JavaScript 201: Patterns

Objekte, die Zweite

- Objekte können auch Funktionen als Properties haben
- Diese funktionieren dann wie Methoden, d.h. `this` ist an das Objekt gebunden über das sie aufgerufen werden

```
var obj = {
    field: 10,
    log: function() {
        console.log(this.field);
    }
};

obj.log(); // 10
```

Call / Apply

- call und apply sind Methoden auf Funktionen
- Erlauben das freie Binden an einen anderen Kontext

```
var global = {
    field: "Reingelegt"
};
obj.log.call(global);
// => ???
```

Singleton

- Wie gesehen, kann man in JavaScript Objekte auch ohne einen Typ erzeugen
- In klassenbasierten Sprachen muss man auch dafür einen Typ erzeugen und darauf achten, dass man nur eine Instanz erzeugen kann
- Das Pattern dazu heißt `Singleton`
- In JavaScript braucht man dieses Pattern nicht
- Manchmal möchte man allerdings auch mehrere Objekte mit dergleichen oder ähnlichen Struktur erzeugen

Klassen mit JavaScript

- Klassen und Konstruktoren sind Mechanismen, um mehrere, strukturell gleiche oder ähnliche Objekte zu erzeugen
- Auch in JavaScript können eigene Typen definiert werden
- Einfachvererbung ist ebenso möglich
- Der Mechanismus ist nicht direkt in die Sprache eingebaut
- Stattdessen benutzen wir Best-Practice-Patterns
- Grundlage ist die prototypische Vererbung

Prototypen

- Jedes Objekt hat zusätzlich eine Referenz auf seinen Prototyp
  - `__proto__` oder `Object.getPrototypeOf()` in neueren Browsern
- `Object` hat keinen Prototypen, ist aber Prototyp aller anderen Objekte
- Lesende Property-Zugriffe können transitiv an Prototypen delegiert werden
- Dies heißt prototypische Vererbung

Setzen des Prototypen

Der Prototyp kann nicht direkt, aber durch Aufruf von `new` gesetzt werden

```
/** @constructor */
function Person(name) {
    this.name = name;
}

// Methode
Person.prototype.getName = function() {
    return this.name;
};


var olli = new Person('Olli');
olli.getName() === 'Olli';
```

Ablauf eines Konstruktoraufrufs mit `new`

1. ein leeres, neues Objekt wird erzeugt
2. die Konstruktor-Funktion hat ein Property `prototype`, dies wird als Prototyp des neuen Objekts verwendet
3. `this` wird an dieses neue Objekte gebunden
4. die Konstruktor-Funktion wird aufgerufen (mit `this` gebunden)
5. das neue Objekt wird implizit zurückgegeben (wenn die Funktion kein explizites `return` hat)

”Typsystem”

Ein Objekt ist `instanceof` aller seiner Prototypen

```
var olli = new Person('Olli');
olli.__proto__ === Person.prototype;
olli.constructor === Person;
olli instanceof Object;
olli instanceof Person;
```

Übung 3: Klassen

1. Schreibe eine Klasse für Person
   - Lasse im Konstruktor die drei bekannten Parameter für `name`, `alter` und `geschlecht` zu
   - Mache aus allen Funktionen, die auf Personen arbeiten, Methoden
2. Erzeuge ein Objekt vom Typ Person und rufe Methoden darauf auf

Vererbungshierarchien

- Das verwirrendste Thema in JavaScript
- Klassen-Hierarchien und Instanzen nutzen beide Prototypische Vererbung
- Klassen-Hierarchien werden einmal aufgebaut und als Prototypen der Instanzen verwendet
- Klassen-Hierarchien werden ebenso über Prototypen erstellt
- Aufruf von Super-Konstruktoren und Super-Methoden über call / apply
- Alternative zu Klassen-Hierarchien sind Mixins

Mixins

```
function Logable(name) {
    this.name = name;
}
Logable.prototype.log = function(msg) {
    console.log(this.name + ": " + msg);
}

function Person(name, age) {
    Logable.call(this, name);
    this.age = age;
}
_mixin(Person, Logable);

var olli = new Person('Olli');
console.log(olli instanceof Person); // true
console.log(olli instanceof Logable); // false
olli.log('Hi');
```

Implementierung von _mixin

```
function _mixin(_sub, _super) {
    for (var p in _super.prototype) {
        if (p === 'constructor') {
            continue;
        }
        _sub.prototype[p] = _super.prototype[p];
    }
}
```

Klassen-Hierarchien über Prototypen

```
function Logable(name) {
    this.name = name;
}
Logable.prototype.log = function(msg) {
    console.log(this.name + ": " + msg);
}

function Person(name, age) {
    Logable.call(this, name);
    this.age = age;
}
_extends(Person, Logable);

var olli = new Person('Olli');
console.log(olli instanceof Person); // true
console.log(olli instanceof Logable); // true
olli.log('Hi');
```

Implementierung von _extends

```
function _extends(_sub, _super) {
    _sub.prototype = new _super();
}
```

Achtung: Vereinfachte Version mit Schwächen

Static

```
// Statisches Feld
Person.lieblingsName = "Olli";

// Statische Funktion, greift nicht auf this zu
Person.getLieblingsName = function() {
    return Person.lieblingsName;
};

Person.lieblingsName === "Olli";
Person.getLieblingsName() === "Olli";
```

Module

Module in JavaScript werden über `Closures` realisiert

Closure in einem Satz

Eine innere Funktion hat immer Zugriff auf alle Variablen und Parameter ihrer äußeren Funktion, *auch wenn diese äußere Funktion bereits beendet ist*.

Frei nach *Douglas Crockford*

Beispiel Closure

```
function outer() {
    var used = "Olli";
    var unused = "Weg";
    return (function() {
        return "Text: " + used;
    });
}

var inner = outer();
console.log(inner());
```

Closure Definition

Eine Closure ist eine spezielle Art von Objekt, welche zwei Dinge kombiniert

1. Eine Funktion
2. die Umgebung in welcher diese Funktion erzeugt wurde - diese Umgebung besteht aus allen lokalen Variablen und Parametern, die sichtbar waren als die Closure erzeugt wurde

Aus der Definition auf MDN

Revealing Module Pattern

```
var humanresources = {};
(function () {
    function InternalStuff() {
    }

    function Person(name) {
        this.name = name;
        this.inner = new InnerStuff();
    }

    humanresources.Person = Person;
})();
```

Sichtbarkeit bei Revealing Module Pattern

```
var olli = new humanresources.Person('Olli');
olli.name === 'Olli';
// TypeError: undefined is not a function
new humanresources.InternalStuff();
```

Namespaces

```
var eu = eu || {};
eu.zeigermann = eu.zeigermann || {};
eu.zeigermann.person = eu.zeigermann.person || {};
(function (person) {
    // Constructor
    function Person(name) {
        this.name = name;
    }

    // Factory
    function create(name) {
        return new Person(name);
    }

    // Export der Factory-Methode
    person.createPerson = create;
})(eu.zeigermann.person);
```

Import

```
eu.zeigermann.main = {};
(function (main, person) {
    function run() {
        var olli = person.createPerson("Olli");
        console.log(olli.name); // => "Olli"

        console.log(person.Person); // => undefined
    }
    main.run = run;
})(eu.zeigermann.main, eu.zeigermann.person);

eu.zeigermann.main.run();
```

Übung 5: Module

1. Schreibe ein Modul, in das du die vorhandenen Typendefinitionen verschiebst. Dieses Modul soll
   - nach außen nur den `Customer`-Typen exportieren
2. Schreibe den aufrufenden Code so um, dass er mit den neuen Modulen arbeitet
3. Optionale Zusatzübung:
   - erstelle ein Utility-Modul, das zumindest die _extends-Funktion enthält
   - erstelle ein Main-Modul, das den aufrufenden Code enthält

Modul-Systeme

Es gibt zwei grundsätzlich unterschiedliche Modul-Systeme als Defacto-Standard

- AMD
  - Module werden nicht blockierend und potentiell asynchron geladen
  - Module werden über bower oder manuell installiert
  - Verwendung auf Client-Seite
  - Default-Implementierung: RequireJS
- CommonJS
  - Module werden blockierend und synchron geladen
  - Module werden über npm installiert
  - Verwendung auf Server-Seite
- Beide Modul-Systeme können über r.js / Browserify auch auf Server / Client benutzt werden

Definieren eines commonJS Moduls bei node

```
// in der Datei "eater.js"
var eatThis = function (name) {
    console.log(name);
};
exports.eatThis = eatThis;
```

Benutzen eines commonJS Moduls bei node

```
var eaterModule = require("eater");

eaterModule.eatThis(name);
```

Definieren eines AMD Moduls mit requireJS

```
// Ein Modul pro Datei:
// js/modules/accounting.js
define(function () {
    return {
        getIdNumberForName: function(name) {
            // ...
        }
    };
});
```

Benutzen eines AMD Moduls mit requireJS

```
require(['js/modules/accounting'], function (Accounting) {
    var name = // ...
    var id = Accounting.getIdNumberForName(name);
    // ...
});
```