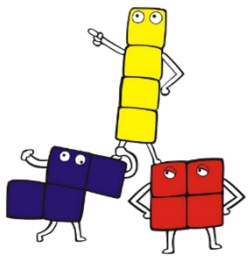
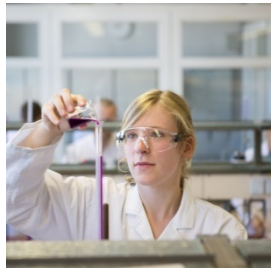
Web Engineering II

07
Modularisierung und Strukturierung

Johannes Konert

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences



Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser

optional

- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype

- Zusammenfassende Fragen
- Ausblick

„Pick-and-Challenge“

Zusammenfassende Fragen der letzten Woche

1. Wie können Sie mit express.js Werte von einer Handler-Funktion an die nächste weitergeben (obwohl next() ohne Parameter genutzt wird)?
2. Wie lautet die Definition des Begriffes Bug?
3. Welche 4 Rollen nehmen Sie als Softwarentwickler/in ein? Für welche Aufgaben?
4. Welche 5 Teile gehören zu einer Bugbeschreibung?
5. Was ist mit Schritt 2 Stabile Reproduzierbarkeit herstellen gemeint? Wie geht das?
6. Wieso ist es wichtig ein Minimized Example zu erstellen?
7. Welche drei Aspekte beachten Sie beim formulieren von Hypothesen, um zielgerichtet den Bug zu untersuchen (Detektiv)?
8. Für welche Ziele verwenden Sie Inspektion (Debugger) und wann besser Logging?
9. Wofür stehen die Logger-Level TRACE und FATAL? Wozu gibt es überhaupt verschiedene Level?
10. Welche drei Gruppen von Fällen sollten Sie stets testen (manuell oder mit UnitTests)?
11. Welche Ziele werden mit dem Unit-Testing verfolgt?
12. Wie funktioniert eine BDD oder TDD-basierte Test-Beschreibung?
13. Wozu dient das Modul `should`? Was wird damit vereinfacht in UnitTests?
14. Im Unterschied zu `jasmine/frisby` ist bei `mocha` die Testausführung asynchron. Auf welchen Code-Aufruf müssen Sie achten, um `mocha` mitzuteilen, dass ein Test fertig ist?
15. Was sagt eine Code-Coverage von 100% über die Qualität des Programmcodes aus?
16. Bonus-Frage: Wäre es sinnvoll, dass eine Programmiererin, die nicht Teil des Entwicklungsteams war, die Tests schreibt? Wenn ja, wie sollte Sie das tun? Wenn nein, warum nicht?

Wiederholung / Bonusfrage

Wäre es sinnvoll, dass eine Programmiererin, die nicht Teil des Entwicklungsteams war, die Tests schreibt? Wenn ja, wie sollte Sie das tun? Wenn nein, warum nicht?

JA, weil..

- Sicht von Außen nötig
- Zusammenhängendes System soll getestet werden
- Außenstehende sollen definieren, was funktionieren soll

NEIN, weil..

- Koordination/Abstimmung zwischen Externen und Coding-Team zu aufwändig
- Tests falsch geschrieben werden (Setup der zu testenden SW usw.)

JA, Extern sinnvoll bei Integration Testing ganzer Systeme / APIs

NEIN, Intern sinnvoll bei Unit Testing von Einzelkomponenten

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

JavaScript: Zwei Ursachen für Laufzeitprobleme vermeiden

- **Codebeispiel**

```
<script src="js/utils.js"></script>

<script src="js/models/User.js"></script>
<script src="js/models/Login.js"></script>
<script src="js/models/Contact.js"></script>
<script src="js/models/Contract.js"></script>
<script src="js/models/Customer.js"></script>
<script src="js/models/Product.js"></script>
<script src="js/models/License.js"></script>
<script src="js/models/Invoice.js"></script>
<script src="js/models/InvoiceItem.js"></script>

<script src="js/views/menu.js"></script>
<script src="js/views/login.js"></script>
<script src="js/views/loginManage.js"></script>
<script src="js/views/modal_error.js"></script>
<script src="js/views/modal_confDialog.js"></script>
```

- **Problem**
  - **Abhängigkeit von der Lade-Reihenfolge**
  - Entwickler müssen beachten, wie *.js-Inhalte aufeinander aufbauen
  - Viele Dateien werden geladen, obwohl ggf. unnötig für erste UI Anzeige

JavaScript: Zwei Ursachen für Laufzeitprobleme vermeiden

- **Codebeispiel 2**

Datei: myscript.js

```
var current = 1;
function init(){
    // some code
}

function validate(){
    // some code
}
```

Datei: otherLib.js

```
var libid = 1.2.3;
function init(){
    // some other code
}

function validate(){
    // some other code
}
```

- **Problem?**
  - **Globaler Namespace wird „verschmutzt“**
  - Konflikte mit gleichen Namen aus anderer *.js Datei
  - Die später geladene Funktion überschreibt vorherige

JavaScript: Zwei wesentliche Laufzeitprobleme vermeiden

- **Problem 1: Abhängigkeiten**
  - **Gesucht: Abhängigkeitsmanagement (Dependency Management)**
- **Problem 2: Konflikte im globalen Namensraum**
  - **Gesucht: Kapselung und Namensräume**

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

Objekt zur Kapselung

Idee

- **Namensräume durch Objekthierarchien schaffen**
- **Das Objekt kapselt zusammen, was zusammen gehört**

```
var myscript = new Object();
myscript.current = 1;
myscript.init = function(){
    // some code
};
myscript.validate = function(){
    // some code
};
…
myscript.validate();
```

Objekt Literale zur Kapselung

Idee

- **Namensräume durch Objekthierarchien schaffen**
- **Das Objekt kapselt zusammen, was zusammen gehört**

```
var myscript = { };
myscript.current = 1;
myscript.init = function(){
    // some code
};
myscript.validate = function(){
    // some code
};
…
myscript.validate();
```

```
var myscript = {
    current: 1,
    init: function () {
        // some code
    },
    validate: function () {
        // some code
    }
};
…
myscript.validate();
```

JavaScript Objekte sind wie Maps
mit <key>: <value> Paaren

Objekt Literale zur Kapselung

- **Vorteile**
  - **Kapselung des Werte und Funktionen in einem Objekt**
  - **Geschlossener Kontext („Namensraum“)**
  - **Vermeidung von Konflikten**

Die **Kapselung** von Eigenschaften und Funktionen
in Objekten ist das Minimum und
die Basis jeglicher Modularisierung.

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

JavaScript: Das Module Pattern

Schrittweise Herleitung

- Anonyme Funktionen
- Modul Parameter
- Private und Public
- Erweiterungen

JavaScript: Das Module Pattern: Herleitung

```
var myscript = {
    current: 1,
    init: function () {
        // some code
    },
    validate: function () {
        // some code
    }
};
…
myscript.validate();
```

Problem

- Immer noch globale Variable `myscript`

Idee

- Nutze Funktion drum herum als Ausführungskontext („Blockkontext“)

Quelle für Herleitung: u.A. http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html

JavaScript: Das Module Pattern: Herleitung

```
var myscript = {
    current: 1,
    init: function () {
        // some code
    },
    validate: function () {
        // some code
    }
};
…
myscript.validate();
```

```
var wrapper = function() {
    var myscript = {
        current: 1,
        init: function () {
            // some code
        },
        validate: function () {
            // some code
        }
    };

    myscript.validate();
};
wrapper();
```

Problem

- Immer noch globale Variable, nur jetzt eben `wrapper`

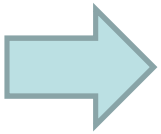
Idee

- Aufruf direkt nach der Definition der Funktion
- Dann brauchen wir keinen Namen mehr, können anonyme Funktion nutzen

JavaScript: Das Module Pattern: Herleitung

- **Anonyme Funktionen (keine Referenz zeigt auf die Funktion)**

```
var a = function () {
        // ...
}();
```



```
(function () {
        // ...
})();
```

  - Alle Funktionen und Variablen befinden sich nur in dem definiertem Kontext!
  - Die Klammern () **hinter** der Funktion sorgen für eine unmittelbare Ausführung
  - Die Klammern () **um** die Funktion machen diese Deklaration zu einem Ausdruck auch ohne Variablenzuweisung

JavaScript: Das Module Pattern: Herleitung

- **Anonyme Funktionen (keine Referenz zeigt auf die Funktion)**

```
var wrapper = function() {
    var myscript = {
        current: 1,
        init: function () {
            // some code
        },
        validate: function () {
            // some code
        }
    };

    myscript.validate();
};
wrapper();
```

```
(function() {
    var myscript = {
        current: 1,
        init: function () {
            // some code
        },
        validate: function () {
            // some code
        }
    };

    myscript.validate();
})();
```

→ **Absolut keine Nutzung des globalen Namensraumes mehr!**

Diese Nutzung von anonymen JavaScript Funktionen zur Kapselung nennt man das *Module Pattern*

JavaScript: Das Module Pattern

Schrittweise Herleitung

- Anonyme Funktionen
- Modul Parameter
- Private und Public
- Erweiterungen

Module: Parameter

```
…
(function ($, userModel) {
    // ...
})(jQuery, Models.User);
```

- Import von fremdem Kontext
- Gut sichtbare Herkunft
- Schnellerer Zugriff als über globale Namensräume
- Definition eigener Namensräume (`jQuery` → `$`)

JavaScript: Das Module Pattern

Schrittweise Herleitung

- Anonyme Funktionen
- Modul Parameter
- Private und Public
- Erweiterungen

Module verfügbar machen

- **Private und Public Methoden**

```
var module = (function () {
    var privateCounter = 1;
    function privateMethod() {
        // some code
    }
    var my = {
        instanceID: 1,
        init: function () {
            // some code
            privateMethod();
        },
        validate: function () {
            // some code
        }
    };
    return my;
})();
```

- Über das **return** wird **module** zu **my**
- Kein Zugriff auf die nicht an **my** angehängten Eigenschaften und Methoden (private)

Spoiler: private Methoden und Eigenschaften sind sinnvoll im Zusammenhang mit Closures

JavaScript: Das Module Pattern

Schrittweise Herleitung

- Anonyme Funktionen
- Modul Parameter
- Private und Public
- Erweiterungen

Module erweitern

```
var newModule = (function (my) {
    my.anotherMethod = function () {
        // added method...
    };
    return my;

})(module);
```

- `module` wird `my` und kann erweitert werden

- **Problem?**

  `module` kann `undefined` sein!

Module erweitern

```
var newModule = (function (my) {
    my.anotherMethod = function () {
        // added method...
    };
    return my;

})(module || {});
```

- `module || {}` ermöglicht es, dass auf `my` zugegriffen werden kann, sollte `module` nicht existieren

Module erweitern

- **Alternative: Methoden ersetzen**

```
var newModule = (function (my) {
    var old_init = my.init;
    my.init = function () {...};
    return my;

})(module || {});
```

  - `old_init` kann innerhalb des gesamten Blocks benutzt werden **(bspw. in Closure-Funktion my.init())**
  - **Achtung: Hier ist die Lade-Reihenfolge nicht egal, wenn mehrere Modul-Erweiterungen existieren.**

- **Was ist ein Closure?**

Ein Closure ist eine Funktion, welche auf Ihren Definitionskontext zugreift, (obwohl dieser schon beendet wurde.)

siehe u.A: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

JavaScript: Das Module Pattern

Schrittweise Herleitung

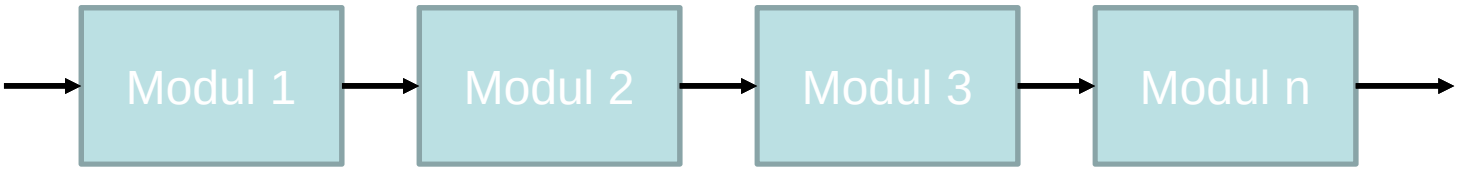
- Anonyme Funktionen
- Modul Parameter
- Private und Public
- Erweiterungen

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

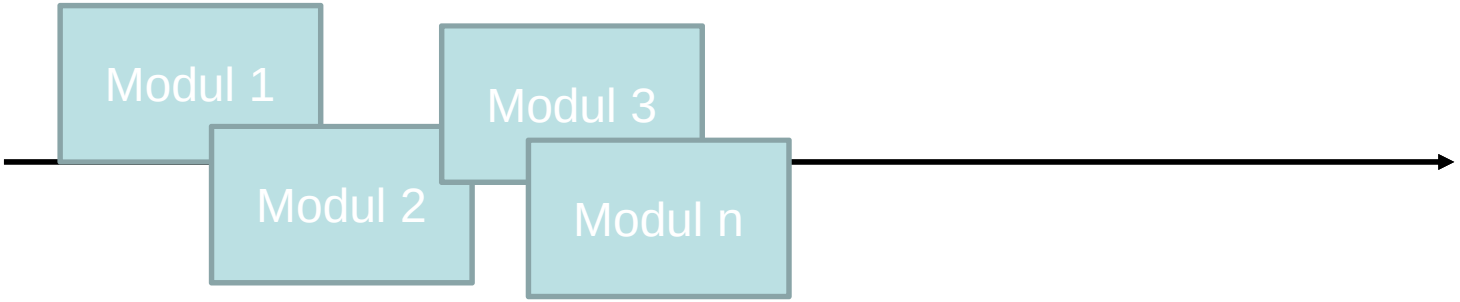
Modularisierungslösungen (JavaScript)

- **CommonJS Spezifikation**



  - **primär für Server-seitiges Modulmanagement gedacht. Browser geht auch.**

- **Asyncronous Module Definition (AMD)**



  - **eher für Browser-seitiges Modulmanagement gedacht. Server geht auch.**

CommonJS Spezifikation

- **require: eine Funktion**
  - Parameter: Modulname (ID)
  - Rückgabe: liefert die Modul-API

```
var express = require('express');
```

- **module: eine Datei**
  - Variable `require`
  - Variable `exports`
  - Variable `module`
    - `module.id`
    - `module.exports`

```
'use strict';
//...
var proto = require('./application');
var req = require('./request');
var res = require('./response');

module.exports = createApplication;
exports = module.exports;


/**  */
function createApplication() {
    //..
    return app;
}
```


Zu exports und module.exports siehe auch Erörterung unter http://stackoverflow.com/questions/16383795/difference-between-module-exports-and-exports-in-the-commonjs-module-system

CommonJS Spezifikation

- **require: eine Funktion**
  - **Parameter: Modulname (ID)**
  - **Rückgabe: liefert die exportierte API des Moduls zurück**

- **module: eine Datei**
  - **erhält eine Variable `require`, die auf die `require`-Funktion verweist**
  - **erhält eine Variable `exports`, an welche das Modul seine API anheften kann**
  - **erhält eine Variable `module`, welche auf ein Objekt verweist**
    - **Das module-Objekt hat ein Attribut `module.id`, welches der Modulname (ID) ist**
    - **module hat ein Attribut `module.exports`, die auf exports verweist und ersetzt werden darf**

Quelle: http://www.commonjs.org/ und http://wiki.commonjs.org/wiki/Modules/1.1.1

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

node.js: Was passiert bei einem `require('moduleID')` ?

1. Datei finden

- **Suche im Verzeichnis ./node_modules/moduleID/**
  - **Wenn eine package.json, lade die Datei des Abschnitts „main“**
  - **Ansonsten lade eine index.js oder index.json**
  - **Wenn nicht gefunden, dann gehe ein Verzeichnis hoch und Suche weiter (solange bis bei /node_modules/... im globalen node.js Verzeichnis)**

Quelle: https://nodejs.org/api/modules.html#modules_modules und http://stackoverflow.com/questions/7137397/module-exports-vs-exports-in-node-js

node.js: Was passiert bei einem `require('moduleID')` ?

2. Dateiinhalt ausführen und zurückliefern

- **Dateiinhalt in eigenem Namensraum (scope) ausführen**
  - **Mittels anonymer Funktion**
  - **`module.exports`, `require` und `module` als Parameter übergeben**

PseudoCode (wie require funktioniert)**

```
function require(moduleID) {
    // 1. Load the file content of moduleID
    // 2. Prepare the variable module.exports, module.id etc.
    // 3. inject loaded file content into and run:
    (function (exports, require, module) {
        // LOADED FILE CODE INJECTED HERE!
    })(module.exports, require, module);

    return module.exports;

}
```


Quelle: u.A. http://fredkschott.com/post/2014/06/require-and-the-module-system/ ** require enthält auch etwas Code für Abhängigkeitsmanagement, Events und einen Modulcache

node.js: Was passiert bei einem `require('moduleID')` ?

2. Dateiinhalt ausführen und zurückliefern

- **Dateiinhalt in eigenem Namensraum (scope) ausführen**
  - **Mittels anonymer Funktion**
  - **`module.exports`, `require` und `module` als Parameter übergeben**

Fazit

CommonJS Spezifikation

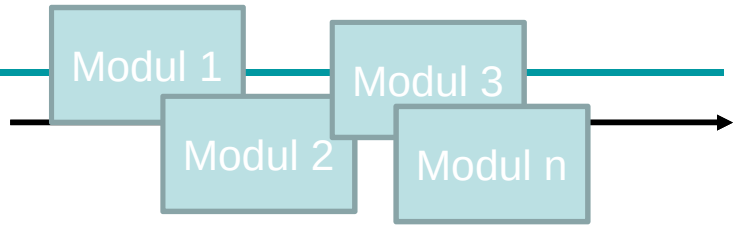
- synchrones Laden von abhängigen Modulen mittels **`require('moduleID')`**
- `moduleID` muss eindeutig sein (oder eine Pfadangabe)
- pro Moduldatei genau ein Modul, welches im Scope einer anonymen Funktion ausgeführt wird
- Nur der Inhalt von `module.exports` wird als öffentliche API dieses Moduls zurückgegeben

Quelle: u.A. http://fredkschott.com/post/2014/06/require-and-the-module-system/ ** require enthält auch etwas Code für Abhängigkeitsmanagement, Events und einen Modulcache

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

Asynchronous Module Definition (AMD)



- **require-Funktion**
  - **Parameter 1:** Array an zu ladenden Modulen
  - **Parameter 2:** Callback-Funktion mit den Modulen aus Parameter 1 als Funktions-Parameter!
  - **Rückgabe:** keine

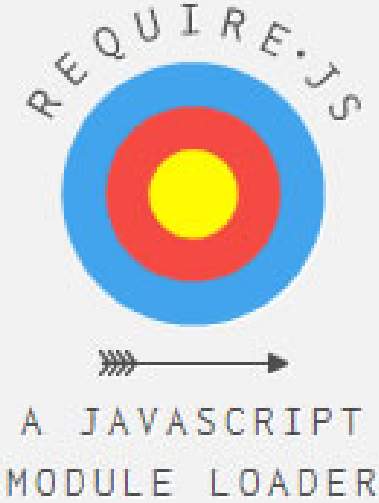
```
require(['moduleID'], function(loadedModule) {
    loadedModule.foobar();
});
```

- **define-Funktion**
  - **Parameter 1:** moduleID des zu definierenden Moduls
  - **Parameter 2:** Array an zu ladenden Modulen (Abhängigkeiten)
  - **Parameter 3:** Factory-Funktion für Modul-Inhalt (API)

```
define('moduleID', ['otherLibModuleID'], function (libModule) {
    function foo() {
        return libModule.foo();
    }
    // export (expose) foo
    return  foo;
});
```


Quelle: u.A: https://github.com/amdjs/amdjs-api/blob/master/AMD.md

Require.js - eine AMD Umsetzung



Quelle: http://www.sitepoint.com/understanding-requirejs-for-effective-javascript-module-loading/

Require.js - eine AMD Umsetzung

```
/* ---

RequireJS is a JavaScript file and module loader. It
is optimized for in-browser use, but it can be used
in other JavaScript environments, like Rhino and
Node. Using a modular script loader like RequireJS
will improve the speed and quality of your code.

IE 6+ .......... compatible ✓
Firefox 2+ ..... compatible ✓
Safari 3.2+ .... compatible ✓
Chrome 3+ ...... compatible ✓
Opera 10+ ...... compatible ✓

Get started then check out the API.

--- */
```


Quelle: http://requirejs.org/

Ein modulares Beispiel

credits.js

```
1  function getCredits(){
2    console.log("Function : getCredits");
3
4    var credits = "100";
5    return credits;
6  }
```

purchase.js

```
1  function purchaseProduct(){
2    console.log("Function : purchaseProduct");
3
4    var credits = getCredits();
5    if(credits > 0){
6      reserveProduct();
7      return true;
8    }
9    return false;
10 }
```

products.js

```
1  function reserveProduct(){
2    console.log("Function : reserveProduct");
3
4    return true;
5  }
```

`main.js` , initializes the code by calling `purchaseProduct()`

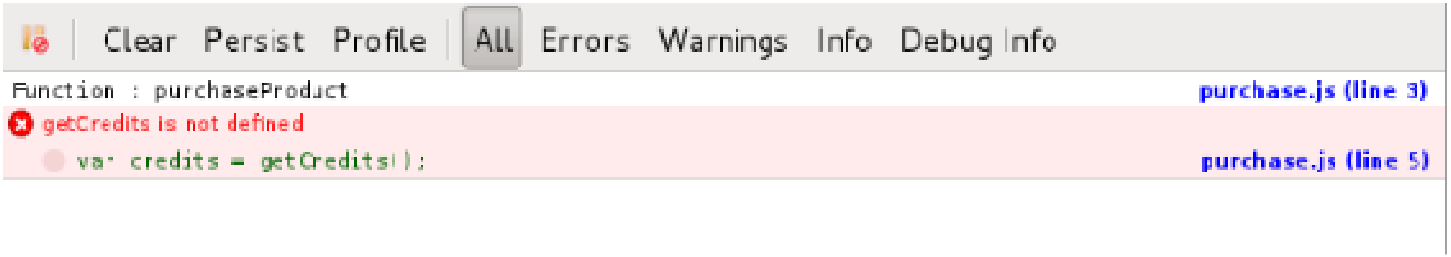
```
1  var result = purchaseProduct();
```

Ein modulares Beispiel

- **Browser-Beispiel: Lade-Reihenfolge**

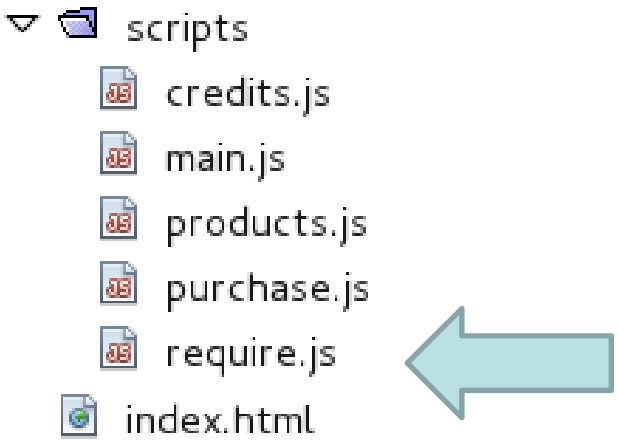
```
<script src="products.js"></script>
<script src="purchase.js"></script>
<script src="main.js"></script>
<script src="credits.js"></script>
```

- **Fehler, da bei der Ausführung von main.js → purchase.js,da getCredits() noch nicht verfügbar ist!**



Beispiel: Laden mit require.js

- **Dateistruktur der „public“-Files**



- **index.html definiert nur noch einen Einstiegspunkt (ein Modul wird geladen)**

```
<script data-main="scripts/main" src="scripts/require.js">
</script>
```

Beispiel: Laden mit require.js

- **Dateien nutzen dann require und define**

main.js
```
require(["purchase"], function(purchase){
    var result = purchase.purchaseProduct();
});
```

  - `require` – **Aufruf über die require-Bibliothek**
  - `["purchase"]` – **Array mit benötigten Modulen**
  - `(purchase)` – **Übergabe des geladenen Moduls**

  - **Achtung: bei mehreren geladenen Modulen ist die Reihenfolge der moduleID-Strings dann auch die Reihenfolge der Parameter. Parameter-Namen sind „beliebig“.**

main.js - Variante mit fiktivem Modul other
```
require(["purchase", "other"], function(a, b){
    var result = a.purchaseProduct();
});
```

Beispiel: Laden mit require.js

- **Dateien nutzen dann require und define**

main.js

```
require(["purchase"], function(purchase){
    var result = purchase.purchaseProduct();
});
```

purchase.js

```
define('purchase', ['credits, products'], function(creditsModule,
                                                   productsModule) {
    return {
        purchaseProduct: function() {
            console.log('Function: purchaseProduct');
            var credits = creditsModule.getCredits();
            if (credits > 0) {
                productsModule.reserveProduct();
                return true;
            }
            return false;
        }
    }
});
```

AMD mittels Require.js

Fazit

AMD mit Require.js

- asynchrones Laden von abhängigen Modulen mittels
  `require(['a', 'b'], function(a, b) { …});`
- moduleID muss eindeutig sein (oder eine Pfadangabe)
- pro Moduldatei auch mehrere Module definierbar
- Module werden definiert über

```
define('moduleID', ['a', 'b', 'c'], function(a, b,c) {
    …
    return moduleObject;
}
```


Quelle: siehe auch http://requirejs.org/docs/whyamd.html und http://www.sitepoint.com/understanding-requirejs-for-effective-javascript-module-loading/

jQuery ist CommonJS ~und~ AMD-kompatibel

```
if ( typeof module === "object"
     && typeof module.exports === "object" ) {

   module.exports = factory( global, true );
    …
} else {
   …
}
```

```
…
jQuery = function( selector, context ) {
   return new jQuery.fn.init( selector, context );
}, …
```

```
if ( typeof define === "function" && define.amd ) {
   define( "jquery", [], function() {
      return jQuery;
   } );
}
```


Quelle: jQuery 3.1.1 uncompressed development source von https://jquery.com/download/

Agenda

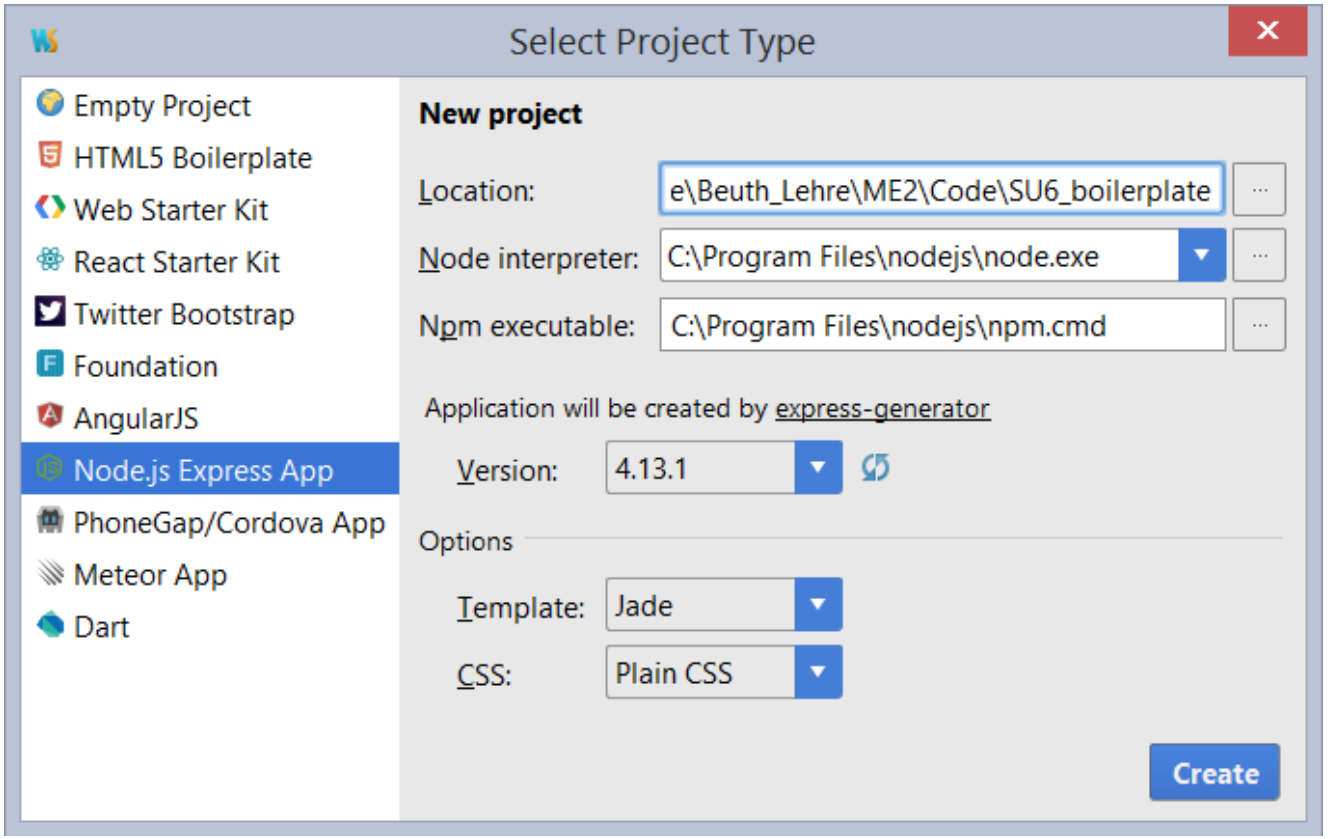
- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

Bonus:
Modulare Code-Skelette generieren

CommonJS in node.js/express

- Es gibt viele Generatoren für ein App-Skelett (sogenannte Boilerplate Generatoren)

- Beispiel WebStorm

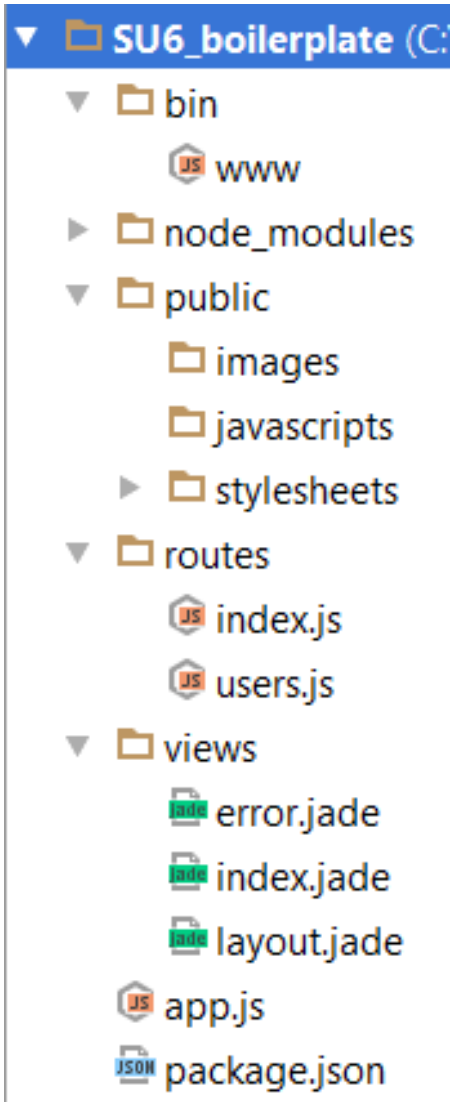


CommonJS in node.js/express

- Es gibt viele Generatoren für ein App-Skelett (sogenannte Boilerplate Generatoren**)

- Beispiel
  WebStorm
  - generiert eine Bootstrap `\bin\www` Datei, die `app.js` mit `require` läd und den Server startet
  - bindet Jade Template-Engine ein
  - Generatoren-Problem: Sie tun nie genau das, was man braucht (meist zu viel Code-Skelett)

- WebStorm nutzt
  `npm install -g express-generator`



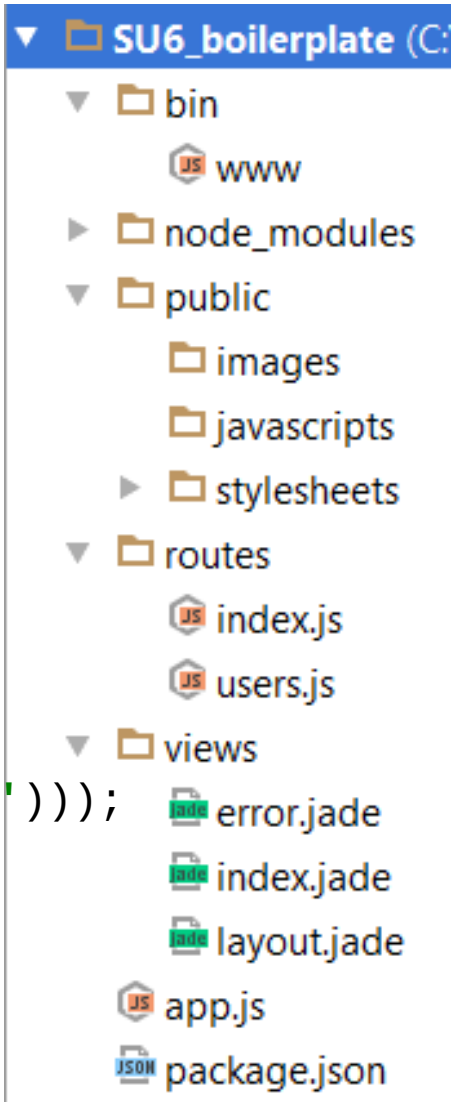
siehe auch http://expressjs.com/starter/generator.html

**Suche im Web bringt hier viele Varianten, u.A. via Yeoman.io

CommonJS in node.js/express

- **Auszug aus express-generator Modul `app.js`**

```
var express = require('express');
var path = require('path');
var logger = require('morgan');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);

module.exports = app;
```



Agenda

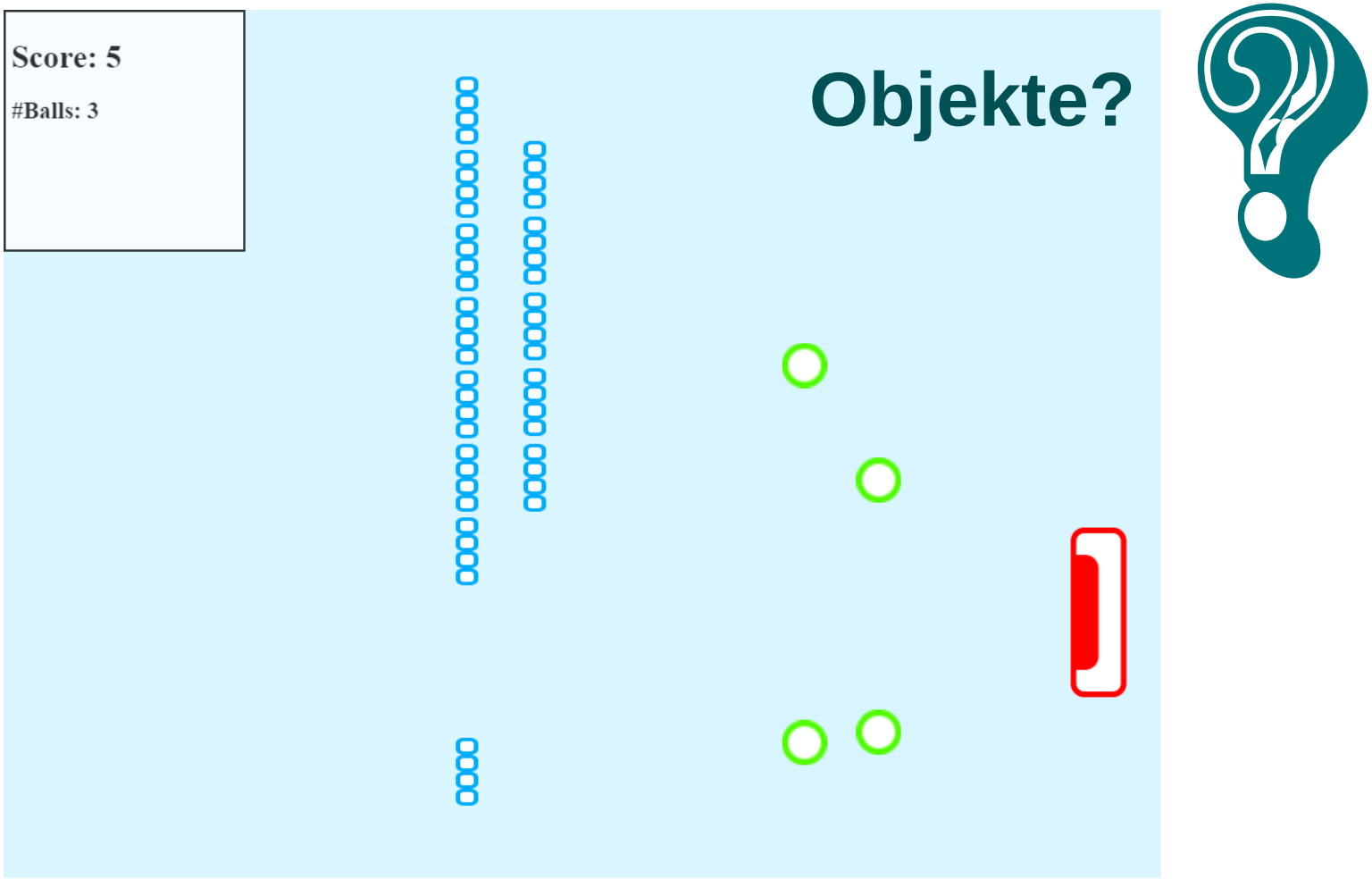
- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser

- **Vererbung in JavaScript**
  - Object.create()
  - Konstruktor-Funktionen
  - prototype

- Zusammenfassende Fragen
- Ausblick

Objektorientierung und Vererbung in JavaScript

- **Beispiel: ein Browser-Game (Canvas-basiert)**



Quelle: eigener Code, Grafiken: http://opengameart.org/content/breakout-graphics-no-shadow, Code-Idee: https://github.com/SpookyFM/PongOut

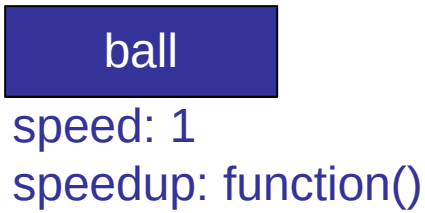
Javascript Vererbung (Objekthierarchien)

- **Objekthierarchien am Beispiel (verschiedene Ball-Typen)**

```
var ball = {
    speed: 1,
    speedup: function() {
        this.speed = this.speed + 1;
    }
};
 console.log(ball.speed);
var ballChild = Object.create(ball);
ballChild.speedup();
 console.log(ball.speed, ballChild.speed);
```

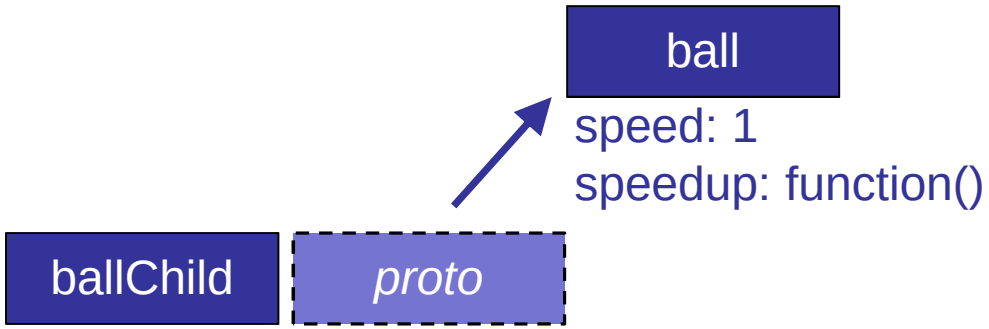
Javascript Vererbung (Objekthierarchien)

```
var ball = {
    speed: 1,
    speedup: function() {
        this.speed = this.speed + 1;
    }
};
 console.log(ball.speed);
var ballChild = Object.create(ball);
ballChild.speedup();
 console.log(ball.speed, ballChild.speed);
```



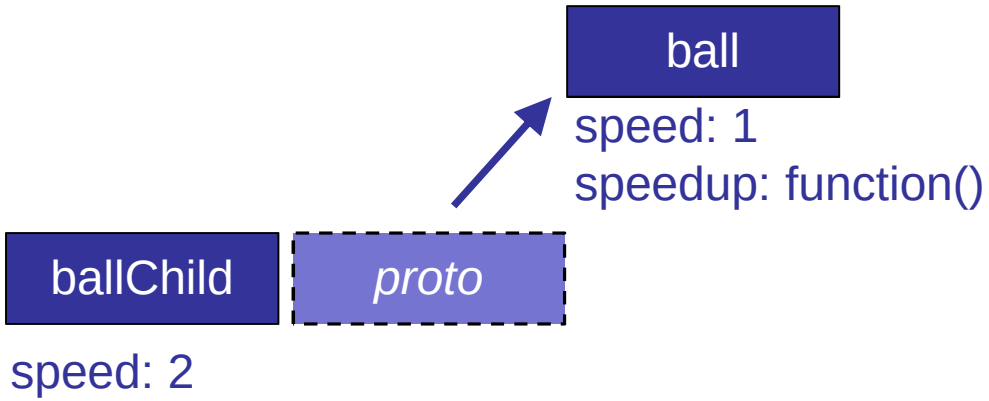
Javascript Vererbung (Objekthierarchien)

```
var ball = {
    speed: 1,
    speedup: function() {
        this.speed = this.speed + 1;
    }
};
 console.log(ball.speed);
var ballChild = Object.create(ball);
ballChild.speedup();
 console.log(ball.speed, ballChild.speed);
```



Javascript Vererbung (Objekthierarchien)

```
var ball = {
    speed: 1,
    speedup: function() {
        this.speed = this.speed + 1;
    }
};
 console.log(ball.speed);
var ballChild = Object.create(ball);
ballChild.speedup();
 console.log(ball.speed, ballChild.speed);
```



Javascript Vererbung (Objekthierarchien)

```
var ball = {
    speed: 1,
    speedup: function() {
        this.speed = this.speed + 1;
    }
};
 console.log(ball.speed);
var ballChild = Object.create(ball);
ballChild.speedup();
 console.log(ball.speed, ballChild.speed);
```

(1)

(2)

An welcher Stelle muss man **`ball.speedup();`** einfügen, damit **beide Bälle gleich schnell** sind?

(1) oder (2)?

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

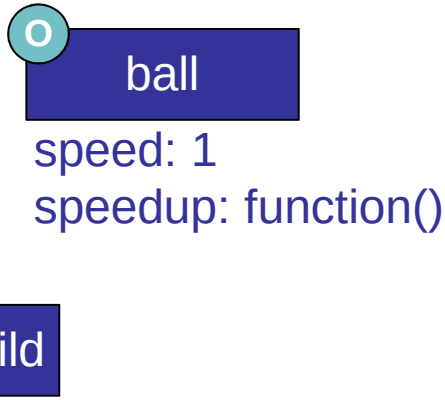
Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
...
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};


FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```

Objekt

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};
```

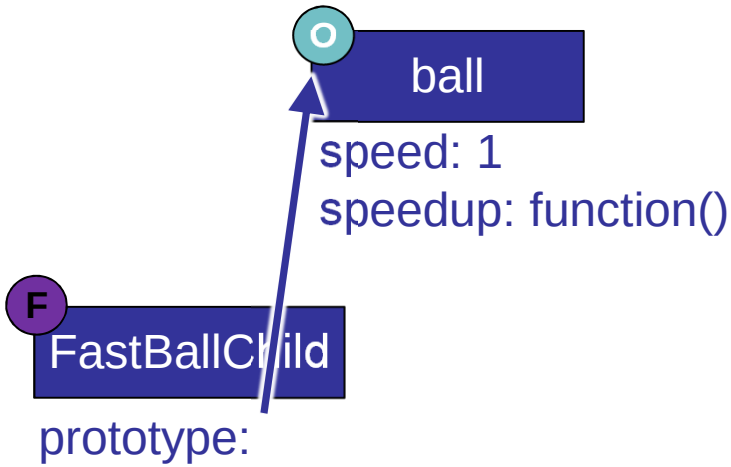



```
FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```

O Objekt F Funktion

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};


FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```

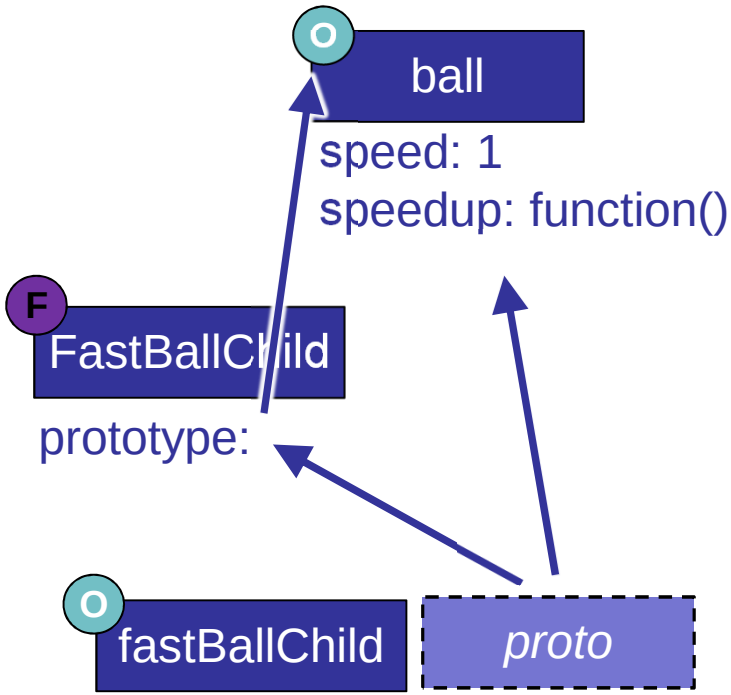


O Objekt F Funktion

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};


FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();
```

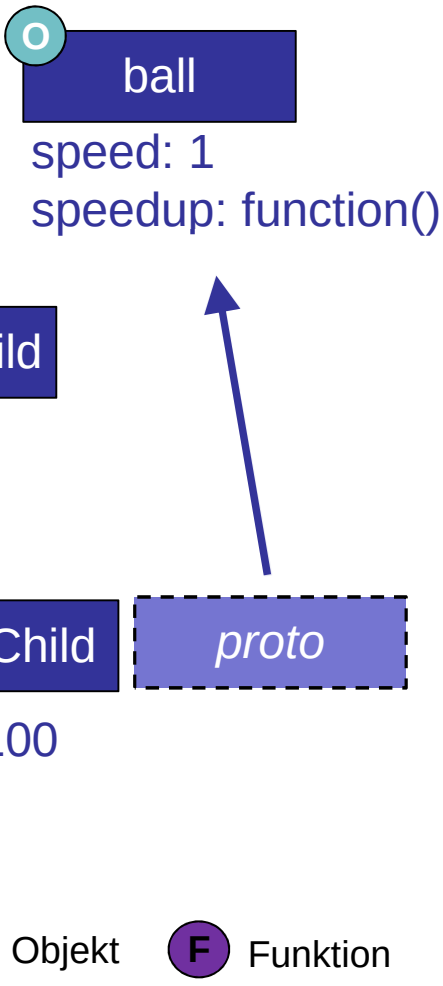
```
fastBallChild.speedup();
console.log(fastBallChild.speed);
```



O Objekt F Funktion

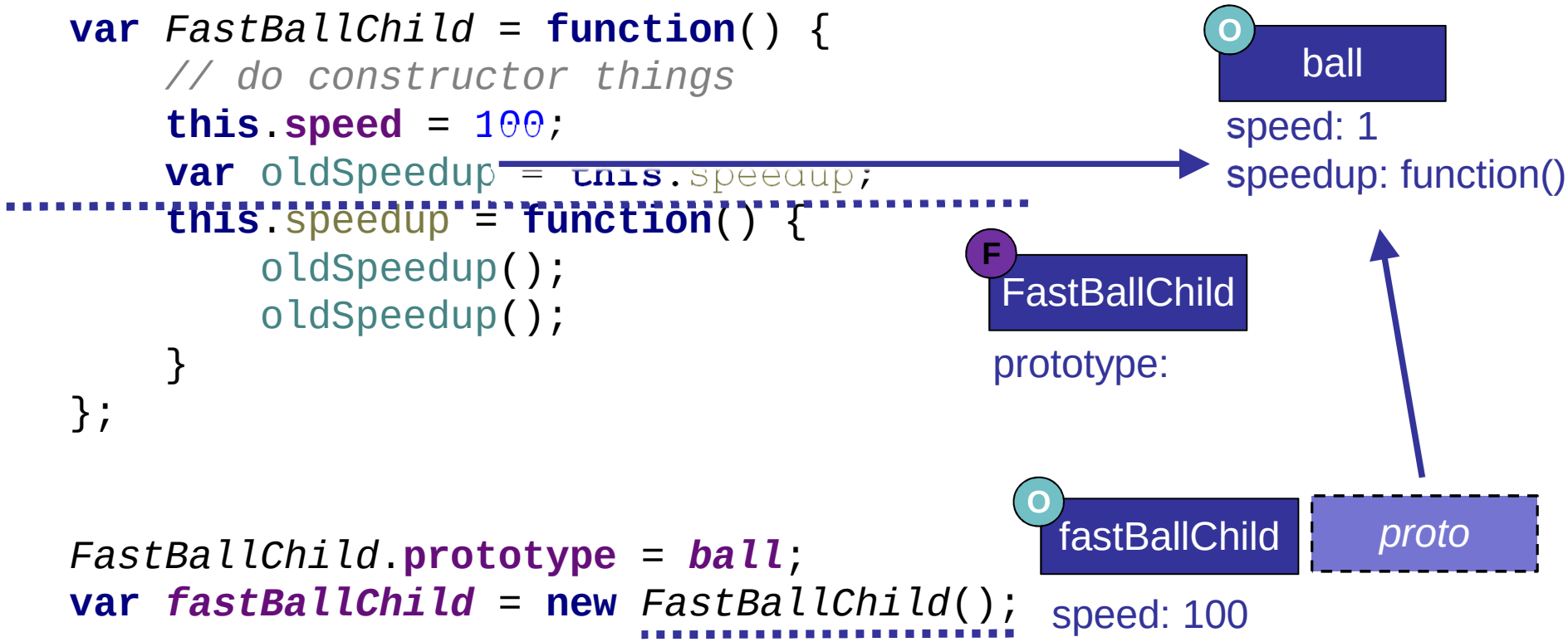
Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};

FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```



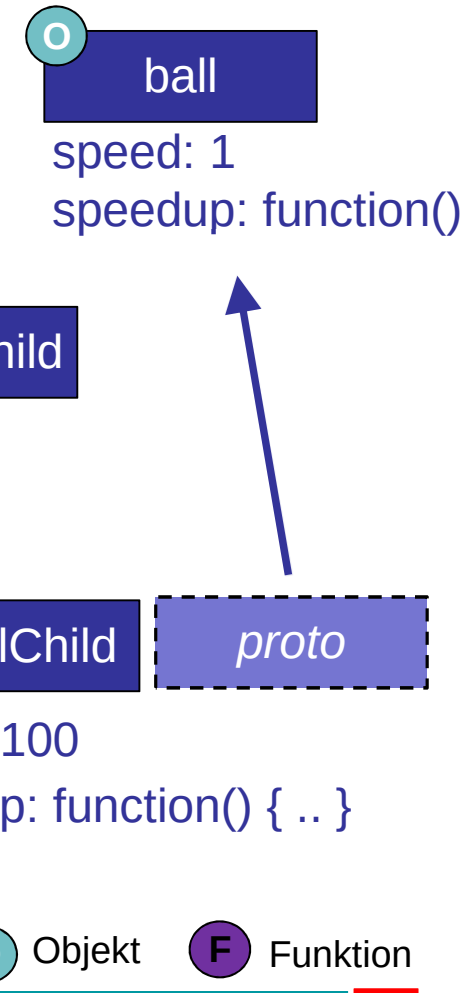
Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};


FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```



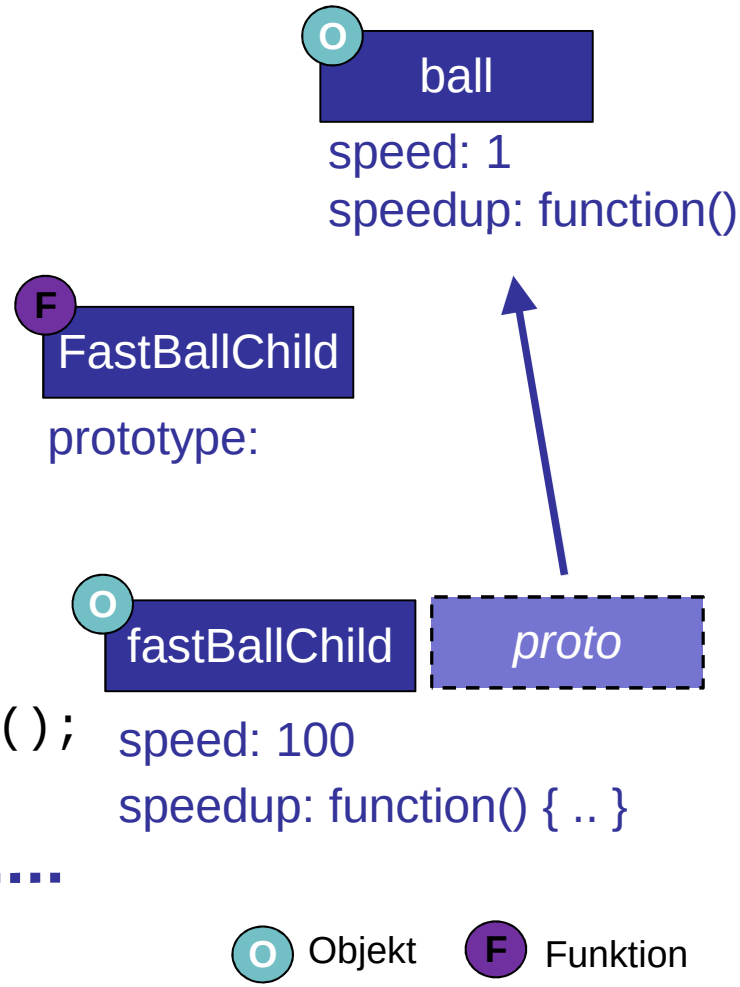
Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};

FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```



Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};


FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```

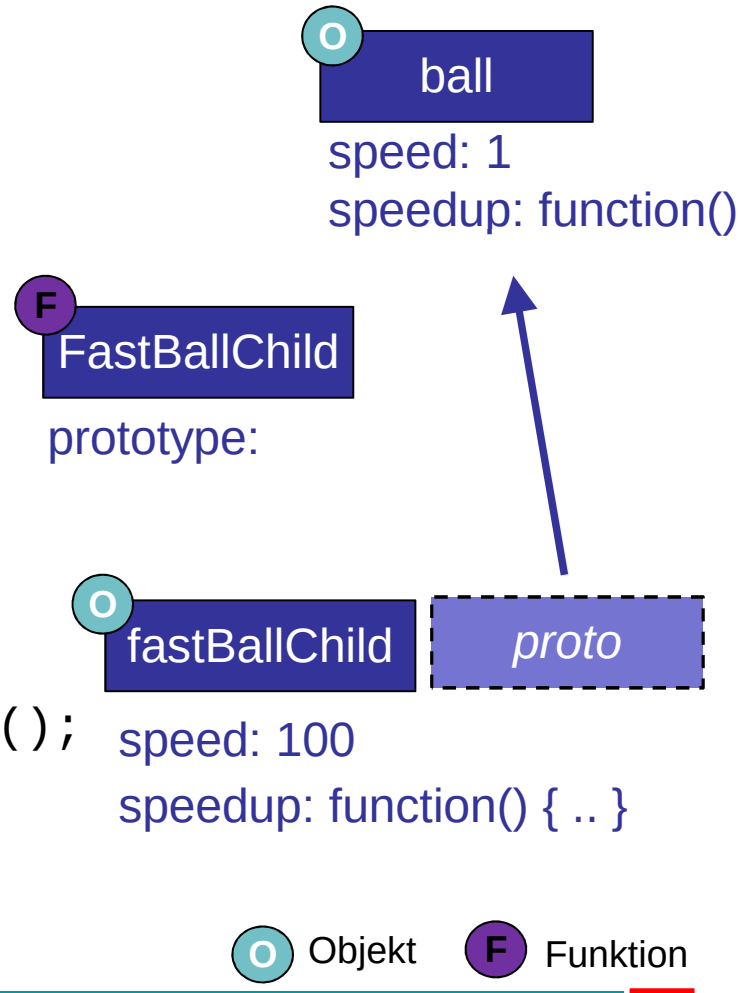


Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
..
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
        oldSpeedup();
        oldSpeedup();
    }
};
```

Ausführungskontext global! (Autsch)

```
FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```



Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
            oldSpeedup.apply(this);      = this.oldSpeedup();
            oldSpeedup.apply(this);      = this.oldSpeedup();
        };
};


FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

fastBallChild.speedup();
console.log(fastBallChild.speed);
```

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```
var FastBallChild = function() {
    // do constructor things
    this.speed = 100;
    var oldSpeedup = this.speedup;
    this.speedup = function() {
            oldSpeedup.apply(this);
            oldSpeedup.apply(this);
        };
};


FastBallChild.prototype = B
var fastBallChild = new Fas

fastBallChild.speedup();
console.log(fastBallChild.s
```

Konvention:
Konstruktor-Funktionen fangen mit Großbuchstaben an

Sehr mächtiges funktionales Paradigma

`<func>.apply(<context>)`

erlaubt das ausführen von Funktionen als Methoden im Kontext anderer Objekte
(this = <context>)

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype im Detail
- Zusammenfassende Fragen
- Ausblick

`prototype` im Detail

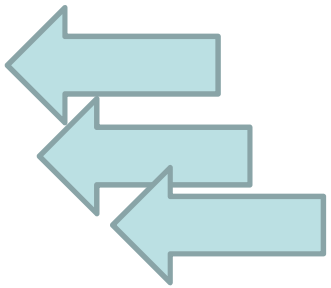
- **Jede Konstruktor-Funktion besitzt die Prototype-Eigenschaft**

```
var Car = function(){};
Car.prototype  // = {};
```

- **Alles was in dieser definiert wird gilt für ALLE Instanzen ~~der Klasse~~ dieses Konstruktors!**
  - **Öffentliche Methoden**
  - **Konstante Eigenschaften**
- `prototype` ist auch zur Laufzeit für alle existierenden Instanz-Objekte änderbar
- Jedes Objekt kann lokal `prototype`-Eigenschaften/Methoden mit neuen Definitionen überschreiben

prototype im Detail

```
var Car = function(){};
Car.prototype.doors = 5;
Car.prototype.getDoors = function(){ // public method
    return this.doors;
};

var myCar = new Car();
var yourCar = new Car();
yourCar.doors = 3;

Car.prototype.color = 'white';
Car.prototype.getColor = function() {
    return this.color;
};
logger.log(myCar.getDoors());
logger.log(yourCar.getDoors());
logger.log(myCar.getColor());
```



prototype im Detail

```
var Car = function(){};
Car.prototype.doors = 5;
Car.prototype.getDoors = function(){ // public method
    return this.doors;
};

var myCar = new Car();
var yourCar = new Car();
yourCar.doors = 3;

Car.prototype.color = 'white';
Car.prototype.getColor = function() {
    return this.color;
};
logger.log(myCar.getDoors()); // 5
logger.log(yourCar.getDoors()); // 3
logger.log(myCar.getColor()); // white
```

`prototype` im Detail

Vorteile prototypischer Vererbung

- **bestehende Instanzen können nachträglich manipuliert werden**
- **Spart Speicherplatz, da Methoden/Eigenschaften nur 1x im Prototype definiert, statt in jedem Objekt neu**

- **Achtung:** Änderungen an prototype wirken sich zwar auf alle Instanzen dieser Konstruktor-Funktion aus,
  aber lokale, überschriebene Werte von Eigenschaften/Methoden bleiben bei Objekten erhalten (wie yourCar.doors im Beispiel)
- **Achtung:** Namens-Konvention, dass Konstruktorfunktionen mit ersten Buchstaben Groß geschrieben werden
  (bspw. `var Car = function() {}` bzw. `function Car() { }`)

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

Zusammenfassende Fragen

- **Themen heute**
  - Objekt-Literale, Anonyme Funktionen, JS Module Pattern
  - CommonJS und require() in node.js
  - AMD und require.js
  - Code-Generatoren // Boilerplates
  - Vererbung mit Object.create()
  - Vererbung mit Konstruktor-Fu
  - prototype im Detail

- **Ihre Karten**

(1) Einsammeln

(2) Nächstes Mal damit Quiz

Zusammenfassende Fragen

1. Welche zwei wesentlichen Probleme sollen durch Modularisierung gelöst werden? Wie gelingt das?
2. Was ist das JavaScript Module Pattern? Wie sieht die Syntax aus?
3. Wie gelingt die Übergabe von Parametern in das Modul hinein beim Modul-Pattern?
4. Wie kann eine Methode oder Eigenschaft vor dem Zugriff von außerhalb des Moduls geschützt werden, obwohl es kein `private` in JavaScript gibt?
5. Worin unterscheiden sich die Modularisierungslösungen CommonJS und AMD? Nennen Sie mindestens 2 Unterschiede.
6. Welche Modularisierungslösung kann server-seitig genutzt werden, welche client-seitig? Warum?
7. Bei welcher Modularisierungslösung stehen ihnen die Funktionen `require(..)` und `define(..)` innerhalb eines Moduls zur Verfügung?
8. Wo liegt der Unterschied der Variablen `exports` und `module.exports` bei CommonJS?
9. Wie geben Sie bei einer Modul-Definition in AMD-Stil an, welche anderen Module geladen werden sollen, bevor ihr Code laufen kann?
10. Nach welchen Richtlinien sucht `require(..)` bei nodeJS die Module?
11. Welche Modularisierungslösung müssen Sie einsetzen, um jQuery damit verwenden (laden) zu können? Warum?
12. Welche Vor/Nachteile hat ein Boilerplate Generator?
13. Was ist die Gemeinsamkeit von `Object.create(…)` und einer Konstruktorfunktion, wie `Factory()` ?
14. Welche Vorteile hat prototypische Vererbung?

Agenda

- Wiederholung
- Wozu Modularisierung? Problemfälle
- Objekt-Literale
- Das Module Pattern
  - Anonyme Funktionen
  - private und public
  - Erweiterungen
- Zwei Ansätze: CommonJS und AMD
  - CommonJS und require() in node.js
  - AMD und require.js im Browser
- Vererbung in JavaScript
  - Object.create()
  - Konstruktor-Funktionen
  - prototype
- Zusammenfassende Fragen
- Ausblick

Achtung: Nächste Woche Christi Himmelfahrt → Wechsel der Züge)

	Datum	Thema
1	06.04.2017	Einführung, Ziele, Ablauf, Benotung usw.
2	13.04.2017	Wiederholung HTML/CSS/JS
3	20.04.2017	Client-Server Architekturen und WebStacks
4	27.04.2017	REST-APIs
5	04.05.2017	REST in node.js
6	11.05.2017	Debugging und Testen
7	18.05.2017	Strukturierung, Modularisierung
8	**01.06.2017**	Datenhaltung, SQL, NoSQL
9	08.06.2017	backbone.js als Gegenpart zu REST/Node.js
10	15.06.2017	Vertiefung (DB, Sicherheit, ..) als **Flipped Classroom (FC)**
11	22.06.2017	Nachbesprechung FC; Authentifizierung
12	29.06.2017	Mobile Development/Cross-Plattform-Development
13	**06.07.2017**	Zusammenfassung/Semesterüberblick **(alle Züge!)**
14	13.07.2017	Gastdozent(en) mit Anwesenheitspflicht
15	18.07.2017	**Klausur PZR1** (**Di**, 18.07. 10:00 Uhr, Raum ?)
16	25.07.2017	Klausureinsicht (**Di**, 25.07. 10:00 Uhr, Raum ?)
	21.09.2017	**Klausur PZR2** (**Do**, 21.09. 12:00 Uhr, Raum ?)

Zug 1/3

25.05. Christi Himmelfahrt
(ab hier Moodle Wochenwechsel)

Achtung: Nächste Woche Christi Himmelfahrt → Wechsel der Züge)

	Datum	Thema
1	11.04.2017	Einführung, Ziele, Ablauf, Benotung usw.
2	18.04.2017	Wiederholung HTML/CSS/JS
3	25.04.2017	Client-Server Architekturen und WebStacks
4	02.05.2017	REST-APIs
5	09.05.2017	REST in node.js
6	16.05.2017	Debugging und Testen
7	23.05.2017	Strukturierung, Modularisierung
8	30.05.2017	Datenhaltung, SQL, NoSQL
9	06.06.2017	backbone.js als Gegenpart zu REST/Node.js
10	13.06.2017	Vertiefung (DB, Sicherheit, ..) als **Flipped Classroom (FC)**
11	20.06.2017	Nachbesprechung FC; Authentifizierung
12	27.06.2017	Mobile Development/Cross-Plattform-Development
13	**06.07.2017**	**Do,** Zusammenfassung/Semesterüberblick **(alle Züge!)**
14	11.07.2017	Gastdozent(en) mit Anwesenheitspflicht
15	18.07.2017	**Klausur PZR1** (**Di**, 18.07. 10:00 Uhr, Raum ?)
16	25.07.2017	Klausureinsicht (**Di**, 25.07. 10:00 Uhr, Raum ?)
	21.09.2017	**Klausur PZR2** (**Do**, 21.09. 12:00 Uhr, Raum ?)

Zug 2

25.05. Christi Himmelfahrt
(ab hier Moodle Wochenwechsel)

Nächster Unterricht

- Datenhaltung
- NoSQL
- MongoDB
- mongoose für node.js

Vielen Dank und bis zum nächsten Mal

Bildquelle:http://geekandpoke.typepad.com/.a/6a00d8341d3df553ef0148c80ac6ef970c-800wi