Fehlerbehandlung mit dem try..catch-Statement

Mit Hilfe der **Fehlerbehandlung mit dem Event-Handler onerror** können Sie in einem JavaScript Fehler abfangen, **nachdem** diese aufgetreten sind. Mit der hier vorgestellten `try..catch`-Methode können Sie jedoch bereits im Vorfeld verhindern, dass in kritischen Situationen überhaupt Fehler auftreten.

```
<html><head><title>Test</title>
<script type="text/javascript">
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);
</script>
</head><body>
</body></html>
```

Erläuterung:

Im Dateikopf 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 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` wird eine öffnende geschweifte Klammer notiert, gefolgt von der zu prüfenden Bedingung. Je nach Erfordernissen können Sie mit `throw` eigene Fehlerwerte definieren. Die `throw`-Definition ist optional, eine schließende geschweifte Klammer beendet den `try`-Block. Daran anschließend notieren Sie das Schlüsselwort `catch`. `catch` hat Funktions-Charakter und erwartet einen Parameter `e`. Die Variable `e` ist erforderlich, den Namen der Variablen können Sie jedoch frei wählen (der Name muss also nicht `e` sein). Innerhalb des Funktionsblocks von `catch` können Sie so die mit `throw` definierten Fehlerwerte auswerten und darauf reagieren.
Zuletzt können Sie noch das optionale Schlüsselwort `finally` notieren. Die in diesem Anweisungsblock enthaltenen 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.

Die Ausgabe zeigt übrigens sehr deutlich, wie unterschiedlich die Browser mit `setTimeout()` umgehen. Während Internet Explorer 6 und Netscape 6 bei Tests im Schnitt sieben Durchläufe benötigten, brauchten Opera 8.5 und Firefox 1.5 durchschnittlich drei.

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.