Objektorientiertes JavaScript

Christoph Fabritz
dm121506@fhstp.ac.at

http://goo.gl/JZQxNW

Inhalt

- JavaScript
- Objektorientierung
- OO in JavaScript
- Literatur

JavaScript

Interpretiert / gescriptet

Dynamische Typisierung

Objektorientiert, aber klassenlos

Mögliche Stile: prozedural, funktional, objektorientiert

JavaScript

Interpretiert / gescriptet

Es wird nicht in eine maschinennahe Sprache kompiliert. Da es keinen Compiler gibt, kann dieser auch nicht auf Fehler hinweisen.

JavaScript

Dynamische Typisierung

Variablen können im Laufe des Programms verschiedene Typen annehmen. Erst zur Laufzeit ist klar, welchen Typ eine Variable hat.

JavaScript

+ Programm schnell erstellt
+ Viele Wege zum Ziel

aber

- Schnell unübersichtlich
- Viele Wege zum Ziel
- Fehleranfällige Programme

Sicheres JavaScript

JSLint
http://www.jslint.com/

=

Syntaxchecker und Validator
+
Erzwingt Stilkonventionen

Objektorientierung

Eine Möglichkeit (von vielen), Kode zu **organisieren** und **übersichtlicher** zu gestalten.

Vorsicht:
Muss nicht auf alle Probleme passen. Möglichkeiten abwägen und den **Objekte-Blick vermeiden**. Ist mit einem **höheren Aufwand** verbunden.

Objektorientierung

Verbundene Konzepte

Kapselung

Polymorphie

Vererbung

Objektorientierung

Kapselung

Verbergen von Implementierungs-Details, Abschottung von Variablen und Funktionen von Außen. Es gibt alleinig eine definierte Schnittstelle. (Blackbox)

Objektorientierung

Kapselung

+ Innere Implementierung kann jeder Zeit verändert werden

+ Übersichtlicher

+ Vermeidet unerwünschte Interaktions-Möglichkeiten

Objektorientierung

Polymorphie

„Vielgestaltigkeit“
Ein Objekt/Methode kann je nach Verwendung/Klasse, unterschiedliches Verhalten zeigen.

Z.B.: ToString()

Objektorientierung

Vererbung

Abgeleitete Klassen besitzen ebenfalls die Methoden und Attribute der Basisklasse.
„ist ein“-Verhältnis

OO in JavaScript

In JavaScript durch Prototypisierung realisiert und verzichtet somit auf Klassen.

Will man vergleichbare Konzepte der klassischen OO-Programmierung, muss man sie sich selbst implementieren.

Unterschiedliche Ansätze zur Realisierung von OO-Konzepten.

OO in JavaScript

Klassen

Es gibt kein „class“-Schlüsselwort. Es werden Funktionen verwendet, um Klassen zu realisieren.

Definieren Klassen- (statische) und Objektmethoden (brauchen ein Objekt), mit unterschiedlicher Zugänglichkeit (private, public, ...).

OO in JavaScript

Klassen & Kapselung

```
var Person = (function () {

  function Person(gender) {

  }


  return Person;
}());

var p1 = new Person('male');
```

Konstruktor

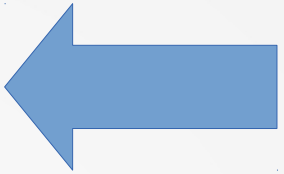
Klassendefinition

Verwendung
des Konstruktors

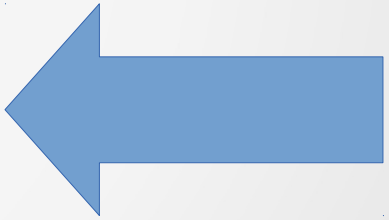
OO in JavaScript

Klassen & Kapselung

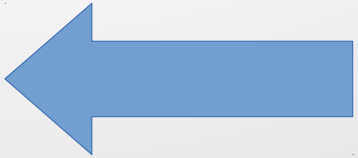
```
var Person = (function () {
  var private_klassen_variable = 'text';

  function Person(gender) {


  }


  Person.public_klassen_function = function () {};

  return Person;
}());

var p1 = new Person('male');


Person.public_klassen_function();
```



Private Klassenvariable



Public Klassemethode



Verwendung

OO in JavaScript

Klassen & Kapselung

```
var Person = (function () {
  var private_klassen_variable = 'text';

  function Person(gender) {
    var private_variable = 23;
    this.public_function = function() {};
  }

  Person.prototype.public_function2 = function () {};
  Person.public_klassen_function = function () {};

  return Person;
}());

var p1 = new Person('male');
p1.public_function();
p1.public_function2();
Person.public_klassen_function();
```

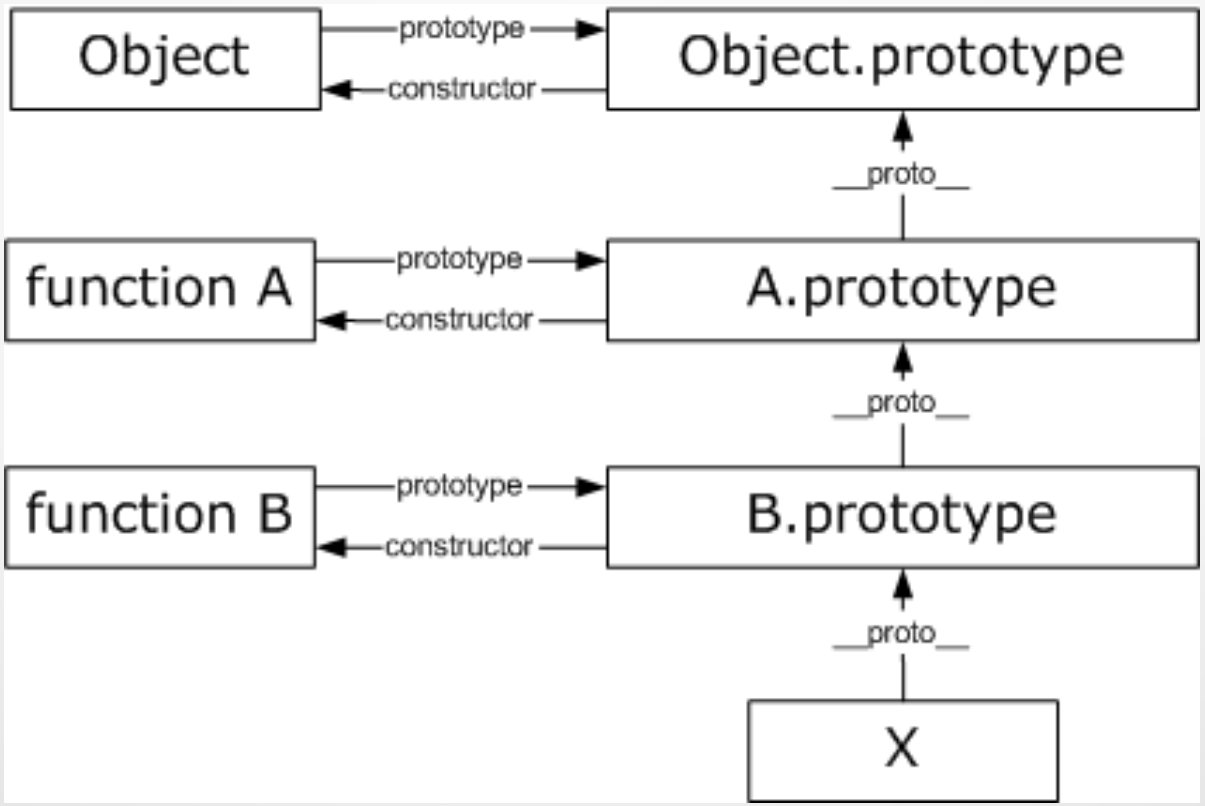
Private Variable und public Methode

Alternative Schreibweise

Verwendung

OO in JavaScript

Vererbung
Prototypische Vererbung



OO in JavaScript

Vererbung

```
function Person(name) {}          // Superklasse


function Author(name, books) {    // SubKlasse

}
```

```
var a1 = new Author('Ein Name', ['Buch 1', 'Buch 2']);
console.log(a1 instanceof Person);  // false
console.log(a1 instanceof Author);  // true
```

OO in JavaScript

Vererbung

```
function Person(name) {}          // Superklasse


function Author(name, books) {    // SubKlasse
  Person.call(this, name);        // Konstruktor der Superklasse
}




Author.prototype = new Person();        // Person als Prototype
Author.prototype.constructor = Author; // Konstruktor zurücksetzen

var a1 = new Author('Ein Name', ['Buch 1', 'Buch 2']);
console.log(a1 instanceof Person);  // true
console.log(a1 instanceof Author);  // true
```

OO in JavaScript

Vererbung

```
function Person(name) {}          // Superklasse
Person.prototype.say = function () { console.log('in Person'); };

function Author(name, books) {    // SubKlasse
  Person.call(this, name);        // Konstruktor der Superklasse
}
Author.prototype.say = function () {   // Überschreiben
  Person.prototype.say.call(this);     // Funktion der Superklasse
  console.log('in Author');
};

Author.prototype = new Person();       // Person als Prototype
Author.prototype.constructor = Author; // Konstruktor zurücksetzen

var a1 = new Author('Ein Name', ['Buch 1', 'Buch 2']);
console.log(a1 instanceof Person);  // true
console.log(a1 instanceof Author);  // true
```

OO in JavaScript

Vererbung

Bequemlichkeits-Funktion:
http://goo.gl/ARY8SL

```
extend(Author, Person);
```

OO in JavaScript

Interfaces

„Program to an interface, not an implementation.“

Spezifiziert, welche Methoden ein Objekt haben soll, aber nicht, wie diese implementiert sein sollen.

OO in JavaScript

Interfaces

+ Selbst-Dokumentierender Kode

+ Wiederverwendbarkeit

+Können an andere Programmierer weitergereicht werden, um vorab die Anforderungen festzulegen

OO in JavaScript

Interfaces

durch Kommentare

```
/*
  interface Composite {
    function add(child);
    function remove(child);
    function getChild(index);
  }
  interface FormItem {
    function save();
  }
*/

var CompositeForm = function() { // implements Composite, FormItem
  ...
};
```

OO in JavaScript

Interfaces

durch Duck-Typing http://goo.gl/hRl8af

```
var Composite = new Interface('Composite',['add','remove','get']);
var FormItem = new Interface('FormItem', ['save']);

var CompositeForm = function() { // implements Composite, FormItem
  ...
};

function addForm(formInstance) {
  Interface.ensureImplements(formInstance, Composite, FormItem);
  ...
}
```

Literatur (1)

Pro JavaScript Design Patterns
Apress
Ross Harmes, Dustin Diaz

http://goo.gl/1RZzzs

Literatur (2)

Secrets of the JavaScript Ninja
Manning
John Resig, Bear Bibeault

http://goo.gl/1Mqpoi

Danke für Ihre Aufmerksamkeit