assono

BLUG 2010
Advanced JavaScript

Zaventem (Brussels), March 30th, 2010

Innovative Software Solutions.

www.assono.de

Thomas Bahn

- Graduated in mathematics, University of Hannover
- Developing in Java and RDBMS since 1997
- Dealing with Notes/Domino since 1999: development, administration, consulting and trainings

- Frequent speaker at technical conferences about IBM Lotus Notes/Domino and author for THE VIEW

tbahn@assono.de
www.assono.de/blog
+49/4307/900-401

assono
IT-Consulting & Solutions

Tools

- Mozilla **Firefox** with
  - **Firebug**: http://getfirebug.com/
  - Venkman (JavaScript debugger): http://www.hacksrus.com/~ginda/venkman/
- Other browsers for testing
  - Firebug Lite: http://getfirebug.com/firebuglite
  - Fiddler: http://www.fiddler2.com/fiddler2/

Declaration of Variables

- Simple variable declaration:
  `var name;`
- Variable declaration with value assignment:
  `var name = value;`
- Specialty of **SSJS** (Server Side JavaScript): **typed** variable declaration:
  `var name : type = value;`

Mind the Trap: Test, if Variable is Declared

- Seen often, but simplified:

```
if (a) {
    …
}
```

- **Beware**: the code block will also be executed, if a is equal to `false`, `null`, "" or 0
- **Better way**:

```
if (typeof a !== "undefined") {
    …
}
```

Types of Variables

- “loose typing” doesn't mean untyped!
- There are **exactly 6 types** in JavaScript
  - String
  - Number
  - Boolean
  - Object
  - `null`
  - `undefined`

Special Strings

- \ escape character
- \\ one backslash
- \' simple quote
- \" double quote
- \n new line
- \r carriage return
- \t tabulator
- \u Unicode character, e.g. \u042F ⇒ Я

String Object

- Useful methods of the String object
  - toUpperCase(), toLowerCase(): like LotusScript
  - charAt(position): character at given position
  - indexOf(searchString): position of searchString
  - indexOf(searchString, startPos): Position of searchString after startPos
  - lastIndexOf(suchstring): ditto, but backwards
  - substring(start, end): part of string
  - slice(start, end): like substring, but when end is negative, length is added (counts from end)
  - split(separator): splits string into an array

Numbers and the Number Object

- All numbers are 64-bit floating point (“double”).
- **Rounding differences** through binary-decimal-conversion
- Bit-shifting (<<, >>) possible, but **inefficient**
- `Number.MAX_VALUE ⇒ 1.7976931348623157e+308`
  `Number.MIN_VALUE ⇒ 5e-324`
- `Number.toString(basis)`, e. g.

```
var a = 100;
a.toString(16) ⇒ "64"
a.toString(2) ⇒ "1100100"
```

Numbers (cont.)

- `0x` prefix for hexadecimal numbers, e. g. `0xFF`
  `0` prefix for octal numbers, e. g. `014` (=12)
- **Beware** using `parseInt()`, e. g.
  - `parseInt("08")` ⇒ 0,
    parsed until first invalid character and
  - `parseInt(08)` ⇒ Error:
    `08 is not a legal ECMA-262 octal constant`
- Particularly with regard to user input, e. g. month
- **Better**:
  always call with radix as second parameter:
  `parseInt("08", 10)`

Math Object

- Useful constants and methods of the Math object:
  - Math.PI ⇒ 3.141592653589793
  - Math.E ⇒ 2.718281828459045
  - Math.SQRT2 ⇒ 1.4142135623730951
  - Math.LN2 ⇒ 0.6931471805599453
  - Math.LN10 ⇒ 2.302585092994046
  - Math.random() ⇒ 0 <= random number < 1
  - Math.round(), Math.floor(), Math.ceil()
  - Math.sin(), Math.cos(), Math.atan()
  - Math.min(), Math.max()
  - Math.pow(), Math.sqrt()

NaN = Not a Number

- “contagious” in calculations
- Not equal to anything (including NaN)
- Test with `isNaN()`
- `typeof NaN` ⇒ `"number"` 

Infinity

- e. g. 1/0 ⇒ Infinity
- Greater than the largest valid number:
  Infinity > Number.MAX_VALUE
- “contagious” in calculations, but
  - 1/Infinity ⇒ 0
  - Infinity/Infinity ⇒ NaN

Boolean

- `!` Logical Not
- `&&` Logical And
- `||` Logical Or
- **Lazy evaluation:** calculate only, if (still) necessary
- Example:

```
var a = 1;
true || (a = 2);
```

a ⇒ a is still equal 1

“Falsy” Values

- Following values are “**falsy**”:
  - `false`
  - `null` (empty, set intentionally)
  - `undefined` (uninitialized or non-existant)
  - `""` (empty string)
  - `0`
  - `NaN`
- **All** other values are considered `true`, even
  - `"0"`
  - `"false"`

Comparison Operators

- `==` Equality **with** type conversion (if necessary)
  `!=` Inequality **with** type conversion (if necessary)
- `===` Equality **without** type conversion
  `!==` Inequality **without** type conversion
- Examples
  - `"1" == 1 ⇒ true`
  - `"1" === 1 ⇒ false`

Comparison Operators (cont.)

- `switch`
  - Comparison **with** type conversion
- `a ? b : c`
  - Ternary operator
  - Returns `b`, if `a` is `true`, else `c`
  - Example:

```
(typeof a !== "undefined") ? a : "Default"
```

Default Operator ||

- `||` is logical Or
- Example
  ```
  var value = arg || "Default value";
  ```
- Assigns `arg` to `value`, if `arg` is `true` (not falsy), else `"Default value"`
- **Useful for optional parameters** of a function
- **Beware**, if `false`, `null`, `""`, `0` could be valid values!

Guard Operator &&

- && is logical And
- Example:
  `return obj && obj.member;`
- Returns `obj.member`, if `obj` is `true` (not falsy), else `obj`
- **Circumvents possible error** when accessing `undefined.member`
- **Beware**, if `false`, `null`, `""`, `0` could be valid values!

More Operator Magic

- `+` as unary operator converts strings into numbers:
  `+"1" === 1`
- `!!` converts anything into boolean:
  `!!"1" === true`

Arrays

- Arrays are objects (inherit from `Object`)
- Their `length` attribute is always the greatest index plus 1
- `length` can also be set.
  - setting a **larger number enlarges** the array by appending `undefined` values
  - setting it to a **smaller number shortens** the array by cutting surplus elements
- Array literals: `["one", "two", "three"]`
- You can use strings like character arrays:
  `var text = "abcd"; text[2]` ⇒ `"c"`

Arrays (cont.)

- Useful methods of the `Array` object:
  - `push` and `pop`, to use an array as a stack
  - `sort`: sorts array
  - `join(separator)`: joins the elements to a string
  - `slice(start, end)`: returns part of array, doesn't change the source array
  - `splice(start, end, additional parameters)`: returns part of array, replaces it by the additional parameters or removes it without leaving a gap

Mind the Trap: delete

- **`delete arr[2]`**
  Deletes the 3rd element, or to be more precise: **sets** the 3rd element **to `undefined`.**

```
var arr = ["one", "two", "three", "four"];
delete arr[2];
arr ⇒ ["one", "two", undefined, "four"]
```

- **`arr.splice(2, 1)`**
  Removes the 3rd element and the **remaining elements move up**

```
var arr = ["one", "two", "three", "four"];
arr.splice(2, 1);
arr ⇒ ["one", "two", "four"]
```

typeof Operator

- Results of **typeof** are sometimes not too useful:

Object of type	Result of typeof
object	"object"
function	"function"
array	"object"
number	"number"
string	"string"
boolean	"boolean"
null	"object"
undefined	"undefined"

- Use **instanceof** on objects (details follow)

for Loops

- ```
  for (init; comparison; increment) {
      ...
  }
  ```
- All three elements can contain **multiple statements,** separated by commas (!), e.g.

```
for (
    var i = 0, result = "";
    i < 100;
    i++, result += i + "\n"
) {...}
```
```

Mind the Trap: with

- ```
  with (obj) {
      member = "value";
  }
  ```
- If obj contains an attribut member, its value will be set to "value" …
- … else a **new global variable** member is created and "value" is assigned
  ```

Mind the Trap: Semicolon Insertion

- When an error occurs, the JavaScript interpreter **inserts an semicolon** at the end of a line and **retries** the execution.
- Example:

```
return // semicolon will be added here
{  // is interpreted as code block
   state: "ok" // state is a label and
               // a semicolon gets inserted
}; // block will never be executed
```

- This behavior can mask errors.

Error Handling with try – catch – finally

- Just try it... and react on errors
- Example:

```
try {
    // an error could occur in here
} catch (e) { // e is an Error object
    // do something about this error
    alert(e.name + ": " + e.message);
} finally {
    // will be executed regardless of an
    // error, e. g. to free up ressources
}
```

- Only **one** catch-block (not many as in Java)

Error Handling (cont.)

- `throw` creates (fires) a new error
- Examples:

```
throw new Error("Message");
throw {
    name: "Name",
    message: "Message"
}
```

- `window.onerror = myErrorHandler`
  Sets error handling function in the browser, which will be called on every error afterwards

References

- Variables contain references
- Parameters are passed to functions as references, not (copies of) values

Functions – Data – Objects

- **Functions are objects!**
  - They can have attributes and methods (i.e. inner functions).
- **Functions are data!**
  - They can be stored in variables.
  - They can be passed to and returned by functions.
- This can't be done in LotusScript, nor Java.

Definition of Functions

- `function f (args) { ... };`
  is an **abbreviation** for
  `var f = function (arg) { ... };`
- **Anonymous** functions
  `function (args) { ... };`
- **Inner** functions

```
function f(args) {
    function g(args) { ... };
    var h = function (args) { ... };
    this.i = function (args) { ... };
}
```

- `new Function(args, body)`
  `f = new Function("a", "b", "return a + b");`

Source Code of a Function

- **Useful: `f.toString()`** returns the definition of function `f`, i.e. **its source code**
- Doesn't work with predefined functions though.
- Examples:

```
function f() {
    alert(new Date());
};
f.toString()
⇒ "function f() { alert(new Date); }"

Object.toString.toString()
⇒ "function toString() { [native code] }"
```

Function Parameters

- Declaration of a function contains the list of the **expected parameters**
- Parameters are local variables of the function

"Unexpected" Function Parameters with arguments

- Access to **all** parameters with arguments, an **array-like construct**
- Example:

```
function sum() {
    var i, result = 0;
    for (i = 0; i < arguments.length; i++) {
        result += arguments[i];
    }
    return result;
}
sum(1, 2, 3, 4, 5) ⇒ 15
```

arguments (cont.)

- arguments has the attribute length, but not the other methods of the Array object.
- **Work around**: Function.prototype.apply()

```
function removeFirstArgument() {
    var slice = Array.prototype.slice;
    return slice.apply(arguments, [1]);
} ;
removeFirstArgument(1, 2, 3, 4)
⇒ [2, 3, 4]
```

- arguments.callee contains a reference to the called function, which can become handy in anonymous functions

Value Object

- **Tip**: Use an object as (only) parameter, which contains all needed information.
- **Benefits**:
  - You can always add new “inner parameters” without changing existing code calling this function.
  - You have “named parameters”.
  - Defaults should be used for all parameters not in the value object.

Value Object (cont.)

- Example:

```
function showHide(argsList) {
    var node = argsList.node || document;
    var state = argsList.state || "show";
    var term = argsList.term || "true";
    ...
};
showHide({
    node: document.getElementById("blubb"),
    state: "hide";
})
```

Function Results

- A function without `return` returns `undefined`.
- A function with `return x` returns x. 
- A constructor (details follow) without `return` returns a reference to the new object.
- A constructor with `return x` returns x, **if x is an object**, else they return the new object.

Function Calls and `this`

- Functions always run in a **context**.
- **`this`** points to the current context.
- An inner function cannot access its outer function's current context.
- **Convention**: `var that = this;`
- There are (at least) 4 ways to call functions:

Function Calls (cont.)

- **1st Functional form** `f(args)`:
  `f` will be executed in the global context.
- **2nd Method form** `obj.f(args)` and `obj["f"](args)`:
  `f` will be executed with `obj` as context.
- **3rd Constructor form** `new f(args)`:
  `f` runs in the context of the newly created, empty object, which will be returned by the function.
- **4th** `f.apply(obj, args)` and `f.call(obj, arg1, arg2, ...)`:
  `f` runs in the context of `obj`.
  `f` doesn't need to be method of `obj`!

Function Calls (cont.)

- **Beware**:
  If you want to call a method of an object, but forget to specify the object, no error will be thrown and the method will be executed in the global context.

Anonymous Functions

- Used as parameters for other functions
  - when they are used only for this purpose
  - in the called functions they have a name: the name of the parameter
- Example:
  call-back functions for Ajax calls, for `setTimeout()` or `setInterval()`:

```
setTimeout(
    function() {
        alert(new Date());
    },
    1000
)
```

Anonymous Functions (cont.)

- For immediate, one-time execution, e.g. for initializing
- Example on the following page

Anonymous Functions (cont.)

- You can **hide** code with anonymous functions.
- Example:

```
(function () {
    var message = "Bye Bye!";
    window.onunload = function () {
        alert(message);
    };
})(); // define function and execute it
```

- Keeps the global object clean
- Variable `message` is accessible by the anonymous function, but is invisible and inaccessible for other code (Closure!)

Self-Modifying Functions

- Example:

```
f = function() {
    alert("First time");
    return function() {
        alert("Next time");
    };
}();
f();
f.toString()
⇒ "function() { alert("Next time"); }"
```

- 2 Alerts: "First time" and "Next time"
- **Typical usage**: one-time initialization and creation of an appropriate function for a special environment, e.g. the current browser

Currying

- Create a new function with at least one parameter less than an existing one.
- Example:

```
function addGenerator(a) {
    return function(b) {
        return a + b;
    };
};
addOne = addGenerator(1);
addOne(47) ⇒ 48
```

Function Object

- Useful attributes and methods of the `Function` object:
  - `length`: number of expected parameters
  - `caller`: calling function (not standardized)
  - `call(obj, param1, param2, ...)`:
    executes function in the context of `obj`
  - `apply(obj, paramsArray)`:
    like `call`, but all function parameters in one array

"eval is evil"

- Performance is bad.
- Security: use only on **very trustworthy** argument
- Same considerations apply to `new Function(args, body)`

`alert()`

- Auxiliary „debugger“
- Blocks execution of JavaScript
- **Beware** when using `alert()` in combination with asynchronous Ajax calls

Scopes

- There are only two kinds of scopes in JavaScript:
  - **global** scope
  - **function** scope
- In particular, there is no block scope like in Java
- Accessing an undeclared variable doesn't throw an error, but creates a **new, global** variable (think about typos).
- Example:

```
function () {
    a = 1; // without var: a is global!
}
```

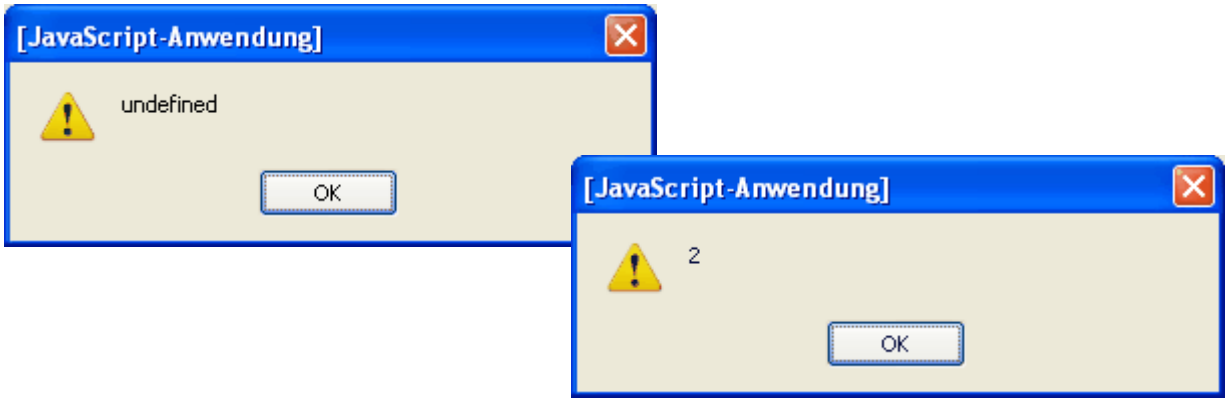
Scopes (cont.)

- Which output do you expect here?

```
var a = 1;
function f() {
    alert(a);
    var a = 2;
    alert(a);
}
f()
```

Scopes (cont.)

- Results in 2 alerts



- 2 phases:
  - local variables are found and memory reserved
  - code is executed

Scopes (cont.)

- JavaScript has a **lexical** scope.
- The context of a function is created, when the function is **defined**, not when it is executed.

Context

- Each object has it **own context**.
- Plus there is a **global context**.
- Functions have access to the attributes and method of the context, in which they have been defined …
- … and all the way up to the global context.

Context (cont.)

- The global object in browsers is the `window` object!
- Example:
  `a = 1;`
  `window.a` ⇒ `1`
- Global functions like `parseInt` are methods of the global object:
  `parseInt === window.parseInt` ⇒ `true`

Context (cont.)

- **Lexical context**, i.e. “as written in code”, not as executed
- Example:

```
function outer() {
    var o = 1;
    function inner() {
        alert(o);
    }
}
```

- At the time `inner` is defined, `o` is already known.

Context (cont.)

- Another example:

```
function outer() {
    var o = 1;
    return (function() {
        alert(o);
    })
};
var f = outer();
f() ⇒ 1
delete outer;
f() ⇒ 1 (!)
```

- o is still known, after `outer` has been executed – even **after it has been deleted**, `f` has still access to o!

Closures

- Even after an outer function has been completed, inner functions can access their attributes and functions.
- This feature is called **Closure**.
- **Most unusual concept** of JavaScript for LotusScript and Java developers.
- But Closures are probably the single **most important feature** of the JavaScript language!

Mind the Trap: Reference, not Copy of Value

- f has a reference to o, not a copy of its value at definition time.
- When o is modified after f has been defined, this effects f.
- Decoupling with mediator function: a function, which is defined and immediately executed.

Mind the Trap: Reference, not Copy of Value (cont.)

- Example with an unexpected result:

```
function f() {
    var a = [];
    for (var i = 0; i < 5; i++) {
        a[i] = function() {
            return i;
        }
    }
    return a;
};
var a = f();
a[0]() ⇒ 5 (should be 0)
```

Mind the Trap: Reference, not Copy of Value (cont.)

- Improved example:

```
function f() {
    var a = [];
    for (var i = 0; i < 5; i++) {
        a[i] = (function(x) {
            return function() {
                return x;
            }
        })(i);
    }
    return a;
};
var a = f();
a[0]() ⇒ 0
```

Objects

- Objects are sets of name-value pairs.
- Names are Strings.
- Values can be of any type.
- Correlate to „associative arrays“ or „hash maps“ in other programming languages

Creation of Objects

- Object literals like {}
  ```
  var object = {};
  ```
- new
  ```
  var object = new Constructor();
  ```
- Object.create(...)
  actually, you probably never need this

Object Literals and Access to Members

- ```
  var obj = {
      firstName: "Thomas",
      "city of birth": "Hannover",
      "123": 123,
      "@$&%": 1,
      doIt: function() {
          alert(this.firstName);
      }
  };
  obj.doIt()
  ```
- `obj.firstName` is `"Thomas"`, `obj["city of birth"]` is `"Hannover"` etc.
- The second kind is **necessary** for reserved words and invalid identifiers.
  ```

Modifying Objects

- ```
  var obj = {
      firstName: "Thomas",
      doIt: function() {
          alert(this.firstName);
      }
  };
  obj.firstName = "Lydia";
  obj.doIt = function() {
      alert("First Name: " + this.firstName);
  };
  obj.doIt()
  ```
- Attributes and **methods** can be changed at run-time.
- This isn't possible in class-based programming languages like LotusScript.
  ```

Constructors

- Functions can be used as constructors (**with new**):

```
function Factory(location) {
    this.location = location;
};
var f = new Factory("Kiel")
```

- A new, empty object is created and `Factory` is executed in its context.
- Default return value is this newly constructed object.
- When a constructor is completed with `return x`, `x` will be returned, (only) if `x` is an object, else the new object is returned.

Constructors (cont.)

- Objects have an `constructor` attribute, which points to the constructor of the object **at its creation time**.
- Example creates new, similar object:

```
function Factory(totalEmployees) {
    this.totalEmployees = totalEmployees;
};
var f1 = new Factory(100);
var f2 = new f1.constructor(200);
f2 ⇒ Object { totalEmployees = 200 }
```

Constructors (cont.)

- **Convention**: Constructor names should start with an **capital letter**.
- **Beware**:
  If you forget new, the function will still be executed, but in the **global** context instead in the context of the newly created object.

instanceof Operator

- instanceof operator compares with constructor:
  f1 instanceof Factory ⇒ true
  f1 instanceof Object ⇒ true
  f1 instanceof String ⇒ false
- instanceof is true for “super-objects”, too.

for ... in Loops

- Iterate over all attributes and methods of an object – and its "ancestors" (but not all properties are "enumerable").
- Example:

```
for (prop in obj) {
   alert(prop.toString());
}
```

- `obj.hasOwnProperty("name")` is `true`, only if `obj` has an attribute `name`.

Enhancing Existing Objects

- You can **enhance existing objects** after they have been created …
- … even objects of the JavaScript language, like `Function`, `Object` and `String`!
- Example:

```
String.prototype.trim = function() {
    return this.replace(/^\s+|\s+$/g, "");
}
"'" + "   test   ".trim() + "'" ⇒ "'test'"
```

Enhancing Existing Objects (cont.)

- **Beware**: What if the next version of JavaScript has your addition built-in?
- Or other developers have the same idea?
- At least, you should **check before** you enhance language objects.
- Example:

```
if (!String.prototype.trim) {
    String.prototype.trim = function() {
        return this.replace(/^\s+|\s+$/g, "");
    }
}
```

Public Attributes and Methods

- Principally, all properties, i.e. attributes and methods of an object are **publically visible and usable**.
- Example:

```
function Factory() {
    this.totalGoods = 0;
    this.produce = function() {
        this.totalGoods++;
    };
};
var f = new Factory();
f.totalGoods = 1000;
f.produce();
f.totalGoods ⇒ 1001
```

Private Attributes and Methods

- **Local variables and parameters** of the constructor get **private attributes and methods** of all objects created with function.
- Only inner functions of the constructor can access its local variables and functions aka private properties.
- Example:

```
function Factory(totalEmployees) {
    var totalGoods = 0;
    var produce = function() {
        totalGoods++;
    };
};
var f = new Factory();
f.produce() ⇒ Error
```

Privileged Methods

- Privileged methods are **publically callable** and **can access private properties**.
- Definition in the constructor:
  `this.privilegedFunction = function() {...}`
- They **cannot** be added to the constructor **later**!
- Closures again!

Privileged Methods (cont.)

- Example:

```
function Factory() {
    var totalGoods = 0;
    this.produce = function(count) {
        totalGoods += count;
    };
    this.getTotalGoods = function() {
        return totalGoods;
    }
};
var f = new Factory();
f.produce(5);
f.getTotalGoods() ⇒ 5
```

”Class Attributes”

- In Java, there is a **`static`** modifier for class attributes and methods.
- In JavaScript, there are no classes, but …
- … you can add attributes to the constructor function itself (the Function object), which are usable like class attributes.
- Useful for logging, book keeping (counting created objects), object pools, configuration of an “object factory” etc.

"Class Attributes" (cont.)

- Example:

```
var Factory = function() {
    Factory.totalFactories++;
    var totalGoods = 0;
    this.produce = function(count) {
        totalGoods += count;
    };
    this.getTotalGoods = function() {
        return totalGoods;
    }
};
Factory.totalFactories = 0;
var f1 = new Factory();
var f2 = new Factory();
Factory.totalFactories ⇒ 2
```

Inheritance à la JavaScript

- Each object has a **prototype** attribute, normally `{}`.
- Functions are objects, constructors are functions, therefore constructors have this attribute, too.
- You can **add new properties** to a prototype object or even **replace** it completely.

Inheritance à la JavaScript (cont.)

- Search sequence for properties:
  - current object a
  - `a.constructor.prototype`
  - `a.constructor.prototype.constructor.prototype`
  - etc.
  - at last: `Object`
- **Prototype chain**
- This can be used as "**inheritance**".
- All changes to an object's `prototype` take effect immediately – to the object itself and it "successors" in the prototype chain.

Inheritance à la JavaScript (cont.)

- Properties of an object superimpose properties with the same name up in the chain – they **override "inherited" properties**.
- Methods in a constructor are **copied** into each created object and use up some memory.
- You can add them to the object's `prototype` instead.
- This consumes less memory, but costs some performance (more lookups up the chain).

Prototypical Inheritance

- Example:

```
var Factory = function() {
    this.totalGoods = 0;
};
var f = new Factory();
Factory.prototype.produce =
    function(count) {
        this.totalGoods += count;
    };
f.produce(10);
f.totalGoods ⇒ 10
```

Prototypical Inheritance (cont.)

- This works even **after the object's creation!**
- In the example: Object `f` is created, then `produce` is defined and added to the `Factory`'s `prototype`.
- The function `produce` is not found directly in `f`, but in its prototype chain.
- Instead of
  `Factory.prototype.produce = ...`
  you can write
  `f.constructor.prototype.produce = ...`

Prototypical Inheritance (cont.)

- Another example:

```
var Person = function (name) {
    this.name = name;
};
Person.prototype.getName = function () {
    return this.name;
};
var User = function(name, password) {
    this.name = name;
    this.password = password;
};
User.prototype = new Person();
var me = new User("Thomas", "secret");
alert("User " + me.getName() + " created")
```

Mind the Trap: Replacing the prototype Object

- If you **replace the prototype** object instead of enhancing the existing one, this only takes effect on **objects created afterwards**.
- Objects has an **internal pointer** to the prototype object at the time, they were created. This pointer isn't changed, when you overwrite the prototype object.
- In Firefox, this internal attribute is called __proto__.
- Besides, after replacing the prototype object the constructor attribute sometimes points to the wrong object. Therefore you should also **set the constructor** after replacing the prototype object.

Mind the Trap: Replacing the prototype Object (cont.)

- Example:

```
var Person = function (name) {
    this.name = name;
};
Person.prototype.getName = function () {
    return this.name;
};
var User = function(name, password) {
    this.name = name;
    this.password = password;
};
User.prototype = new Person();
User.prototype.constructor = User;
```

Access the „Super Class“ with uber

- There is **no direct access** to the “super class”
- **Convention**: set the **uber attribute** to the prototype of the “super class”
- Example:

```
User.prototype = new Person();
User.prototype.constructor = User;
User.uber = Person.prototype;
```

Simplification

- Simplify this by introducing a helper function:

```
function extend(child, parent) {
    var F = function() {}; // empty function
    F.prototype = parent.prototype;
    Child.prototype = new F();
    Child.prototype.construtor = child;
    Child.uber = parent.prototype;
}
```

- Usage:

```
extend(User, Person);
```

Classical Inheritance in JavaScript

- There are different approaches to “simulate” classical, i.e. class-based inheritance in JavaScript
  - Douglas Crockford
  - Markus Nix
  - Prototype (JavaScript library)
  - many, many more

Classical Inheritance: Crockford's Way

- 3 enhancements of Function
- Method method to simplify additions to Function

```
Function.prototype.method = function (name, func)
{
  this.prototype[name] = func;
  return this; // useful for call chaining
};
```

- Method inherits for inheritance

```
Function.method("inherits", function(parent) {
  this.prototype = new parent();
  return this;
}); // minimalistic version without uber();
```

Classical Inheritance: Crockford's Way (cont.)

- 3 enhancements of `Function` (cont.)
- Method `swiss` to **copy properties** from a parent object:

```
Function.method("swiss", function (parent) {
    for (var i = 1; i < arguments.length; i += 1) {
        var name = arguments[i];
        this.prototype[name] = parent.prototype[name];
    };
    return this;
});
```

- With `swiss` you can simulate multiple-inheritance or interfaces.

Classical Inheritance: Crockford's Way (cont.)

- Example of usage:

```
var Person = function(name){this.name = name};
Person.method("getName", function() {
    return this.name;
});
var User = function(name, password) {
    this.name = name;
    this.password = password;
};
User.inherits(Person);
User.method("getPassword", function() {
    return this.password;
});
var b = new User("Thomas", "secret");
b.getName() ⇒ "Thomas"
```

Parasitic Inheritance

- In a constructor **call another constructor** and return the object, it returns, after enhancing it.
- Example:

```
function ExistingConstructor(a) {...};
function NewConstructor(a, b) {
    var that = new ExistingConstructor(a);
    that.anotherAttribute = 0;
    that.anotherMethod = function (b) {...};
    return that; // instead of this (implicit)
}
var obj = new NewConstructor();
obj
```

Parasitic Inheritance (cont.)

- **Drawback**: Changes and enhancements of `NewConstructor.prototype` **won't be inherited**.
- Example:

```
var obj = new NewConstructor();
NewConstructor.prototype.a = 1;
typeof obj.a ⇒ "undefined"
```

Global Variables

Global Variables are evil  and you should avoid using them, wherever possible.

Imagine the problems, if some JavaScript libraries use the same global variables – you'll get unpredictable (undeterministic) results...

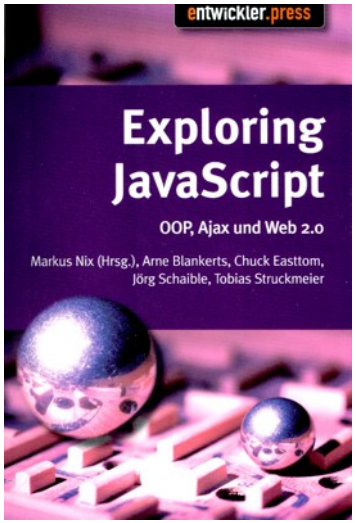
Namespaces

- Namespaces help to keep the global context clean.
- Example:

```
var de = {};
de.assono = {};
de.assono.HtmlHelper = {
    appendDiv: function(child, parent) {...};
    ...
}
```

Books

- Markus Nix (Hrsg.), et. al.
  "Exploring JavaScript – OOP, Ajax und Web 2.0"
  entwickler.press, ISBN 978-3-939084-28-0, 165 S.



Markus' Web site: http://www.markusnix.com/

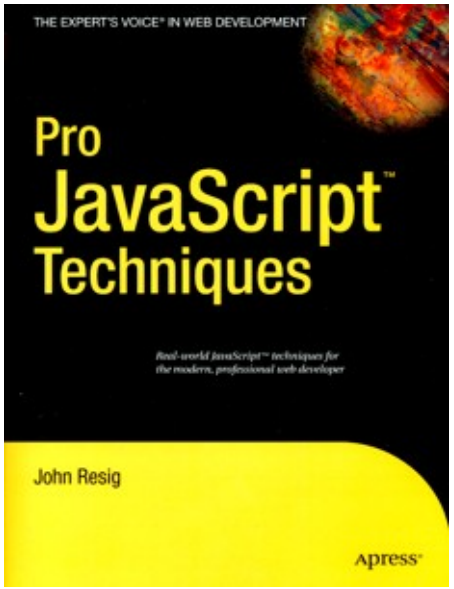
Books (cont.)

- Douglas Crockford: „JavaScript: The Good Parts“ O'Reilly, ISBN 978-0-596-51774-8, 155 S.

his Web site: http://www.crockford.com/

Books (cont.)

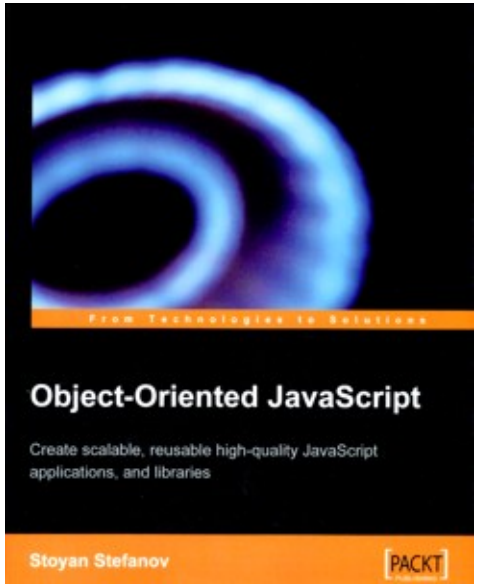
- John Resig: „Pro JavaScript Techniques“ Apress, ISBN 1-59059-727-3, 359 S.



his Web site: http://ejohn.org/

Books (cont.)

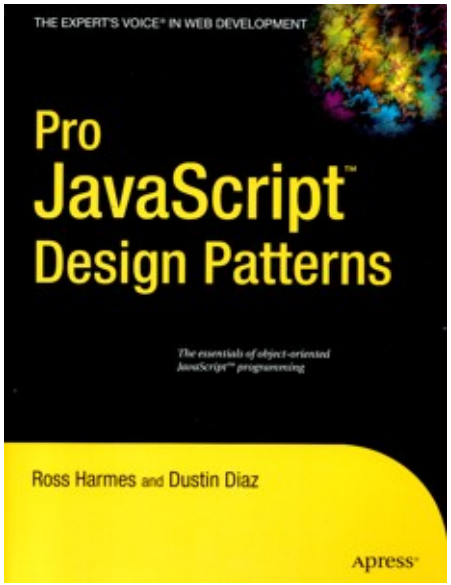
- Stoyan Stefanov: “Object-Oriented JavaScript“ Packt Publishing, ISBN 978-1-847194-14-5, 337 S.



his Web site: http://www.phpied.com/

Books (cont.)

- Ross Harmes, Dustin Diaz:
  „Pro JavaScript Design Patterns“
  Apress, ISBN 978-1-59059-908-2, 269 S.



Ross' Web-Seite: http://techfoolery.com/
Dustin's Web-Seite: http://www.dustindiaz.com/

The JavaScript Standard

- Standard ECMA-262
  ECMAScript Language Specification
  http://www.ecma-international.org/publications/standards/Ecma-262.htm

Questions?

- Ask now – or later

 tbahn@assono.de
 www.assono.de/blog
 +49/4307/900-401

assono
IT-Consulting & Solutions

- Presentation will be available at
  www.assono.de/blog/d6plinks/BLUG-2010-JavaScript