**WBE-Praktikum 5**

# Prototypen, Tests mit Jasmine

## Aufgabe 1: Node REPL

Überlegen Sie, was folgende Anweisungen auf der Node REPL liefern werden und überprüfen Sie anschliessend, ob Ihre Annahmen stimmen:

```
> let F = function (n) { this.a = n }
> let f = function () { return this.a }
> let fs = function () { "use strict"; return this.a }
> let value = new F(12)
> value.a
> f()
> fs()
> fs.call({ a: 11, b: 22 })
> F(99)
> a
> let obj = Object.create({ f })
> obj
> obj.a = "yeah"
> obj
> obj.f()
```

Ein paar Fragen dazu:

- Wie sind die unterschiedlichen Ergebnisse bei den Aufrufen von `f()` und `fs()` zu erklären?
- Warum ist es keine gute Idee, die Funktion `F` ohne *new* aufzurufen?
- Das Objekt *obj* enthält nur ein Attribut *a*. Warum kann `obj.f()` aufgerufen werden?

Hier zwei weitere Aufrufe mit Ergebnis:

```
> Object.getOwnPropertyNames(obj)
[ 'a' ]
> Object.getOwnPropertyNames(Object.getPrototypeOf(obj))
[ 'f' ]
```

## Aufgabe 2: Tests mit Jasmine

Jasmine ist ein Test-Framework für JavaScript, das sowohl mit *Node.js* als auch im Browser eingesetzt werden kann. Am besten legen Sie ein neues Projekt an und ergänzen es um Jasmine mit Code- und Testbeispielen ($ soll hier das Promptzeichen des Terminals sein):

```
$ cd wbelabs                        // wo auch immer Sie Ihre Projekte haben
$ mkdir jasmine_demo                // Projektverzeichnis anlegen
$ cd jasmine_demo                   // ins Verzeichnis wechseln
$ npm init                          // paar Angaben zum Projekt machen
```

Im letzten Schritt werden einige Angaben zum Projekt abgefragt. Lassen Sie zunächst einfach leer, was Sie nicht wissen. Die Angaben landen in der Datei *package.json*, die Sie auch später ergänzen können. Nun zu Jasmine:

```
$ npm install --save-dev jasmine    // Jasmine installieren
$ npx jasmine init                  // Jasmine initialisieren
$ npx jasmine examples              // Beispiel mit Tests hinzufügen
```

Falls *npx* nicht installiert ist, können Sie es folgendermassen installieren. Dazu sind ggf. Administratorrechte erforderlich (unter Mac und Linux stellen Sie *sudo* voran, unter Windows am besten das Terminal als Administrator starten):

```
$ npm install -g npx
```

Weitere Hinweise für die ersten Schritte mit Jasmine finden Sie auf dieser Seite:
https://jasmine.github.io/pages/getting_started.html

**Aufgaben:**

- Sehen Sie sich die Datei *package.json* sowie die ganze Verzeichnisstruktur des Projekts an. Wozu dient das Verzeichnis *node_modules*?
- Unter *lib/jasmine_examples* finden Sie den zu testenden Code. Die beiden „Klassen" *Player* und *Song* sind hier in Form von Konstruktorfunktionen angelegt.[1] Untersuchen Sie den Aufbau von *Player* und *Song*. Ergänzen Sie den Konstruktor *Song* um einen Parameter *title*, der neu erstellten Objekten als Attribut hinzugefügt wird.
- Erstellen Sie auf der obersten Ebene in Ihrem Projekt eine Datei *index.js*, in der Sie *Song* und *Player* importieren (schauen Sie in die Folien), einen *Song* anlegen sowie einen *Player*, der diesen Song gerade spielt. Falls `player.isPlaying==true` soll der Titel des Songs ausgegeben werden.
- Ihr Script können Sie mit `node index.js` starten. Oder mit `npm start`, wenn Sie die *scripts*-Angabe in *package.json* um `"start": "node ."` ergänzen.

[1] Das war zumindest bisher so. In der aktuellen Version von Jasmine wurden die Konstruktoren durch Klassen ersetzt.

- Untersuchen Sie die Tests unter *spec*. Diese sollten grösstenteils auch ohne genauere Kenntnisse von Jasmine verständlich sein. Führen Sie die Tests aus:
  ```
  $ npx jasmine spec/jasmine_examples/PlayerSpec.js
  ```
- Bauen Sie verschiedene Fehler in den Code ein und starten Sie weitere Testläufe.

Um die Ergebnisse der Testläufe im Browser anzuzeigen, verwenden Sie am besten eine aktuelle Version von *Jasmine Standalone*, welches Sie hier laden können:
https://github.com/jasmine/jasmine/releases

- Testen Sie auch die Web-Variante von Jasmine.

Wenn man den gleichen Code und die gleiche Test-Suite sowohl unter Node.js als auch im Browser verwenden möchte, taucht ein Problem auf: Node-js verwendet standardmässig ein anderes Modulsystem als JavaScript im Browser. Lösungshinweise zu diesem Problem finden Sie im Anhang.

## Aufgabe 3: Funktion unter Test (Abgabe)

In dieser Aufgabe soll eine Funktion „test-driven" entwickelt werden. Da es uns hier vor allem um das Testen geht, verwenden wir eine relativ einfache Funktion: Die Funktion *parseToProto(json, proto)* soll einen String, welcher ein Objekt in JSON-Format enthält, parsen und das resultierende Objekt mit dem Prototyp *proto* verbinden und zurückgeben.

```
> let proto = { category: "animal" }
> let obj = parseToProto('{"type":"cat","name":"Mimi","age":3}', proto)
> obj.age
3
> obj.category
"animal"
```

- Schreiben Sie Tests für *parseToProto*.
- Implementieren Sie die Funktion, so dass Sie die Tests erfüllt.
- Geben Sie die Funktion *parseToProto* ab. Kopieren Sie Ihre Tests in auskommentierter Form in die abzugebende Datei.

**Abgabe**

Geben Sie die Funktion *parseToProto* ab.

| | |
|---|---|
| Abgabeserver: | https://radar.zhaw.ch/python/UploadAndCheck.html |
| Name Praktikum: | WBE6 |
| Dateiname: | parse-to-proto.js |
| Funktionsname: | parseToProto |
| Export im Script: | `module.exports = { parseToProto}` |

## Aufgabe 4: Objekt mit Prototyp (Vertiefung)

In ECMAScript 5 wurde *Object.create* eingeführt, das es erlaubt, ein neues Objekt anzulegen, welches ein bestehendes Objekt als Prototyp hat: https://devdocs.io/javascript/global_objects/object/create

Eine solche Funktion man auch selbst implementieren, wie dieses Beispiel aus dem Buch «JavaScript: The Good Parts» (Douglas Crockford, 2008) zeigt:[2]

```
if (typeof Object.beget !== "function") {
  Object.beget = function (o) {
    var F = function () {};
    F.prototype = o;
    return new F();
  }
}
```

Machen Sie sich klar, wie *Object.beget* funktioniert und schreiben Sie ein paar Zeilen Code, welche *Object.beget* im Einsatz zeigen.

## Anhang: Jasmine in Node.js und im Browser

Node-js verwendet standardmässig ein anderes Modulsystem als JavaScript im Browser. Wenn man die gleichen Sources und Tests in beiden JavaScript-Umgebungen verwenden möchte, hat es sich als vorteilhaft erwiesen, das ES6-Modulsystem zu verwenden. Dazu sind zwei Anpassungen nötig.

In *package.json* folgende Zeile hinzufügen:
`"type": "module",`

In *spec/support/jasmine.json* diese Zeile eintragen:
`"jsLoader": "import",`

Fakultativ kann auch die Standalone-Version von Jasmine hinzugefügt werden, um das Ergebnis der Tests direkt im Browser anzeigen zu können. Dazu gehört das Verzeichnis *lib/jasmine* und die Datei *SpecRunner.html*. Im SpecRunner wird das Spec-File mit *type="module"* geladen, welches seinerseits den zu testenden Code importiert.

[2] *JavaScript: The Good Parts* war sicher eines der besten JavaScript-Bücher. Seither hat sich JavaScript stark verändert, so dass das Buch heute leider nicht mehr aktuell ist.