



# JavaScript

```
var foo = function() {  
  
};
```

It's not C!

# JavaScript

- Einführung
  - Don't/Do, Good/Bad, Tools, Environments
- Scope, Closure und Lambda
- prototypbasiertes Objektmodell
- Fallen

# Einführung

# Einführung

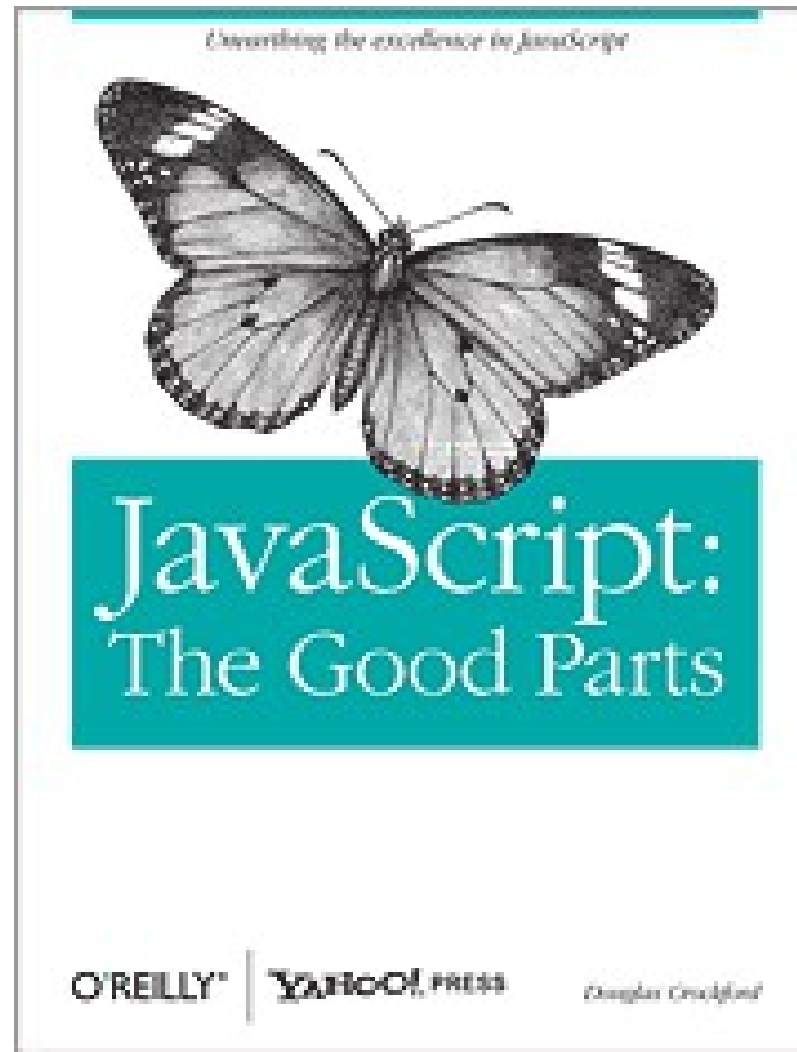
- 1995: Mocha → LiveScript → JavaScript
  - Netscape (Brendan Eich)
  - Marketing driven name (Java Hype)
  - DHTML
- 2005: AJAX → Ajax
  - Jesse James Garrett
  - ECMA Script (auch Flash)
  - Libraries: Prototype, jQuery, YUI, ...

# Good

- Funktionen: first-class, verschachtelt, Closures
- Dynamische Typisierung: Werte haben Typ, nicht Variablen
- Dynamische Objekte: Zur Laufzeit veränderbar (Methoden/Properties)
- Ausdrucksstarke Objektliteral Syntax: **JSON**  
**var** o = {prop1: 1, prop2: 'foo'};

# Bad

- Model basiert auf globalen Variablen
  - „Globals are evil“ aber in JavaScript fundamental
  - JavaScript bietet Mittel um das zu entschärfen
- Der Name / die Syntax
  - Hat mehr mit Lisp gemeinsam als mit Java
  - Lisp in den Kleidern von C
  - Verleitet zu prozeduralem statt funktionalem Code
- s. Crockford für mehr



**Working with the Shallow Grain of JavaScript**  
von *Douglas Crockford*  
ISBN-10: 0596517742 - ISBN-13: 978-0596517748



# Don't

- SelfHTML: HTML ok, JavaScript schlecht
- Die meisten Bücher zum Thema JavaScript
  - Alles was älter als 10 Jahre ist
  - Ausnahmen: Ressig, Crockford, „High Performance JavaScript“, „jQuery in Action“
- Vorsicht bei Google: Viele Treffer, manchmal outdated Lösungen die man heute nicht mehr verwenden will.

# Do

- Most Misunderstood Programming Language
- Mozilla Developer Network (MDN)
- Learning Advanced JavaScript (Resig)
- Yahoo! Developer Network (Videos)
- Read Code (e.g. jQuery, YUI etc.)
  - JS Libs Deconstructed
- Secrets of the JavaScript Ninja (Resig)

# Tools

- Firefox
  - Firebug Extension
  - ~~Venkman Debugger Extension~~
- Alle Browser bieten mittlerweile Dev-Tools
- Editor: Atom, VisualStudio Code, InettliJ, Netbeans, ...
- Code linting is essential!
  - JSLint
  - JSHint

# Runtime Enviroments

- Browser :-)
- [APE](#) - Ajax Push Engine
- [node.JS](#) - Non-blocking IO & Google V8
- [Rhino](#) - JavaScript for Java
- [CouchDB](#) - Views in JavaScript (map/reduce)
  - [Spidermonkey](#) - JavaScript-C Engine

# Scope, Closure und Lambda

# Scope, Closure und Lambda

- Lexikaler Funktions-Scope (kein Block-Scope wie C, Java etc.)
- Eine Funktion ist ein Closure
  - Es „merkt“ sich was um es herum passiert
- Funktionen können als Argument übergeben werden. (Nicht nur der Return-Wert kann übergeben werden.)

# Scope

```
function badScope(param) { // Scope Begin
    if (param) {
        var foo = true;
    }

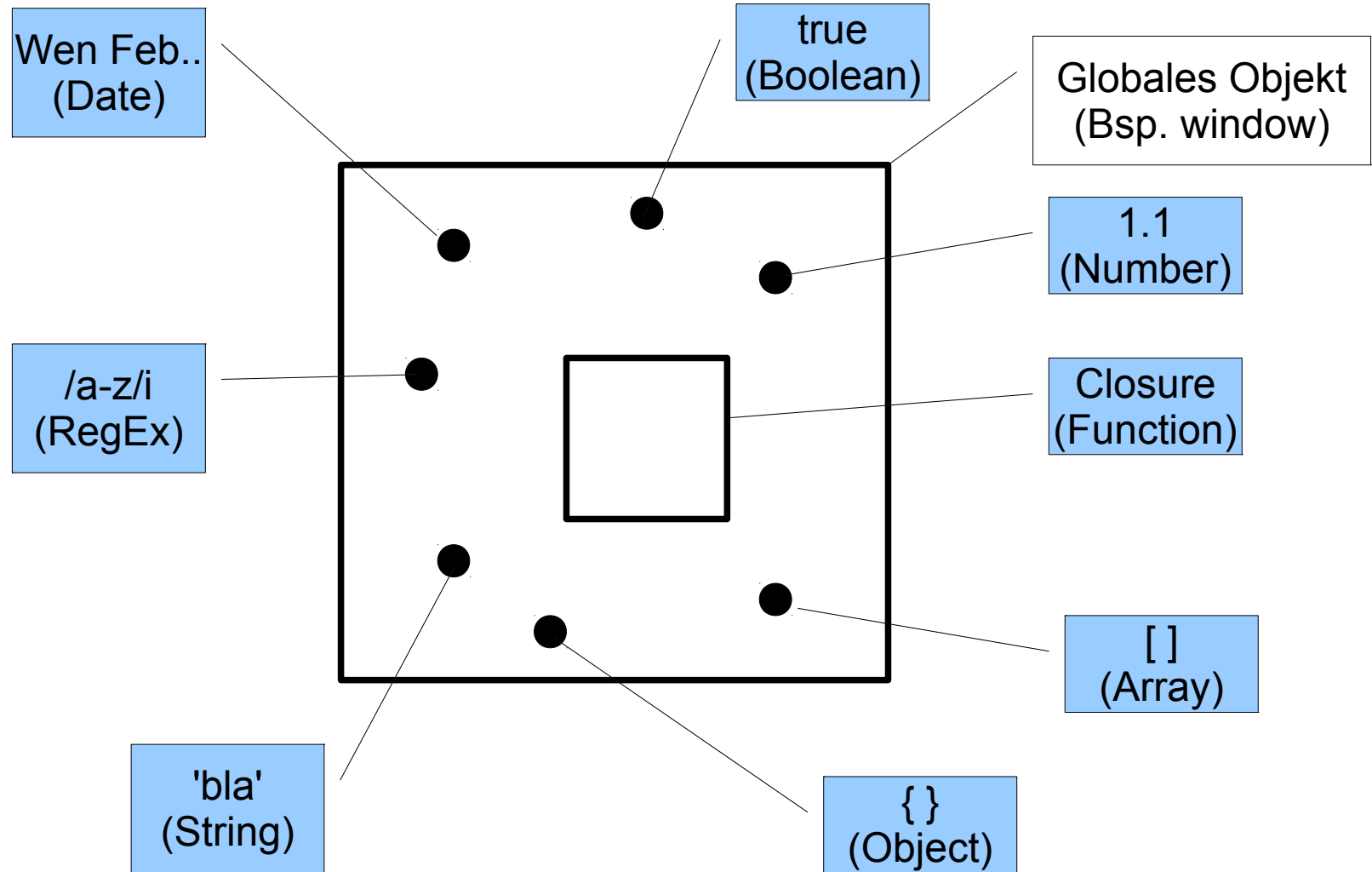
    // ...

    if (foo) {

    }
} // Scope Ende
```

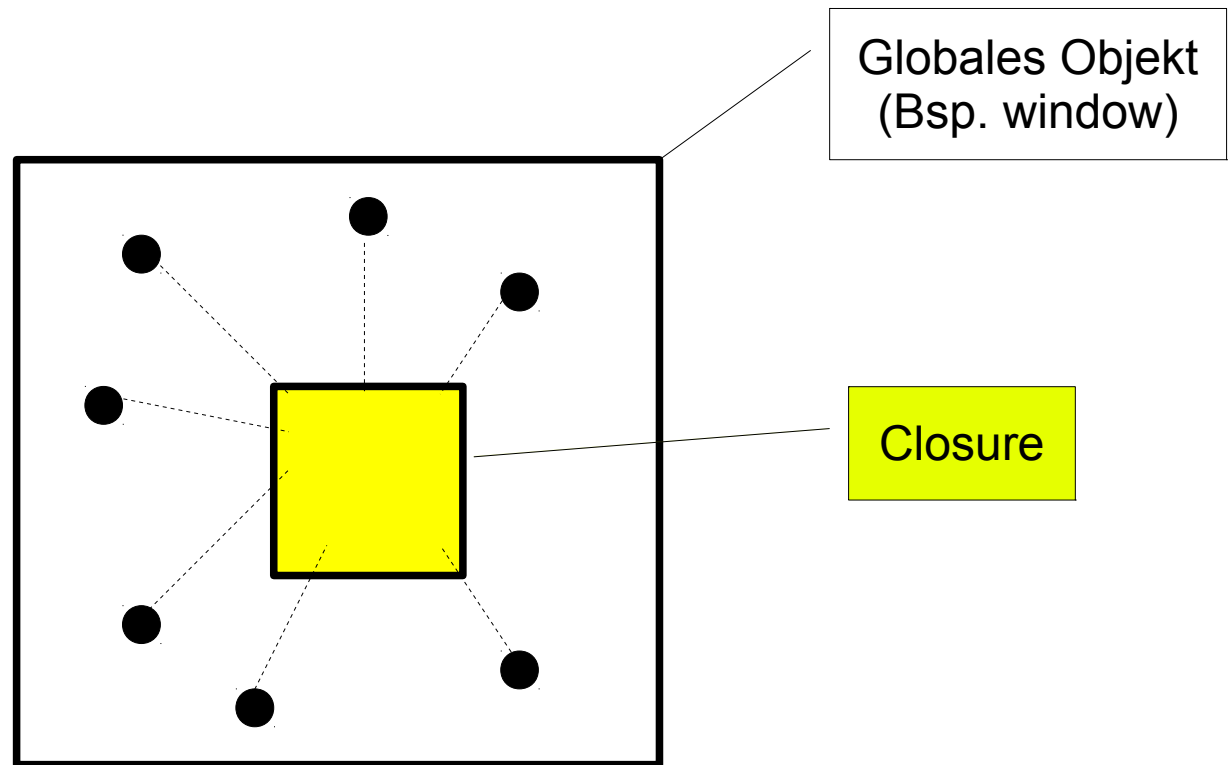
- Würde in C/Java zu Compilerwarnungen führen
- `foo` ist ausserhalb des Blocks (`if`) sichtbar
- Wenn `param false` → `foo === undefined`
- Alles außerhalb ist global Scope (z.B. `window`)

# Closure

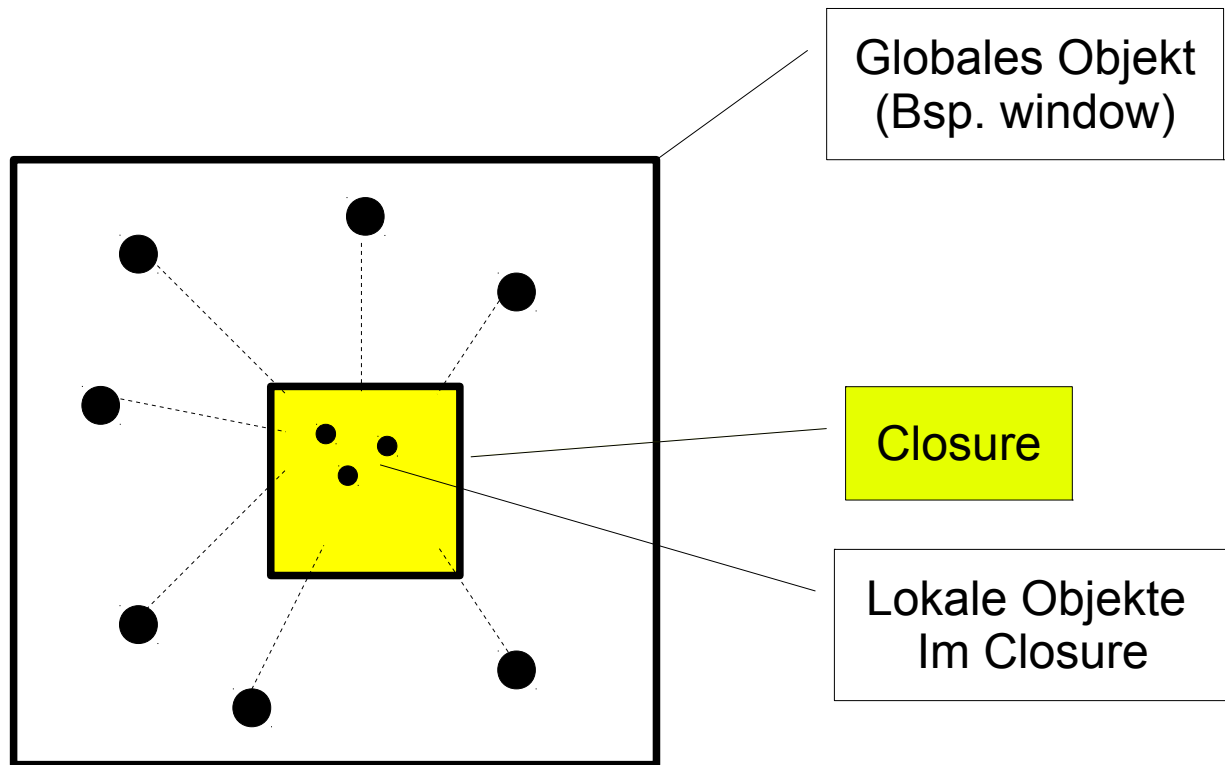




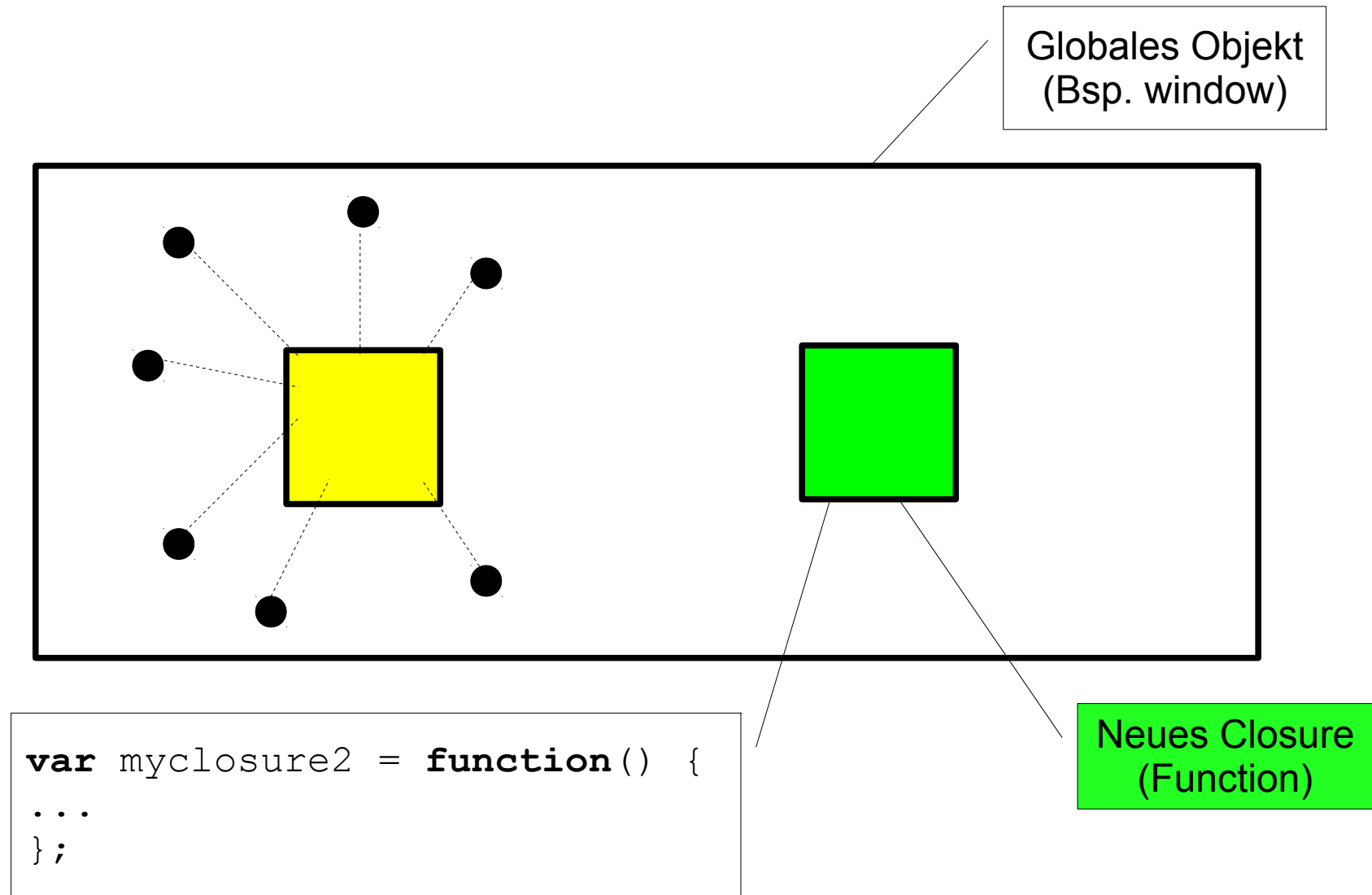
# Closure



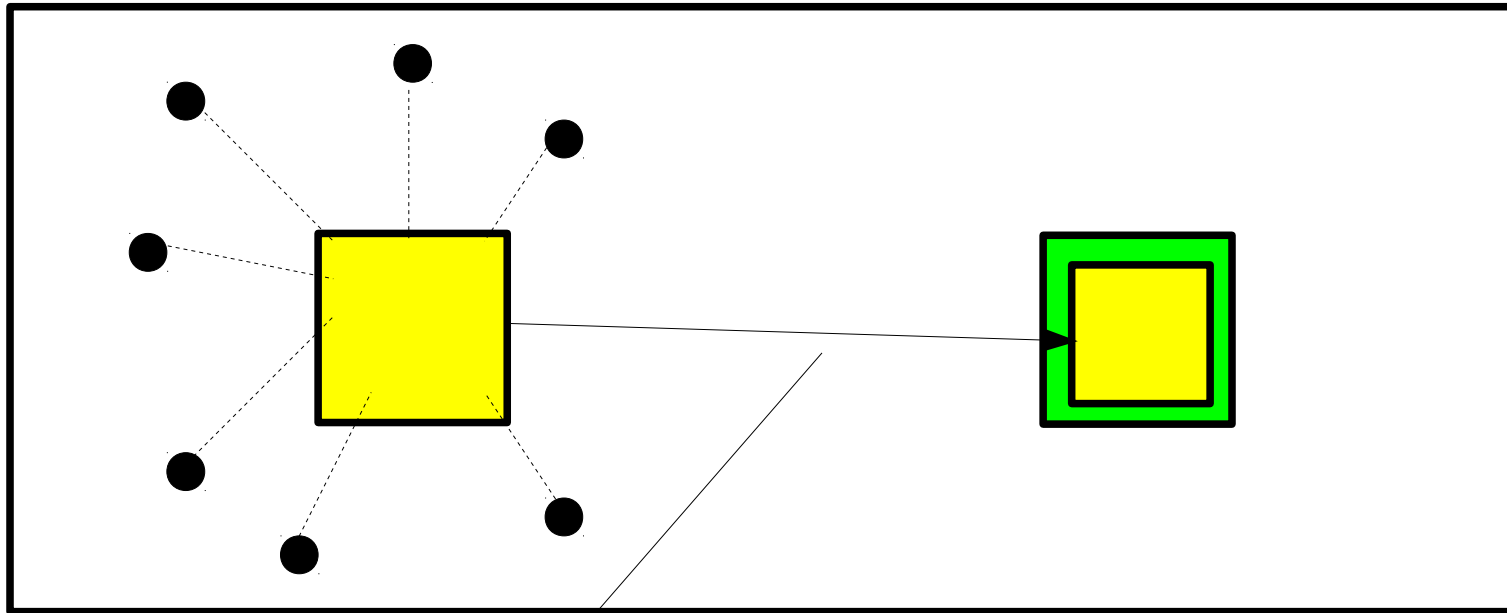
# Closure



# Closure

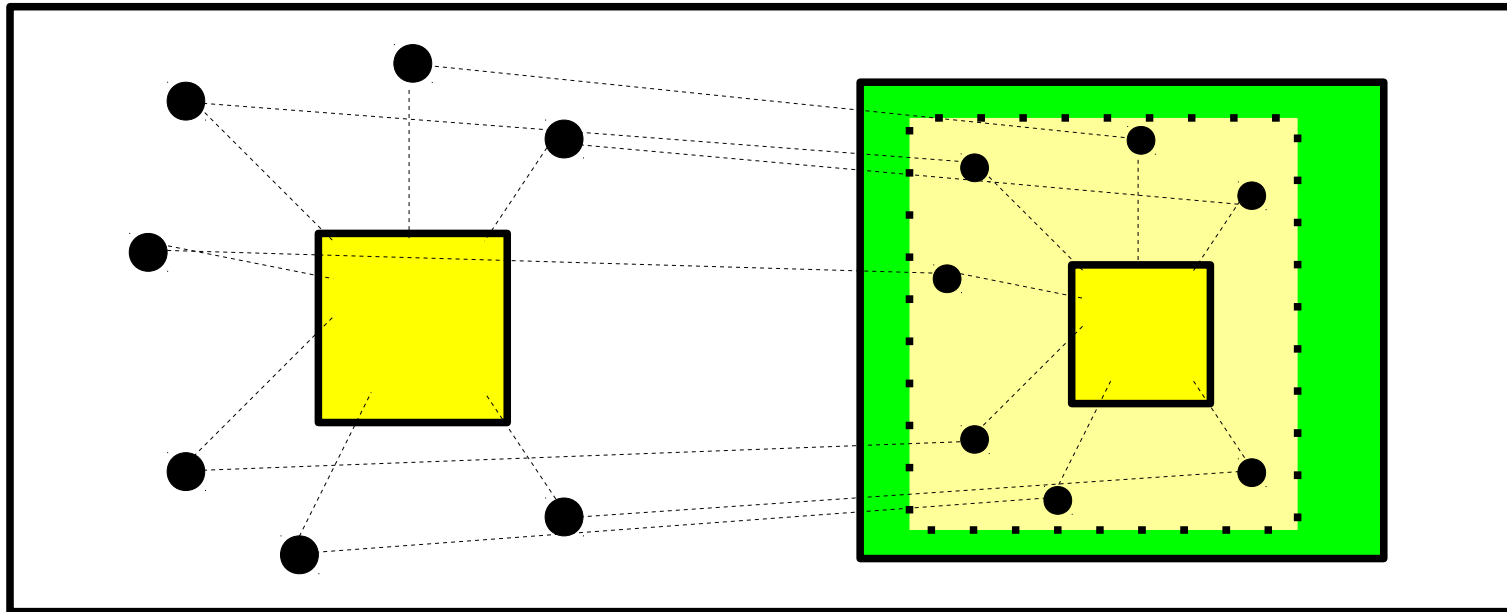


# Closure

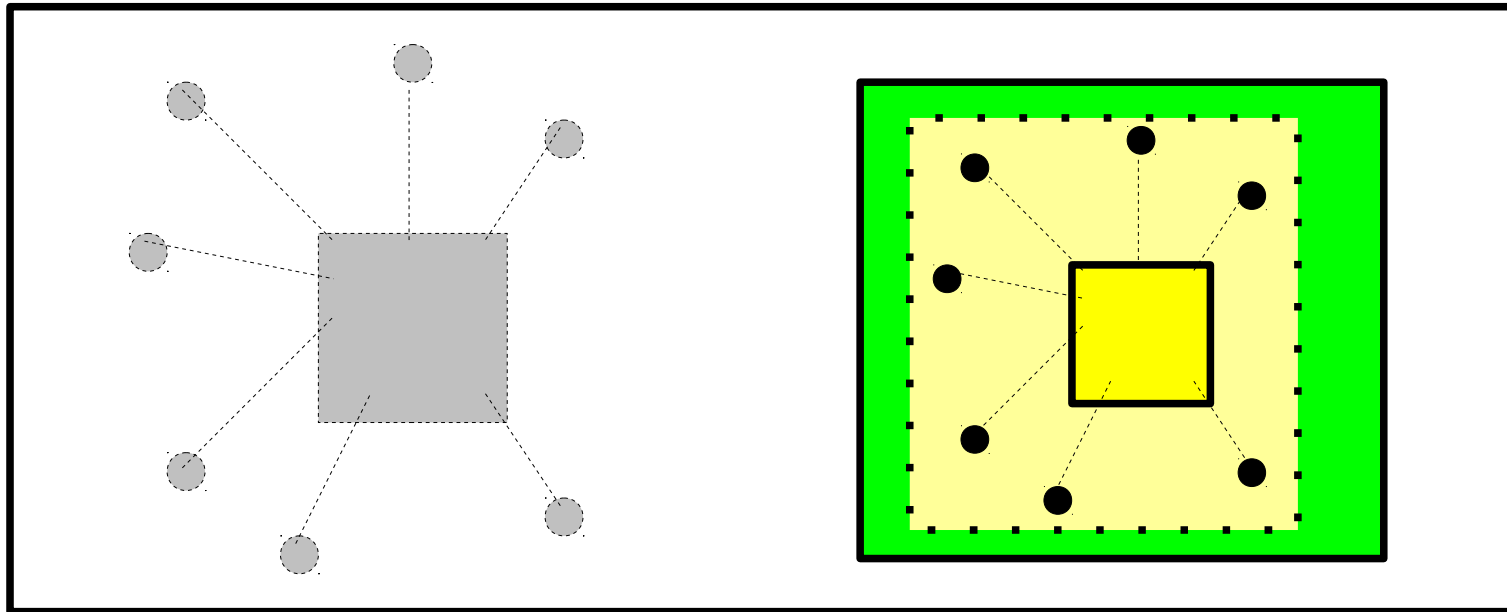


```
var myclosure2 = function(dasAndereClosure) {  
    var outerClosure = dasAndereClosure;  
    // ...  
};
```

# Closure

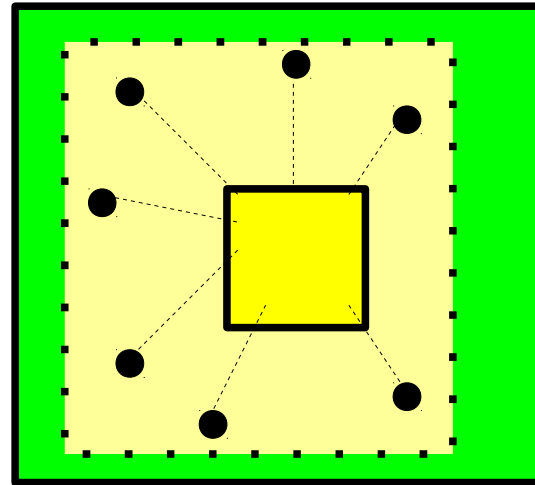


# Closure



„originale“ Objekte hören auf zu existieren,  
weil z.B. der Scope verlassen wurde (`return` z.B.)

# Closure



Closure „konserviert“ seinen ursprünglichen Scope und hat dadurch die Variablen weiterhin zur Verfügung. Das „grüne“ closure sieht die „schwarzen“ Kreise nicht mehr. Nur das „gelbe“ Closure.

# Closure

```
function showclosure() {  
    var inc = makeinc(1);  
  
    console.debug(inc()); // 1  
    console.debug(inc()); // 2  
    console.debug(inc()); // 3  
}  
  
function makeinc(initialValue) {  
    var count = initialValue;  
  
    return function () { // <-- Closure merkt sich count  
        return count++; //      und auch initialValue  
    };  
}  
  
showclosure();
```



# Self invoking Closure

```
// does not pollute global scope
(function() {
    var myFancyApi = {};

    function aHelper() {
        // ...
    }

    myFancyApi.coolFunction = function() {
        // ...
        aHelper();
        // ...
    };

    // expose API to global scope
    window.fancy = myFancyApi;
})(); // () is the Invoke-Operator
```

# Self invoking Closure

```
// does not pollute global scope
(function(global) {
    var myFancyApi = {};

    function aHelper() {
        // ...
    }

    myFancyApi.coolFunction = function() {
        // ...
        aHelper();
        // ...
    };

    // expose API to global scope
    global.fancy = myFancyApi;
})(window); // You only reference it one time in your code ;)
```

# Lambda

- Komplizierte Erklärung die keiner versteht auf Wikipedia ;-)
- Einfach: eine Funktion als solche anderen Funktionen als Parameter übergeben

// klassisch

```
function add(a, b) {  
    return a + b;  
}
```

```
function doFoo(result) {  
    console.log(result);  
}
```

```
doFoo(add(1, 2));
```

// Lambda

```
function add(a, b) {  
    return a + b;  
}
```

```
function doFoo(fn) {  
    console.log(fn(1, 2));  
}
```

```
doFoo(add); // No () invoke op!
```

# Lambda

```
var add = function(a, b) {  
    return a + b;  
};  
  
var sub = function(a, b) {  
    return a - b;  
};  
  
var Operation = function(firstOp, secondOp) {  
    var op1 = firstOp || 0,  
        op2 = secondOp || 0;  
  
    this.calc = function(operation) {  
        return operation(op1, op2);  
    };  
};  
  
var op = new Operation(3, 1);  
console.log(op.calc(add)); // 4  
console.log(op.calc(sub)); // 2
```

# Lambda

```
// $ === jQuery - jquery.com
$('#anDomElement').bind('click', function(event) {
    // anonymes closures als lambda uebergeben
});

var anEventHandler = function(event) {
    // ordinäre Funktion
};

$('#otherDomElement').bind('click', anEventHandler);
```

- Callbacks sind Funktions-Objekte (Lambda) nicht ein String mit Funktionsname (Reflection)
- Das Funktionsobjekt wird mit Invoke-Operator Ausgeführt
- Funktionen können anonym sein

prototypbasiertes Objektmodell

# prototypbasiertes Objektmodell

- Prototype based
  - JavaScript, Lua, Self
  - Beschreibung durch Objekte (Prototypen)
  - Erzeugung durch klonen des Prototypen
  - Factory Method
- Class based
  - Java, C++, PHP
  - Beschreibung durch Schablonen (Klassen)
  - Erzeugung durch „Runtime-Magic“
  - s. [Prototype Pattern](#)

# prototypbasiertes Objektmodell

- Es gibt in JavaScript keine Klassen!

```
// JSON
var simpleObject = {
  publicProperty: 'foo',
  otherProperty: true
};
```

- Reflection

```
typeof simpleObject; // 'object'
typeof simpleObject.toString; // 'function'
typeof simpleObject.publicProperty; // 'string'
typeof simpleObject.otherProperty; // 'boolean'
```

```
simpleObject.hasOwnProperty('publicProperty'); // true
simpleObject.hasOwnProperty('foo'); // false
```



# prototypbasiertes Objektmodell

- Funktionen die mit `new` aufgerufen werden sind sog. Konstruktor-Funktionen.
- In Konstruktor-Funktionen ist `this` das erzeugte Objekt
  - Rückgabewert ist implizit das erzeugte Objekt
  - Kann auch explizit zurückgegeben werden.
  - Wird eine Konstruktorfunktion ohne `new` aufgerufen zeigt `this` auf das globale Objekt (`window`)!
  - Deswegen per Konvention immer Upper Case First!

# prototypbasiertes Objektmodell

```
// implizite Rückgabe
function Animal(name) {
    this.name = name;
}

var anAnimal = new Animal('hans');
```

```
// explizite Rückgabe
function Animal(name) {
    var obj = {
        name: name
    };

    return obj;
}

var anAnimal = new Animal('hans');
```

# prototypbasiertes Objektmodell

```
var Animal = function(name) { // Vater-'Klasse'  
    this.name = name;  
};
```

```
Animal.prototype.getName = function() {  
    return this.name;  
};
```

```
Animal.prototype.says = function() {  
    return this.saying || '';  
};
```

```
var myAnimal = new Animal('Belo the dog');  
console.log(myAnimal.getName()); // Belo the dog
```

```
var Cat = function(name) { // abgeleitete 'Klasse'  
    this.name    = name;  
    this.saying = 'miau';  
};
```

```
Cat.prototype = new Animal(); // erbt von Animal  
var myCat = new Cat('Mauzi the cat');  
myCat.says(); // 'miau'
```

# prototypbasiertes Objektmodell

```
// short hand function for inheritance
Function.prototype.inherit = function(Parent) {
    this.prototype = new Parent();
    return this;
};

Cat.inherit(Animal);
```

## Oder

```
var anAnimal = { // The prototype object
    name: '',
    getName: function() { return this.name; },
    says: function() { return this.saying || ''; }
};

var aDog = Object.create(anAnimal); // create from prototype
aDog.name = 'Belo';

var aCat = Object.create(anAnimal);
aCat.name = 'Mauzi';
aCat.saying = 'miau';
```

# Public, private & protected?

```
var Animal = function(n) { // <-- ein Closure ;)
  var name = n; // 'private'
  this.id = 0; // 'public'

  function aPrivatefunc() { // <-- auch ein Closure
    // ...
  }

  // auch eins
  this.getName = function() { return name; } // 'privilegiert'
};

Animal.protoype.says = function() { // 'public' (noch eins)
  return this.saying || '';
};
```

- Sichtbarkeit auf Scope-Basis (Closure)
- public Methoden sehen nur `this`

# Ducktyping

- "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
- „... object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface.“  
([Wikipedia](#))

```
if (typeof foo.getName === 'function' && typeof foo.says === 'function') {  
  // it's like an animal object...  
  if (typeof foo.saying === 'string') {  
    // it's like a cat object...  
  }  
}
```

# Method stealing

```
function foo() {  
    console.log(typeof arguments); // 'object'  
    console.log(typeof arguments.length); // 'number'  
}
```

```
foo();  
foo(1);  
foo(1, 2);  
foo(1, 2, 3);
```

- Jede function hat `arguments`
- Fühlt sich an wie ein Array, ist aber keins!

```
function foo() {  
    var args = Array.prototype.slice.call(arguments, 1);  
    console.debug(args);  
}
```

```
foo(1, 2, 3, 4); // [2, 3, 4]
```

Zu guter letzt



# Fallen - Numbers

```
var a = 0.1, b = 0.2, c = 0.3;

console.debug((a + b) + c === a + (b + c)); // false

// ABER

var a = 0.1 * 100, b = 0.2 * 100, c = 0.3 * 100;

console.debug((a + b) + c === a + (b + c)); // true
```

- Number ist immer ein 64 Bit Float
- Es gibt keine reinen Integers

# Fallen - Variable Shadowing

```
var a = 'foobar';

console.debug(a); // 'foobar'

function() {
    var a = 1.0; // shadows outer variable
    console.debug(a); // 1.0
}

console.debug(a); // 'foobar'
```

- Kann zu sehr dubiosen Bugs führen
- JSLint/JSHint hilft!

# Fallen

- Konstruktor-Funktion ohne **new**
  - **this** zeigt auf das globale Objekt (z.B. **window**)
  - No Errors, no Warnings → Konvention: Beginnen immer mit Großbuchstaben
- This und der Kontext
  - Funktion → global object oder undefined
  - Methode → Objekt (Eigener d. Methode)
  - Konstruktor → Das „neue“ Objekt
  - apply/call → Funktionsargument

# Fallen

- Null ist ein Objekt! `typeof null; -> 'object'`
- `null !== undefined`
- `true/false` (ähnlich PHP)
  - `true`: `'false', '0'`
  - `false`: `false, null, undefined, '', 0, NaN`
- Memory Leaks in IE
  - Zyklische Referenzen von Closures
  - Wenn DOM-Objekt ein JavaScript Objekt referenziert (bspw. Eventhandler) und vv.
  - s. [Crockford](#)

Kontakt:

Sven Strittmatter <ich@weltraumschaf.de>

Q & A

