

[SELFHTML aktuell](#) [Artikel](#) [JavaScript](#)



Organisation von JavaScripten



- ↓ [Mathias Schäfer](#)
- ↓ [Das Schichtenmodell: Trennung von Inhaltsstruktur, Präsentation und Verhalten](#)
- ↓ [Warum sind geordnete und strukturierte Skripte sinnvoll?](#)
- ↓ [Unstrukturierte Skripte](#)
- ↓ [Objektstrukturen mit Object](#)
- ↓ [Object-Literale](#)
- ↓ [Object-Methoden in anderen Kontexten ausführen](#)
- ↓ [Möglichkeiten der Datenspeicherung](#)
- ↓ [Eigene Objekte](#)
- ↓ [Methoden eigener Objekte in anderen Kontexten ausführen](#)
- ↓ [Einführung in Closures](#)
- ↓ [Anwendung von Closures](#)
- ↓ [Closures zur Kapselung bei Object und eigenen Objekten](#)
- ↓ [Alternativlösungen zur Kontext-Problematik: bind\(\) und bindAsEventListener\(\)](#)
- ↓ [Modularisierung und Namensräume](#)
- ↓ [Ausblick auf JavaScript-Frameworks](#)
- ↓ [Literaturhinweise](#)



Mathias Schäfer

E-Mail: [✉ zapperlott@gmail.com](mailto:zapperlott@gmail.com)



Bei Fragen zu diesem Beitrag bitte den Autor des Beitrags kontaktieren!



Das Schichtenmodell: Trennung von Inhaltsstruktur, Präsentation und Verhalten

Im modernen Webdesign kommt den Webtechniken HTML, CSS und JavaScript jeweils eine bestimmte Rolle zu. HTML soll die Texte sinn- und bedeutungsvoll strukturieren, indem z.B. Überschriften, Listen, Absätze, Datentabellen, zusammenhängende Bereiche sowie wichtige Abschnitte, Zitate usw. als solche ausgezeichnet werden. CSS ist dafür da, die Regeln für Darstellung dieser Inhalte vorzugeben, sei es auf einem Desktop-Bildschirm, auf einem Handheld, auf Papier oder anders.

Um eine Website möglichst effizient und einfach zu entwickeln sowie sie nachträglich mit geringem Aufwand pflegen zu können, sollen diese beiden Aufgaben strikt voneinander getrennt werden: Im HTML-Code werden keine Angaben zur Präsentation gemacht. Im Stylesheet befinden sich demnach alle Angaben zur Präsentation in möglichst effizienter Weise. Dadurch müssen im HTML-Code nur genau so viele Angriffspunkte für CSS-Selektoren gesetzt werden, wie gerade nötig sind (z.B. zusätzliche `div`- oder `span`-Elemente sowie `id`- und `class`-Attribute). Ein und dasselbe Dokument kann auf diese Weise durch den Wechsel des Stylesheets ein völlig anderes Layout bekommen. Aber auch ganz ohne Stylesheet sind die Inhalte noch sinnvoll strukturiert und die Inhalte zugänglich.

JavaScript kommt in diesem Konzept die Aufgabe zu, dem Dokument »**Verhalten**« (*Behaviour*) hinzuzufügen. Damit ist gemeint, dass das Dokument auf gewisse Anwenderereignisse reagiert und z.B. Änderungen im Dokument vornimmt. Diese **Interaktivität** wird dem Dokument automatisch hinzugefügt – im HTML-Code sollte sich kein JavaScript in Form von Event-Handler-Attributen befinden (`onload`, `onclick`, `onmouseover` usw.). Stattdessen werden Elemente, denen ein bestimmtes Verhalten hinzugefügt werden soll, z.B. mit einer Klasse markiert. Zeitgemäße Skripte werden automatisch beim Ladens des Dokuments aktiv und starten die Ereignisüberwachung an den betreffenden Elementen. Diese Anwendung von JavaScript nennt sich  [Unobtrusive JavaScript](#), »unaufdringliches« JavaScript, oder auch  [DOM Scripting](#).



Warum sind geordnete und strukturierte Skripte sinnvoll?

Sobald eine Webseite mittels »unaufdringlichem« JavaScript aufgewertet und interaktiver gestaltet wird, entstehen komplexe Skripte. Der Ablauf des Scripts wird in Teilaufgaben geteilt, die verschiedene Funktionen übernehmen. Anstatt ein und denselben Code zu wiederholen, wird er in eine Funktion ausgelagert, die Parameter entgegennehmen kann. Insbesondere das Event-Handling erfordert

verschiedene Funktionen, die als Event-Handler dienen oder bei der Verarbeitung von Events helfen.

Um bestimmte Funktionalität umzusetzen, werden meist unzählige Variablen und mehrere Funktionen benutzt – ein zusammenhängendes Script, dessen Teile miteinander arbeiten. In ein Dokument werden gerne mehrere »unaufdringliche« Scripte verschiedenen Ursprungs eingebunden. Sie sollen unabhängig voneinander arbeiten, aber auch reibungslos miteinander funktionieren. So stellen sich folgende Fragen:

- Wie kann man Scripte **klar strukturieren**, um zusammenhängende Funktionen übersichtlich zu gruppieren?
- Wie kann man **wiederverwendbare** Scripte schreiben, die einfach und zentral konfigurierbar sind?
- Wie kann man Scripte schreiben, die **problemlos mit anderen Scripten zusammenarbeiten**, ohne dass sie sich gegenseitig dazwischenfunken?

Diese Fragen sind insbesondere dann wichtig, wenn mehrere Personen an einem Script arbeiten, wenn ein Script für andere Webautoren veröffentlicht werden soll oder wenn man selbst sein eigenes Script auch nach einiger Zeit wieder verstehen will.



Unstrukturierte Scripte

Die meisten Scripte, die JavaScript-Programmierer im Netz anbieten, liegen in einer gesonderten Datei vor und sind darüber hinaus unstrukturiert. Es handelt sich um eine äußerlich lose Sammlung von dutzenden **globalen** Variablen und Funktionen.

```
var variable1 = "wert";
var variable2 = "wert";
var variable3 = "wert";

function funktion1 () {
    /* ... */
}
function funktion2 () {
    /* ... */
}
function funktion3 () {
    /* ... */
}
```

Diese Organisation bringt in der Regel mit sich, dass das Script nicht einfach konfigurierbar, anpassbar und erweiterbar ist. In den wenigsten Fällen sind diese Scripte »unaufdringlich«, sondern fördern die Vermischung von HTML, CSS und JavaScript. Sie enthalten einerseits selbst »hartkodierten«, das heißt fest eingebundenen HTML- und CSS-Code und erfordern andererseits große Änderungen im HTML-Dokument.

Manche Scripte sind durch Konfigurationsvariablen anpassbar, die vor den tatsächlichen Code gesetzt werden. Ein Seitenautor, der ein fremdes Script in seine Seite einbaut, kann auf diese Weise auch ohne Kenntnis des Scriptes dessen Verhalten ändern.

```
/* Konfigurationsvariablen */
var konfiguration1 = "anpassbar";
var konfiguration2 = "anpassbar";
var konfiguration3 = "anpassbar";

/* Der folgenden Code sollte unverändert bleiben: */
var variable1 = "wert";
var variable2 = "wert";
var variable3 = "wert";
function funktion1 () { /* ... */ }
function funktion2 () { /* ... */ }
function funktion3 () { /* ... */ }
```

Wird nun ein weiteres Script eingebunden, so ist die Wahrscheinlichkeit hoch, dass es ähnliche Namen für die Variablen und Funktionen verwendet. In diesem Fall kommt es zu unerwünschten Wechselwirkungen, wodurch die beteiligten Scripte nicht mehr ordnungsgemäß funktionieren. Viele Script-Autoren versehen daher alle Bezeichner mit einem Präfix (einer Vorsilbe), der die Zugehörigkeit zu einem bestimmten Script verdeutlicht:

```
/* Konfigurationsvariablen */
var präfix_konfiguration1 = "anpassbar";
var präfix_konfiguration2 = "anpassbar";
```

```

var präfix_konfiguration3 = "anpassbar";

/* Der folgenden Code sollte unverändert bleiben: */
var präfix_variable1 = "wert";
var präfix_variable2 = "wert";
var präfix_variable3 = "wert";
function präfix_funktion1 () { /* ... */ }
function präfix_funktion2 () { /* ... */ }
function präfix_funktion3 () { /* ... */ }

```

Damit sind zwar schon wichtige Grundlagen geschaffen, allerdings handelt es sich immer noch um eine Zahl von losen Objekten im **globalen Geltungsbereich (Scope)**.



Objektstrukturen mit **Object**

Sinnvoller ist es, alle Objekte – Variablen und Funktionen sind nichts anderes als Objekte – eines Scripts in einer echten JavaScript-Objektstruktur zu gruppieren. Im globalen Geltungsbereich taucht dann nur noch diese eine Objektstruktur auf, andere globale Variablen oder Funktionen werden nicht belegt. Das Script ist in der Objektstruktur in sich abgeschlossen. Damit sind unerwünschte Wechselwirkungen mit anderen Scripten ausgeschlossen, solange der Bezeichner der Objektstruktur eindeutig ist.

Allgemeines zu Objekten

Ein JavaScript-Objekt ist erst einmal nichts anderes als ein Container für weitere Daten. Ein Objekt ist eine Liste, in dem unter bestimmten Namen gewisse Unterobjekte (auch *Member* genannt) gespeichert sind. Aus anderen Programmiersprachen ist eine solche Datenstruktur als *Hash* oder *assoziativer Array* bekannt. In JavaScript sind alle vorgegebenen Objekte und Methoden in solchen verschachtelten Objektstrukturen organisiert, z.B. `window.document.body`.

Object-Objekte

In JavaScript gibt es den allgemeinen Objekttyp **Object**. Vom **Object**-Prototypen stammen alle anderen JavaScript-Objekte ab. Das heißt, jedes JavaScript-Objekt ist immer auch ein **Object**-Objekt. **Object** ist die allgemeine Grundlage, auf der die restlichen, spezifischeren Objekttypen aufbauen.

Für die Organisation von eigenen Scripten bieten sich solche unspezifischen **Object**-Objekte an. Denn sie sind nichts anderes als Container, in denen man unter einem Namen ein weiteres Objekt speichern kann. Über `new Object()` lässt sich der **Object**-Konstruktor aufrufen und ein **Object**-Objekt erzeugen:

 **Anzeigebeispiel: So sieht's aus**

```

var Container = new Object();
Container.eigenschaft = "wert";
Container.methode = function () {
    alert("Container-Eigenschaft: " + this.eigenschaft);
};
Container.methode();

```

Der Name **Container** ist hier selbstverständlich als Platzhalter gemeint. Sie sollten das **Object**-Objekt (im Folgenden kurz **Object** genannt) eindeutig und wiedererkennbar nach der Aufgabe bzw. dem Zweck ihres Scriptes benennen.

Über die gewohnte Schreibweise zum Ansprechen von Unterobjekten werden dem **Object** weitere Objekte angehängt. Im Beispiel werden dem **Object** zwei Objekte angehängt, ein String und eine Funktion. Die entstehende Verschachtelung könnte man so illustrieren:

- **Container** (Object)
 - **eigenschaft** (String)
 - **methode** (Function)

Da die Funktion **methode** ein Unterobjekt von **Container** ist, bezeichnet man sie als *Methode* dieses Objektes. Andere Unterobjekte, die nicht Funktionen sind, bezeichnet man als *Eigenschaften*.

Durch diese Zugehörigkeits-Beziehung bezieht sich das Schlüsselwort **this** innerhalb der Methode auf das Objekt, dem die Methode anhängt, im Beispiel **Container**. Ein Zugriff auf die Eigenschaft namens **eigenschaft** ist daher über **this.eigenschaft** möglich.

Auf dieselbe Weise können sich Methoden untereinander ansprechen und aufrufen. Zum Beispiel ließe sich dem **Object** eine zweite Methode hinzufügen, die die erste aufruft:

```
Container.zweiteMethode = function () {
    this.methode();
};
Container.zweiteMethode();
```

Wie ebenfalls aus den Beispielen ersichtlich wird, ist der Zugriff auf die Unterobjekte (Member) des **Objects** »von außen« nur über den Namen des **Objects** nach dem Schema **Objectname.Membername** möglich.



Object-Literale

JavaScript bietet für das Definieren von **Object**-Objekten eine Kurzschreibweise an, den sogenannten *Object-Literal*. Ein **Object**-Literal beginnt mit einer öffnenden geschweiften Klammer **{** und endet mit einer schließenden geschweiften Klammer **}**. Dazwischen befinden sich, durch Kommas getrennt, die Zuweisungen von Namen zu Objekten. Zwischen Name und Objekt wird ein Doppelpunkt notiert. Das Schema ist also: **{ name1 : objekt1, name2 : objekt2, ... nameN : objektN }**

Das obige Beispiel-**Object** lässt sich in der Literalschreibweise so umsetzen:

Anzeigebeispiel: So sieht's aus

```
var Container = {
    eigenschaft : "wert",
    methode : function () {
        alert("Container-Eigenschaft: " + this.eigenschaft);
    }
};
Container.methode();
```

Mittlerweile bedienen sich unzählige »unaufdringliche« Scripte dieser Schreibweise und sie hat sich zu einem Standard gemausert. Insbesondere *Christian Heilmann* hat sich für diese Schreibweise stark gemacht (**Object Literal – Warum neuere Skripte anders aussehen**, deutsche Übersetzung leider offline), seine **Scripte** sind gute Beispiele dafür, wie **Object**-Literale in der Praxis verwendet werden.



Object-Methoden in anderen Kontexten ausführen

Beim »unaufdringlichen« JavaScript ist es meist unerlässlich, dass im **Object** gespeicherte Methoden als Event-Handler dienen (siehe **Ereignisüberwachung mit JavaScript programmieren**). Dies wirft das Problem auf, dass solche Methoden außerhalb des **Object**-Kontextes ausgeführt werden, wenn das überwachte Ereignis eintritt.

Außerhalb des Kontextes bedeutet, dass **this** nicht mehr wie beschrieben auf das **Object** zeigt, sondern auf das Elementobjekt, dessen Handler ausgelöst wurde. (Siehe **die Bedeutung des this-Schlüsselwortes beim Event-Handling**.) In vielen Fällen aber ist im Event-Handler ein Zugriff auf beide Objekte gewünscht, auf das Elementobjekt sowie auf den **Object**-Container.

Folgendes Beispiel illustriert das Problem:

Anzeigebeispiel: So sieht's aus

```
var Container = {
    eigenschaft : "wert",
    methode : function () {
        // Funktioniert:
        alert(
            "methode wurde aufgerufen\n" +
            "Container-Eigenschaft: " + this.eigenschaft
        );
    },
    handler : function (eventobjekt) {
        if (!eventobjekt)
            eventobjekt = window.event;
        // Fehler: this verweist auf das Element, dem der Event-Handler anhängt
        alert(
```

```

        "handler wurde aufgerufen\n" +
        "Container-Eigenschaft: " + this.eigenschaft
    );
}
};
Container.methode();
document.getElementById("button").onclick = Container.handler;

```

Die Methode `handler` wird als Handler für das `click`-Ereignis bei einem Button definiert. Während der Zugriff auf das `Object` über `this` beim regulären Aufruf der Methode funktioniert, verweist `this` in diesem Fall auf das `window`-Objekt.

Dasselbe Problem tritt auf, wenn eine Methode eine andere Methode desselben `Objects` mit einer Verzögerung (`setTimeout`) oder als Intervall (`setInterval`) aufrufen will. `this` zeigt dann auf `window`, da die verzögert aufgerufene Methode im globalen Kontext aufgerufen wird:

☐ Anzeigebeispiel: So sieht's aus

```

var Container = {
    eigenschaft : "wert",
    methode : function () {
        // Funktioniert:
        alert(
            "methode wurde aufgerufen\n" +
            "Container-Eigenschaft: " + this.eigenschaft
        );
        window.setTimeout(this.verzögert, 500);
    },
    verzögert : function () {
        // Fehler: this verweist window
        alert(
            "verzögert wurde aufgerufen\n" +
            "Container-Eigenschaft: " + this.eigenschaft
        );
    }
};
Container.methode();

```

Lösung: `this` vermeiden

Eine mögliche Lösung ist, das `Object` immer explizit über dessen Namen anzusprechen anstatt über `this`.

`this` wird dann nur noch in Methoden verwendet, die als Event-Handler dienen. Denn `this` ist die einzige Möglichkeit, im Internet Explorer auf das Element zuzugreifen, dessen Handler das Ereignis ausgelöst hat. In Browsern, die dem DOM-Events-Standard folgen, gibt es dafür die Eigenschaft `currentTarget` des Event-Objektes.

☐ Anzeigebeispiel: So sieht's aus

```

var Container = {
    eigenschaft : "wert",
    methode : function () {
        alert(
            "methode wurde aufgerufen\n" +
            "Container-Eigenschaft: " + Container.eigenschaft
        );
    },
    handler : function (eventobjekt) {
        if (!eventobjekt)
            eventobjekt = window.event;
        alert(
            "handler wurde aufgerufen\n" +
            "Event-Objekt: " + eventobjekt + "\n" +
            "Element, das den Event behandelt: " + this + "\n" +
            "Container-Eigenschaft: " + Container.eigenschaft
        );
    }
};
Container.methode();

```

```
document.getElementById("button").onclick = Container.handler;
```

In diesem Beispiel wurde `this` durch `Container` ersetzt. `this` wird in der Methode `handler` verwendet, um auf das Elementobjekt zuzugreifen, bei dessen Handler vom Ereignis ausgelöst wurde.

Das folgende Beispiel zeigt, wie `this` bei der Benutzung von `setTimeout` vermieden werden kann:

Anzeigebeispiel: So sieht's aus

```
var Container = {
  eigenschaft : "wert",
  methode : function () {
    alert(
      "methode wurde aufgerufen\n" +
      "Container-Eigenschaft: " + Container.eigenschaft
    );
    window.setTimeout(Container.verzögert, 500);
  },
  verzögert : function () {
    alert(
      "verzögert wurde aufgerufen\n" +
      "Container-Eigenschaft: " + Container.eigenschaft
    );
  }
};
Container.methode();
```

Solange eine Methode nicht in anderen Kontexten ausgeführt wird, kann darin `this` verwendet werden, um das `Object` anzusprechen. Aus Gründen der Einheitlichkeit und Einfachheit wurde in den Beispielen immer `Container` verwendet.



Möglichkeiten der Datenspeicherung

Das definierte `Object`, das alle Variablen und Funktionen eines Scriptes kompakt speichert, muss dokumentweit eindeutig sein. Es kann keine weiteren gleichnamigen globalen Objekte geben. Das heißt, es ist nur eine Instanz des `Objects` möglich.

Bei »unaufdringlichem« JavaScript wird gewissen Elementen Interaktivität hinzugefügt. Beispielsweise kann allen Tabellen im Dokument mit der Klasse `sortierbar` automatisch eine Sortier-Funktionalität hinzugefügt werden. Wenn also mehrere Tabellen sortierbar sind, muss z.B. der jeweilige Sortierstatus irgendwo gespeichert werden. Dazu bieten sich verschiedene Möglichkeiten an:

- Das `Object` könnte dazu einen Array von `Objects` enthalten, in denen jeweils die Daten für eine Tabelle gespeichert werden.
- Alternativ können die Daten nicht am `Object`, sondern *im Dokument selbst* in Form von Attributen bzw. Unterobjekten der jeweiligen Elementobjekte gespeichert werden. In JavaScript können nämlich jedem Objekt beliebig Unterobjekte angehängt werden. So gehen viele Scripte vor, in den meisten Fällen ist dies auch der beste Weg, um mit einem dokumentweiten `Object` auszukommen.
- In manchen Fällen ist es sinnvoll, mehrere in ihrer Funktionalität ähnliche Objekte zu haben, die jeweils das Verhalten z.B. eines bestimmten Elementes im Dokument regeln und die jeweiligen Daten gesondert speichern. Eine `Object`-Struktur als Container eignet sich dafür nicht, denn sie kann nicht ohne Aufwand beliebig dupliziert werden. Für diesen Fall eignen sich *Eigene Objekte*, deren Grundlagen im folgenden Abschnitt diskutiert werden.



Eigene Objekte mittels Konstruktoren

Anstatt alle Eigenschaften und Funktionen an ein `Object` anzuhängen, kann man ein *eigenes Objekt* erstellen.

Aus anderen Programmiersprachen kennt man das Definieren von eigenen *Klassen*. In JavaScript gibt es strenggenommen keine Klassen, sondern nur **Konstruktor-Funktionen** (kurz: *Konstruktoren*). Der Name stammt vom englischen *construct* = erzeugen, konstruieren, bauen. Eine Konstruktor-Funktion ist demnach ein Erzeuger neuer Objekte.

Sie werden sich sicher fragen, wie die Syntax zum Notieren von Konstruktoren lautet. Ein Konstruktor ist jedoch keine eigene Sprachstruktur, sondern erst einmal eine ganz normale Funktion. Zu einem Konstruktor wird sie lediglich dadurch, dass sie mit dem Schlüsselwort `new` aufgerufen wird.

Wenn eine Funktion mit `new` aufgerufen wird, wird intern ein neues, leeres `Object` angelegt und die Funktion im Kontext dieses Objektes ausgeführt. Im Konstruktor können diesem neuen Objekt Eigenschaften und Methoden dann über `this` hinzugefügt werden.

Auch wenn das so entstehende Objekt der **Object**-Struktur ähnelt, können auf diese Weise unzählige gleiche Abkömmlinge, sogenannte *Instanzen* erzeugt werden.

☐ **Anzeigebeispiel: So sieht's aus**

```
// Konstruktorfunktion
function Konstruktor () {
  // Zugriff auf das neue Objekt über this,
  // Hinzufügen der Eigenschaften und Methoden
  this.eigenschaft = "wert";
  this.methode = function () {
    // In den Methoden wird über this auf das Objekt zugegriffen
    alert(
      "methode wurde aufgerufen\n" +
      "Instanz-Eigenschaft: " + this.eigenschaft
    );
  };
}
// Erzeuge Instanzen
var instanz1 = new Konstruktor();
instanz1.methode();
var instanz2 = new Konstruktor();
instanz2.methode();
// usw.
```

Indem der Konstruktor bestimmte Parameter erhält, können Instanzen mit unterschiedlichen Eigenschaften erzeugt werden. Sie können aber auch im Laufe der Benutzung unterschiedliche Werte bekommen. Der Zugriff »von außen« auf sogenannte *öffentliche Eigenschaften* erfolgt über das bekannte Schema **Instanzname.Membername**.

Die Bezeichnung *eigenes Objekt* ist unglücklich und missverständlich, schließlich haben wir mit dem **Object-Container** ebenfalls ein eigenes Objekt erzeugt. Andere Quellen verwenden den bekannten Begriff *Klasse* auch für JavaScript-Konstrukturen. Allerdings führt diese Bezeichnung nicht weniger in die Irre, da sich die objektorientierte Programmierung in JavaScript grundlegend von der klassenbasierter Sprachen unterscheidet.



Methoden eigener Objekte in anderen Kontexten ausführen

Will man nun eine Methode einer Instanz als Event-Handler nutzen oder sie verzögert aufrufen, tritt das besagte Phänomen auf: Die Methode wird außerhalb des Kontextes der Instanz ausgeführt und **this** zeigt nicht mehr auf die Instanz. Folgendes Kombinationsbeispiel veranschaulicht das Problem, das sowohl bei der Ereignisüberwachung als auch bei der Nutzung von **setTimeout** oder **setInterval** auftritt:

☐ **Anzeigebeispiel: So sieht's aus**

```
function Konstruktor () {
  this.eigenschaft = "wert";
  this.methode = function () {
    // Funktioniert:
    alert(
      "methode wurde aufgerufen\n" +
      "Instanz-Eigenschaft: " + this.eigenschaft
    );
    window.setTimeout(this.verzögert, 500);
  };
  this.verzögert = function () {
    // Fehler: this verweist window
    alert(
      "verzögert wurde aufgerufen\n" +
      "Instanz-Eigenschaft: " + this.eigenschaft
    );
  };
  this.handler = function (eventobjekt) {
    if (!eventobjekt)
      eventobjekt = window.event;
    // Fehler: this verweist auf das Element, dem der Event-Handler anhängt
    alert(
```



```

        "handler wurde aufgerufen\n" +
        "Event-Objekt: " + eventobjekt + "\n" +
        "Instanz-Eigenschaft: " + this.eigenschaft
    );
};
document.getElementById("button").onclick = this.handler;
}
var instanz = new Konstruktor();
instanz.methode();

```

Die Lösung dieses Problems ist kompliziert und führt uns auf eine weitere hochinteressante, aber auch schwer zu meisternde Eigenheit der JavaScript-Programmierung, die im Folgenden vorgestellt werden soll.



Einführung in Closures

Eine *Closure* ist allgemein gesagt eine Funktion, die in einer anderen Funktion notiert wird. Diese verschachtelte, innere Funktion hat Zugriff auf die Variablen des Geltungsbereiches (Scopes) der äußeren Funktion – und zwar über die Ausführung der äußeren Funktion hinaus.

Durch dieses Einschließen der Variablen kann man bestimmte Objekte in Funktionen verfügbar machen, die darin sonst nicht oder nur über Umwege verfügbar wären. Closures werden damit zu einem Allround-Werkzeug in der fortgeschrittenen JavaScript-Programmierung.

Dieses Beispiel erläutert die Variablen-Verfügbarkeit bei verschachtelten Funktionen:

Anzeigebeispiel: So sieht's aus

```

function äußerefunktion () {
    // Definiere eine lokale Variable
    var variable = "wert";
    // Lege eine Funktion als lokale Variable an
    var innerfunktion = function () {
        // Obwohl diese Funktion einen eigenen Scope mit sich bringt,
        // ist die Variable aus dem umgebenden Scope hier verfügbar:
        alert("Wert der Variablen aus der äußeren Funktion: " + variable);
    };
    // Führe die eben definierte Funktion aus
    innerfunktion();
}
äußerefunktion();

```

Das Beispiel zeigt, dass die innere Funktion Zugriff auf die Variablen der äußeren Funktion hat. Der entscheidende Punkt bei einer Closure ist jedoch ein anderer:

Normalerweise werden alle lokalen Variablen einer Funktion aus dem Speicher gelöscht, nachdem die Funktion beendet wurde. Eine Closure aber führt dazu, dass die Variablen der äußeren Funktion nach deren Ausführung nicht gelöscht werden, sondern im Speicher erhalten bleiben. Die Variablen stehen der inneren Funktion weiterhin über deren ursprüngliche Namen zur Verfügung. Die Variablen werden also *eingeschlossen* und konserviert – daher der Name »Closure«.

Auch lange nach dem Ablauf der äußeren Funktion hat die Closure immer noch Zugriff auf deren Variablen – vorausgesetzt, die Closure wird woanders gespeichert und kann dadurch zu einem späteren Zeitpunkt ausgeführt werden. (Im obigen Beispiel ist die innere Funktion nur eine lokale Variable, die zwar Zugriff auf die Variablen der äußeren Funktion hat, aber bei deren Beendigung selbst verfällt.)

Eine Möglichkeit, die innere Funktion zu speichern, ist das Registrieren als Event-Handler. Dabei wird das Funktionsobjekt in einer Eigenschaft (hier `onclick`) eines Elementobjektes gespeichert und bleibt damit über die Ausführung der äußeren Funktion hinweg erhalten:

Anzeigebeispiel: So sieht's aus

```

function äußerefunktion () {
    var variable = "wert";
    // Lege die Closure-Funktion als lokale Variable an
    var closure = function () {
        alert("Wert der Variablen aus der äußeren Funktion: " + variable);
    };
}

```



```

};
// Speichere die Closure-Funktion als Event-Handler
document.getElementById("button").onclick = closure;
}
äußerefunktion();

```

Bei einem Klick auf das Dokument wird die Closure als Event-Handler ausgeführt. `äußerefunktion` wird schon längst nicht mehr ausgeführt, aber `variable` wurde in die Closure eingeschlossen.

Zusammengefasst haben wir folgendes Schema zur Erzeugung einer Closure:

1. Beginn der Ausführung der äußeren Funktion
2. Lokale Variablen werden definiert
3. Innere Funktion wird definiert
4. Innere Funktion wird außerhalb gespeichert, sodass sie erhalten bleibt
5. Ende der Ausführung der äußeren Funktion
6. Unbestimmte Zeit später: Innere Funktion (Closure-Funktion) wird ausgeführt



Anwendung von Closures

Wie helfen uns Closures nun beim ursprünglichen Problem weiter? Zunächst einmal ist festzustellen, dass beim Erzeugen von eigenen Objekten mit verschachtelte Funktionen gearbeitet wird: Der Konstruktor stellt die äußere Funktion dar und die Methoden, die der Instanz im Konstruktor zugewiesen werden, sind innere Funktionen.

Aus diesem Grund wirken die Methoden als Closures, die die Variablen des Konstruktors einschließen. Im [↑ ursprünglichen Beispiel](#) handelt es sich um die Methoden `methode`, `handler` und `verzögert`.

Im Konstruktor kann man daher eine lokale Variable als Referenz auf das Instanzobjekt `this` anlegen. (Solche lokalen Variablen im Konstruktor werden *private Eigenschaften* genannt.) Alle Methoden, die der Instanz im Konstruktor hinzugefügt werden, schließen diese Variable ein – sie ist in diesen Methoden auch dann noch verfügbar, wenn sie als Event-Handler oder mit Verzögerung in einem anderen Kontext ausgeführt werden. Folgendes Beispiel demonstriert beide Fälle:

☐ [Anzeigebeispiel: So sieht's aus](#)

```

function Konstruktor () {
    // Äußere Funktion

    // Referenz auf das Instanz-Objekt anlegen
    var thisObject = this;

    this.eigenschaft = "wert";

    this.methode = function () {
        // wirkt als Closure und schließt thisObject ein

        alert(
            "methode wurde aufgerufen\n" +
            "Instanz-Eigenschaft: " + thisObject.eigenschaft
        );
        window.setTimeout(thisObject.verzögert, 500);
    };

    this.verzögert = function () {
        // wirkt als Closure und schließt thisObject ein

        alert(
            "verzögert wurde aufgerufen\n" +
            "Instanz-Eigenschaft: " + thisObject.eigenschaft
        );
    };

    this.handler = function (eventobjekt) {
        // wirkt als Closure und schließt thisObject ein

        if (!eventobjekt)
            eventobjekt = window.event;
    };
}

```

```

        alert(
            "handler wurde aufgerufen\n" +
            "Event-Objekt: " + eventobjekt + "\n" +
            "Element, das den Event behandelt: " + this + "\n" +
            "Instanz-Eigenschaft: " + thisObject.eigenschaft
        );
    };

    // Hier im Konstruktor sind this und thisObject noch identisch
    document.getElementById("button").onclick = this.handler;
}

var instanz = new Konstruktor();
instanz.methode();

```

Wichtig ist hier die Unterscheidung zwischen `this` und `thisObject`. `this` zeigt in den drei Methoden `methode`, `verzögert` und `handler` auf drei unterschiedliche Objekte. `thisObject` hingegen ist die eingeschlossene Variable, die auf das Instanzobjekt zeigt – und zwar in allen drei Methoden.



Closures zur Kapselung bei **Object** und eigenen Objekten

Bei eigenen Objekten lässt sich festlegen, welche Unterobjekte »von außen« eingesehen und geändert werden können. In der Fachsprache wird zwischen *öffentlichen* und *privaten* Membern unterschieden. (*Member* ist hier ein Sammelbegriff für Eigenschaften und Methoden.) Nach außen sollte eine Instanz eine wohlüberlegte und gut dokumentierte Programmierschnittstelle (*API*) anbieten, in der intern verwendete Variablen und Funktionen nicht vorkommen.

Dieses wichtige Konzept der *Kapselung* in der objektorientierten Programmierung soll hier nur kurz angeschnitten werden. Einen vollständigeren Einstieg bieten die Quellen in den [Literaturhinweisen](#).

Bei den bisher vorgestellten [Object-Containern](#) sind alle Unterobjekte öffentlich. Über einen Umweg sind auch private Member bei **Objects** möglich. Das Mittel dazu sind wieder Closures. Das Konzept lautet folgendermaßen:

- Man definiert eine Funktion, in deren lokalen Geltungsbereich man die gewünschten privaten Member definiert.
- Innerhalb dieser Funktion notiert man wie gehabt das **Object**-Literal. Die Methoden des **Objects** haben Zugriff auf die Variablen der äußeren Funktion und schließen diese ein (Closures).
- Die Funktion gibt das **Object** als Rückgabewert zurück.
- Das zurückgegebene **Object** wird wie gewohnt in einer globalen Variable gespeichert, über die es ansprechbar sein soll.

Ausführliche Schreibweise

In der ausführlichen Schreibweise könnte die Umsetzung so aussehen:

[Anzeigebeispiel: So sieht's aus](#)

```

function erzeugeContainer () {

    // Notiere private Eigenschaften und Methoden

    var privateEigenschaft = "privat";

    function privateMethode () {
        window.alert("privateMethode wurde aufgerufen");
        window.alert("Private Eigenschaft: " + privateEigenschaft);
        window.alert("Öffentliche Eigenschaft: " + Container.öffentlicheEigenschaft);
    }

    // Erzeuge Object mit den öffentlichen Eigenschaften und Methoden
    var öffentlicheSchnittstelle = {

        öffentlicheEigenschaft : "öffentlich",

        öffentlicheMethode1 : function () {
            // öffentlicheMethode1 wirkt als Closure und
            // schließt privateEigenschaft und privateMethode ein

```

```

        window.alert("öffentlicheMethodel wurde aufgerufen");

        // Zugriff auf private Eigenschaften und Methoden
        window.alert("Private Eigenschaft: " + privateEigenschaft);
        privateMethode();

        // Zugriff auf öffentliche Eigenschaften und Methoden
        window.alert("Öffentliche Eigenschaft: " + Container.öffentlicheEigenschaft);
        Container.öffentlicheMethode2();

    },

    öffentlicheMethode2 : function () {
        // öffentlicheMethode2 wirkt ebenfalls als Closure und
        // schließt die privaten Eigenschaften und Methoden ein

        window.alert("öffentlicheMethode2 wurde aufgerufen");

    }

};

return öffentlicheSchnittstelle;
}
var Container = erzeugeContainer();
Container.öffentlicheMethodel();

// Ergibt undefined, weil der Zugriff durch die Kapselung unmöglich wird:
window.alert("Container.privateMethode von außerhalb: " + Container.privateMethode);

```

Das **Object** hat schließlich zwei öffentliche Methoden, die ihrerseits Lese- und Schreibzugriff auf die privaten Eigenschaften und Methoden haben. Von außen sind diese privaten Member aber nicht sichtbar.

Kurzschreibweise

Die obige Schreibweise ist absichtlich lang und ausführlich, um das Schema zu vereinheitlichen. In der Praxisanwendung hat sie einen Nachteil: Es muss eine globale Funktion **erzeugeContainer** geben, die aufgerufen wird. Diese Funktion wollen wir nur einmal ausführen, um das Container-Objekt zu initialisieren. Daher ist es unnötig, eine globale Funktion zu notieren, die über diese Initialisierung hinaus Bestand hat und im globalen Geltungsbereich einen Bezeichner belegt. Es reicht aus, eine *anonyme (namenlose) Funktion* zu notieren. Dazu nutzen wir einen sogenannten *Funktionsausdruck* (engl. *Function Expression*).

Wir haben Funktionsausdrücke schon die ganze Zeit benutzt, wenn wir **this.methode = function (...) { ... };** notiert haben. Funktionsausdrücke bilden den Gegenpart zur gewohnten Schreibweise von Funktionen, der sogenannten *Funktionsdeklaration* (engl. *Function Declaration*).

Zudem kann die Variable **öffentlicheSchnittstelle** in der Funktion eingespart werden, indem direkt hinter **return** das **Object**-Literal notiert wird. Die Kurzschreibweise lautet des obigen Beispiels lautet demnach:

Anzeigebeispiel: So sieht's aus

```

var Container = (function () {
    // Definiere eine Funktion mit einem Funktionsausdruck,
    // durch runde Klammern umschlossen

    var privateEigenschaft = "privat";
    function privateMethode () {
        window.alert("privateMethode wurde aufgerufen");
        window.alert("Private Eigenschaft: " + privateEigenschaft);
        window.alert("Öffentliche Eigenschaft: " + Container.öffentlicheEigenschaft);
    }

    // Direkt das Object mit der öffentlichen Schnittstelle zurückgeben
    return {
        öffentlicheEigenschaft : "öffentlich",
        öffentlicheMethodel : function () {
            window.alert("öffentlicheMethodel wurde aufgerufen");
            window.alert("Private Eigenschaft: " + privateEigenschaft);
            privateMethode();
            window.alert("Öffentliche Eigenschaft: " + Container.öffentlicheEigenschaft);
            Container.öffentlicheMethode2();
        }
    };
})();

```

```

    },
    öffentlicheMethode2 : function () {
        window.alert("öffentlicheMethode2 wurde aufgerufen");
    }
};
})();
// Ende des eingeklammerten Funktionsausdrucks, dahinter
// direkt () zum Aufruf der soeben definierten Funktion

Container.öffentlicheMethode1();

// Ergibt undefined, weil von außen nicht sichtbar:
window.alert("Container.privateMethode von außerhalb: " + Container.privateMethode);

```

Diese Schreibweise mag auf den ersten Blick unverständlich scheinen, deshalb noch einmal aufgedrösel:

1. Definiere einen Funktionsausdruck: `function (...) {...}`
2. Dieser alleine ergibt ein Funktionsobjekt. Um den Ausdruck werden Klammern notiert, sodass die Funktion gleich direkt aufgerufen werden kann: `(function (...) {...})`
3. Das Funktionsobjekt wird nun wie gewohnt mit den Klammern dahinter samt Parameterliste ausgeführt: `(function (...) {...}) (...)`
4. Der Rückgabewert der ausgeführten Funktion – in diesem Fall ein `Object` – wird wie üblich gespeichert: `var Container = (function (...) {...}) (...);`

Als Resultat haben wir eine namenlose Funktion notiert, die direkt ausgeführt wird. Ihr Rückgabewert wird gespeichert, die Funktion selbst geht aber verloren – denn sie hat ihren Zweck erfüllt und wir brauchen sie im weiteren Programmverlauf nicht mehr.



Alternativlösungen zur Kontext-Problematik: `bind()` und `bindAsEventListener()`

Für die hier beispielhaft gelösten Probleme gibt es viele andere Lösungsansätze, von einfach bis kompliziert. Die beschriebenen Lösungen sind bewusst einfach gehalten, da sie sich an Einsteiger richten – in diesem Artikel sollen lediglich gewisse ausgewählte Strukturen vorgestellt sowie deren praktische Eigenheiten diskutiert werden. Andere, mächtigere Strukturen sowie allgemeine Objektorientierte Programmierung sind nicht der direkter Gegenstand des Artikels. Die verlinkten Quellen in den [↓ Literaturhinweisen](#) beschreiben fortgeschrittene Herangehensweisen sowie grundlegende Einführungen.

Es soll allerdings auf eine verbreitete Technik hingewiesen werden, mit der sich Objektmethoden einfach in bestimmten Kontexten ausführen lassen: Das JavaScript-Framework [Prototype](#) bietet dazu zwei Funktionen namens `bind` und `bindAsEventListener` an. Beide werden über *prototypische Erweiterung* allen Funktionsobjekten hinzugefügt – daraufhin besitzt eine beliebige Funktion namens `funktion` die Methoden `funktion.bind(...)` und `funktion.bindAsEventListener(...)`. Auf die vielfältigen Möglichkeiten der prototypischen Erweiterung, die einen der Grundpfeiler der Objektorientierten Programmierung in JavaScript darstellt, soll an dieser Stelle nicht näher eingegangen werden.

Diese Helfermethoden geben dynamisch erzeugte Funktionsobjekte zurück, die die eigentlichen Funktionen umhüllen. In diesen *Wrapper-Funktionen* werden die vordefinierten JavaScript-Funktionen [apply](#) und [call](#) verwendet, um die eigentlichen Funktionen im Kontext des angegebenen Objektes auszuführen. Die Wrapper-Funktion wirkt als Closures, wodurch ihr die benötigten Objekte zur Verfügung stehen. (Anmerkung: `apply` und `call` werden in SELFHTML 8.1.2 noch nicht dokumentiert.)

Kommentierter ausführlicher Quellcode

Die Funktionen `bind` und `bindAsEventListener` sehen ausführlich und kommentiert so aus:

```

// Erweitere alle Funktionsobjekte um eine Methode »bind«
// über die prototype-Eigenschaft des Function-Konstruktors.
Function.prototype.bind = function () {

    // Speichere die gegenwärtige Funktion in »method«.
    var method = this;

    // Die Funktion nimmt eine beliebige Anzahl von Parametern entgegen,
    // auf die über den »arguments«-Pseudoarray zugegriffen wird.
    // Wandle »arguments« mit einer Helferfunktion in einen echten Array um.
    var args = $A(arguments);

    // Entnehme dem Array den ersten Parameter. Das ist das Objekt, in

```

```
// dessen Kontext die Funktion ausgeführt werden soll.
// »args« enthält nun die restlichen Parameter.
var object = args.shift();

// Notiere einen Funktionsausdruck, der als Closure wirkt.
var wrapper = function () {

    // Die Closure schließt »method«, »object« und »args« ein.
    // Rufe die Funktion im Kontext des Objektes »object« auf,
    // reiche dabei die restlichen Parameter durch und
    // gib den Rückgabewert der Funktion zurück.
    return method.apply(object, args);

};

// Gib die Wrapper-Funktion zurück.
return wrapper;

};

// Erweitere alle Funktionsobjekte um eine Methode »bindAsEventListener«
// über die prototype-Eigenschaft des Function-Konstruktors.
Function.prototype.bindAsEventListener = function (object) {

    // Die Funktion nimmt einen Parameter entgegen, der das
    // Objekt darstellt, in dessen Kontext die gewünschte Funktion
    // ausgeführt werden soll.

    // Speichere die gegenwärtige Funktion in »method«.
    var method = this;

    // Notiere einen Funktionsausdruck, der als Closure wirkt.
    var wrapper = function (event) {
        // Die Closure schließt »method« und »object« ein.

        // Vereinheitliche den Zugriff auf das Event-Objekt.
        // Dieses wird der Handler-Funktion entweder als Parameter
        // übergeben (hier »event«) oder steht im Internet Explorer
        // unter »window.event« zur Verfügung.
        var eventObject = event || window.event;

        // Rufe die Methode im Kontext des Objektes »object« auf und
        // reiche das Event-Objekt durch.
        return method.call(object, eventObject);

    };

    // Gib die Wrapper-Funktion zurück.
    return wrapper;

};
```

Kurzschreibweise

Ohne Kommentare und Variablen, die bloß der Lesbarkeit dienen, sehen die beiden Funktionen `bind` und `bindAsEventListener` wie folgt aus:

```
Function.prototype.bind = function () {
    var method = this, args = $A(arguments), object = args.shift();
    return function () {
        return method.apply(object, args);
    };
};

Function.prototype.bindAsEventListener = function (object) {
    var method = this;
    return function (event) {
        return method.call(object, event || window.event);
    }
};
```

Der Code für die verwendete Helferfunktion `$A`, die den `arguments`-Pseudoarray in einen echten JavaScript-Array umwandelt, lautet:

```
function $A (iterable) {
    return Array.prototype.slice.apply(iterable);
}
```

Anwendung

Die `bind`-Methode findet Verwendung bei Timeouts und Intervallen, `bindAsEventListener` bei Event-Handlern. Der folgenden Code zeigt, wie sich das obige [↑ Kombinationsbeispiel](#) mithilfe von `bind` und `bindAsEventListener` umsetzen lässt.

Anzeigebeispiel: So sieht's aus

```
function Konstruktor () {
    this.eigenschaft = "wert";
    this.methode = function () {
        window.setTimeout(this.verzögert.bind(this), 500);
    };
    this.verzögert = function () {
        alert(
            "verzögert wurde aufgerufen\n" +
            "Instanz-Eigenschaft: " + this.eigenschaft
        );
    };
    this.handler = function (eventobjekt) {
        alert(
            "handler wurde aufgerufen\n" +
            "Event-Objekt: " + eventobjekt + "\n" +
            "Element, das den Event behandelt: " + this.button + "\n" +
            "Instanz-Eigenschaft: " + this.eigenschaft
        );
    };
    this.button = document.getElementById("button");
    this.button.onclick = this.handler.bindAsEventListener(this);
}
var instanz = new Konstruktor();
instanz.methode();
```

Hier mag zunächst die seltsame Schreibweise `this.verzögert.bind(this)` und `this.handler.bindAsEventHandler(this)` irritieren. Diese Aufrufe hüllen `verzögert` und `handler` in Closures, sie werden daraufhin im Kontext der Instanz ausgeführt.

Der Unterschied gegenüber der vorherigen, einfacheren [↑ Closures-Methode](#) mag zunächst nicht groß scheinen. Allerdings bringen `bind` und `bindAsEventListener` eine etwas andere Arbeitsmethode mit sich und sind zugleich vielseitiger. Mittlerweile haben diese Methoden weite Verbreitung auch außerhalb des Prototype-Frameworks gefunden.

Fallstricke von `bindAsEventListener`



`this` zeigt in einer Event-Handler-Funktion üblicherweise auf das Elementobjekt, bei dem der Handler registriert wurde und gerade ausgeführt wird. Bei der Benutzung von `bindAsEventListener` zeigt `this` stattdessen auf das auf das Instanzobjekt. Damit der Zugriff auf beide Objekte möglich ist, kann das Element beim Registrieren des Event-Handlers in einer Eigenschaft der Instanz gespeichert werden, im Beispiel `this.button`.

In manchen Fällen kann man dieses Problem nicht so einfach umgehen. Oftmals ist eine Unterscheidung zwischen dem Element nötig, das bei dem das Ereignis ursprünglich passierte, und dem Element, dessen Handler das Ereignis ausgelöst hat. Der Hintergrund ist folgender: Ein Event steigt im Elementenbaum auf und löst die Handler seiner Eltern-Elementen aus. Dieser Effekt nennt sich *Bubbling*. Das Ursprungselement ist daher nicht immer identisch mit dem Element, bei dem ein Handler für das jeweilige Ereignis angestoßen wurde.

In standardkonformen Browsern ist eine Unterscheidung zwischen diesen Elementen über die Eigenschaften `target` und `currentTarget` des Event-Objektes problemlos möglich. Der Internet Explorer hingegen erlaubt lediglich den Zugriff auf das Ursprungselement des Ereignisses über die Eigenschaft `srcElement`. Das Element, dessen Handler gerade ausgeführt wird, ist standardmäßig über `this` ansprechbar. Wenn `this` nun durch den Einsatz von `bindAsEventHandler` auf ein anderes Objekt zeigt, dann ist der Zugriff auf das besagte Element nicht mehr browserübergreifend möglich – zumindest nicht mit der oben genannten unmodifizierten Variante von `bindAsEventHandler`.


Modularisierung und Namensräume

Mittlerweile sind ganze JavaScript-Bibliotheken und -Frameworks entstanden, bestehend aus verschiedenen Modulen und Unterschriften. Bei zunehmender Komplexität ist es nicht mehr praktikabel, dass eine Ansammlung von aufeinander aufbauenden Scripten aus einer losen Ansammlung von **Objects** oder Konstruktoren bestehen.

Man geht daher dazu über, verwandte und zusammenhängende **Objects** und Konstruktoren in weitere **Object**-Container einzuordnen und zu verschachteln. Auf diese Weise entstehen mehrdimensionale Objektstrukturen, oft *Module* oder *Pakete* genannt. Einzelne Methoden werden dann über eine Kette von verschachtelten Objekten angesprochen, zum Beispiel `YAHOO.util.Dom.methode()` bei der  [Yahoo! User Interface Library](#) oder `dojo.dom.methode()` bei  [Dojo](#).

Wie man an diesen Beispielen sieht, werden die Scripte nicht nur nach Funktionalität, sondern auch nach Zugehörigkeit zur Bibliothek geordnet. Eine Bibliothek besteht damit aus einem riesigen **Object**, das viele Unterobjekte enthält. Diese Ordnung nach Herkunft wird *Namensraum* (*Namespace*) genannt, in den obigen Beispielen **YAHOO** und **dojo**.

Bei kleineren zusammenhängenden Scripten lohnt sich ein eigener Namensraum nicht, sobald aber eine größere modularisierte Bibliothek entwickelt wird, bringen Namensräume Ordnung in die Scripte und sorgen dafür, dass sie nicht mit anderen kollidieren können.

Praktisch werden Namensräume über ein **Object** gelöst, das zunächst mit einem leeren  **Object-Literal** erzeugt wird. Danach können dem **Object** Member hinzugefügt werden:

```
var Namensraum = {};  
Namensraum.Container = {  
    ...  
};  
Namensraum.Konstruktor = function (...) {  
    ...  
};  
var instanz = Namensraum.Konstruktor(...);
```





Ausblick auf JavaScript-Frameworks

Wir haben einige grundlegende formale Aspekte der Programmierung von »unaufdringlichem« JavaScript betrachtet. Diese bilden ein zuverlässiges Fundament für eine umfangreiche JavaScript-Anwendung.

Außen vor gelassen haben wir Probleme und Aufgaben, die uns immer wieder in der Praxis des *DOM Scripting* begegnen. Dies sind vor allem:

- Effizientes Ansprechen von Elementen im Dokument z.B. über CSS-artige Selektoren
- Daran anschließend: Durchlaufen und Durchsuchen von Elementknotenlisten
- Zuverlässiges Registrieren und Entfernen von beliebigen Event-Handlern sowie die Kontrolle des Event-Flusses
- Browserübergreifender Zugriff auf Event-Eigenschaften wie die Mausposition und Tastendrücke
- Aufrufen von Initialisierungs-Funktionen, sobald der DOM-Knotenbaum verfügbar ist, um dem Dokument die gewünschte Interaktivität hinzuzufügen
- Einfaches Ändern des DOM-Knotenbaumes, Hinzufügen von neuen Elementen
- Ändern der CSS-Eigenschaften von Elementen, Hinzufügen und Entfernen von Klassen, Animationen und Effekte
- Browserübergreifender Zugriff auf Eigenschaften des sogenannten Viewports, der Position sowie der Ausmaße eines Elements
- Kommunikation mit dem Webserver, um Daten zu übertragen oder nachzuladen, ohne das Dokument zu wechseln ([XMLHttpRequest](#) bzw. *Ajax*)

Mittlerweile werden mehrere JavaScript-Frameworks entwickelt, um diese grundlegenden Aufgaben von DOM Scripting zu lösen. Ziel ist es, dass der JavaScript-Programmierer all diese Aufgaben nicht immer wieder von Hand lösen muss. Anstatt direkt mit dem DOM zu programmieren, führen diese Frameworks zahlreiche Objekte und Methoden als Abstraktionsschicht ein. Diese sind einfacher und intuitiver zu bedienen und nehmen dem Webautor einen Großteil der Arbeit ab.

Trotzdem werden Frameworks wie  [jQuery](#),  [Prototype](#) sowie die bereits genannten Dojo und Yahoo UI kritisch betrachtet. Sie legen einen einheitlichen Abstraktionslayer über die Browsereigenheiten, verbergen die tatsächlichen internen Vorgänge und geben vor, jedem einen Einstieg in die schwierige Materie des DOM Scripting zu ermöglichen.

Dabei ist es in vielen Fällen unverzichtbar, die interne Arbeitsweise zu kennen. Hier gilt: Wer die Aufgaben schon einmal »zu Fuß« gelöst hat und die Lösungsansätze kennt, steht nicht im Regen, wenn die Abstraktion in der Praxis nicht mehr greifen sollte.

Die meisten Helferscripte, Bibliotheken und Frameworks bedienen sich den vorgestellten Methoden zur Organisation. Die Kenntnis dieser Methoden ist daher nicht nur für das Schreiben von eigenen Scripten hilfreich, sondern auch für die Benutzung und das

Verständnis von fremden Scripten.



Literaturhinweise

Einführung in Unobtrusive JavaScript und DOM Scripting

- [Der sinnvolle Einsatz von JavaScript](#), SELFHTML Weblog
- [Unobtrusive JavaScript](#), Christian Heilmann
- Deutsche Übersetzung: [Barrierefreies JavaScript](#), Christian Heilmann
- [From DHTML to DOM scripting – an example of how to replace outdated JavaScript techniques](#), Christian Heilmann
- [The Importance of Maintainable JavaScript](#), Christian Heilmann

Programmiertechniken für strukturierte und wartbare Skripte

- [Show love to the object literal](#), Christian Heilmann
- Deutsche Übersetzung leider offline: [Object Literal - Warum neuere Skripte anders aussehen](#), Christian Heilmann
- [Objectifying JavaScript](#), Jonathan Snook
- [Namespacing your JavaScript](#), Dustin Diaz
- [Scope in JavaScript](#), Mike West

Closures

- [JavaScript Closures for Dummies](#), Morris Johns
- [Javascript Closures](#), Richard Cornford (umfangreich und technisch detailreich, aber für Einsteiger schwer verständlich)

Objektorientierte Programmierung, speziell Kapselung

- [Eigene Objekte definieren](#), SELFHTML-Dokumentation
- [Object Oriented Programming in JavaScript](#), Mike Koss
- [OOP mit JavaScript](#), Peter Kropff
- [JavaScript: Eigene Objekte definieren, Kapselung, Vererbung über die prototype-Eigenschaft](#), Wikipedia
- [Core JavaScript 1.5 Guide: Details of the Object Model](#)
- [Private Members in JavaScript](#), Douglas Crockford

JavaScript-Bibliotheken und -Frameworks

- [JavaScript-Bibliotheken und die jüngere JavaScript-Geschichte](#), Mathias Schäfer
- [Sieben Thesen zum gegenwärtigen JavaScript-Gebrauch](#), Mathias Schäfer
- [The JavaScript Library World Cup](#), Dan Webb
- [Notes on JavaScript Libraries](#), Simon Willison
- [Again JavaScript libraries](#), Peter-Paul Koch (Links und Zusammenfassung der Diskussion)
- [Keep JavaScript Simple](#), Peter-Paul Koch



[SELFHTML aktuell](#) [Artikel](#) [JavaScript](#)

© 2007 [Impressum](#), für diese Seite: zapperlott@gmail.com