

EINE EINFÜHRUNG IN PROGRAMMIERUNG VON  
WEBAPPLIKATIONEN MIT JAVASCRIPT.

JAVASCRIPT ADVANCED

FUNKTIONEN, DIE SICH SELBST AUSFÜHREN.  
SIE WERDEN VOR ALLEN ANDEREN FUNKTIONEN INITIALISIERT UND KÖNNEN NEBENBEI  
DAZU VERWENDET WERDEN, PRIVATE KEYS UND METHODEN ZU VERWENDEN.

# IMMEDIATE FUNCTIONS

# IIFE - DIE IMMEDIATE INVOKED FUNCTION EXPRESSION

`()();`

Was, das soll eine Funktion sein?

Ja. Das erste Klammerpaar ist Rangfolge-Klammer (für eine anonyme, sich nach dem Kompilieren sofort selbst ausführende Funktion.)

Das zweite Klammerpaar ist eine Aufruf- und Argumente-Klammer, mit der Argumente in die Funktion übergeben werden können.

Der ganze Ausdruck: **(function (){})()**

# AUFBAU EINER IMMEDIATE FUNCTION

```
var extArgs;  
( function (intArgs) { ... } )(extArgs);  
  
/*  
innerhalb der Immediate Function wird in der Regel eine anonyme  
Funktion platziert, die durch die Immediate Function ausgeführt  
wird. Ihr werden die Argumente übergeben.  
*/
```

# IMMEDIATE FUNCTION MIT RÜCKGABEWERT

```
var extArgs,  
    ExtObj;  
  
extObj = (function (intArgs) {  
    var intObj = {};  
    ...  
    return intObj;  
} )(extArgs);  
  
/*  
Da die Funktion sofort ausgeführt ist, gibt sie anstelle eines  
Funktionszeigers ein Ergebnis zurück.  
*/
```

# EIN OBJEKT MIT PUBLIC UND PRIVATE KEYS.

```
var animal = (function (type, legs, wings, sound) {  
  
    var type  = type  || 'Hund',  
        legs   = legs   || '4',  
        wings  = wings  || '0',  
        sound   = sound  || 'wau'  
;  
    function getType () { return type; }  
    function setType (value) { type = value; }  
    return ({  
        getType : getType,  
        setType : setType,  
        sound   : sound  
    });  
}());  
  
// Getter!  
console.log('Getterzugriff: ' + animal.getType());
```

# JAVASCRIPT OBJEKTORIENTIERUNG

# KONSTRUKTOREN UND PROTOTYPISCHE OBJEKTE

# KONSTRUKTOREN

Eine Funktion, mit der neue Objekte erzeugt werden, muss mit den new-Präfix aufgerufen werden. Dabei wird ein leeres Objekt erzeugt, das auf das neue Objekt verweist.

```
function Tier(a) {  
    this.art = a; // attribut  
    this.method = function () {};  
}  
  
var tier = new Tier('tier');
```

# THIS KANN SICH ÄNDERN

Da `this` sich während der Laufzeit ändern kann, ist es üblich, eine Kopie nach Instanzierung anzulegen.

```
function Tier(a) {  
    var that = this; // das sieht man häufig ...  
    var self = this; // oder auf das  
  
    ...  
};
```

# THIS KANN SICH ÄNDERN

```
function Person (vorname, nachname) {  
    var self = this;  
    self.vorname = vorname;  
    self.nachname = nachname;  
  
    self.vollerName = function () {  
        return self.vorname + ' ' + self.nachname;  
    }  
}  
  
var rudi = new Person ('Rudolf', 'Rentier');  
console.log(rudi.vorname);  
console.log(rudi.nachname);  
console.log(rudi.vollerName());
```

# KONSTRUKTOR MIT ÖFFENTLICHEN UND PRIVATEN EIGENSCHAFTEN

```
function Konto (inhaber, startGuthaben) {  
    var self = this;  
    self.inhaber = inhaber;  
    var guthaben = Startguthaben;  
    var protokoll = [];  
    var addToProtocol (action, betrag, text) {  
        protokoll.push({  
            aktion : action,  
            betrag : betrag,  
            text   : text  
        });  
    };  
    self.einzahlen = function (betrag, text) {  
        addToProtocol('EIN', betrag, text);  
        guthaben += betrag;  
    };  
    self.abheben = function (betrag, text) {  
        addToProtocol('AUS', betrag, text);  
        guthaben -= betrag;  
    };  
    self.getGuthaben = function () {  
        return guthaben;  
    };  
    self.getProtokoll = function () {  
        return protokoll;  
    };  
}
```

# VERWENDEN EINER KONSTRUKTORGFUNKTION

```
var myKonto = new Konto('Susi Sorglos', 200);

myKonto.einzahlen(50);
myKonto.abheben(30=;

var guthaben = myKonto.getGuthaben();
var protokoll = myKonto.getProtokoll();

console.log(myKonto.guthaben); // undefined!
```

# VEREINFACHUNG ZUR ERZEUGUNG EINES OBJEKTES

```
function create(o) {  
    var F = function() {};  
    F.prototype = o;  
    return new F();  
}  
  
// Anwendung:  
var insekt = create(tier);  
// oder nach ES5: Object.create()
```

# VERERBUNG

# PROTOTYPISCHE VEREBUNG

- Die Prototypen Kette
- 1 Hole das Attribut vom Konstruktorobjekte
- 2 Wenn keines da ist, nimm das vom Prototypenobjekt
- 3 Wenn der Prototyp existiert gehe zu 1.
- 4 Ansonsten gibt undefined zurück

Objekt  
(Prototyp)

Funktion  
(Konstruktor)

Objekt  
(abgeleitet)



# VERERBUNG MIT PROTOTYPEN

```
function GiroKonto (inhaber, startguthaben, limit) {  
    var that = this;  
  
    Konto.call(this, inhaler, startguthaben);  
  
    var base_abheben = that.abheben;  
    that.abheben = function (betrag, text) {  
        base_abheben( betrag, '*' + text '*' );  
    };  
};  
  
GiroKonto.prototype = new Konto();  
  
var gk = new GiroKonto('Susi', 100, 300);  
gk.abheben(99, 'test');  
console.log(gk.getGuthaben());
```

# ALTERNATIVE VERERBUNG

```
function Konto (inhaber, startGuthaben) {  
    this.inhaber = inhaber;  
    this.guthaben = guthaben;  
    this.protokoll = [];  
};  
  
Konto.prototype.addToProtocol = function (action, betrag,  
text) { ... };  
Konto.prototype.einzahlen = function (betrag, text) { ... };  
Konto.prototype.abheben = function (betrag, text) { ... };  
Konto.prototype.getGuthaben = function () { ... };  
Konto.prototype.getProtokoll = function () { ... };
```

# ALTERNATIVE VERERBUNG

```
function GiroKonto = function (inhaber, startGutgaben, limit) {  
    Konto.call(this, inhaber, startGuthaben);  
};  
  
GiroKonto.prototype = new Konto();  
  
// Original abheben sichern:  
GiroKonto.prototype.base_abheben = GiroKonto.prototype.abheben;  
  
GiroKonto.prototype.abheben = function (betrag, text) {  
    if ( betrag <= this.guthaben) {  
        this.base_abheben(betrag, '*' + text + '*');  
    };  
};  
  
-> this verweist in dieser Variante auf das GiroKonto
```

# DAS FOR IN PROBLEM

Vererbte Funktionen enthalten beim Durchzählen auch den Prototypen.

```
for (name in object) {  
    if (object.hasOwnProperty(name)) {  
        ...  
    }  
}
```

Das kann mit der `hasOwnProperty` Eigenschaft abgefangen werden.

# ECMA6 - KLASSEN UND VERERBUNG

# ECMA6 - KLASSEN

```
class Person {  
    constructor(vorname, nachname) {  
        this.vorname = vorname;  
        this.nachname = nachname;  
    }  
    // Objekt-Methode  
    toString () {  
        return this.vorname + ' ' + this.nachname;  
    };  
    static beispiel () {  
        return 4711;  
    };  
}  
  
var max = new Person('Max', 'Mustermann');
```

# ECMA6 - VERERBUNG

```
class Mitarbeiter extends Person {  
    constructor(vorname, nachname,mitarbeiterId){  
        super(vorname, nachname);  
        this.mitarbeiterId = mitarbeiterId;  
    };  
  
    toString () {  
        return super.toString() + ' ' + mitarbeiterId;  
    };  
}
```

# SIMPLE CHAINING PATTERN

```
// a class

var Device = function () {
    this.version = '0.1';
    this.os      = 'MacOS X';
    this.browser = 'Chrome';
};
```

# CHAINING METHODS

```
Device.prototype.setVersion = function (version) {  
    this.version = version;  
    return this;  
};
```

```
Device.prototype.setOs = function (os) {  
    this.os = os;  
    return this;  
};
```

```
Device.prototype.setBrowser = function (browser) {  
    this.browser = browser;  
    return this;  
};
```

```
Device.prototype.save = function () {  
    console.log(  
        'saving ' + this.version + ', on ' +  
        this.os + ' with ' + this.browser + '.'  
    );  
    return this;  
};
```

```
// WITHOUT CHAINING
// asynchronous processing!
// the save() might be at the wrong time

var device = new Device();

device.setVersion('1');
device.setOs('MacOS XI');
device.setBrowser('Chrome Xtra');

device.save();
```

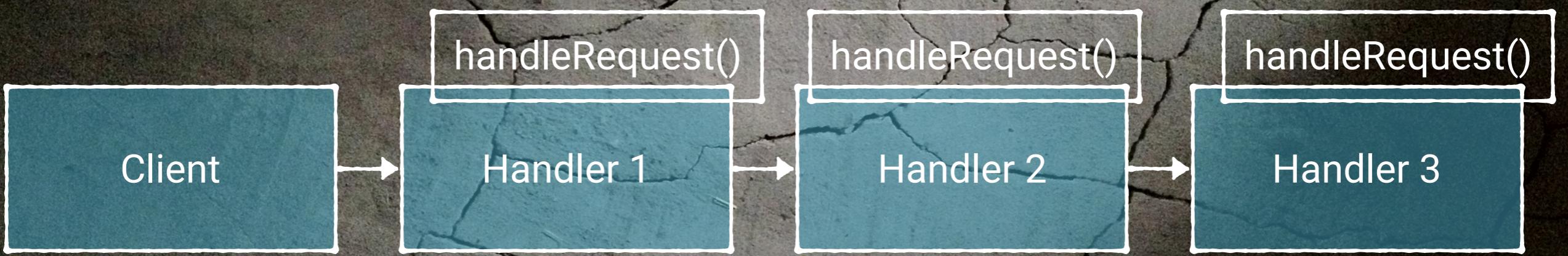
```
// WITH CHAINING
device = new Device();

device
    .setVersion('2')
    .setOs('MacOS XII')
    .setBrowser('Chrome Xtra Large')
    .save();
```

# CHAIN OF RESPONSIBILITY PATTERN

# VERKETTEN VON OBJEKten

- Das „Chain of Responsibility“ Entwurfsmuster stellt eine Kette lose gekoppelter Objekte zur Verfügung, die eine Anfrage abarbeiten.
- Das Muster ist im Wesentlichen eine lineare Suche nach einem Objekt, das Anfragen auch teilweise abarbeiten kann.
- Ein Beispiel ist das Ereignis-Bubbling, in dem ein Ereignis durch eine Folge verschachtelter Controls propagiert wird, von denen einer den Event abarbeiten möchte und kann.
- Das „Chain of Responsibility“ Pattern bezieht sich auf das Verkettungsmuster, das in Javascript häufig eingesetzt.



# TEILNEHMER AM CHAINING PATTERN

- **Client** -- Im Beispiel: Request  
Er startet die Anfrage in eine Kette von Handler-Objekten.
- **Handler** -- Im Beispiel: Request.get()  
definiert eine Schnittstelle, um die Anfragen abzuarbeiten. Die Schnittstelle implementiert eine Objektweitergabe des veränderten Objektes (successor link). (return this).

# EIN CODEBEISPIEL

- Das Beispiel zeigt eine nicht ganz klassische „Chain of Responsibility“, in der alle Handler die Anfrage abarbeiten und verändern. (Klassisch verarbeitet nur eine der Handler-Methoden die Anfrage).
- Der Code zeigt eine elegante Lösung für das Geldausgabe Problem eines Geldautomaten. Ein Kunde möchte 247€ von seinem Konto abheben. Mit welcher Banknotenkombination (100€, 50€, 20€, 10€, 5€) kann der Betrag addiert werden?
- Die Anfrage wird mit dem geforderten Betrag initiiert. Als nächstes durchläuft die Anfrage eine Reihe von get()-Schritten, die jeweils eine Schein-Größe abarbeiten. Jeder Handler bestimmt die Anzahl der möglichen Scheine, zieht den erreichten Teilbetrag von der Restsumme ab und übergibt so an den nächsten Handler.

```
var Request = function(amount) {
    this.amount = amount;
    log.add("Requested: €" + amount + "\n");
}

Request.prototype = {
    get: function(bill) {
        var count = Math.floor(this.amount / bill);
        this.amount -= count * bill;
        log.add("Dispense " + count + " €" + bill + " bills");
        return this;
    }
}
```

```
// log helper

var log = (function() {
  var log = "";

  return {
    add: function(msg) { log += msg + "\n"; },
    show: function() { alert(log); log = ""; }
  }());
});
```

```
function run() {  
    var request = new Request(378);  
  
    request.get(100).get(50).get(20).get(10).get(5).get(1);  
  
    log.show();  
}
```

<http://www.dofactory.com/javascript/>

# CLOSURES

```
// FUNCTION SCOPES
// Variablen, die innerhalb eines Sichtbarkeitsbereichs
// einer äußeren Funktion liegen, können von einer inneren
// Funktionsdefinition aus angesprochen werden.

function init() {
    var name = "Mozilla";
    function display() {
        console.log(name);
    }
    display();
}

init();
```

# CLOSURE - FUNKTIONSABSCHLUSS

```
function makeAFunction() {  
  var name = "Mozilla";  
  
  function display() {  
    console.log(name);  
  }  
  
  return display;  
}  
  
var myFunction = makeAFunction();  
myFunction();
```

# CLOSURE - FUNKTIONSABSCHLUSS

- Die innere `display()`-Funktion wird über `return` zurückgegeben wird, bevor diese ausgeführt wird.
- Die Funktion `myFunction` wurde zu einer Closure - eine Funktion und die Umgebung, in der die Funktion erstellt wurde.
- Die Umgebung schließt alle lokalen Variablen ein, die zur Zeit der Closure-Erstellung im selben Sichtbarkeitsbereich lagen.

# EINE MAKEADDER-FACTORY

```
function makeAdder(x) {  
    return function(y) {  
        return x + y;  
    };  
}  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
print( add5(2) ); // 7  
print( add10(2) ); // 12
```

# EIN PRAKTISCHES CLOSURE

```
function makeSizer(size) {  
    return function() {  
        document.body.style.fontSize = size + 'px';  
    };  
}  
  
var size12 = makeSizer(12);  
var size14 = makeSizer(14);  
var size16 = makeSizer(16);  
  
document.getElementById('size-12').onclick = size12;  
document.getElementById('size-14').onclick = size14;  
document.getElementById('size-16').onclick = size16;  
  
<a href="#" id="size-12">12</a>  
<a href="#" id="size-14">14</a>  
<a href="#" id="size-16">16</a>
```

# EMULIEREN VON PRIVATEN METHODEN MIT CLOSURES

```
var makeCounter = function() {  
    var privateCounter = 0;  
  
    function changeBy(val) {  
        privateCounter += val;  
    }  
  
    return {  
        increment: function() {  
            changeBy(1);  
        },  
        decrement: function() {  
            changeBy(-1);  
        },  
        value: function() {  
            return privateCounter;  
        }  
    }  
};
```

# MEHRFACH AUFRUFEN - CLOSURE VARIABLEN LAUFEN UNABHÄNGIG VONEINANDER ALS INSTANZEN!

```
var Counter1 = makeCounter();
var Counter2 = makeCounter();

console.log(Counter1.value()); /* 0 */

Counter1.increment();
Counter1.increment();

console.log(Counter1.value()); /* 2 */

Counter1.decrement();

console.log(Counter1.value()); /* 1 */
console.log(Counter2.value()); /* 0 */
```

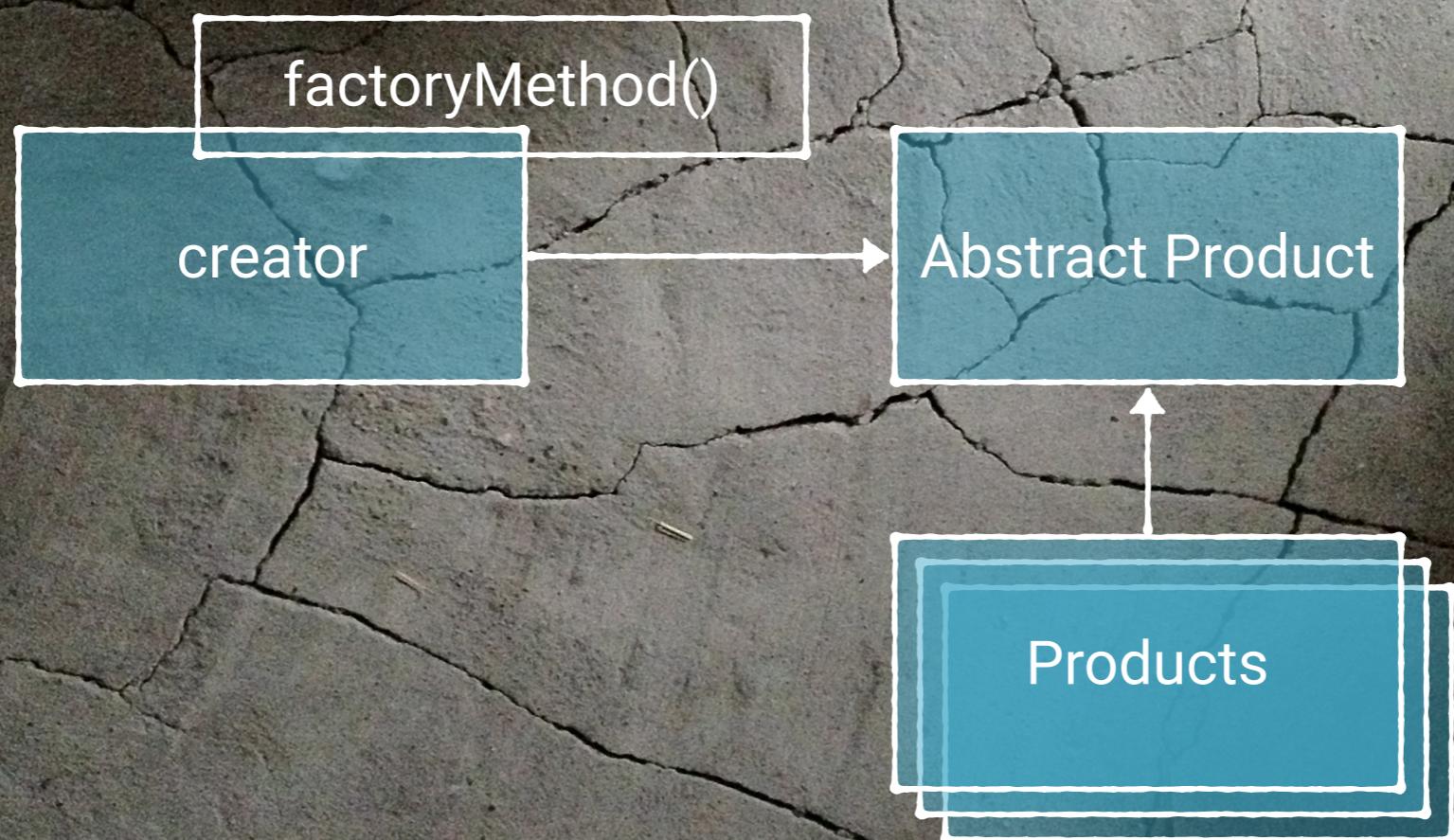
# FACTORY PATTERN

# OBJEKTE MIT ANWEISUNGEN ERZEUGEN

- Eine Factory Methode erzeugt neue Objekte, die jeweils vom Client instruiert werden.
- Der new - Operator erzeugt Objekte (Clients) aus Konstruktoren.
- There are situations however, where the client does not, or should not, know which one of several candidate objects to instantiate.
- The Factory Method allows the client to delegate object creation while still retaining control over which type to instantiate.

# ERWEITERBARKEIT

- The key objective of the Factory Method is extensibility.
- Factory Methods are frequently used in applications that manage, maintain, or manipulate collections of objects that are different but at the same time have many characteristics (i.e. methods and properties) in common.
- An example would be a collection of documents with a mix of Xml documents, Pdf documents, and Rtf documents.



# TEILNEHMER AM FACTORY PATTERN

- **Creator** -- Im Codebeispiel: Factory the 'factory' object that creates new products implements 'factoryMethod' which returns newly created products
- **AbstractProduct** -- not used in JavaScript declares an interface for products
- **ConcreteProduct** -- Im Codebeispiel: Employees the product being created all products support the same interface (properties and methods)

# EIN CODEBEISPIEL

- In this JavaScript example the Factory object creates four different types of employees. Each employee type has a different hourly rate. The `createEmployee` method is the actual Factory Method. The client instructs the factory what type of employee to create by passing a type argument into the Factory Method.
- The `AbstractProduct` in the diagram is not implemented because Javascript does not support abstract classes or interfaces. However, we still need to ensure that all employee types have the same interface (properties and methods).
- Four different employee types are created; all are stored in the same array. Each employee is asked to say what they are and their hourly rate.

# A EMPLOYEE FACTORY

```
function Factory() {  
    this.createEmployee = function (type) {  
        var employee;  
  
        if (type === "fulltime") {  
            employee = new FullTime();  
        } else if (type === "parttime") {  
            employee = new PartTime();  
        } else if (type === "temporary") {  
            employee = new Temporary();  
        } else if (type === "contractor") {  
            employee = new Contractor();  
        }  
  
        employee.type = type;  
  
        employee.say = function () {  
            log.add(this.type + ": rate " + this.hourly + "/hour");  
        }  
  
        return employee;  
    }  
}
```

```
var FullTime = function () {
    this.hourly = "$12";
};

var PartTime = function () {
    this.hourly = "$11";
};

var Temporary = function () {
    this.hourly = "$10";
};

var Contractor = function () {
    this.hourly = "$15";
};
```

```
// log helper
var log = (function () {
    var log = "";

    return {
        add: function (msg) { log += msg + "\n"; },
        show: function () { alert(log); log = ""; }
    }());
});
```

```
function run() {  
    var employees = [];  
    var factory = new Factory();  
  
    employees.push(factory.createEmployee("fulltime"));  
    employees.push(factory.createEmployee("parttime"));  
    employees.push(factory.createEmployee("temporary"));  
    employees.push(factory.createEmployee("contractor"));  
  
    for (var i = 0, len = employees.length; i < len; i++) {  
        employees[i].say();  
    }  
  
    log.show();  
}
```

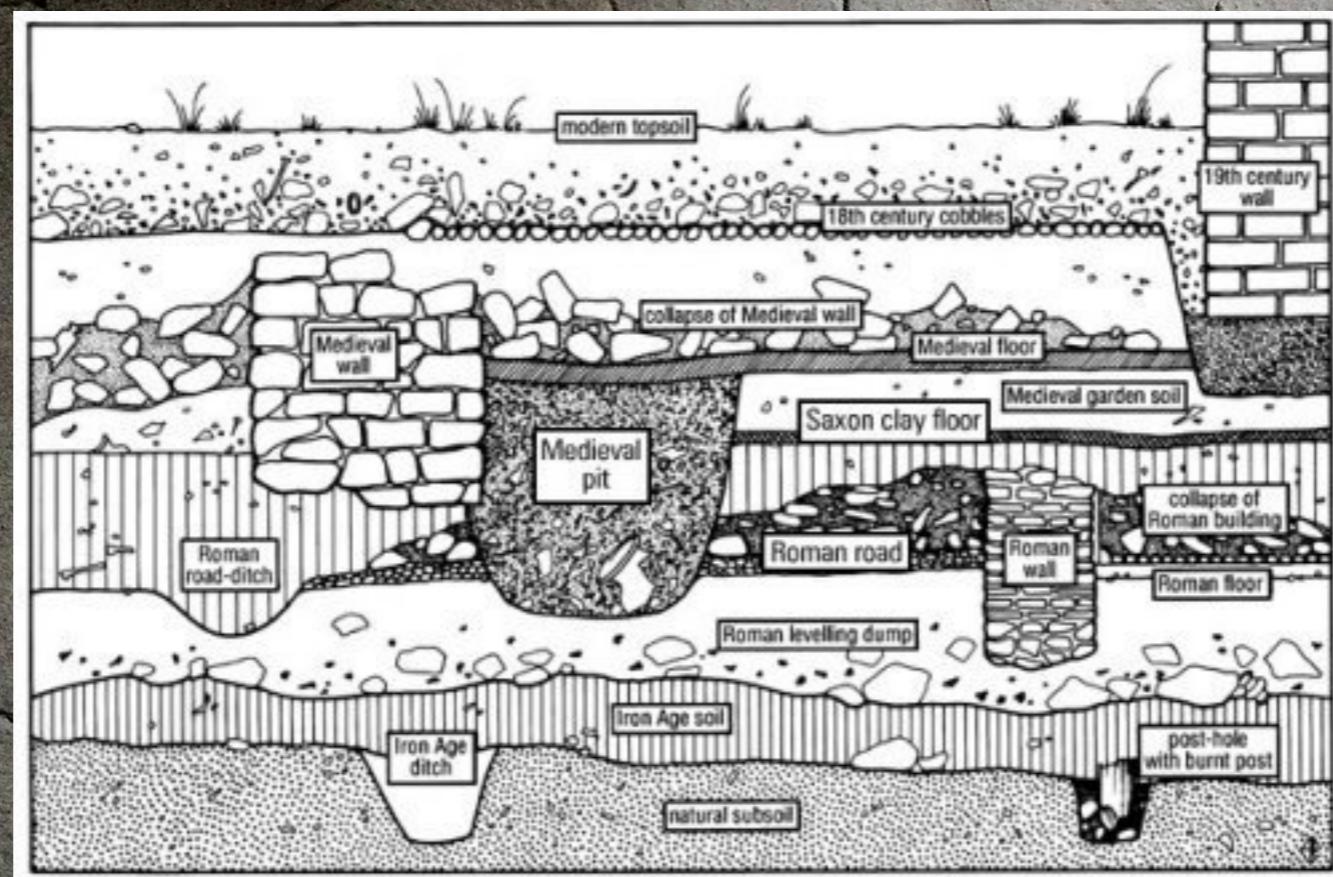
# UNITTESTS FÜR JAVASCRIPT

# QUNIT

# Software Architektur?

- Das Unit Testing beschreibt Methoden zur Prüfung und Validierung, mit denen ein Programmierer testen kann, ob einzelne Units (Einheiten) des Quelltextes gebrauchsfertig (stabil) sind.
- Eine Unit ist der kleinste testbare Teil einer Anwendung.
- In der prozeduralen Programmierung ist eine Unit eine bestimmte Funktion oder Prozedur.





# SOFTWARE ARCHITEKTUR!

- Das Unit Testing beschreibt Methoden zur Prüfung und Validierung, mit denen ein Programmierer testen kann, ob einzelne Units (Einheiten) des Quelltextes gebrauchsfertig (stabil) sind.
- Eine Unit ist der kleinste testbare Teil einer Anwendung.
- In der prozeduralen Programmierung ist eine Unit eine bestimmte Funktion oder Prozedur.

# UNIT TESTING

- Das Unit Testing beschreibt Methoden zur Prüfung und Validierung, mit denen ein Programmierer testen kann, ob einzelne Units (Einheiten) des Quelltextes gebrauchsfertig (stabil) sind.
- Eine Unit ist der kleinste testbare Teil einer Anwendung.
- In der prozeduralen Programmierung ist eine Unit eine bestimmte Funktion oder Prozedur.

# WARUM SOLLTE CODE GETESTET WERDEN?

- Wer noch niemals Code getestet hat, der probiert seinen Code einfach direkt in der Webseite aus, klickt ein wenig herum und schaut, ob irgendein Problem auftritt, das dann ge"fixt" werden kann.
- Diese Methode bringt einige Probleme mit sich.

# 1. ES IST WIRKLICH LÄSTIG!

- Klicken ist nicht wirklich einfach! Ist alles angeklickt? Jedes Mal? Nichts vergessen?
- Es ist sehr wahrscheinlich, dass beim Wiederholen eines Tests das eine oder andere vergessen wird.

## 2. ALTE FEHLER BLEIBEN UNENTDECKT

- Ein Programmteil, das erfolgreich getestet wurde, enthält durchaus noch Fehler. Diese tauchen aber erst auf, nachdem ein Codeupdate durchgeführt wurde.
- Solche Fehler (Regressions genannt) sind nicht einfach zu finden.

# UNITTEST BEHEBEN SOLCHE PROBLEME

- Unitests finden grundsätzliche Fehler in Codes.
- Falls nach einer Änderung des Codes derselbe nicht mehr korrekt arbeitet (auch wenn es vorher scheinbar so war), so wird einer Tests sicher den Fehler anzeigen.
- Dann ist es einfach herauszubekommen, wo und wie der Fehler verursacht wird - das Beheben wird einfach sein.

# UNITTESTS TESTEN DIE ERGEBNISSE VON FUNKTIONEN

```
function add (a, b) {  
    return (a+b);  
}  
var result = add (3,4);          // result muss 7 sein! === 7  
var resut = add(3, "4");        // result ist "34"           !== 7  
var result = add ("3", "4");    // result ist "34"           !== 7  
var result = add ("drei", "vier"); // result ist "dreivier" !  
                                == 7  
var result = add (drei, vier)   // reference error: drei is  
not defined
```

# EINE TESTBARE FUNKTION MUSS IM BESTEN FALL EINEN RÜCKGABEWERT BESITZEN

```
var eventHandler = function (event) {  
    var url = $(event.target).attr(href);  
}
```

Hm. Das ist problematisch!

```
function getValueFromAttr(object, attribute) {  
    return $(object).attr(attribute)  
}  
  
var url = getValueFromAttribute(event.target, 'href');  
  
test('test_url_isPage.html', function() {  
    strictEqual(getValueFromAttribute(event.target, 'href'),  
    'page.html');  
})
```

# AUFBAU EINES UNITTESTS MIT QUNIT

# QUNIT - HERKUNFT

- Qunit wurde ursprünglich von John Resig als Teil des jQuery Projektes entwickelt.
- Seit 2008 ist es ein eigenständiges Projekt innerhalb der jQuery Foundation und wird von Jörn Zäfferer geführt.
- Die Assertions von Qunit folgen der CommonJS Unit Testing Spezifikation. [[http://wiki.commonjs.org/wiki/Unit\\_Testing/1.0](http://wiki.commonjs.org/wiki/Unit_Testing/1.0)]

# DIE TESTSUITE

```
...<head>
    <title>QUnit Test Suite</title>
    <link ... href="_qunit/qunit.css">
    <script ... src="_qunit/qunit.js"></script>

    <script ... src="myProject.js"></script>
    <script ... src="myTests.js"></script>
</head>
<body>
    <h1 id="qunit-header">QUnit Test Suite</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests"></ol>
</body>
```

# TESTS MÜSSEN ERGEBNISSE VORHERSAGEN

- Die Bausteine eines Unittest sind die sogenannten Assertions, auf deutsch Behauptungen oder Annahmen

# ASSERTIONS - BEHAUPTUNG

- Eine assertion ist eine Aussage, die das zurückgegebene Ergebnis einer codierten Funktion vorhersagt.
- Wird die Vorhersage nicht erfüllt, ist klar, dass in der Funktion etwas falsch ist.

# WELCHE METHODEN STEHEN FÜR TESTS ZUR VERFÜGUNG? ASSERTION TYPES

# ASSERT.OK()

- Ein einfacher Test mit einem boolschen Ergebnis. Er entspricht einem assert.ok() in CommonJS oder auch einem JUnit assertTrue().
- Der Test läuft erfolgreich durch, wenn das erste Argument truthy ist

# EINE EINFACHE TEST - FUNKTION MIT OK()

```
// Funktion, die getestet werden soll
function isEven(val) {
    return val % 2 === 0;
}

QUnit.test('test_isEven_isTrue', function() {
    assert.ok(isEven(0), 'Null ist eine gerade Zahl');
    assert.ok(isEven(2), 'Zwei ebenso');
    assert.ok(isEven(-4), 'Oder minus Vier');
    assert.ok(!isEven(1), 'Eins ist keine gerade Zahl');
    assert.ok(!isEven(-7), 'Auch nicht minus Sieben');
})

QUnit.test('test_isEven_isNumber', function() { ... })
```

## QUnit Test Suite ■ noglobals ■ notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_1) AppleWebKit/535.1 (KHTML, like Gecko)  
Chrome/14.0.835.186 Safari/535.1

Tests completed in 21 milliseconds.  
5 tests of 5 passed, 0 failed.

### 1. isEven() (0, 5, 5) Rerun

- 1. Null ist eine gerade Zahl
- 2. Zwei ebenso
- 3. Oder minus Vier
- 4. Eins ist keine gerade Zahl
- 5. Auch nicht minus Sieben

# WENN EIN TEST FEHLSCHLÄGT ...

```
// Funktion, die getestet werden soll
function isEven(val) {
    return val % 2 === 0;
}

Qunit.test('isEven()', function() {
    assert.ok(isEven(0), 'Null ist eine gerade Zahl');
    assert.ok(isEven(2), 'Zwei ebenso');
    assert.ok(isEven(-4), 'Oder minus Vier');
    assert.ok(!isEven(1), 'Eins ist keine gerade Zahl');
    assert.ok(!isEven(-7), 'Auch nicht minus Sieben');

    // Fehler!
    assert.ok(isEven(3), 'Drei ist eine gerade Zahl');
})
```

## QUnit Test Suite ■ noglobals ■ notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_1) AppleWebKit/535.1 (KHTML, like Gecko)  
Chrome/14.0.835.186 Safari/535.1

Tests completed in 20 milliseconds.  
5 tests of 6 passed, 1 failed.

### 1. `isEven()` (1, 5, 6) Rerun

- 1. Null ist eine gerade Zahl
- 2. Zwei ebenso
- 3. Oder minus Vier
- 4. Eins ist keine gerade Zahl
- 5. Auch nicht minus Sieben
- 6. Drei ist eine gerade Zahl

# EQUAL()

- Eine Vergleichsbehauptung, etwa `result == "vier"`, die nicht typensicher vergleicht.

# NOTEQUAL()

- Eine nicht typensichere Vergleichsbehauptung, die prüft, ob Werte in einem Vergleich einander nicht entsprechen.
- `result != "vier";`

# DIE COMPARISON ASSERTION MIT EQUAL, NOTEQUAL

```
Qunit.test('assertions', function() {
    assert.equal( 1, 1, 'Eins ist gleich Eins');
})

Qunit.test( "a test", function() {
    assert.notEqual( 1, "2", "String '2' and number 1 don't have
the same value" );
});

Qunit.test( "equal test", function() {
    assert.equal( 0, 0, "Zero; equal succeeds" );
    assert.equal( "", 0, "Empty, Zero; equal succeeds" );
    assert.equal( "", "", "Empty, Empty; equal succeeds" );
    assert.equal( 0, 0, "Zero, Zero; equal succeeds" );

    assert.equal( "three", 3, "Three, 3; equal fails" );
    assert.equal( null, false, "null, false; equal fails" );
});
```

## QUnit Test Suite ■ noglobals ■ notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_1) AppleWebKit/535.1 (KHTML, like Gecko)  
Chrome/14.0.835.186 Safari/535.1

Tests completed in 19 milliseconds.  
7 tests of 7 passed, 0 failed.

1. [isEven\(\)](#) (0, 6, 6) Rerun
2. [assertions\(\)](#) (0, 1, 1) Rerun

1. Eins ist gleich Eins  
Expected: 1

# ... UND MIT FEHLER

```
Qunit.test('assertions', function() {  
    assert.equal( 2, 1, 'Eins ist gleich Eins');  
})
```

## QUnit Test Suite ■ noglobals ■ notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_1) AppleWebKit/535.1 (KHTML, like Gecko)  
Chrome/14.0.835.186 Safari/535.1

Tests completed in 22 milliseconds.  
6 tests of 7 passed, 1 failed.

1. [isEven\(\)](#) (0, 6, 6) Rerun

2. [assertions](#) (1, 0, 1) Rerun

1. Eins ist gleich Eins

Expected: 1

Result: 2

Diff: 1 2

Source: at Object.<anonymous>  
(file:///Applications/XAMPP/xamppfiles/htdocs/comdirect/qunit.

Die Vergleichsbehauptung verwendet ein “==” um Parameter zu vergleichen, deshalb ist sie nicht typensicher und kann weder Arrays noch Objekte behandeln:

```
Qunit.test('test', function() {  
    assert.equals( {}, {}, 'schlägt fehl, da zwei Objekte');  
    assert.equals( {a: 1}, {a: 1} , 'schlägt fehl');  
    assert.equals( [], [], 'schlägt fehl, da zwei Arrays');  
    assert.equals( [1], [1], 'schlägt fehl');  
})
```

# STRICTEQUAL()

- Eine Vergleichsbehauptung, die typensicher vergleicht.
- Sie eignet sich für Wertevergleiche von Typ number, string oder boolean. Aber nicht für Arrays oder Objekte.

# NOTSTRICTEQUAL()

- Eine Vergleichsbehauptung, die typensicher vergleicht, aber Ungleichheit erwartet.

# STRICTEQUAL, NOTSTRICTEQUAL

```
Qunit.test( "a test", function() {  
    assert.strictEqual( 1, 1, "number 1 and number 1 have the  
same value" );  
});  
  
Qunit.test( "a test", function() {  
    assert.notStrictEqual( 1, "1", "String '1' and number 1  
don't have the same value" );  
});
```

# DEEPEQUAL()

- Eine tiefe rekursive Vergleichsbehauptung, die für sämtliche Datentypen herangezogen werden kann: einfache Typen, aber auch Arrays, Objekte, Reguläre Ausdrücke, Datumsobjekte und Funktionen.

# DIE IDENTICAL ASSERTION MIT DEEPEQUAL()

deepEqual() verwendet '===' für den typensicheren Vergleich, damit können auch Objekte und Arrays verglichen werden.

```
Qunit.test('test', function() {
    assert.deepEqual( {}, {}, 'läuft, Object gleichen
Inhalts');
    assert.deepEqual( {a: 1}, {a: 1} , 'läuft');
    assert.deepEqual( [], [], 'läuft, Array gleichen
Inhalts');
    assert.deepEqual( [1], [1], 'läuft');
})
```

# NOTDEEPEQUAL()

- Eine inverse tiefe rekursive Vergleichsbehauptung, die für sämtliche Datentypen herangezogen werden kann: einfache Typen, aber auch Arrays, Objekte, Reguläre Ausdrücke, Datumsobjekte und Funktionen.

# NOTDEEPEQUAL()

```
Qunit.test( "notDeepEqual test", function() {  
    var obj = { foo: "bar" };  
  
    assert.notDeepEqual( obj, { foo: "bla" }, "Different  
object, same key, different value, not equal" );  
});
```

# PROPEQUAL(), NOTPROPEQUAL()

- Ein strikter Vergleich vom Objekteigenschaften. Dabei können auch Objekte mit verschiedenen Konstruktoren und Prototypen herangezogen werden.

# THROWS()

- Mit throws() kann geprüft werden, ob eine Callbackfunktion beim Ausführen einen Fehler wirft.

# THROWS()

```
Qunit.test( "throws", function() {  
  
    function CustomError( message ) { this.message = message; }  
    CustomError.prototype.toString = function() { return this.message; };  
  
    throws(  
        function() { throw "error" },  
        "throws with just a message, no expected"  
    );  
  
    throws(  
        function() { throw new CustomError(); },  
        CustomError,  
        "raised error is an instance of CustomError"  
    );  
  
    throws(  
        function() { throw new CustomError("some error description"); },  
        /description/,  
        "raised error message contains 'description'"  
    );  
});
```

# STRUKTURIERUNG VON ASSERTIONS

- Test können sehr umfangreich werden, daher ist es sinnvoll sie in verschiedene Testfälle (test cases) zu gliedern, von denen jeder eine einzelne Funktionalität abprüfen soll.
- So können Test in Modulen organisiert werden.

# MODULE()

```
Qunit.module('Modul A');
Qunit.test('isEven()', function() {
    assert.ok(isEven(0), 'Null ist eine gerade Zahl');
    assert.ok(isEven(2), 'Zwei ebenso');
    ...
})
Qunit.module('Modul B');
Qunit.test('assertions', function() {
    assert.equals( 2, 1, 'Eins ist gleich Eins');
})
Qunit.test('assertions2', function() {
    assert.equals( 1, 1, 'Eins ist gleich Eins');
})
```

# QUnit Test Suite

■ noglobals ■ notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_1) AppleWebKit/535.1 (KHTML, like Gecko)  
Chrome/14.0.835.186 Safari/535.1

Tests completed in 21 milliseconds.  
1 tests of 2 passed, 1 failed.

## 1. Modul B: assertions (1, 0, 1) Rerun

1. Eins ist gleich Eins

Expected: 1

Result: 2

Diff: 1 2

Source: at Object.<anonymous>  
(file:///Applications/XAMPP/xamppfiles/htdocs/comdirect/qunit.  
filter=Modul%20B%3A%20assertions:33:2)

## 2. Modul B: assertions2 (0, 1, 1) Rerun

1. Eins ist gleich Eins

Expected: 1

GEPFLOGENHEITEN BEIM ANLEGEN EINES TESTS.

TRIPLE A

# GLIEDERN SIE TESTREIHEN IN MODULE

```
Qunit.module('result testing in zen.fn');  
Qunit.module('type testing in zen.fn');  
Qunit.module('DOM testing in zen.fn');
```

# AUFBAU EINES TESTNAMENS

Test -> Modul -> Submodul->Funktionsname->Assertion  
test\_zen\_fn\_add\_returnIs3  
test\_zen\_fn\_add\_typeIsNumber  
...

FÜR EIN ERGEBNIS SIND MEIST VIELE TESTS NÖTIG.  
TESTEN SIE STANDARD- UND EXTREMWERTE.

```
test_zen_myFunc_add_returnIs3
// TEST 1: Integerwerte
    var a = 1;
    var b = 2;
// TEST 2 - Nachkommastellen
    var a = 1.33333333;
    var b = 1.66666667;
// TEST 3 - Große Zahlen
    var a = 2123456789423456781897;
    var b = -2123456789423456781894;
```

# TRIPLE A

- Ein Test sollte den drei A's folgen:
- Assert - Behauptung aufstellen
- Arrange - Vorbereitungen treffen
- Act - Test ausführen

# AUFBAU EINES TESTS NACH TRIPLE A

```
Qunit.test(  
  'test_zen_myFunc_add_resultIs3',  
  function () {  
    // ASSERTION  
    var assertion = 3;  
    // ARRANGE (set up)  
    var a = 1; var b = 2;  
    // ACT  
    assert.equal( zen.myFuncs.add(a,b), assertion,  
      '3 ist die Summe von ' + a + ' und ' + b);  
  }  
  // ggf. aufräumen (tear down)  
);
```

- Tests müssen einfach sein.
- Sie dürfen keine Fehler haben
- Benutzen Sie deshalb in Tests keine Kontrollstrukturen für einen Programmablauf. Schreiben Sie einfach einen weiteren Test. Oder mehrere.

# ASYNCHRONES TESTEN

- Da es zum Beispiel mit AJAX Aufrufen oder Funktionen mit setTimeout() und setInterval() zu Zeitverzögerungen kommt, werden dafür sinnvollerweise asynchrone Tests geschrieben.

# ASYNCRONES TESTEN

```
test('asynchronous_test', function() {  
    // Erst den Test anhalten  
    stop();  
  
    setTimeout(function() {  
        ok(true);  
  
        // Nach dem Aufruf der Behauptung  
        // kann der Test fortgesetzt werden  
        start();  
    }, 100)  
})
```

## QUnit Test Suite ■ noglobals■ notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_1) AppleWebKit/535.1 (KHTML, like Gecko)  
Chrome/14.0.835.186 Safari/535.1

Tests completed in 134 milliseconds.  
1 tests of 1 passed, 0 failed.

1. [asynchronous\\_test \(0, 1, 1\)](#) Rerun

1. okay

# EINFACHER MIT ASYNCTEST()

Da das Aufrufen der stop() Methode beim asynchronen Testen allgemein ist, gibt es eine eigene Shortcut–Methode: asyncTest().

```
asyncTest('asynchronous_test', function() {  
    setTimeout(function() {  
        ok(true);  
  
        // Nach dem Aufruf der Behauptung  
        // wird der Test fortgesetzt  
        start();  
    }, 100)  
})
```

# TESTEN VON FUNKTIONEN MIT CALLBACK

- setTimeout() ruft seinen eigenen Callback auf ... aber wie geht ein Test auf eine Funktion, die selbst einen Callback besitzt?
- Hier muss der Test warten, bis der Callback ausgeführt wird, bevor er gestartet werden kann. Andernfalls hängt sich das Unittesting ohne Ergebnis auf.

# TESTEN VON FUNKTIONEN MIT CALLBACK

```
// Eine Funktion mit Callback
function ajax(successCallback) {
    $.ajax({
        url: 'server.php',
        timeout : 1000,
        success: successCallback
    });
}

test('asynchronous test', function() {
    // Stoppt den Test und gibt einen Fehler aus
    // falls innerhalb einer Sekunde kein Callback kommt
    stop(1100);

    ajax(function() { ...
        start();
    })
})
```

# MEHRERE ASYNCHRONE FUNKTIONEN?

Genügend Zeit geben ...

```
test('asynchronous test', 3, function() {
  stop();
  ajax(function() {
    ok(true);
  })
  ajax(function() {
    ok(true);
    ok(true);
  })
  setTimeout(function() {
    start();
  }, 2000);
})
```

# VON DREI CALLBACKS KOMMEN NUR ZWEI?

- Es ist möglich, dass von den Callbacks nur ein Teil ausgeführt wird.
- In diesem Fall muss der Test wissen, wieviele Funktionen getestet werden sollen. Dies kann mit der Methode `expect()` angekündigt werden.

# EXPECT()

```
test('asynchronous_test', function() {
    stop();
    // QUnit mitteilen, dass 3 Behauptungen geprüft werden
    expect(3);

    ajax(function() {
        ok(true);
    })
    ajax(function() {
        ok(true);
        ok(true);
    })
    setTimeout(function() {
        start();
    }, 2000);
})
```

# EXPECT() ALS SHORTCUT

```
// QUnit mitteilen, dass 3 Behauptungen geprüft werden

test('asynchronous_test', 3, function() {
    // Pause the test
    stop();
    ajax(function() {
        ok(true);
    })
    ajax(function() {
        ok(true);
        ok(true);
    })
    setTimeout(function() {
        start();
    }, 2000);
})
```