

JavaScript und Java

... haben nichts miteinander zu tun

Der Name "JavaScript" wurde von Netscapes Marketing-Abteilung erfunden.

- Java eher für teure Komponenten (traditionelle Softwaretechnik)
- JavaScript für Webdesigner

JavaScript: Geschichte

1995 von Brendan Eich bei Netscape erfunden (in einer Woche)

1996: Schnelle Verbreitung

1996: JScript von Microsoft

1997: Erste Version des ECMA-Script-Standards für Sprachkern

1998: DOM Level 1 als Standard für Dokumentrepräsentation

2000: DOM Level 2 für Views, Events, Traversal, ...

2005: JS-Revival mit Ajax (Google Maps, GMail, ...)

Philosophie der Sprache

- Einfaches Copy/Paste von Code-Schnipseln
- Tolerieren kleiner Fehler (fehlende Strichpunkte)
- Einfaches Event-Handling
- Wähle einige wenige, mächtige Primitiva:
 - Higher-order Funktionen für prozedurale Abstraktion
 - Prototyp-basierte Objekte für alles andere
- Alles andere weglassen

Interpreter

JavaScript wird (fast) immer interpretiert

- Einbetten in HTML: Mit `script`-Tag und Code im Kommentar

```
<script type="text/javascript">  
  <!--  
    document.write("JavaScript is here!");  
  -->  
</script>
```

- Außerdem: Standalone-Interpreter `js`

Syntax

Statements, durch `;` getrennt

Blöcke innerhalb von `{ ... }`

Kontrollstrukturen ähnlich zu C/Java:

- `if (test) stmt else stmt`
- `do stmt while (stmt)`
- `while (stmt) stmt`
- `for (expr ; expr ; expr) stmt`
- `for (var in expr) stmt`
- `switch (expr) CaseBlock`

Exception-Handling wie in Java

Variablendefinition

Syntax:

```
var name = expr
```

Achtung:

- Globales **var** kann auch weggelassen werden
- Lokaler Scope: Funktionsrumpf (Blöcke begrenzen den Scope *nicht*)
- Lokale Definition gilt im gesamten Scope, auch vor der Definition

Typen in JavaScript

JavaScript ist latent getypt

- Undefined-Typ mit einzigem Wert **undefined**
- Null-Typ mit einzigem Wert **null**
- Zeichenketten: Literale sind in einfachen oder doppelten Anführungszeichen eingeschlossen
- Booleans: **true** und **false**
- Zahlen
- Object-Typ: Alle Objekte

Automatische Typkonversion

JS führt automatische Typkonversion durch, wenn dies für die Anwendung von Primitiva notwendig ist:

- Nach Boolean: `undefined`, `0`, `NaN`, `""` ergeben `false`, alles andere `true`
- In eine Zahl: Strings parsen, für Objekte `toNumber`-Methode
- In einen String: Literale als Strings

Automatische Konversionen führen leicht zu subtilen Fehlern

Funktionen sind Werte erster Klasse

Syntax für Funktionsdefinition:

```
function fname ( para-list ) { statement-list }
```

- Funktionen geben Wert mittels **return** zurück
- Ohne **return** ist der Rückgabewert **undefined**
- Name ist optional (-> Anonyme Funktionen)

Beispiel für Funktionen

```
function add_first(x) {  
  function add_second(y) {  
    return x + y;  
  }  
  return add_second;  
}  
var add_ft = add_first(42);  
add_ft(23); // -> 64;
```

Funktionsaufruf

Syntax: `f (arg , ...)`

Achtung: Es können auch mehr oder weniger Argumente übergeben werden, als die Funktionsdeklaration vorsieht:

- Zu wenig Argumente: Restliche Parameter werden an **undefined** gebunden
- Zu viele Argumente verfallen

Aber: Array **arguments** enthält alle Argumente

Damit möglich: Funktionen mit variabler Parameteranzahl

Einige Primitiva

- `eval()` erwartet Code als String, wertet den Code aus
- Typkonversion: `Number()`, `Boolean()`, `String()`
- Typ als String: `typeof()`
- `+` für Addition und Stringkonkatenation
- ...

OO in JavaScript

Objekt-basiert:

- Es gibt keine Klassen sondern nur Objekte
- Erstellen neuer Objekte unter Verwendung bestehender Objekte

Motivation für Prototypen

Problem mit Klassen: Wegen Vererbung lassen sich die Auswirkung von Veränderungen an einer Klasse nicht absehen (*fragile base class problem*)

Ausweg: Klassen abschaffen

Außerdem: Eigentlich wollen wir Objekte haben, wozu brauchen wir da Klassen?

Wozu Klassen?

Leistungen von Klassen:

- Erschaffen Objekte
- Definieren Felder für Objekte
- Definieren Methoden und stellen deren Implementierung bereit

Wiederverwendung durch Vererbung

OO ohne Klassen

- Erschaffen von Objekten ohne Klassen:
 - Ad-hoc beschreiben
 - Bestehende Objekte klonen
 - Objekte erschaffen und auf bestehende verweisen lassen
- Felder und Methoden: Direkt zu Objekten hinzufügen

Wiederverwendung beim Erschaffen

Objekte in JS

Objekte sind Sammlungen von Properties

Eine Property besteht aus:

- Name (String)
- Wert
- Menge von Attributen (ReadOnly, DontEnum, DontDelete, Internal)

Objekte on-the-fly erstellen

Syntax: `{ name : expr , ... }`

Beispiel:

```
var holger = {name : "Holger", age : 30}
```

Erstellt ein Objekt mit den aufgeführten Properties

Zugriff auf Properties

Property-Name liefert deren Wert. Syntax:

- *obj . name*
- *obj [expr]*

Dabei wird *expr* in einen String konvertiert

```
js> var martin = {name : "Martin", age : 31}
```

```
js> function describe (person){  
  return person.name + " ist " + person["age"];  
}
```

```
js> describe (holger);
```

```
Holger ist 30
```

```
js> describe (martin);
```

```
Martin ist 31
```

Methoden

Bis jetzt hatten wir nur "Felder", keine Methoden

Aber: Funktionen sind Werte erster Ordnung!

⇒ Speichern Methoden als Property

```
js> function f(x){ return x;}  
js> var o = {myFun : f};  
js> o.myFun(42);  
42
```

Lies Aufruf als `(o.myFun)(42)`, d.h. normaler Funktionsaufruf

this

(Haupt-)Sinn einer Methode: Zugriff auf Objekt

In JavaScript (wie üblich) über `this`

Problem: `this` passt nicht zu Funktionen als Werten:

- Wert einer Funktion ist eine Closure (->Mini)
- Closure: Datenstruktur, bestehend aus Parametern, Code und Umgebung
- Alle Variablen im Code einer Closure sind also statisch gebunden
- Wert für `this` ist aber bei Erstellen der Closure nicht bekannt

Lösung: `this` ist *dynamisch gebunden* (-> Mini)

Methoden mit Zugriff auf das Objekt

Methoden greifen also mittels `this` auf das Objekt zu:

```
js> function squareX(){  
  this.x = this.x * this.x;  
}
```

```
js> var o = {squareMyX : squareX, x : 23}
```

```
js> o.squareMyX();
```

```
js> o.x;
```

```
529
```

Felder und Methoden hinzufügen

Zu einem Objekt können nachträglich Properties hinzugefügt werden.

Syntax:

- *obj . name = expr*
- *obj [name] = expr*

Methoden hinzufügen: *expr* kann (natürlich) auch Funktion sein

```
holger.room = 113;  
martin ["room"] = 114;  
martin.room - holger ["room"] // -> 1
```

Globales Objekt

Woran ist `this` gebunden, wenn eine Top-Level-Funktion aufgerufen wird?

Antwort: An das *globale Objekt*

Die globalen Funktionen und Variablen sind Properties dieses Objekts

```
js> function self() { return this; }
```

```
js> var t = 42;
```

```
js> self().t;
```

```
42
```

```
js> self().self().t
```

```
42
```

Objekte mittels Funktionen erstellen

Bis jetzt haben wir alle Properties von Hand erzeugt

Nachteile: Wildwuchs, Polymorphie nur per Zufall, umständlich

Idee: Lassen Funktionen die Felder hinzufügen

```
function makePerson (obj, name, age, room) {  
    obj.name = name;  
    obj.age = age;  
    obj.room = room;  
}  
var holger = {};  
makePerson(holger, "Holger", 30, 113);
```

new

Diese Idee ist in JavaScript schon eingebaut:

- `new f (args)` erzeugt neues Objekt und ruft `f(args)` auf
- Beim Aufruf ist `this` an das neue Objekt gebunden
- `f` heißt dann *Konstruktor*

```
js> function Person (name, age, room)
  this.name = name;
  this.age = age;
  this.room = room;
}
js> var martin = new Person("Martin", 31, 114);
js> martin.room
114
```

Methoden ohne globale Funktionen

Problem mit "Methode = Funktion in Property": Funktionen müssen vorher definiert sein, sind also global sichtbar

⇒ Namespace-Pollution, Methoden funktionieren nur auf bestimmten Objekten

Lösung: Funktionen lokal im Konstruktor definieren

Funktionen im Konstruktor definieren

```
js> function Point (x,y){  
  this.x = x;  
  this.y = y;  
  
  function dist(){  
    return Math.sqrt(this.x*this.x + this.y*this.y);  
  }  
  this.dist = dist;  
}
```

```
js> var p = new Point (3,4);
```

```
js> p.dist();
```

```
5
```

Alternative: Anonyme Funktionen

Klonen

Andere Methode um an neue Objekte zu kommen: Bestehende Objekte kopieren ("klonen")

```
function cloneObj(from) {  
    to = {};  
    for (i in from) {  
        to[i] = from[i];  
    }  
    return to;  
}  
var holger_march = cloneObj(holger);  
var p2 = cloneObj(p);
```

Geklonte Objekte

Nach dem Klonen: Objekte anpassen

```
holger_march.room = 207;
```

```
move(holger, holger_march); // Renovierung!
```

Auch für Methoden möglich:

```
function dist_manhattan(){  
  return this.x + this.y;  
}  
p2.dist = dist_manhattan;
```

Vererbung

Neue Feldern und Methoden erweitern das geklonte Objekt

```
var drHolger = cloneObj(holger);  
drHolger.title = "Dr.";  
  
drHolger.promote() = function(){  
    this.title = this.title + " habil."  
}
```

Wie bei Vererbung

Abstraktion über Vererbung

Konstruktor-Funktion

```
function TitlePerson(person, title) {  
    self = cloneObj(person);  
  
    self.title = title;  
    self.promote = function(){...}  
    return self;  
}  
  
drHolger = TitlePerson(holger, "Dr");
```

Anmerkungen

1.) Zu "globales `var` kann weggelassen werden":

- Auf oberster Ebene führt `x = 3` zu einer Bindung
- Läßt man lokal `var` weg, so wird ebenfalls an eine globale Variable zugewiesen/gebunden

2.) Die Attribute von Properties kann man nicht setzen

- Einige Primitiva setzen die Attribute von bestimmten Properties
- Der User sieht nur die Auswirkungen

JavaScript und Prototypes

JavaScript basiert also auf Prototypes/Delegation

Oberster Prototype ist `Object.prototype`

Primitiva, "geerbt" von `Object.prototype`:

- `toString()` Stringrepräsentation
- `hasOwnProperty(v)` Ist Property V direkt im Objekt vorhanden?
- `isPrototypeOf(v)` Ist dieses Objekt ein Prototype von V?
- `constructor` Funktion, die `new` aufgerufen hatte, um das Objekt zu erstellen
- `propertyIsEnumerable(v)` Hat die Property V das Attribut `DontEnum` nicht gesetzt

Weitere Techniken für objekt-basierte Programmierung

- Explizite Vererbung
- Mode-Switching
- Traits

Gehen über JavaScript hinaus

Explizite Vererbung

Schön wäre: von mehreren bestehenden Objekten erben

Bis jetzt: Erstellen aus bestehendem Objekt hat alle Felder/Methoden vererbt

Problem: woher wissen wir, dass sich die ererbten Namen nicht überschneiden?

Lösung: Die zu erbenden Namen explizit angeben

⇒ Keine Überraschungen

Explizite Vererbung implementieren

Explizit angeben, welche Felder vererbt werden sollen

Bei Embedding:

- Methoden einzeln (und selektiv) in neues Objekt kopieren

Delegation bräuchte anderes Modell als JavaScript:

- Prototype-Verweis für gesamtes Objekt entfällt
- Jede Methode verweist einzeln auf Methode in anderem Objekt
- Dann Methoden einzeln (und selektiv) verlinken

Explizite Vererbung und Klassen

- In klassenbasierte Sprachen nicht vorgesehen, da Subklassen das Protokoll nur erweitern sollen
- Explizite Vererbung schränkt das Protokoll aber ein
- Kein Problem in objekt-basierten Sprachen, da die Objekte eigenständig sind

Mode-Switching

Anforderung: Verhalten eines Objekts soll sich in manchen Situationen komplett ändern

Beispiel:

- Window-Manager repräsentiert Fenster als Objekt
- Beim Minimieren ändert sich das Verhalten des Objekts völlig:
 - Draw-Request soll Icon zeichnen
 - Größe ändern nicht mehr möglich
 - Doppelklick stellt ursprüngliche Größe wieder her

Mode-Switching in objekt-basierten Sprachen

Idee für objekt-basierte Sprache: ändere das Objekt von dem "geerbt" wurde zur Laufzeit

Anforderungen

- Beim Klonen: Müssen die Methoden ändern können
- Bei Delegation: Müssen den Prototype-Verweis ändern können

Anmerkungen zu Mode-Switching

Methoden mitten im Ablauf auszuwechseln ist gefährlich, aber mächtig

⇒ nur diszipliniert verwenden

In klassenbasierten Sprachen nur von Hand mittels zusätzlicher Indirektion realisierbar (State-Pattern)

Traits

Pattern, entstanden bei der Verwendung objekt-basierter Sprachen mit Delegation

Traits sind Objekte, die nur dazu da sind, anderen Objekten Methoden zur Verfügung zu stellen.

- Dienen als Prototype
- Können selbst wieder Traits als Prototype haben (-> Hierarchie)
- Können allein nicht verwendet werden (Felder fehlen)
- Verwendung zusammen mit Objekten, geklont werden um die Felder zu definieren

Unschön: Traits sind keine "echten" Objekte

Traits und Klassen

Traits ahmen Klassen nach:

- Traits halten die Implementierung für andere Objekte
- Felder (Zustand) nicht in Traits, sondern entsteht aus Beschreibung
- Dynamic-Dispatch sucht Implementierung durch Ablaufen der Traits-Kette

Unterschiede:

- Felddefinition getrennt von Methodendefinition
- Zweistufige Erstellung von Objekten: Erst Zustand, dann Verweis auf Methoden

Zusammenfassung Prototypes/Delegation

Vorteile:

- Speichereffizient

Nachteile:

- Komplizierte Abhängigkeitsnetze entstehen
- Zugriff auf Felder/Methoden muss eventuell Prototype-Verweis folgen

Zusammenfassung Klonen/Embedding

Vorteile:

- Objekte sind unabhängig von Ihren Originalen
- Zugriff auf Felder/Methoden ist schnell
- Felder und Methoden sind immer zusammen (-> Gut für verteilte Systeme)

Nachteile:

- Hoher Speicherverbrauch (Aber: Optimierung möglich)

AJAX

AJAX steht für "Asynchronous Javascript And Xml"

Verbindung aus clientseitiger und serverseitiger Webprogrammierung

JavaScript lädt bei Bedarf XHTML-Code vom Server nach

Dann fügt JavaScript diesen Code in das aktuelle Dokument ein (-> DOM)

JavaScript 2.0

Nächste Version von JavaScript

Ziele:

- Versuch, einige Fehler zu korrigieren

Nicht-Ziele:

- JavaScript zur Allzweck-Programmiersprache zu machen

Die Veränderungen in JavaScript 2.0

- Typen
 - Optionale Typdeklarationen für Variablen und Parameter
 - Compiler signalisiert Fehler, wenn zur Laufzeit auf jeden Fall ein Fehler auftritt
- Strict-Mode
 - Variablen müssen deklariert werden
 - Funktionsdeklarationen sind unveränderbar
 - Funktionsaufruf überprüft Parameteranzahl
 - Semikolons dürfen nicht weggelassen werden

Weitere Veränderungen in JavaScript 2.0

- Klassen zusätzlich zu Prototypes
 - Klassen bieten Basis für Zugriffsschutz und Versionen
 - Prototype-basierte Sprachen simulieren Klassen sowieso
 - Programmieren mit Prototype-Hierarchien ist schwierig
 - Klassen sind selbstbeschreibender als Prototypen
 - Die Benutzer wollen Klassen
- Namespaces, Versioning, Packages
 - Namespaces sind Werte
 - Packages als Modulsystem
 - Versionen für Zugriff auf Packages

Zusammenfassung JavaScript

Objekt-basiert, mit Prototypes/Delegation

Winziger Sprach-Kern:

- First-class Funktionen
- Objekte als Menge von Properties
- Prototype-Verweis

Größte Schwächen:

- Felder immer public
- Automatische Typkonversion

Zusammenfassung JavaScript

Dynamisches Hinzufügen von Feldern/Methoden

- Eigentlich unüblich
- Statisch nicht typbar
- Andere Sprachen:
 - Spezielle Syntax für Erstellen von Objekten
 - Dabei Angabe von neuen Feldern und Methoden
- Erlaubt aber Experimentieren (-> Kloning)

Zusammenfassung Objekt-basierte Sprachen

- Können Klassen simulieren
 - Elementarer als Klassen
 - Flexibler als Klassen
 - Damit neue Einblicke in Klassen
 - Delegation vs. Embedding
 - Bei Embedding wird Dynamic-Dispatch überflüssig
 - Klassen: Traits + Zustand + Suche
 - Methodenupdate für einzelne Objekte ist möglich
- ⇒ Objekte sind genug