

Multimedia Engineering II

06 Modularisierung und Strukturierung

Johannes Konert



BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN

University of Applied Sciences



Agenda

- Wiederholung als Teamquiz
- 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`

optional

- Ihre zusammenfassenden Fragen
- Ausblick

Agenda

- **Wiederholung als Teamquiz**
- **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`
- **Ihre zusammenfassenden Fragen**
- **Ausblick**

Zusammenfassende Fragen und Wiederholung

Wiederholung als Teamquiz

Der Raum wird in zwei Teams eingeteilt



1. Ein Teammitglied (A) zieht 2 Karten
 - Wählt eine von beiden aus
2. Stellt die Frage dem anderen Team (B)
 - Das Team darf diskutieren über die Lösung
3. Das Team (B) antwortet
 - Team A ergänzt.
4. Dozent als Schiedsrichter vergibt endgültigen Punkt an Team A oder B



Anschließend ist das andere Team dran (1.-3.)

Insgesamt werden **vier Fragen** besprochen.

Zusammenfassende Fragen und Wiederholung



Aufgabe:

- 1.** Sie schreiben die zusammenfassenden Fragen + Antworten selbst auf
- 2.** Nutzen Sie dazu die Moderationskarten
 - eine Seite Frage
 - Andere Seite Antwort(en)
+ ggf. Folienummer v. heute
- 3.** Von jedem am Ende mindestens eine Karte bei mir abgeben



Agenda

- Wiederholung als Teamquiz
- 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
- Ihre zusammenfassenden 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 Ursachen für Laufzeitprobleme vermeiden

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

Agenda

- Wiederholung als Teamquiz
- 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
- Ihre zusammenfassenden Fragen
- Ausblick

Objekt Literale 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 als Teamquiz
- 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`
- Ihre zusammenfassenden 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“)

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**

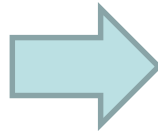
Idee

- 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 definierten 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  
            // using 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)

JavaScript: Das Module Pattern

Schrittweise Herleitung

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



Module erweitern

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

- `old_init` kann innerhalb des gesamten Blocks benutzt werden (bspw. via **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.)

JavaScript: Das Module Pattern

Schrittweise Herleitung

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

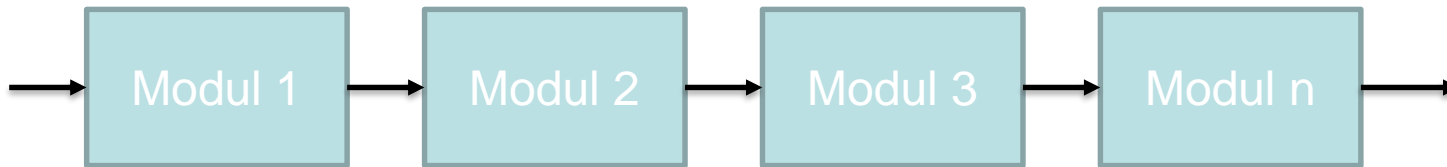


Agenda

- Wiederholung als Teamquiz
- 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
- Ihre zusammenfassenden Fragen
- Ausblick

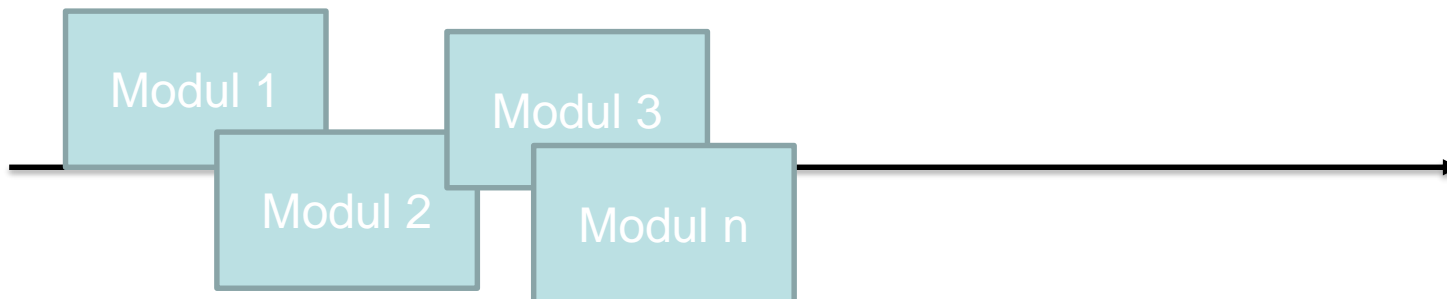
Modularisierungslösungen (JavaScript)

■ CommonJS Spezifikation



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

■ Asynchronous Module Definition (AMD)



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

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**
 - **bei node.js: module hat ein Attribut module.exports, die auf exports verweist und ersetzt werden darf**

CommonJS Spezifikation



- **require: eine Funktion**
 - **Parameter: Modulname (ID)**
 - **Rückgabe: liefert die API**
- **module: eine Datei**
 - **Variable require**
 - **Variable exports**
 - **Variable module**
 - **module.id**
 - **node.js: module.exports**

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

```
'use strict';  
//...  
var proto = require('./application');  
var req = require('./request');  
var res = require('./response');  
  
module.exports = createApplication;  
exports = module.exports;  
  
/** */  
function createApplication() {  
    //..  
    return app;  
}
```


Agenda

- Wiederholung als Teamquiz
- 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
- Ihre zusammenfassenden Fragen
- Ausblick

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

1. Datei finden

- in der Hauptdatei (app.js): Suche im Verzeichnis `./node_modules/moduleID/`
 - Wenn eine `package.json`, lade die Datei des Abschnitts „main“
 - Ansonsten lade eine `index.js` oder `index.json`
- in einer Moduldatei (express.js): Suche im Verzeichnis `./node_modules/moduleID.js`
 - Wenn nicht gefunden, dann gehe ein Verzeichnis hoch und Suche weiter (solange bis bei `./node_modules/moduleID.js`)

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

2. Dateiinhalt ausführen und zurückliefern

- Dateiinhalt in eigenem Namensraum (scope) ausführen
 - Mittels anonymer Funktion
 - `require`, `module.exports` 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;  
  
}
```

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

2. Dateiinhalt ausführen und zurückliefern

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

Fazit

CommonJS Spezifikation

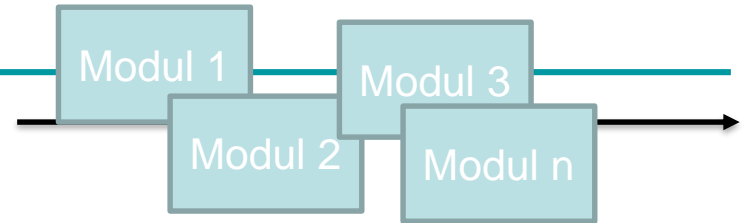
- synchrones Laden von abhängigen Modulen mittels `require('moduleId')`
- `moduleId` muss eindeutig sein (oder eine Pfadangabe)
- pro Moduldatei genau ein Modul, welches im Scope einer anonymen Funktion ausgeführt wird
- Nur der Inhalt von `module.exports` wird als öffentliche API zurückgegeben

Agenda

- Wiederholung als Teamquiz
- 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`
- Ihre zusammenfassenden 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.foo();  
});
```

■ 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', ['otherLibModule'], function (libModule) {  
    function foo() {  
        return libModule.foo();  
    }  
    // export (expose) foo  
    return foo;  
});
```

Require.js - eine AMD Umsetzung



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.

```
--- */
```


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()

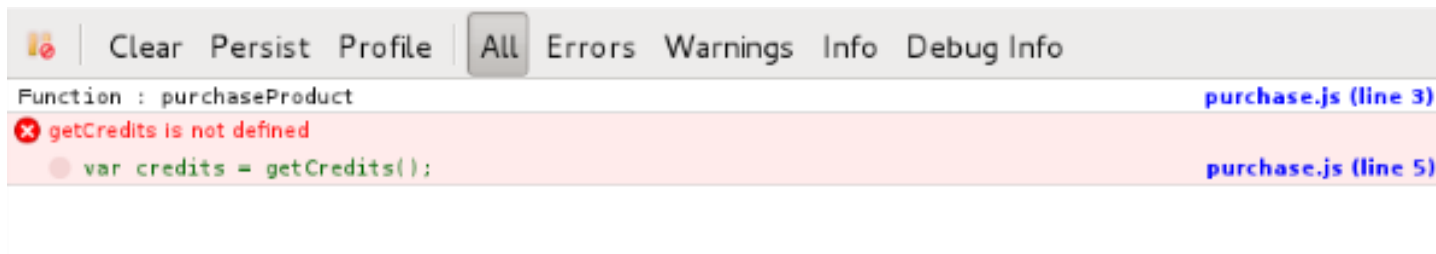
```
1 var result = purchaseProduct();
```

Ein modulares Beispiel

■ Browser-Beispiel: Lade-Reihenfolge

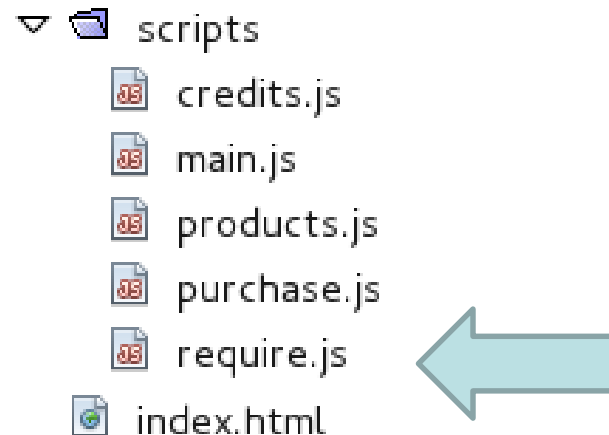
```
<script src="products.js"></script>
<script src="purchase.js"></script>
<script src="main.js"></script>
<script src="credits.js"></script>
```

- 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 Kontext/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('id', ['a', 'b', 'c'], function(a, b, c) {  
    ...  
    return moduleObject;  
})
```

Agenda

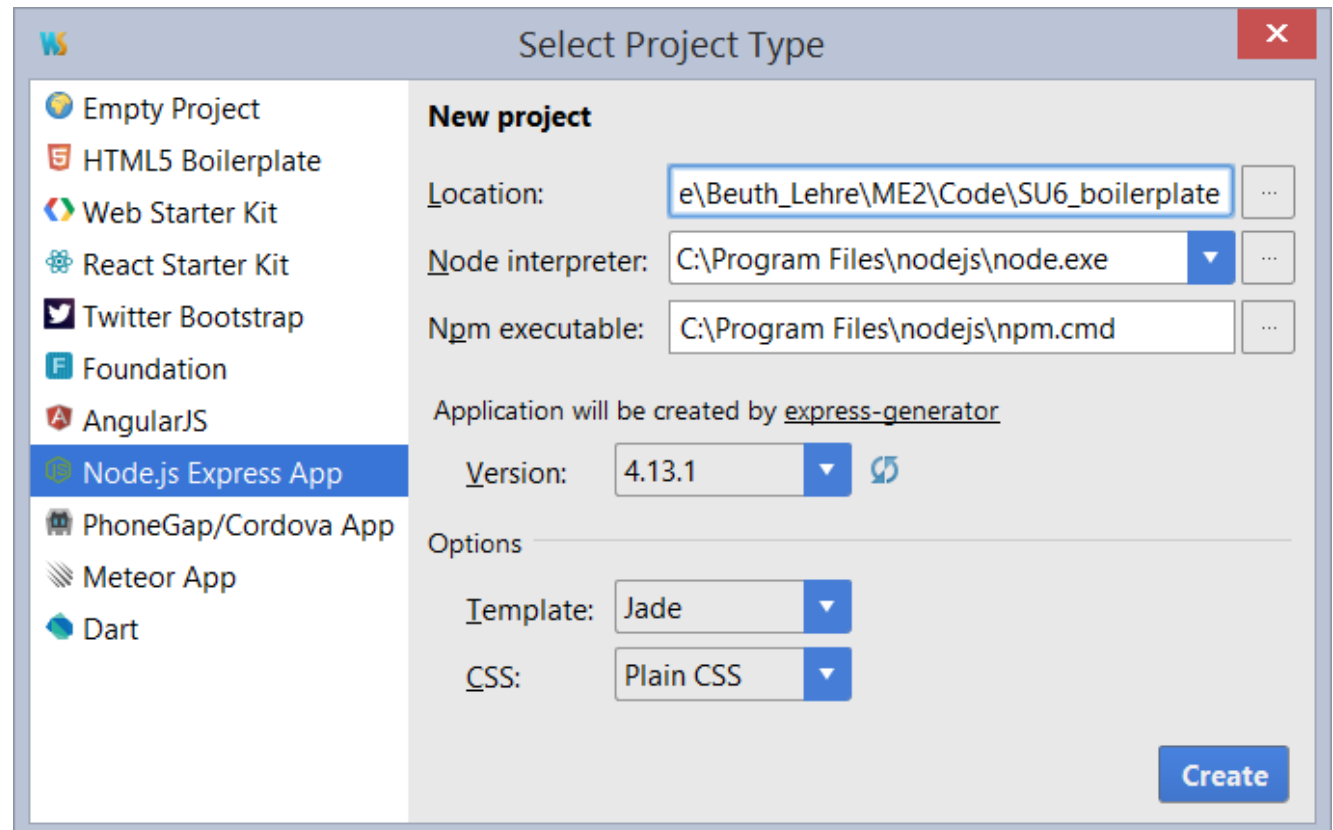
- Wiederholung als Teamquiz
 - Wozu Modularisierung? Problemfälle
 - Objekt-Literale
 - Das Module Pattern
 - Anonyme Funktionen
 - `private` und `public`
 - Erweiterungen
 - Zwei Ansätze: CommonJS und AMD
 - CommonJS und `require()` in `node.js`
 - AMD und `require.js` im Browser
-
- Vererbung in JavaScript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

Bonus:
Modulare Code-
Skelette generieren

CommonJS in node.js/express

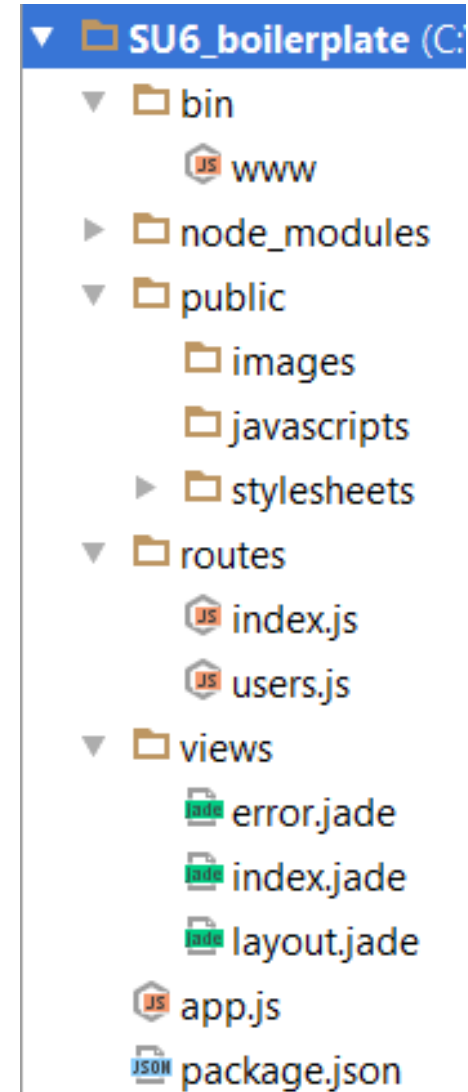
- Es gibt viele Generatoren für ein App-Skelett (sogenannte Boilerplates)

- Beispiel WebStorm



CommonJS in node.js/express

- Es gibt viele Generatoren für ein App-Skelett (sogenannte Boilerplates**)
- Beispiel WebStorm
 - generiert eine Bootstrap \bin\www Datei, die app.js mit require lädt und den Server startet
 - Generatoren-Problem: Sie tun nie genau das, was man braucht (meist zu viel Code-Skelett)
- WebStorm nutzt
`npm install -g express-generator`



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');

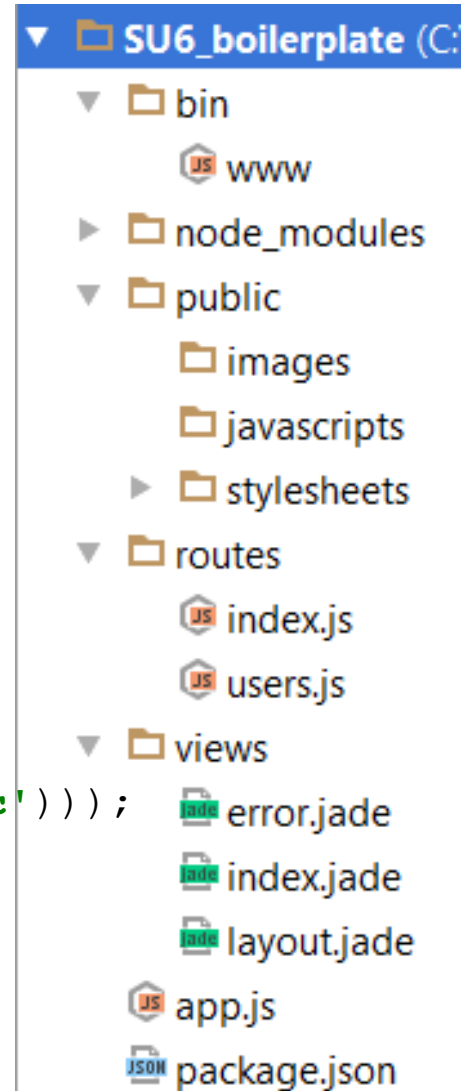
var app = express();

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;
```

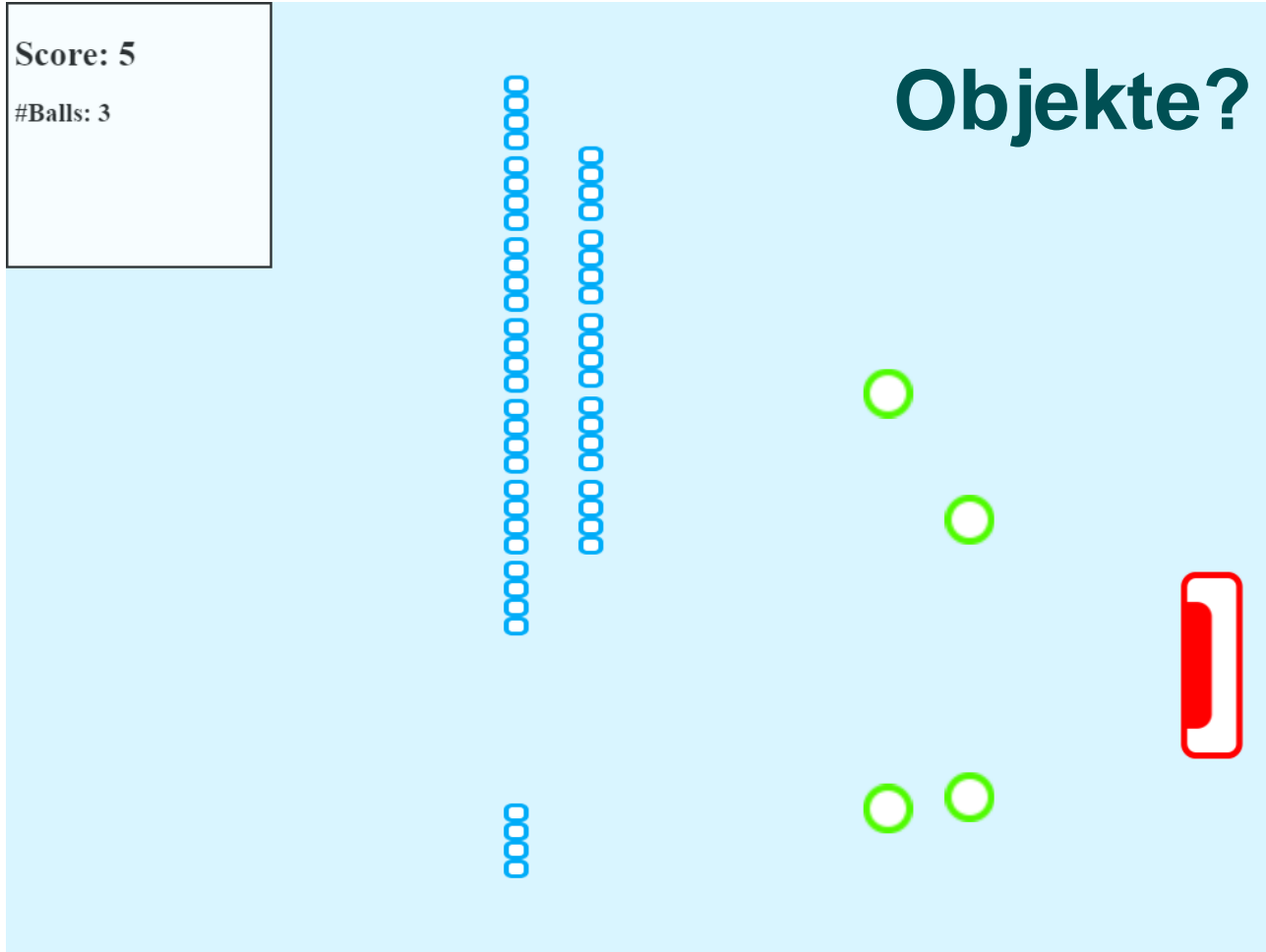


Agenda

- Wiederholung als Teamquiz
 - 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
-
- Ihre zusammenfassenden Fragen
 - Ausblick

Objektorientierung und Vererbung in JavaScript

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



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);
```

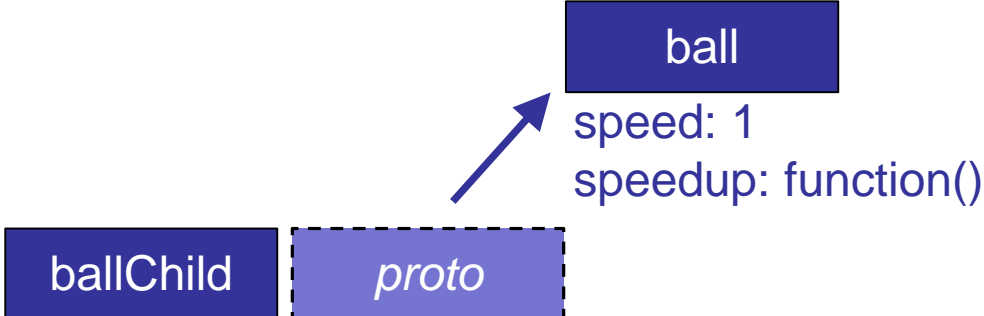
ball

speed: 1

speedup: function()

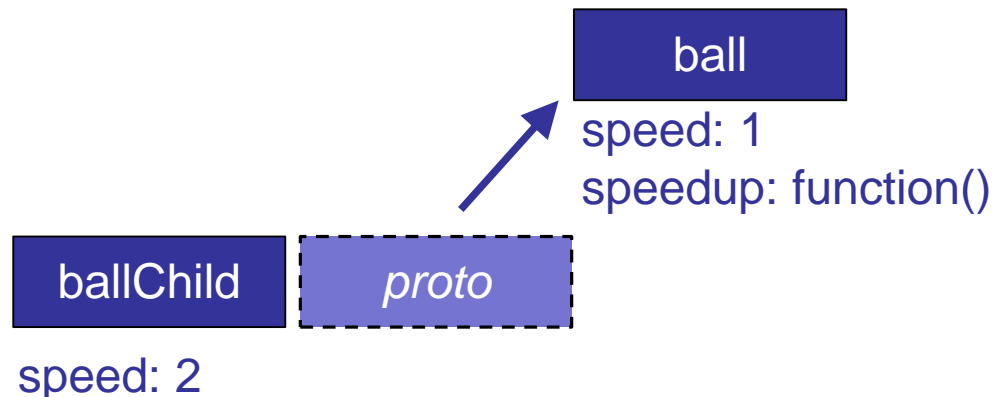
Javascript Vererbung (Objekthierarchien)

```
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);  
(1) var ballChild = Object.create(ball);  
(2) ballChild.speedup();  
console.log(ball.speed, ballChild.speed);
```



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

(1) oder (2)?

Agenda

- Wiederholung als Teamquiz
 - 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
-
- Ihre zusammenfassenden 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);
```

o

ball

speed: 1

speedup: function()

o

Objekt

Javascript Vererbung (Objekthierarchien, Konstruktoren)

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

O ball
speed: 1
speedup: function()

F FastBallChild

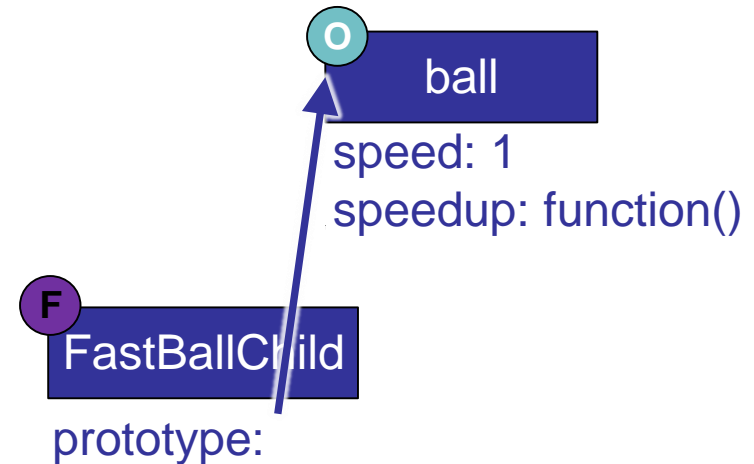
```
FastBallChild.prototype = ball;  
var fastBallChild = new FastBallChild();
```

```
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);
```

 Objekt  Funktion

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```

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

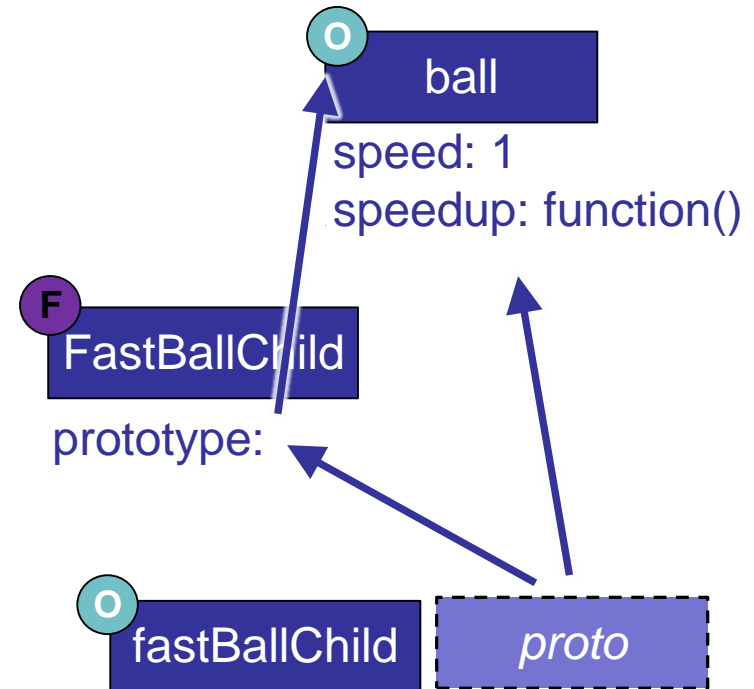
FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

```

```

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

```



 Objekt  Funktion

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```

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

```

```

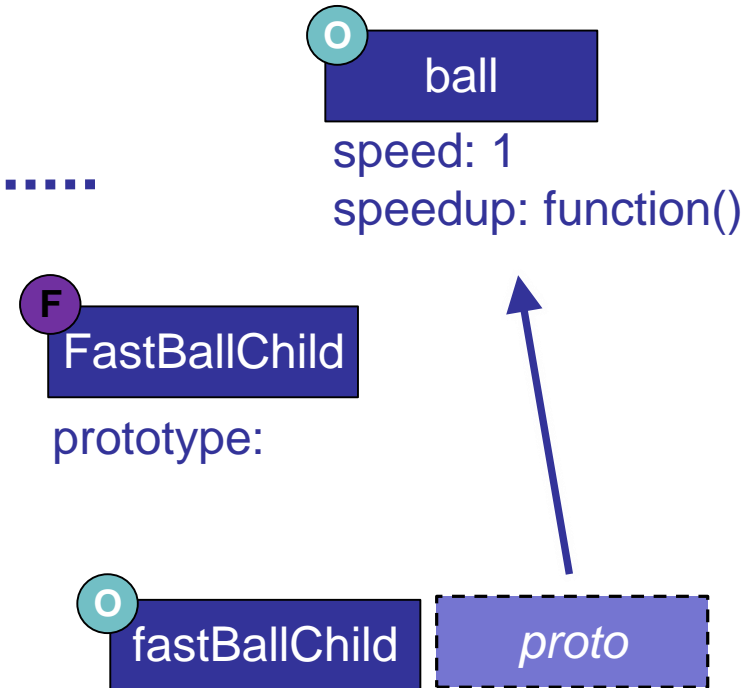
FastBallChild.prototype = ball;
var fastBallChild = new FastBallChild();

```

```

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

```



 Objekt  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);
```

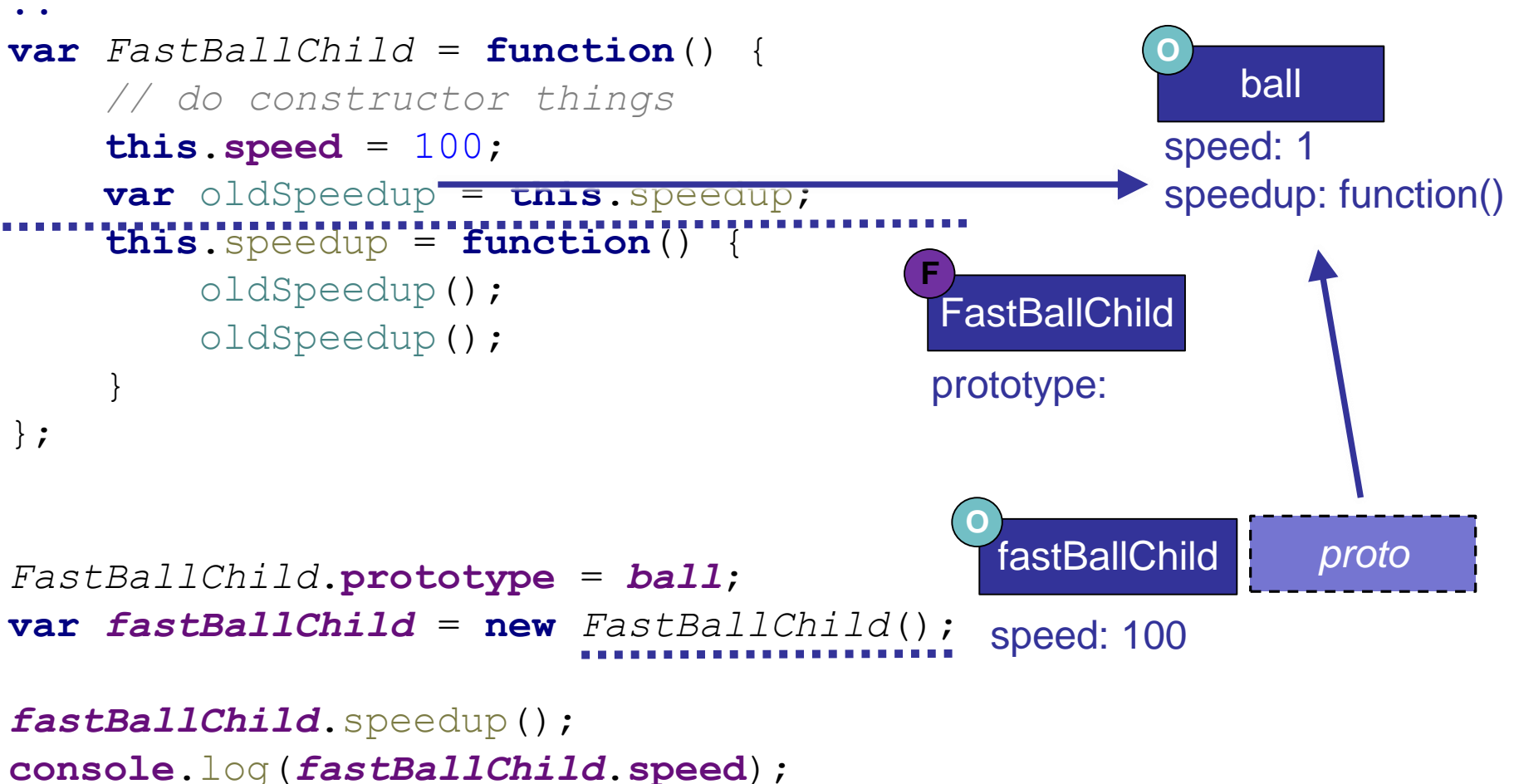


Diagram illustrating JavaScript inheritance:

- ball** (Object):
 - speed: 1
 - speedup: function()
- FastBallChild** (Function):
 - prototype: ball
- fastBallChild** (Object):
 - speed: 100
 - prototype: FastBallChild

Objekt Funktion

Javascript Vererbung (Objekthierarchien, Konstruktoren)

```

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

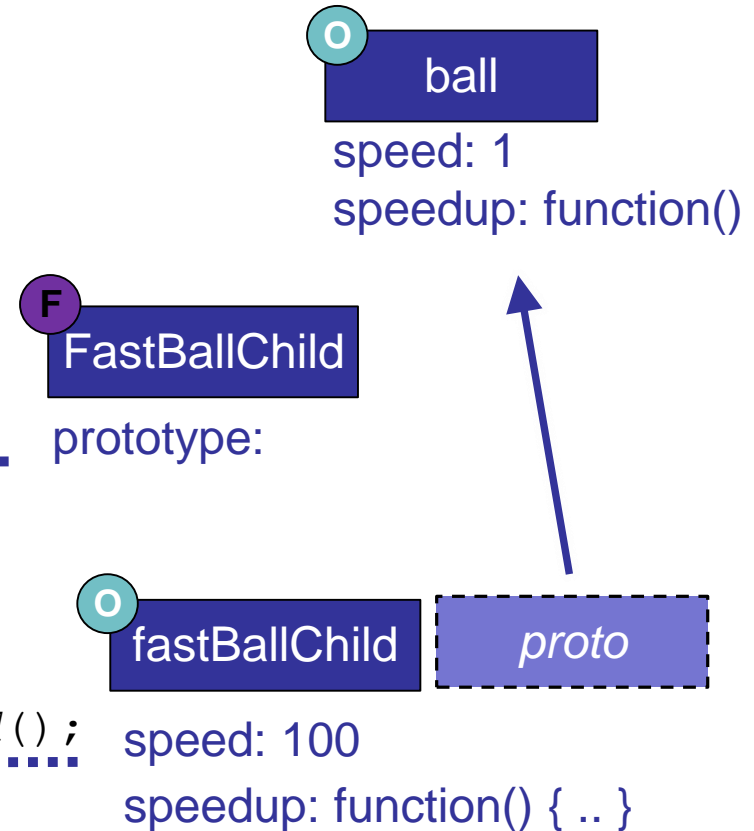
```

```

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

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

```



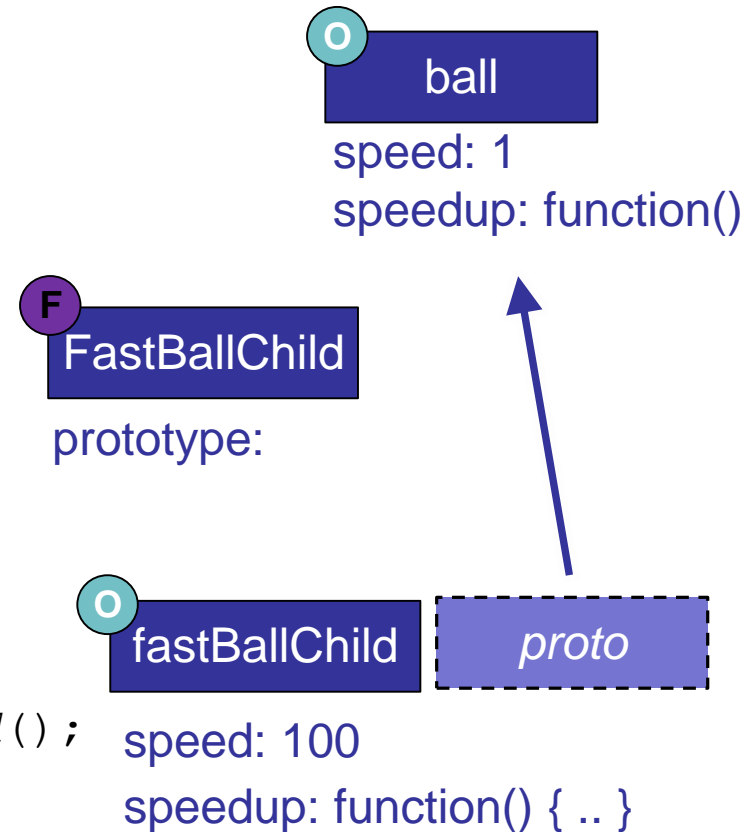
O Objekt
 F Funktion

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

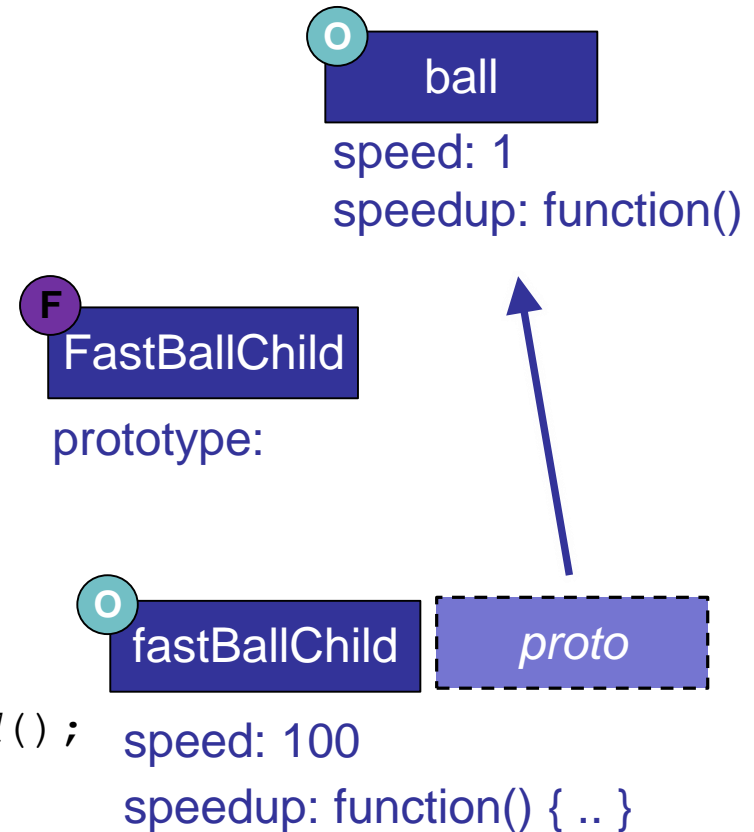
Javascript Vererbung (Objekthierarchien, Konstruktoren)

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

*Ausführungskontext global!
(Autsch)*

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

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



Objekt Funktion

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);  
    };  
};
```

*= this.oldSpeedup();
= 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 = 1  
var fastBallChild = new FastBallChild()  
  
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 als Teamquiz
- 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

- Ihre zusammenfassenden Fragen
- Ausblick

prototype im Detail

- Jede Konstruktor-Funktion besitzt die Prototype-Eigenschaft

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

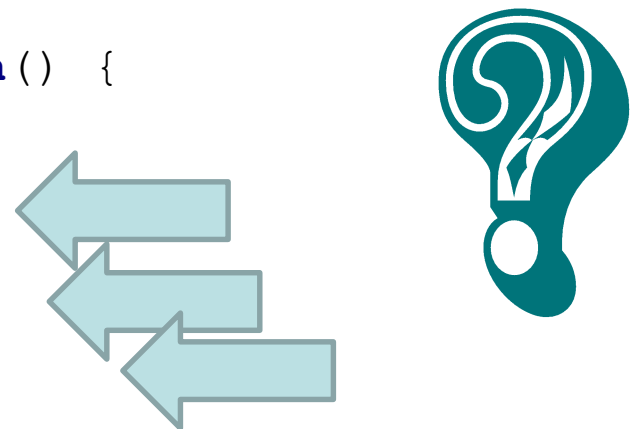
- Alles was in dieser definiert wird gilt für ALLE Instanzen ~~der Klasse~~ dieses Konstruktors!
 - Öffentliche Methoden
 - Konstante Eigenschaften
- prototype ist auch zur Laufzeit für alle existierenden Instanz-Objekte änderbar
- Jedes Objekt kann lokal prototype-Eigenschaften/Methoden mit neuen Definitionen 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 Konstrukturfunktionen mit ersten Buchstaben Groß geschrieben werden
(bspw. `var Car = function() {}` bzw. `function Car() { }`)

Agenda

- Wiederholung als Teamquiz
- 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
- Ihre zusammenfassenden 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-Funktionen
- prototype im Detail

■ Ihre Karten

(1) Einsammeln

(2) Drei Beispiele ziehen wir direkt
(und nutzen die Karten wieder
nächstes Mal)



Agenda

- **Wiederholung als Teamquiz**
- **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**
- **Ihre zusammenfassenden Fragen**
- **Ausblick**

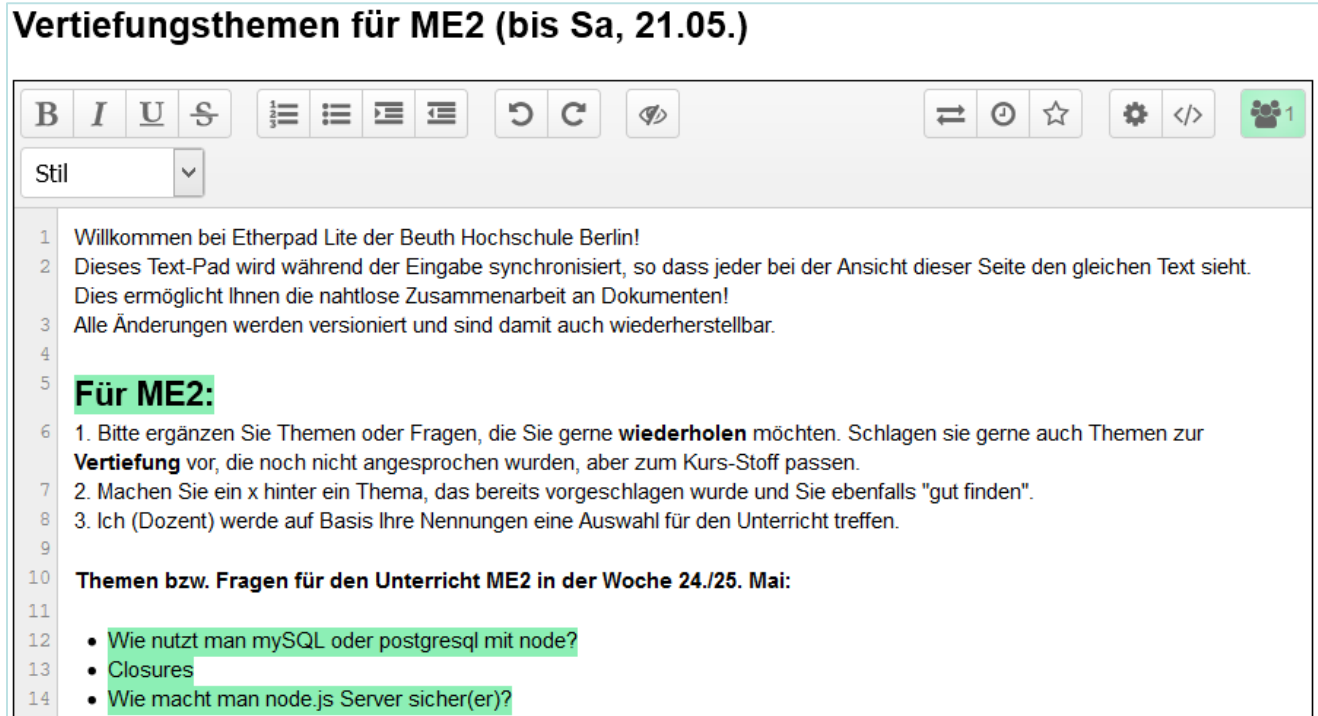
Wo sind wir?

	Datum	Thema	Übung
1	05.04.2016		Ü1: Client-Website
2	12.04.2016	Einführung, Ziele, Ablauf, Benotung	Ü1
3	19.04.2016	Client-Server Architektur	Ü2: Server mit node.js
4	26.04.2016	REST-APIs	Ü2
5	03.05.2016	REST in node.js	- Feiertag - (5.5.)
6	10.05.2016	Debugging und Testen	Ü3: API mit node.js
7	17.05.2016	Strukturierung, Modularisierung	Ü3
8	24.05.2016	Vertiefung einzelner Themen	Ü4: Umfangreiche REST API
9	31.05.2016	Datenhaltung, SQL, NoSql, primär mit MongoDB	Ü4
10	07.06.2016	backbone.js als Gegenpart zu REST/node	Ü4
11	14.06.2016	Authentifizierung und Patterns	Ü5: mongoDB-Anbindung
12	21.06.2016	Mobile Development/Cross-Plattform-Development	Ü5
13	28.06.2016	Gastdozent(en)	Ü6: Backbone.js
14	05.07.2016	Klausurvorbereitung	Ü6
15	12.07.2016	Klausur PZR1 (Di, 12.07. 14:00 Uhr, Ingeborg-Meising-S.)	-
16	19.07.2016	Klausureinsicht	-
	21.09.2016	Klausur PZR2 (Mi, 21.09. 12:00 Uhr, Raum B101, H Gauß)	-

Ausblick / Nächster Unterricht

- **Vertiefung der Themen, die Sie vorschlagen**
- Bitte bis Samstag in Moodle Themen vorschlagen oder Fragen einstellen

Vertiefungsthemen für ME2 (bis Sa, 21.05.)



Willkommen bei Etherpad Lite der Beuth Hochschule Berlin!

Dieses Text-Pad wird während der Eingabe synchronisiert, so dass jeder bei der Ansicht dieser Seite den gleichen Text sieht. Dies ermöglicht Ihnen die nahtlose Zusammenarbeit an Dokumenten!

Alle Änderungen werden versioniert und sind damit auch wiederherstellbar.

Für ME2:

1. Bitte ergänzen Sie Themen oder Fragen, die Sie gerne **wiederholen** möchten. Schlagen sie gerne auch Themen zur **Vertiefung** vor, die noch nicht angesprochen wurden, aber zum Kurs-Stoff passen.
2. Machen Sie ein x hinter ein Thema, das bereits vorgeschlagen wurde und Sie ebenfalls "gut finden".
3. Ich (Dozent) werde auf Basis Ihre Nennungen eine Auswahl für den Unterricht treffen.

Themen bzw. Fragen für den Unterricht ME2 in der Woche 24./25. Mai:

- Wie nutzt man mySQL oder postgresql mit node?
- Closures
- Wie macht man node.js Server sicher(er)?

Vielen Dank und bis zum nächsten Mal