JavaScript für C# Entwickler

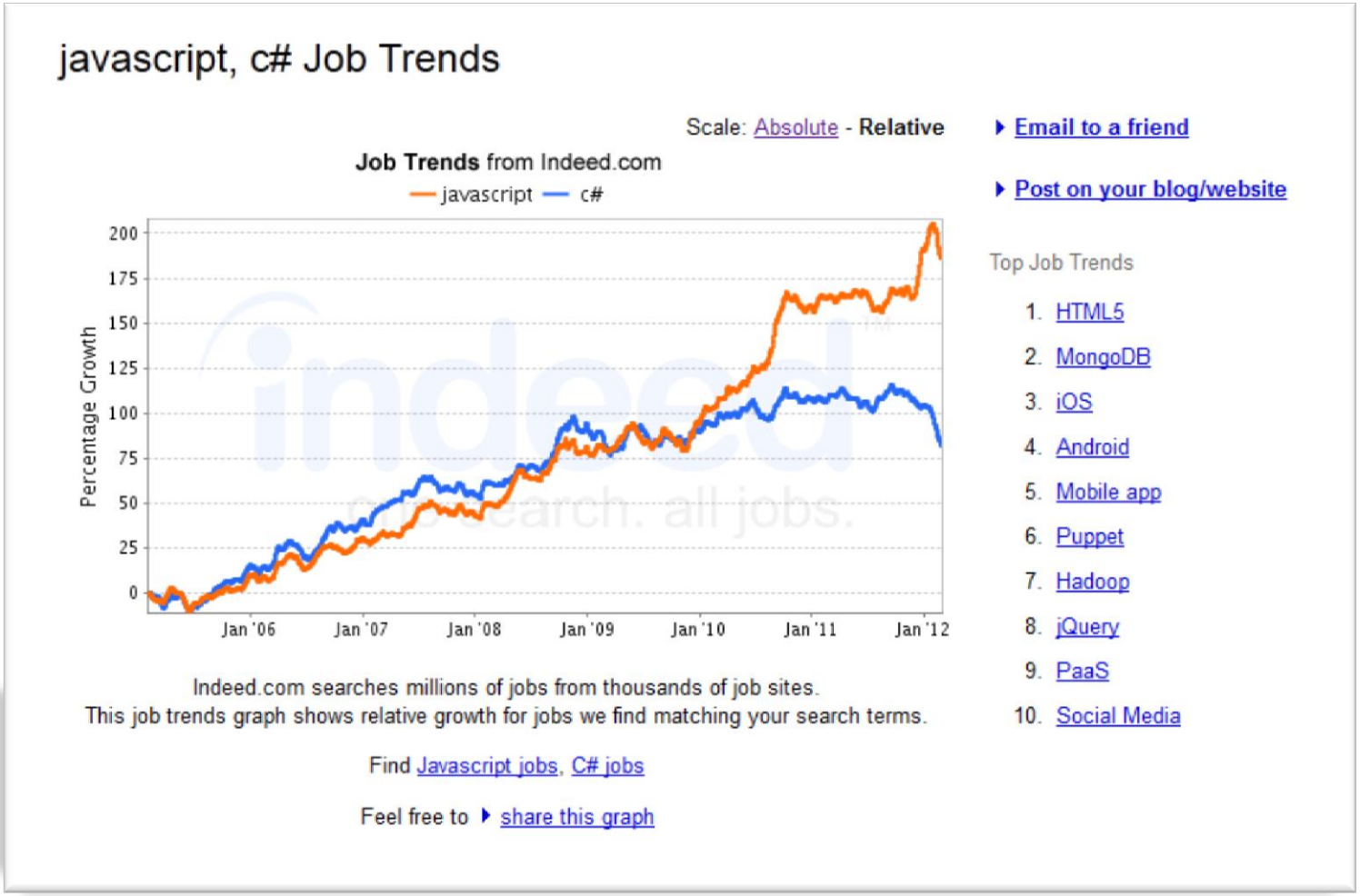
Hardy Erlinger
hardy.erlinger@netspectrum.de
www.netspectrum.de

Agenda

- Warum JavaScript?
- Sprachgrundlagen
- Patterns
- APIs
- Libraries

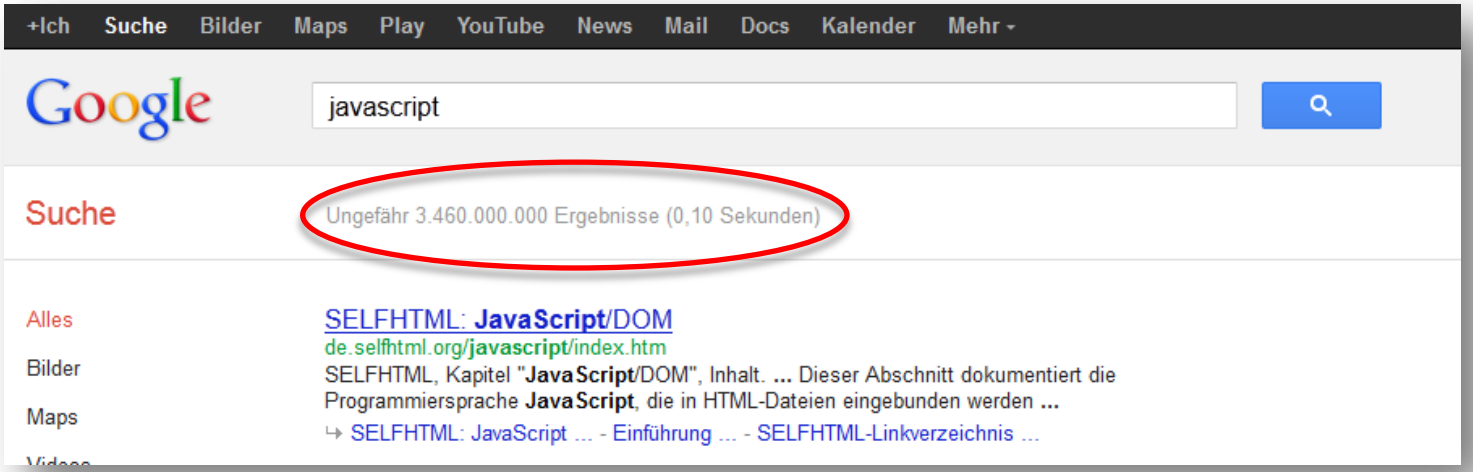
WARUM JAVASCRIPT?

Job Trends (USA)

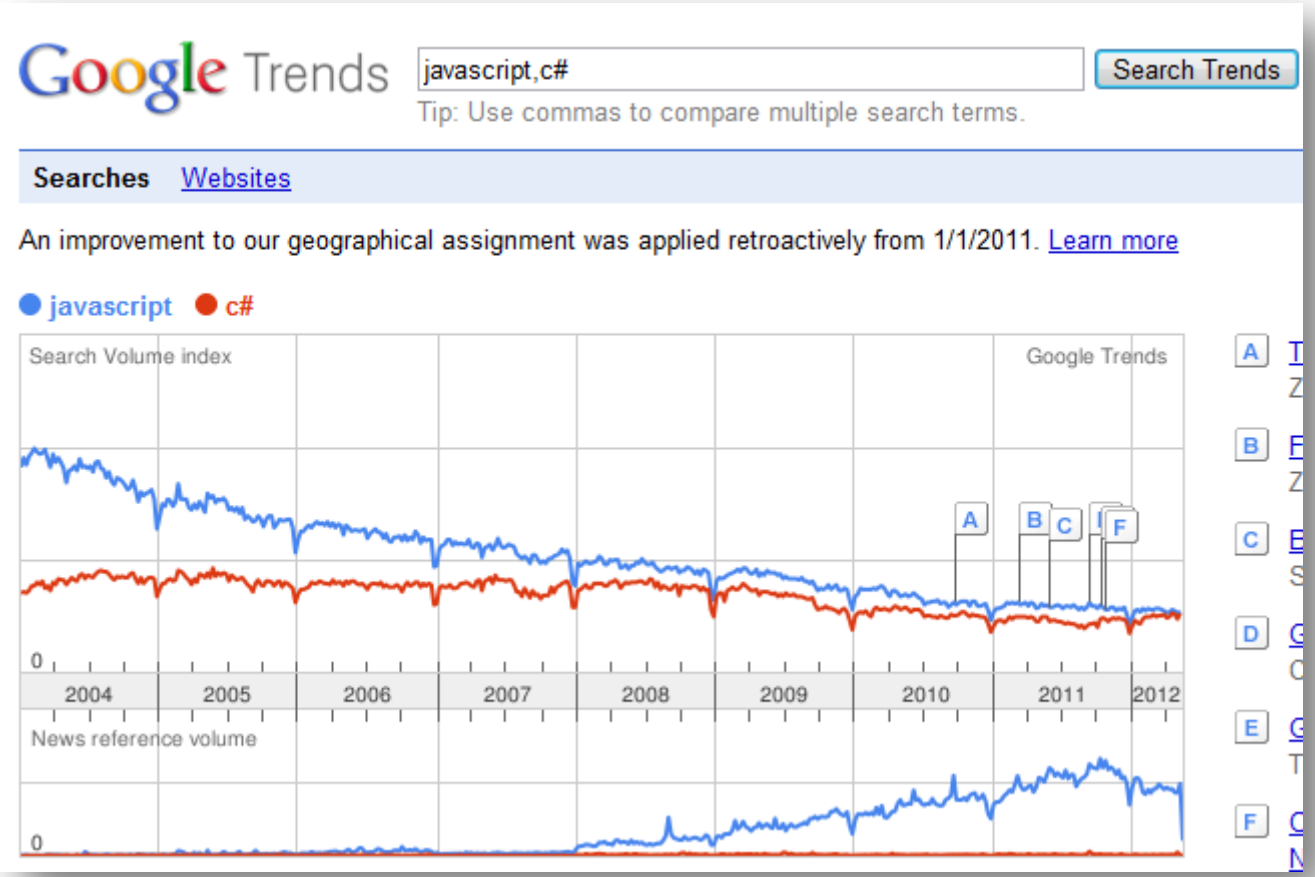


Quelle: http://www.indeed.com/jobtrends?q=javascript%2C+c%23&l=&relative=1

Suchergebnisse

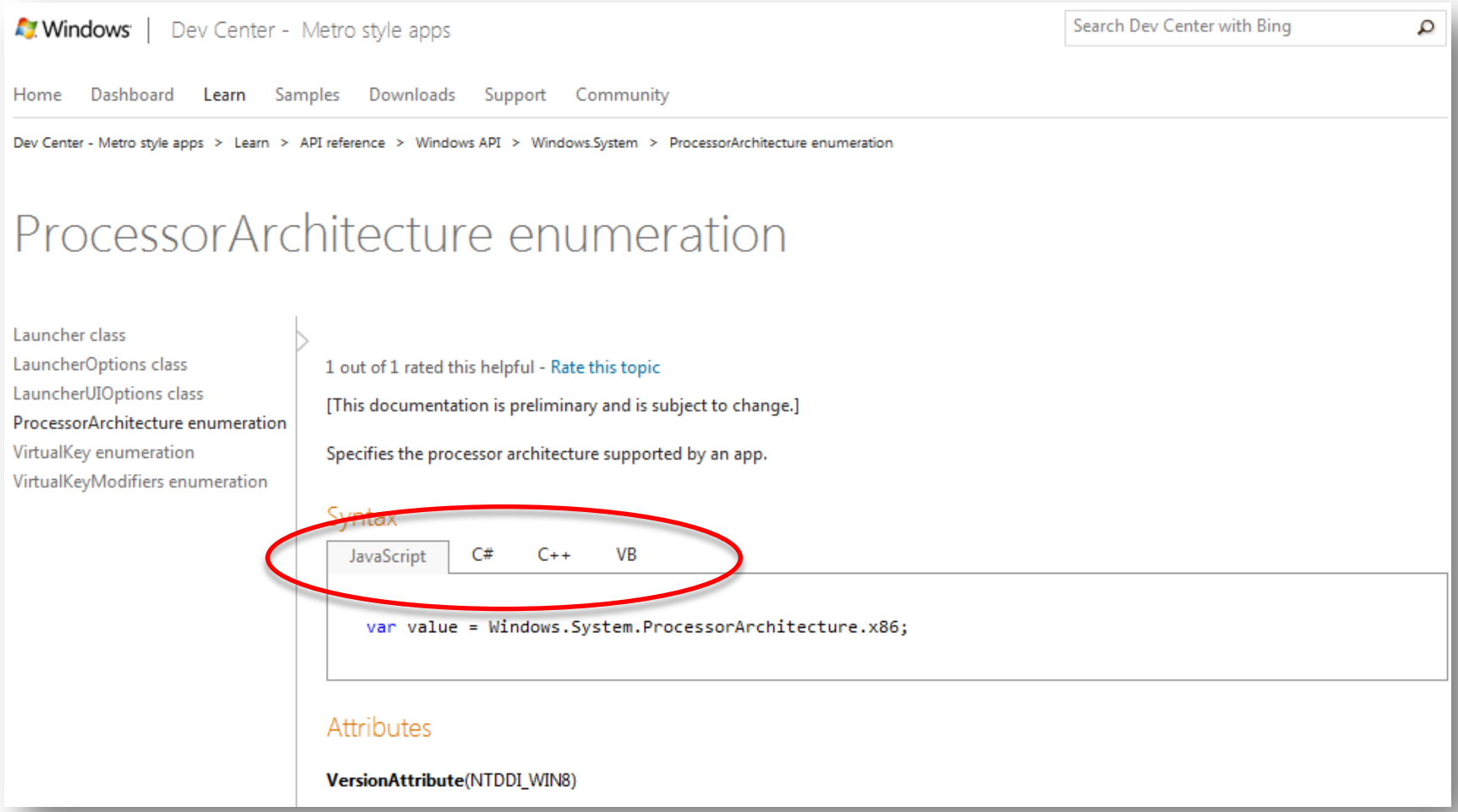


Google Trends



Quelle: http://www.google.com/trends/?q=javascript,c%23&ctab=0&geo=all&date=all

Windows 8


Windows | Dev Center - Metro style apps
Search Dev Center with Bing
Home Dashboard Learn Samples Downloads Support Community
Dev Center - Metro style apps > Learn > API reference > Windows API > Windows.System > ProcessorArchitecture enumeration
ProcessorArchitecture enumeration
Launcher class
LauncherOptions class
LauncherUIOptions class
ProcessorArchitecture enumeration
VirtualKey enumeration
VirtualKeyModifiers enumeration
1 out of 1 rated this helpful - Rate this topic
[This documentation is preliminary and is subject to change.]
Specifies the processor architecture supported by an app.
Syntax
JavaScript C# C++ VB
var value = Windows.System.ProcessorArchitecture.x86;
Attributes
VersionAttribute(NTDDI_WIN8)

Unterstützte Plattformen

C# und .NET Framework

- Desktop:
  - Windows
  - Linux (Mono)
- Browser (Silverlight-Plugin):
  - Windows
  - Mac
  - Linux (Moonlight)
- Mobil:
  - Windows Phone 7 Apps

JavaScript

- Desktop:
  - Windows 8
- Browser:
  - Alle modernen Browser
  - Kein Plugin erforderlich
- Mobil:
  - Alle aktuellen Smartphone- und Tablet-Betriebssysteme

„JavaScript is most despised because it isn‘t SOME OTHER LANGUAGE.“
(Douglas Crockford)

SPRACHGRUNDLAGEN

Datentypen

C# Primitive Types

- Integer:
  - byte
  - sbyte
  - short
  - int
  - long
  - ushort
  - uint
  - ulong
- Floating point:
  - float
  - double
- Logical:
  - bool
- Other:
  - char
  - decimal
  - IntPtr
  - UIntPtr
- Class objects:
  - object
  - string

JavaScript Types

- Primitive Types:
  - Number: 64-bit Fließkomma-zahl
  - String: immutable
  - Boolean
  - Null: kein Wert
  - Undefined: nicht initialisiert oder nicht vorhanden
- Object Types:
  - Object
  - Array
  - Function (callable Object)

Type Conversions: „thruthy“ vs. „falsy“

„falsy“

- `false`
- `null`
- `undefined`
- Leerstring
- Number `0`
- Number `NaN`

„truthy“

- Alles andere

Variablen

C#

- Deklaration:
  - `[Type] myVar = [value];`
  - Alternativ (Type Inference): `var myVar = [value];`
- Scope: Block
- Stark typisiert: zur Laufzeit keine Änderung des Typs erlaubt

JS

- Deklaration:
  - `var myVar = [value];`
- Scope:
  - Global-Scope oder Function-Scope
  - **kein** Block-Scope
- Deklaration ohne `var` -> Global Scope
- Schwache Typisierung: Typ kann jederzeit geändert werden

JavaScript Variable Hoisting*

Problem

- Variablen können an beliebiger Stelle innerhalb einer Funktion deklariert werden, verhalten sich jedoch, als ob sie alle zu Beginn der Funktion deklariert worden wären -> logische Fehler möglich
- Abhilfe: alle Variablen stets am Anfang einer Funktion definieren, egal wann sie benutzt werden

Single var Pattern

```
function myFunction() {
    var a = 1,
        b = 2,
        sum = a + b,
        i,
        j;

    //function body
}
```

* Engl. „hoist“: Hebevorrichtung, auch Flaschenzug. „to hoist something“: etwas heben, anheben.

Variable Hoisting: Beispiel

```
myname = "global"; //global variable
function func() {
    console.log(myname); // ??
    var myname = "local";
    console.log(myname); // ??
}
func();
```

Quelle: JavaScript Patterns, O'Reilly, S. 14f

Variable Hoisting: Beispiel

```
myname = "global"; //global variable
function func() {
    console.log(myname); // undefined
    var myname = "local";
    console.log(myname); // "local"
}
func();
```

Quelle: JavaScript Patterns, O'Reilly, S. 14f

JavaScript Objekte

- Definition: „An Object is an unordered collection of *properties*, each of which has a name and a value.“
- Keine Klasse erforderlich!
- Mögliche Property-Werte: alle Java-Script-Werte, einschließlich Funktionen
- Properties eines Objekts sind immer sichtbar. Kein Zugriffsschutz durch `private`, `protected` etc. wie in C#.
- Initialisierung als Object-Literal:
  - `var myObject = {};` <- Leeres Objekt
  - `var person = {name: "Paul", age: 42};`
- Zugriff:
  - `myObject.myProperty` oder
  - `myObject["myProperty"]`
- Neue Properties können jederzeit hinzugefügt und wieder entfernt werden:
  - `myObject.myNewProperty = 42;`
  - `delete myObject.myProperty`

JavaScript Arrays

- Definition: „An array is an ordered collection of values. Each value is called an *element*, and each element has a numeric position in the array, known as its *index*.“
- Index beginnt bei 0
- Nicht typisiert
- Größe dynamisch: wachsen und schrumpfen nach Belieben
- Initialisierung als Array-Literal:
```
var empty = [];
var numbers = [4,5,6,7,8,9];
var mix = ['hello',34,"34",true];
```
- Zugriff über Index:
```
numbers[2] // 6
numbers[6] = 10
```
<- erzeugt neues Element
- `myArray.length` gibt die Anzahl der Elemente zurück

JavaScript Funktionen

- Grundlegende Einheit der Modularisierung
- Sind Objekte, können also selbst Eigenschaften aufweisen
- Können beliebig tief ineinander verschachtelt werden
- Werden als „Methoden“ bezeichnet, sobald sie eine Property eines Objekts sind
- Deklaration: `function (myParam) { /*function body*/ }`
- Konstruktor-Funktionen:
  - intialisieren Objekte durch Zugriff auf `this`
  - werden mit `new` verwendet
- Namenskonvention: Konstruktor-Funktionen beginnen mit Großbuchstaben, gewöhnliche Funktionen mit Kleinbuchstaben

JavaScript Prototypes

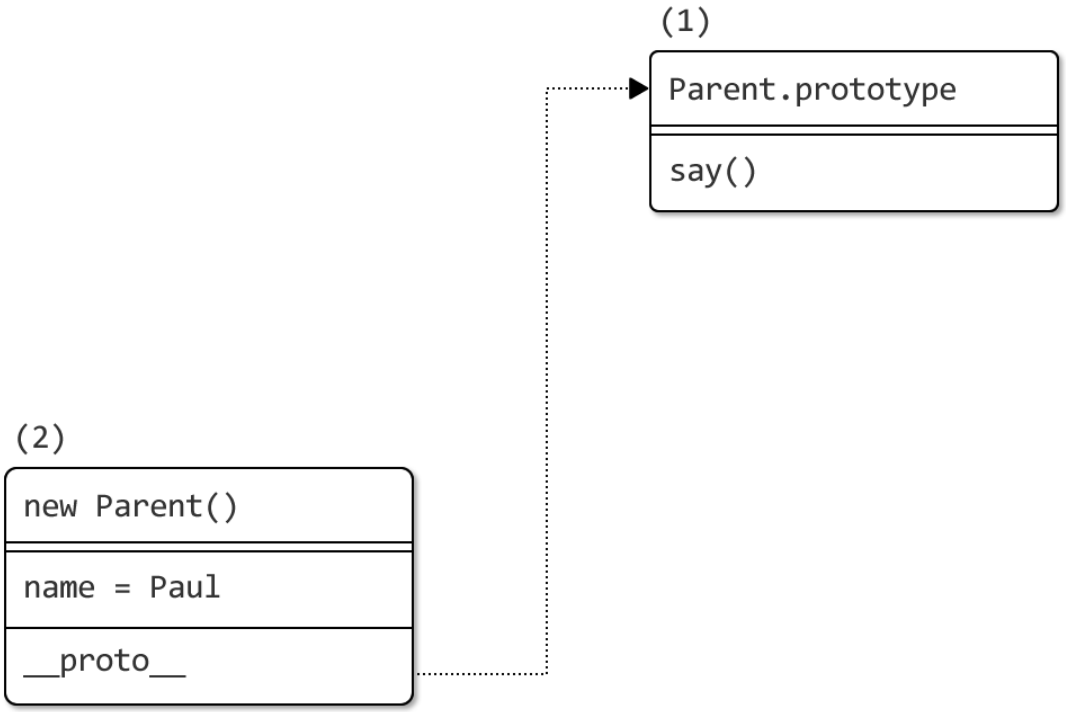
- Vererbungsmechanismus in JavaScript
- Jede Funktion hat eine Property `prototype` vom Typ Object
- Bei der Verwendung als Konstruktor-Funktion werden Properties von `prototype` auf alle Objekte vererbt, die mit dieser Funktion erzeugt wurden
- Bei Verwendung von Konstruktor-Funktionen mit `new` wird:
  1. … ein neues Objekt erzeugt, das Properties vom `prototype`-Objekt der Konstructor-Funktion erbt
  2. … die Konstructor-Funktion als Member des neuen Objekts aufgerufen. `this` zeigt auf das neue Objekt
  3. … das frisch initialisierte Objekt zurückgegeben

Vererbung mit Prototypes: Beispiel

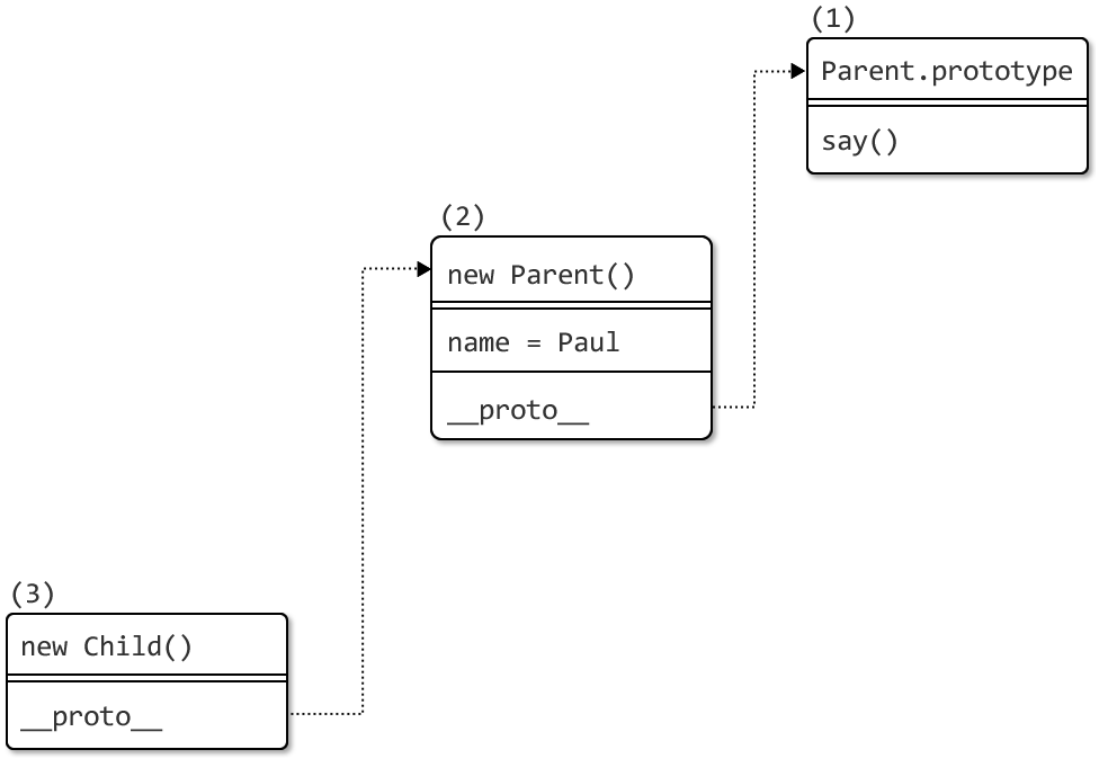
```
// parent constructor
function Parent(name) {
    this.name = name || 'Paul';
}

// adding functionality to the
prototype
Parent.prototype.say = function(){
    return this.name;
};

// empty child constructor
function Child(name){}

// inheritance happens here
Child.prototype = new Parent();
```

```
// testing ...
var kid = new Child();
kid.say() // 'Paul'

// modify the child property value
var kid2 = new Child();
kid2.name = 'Patrick';
kid2.say(); // 'Patrick'
```

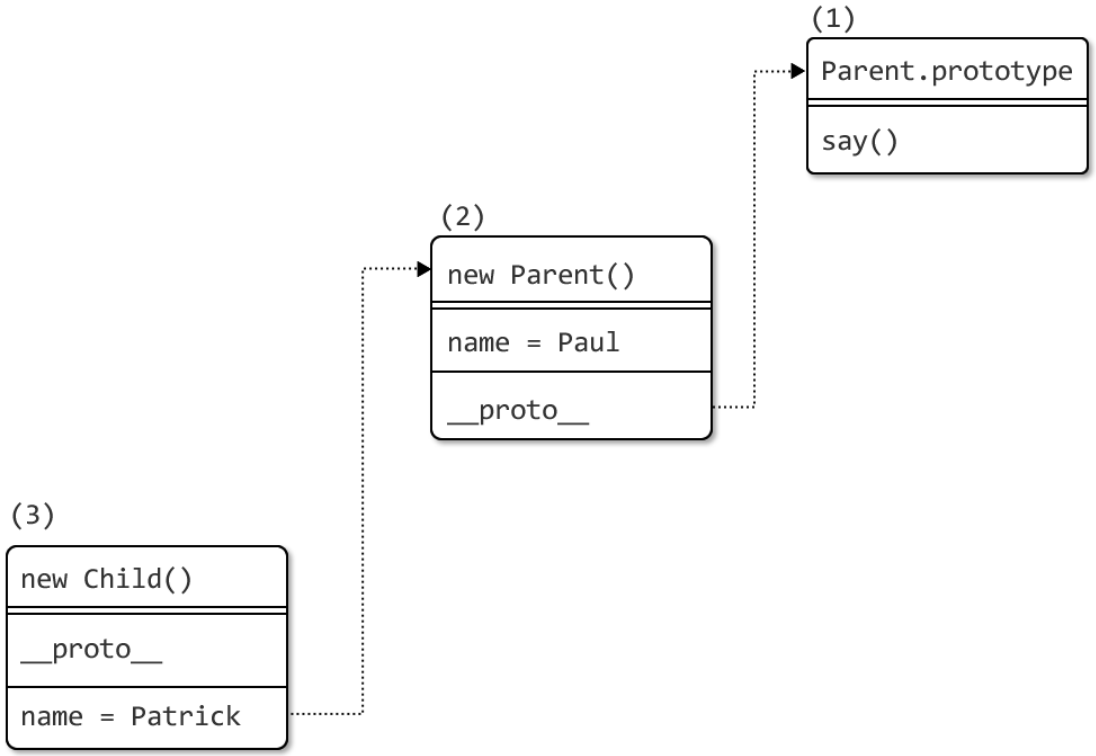
Quelle: JavaScript Patterns, O'Reilly, S. 116ff



Prototype-Kette für die Parent-Konstruktorfunktion



Prototype-Kette nach Vererbung



Prototype-Kette nach Vererbung und aktualisierter Child-Property

Exkurs: Klassen

C#

Definition:

„A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created ***instances*** of the class, also known as ***objects***. “

C# 4.0 Language Specification

JavaScript

- Sprache selbst enthält keine Klassen
- Ausdruck „Class“ wird dennoch häufig verwendet
- Mögliche Definition:
  Eine Reihe von Objekten, die Properties vom selben **Prototype-Objekt** erben

JavaScript Closures

- Closures sind Objekte, die aus zwei Dingen bestehen:
  1. Einer Funktion
  2. Der Umgebung, in der diese Funktion erzeugt wurde. Sie enthält alle lokalen Variablen, die dabei in Scope waren.
- Vereinfacht: bei verschachtelten Funktionen hat die innere Funktion Zugriff auf Variablenwerte der äußeren Funktion

Closures: Beispiel

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Quelle: https://developer.mozilla.org/en/JavaScript/Guide/Closures

Closures: Beispiel 2

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

```
console.log(counter.value()); // 0
counter.increment();
counter.increment();
console.log(counter.value()); // 2
counter.decrement();
console.log(counter.value()); // 1
```

Quelle: https://developer.mozilla.org/en/JavaScript/Guide/Closures

Serialisierung

C#

- Data Contract Serializer
- Binary Serializer
- XmlSerializer
- Etc.

JavaScript

- JavaScript Object Notation (JSON)
- Native Unterstützung in allen modernen Browsern
- Serialisierung:
```
var myString =
JSON.stringify(myObject);
```
- Deserialisierung:
```
var myObject =
JSON.parse(myString);
```
- JSON ist nicht auf JavaScript beschränkt. Parser für viele Plattformen vorhanden.

JSON: Beispiel

Array von Kategorien eines Blogs mit Name, Anzahl der Artikel und URL


```
[{
  "count":4,
  "name":"Tools",
  "url":"\/tools-category.aspx"
  },
 {
  "count":4,
  "name":"Webentwicklung",
  "url":"\/webentwicklung-category.aspx"
 },
 {
   "count":2,
   "name":"C#",
   "url":"\/c%23-category.aspx"
 },
 {
   "count":2,
   "name":"Visual Basic",
   "url":"\/visual%20basic-category.aspx"
 }]
```

PATTERNS (BEISPIELE)

Konventionen

- Globale Variablen und Konstanten werden mit Großbuchstaben benannt:
  ```
  var MYGLOBALVARIABLE = {}; bzw.
  var someValue = 3 * Math.PI;
  ```
- Namen von Konstruktorfunktionen beginnen mit einem Großbuchstaben:
  ```
  function Location(x,y) {}
  var myLocation = new Location(17,23);
  ```
- Namen von gewöhnlichen Funktionen beginnen mit einem Kleinbuchstaben:
  ```
  function add(x,y) {}
  var result = add(17,23);
  ```

Immediate Function Pattern

- Zweck:
  - Kapselung
  - Vermeidung von gleichnamigen Variablen im globalen Scope
- Herangehensweise:
  - anonyme Funktion wird in Klammern gesetzt und sofort ausgeführt
- Ergebnis:
  - alle Variablen bleiben lokal, globaler Scope wird nicht beeinträchtigt

Immediate Function Pattern: Beispiel

```
(function() {
    var days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],
        today = new Date(),
        msg = 'Today is ' +
                days[today.getDay()] + ', ' +
                today.getDate();

    console.log(msg);
})();
```

Quelle: JavaScript Patterns, O'Reilly, S. 69f

Namespace Pattern

- Zweck:
  - Reduzierung der Anzahl von globalen Variablen
  - Organisation des Codes
- Herangehensweise:
  - Eine globale Variable pro Applikation
  - Alle notwendigen globalen Variablen werden als Properties dieses Objekts implementiert
- Ergebnis: minimale Beeinträchtigung des globalen Scopes

Namespace Pattern: Beispiel

Vorher

```
// constructors
function Parent() {}
function Child() {}

// a variable
var some_var = 1;

// some objects
var module1 = {};
module1.data = {a: 1,b: 2};
var module2 = {};
```

Nachher

```
// global object
var MYAPP = {};

// constructors
MYAPP.Parent = function () {};
MYAPP.Child = function () {}

// a variable
MYAPP.some_var = 1;

// some objects
MYAPP.module1 = {};
MYAPP.module1.data = {a: 1,b: 2};
MYAPP.module2 = {};
```

Memoization Pattern

- Zweck: Caching von Funktionsrückgabewerten
- Herangehensweise:
  - Die Funktion wird um eine Objekt-Property erweitert
  - Diese speichert alle bisher zurückgegebenen Werte
- Ergebnis: aufwändige Operationen innerhalb der Funktion müssen nicht mehrfach ausgeführt werden

Memoization Pattern: Beispiel

```
var getData = function (id) {
    var result;
    if(!getData.cache[id]) {
        // ... expensive operation ...
        result = callExternalService(id);
        getData.cache[id] = result;
    }
    return getData.cache[id];
};
getData.cache = {};
```

Code Reuse Patterns

- Zahlreiche Varianten, von einfach bis sehr komplex
- Vererbung ist nur eine Möglichkeit von vielen
- Die meisten größeren JS-Libraries bringen einen Reuse-Mechanismus mit (u.a. Ext JS, Dojo, YUI)

Anleitungen zum Selbstbau

APIS

Wissenswerte Browser-APIs

- DOM: Selektieren von Elementen und Bearbeitung ihrer Eigenschaften
- DOM Events: Minenfeld – vielfach inkonsistente Implementierung, vor allem in IE -> Libraries erleichtern die Programmierung
- Ajax/XmlHttpRequest: asynchrone Kommunikation mit dem Server
- Client-side Storage: Cookies, localStorage, sessionStorage, offline Apps, IndexedDB
- Multimedia: Images, Video, SVG, Canvas
- HTML5: Geolocation, Web Workers, Web Sockets
- etc.
- etc.
- etc.
- -> Libraries verschaffen Abstraktion und rüsten Funktionalität nach, die in älteren Browsern nicht vorhanden ist

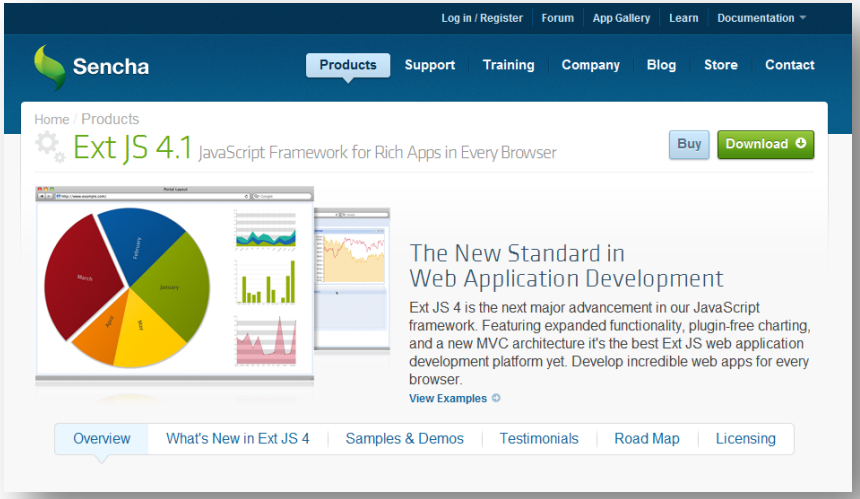
JAVASCRIPT LIBRARIES (BEISPIELE)

jQuery

- Open Source
- Mit Visual Studio ausgeliefert
- DOM + Ajax
- Hunderte von Plugins
- Einfacher Einstieg

http://jquery.com

Ext JS



http://www.sencha.com/products/extjs/

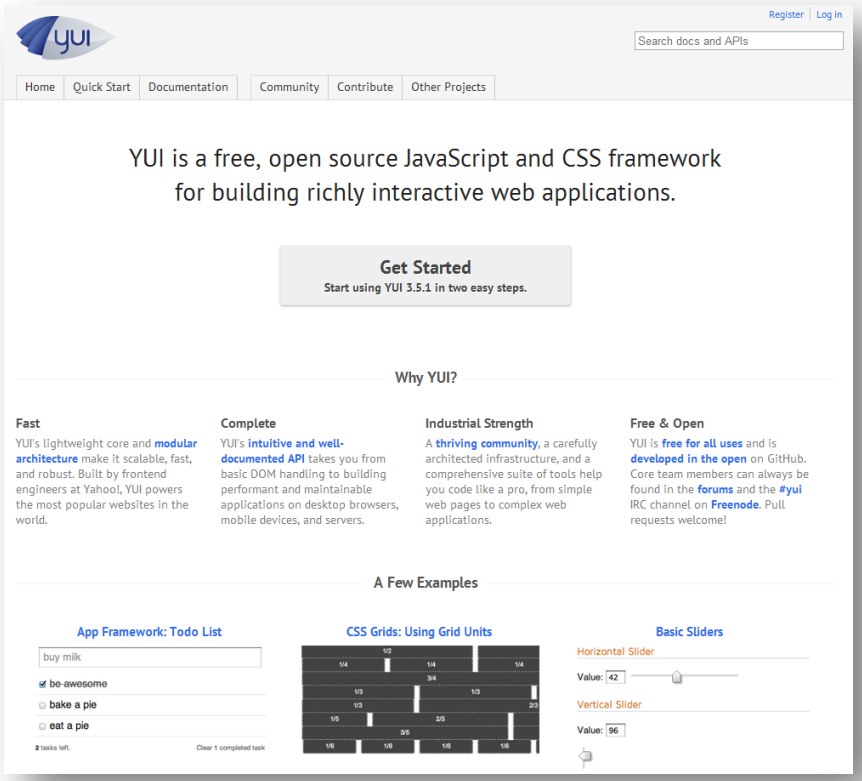
- Kommerzielles Produkt mit kostenfreier Basis-Library (Ext Core)
- Spezialisiert auf Single-Page-Applications
- Umfangreiche Funktionalität (MVC, Class System, Themes, Widgets, Charting etc.)
- Mobile Support durch Sencha Touch Library
- IDE: Sencha Architect

Dojo Toolkit

http://dojotoolkit.org/

- Open Source
- Modularer Aufbau
- Dynamisches Laden von Komponenten
- Umfangreiche Funktionalität (Packaging System, Themes, Widgets, Charting etc.)
- Umfangreichere Dokumentation wünschenswert

Yahoo! User Interface Library (YUI3)



http://yuilibrary.com

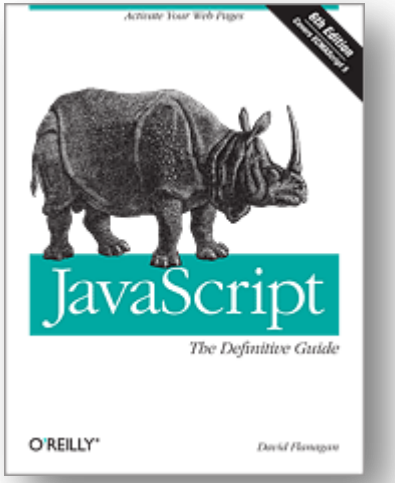
- Open Source
- Modularer Aufbau
- Dynamisches Laden von Komponenten
- Umfangreiche Funktionalität (MVC- und Widget-Infrastruktur, CSS-Framework, Widgets, Test Framework, Profiler, Charting etc.)
- Sehr ausführliche Dokumentation mit zahlreichen Beispielen
- YUI2 noch vielfach verwendet, aber inzwischen deprecated -> aktuell: YUI3

RESSOURCEN

Bücher

Douglas Crockford:
JavaScript: The Good Parts
O'Reilly 2008

Stoyan Stefanov
JavaScript Patterns
O'Reilly 2010



David Flanagan
JavaScript, The Definitive Guide
O'Reilly 2011

Online

- Mozilla JavaScript Reference: https://developer.mozilla.org/en/JavaScript/Reference
- Douglas Crockford on JavaScript: http://www.youtube.com/playlist?list=PL7664379246A246CB
- JsLint (Syntax-Checker): http://jslint.com
- Online-IDEs:
  - JsFiddle: http://jsfiddle.net
  - Jsbin: http://jsbin.com

FRAGEN?