Vorlesungsreihe
EwA

JavaScript –Erweiterungen und Alternativen am Beispiel von

TypeScript und CoffeeScript

Prof. Dr.-Ing. Thomas Wiedemann
email: wiedem@informatik.htw-dresden.de

HOCHSCHULE FÜR TECHNIK UND WIRTSCHAFT DRESDEN (FH)
Fachbereich Informatik/Mathematik

Gliederung

- JavaScript-Alternativen und Erweiterungen
  - Motivation
- **TypeScript**
  - Kurze Historie
  - Wesentliche Erweiterungen
- **CoffeeScript**
  - Historie - Eigenschaften – Beispiele
- Zusammenfassung und Bewertung

Quelle(n) (und weitere im Text):

[1] https://www.typescriptlang.org (TypeScript-Homepage)

[2] https://www.tutorialspoint.com/typescript/index.htm (Tutorial)

[3] http://coffeescript.org/ CoffeeScript-Homepage und Tutorial

Motivation zur Erweiterung von JavaScript

- JavaScript Nutzung war 1995 vorrangig zur In-Browsermanipulation gedacht
- schnelle, unkomplizierte Integration in HTML und geringer Entwicklungsaufwand waren die Hauptziele
- Eine vollwertige Programmiersprache war eher zweitrangig
- Mit der zunehmenden Entwicklung komplexer Applikationen und dem wachenden Servereinsatz kommen die Nachteile immer deutlicher zum Vorschein
  - https://secure.phabricator.com/book/phabflavor/article/javascript_pitfalls/
  - http://nrn.io/view/javascript-common-pitfalls/view/block-statements
  - http://jonathan.bergknoff.com/journal/some-obscure-javascript-pitfalls

- Zur Bewältigung größere Projekte entstanden daher Erweiterungen und effizientere Syntaxformen

Übersicht zu TypeScript

- entwickelt von Microsoft ab 2012
- **TypeScript (TS) ist ein Superset von JavaScript**
  - jedes gültige JavaScript-Programm ist auch ein (syntaktisch) gültiges TypeScript-Programm
  - TypeScriptprogramm werden von einem TS-Compiler in normales JavaScript umgewandelt,

2 Hauptziele

- deutlich strengeres Typ-System
- Verfügbarmachung von Funktionen aus neueren ECMAScript-Versionen (welche noch nicht in jedem Browser vorhanden sind)
- Interne Diskussion (eher aus Entwickler-Typensicht)
  - Mit der zunehmenden Verwendung von JS im Backendbereich steigt auch die Anzahl der klassischen Backendprogrammierer (und diese kennen / wollen noch echte Klassen / Vererbung …)

TypeScript – Überblick II

Historie

- Erscheinungsjahr: 2012
- Entwickler: Anders Hejlsberg (Chefentwickler von Microsoft (ehemals verantwortlich für Turbo Pascal und Delphi, später zu MS gewechselt)

Softwaretyp

- Opensource unter Apache – Lizenz

- Website: www.typescriptlang.org

Beschaffung

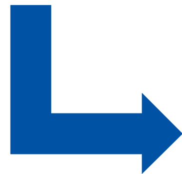
- via npm: **npm install -g typescript**
- oder Installation als VisualStudio-Plugin (Visual Studio 2015 and Visual Studio 2013 Update 2 enthalten es bereits)
- auch integrierte Quelltextanalyse in MS-Produkten (siehe MS Code)

Allgemeine Vorgehensweise

- Wie bisher JS-Codeentwicklung mit den neuen TypeScript-Optionen
- Übersetzung (Abwärtskompilierung) zu Standard-JavaScript

TypeScript prog.ts

```
function greet(person : string) {
return "Hello, " + person; }
var user = "Tom";
console.log(  greet(user) );
```



tsc   prog.ts

JavaScript prog.js

```
function greet(person            ) {
    return "Hello, " + person;
}
var user = "Tom";
console.log( greet(user) );
```

TypeScript – Warnungen und Fehlermeldungen

- Wie bisher JS-Codeentwicklung mit den neuen TypeScript-Optionen
- Übersetzung (Abwärtskompilierung) zu Standard-JavaScript

TypeScript prog.ts

```
function greet(person : string) {
return "Hello, " + person; }
var user : number  = 123 ;
console.log(  greet( user ) );
```

```
tsc   prog.ts
```

```
function greet(person             ) {
    return "Hello, " + person;
}
var user = 123 ;
console.log( greet( user ) );
```

error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.

JavaScript prog.js

Achtung: Der JS-Code ist trotzdem ohne Fehler ausführbar !!!!

Neue Sprachoptionen von TS : Datentypen und Zuweisungen

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Basic types

TS-Basisdatentypen

- : number - für alle Zahlenwerte (JS stellt alles als 64 bit float dar !)
- : string - für Texte
- : boolean - für Wahrheitswerte

Spezielle Typen und Values

- any - entspricht wieder einem beliebigen Typ (wie in JS var)
- void - kein Typ (z.B. KEIN Rückgabewert in Funktionen)
- Value "undefined" - wie in JS für nicht initialisierte Variablen

Beispiele für in TS erlaubte Datendeklarationen

- var t1 : string ="text"; // volle Deklaration als Text-Variable
- var t1 ="text"; // wie JS – Typ ergibt sich aus Zuweisung
- var t1 ; // value = undefined Typ = any !

Neue Sprachoptionen von TS : Komplexere Datentypen

- https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Arrays …

Komplexere Datentypen bauen auf den JS-Arrays auf

- TS-Array erlauben nur einen Datentyp

```
var werte:number[4] = [1,2,3,4 ];
```

- Tupel erlauben in Analogie zu JS-Arrays beliebige Typen

```
var mytuple = [10,"Hello"]; //create a tuple
```

- Enums

```
enum Direction { Up = 1, Down, Left, Right }
```

- auch generische Ansätze (analog zu Template-Systemen)

```
function myfunc<T>(arg: T): T { return arg; }
var output1= myfunc<string>("myString");
var output2 = myfunc("myString"); // Typ ergibt sich aus Parameter
```

Neue Sprachoptionen von TS : Datentypen und Zuweisungen

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Basic types

Bestimmung des Typs

- typeof( ) – Operator liefert String mit Typangabe

```
var num = 12;
console.log(typeof num);    //output: number
```

- auch direkt zur Verarbeitung von any-Typangaben

```
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}'.`); }

padLeft("Hello world", 4); // returns " Hello world"
```

Neue Sprachoptionen von TS : Zuweisungen mit let

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Var declarat.

Problem mit var

Der Scope von JS- var in Blöcken entspricht nicht dem anderer Progsprachen ,
da auch NACH dem Block eine im Block definierte Variable weiterhin gültig ist:

```
for (var i = 0; i < 10; i++)
   { /* ... */ }
 console.log(i);     //  zeigt   10 an -  weil  i  noch verfügbar ist
```

Neuer korrekter Block-Scope-Bereich mit let (analog zu var)

```
for ( let  j = 0; j < 10; j++)
   { /* ... */ }
 console.log( j  );     //  erzeugt  TS-Fehlermeldung:  Can not find  name 'j'
```

- interessanter Seiteneffekt: bei gleicher Namensgebung benennt
  TS die 2. Variable um (i_1)

Neue Sprachoptionen von TS : Funktionsdefinitionen

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Functions

Functions

- alle bisherigen JS-Definition werden unterstützt (auch ohne Typparameter)
- zusätzlich auch typisierte Aufruflisten mit Typprüfung

```
function add(x: number, y: number): number   { return x + y; }
var  c =    add (  2 , 3 ) ;    // ok
var  d =    add (  “ttt” , “kkk”  ) ;    // TS- Fehler
// error TS2345: Argument of type 'string' is not assignable to parameter of
// type 'number'.
```

- analog auch mit anonymen Funktionen

```
var res = function(a: number ,b: number ) { return a*b; };
console.log(res(12,2))
```

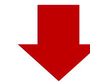
- auch als Fat-arrow (bzw. Lambda –Function)

```
 var add10 = (x:number)  => {       x = 10 + x ;     console.log(x)  }
console.log(add10 (12 ))
```

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Functions

Functions

- bei JS sind alle Parameter optional !
- TS prüft die Anzahl genau und unterstützt optionale und Rest-Parameter



```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
let result1 = buildName("Bob", "Adams");   // correct call
let result2 = buildName("Bob");                 // ok with ?  In parameter list
let result3 = buildName("Bob", "Adams", “junior"); // TS – error
// TS2346: Supplied parameters do not match any signature of call target.
```

Neue Sprachoptionen von TS : Zusätzliche Funktionsparameter

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Functions

Functions

- Mit **…parname** können auch beliebig viele Parameter erfasst werden

```
function buildName2(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}
let name1 = buildName2("Joseph", "Samuel", "Lucas", "junior");
```

Generierter JS-Code

```
function buildName2(firstName) {
    var restOfName = [];
    for (var _i = 1; _i < arguments.length; _i++)
    { restOfName[_i - 1] = arguments[_i];
    }
    return firstName + " " + restOfName.join(" ");
}
```

Neue Sprachoptionen von TS : Klassen und Objekte

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Classes
- mit class sind OO-Definitionen in sehr großer Ähnlichkeit zu C++ / C# möglich

TypeScript-Code

```
class  Car {
     engine:string;  //field
    //constructor
   constructor(engine:string) {
      this.engine = engine
   }
disp():void {   // normal class function
 console.log("Engine is : "+this.engine)
  }
}  // end of class definition
var obj = new Car("VW"); // create
//access the field
console.log("Engine :  "+obj.engine)
obj.disp(); // call object method
```

Generierter JS-Code

```
var Car = (function () {
    //constructor
    function Car(engine) {
        this.engine = engine;
    }
   // object function
    Car.prototype.disp = function () {
  console.log("Engine is  :   " +
this.engine);     };
    return Car;
}());
var obj = new Car("VW"); // create
console.log("Engine:" + obj.engine);
//access the function
obj.disp();
```

Neue Sprachoptionen von TS : Klassen-Vererbung

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Classes
- Auch die Vererbung mit extends hat sehr große Ähnlichkeit zu C++ / Java

TypeScript-Code

```
class Shape {
   Area:number
   constructor(a:number) {
      this.Area = a
   }
}
class Circle extends Shape {
disp():void {
      console.log("Area of the circle:
"+this.Area)
   }
}
var obj = new Circle(223);
obj.disp()
```

Generierter JS-Code

```
var __extends = (this && this.__extends) ||
function (d, b) {  for (var p in b)
  if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b)
: (__.prototype = b.prototype, new __());
};
var Shape = (function () {
    function Shape(a) {
        this.Area = a;
    }   return Shape;
}());
var Circle = (function (_super) {
    __extends(Circle, _super);
    function Circle() {
        _super.apply(this, arguments);
    }  ...
var obj = new Circle(223);
obj.disp();
```

Neue Sprachoptionen von TS : Klassen-Zugriffspezifier

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Classes
- Es werden auch private / protected und weitere Zugriffsspezifier unterstützt

TypeScript-Code

```
class Encapsulate {
   str:string = "hello"
   private str2:string = "world"
}

var obj = new Encapsulate()
console.log(obj.str)     //accessible
console.log(obj.str2)   // TS Error as str2
is private
```

Generierter JS-Code

```
var Encapsulate = (function () {
    function Encapsulate() {
        this.str = "hello";
        this.str2 = "world";
    }
    return Encapsulate;
}());
var obj = new Encapsulate();
console.log(obj.str); //accessible
console.log(obj.str2); // geht in JS
```

Weitere Zugriffsspezifier:

- readonly - nur über Init oder Konstruktor sind Änderungen möglich
- abstract - keine direkte Ableitung möglich !

Neue Sprachoptionen von TS : Interfaces

- Details / Quelle: TypeScript-Handbuch
  https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Interfaces

Interfaces

- dienen zur Definition **wiederverwendbarer** Typdefinitionen
- sind nur in TS relevant – keine Kompilierung zu JS

```
interface IPerson
 { firstName:string,
   lastName:string,
    sayHi: ()=>string
 }
 var customer: IPerson = {
      firstName:"Tom",
      lastName:"Hanks",
      sayHi: ():string => {   return "Hi there"}
   }
```

Neue Sprachoptionen von TS : Module und Namespaces

- bis TypeScript 1.5. gab es interne und externe Module
- ab TS 1.5. entsprechen interne Module den Namespaces
- https://www.typescriptlang.org/docs/tutorial.html => Handbook -> Namespace

Namespaces

- bilden einen eigen Scope für die Bezeichner
- ausserhalb benötigte Funktionen müssen mit **export** deklariert werden !

```
namespace Validation {
    export interface StringValidator { isAcceptable(s: string): boolean; }
```

Module

- sind sehr ähnlich zu namespaces erlauben, jedoch zusätzlich die Definition von Abhängigkeiten (sind ab ECMAScript 2015 auch Sprachbestandteil und werden in node als Standard-Methode zur Codestrukturierung empfohlen)
- `declare module “MyModule1" {    // in file   module1.d.ts`
  `export function fn(): string; }    …`
- `/// <reference path=“module1.d.ts" />        // Ref. zu 1. Modul  in 2. File`
  `import * as m from “MyModule1";`

CoffeeScript – Überblick

Historie

- Erscheinungsjahr: 2009
- Entwickler: Jeremy Ashkenas (auch Backbone.js framework)
- http://www.linuxjournal.com/content/introducing-coffeescript

Softwaretyp

- Opensource unter MIT – Lizenz
- Website(n): http://coffeescript.org/
  http://courseware.codeschool.com/coffeescript_slides.pdf

Haupteigenschaften

- wird ebenfalls übersetzt zu Standard JavaScript
- durch sehr spezielle Syntaxformen („syntaktischer Zucker") ist der CoffeeScript-Code ca. 30 bis 50% KLEINER als JS-Code
- besonders gut für Anfänger und WebDesigner OHNE C/Java-Vorbelastung geeignet

CoffeeScript – Installation / Einbindung in eigene Projekte

Beschaffung

- via npm: **npm install -g coffee-script**

Aufruf

- **coffee --compile test1.coffee test.js**
- **coffee --compile --output lib/ src/**

ach mit Komandozeilenoptionen für

- **Watch – Überwachung von Source und autom. Neukompilierung**
- **Join - Zusammenführen von Source und Libs**
- und viele weitere Optionen

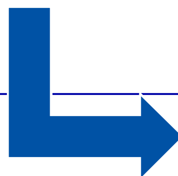
CoffeeScript- Beispiele : Variablen und Basisbefehle

- Codeentwicklung mit CoffeeScript-Syntax
- Übersetzung (Abwärtskompilierung) zu Standard-JavaScript

CoffeeScript test.coffee

```
# Assignment:
number = 42
 opposite = true
# Conditions:
number = -42 if opposite
# Functions:
square = (x) -> x * x
# Arrays:
 list = [1, 2, 3, 4, 5]
```

coffee



JavaScript prog.js

```
var cubes, list, number, opposite, square;
number = 42;
opposite = true;

if (opposite) { number = -42; }

square = function(x) { return x * x; };

list = [1, 2, 3, 4, 5];
```

- **Allgemeine Regeln: Whitespaces und Einrückungen sind sehr relevant für die Syntax ! (bei Verstößen werden entsprechende Syntaxfehler gemeldet !)**
  - tw. minimalistische Syntax

CoffeeScript- Beispiele : sehr code-effiziente Syntax

CoffeeScript

```
# test for existence:
alert "I knew it!" if elvis?

#loops
countdown = (num for num in [10..1])

# Eat lunch.
eat food for food in ['toast', 'cheese']
```

JavaScript

```
if (typeof elvis !== "undefined" && elvis
!== null)
     { alert("I knew it!"); }

var countdown, num;
countdown = (function()
 { var i, results; results = [];
 for (num = i = 10; i >= 1; num = --i)
   { results.push(num); }
   return results; })();

ref = ['toast', 'cheese'];
for (j = 0, len = ref.length; j < len; j++)
{ food = ref[j];
 eat(food);
}
```

CoffeeScript- Beispiele : Arrays und Objekte

CoffeeScript

```
#arrays  (Newline ersetzt Trenner)
bitlist = [ 1, 0, 1
            0, 0, 1 ]
# Objects:
math =
  root: Math.sqrt
  square:  square
  cube: (x) ->  x * square x

#auch geschachelt – Indents beachten!
kids =
   brother:
      name: "Max"
      age: 11
    sister:
      name: "Ida"
      age: 9
```

JavaScript

```
var bitlist, math, kids; // var ganz oben
bitlist = [1, 0, 1, 0, 0, 1,];
// Objects
math = {
  root: Math.sqrt,
  square: square,
  cube: function(x) {
    return x * square(x); }
 };
kids = {
 brother:
   { name: "Max",
      age: 11 },
    sister:
    { name: "Ida",
    age: 9 }
};
```

Zusammenfasssung

- Die Erweiterungen sind prinzipiell interessant, jedoch mit sehr unterschiedlicher Zielrichtung:

TypeScript vom Microsoft

- Bringt JS näher an die bisherige Backend-Programmierung mit C++, C# und JAVA und vermeidet viele JS-Fehler
- Es ist zu erwarten dass viele TS-Optionen zukünftig in die neuen JS-Versionen einfließen werden (man arbeitet damit quasi in der Zukunft von JS!)

CoffeeScript

- ist syntaktisch sehr interessant, jedoch bei Fehlern nicht sehr tolerant (tw. unklare Fehlermeldungen und auch Compiler-Abstürze)
- Ergebnis eines von einem einzelnen Programmierers betriebenen Projekts (???)

- **Die Anwendung ist sicher auch eine persönliche Entscheidung (und eine Funktion der bisherigen Programmierpraxis)**
- **Beide JS-Erweiterungen (und andere Optionen wie Googles DART) sollten beobachtet werden !**