektorientierte Programmierung mit JavaScript gegeben.

Die objektorientierte Programmierung galt bei ihrer Einführung als die Lösung für die immer komplexer werdenden Problemstellungen und deren Umsetzung in Programmen. Statt die reale Welt in Prozeduren und Funktionen zu transformieren, war es mit der objektorientierten Programmierung erstmals möglich, die Struktur der Wirklichkeit in Form eines Programms nachzubilden.

Anstatt also eine Person nur mit Funktionen und Prozeduren zu beschreiben beziehungsweise die einzelnen Daten wie zum Beispiel Nachname, Vorname etc. zu verarbeiten, ist es mit der objektorientierten Programmierung möglich, ein reales Objekt in Form eines Programms nachzubilden. Diese Art der Programmierung kommt dem menschlichen Denken näher als die prozedurale Programmierung.

So verfügt ein Objekt, zum Beispiel eine Person, über Eigenschaften (Name, Größe oder auch Personalnummer) und eben diese Eigenschaften lassen sich mit der objektorientierten Programmierung in Form von Programmcode nachbilden.

Um es vorwegzunehmen: Auch die OOP (Abkürzung für objektorientierte Programmierung) war nicht die endgültige Lösung. Als Baustein trägt sie aber dazu bei, umfangreichen Anforderungen der realen Welt besser in Programmcode abzubilden.

Ein Objekt ist dabei der zentrale Dreh- und Angelpunkt während der Entwicklung. So kann ein Objekt nicht nur Daten enthalten, sondern es stehen auch Methoden/Funktionen innerhalb des Objektes bereit, um die enthaltenen Daten zu manipulieren. Ein Beispiel für ein Objekt ist ein Formular, das Sie zur Programmlaufzeit aufrufen. Dieses Formular besitzt viele unterschiedliche Eigenschaften (*Properties*), beispielsweise die Höhe (*Height*) und Breite (*Width*). Natürlich können diese Eigenschaften beziehungsweise die darin enthaltenen Daten auch zur Laufzeit wieder geändert werden.

Auslöser der Änderungen ist in der Regel ein Ereignis (*Event*), das zu einem definierten Zeitpunkt eintritt. Betrachtet man es genauer, so verbirgt sich hinter einem Ereignis eine entsprechende Methode. Und nun ist die Definition beisammen. Ein Objekt ist, im Prinzip, die Variable eines ►

Typs, der nicht nur Daten enthält, sondern auch Funktionen, um diese Daten zu bearbeiten. In der Begriffswelt der Objektorientierung spricht man allerdings nicht von Funktionen, sondern nur allgemein von den Methoden eines Objekts.

### Klassen und Objekte

Da ja laut Definition im vorherigen Abschnitt Objekte nur die Variablen sind, sollte es natürlich auch (Daten-)Typen geben, von denen die Objekte abgeleitet werden. Hierfür gibt es in vielen OOP-Sprachen die Klasse, welche in der Regel mit dem Schlüsselwort *class* definiert wird. Eine Klasse ist eine Vorlage für ein Objekt. Die Sprache JavaScript allerdings kennt das Konstrukt der Klasse nicht.

Also ist eine Klasse so eine Art Schablone. Sie erinnern sich bestimmt noch an die Förmchen im Sandkasten. Die Form ist die Klasse und das Häufchen Sand ein Objekt. Im letzten Teil der Serie haben Sie Felder (Arrays) als ein Element zur Speicherung von Daten kennen gelernt. Ein Feld besteht aus einer bestimmten Anzahl von Variablen eines Datentyps.

Stellen Sie sich eine Klasse also (erst einmal) einfach wie ein Feld vor, mit dem Unterschied, dass dieses mehrere Variablen unterschiedlichen Datentyps beinhalten kann. Außer den Daten kann eine Klasse zusätzlich noch Programmcode enthalten. Dieser Programmcode wird in Bereichen abgelegt, die im Fachjargon als Methoden (in JavaScript nennt man diese Funktionen) bezeichnet werden.

### JavaScript und Objekte

JavaScript kennt das Konstrukt Klasse nicht und es gibt auch kein Gegenstück. Allerdings kann JavaScript mit Variablen aller Art umgehen. Es hält Sie also nichts davon ab, in JavaScript objektorientiert zu programmieren. Die typischen Merkmale der Objektorientierung (wie zum Beispiel Klassen, Kapselung etc.) fehlen natürlich trotzdem. Da es das Schlüsselwort *class* in JavaScript nicht gibt, wird stattdessen *function* verwendet. In JavaScript muss sämtlicher Code, welcher geschrieben wird innerhalb einer Funktion eingeschlossen werden. Aus diesem Grund bietet sich dieses Konstrukt natürlich auch für das Anlegen einer Variablen mit den Eigenschaften einer Person an:

```
function Person() {
    this.Nachname = '-';
    this.Vorname = '-';
    this.Personalnummer = 0;
}
```

Die Variablen werden unter Angabe des Namens und des Schlüsselwortes *this* innerhalb der Funktion *Person* angelegt. Mit dem Schlüsselwort *this* können Sie einen Bezug zum aktuellen Objekt herstellen. Später folgen weitere Informationen zur Verwendung des Schlüsselworts *this*. In diesem speziellen Fall zur Anlage der drei Variablen. Initialisiert werden Sie in diesem Beispiel mit Standardwerten. Erst im folgenden Programm soll den einzelnen Variablen jeweils ein Wert zugewiesen werden. Dies geschieht in einer JavaScript-Funktion, welche einem HTML-Formular zugeordnet ist. Auch hier

finden Sie einige Schlüsselworte, welche erst später erläutert werden. Wichtig ist der Teil des Programms, in welchem die neue Variable *Person* angelegt und initialisiert wird:

```
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        var person = new Person();
        person.Nachname = 'Meier';
        person.Vorname = 'Hans';
        person.Personalnummer = 12345;
    };
})());
```

Hier werden die Variablen des Objekts über einen Punkt getrennt vom Objektnamen *person* aufgerufen. Auch hier ermöglicht der Punkt-Operator den Zugriff auf Bestandteile des Objekts.

Vielleicht haben Sie schon einmal gesehen, dass an der einen oder anderen Stelle in anderen Sprachen Variablen einer Klasse oder auch eine Klasse selbst mit einem zusätzlichen Schlüsselwort wie *public* oder *private* versehen sind. Mit diesen Schlüsselwörtern wird der Zugriff auf ein gekennzeichnetes Element gesteuert. Ist ein Element, beispielsweise eine Variable, die innerhalb einer Klasse definiert wurde, mit dem Schlüsselwort *public* gekennzeichnet, so ist ein Zugriff von außen möglich. Wird hingegen das Schlüsselwort *private* verwendet, so ist der Zugriff auf die Variable geschützt und von außen nicht möglich.

In JavaScript gibt es die Schlüsselworte *private* oder *public* nicht. Alle definierten Elemente einer Funktion sind somit erst einmal nach außen hin sichtbar. Trotzdem kann man durch Verwendung von Funktionen den Zugriff auf Variablen einschränken, wie das folgende Listing demonstriert:

```
function Konto() {
    this.Kontonummer = "123456";
    function Geheimnummer() {
        var gNummer = 999999;
    }
}
```

Damit die Variable *gNummer*, welche die Geheimnummer enthält, nach außen hin nicht sichtbar ist, wird diese innerhalb der Funktion *Geheimnummer* mit Hilfe des *var* Schlüsselwortes angelegt. Dadurch, dass die Variable *gNummer* innerhalb der Funktion mit dem *var* Schlüsselwort angelegt wurde, ist sie direkt der Funktion *Geheimnummer* zugeordnet und so von außen nicht sichtbar. Mit folgendem Listing lässt sich das Ganze überprüfen:

```
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        var mein_konto = new Konto();
```

```
var test_kontonummer = mein_konto.Kontonummer;
var test_gNummer = mein_konto.gNummer;
};

})();
```

Nach Anlage des neuen Objekts `mein_konto` wird in den beiden folgenden Zeilen versucht, die Variablen `Kontonummer` und `gNummer` auszulesen. Die Variable `Kontonummer` wurde mit dem Schlüsselwort `this` angelegt. Dadurch wird ein direkter Bezug zur Funktion `Konto` hergestellt und die Variable ist auch nach außen hin sichtbar. Die Zuweisung zur lokalen Variable funktioniert problemlos. Im folgenden Schritt soll `gNummer` ausgelesen werden. Dieser Versuch geht allerdings schief. Schaut man sich die Variable `test_gNummer` im Debugger an, so erhält man den Hinweis, dass der Inhalt `undefined` ist also undefiniert ist. Somit wurde dasselbe erreicht wie in anderen OOP-Sprachen allerdings ohne das fehlende Schlüsselwort `private` zu verwenden.

## Funktionen statt Methoden

Neben den Attributen sind die Methoden ein weiteres Merkmal von Klasse und Objekten. Die Bezeichnung Methode ist nur ein anderer Name für Funktion. Eine Funktion besteht aus einer oder mehreren Anweisungen. Sie kann am Ende ein Ergebnis zurückgeben, das muss aber nicht der Fall sein.

Eine Funktion wird über einen definierten Namen im Programm angesprochen. Es ist möglich, aber nicht notwendig, einer Funktion Argumente zur internen Verarbeitung zu übergeben. Die Argumente einer Funktion werden in der Regel nach dem Funktionsnamen definiert und auch übergeben. Die Argumente können Variablen sein, deshalb ist es wichtig, bei der Definition einer Funktion auf die korrekte Angabe des Typs zu achten. In JavaScript wird eine einfache Funktion wie folgt deklariert:

```
function bspGetString() {
    return "Eine Zeichenkette";
}
```

Anschließend folgt der Funktionsname, über den diese aufgerufen werden kann. Es folgen eine öffnende und eine schließende Klammer. Zwischen den Klammern können Pa-

### Listing 1: Parameter übergeben

```
(function () {
    //Quellcode entfernt
    app.onactivated = function (event0bject) {
        //Quellcode entfernt
        var result = Addition(5, 6);
    };
    function Addition(p1, p2) {
        return p1 + p2;
    }
})();
```

rameter platziert werden. Funktionen werden zwischen der ersten öffnenden Klammer `{` und der letzten schließenden Klammer `}` des übergeordneten Elements definiert:

```
(function () {
    function bspGetString()
    {
        return "Eine Zeichenkette";
    }
})();
```

Im Beispiel wurde eine Funktion `bspGetString` definiert, welche eine Zeichenkette als Wert zurück gibt.

Im letzten Abschnitt war bereits davon die Rede, dass einer Funktion Argumente übergeben werden können. Diese Argumente werden bei Funktionen auch als Parameter bezeichnet. Parameter sind immer dann nützlich, wenn in einer Funktion eine bestimmte Aufgabe erledigt werden soll, zu deren Lösung bestimmte Werte benötigt werden. Ein Beispiel: Eine Funktion `Addition` soll zwei Parameter (`p1` und `p2` ganzzahlige Variablen) entgegennehmen, diese addieren und das Ergebnis dann zurückgeben ([Listing 1](#)). Die Parameter werden innerhalb der Funktion wie lokal deklarierte Variablen benutzt. Die Definition der Parameter erfolgt innerhalb der Parameterliste direkt nach dem Funktionsnamen zwischen `(` und `)`. Im Funktionskörper erfolgen die Addition der beiden Werte und die Rückgabe des Ergebnisses.

## Eigenschaften

In der Regel soll es nicht möglich sein, auf Variablen einer Klasse direkt zugreifen zu können. Deshalb werden diese Variablen, wie bereits erwähnt, in anderen Sprachen auch oft mit dem Schlüsselwort `private` gekennzeichnet. Ein Zugriff von außen ist dann nicht mehr möglich. Aber natürlich muss auch auf den Inhalt solcher Variablen zugegriffen werden können. Ermöglicht wird dies durch die Verwendung von Eigenschaften (Properties). Auch JavaScript kennt Eigenschaften (Properties). Allerdings in einer etwas anderen Ausprägung als dies in anderen Programmiersprachen der Fall ist.

Alle Objekte in JavaScript unterstützen so genannte *expandable Properties*. Es handelt sich hierbei um Eigenschaften, welche dynamisch zur Laufzeit angelegt werden. Einmal angenommen, man benötigt eine Eigenschaft `Nachname` in einem Objekt. Diese wurde aber dort nicht definiert. In JavaScript ist es nun möglich das Objekt einfach um diese Eigenschaft zu erweitern indem der Name der Eigenschaften an das Objekt angehängt wird obwohl er dort (eigentlich) nicht existiert. Dieser neuen Eigenschaft kann man dann Werte zuweisen und mit dieser arbeiten, als wäre diese bereits zuvor definiert worden. So ist in JavaScript das folgende Konstrukt problemlos umsetzbar ([Listing 2](#)).

Wie dem Listing gut zu entnehmen ist, sind innerhalb der Funktion `Person` keine Variablen oder Funktionen angelegt worden. Weder `Nachname`, `Vorname` noch `Personalnummer`. Trotzdem ist es kein Problem diese Variablen zur Laufzeit anzusprechen, obwohl es sie eigentlich nicht gibt. Den Eigenschaften `Nachname`, `Vorname` noch `Personalnummer` ►

lassen sich Werte zuweisen und diese können auch wieder ausgelesen werden. Eine ausdrückliche Deklaration wie in anderen Sprachen ist somit nicht erforderlich. Diese Besonderheit/Komfort von JavaScript schafft aber auch Probleme. So lässt sich mit dieser Methode problemlos eine Variable anlegen (zum Beispiel durch einen Tippfehler) welche im restlichen Programm nicht vorhanden ist. Das kann die Fehlersuche natürlich erheblich erschweren. Deshalb sollte diese Vorgehensweise nur mit Bedacht verwendet werden.

Vielleicht ist es Ihnen aufgefallen? An einigen Stellen wird das Schlüsselwort *this* verwendet. Mit dem *this*-Schlüsselwort ist es möglich, innerhalb einer Klasse / einem Objekt auf die zugehörigen Bestandteile direkt zuzugreifen. Also einen direkten Bezug herzustellen. Auch JavaScript kennt das *this* Schlüsselwort. In folgendem Beispiel wird *this* verwendet, um die Variable *Kontostand* abzufragen ([Listing 3](#)).

In der Funktion *Konto* wurde die Variable *Kontostand* angelegt. Zur Abfrage der Variablen wurde die Funktion *getKontostand* implementiert. Innerhalb der Funktion wird die Variable *Kontostand* mittels des *this* Schlüsselwortes direkt referenziert.

## Konstruktoren

Bei Anlage eines neuen Objekts wird automatisch eine mit dem Objekt verknüpfte Funktion aufgerufen. Diese Funktion trägt denselben Namen wie das Objekt und wird in dem Moment aufgerufen, in welchem das *new* Schlüsselwort zur Anwendung kommt ([Listing 4](#)). Um über den Konstruktor Variablen zur Initialisierung zu übergeben, müssen diese im Konstruktor benannt werden. Im Beispiel wird dem Konstruktor von *Konto* die Zahl 88888 übergeben. Die Übergabe findet innerhalb der Klammern nach dem Namen statt. Sobald die Zeile ausgeführt wird, wird innerhalb der Funktion *Konto* der dann in *\_kontostand* enthaltene Wert der mit *this* referenzierten Variablen *Kontostand* zugewiesen.

Das Konzept der Klassen ist, wie bereits erwähnt, in JavaScript unbekannt. Da in der Regel nur Klassen voneinan-

## Listing 3: Das this-Schlüsselwort

```
JavaScript
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        var mein_Konto = new Konto();
        var stand = mein_Konto.getKontostand();
    };
})();

function Konto() {
    this.Kontostand = 99999;
    this.getKontostand = function () {
        return this.Kontostand;
    }
}
```

der erben können gibt es in JavaScript an dieser Stelle ein Problem. Eine Klasse wird in JavaScript deshalb als eine einfache Funktion mit dem Schlüsselwort *function* erstellt. Vererbung wird in JavaScript mit Hilfe der Prototypen-Funktion realisiert. Aber, was bedeutet das? Das folgende Vater / Sohn Beispiel soll das Ganze einmal verdeutlichen ([Listing 5](#)).

Was ist hier passiert? Es wurde eine Funktion *Vater* erstellt, welche eine Variable *Nachname* enthält. Diese Variable wurde mit dem Namen *Schmidt* initialisiert. Zusätzlich wurde die Funktion *getNachname* hinzugefügt mit welcher der Inhalt der Variablen abgefragt werden kann. Soweit so gut. Außerdem wurde ein weitere Funktion *Kind* welche über keinerlei Code verfügt geschrieben. Klar, das *Kind* soll alles vom *Vater* erben. Es ist aber keine Ableitung *Kind - Vater* erkennbar!?

Diese Ableitung wird erst zur Laufzeit mithilfe der Eigenschaft *prototype* realisiert. Mit *prototype* wird (wie es der Na-

## Listing 2: Eigenschaften

```
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        var person = new Person();
        person.Nachname = "Bleske";
        person.Vorname = "Christian";
        person.Personalnummer = 123456;
        var test_nachname = person.Nachname;
        var test_vorname = person.Vorname;
        var test_Personalnummer = person.Personalnummer;
    };
})();

function Person() {
}
```

## Listing 4: Konstruktoren

```
var konto = new Konto();
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        var mein_Konto = new Konto(88888);
        var stand = mein_Konto.getKontostand();
    };
    app.start();
})();

function Konto(_kontostand) {
    this.Kontostand = _kontostand;
    this.getKontostand = function () {
        return this.Kontostand;
    }
}
```

### Listing 5: Vererbung

```
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        Kind.prototype = new Vater();
        var sohn = new Kind();
        var test_nachname = sohn.getNachname();
    };
})();
function Vater() {
    this.Nachname = "Schmidt";
    this.getNachname = function () {
        return this.Nachname;
    }
}
function Kind() {
}
```

me schon vermuten lässt) eine Vorlage für das aktuelle Objekt festgelegt. Diese Vorlage ist die *Vater*-Funktion. Nach dieser Zuweisung kann jetzt problemlos ein neues *Kind*-Objekt (*sohn*) erzeugt werden. Dieses Objekt verwendet die in *prototype* abgelegte Vorlage (Funktion *Vater*) als Vorlage für das neue Objekt und so erhält der *Sohn* den *Nachnamen* des *Vaters* beziehungsweise die Funktion *getNachname* kann abgerufen werden. An dieser Stelle wird der Unterschied von JavaScript zu anderen Programmiersprachen besonders deutlich: Während die Vererbung in anderen Sprachen durch das Klassenkonzept realisiert wird, geschieht dies in JavaScript durch das verändern von Objekten.

In JavaScript ist es möglich, Funktionen und Eigenschaften zu überschreiben. Das bedeutet, eine Funktion die in einem Objekt vorhanden ist, kann in einem abgeleiteten Objekt unter demselben Namen erneut implementiert werden. In JavaScript ist es jederzeit möglich, Elemente zu überschreiben ([Listing 6](#)). Im Beispiel wird in der Funktion *Vater* innerhalb der Funktion *getName* lediglich der Nachname (Schmidt) definiert. Später im Programm muss neben dem Nachnamen aber zusätzlich noch der Vorname ausgegeben werden. Wie wird das gemacht? Der Funktion *getName* wird einfach ein neuer Wert zugewiesen. In diesem Fall eine neue Funktion. Die Funktion wird so neu geschrieben, dass dann die erweiterte Zeichenkette zurückgegeben wird.

JavaScript kennt weder das Konzept der Namespaces, bekannt aus C#, noch das der Packages aus Java. Also gibt es

### Listing 6: Überschreiben

```
(function () {
    //Quellcode entfernt
    app.onactivated = function (eventObject) {
        //Quellcode entfernt
        var n_Vater = new Vater();
        n_Vater.getName = function() {
            return "Hans Schmidt"
        };
        var test_name = n_Vater.getName();
    };
    app.start();
})();
function Vater() {
    this.Name = "Schmidt";
    this.getName = function () {
        return this.Name;
    }
}
```

keine (implementierte) Möglichkeit den Quellcode vernünftig zu strukturieren? Doch, hier gibt es einen Workaround welchen man verwenden kann. Das folgende Listing zeigt ein Beispiel zur Strukturierung von Quellcode in JavaScript:

```
var MeineBibliothek = {
    A: function() {
    },
    B: function() {
    }
}
// Anwendung:
MeineBibliothek.A();
MeineBibliothek.B();
```

Das Konzept im Listing beruht (abermals) darauf die Funktionen in Variablen zusammenzufassen und so eine Strukturierung des Codes zu erreichen.

### Fazit

Wie Sie sehen, kann auch in JavaScript objektorientiert programmiert werden ([Bild 2](#)). Die grundlegenden Bausteine zur Programmierung mit Objekten sind also auch in JavaScript vorhanden.



**Christian Bleske**

ist Autor, Trainer und Entwickler mit dem Schwerpunkt Client / Server und mobile Technologien. Erreichbar ist er unter [cb.2000@hotmail.de](mailto:cb.2000@hotmail.de)

### Link zum Thema

- Online-Entwicklungsumgebung für JavaScript  
<https://jsfiddle.net>



Bild: Shutterstock / Visual Generation

#### ZUKÜNSTIGE FEATURES VON JAVASCRIPT

# Fünfstufiger Prozess

ECMAScript, der Standard hinter der Programmiersprache JavaScript, wird stetig um sinnvolle Features erweitert.

Als der Entwickler-Community kommen immer wieder Vorschläge, um die Sprache JavaScript sinnvoll zu erweitern. Prinzipiell durchläuft ein Proposal dabei einen fünfstufigen Prozess (Stage 0 bis Stage 4), um endgültig in den ECMAScript-Standard aufgenommen zu werden (siehe Tabelle oder für weitere Details die Beschreibung unter <https://tc39.es/process-document/>). TC39, das Technical Committee 39, welches hinter dem ECMAScript-Standard steht und diesen kontinuierlich weiterentwickelt, verwaltet alle diese Proposals in dem GitHub-Repository unter <https://github.com/tc39/proposals>.

Im Folgenden werden sechs besonders interessante beziehungsweise besonders nützliche Features vorgestellt, die bereits Stage 3 in dem TC39-Prozess erreicht haben und daher aller Wahrscheinlichkeit nach in naher Zukunft offiziell in den Standard aufgenommen werden. Es sind dies *Optional Chaining*, *Nullish Coalescing*, *Static Class Fields*, *Class Fields*, *Private Fields* und *Top-level await*. Beim Zugriff auf verschachtelte Eigenschaften oder Methoden eines Objekts ist

es bislang relativ aufwändig sicherzustellen, dass alle Eigenschaften beziehungsweise Methoden innerhalb der Hierarchie auch vorhanden sind. Nehmen wir als Beispiel folgendes Modell für ein *user*-Objekt:

```
const user =  
{  
    firstName: 'Max',  
    lastName: 'Mustermann',  
    address: {  
        street: 'Musterstraße',  
        number: 12345  
    }  
}
```

Angenommen, man möchte nun eine Funktion implementieren, die ein solches *user*-Objekt entgegennimmt und die hinterlegten Informationen auf die Konsole ausgibt. Dabei soll die Ausgabe sowohl für Objekte mit Eigenschaft *address* als

### Listing 1: Optional Chaining

```
const print = (user) =>
{
  // ...
  const street = user.address?.street;
  const number = user.address?.number;
  // ...
}
```

auch für Objekte ohne diese Eigenschaft funktionieren. Innerhalb der Funktion muss man also vor dem Zugriff auf die geschachtelten Eigenschaften `street` und `number` zunächst überprüfen, ob überhaupt die Eigenschaft `address` vorhanden ist. Lässt man diese Überprüfung dagegen weg, kommt es beim Zugriff zu einem Fehler, wenn die Eigenschaft `address` nicht vorhanden ist.

Für die Überprüfung wird oft der logische UND-Operator `&&` verwendet und auf diese Weise schrittweise das Vorhandensein von Eigenschaften sichergestellt:

```
if (user && user.address && user.address.street) {
  // Eigenschaft "street" ist vorhanden
}
```

Insgesamt ist dies jedoch sehr viel Boilerplate-Code, der sich daraus ergibt, dass die Überprüfung die gesamte Hierarchie durchzieht, man sich also Schritt für Schritt in der Hierarchie vorantasten muss, um sicherzustellen, dass eine in der Hierarchie tiefer liegende Eigenschaft (oder Methode) existiert. Und je tiefer die Hierarchie, desto aufwändiger die Überprüfung, desto mehr Bedingungen in dem booleschen Ausdruck und desto mehr Boilerplate-Code.

### Proposal Optional Chaining

Das Proposal *Optional Chaining* (<https://github.com/tc39/proposal-optional-chaining>), schlägt diesbezüglich den sogenannten Optional Chaining Operator `?` vor. Dieser lässt sich beim Zugriff auf eine Eigenschaft direkt hinter diese setzen und prüft implizit, ob es die Eigenschaft gibt. Ist dies der Fall,

### Listing 2: Optional Chaining bei Methoden

```
const number = service
  .getUser()?
  .address?
  .number;
```

wird auf die in der Hierarchie als nächstes angegebene Eigenschaft zugegriffen, ansonsten der Zugriff abgebrochen:

```
const v = object?.property?.property2?.property3;
```

Das Beispiel von weiter oben würde also unter Verwendung von *Optional Chaining* wie in Listing 1 aussehen. Der Boilerplate-Code mit den hintereinander geschalteten booleschen Ausdrücken fällt weg und der Code wird um einiges lesbarer.

Ebenfalls erwähnenswert: das Ganze funktioniert nicht nur beim Zugriff auf Eigenschaften, sondern auch beim Zugriff auf Methoden. Der Optional Chaining-Operator wird in diesem Fall einfach hinter den Methodenaufruf platziert (Listing 2). Liefert die entsprechende Methode als Rückgabewert `undefined`, wird der weitere Zugriff abgebrochen.

### Nullish Coalescing

Ein anderer Fall, mit dem man es als JavaScript-Entwickler häufig zu tun hat, ist die Überprüfung von Werten auf `null` und der davon abhängigen Zuweisung von Default-Werten. Bislang kommt hierzu in der Regel der bedingte ternäre Operator zum Einsatz (Listing 3):

```
const v = object.property
  ? object.property
  : 'defaultValue';
```

Um dies weiter zu vereinfachen, schlägt das sogenannte Nullish Coalescing-Proposal (<https://github.com/tc39/proposal-nullish-coalescing>) einen Operator vor, der diese Überprüfung verkürzt. Mit dem Null Coalescing-Operator (bestehend aus zwei unmittelbar aufeinander folgenden Fragezeichen) ►

### Listing 3: Default-Werte

```
const user = {
  lastName: 'Mustermann',
  address: {
    street: 'Musterstraße',
    number: 12345
  }
}
const firstName = user.firstName ? user.firstName :
'Max';
```

### Listing 4: Null coalescing operator

```
const user = {
  lastName: 'Mustermann',
  address: {
    street: 'Musterstraße',
    number: 12345
  }
}
// Null coalescing operator
const firstName = user.firstName ?? 'Max';
console.log(firstName); // "Max"
```

chen), den der ein oder andere Entwickler schon aus Sprachen wie C# oder PHP kennen dürfte, soll das Ganze wie folgt aussehen ([Listing 4](#)):

```
const v = object.property ?? 'defaultValue';
```

Liefert der Zugriff auf `object.property` einen anderen Wert als `null`, wird dieser Wert der Variable `v` zugewiesen, ansonsten der Default-Wert `defaultValue`.

### Static Class Fields

Während die in ES2015 eingeführte Klassensyntax unter anderem auch Unterstützung für statische Methoden brachte, ist es bislang nicht möglich, explizit statische Eigenschaften zu definieren. Genauer gesagt: statische Eigenschaften können nicht auf die gleiche Weise wie statische Methoden (und aus anderen Sprachen wie Java gewohnt) über das Schlüsselwort `static` innerhalb des Klassenkörpers definiert werden. Stattdessen erfolgt die Definition statischer Eigenschaften bislang wie in [Listing 5](#) zu sehen umständlich außerhalb der Klasse, wodurch wiederum die Übersichtlichkeit und die Lesbarkeit leiden, weil die Zusammengehörigkeit von Methoden und Eigenschaften nicht auf den ersten Blick ersichtlich ist.

Dank dem `Static class features`-Proposal (<https://github.com/tc39/proposal-static-class-features>) soll es zukünftig jedoch möglich sein, statische Eigenschaften auf die gleiche Weise wie statische Methoden zu definieren, sprich innerhalb des Klassenkörpers ([Listing 6](#)).

Genau wie statische Eigenschaften können auch Objekteigenschaften (also nicht-statische Eigenschaften) bislang

nicht im Klassenkörper definiert werden und müssen stattdessen innerhalb von Objektmethoden definiert werden. Üblicherweise verwendet man dabei die `constructor()`-Methode:

```
class Person {
  constructor() {
    this.firstName = 'Max';
    this.lastName = 'Mustermann';
  }
}
```

Laut dem `Class Fields`-Proposal (<https://github.com/tc39/proposal-class-fields>) soll es zukünftig jedoch möglich sein, Objekteigenschaften direkt im Klassenkörper zu definieren:

```
class Person {
  firstName = 'Max';
  lastName = 'Mustermann';
}
```

Bisher ist es in JavaScript nicht möglich, private Objekteigenschaften zu definieren. Einen Modifizierer wie `private`, den man aus anderen Sprachen wie Java kennt, gibt es in JavaScript nicht. Stattdessen greift man üblicherweise auf die Konvention zurück, Eigenschaften, die privat sein sollen, mit einem vorangehenden Unterstrich beginnen zu lassen.

Das allerdings sorgt nicht dafür, dass die Eigenschaft wirklich `private` ist und verhindert nicht, dass trotzdem von außerhalb der entsprechenden Klasse auf die Eigenschaft zugegriffen werden kann (für weitere Techniken zur Emulierung pri-

### Listing 5: Emulierung statischer Eigenschaften

```
class Person {
  static create() {
    return {
      firstName: Person.firstName,
      lastName: Person.lastName
    };
  }
}

// Statische Eigenschaften
Person.firstName = 'John';
Person.lastName = 'Doe';

const john = Person.create();
console.log(john);
// { firstName: 'John', lastName: 'Doe' }

Person.firstName = 'Jane';

const jane = Person.create();
console.log(jane);
// { firstName: 'Jane', lastName: 'Doe' }
```

### Listing 6: Statische Eigenschaften

```
class Person {
  static firstName = 'John';
  static lastName = 'Doe';

  static create() {
    return {
      firstName: Person.firstName,
      lastName: Person.lastName
    };
  }
}

const john = Person.create();
console.log(john);
// { firstName: 'John', lastName: 'Doe' }

Person.firstName = 'Jane';

const jane = Person.create();
console.log(jane);
// { firstName: 'Jane', lastName: 'Doe' }
```

[Bild: https://node.green](https://node.green)

Node.js ESNEXT Support		Nightly!										requires harmony flag	Created by William Kapke								
NODE	kangax's compat-table applied only to Node.js	13.0.0	12.9.1	12.8.1	12.4.0	11.15.0	10.16.3	10.8.0	10.3.0	9.11.2	8.9.4	12% complete	12% complete	12% complete	12% complete	10% complete	9% complete	9% complete	3% complete	1% complete	1% complete
<b>candidate (stage 3)</b>																					
<b>globalThis</b>																					
"globalThis" global property is global object	?	Yes	Yes	Yes	Yes	Flag	No	No	No	No	No	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
"globalThis" global property has correct property descriptor	?	Yes	Yes	Yes	Yes	Flag	No	No	No	No	No	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
<b>WeakReferences</b>																					
WeakRef minimal support	?	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
Finalizers minimal support	?	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
<b>instance class fields</b>																					
public instance class fields	?	Yes	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
private instance class fields basic support	?	Yes	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
private instance class fields initializers	?	Yes	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
computed instance class fields	?	Yes	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
<b>static class fields</b>																					
public static class fields	?	Yes	Yes	Yes	Yes	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
private static class fields	?	Yes	Yes	Yes	Yes	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
computed static class fields	?	Yes	Yes	Yes	Yes	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
<b>optional chaining operator (?)</b>																					
optional property access	?	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
optional bracket access	?	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
optional method call	?	Error	Error	Error	Error	Error	Error	Error	Error	Error	Error	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag
nullish coalescing operator (??)																					
numeric separators	?	Yes	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Yes	Yes	Yes	Flag	Flag	Flag	Flag	Flag	Flag	Flag

Node.js-Support  
zukünftiger  
Features  
(Bild 1)

vater Eigenschaften sei an dieser Stelle auf das Buch »Professionell entwickeln mit JavaScript: Design, Patterns und Praxistipps für Enterprise-fähigen Code« des Autors verwiesen).

In dem oben genannten Proposal *Class Fields* werden private Eigenschaften wie in Listing 8 zu sehen deklariert, das heißt, dem Namen einer Eigenschaft ein #-Symbol vorne ange stellt. Der Zugriff ist dann nur innerhalb der Klasse beziehungsweise innerhalb von Methoden der Klasse möglich.

Übrigens: die Einführung eines Schlüsselwortes *private*, die man an dieser Stelle eher vermuten würde, war aus verschiedenen Gründen nicht möglich, im Detail nachzulesen unter [https://github.com/tc39/proposal-class-fields/blob/master/PRIVATE\\_SYNTAX\\_FAQ.md](https://github.com/tc39/proposal-class-fields/blob/master/PRIVATE_SYNTAX_FAQ.md). Kaum ein Feature wurde in JavaScript – abgesehen von der Klassensyntax – in den

letzten Jahren so ausgebaut wie die asynchrone Programmierung. Musste man vor ein paar Jahren noch die Callback-Hell fürchten oder auf Promise-Bibliotheken wie q (<https://github.com/kriskowal/q>) zurückgreifen, wurden mit ES2015 Promises auch nativ eingeführt und später dann mit ES2017 durch die Schlüsselwörter *async* und *await* ergänzt.

Damit wurde asynchrone Programmierung in JavaScript so einfach wie nie und asynchrone Aufrufe innerhalb des Codes lassen sich fortan so einfach wie synchrone Aufrufe in Code fassen:

```
const URL = '...';
async init(url) {
  const config = await fetchConfig(url);
```

### Listing 7: Konvention für private Eigenschaften

```
class Person {
  constructor() {
    this._firstName = 'Max';
    this._lastName = 'Mustermann';
  }

  toString() {
    return `${this._firstName} ${this._lastName}`;
  }
}

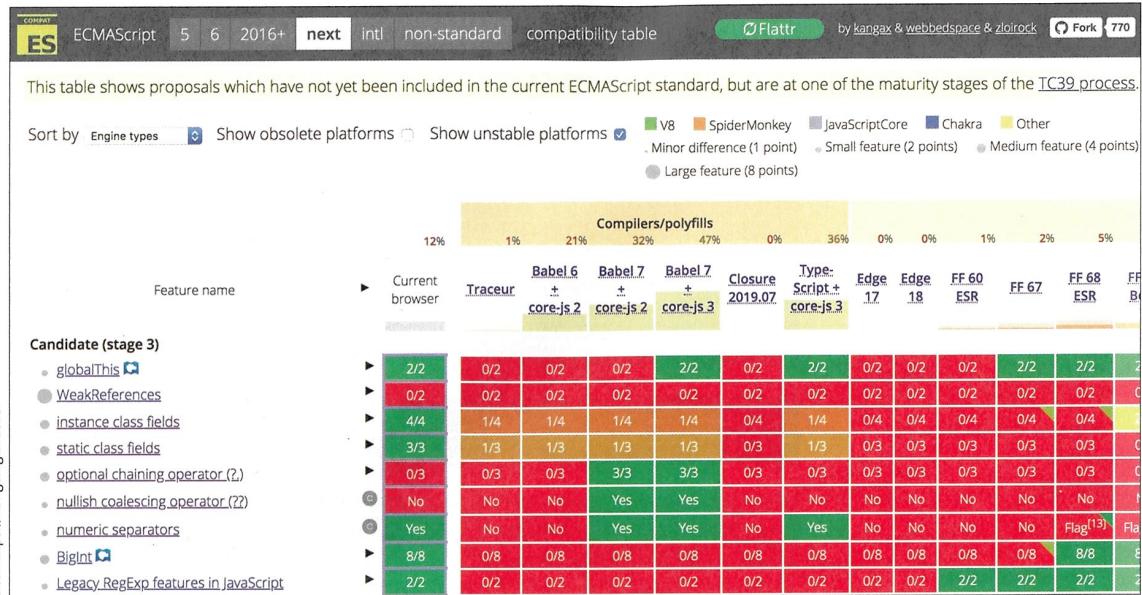
const person = new Person();
console.log(person._firstName); // "Max"
console.log(person._lastName); // "Mustermann"
console.log(person.toString());
// "Max Mustermann"
```

### Listing 8: Private Eigenschaften

```
class Person {
  #firstName = 'Max';
  #lastName = 'Mustermann';

  toString() {
    return `${this.#firstName} ${this.#lastName}`;
  }
}

const person = new Person();
console.log(person.firstName); // undefined
console.log(person.lastName); // undefined
// console.log(person.#firstName); // SyntaxError
// console.log(person.#lastName); // SyntaxError
console.log(person.toString());
// "Max Mustermann"
```

Bild: <https://kangax.github.io>

```
    await initDatabase(config);
}

init(URL);
```

Eines bleibt bei der Verwendung von `async` und `await` allerdings etwas unschön: und zwar ist es bislang nur möglich, dass Schlüsselwort `await` innerhalb einer mit `async` gekennzeichneten Funktion zu verwenden. Diese Einschränkung hat allerdings zur Folge, dass `await` nicht auf oberster Ebene des Codes (also auf Top Level) verwendet werden kann. Aus diesem Grund muss man bislang gezwungenenmaßen auf das IIFE-Pattern (Immediately-invoked Function Expression) zurückgreifen, bei dem eine Funktion erzeugt und unmittelbar aufgerufen. Wieder unnötiger Boilerplate-Code für eine eigentlich triviale Angelegenheit:

```
const URL = "...";
(async () => {
  const config = await fetchConfig(URL);
  await initDatabase(config);
})();
```

Das Proposal *Top Level Await* schlägt nun vor, dass `await` auf oberster Ebene ohne eine umgebende mit `async` markierte Funktion verwendet werden kann. Dadurch entfällt der durch das IIFE-Pattern entstandene Boilerplate-Code und der Code wird schlanker und lesbarer:

```
const URL = "...";

const config = await fetchConfig(URL);
await initDatabase(config);
```

Die Sprache JavaScript entwickelt sich kontinuierlich weiter und wird regelmäßig um weitere Features ergänzt. Die Klas-

sensyntax nähert sich nach und nach der beispielsweise aus Java bekannten Syntax an (auch wenn Klassen in JavaScript natürlich konzeptionell und technisch anders einzuordnen sind als in Java).

Die Definition von statischen Methoden und Eigenschaften sowie von Objektmethoden und Objekteigenschaften wird wie gesehen in zukünftigen Versionen von JavaScript vereinheitlicht und intuitiver.

## Fazit

Unabhängig von der Stufe, in der sich ein bestimmtes Feature offiziell befindet, ist es oft so, dass das Feature von der ein oder anderen Laufzeitumgebung bereits heute unterstützt wird. Statische Eigenschaften, private Eigenschaften und die neue Syntax für die Definition von Objekteigenschaften werden von Node.js beispielsweise seit Version 12.4.0 unterstützt (<https://node.green/#ESNEXT>), ebenso von Google Chrome (Bild 1). Optional Chaining und Nullish Coalescing sind momentan dem Transpiler Babel in Version 7 vorbehalten (Bild 2).

Eine gute Übersicht zum aktuellen Stand bezüglich Support und Implementierungsfortschritt findet sich beispielsweise unter <https://kangax.github.io/compat-table/esnext/>. Wer sich bezüglich neuer Features auf dem Laufenden halten möchte, wirft am besten einen Blick auf das GitHub-Repository unter <https://github.com/tc39/proposals>.



**Philip Ackermann**

ist CTO der Cedalo AG und Autor mehrerer Fachbücher und Fachartikel über Java, JavaScript und Node.js. Seine Schwerpunkte liegen in der Konzeption und Entwicklung von Node.js- und JEE-Projekten in den Bereichen Industrie 4.0 und Internet of Things.