JavaScript: Grundlagen zur Ereignisverarbeitung

Ereignisbasierte Skripte

Der Abschnitt über die Verarbeitung von Skripten hat Ihnen gezeigt, dass der Browser Skripte üblicherweise in dem Moment ausführt, in dem er den Code eines HTML-Dokuments herunterlädt, parst und auf ein **script**-Element trifft.

Der Schicksal von JavaScript ist aber nicht, bloß in diesem kurzen Moment des Ladens des HTML-Dokuments ausgeführt zu werden und dann für immer zur Ruhe zu kommen. Die meisten JavaScripte sollen Interaktivität bieten. Der Schlüssel dazu ist, das haben wir bereits in den Grundkonzepten kennengelernt, die **Überwachung und Behandlung von Ereignissen** (auch **Event-Handling** genannt).

Moderne Skripte durchlaufen deshalb verschiedene Phasen:

Phase Eins: Das Dokument wird empfangen und geparst

Dabei wird das JavaScript erstmals ausgeführt. Objekte und Funktionen werden dabei definiert, sodass sie für die spätere Nutzung zur Verfügung stehen. – Nicht alle notierten Funktionen werden dabei bereits aufgerufen. – Zu diesem Zeitpunkt hat das Script noch keinen vollständigen Zugriff auf das Dokument.

Phase Zwei: Das Dokument ist fertig geladen

Der vollständige Zugriff auf das Dokument über das DOM ist erst jetzt möglich. Nun wird ein Teil des Scripts aktiv, um dem bisher statischen Dokument JavaScript-Interaktivität hinzuzufügen: Das Script spricht vorhandene Elementknoten an und fügt ihnen sogenannte **Event-Handler** hinzu. Das Script kann aber auch den Inhalt oder die Darstellung von bestehenden Elementen verändern und dem Dokument neue Elemente hinzufügen (auch **DOM-Manipulation** genannt).

Phase Drei: Der Anwender bedient das Dokument und das Script reagiert darauf

Wenn die überwachten Ereignisse an den entsprechenden Elementen im Dokument passieren, so werden gewisse andere Teiles des Scripts aktiv, denn die entsprechenden **Handler-Funktionen** werden ausgeführt.

Resultierende Script-Struktur

Dieser chronologische Ablauf gibt die Struktur der meisten Scripte vor:

- Im Code werden vor allem mehrere Funktionen definiert, die später als *Handler* Ereignisse verarbeiten werden.
- Es gibt mindestens eine Hauptfunktion, die ausgeführt wird, wenn der Browser das Dokument fertig geladen hat, sodass Scripte darauf zugreifen können.

Das erste und enorm wichtige Ereignis, mit dem wir uns beschäftigen müssen, ist daher das **load-Ereignis**. Es passiert aus JavaScript-Sicht im Fenster beim **window-Objekt**. Wenn dieses Ereignis eintritt, wird die zweite Phase aktiv. Dem JavaScript steht der gesamte **DOM-Baum** zur Verfügung, einzelne Elemente werden angesprochen und es werden Event-Handler registriert. Diese elementar wichtige Vorgehensweise bei der JavaScript-Programmierung wird uns nun beschäftigen.

Traditionelles Event-Handling

Die Anweisung, die die Überwachung eines Ereignisses an einem Element startet, nennt man das *Registrieren* von **Event-Handlern**. Im Folgenden wird es um die einfachste und älteste Methode gehen, um Event-Handler zu registrieren.

Die typischen Bestandteile der Ereignis-Überwachung sind:

1. ein Elementobjekt
2. der Ereignistyp (z.B. **click**)
3. eine Handler-Funktion. Diese drei Bestandteile finden wir in dem Aufbau der JavaScript-Anweisung wieder. Das allgemeine Schema lautet allgemein:

```
element.onevent = handlerfunktion;
```

- **element** steht für ein JavaScript-Objekt aus dem Dokument, üblicherweise ein Elementknoten. Es kommen auch besondere Objekte wie ***window*** und ***document*** in Frage.

- **onevent** ist eine Objekteigenschaft, die mit der Vorsilbe ***on*** beginnt, auf die der Ereignistyp folgt. ***on*** ist die englische Präposition für ***bei***. Zum Beispiel ***onclick*** bedeutet so viel wie ***beim Klicken***.

- **handlerfunktion** ist der Name einer Funktion. Genauer gesagt steht an dieser Stelle ein beliebiger Ausdruck, der ein Funktionsobjekt ergibt: JavaScript wird diesen Ausdruck auflösen und das Ergebnis als Handler-Funktion verwenden.

Insgesamt hat die Anweisung die Form:

"Führe bei diesem Element beim Eintreten dieses Ereignisses diese Funktion aus."

Der obige Pseudocode soll nur das allgemeine Schema illustrieren. Es gibt natürlich kein Ereignis namens *event*, und *onevent* ist lediglich ein Platzhalter für alle möglichen Eigenschaften, darunter **onclick**, **onmouseover**, **onkeypress** und so weiter.

Betrachten wir ein konkretes Beispiel. Wir wollen nach dem erfolgreichen Laden des Dokuments eine JavaScript-Funktion ausführen. Dazu haben wir bereits das **load**-Ereignis kennengelernt, dass beim **window**-Objekt passiert. Angenommen, wir haben eine Funktion namens **start** definiert, die als Event-Handler dienen wird:

```
function start () {
```

```
    window.alert("Dokument erfolgreich geladen! Wir können nun
    über das DOM darauf zugreifen.");
}
```

Gemäß dem obigen Schema starten wir folgendermaßen das Event-Handling:

```
window.onload = start;
```

Und schon wird die gezeigte Funktion beim erfolgreichen Laden des Dokuments ausgeführt.

Sie werden sich sicher fragen, wie Ereignis-Verarbeitung auf JavaScript-Ebene funktioniert. Dazu schauen wir uns den Aufbau der besagten Anweisungen an:

Wir haben dort eine einfache Wertzuweisung (erkennbar durch das **=**), die einer Objekteigenschaft (**window.onload** auf der linken Seite) einen Wert (starte auf der rechten Seite) zuweist.

Nach dieser Zuweisung ist die Funktion in der Objekteigenschaft gespeichert. Dies funktioniert, weil Funktionen in JavaScripte auch nur Objekte sind, auf die beliebig viele Variablen und Eigenschaften verweisen können.

Passiert nun ein Ereignis am Objekt **window**, sucht der JavaScript-Interpreter nach einer Objekteigenschaft, die den Namen **on** gefolgt vom Ereignistyp trägt (im Beispiel **onload**). Wenn diese Eigenschaft eine Funktion beinhaltet, führt er diese aus. Das ist erst einmal alles – aber enorm wichtig zum Verständnis des Event-Handlings.

Wie Sie später erfahren werden, ist die oben vorgestellte Methode im Grunde überholt. Dieses *traditionelle* Event-Handling ist aber immer noch der Ausgangspunkt jeder JavaScript-Programmierung. Sie sollten sich dieses Schema und dessen Funktionsweise genau einprägen.

Beispiel für traditionelles Event-Handling

Mit dem Wissen über Ereignis-Überwachung und das **load**-Ereignis können wir ein Dokument mitsamt eines Scriptes schreiben, das die beschriebenen drei Phasen illustriert.

Dazu starten wir mit folgendem einfachen Dokument:

```
<!DOCTYPE html>
<html>
<head>
<title>Dokument mit JavaScript</title>
<script type="text/javascript"> … </script>
</head>
<body>
```

```
<p id="interaktiv">Dies ist ein einfacher Textabsatz, aber
mithilfe von JavaScript können wir
ihn interaktiv gestalten. Klicken Sie diesen Absatz doch
einfach mal mit der Maus an!</p>

</body>
</html>
```

Dem p-Element mit der **ID interaktiv** soll nun per JavaScript ein Event-Handler zugewiesen werden. Ziel ist es, dass eine bestimmte JavaScript-Funktion aufgerufen wird, immer wenn der Anwender auf die Fläche des Element klickt. Das Ereignis, das bei einem Mausklick ausgelöst wird, heißt sinnigerweise **click**.

Unser Script läuft in drei Schritten ab:

1. **Warten, bis das Dokument vollständig geladen ist:** Starte die Überwachung des **load**-Ereignisses und führe eine Startfunktion aus, sobald das Ereignis passiert.

2. **Einrichtung der Event-Handler:** Die besagte Startfunktion spricht den Textabsatz an und registriert einen Event-Handler für das **click**-Ereignis.

3. **Ereignis-Verarbeitung:** Die Handler-Funktion, die beim Klick auf den Textabsatz ausgeführt wird.

Schritt 1 ist mit der Anweisung erledigt, die wir bereits oben kennengelernt haben:

```
window.onload = start;
```

Natürlich können wir der Startfunktion auch einen anderen Namen als **start** geben. Üblich ist z.B. **init**.

Die Startfunktion für Schritt 2 könnte so aussehen:

```
function start () {
  document.getElementById("interaktiv").onclick =
  klickverarbeitung;
}
```

Was zunächst kompliziert aussieht, ist nichts anderes als die Anwendung des bekannten Schemas **element.onevent = handlerfunktion;.**

Zur Einrichtung des Event-Handler greifen wir über das **DOM** auf das Dokument zu. Dies ist in der Funktion **start** möglich, denn sie wird beim Eintreten des **load**-Ereignisses ausgeführt.

Damit der Zugriff auf das gewünschte Element so einfach möglich ist, haben wir einen »Angriffspunkt« für das Script geschaffen, indem wir dem **p-Element** eine **ID** zugewiesen haben. Eine solche Auszeichnung **über IDs** und **Klassen (class-Attribute)** spielen eine wichtige Rolle, um Angriffspunkte für Stylesheets und Scripte zu bieten.

Mit der **DOM**-Methode **document.getElementById** (zu deutsch: *gib mir das Element anhand der folgenden ID*) können wir das Element mit der bekannten ID ansprechen. Der Aufruf **document.getElementById("interaktiv")** gibt uns das Objekt zurück, das das **p-Element** repräsentiert.

Wir arbeiten direkt mit diesem Rückgabewert weiter und weisen dem Elementobjekt nun einen Event-Handler zu. Die Objekteigenschaft lautet **onclick**, denn es geht um das **click**-Ereignis. Die auszuführende Handler-Funktion lautet **klickverarbeitung**, dieser Name ist willkürlich gewählt.

Das ist schon alles und damit kommen wir zur Definition der besagten Funktion **klickverarbeitung**:

```
function klickverarbeitung () {
   document.getElementById("interaktiv").innerHTML += " Huhu,
   das ist von Javascript eingefügter Text.";
}
```

Was darin passiert, müssen Sie noch nicht bis ins Detail verstehen. Wie Sie sehen können, wird darin ebenfalls mittels **document.getElementById** das angeklickte p-Element angesprochen. Erneut wird eine Eigenschaft gesetzt, diesmal **innerHTML**. An den bestehenden Wert wird mit dem Operator **+=** ein String angehängt. Wenn Sie das Beispiel im Browser ausführen und auf das Element klicken, ändert sich der Text des Elements.

Zusammengefasst sieht das Beispiel mit eingebettetem JavaScript so aus:

```
<!DOCTYPE html>
<html>
<head>
<title>Beispiel für traditionelles Event-Handling</title>
<script type="text/javascript">

window.onload = start;

function start () {
   document.getElementById("interaktiv").onclick =
   klickverarbeitung;
}

function klickverarbeitung () {
   document.getElementById("interaktiv").innerHTML += " Huhu,
   das ist von Javascript eingefügter Text.";
}

</script>
</head>
<body>

<p id="interaktiv">Dies ist ein einfacher Textabsatz, aber
mithilfe von JavaScript können wir ihn
```

```
interaktiv gestalten. Klicken Sie diesen Absatz doch einfach
mal mit der Maus an!</p>

</body>
</html>
```

Den JavaScript-Code können wir später natürlich in eine externe Datei auslagern.

So einfach und nutzlos dieses kleine Beispiel aussieht: Wenn Sie das dreischrittige Schema verstanden haben, beherrschen Sie einen Großteil der JavaScript-Programmierung und wissen, wie Skripte üblicherweise strukturiert werden und schließlich ausgeführt werden.

Event-Überwachung beenden

Wenn Sie einmal einen Event-Handler bei einem Element registriert haben, wird die Handler-Funktion künftig bei jedem Eintreten des Ereignisses ausgeführt – zumindest solange das Dokument im Browser dargestellt wird und es nicht neu geladen wird. Es ist möglich, die Event-Überwachung wieder mittels JavaScript zu beenden.

Wie beschrieben besteht das traditionelle Event-Handling schlicht darin, dass eine Objekteigenschaft (z.B. **onclick**) durch eine Wertzuweisung mit einer Funktion gefüllt wird. Um das Registrieren des Handlers rückgängig zu machen, beschreiben wir erneut diese Objekteigenschaft. Allerdings weisen wir ihr keine Funktion zu, sondern einen anderen Wert. Dazu bietet sich beispielsweise der spezielle Wert null an, der soviel wie »absichtlich leer« bedeutet.

Das Schema zum Löschen des Event-Handlers lautet demnach:

```
element.onevent = null;
```

Wenn wir im obigen Beispiel die Überwachung des Klick-Ereignisses beim **p-Element** wieder beenden wollen, können wir entsprechend notieren:

```
document.getElementById("interaktiv").onclick = null;
```

Häufiger Fehler: Handler-Funktion direkt aufrufen

Ein häufiger Fehler beim Registrieren eines Event-Handlers sieht folgendermaßen aus:

```
element.onevent = handlerfunktion();   // Fehler!
```

Oft steckt hinter dieser Schreibweise der Wunsch, der Handler-Funktion noch Parameter mitzugeben, damit darin gewissen Daten zur Verfügung stehen:

```
element.onevent = handlerfunktion(parameter);   // Fehler!
```

Sie müssen sich die Funktionsweise des traditionellen Event-Handlings noch einmal durch den Kopf gehen lassen, um zu verstehen, warum diese Anweisungen nicht den gewünschten Zweck erfüllen. Beim korrekten Schema **element.onevent = handlerfunktion;** wird eine Funktion, genauer gesagt ein Funktionsobjekt, in einer Eigenschaft des Elementobjektes gespeichert.

Das ist beim obigen fehlerhaften Code nicht der Fall. Anstatt auf das Eintreten des Ereignisses zu warten, wird die Handler-Funktion **sofort ausgeführt**. Dafür verantwortlich sind die Klammern **()** hinter dem Funktionsnamen – diese Klammern sind nämlich der JavaScript-Operator zum Aufruf von Funktionen.

Das Erste, was der JavaScript-Interpreter beim Verarbeiten dieser Zeile macht, ist der Aufruf der Funktion. Deren **Rückgabewert** wird schließlich in der **onevent**-Eigenschaft gespeichert. In den meisten Fällen hat die Handler-Funktion keinen Rückgabewert, was dem Wert **undefined** entspricht, oder sie gibt **false** zurück, sodass schlicht diese Werte in die Eigenschaft geschrieben werden. Wir wollen die Funktion aber nicht direkt aufrufen, sondern bloß das Funktionsobjekt ansprechen, um es in die Eigenschaft zu kopieren. Daher dürfen an dieser Stelle keine Klammern hinter dem Namen notiert werden.

Der Wunsch, der Handler-Funktion gewisse Daten als Parameter zu übergeben, ist verständlich. Die obige fehlerhafte Schreibweise vermag dies aber nicht zu leisten. Leider ist diese Aufgabenstellung auch nicht so einfach lösbar: Das altbekannte Schema **element.onevent = handlerfunktion;** muss eingehalten werden. Der Funktionsaufruf, der die Parameter übergibt, wird in einer zusätzlichen Funktion untergebracht (**gekapselt**). Schematisch:

```
function helferfunktion (parameter) {
 /* Arbeite mit dem Parameter und verarbeite das Ereignis */
}
function handlerfunktion () {
   helferfunktion("Parameter");
}
element.onevent = handlerfunktion;
```

Das konkrete Beispiel aus dem vorigen Abschnitt können wir so anpassen, dass in der Handler-Funktion bloß eine andere Hilfsfunktion mit Parametern ausgeführt wird:

```
window.onload = start;

function start () {
   document.getElementById("interaktiv").onclick =
   klickverarbeitung;
}

function klickverarbeitung () {
    textHinzufügen(document.getElementById("interaktiv"),
   "Huhu, das ist von Javascript eingefügter Text.");
}

function textHinzufügen (element, neuerText) {
```

```
    element.innerHTML += neuerText;
}
```

In der Handler-Funktion **klickverarbeitung** wird die neue Funktion **textHinzufügen** mit Parametern aufgerufen. Diese wurde verallgemeinert und ist wiederverwendbar: Sie nimmt zwei Parameter an, einmal ein **Elementobjekt** und einmal einen **String**. Die Funktion hängt sie den angegebenen Text in das angegebene Element ein.

Eingebettete Event-Handler-Attribute

Wir haben kennengelernt, wie wir externe JavaScripte einbinden und darin auf »traditionelle« Weise Event-Handler registrieren können. Der Vorteil davon ist, dass wir HTML- und JavaScript-Code und damit das Dokument und das JavaScript-Verhalten trennen können.

Wann immer es möglich ist, sollten Sie diese Vorgehensweise des »**Unobtrusive JavaScript**« wählen. Es soll aber nicht verschwiegen werden, dass es auch möglich ist, JavaScript direkt im HTML-Code unterzubringen und damit auf Ereignisse zu reagieren.

Zu diesem Zweck besitzen fast alle HTML-Elemente entsprechende Attribute, in die Sie den auszuführenden JavaScript-Code direkt hineinschreiben können. In diesem Code können Sie natürlich auch eigene Funktionen aufrufen, die sie in einem script-Element oder einer externen JavaScript-Datei definiert haben. Die Attribute sind genauso benannt wie die entsprechenden JavaScript-Eigenschaften: Die Vorsilbe **on** gefolgt vom Ereignistyp (z.B. **click**). Das Schema lautet dementsprechend:

```
<element onevent="JavaScript-Anweisungen">
```

Ein konkretes Beispiel:

```
<p onclick="window.alert('Absatz wurde geklickt!');">Klicken
Sie diesen Textabsatz an!</p>
```

Hier enthält das Attribut die JavaScript-Anweisung **window.alert('Absatz wurde geklickt!')**;, also einen Aufruf der Funktion **window.alert**. Sie können mehrere Anweisungen in einer Zeile notieren, indem Sie sie wie üblich mit einem Semikolon trennen. Zum Beispiel Funktionsaufrufe:

```
<p onclick="funktion1(); funktion2();">Klicken Sie diesen
Textabsatz an!</p>
```

Wie sie sehen, wird es hier schon unübersichtlich. Sie müssen Ihren Code in eine Zeile quetschen, damit die Browser das Attribut korrekt verarbeiten.

Es gibt viele gute Gründe, HTML und JavaScript möglichst zu trennen und auf solches **Inline-JavaScript** zu verzichten. Natürlich hat diese Grundregel berechtigte Ausnahmen. Als Anfänger sollten sie sich jedoch mit der Trennung sowie dem Registrieren von Event-

Handlern mittels JavaScript vertraut machen, wie es in den vorigen Abschnitten erläutert wurde. Wenn Ihre Scripte komplexer werden, werden Sie vielleicht vereinzelt auf Event-Handler-Attribute zurückgreifen, aber der Großteil sollte ohne sie funktionieren.

Die Verwendung von solchen **Event-Handler-Attrbuten** bringt viele Eigenheiten und Nachteile mit sich, auf die an dieser Stelle nicht weiter eingegangen wird.

Häufiger Fehler: Auszuführenden Code als String zuweisen

Nachdem wir Inline-JavaScript angeschnitten haben, sei auf einen weiteren häufigen Fehler beim traditionellen Event-Handling hingewiesen. Manche übertragen ihr Wissen über Event-Handler-Attribute aus HTML auf das das Registrieren von Event-Handlern in JavaScript. Sie versuchen z.B. folgendes:

```
element.onclick = "window.alert('Element wurde geklickt!');"
```

Oder gleichwertig mithilfe der DOM-Methode setAttribute:

```
element.setAttribute("onclick", "window.alert('Element wurde
geklickt!');");
```

Sprich, sie behandeln die Eigenschaft **onclick** und dergleichen wie Attribute unter vielen. Für die meisten anderen Attribute gilt das auch. Ein Beispiel:

```
<p><a id="link" href="http://de.selfhtml.org/"
    title="Deutschsprachige Anleitung zum Erstellen von
    Webseiten">SELFHTML</a></p>

<script type="text/javascript">
    var element = document.getElementById("link");
    element.title = "Die freie Enzyklopädie";
    element.href = "http://de.wikipedia.org/";
    element.firstChild.nodeValue = "Wikipedia";
</script>
```

Das Script spricht ein Link-Element über seine ID an und ändert dessen Attribute **title** und **href** sowie schließlich dessen Textinhalt. Das Beispiel illustriert, dass sich die Zuweisungen der Attributwerte im HTML und im JavaScript stark ähneln. Die neuen Attributwerte werden im JavaScript einfach als Strings notiert.

Diese Vorgehensweise ist beim Setzen von **Event-Handler-Attributen** über JavaScript nicht völlig falsch. *Theoretisch* haben folgende Schreibweisen denselben Effekt:

```
// Methode 1: Traditionelles Event-Handling
function handlerfunktion () {
   window.alert("Hallo Welt!");
}
element.onevent = handlerfunktion;
```

```
// Methode 2: Auszuführenden Code als als String zuweisen
// (Achtung, nicht browserübergreifend!)
    element.setAttribute("onevent",
    window.alert('HalloWelt!');");
```

Ihnen mag die die zweite Schreibweise in vielen Fällen einfacher und kürzer erscheinen. Doch zum einen hat sie das Problem, dass sie in der Praxis längst nicht so etabliert ist wie die traditionelle: Der Internet Explorer einschließlich der neuesten Version 8 unterstützt diese Schreibweise noch nicht.

Davon abgesehen hat es Nachteile, JavaScript-Code nicht in Funktionen zu ordnen, sondern in Strings zu verpacken. Der Code wird unübersichtlicher und Fehler sind schwieriger zu finden. Sie sollten daher möglichst das traditionelle Schema vorziehen.