Fehlerbehandlung in Javascript

window.onerror

Ihnen steht mit dem **error**-Event-Handler ein Werkzeug zur Verfügung, um innerhalb des Scripts Fehler abzufangen und gezielt darauf zu reagieren.

Zur Unterdrückung von Fehlermeldungen, die zur Laufzeit eines Scripts entstehen, notieren Sie im Kopf der Datei **window.onerror**.

Als Wert weisen Sie den Namen einer Funktion zu, welche die Fehlerbehandlung ausführt. Der Rückgabewert der Funktion muss **true** sein. Damit haben Sie Ihre eigene Fehlerbehandlung.

Der Fehlerbehandlungs-Funktion können Sie bis zu drei optionale Parameter übergeben:

1. Nachricht: enthält die Fehlerbeschreibung des Fehlers
2. Datei: enthält den URI der fehlerverursachenden Datei
3. Zeile: enthält die Zeile in der der Fehler auftritt

Beachten Sie, dass Sie nur logische Fehler überwachen und unterdrücken können, die während der Laufzeit der Scripts entstehen. Syntaxfehler, wie z.B. fehlende Klammern usw. sind damit nicht abfangbar.

Das Beispiel zeigt eine einfache Variante der Fehlerbehandlung. Am Ende des Script-Bereichs wird eine Funktion aufgerufen, die nicht existiert.

```
window.onerror = Fehlerbehandlung;

    function Fehlerbehandlung (Nachricht, Datei, Zeile) {
      Fehler = "Fehlermeldung:\n" + Nachricht + "\n" + Datei + "\n"
+ Zeile;
      zeigeFehler();
      return true;
    }

    function zeigeFehler () {
      alert(Fehler);
    }

    nichtda();
```

Beim Einlesen der Datei wird mit **window.onerror = Fehlerbehandlung;** angewiesen, im Fehlerfall die Funktion Fehlerbehandlung() aufzurufen.

Dieses Ereignis tritt im Beispiel am Ende des Script-Bereichs ein, wo versucht wird, die nichtvorhandene Funktion **nichtda()** aufzurufen.

Die Funktion Fehlerbehandlung()

Die Funktion speichert mit Zeichenkettenverknüpfung die automatisch übergebenen Parameter in der Variablen Fehler. Dann ruft sie im Beispiel die Funktion **`zeigeFehler()`** auf. Diese Funktion dient im Beispiel lediglich dazu, den Fehler zu Demonstrationszwecken auszugeben. Mit **`return true`** wird erreicht, dass der Fehler im Browser nicht angezeigt wird. Mit dem Rückgabewert **`false`** würden Sie bewirken, dass der Browser den Fehler anmeckern würde.

In einer komplexeren Anwendung hätte die Funktion **Fehlerbehandlung()** also alle Möglichkeiten, auf den Fehler zu reagieren. Sie könnte beispielsweise die Zeichenkette des übergebenen Parameters **Nachricht** untersuchen, um herauszufinden, welcher Fehler genau aufgetreten ist. Je nach Situation könnte eine solche Funktion bewirken, dass das Script zu einem früheren Punkt im Programmablauf zurückspringt, Variablen zurücksetzt, den Cursor in ein bestimmtes Formularfeld setzt usw.

Fehlerbehandlung mit try..catch

Mit Hilfe der Fehlerbehandlung mit dem **`error`**-Event-Handler können Sie in einem JavaScript Fehler abfangen, nachdem diese aufgetreten sind. Mit der hier vorgestellten Methode können Sie jedoch bereits im Vorfeld verhindern, dass in kritischen Situationen überhaupt Fehler auftreten. Dazu steht seit der JavaScript-Version 1.5 das **`try..catch`**-Statement zur Verfügung. Es erlaubt Ihnen, Variablen und Werte zu überprüfen und je nach Ergebnis zu reagieren.

Das Beispiel ist so konstruiert, dass es auf eine Variable zugreift, die zunächst noch gar nicht existiert. Das würde zu einem Fehler führen. Das Script "weiß" jedoch, dass dieser Fehler auftreten kann und überprüft deshalb mit dem **`try..catch`**-Statement laufend, ob die Variable schon existiert. Erst wenn die Variable existiert, wird die Überwachung beendet.

```
setTimeout("x=3", 200);

    function zeigeErgebnis (Zaehler, Ergebnis) {
      alert("Nach " + (Zaehler) + " Durchläufen existierte x.\nDie
Zahl x ist " + Ergebnis + ".")
    }

    function teste_x (Zaehler) {
      try {
        if (x == 2) {
          throw "richtig";
        } else if (x == 3) {
          throw "falsch";
        }
      } catch (e) {
        if (e == "richtig") {
          zeigeErgebnis(Zaehler, e);
          return;
        } else if (e == "falsch") {
          zeigeErgebnis(Zaehler, e);
          return;
        }
      } finally {
        Zaehler++;
      }
```

```
setTimeout("teste_x(" + Zaehler + ")", 30);
}
teste_x(0);
```

Am Ende der Datei ist ein Script-Bereich notiert. Darin werden die erste und die letzte Anweisung sofort beim Einlesen ausgeführt. Die restlichen Anweisungen stehen in Funktionen.

Die Variable x

Während des Ladens der Datei wird eine **`Variable x`** zeitverzögert mit dem Wert 3 belegt. Dazu dient die Methode **`setTimeout()`**. Die Variable ist also erst nach 200 Millisekunden verfügbar. Jeder Versuch, vorher auf diese Variable zuzugreifen, würde zu einem Fehler führen.

Die nächste Anweisung, die im Beispiel direkt ausgeführt wird, ist die letzte im Script-Bereich, nämlich **`teste_x(0).`** Damit wird die Funktion **`teste_x()`** aufgerufen, die oberhalb notiert ist.

Die Funktion teste_x()

Diese Funktion versucht, auf die **`Variable x`** zugreifen. Da zum Zeitpunkt des Aufrufes der Funktion jedoch noch nicht sicher ist, ob die **`Variable x`** bereits existiert, ist innerhalb der Funktion zur Vermeidung von Fehlermeldungen das **`try..catch`**-Statement notiert. Beim Aufruf erhält die Funktion **`teste_x()`** einen Parameter namens **`Zaehler`** übergeben. Das dient im Beispiel zu Kontrollzwecken.

Aufbau des try..catch-Statements

Das **`try..catch`**-Statement hat generell folgenden Aufbau: Nach dem Schlüsselwort **`try`** (try = versuchen) wird eine öffnende geschweifte Klammer, gefolgt von der zu prüfenden Bedingung, notiert. Je nach Erfordernissen können Sie dann mit **`throw`** (throw = auswerfen) eigene Fehler definieren. Die **`throw`**-Definition ist jedoch optional. Anschließend folgt eine schließende geschweifte Klammer, die den **`try`**-Block beendet.

Daran anschließend notieren Sie das Schlüsselwort **`catch`** (catch = abfangen). catch hat Funktions-Charakter und erwartet einen Parameter **`e`**. Die **`Variable e`** ist erforderlich, da Sie über diese Variable letztlich die Reaktion des Scripts auf den aktuellen Zustand kontrollieren. Den Namen der Variablen können Sie frei wählen (der Name muss also nicht e sein). Innerhalb des Funktionsblocks von **`catch(),`** der wie üblich in geschweiften Klammern steht, können Sie die mit **`throw`** definierten Fehler auswerten und darauf reagieren.

Zuletzt können Sie noch das Schlüsselwort **`finally`** notieren. In dem davon abhängigen Anweisungsblock können Sie weitere Anweisungen notieren. Diese Anweisungen werden unabhängig von der Fehlerbehandlung in jedem Fall ausgeführt.

Anwendung des try..catch-Statements im Beispiel

Im ersten Teil der Anweisung wird geprüft, ob die **`Variable x`** den Wert 2 oder 3 besitzt. Je nach Ergebnis werden mit **`throw`** verschiedene Fehlerwerte definiert. Hat x den Wert 2, so wird der

"Fehler" mit dem Wert richtig generiert. Hat sie den Wert 3 so erhält der Fehler den Wert falsch. Weitere Fehlervarianten werden nicht behandelt.

Im ersten Durchlauf existiert die **`Variable x`** im Beispiel noch gar nicht, da sie ja erst nach 200 Millisekunden existiert. Sie kann damit weder den Wert 2 noch den Wert 3 besitzen. In der nachfolgenden Fehlerbehandlungsroutine **`catch(e)`** wird geprüft, ob einer der definierten Fehler, also richtig oder falsch, aufgetreten ist. Zunächst ist das offensichtlich nicht der Fall. Die Anweisungen, die von den "Fehlerwerten" richtig und falsch abhängig sind, werden deshalb nicht ausgeführt. Die **`finally`**-Anweisung wird dagegen in jedem Fall ausgeführt. Sie bewirkt im Beispiel, dass der übergebene Parameter **`Zaehler`** um 1 erhöht wird.

Gesamtkontrolle

Am Ende ruft sich die Funktion **`teste_x()`** mit **`setTimeout()`** um 30 Millisekunden zeitverzögert selbst wieder auf. So behält sie die Kontrolle über das Geschehen, bis ein definierter Zustand eintritt. Der Parameter **`Zaehler`** wird dabei mit Hilfe einer Zeichenkettenverknüpfung übergeben.

Interessant wird es, wenn der Zeitpunkt erreicht ist, zu dem die **`Variable x`** existiert. In diesem Fall tritt einer der vordefinierten Fälle ein. Da x den Wert 3 besitzt, wird der **`throw`**-Fehler mit dem Wert falsch generiert (dies soll im Beispiel einfach zeigen, dass **`throw`** zur Erzeugung von Werten gedacht ist, die durchaus und oft auch Fehlerzustände bezeichnen). Im nachfolgenden **`catch(e)-`** Block führt dies dazu, dass die Funktion **`zeigeErgebnis()`** aufgerufen wird. Im Beispiel wird für beide definierten **`throw`**-Werte die gleiche Funktion aufgerufen. Sie können an dieser Stelle jedoch auch völlig verschiedene Anweisungen notieren. Jede dieser Fehlerbehandlungsroutinen bricht gleichzeitig die Funktion **`teste_x()`** mit **`return`** ab, da x ja nun existiert und der "kritische Zustand" beendet ist.

Ausgabe des Ergebnisses

Die Funktion **`zeigeErgebnis()`** erhält als Parameter die Variablen **`Zaehler`** und Ergebnis übergeben. In der Variablen **`Zaehler`** ist die Anzahl der Durchläufe bis zur Existenz der **`Variablen x`** gespeichert und in der Variablen **`Ergebnis`** das Resultat der Fehlerbehandlung. Mit **`alert()`** wird im Beispiel zur Kontrolle ausgegeben, wie viele Durchläufe benötigt wurden und was für ein Ergebnis erreicht wurde.

Anwendungsfälle

Prüfungen mit dem **`try..catch`**-Statement sind z.B. dann sinnvoll, wenn Sie wie im Beispiel mit **`setTimeout()`** zeitversetzte Aktionen ausführen und davon abhängige Anweisungen ausführen wollen. Ebenfalls sinnvoll ist das Statement, wenn Sie z.B. auf Variablen oder Funktionen zugreifen wollen, die in anderen Frame-Fenstern notiert sind, wobei das Script nicht wissen kann, ob die Datei im anderen Frame-Fenster, in der das entsprechende Script notiert ist, bereits eingelesen oder überhaupt die dort aktuell angezeigte Seite ist.