

14 Reguläre Ausdrücke

Wir haben gesehen, wie Formulare eingesetzt werden können, um mit dem Benutzer zu interagieren. So wird etwa vom Anwender verlangt, dass der Name oder die E-Mail-Adresse eingegeben wird. Die Eingaben lassen sich durch JavaScript überprüfen. So könnte ein Skript eine ungültige E-Mail-Adresse zurückweisen. Wie bereits gezeigt wurde, kann die Versendung eines Formulars mit dem Event-Handler `onsubmit` verhindert werden (siehe *Formulareingaben überprüfen*, S. 234). Wie können wir jedoch überprüfen, ob die Benutzereingaben plausibel sind?

Solche Überprüfungen lassen sich mit regulären Ausdrücken durchführen, mit denen in Strings nach bestimmten Mustern gesucht werden kann. Reguläre Ausdrücke sind Teil des ECMAScript-Standards und werden in JavaScript seit der Version 1.2 unterstützt, d.h., alle gängigen Browser können damit umgehen. Da der ECMAScript-Standard reguläre Ausdrücke umfasst, gehört dieses Kapitel eigentlich in den vorderen Teil des Buches. Jedoch wollte ich dieses Thema erst nach den Formularen behandeln, da so der Verwendungszweck von regulären Ausdrücken besser deutlich wird.

Am Ende dieses Kapitels werden Alternativen gezeigt, wie man auch ohne reguläre Ausdrücke einfache Überprüfungen von Formularen durchführen kann.

Reguläre Ausdrücke sind nicht ganz einfach. Die nachfolgenden Kapitel bauen nicht auf diesem Kapitel auf, sodass Sie dieses Kapitel überspringen können, wenn Sie im Moment keine Strings oder Formulareingaben überprüfen wollen.

14.1 Reguläre Ausdrücke verwenden

Stellen Sie sich vor, Sie möchten ein Skript schreiben, das aufgrund der Eingabe eines Benutzernamens entscheidet, welche HTML-Seite angezeigt wird. Dies soll also eine Art Login-Skript darstellen. Wenn Sie eine Überprüfung des Benutzernamens nach dem Muster

```
if (eingabe == "Stefan") ...
```

durchführen, werden Sie feststellen, dass das Skript nicht sehr flexibel ist. Was passiert, wenn der Name *Stefan* mit einem kleinen *s* eingegeben wird? Schon funktioniert das Skript nicht mehr. Sie können dieses Problem natürlich mit

```
if (eingabe.toLowerCase() == "stefan") ...
```

umgehen. Aber auch hier werden Sie feststellen, dass das Skript sehr unflexibel ist. Schon ein Leerzeichen am Ende des Strings bringt das Skript durcheinander. Auch kommt das Skript nicht damit zurecht, wenn man zusätzlich einen Nachnamen eingibt.

Mit regulären Ausdrücken können Sie solche Probleme lösen. Reguläre Ausdrücke sind ein mächtiges Werkzeug. Zugegebenermaßen ist der Einstieg vielleicht nicht ganz einfach. Sie werden jedoch recht bald mit regulären Ausdrücken umgehen können, wenn Sie die hier gezeigten Beispiele durchgehen und versuchen, eigene reguläre Ausdrücke zu definieren.

Reguläre Ausdrücke gibt es auch in anderen Programmiersprachen, z.B. in Perl. Gerade für die Überprüfung von Formulareingaben eignen sich reguläre Ausdrücke hervorragend. Perl wird hierfür häufig auf dem Server eingesetzt.

*Erzeugen von regulären
Ausdrücken*

Um in JavaScript mit regulären Ausdrücken zu arbeiten, haben Sie generell zwei Möglichkeiten. Sie können erstens einen regulären Ausdruck, ähnlich wie in Perl, innerhalb von zwei Schrägstrichen definieren (z.B. `/abc/`), oder Sie können als zweite Möglichkeit mit `new` eine Instanz der `RegExp`-Klasse erzeugen. Wir werden zunächst nur die erste Variante verwenden und später auf die `RegExp`-Klasse zurückkommen.

*Muster in Strings
wiederfinden*

Mit regulären Ausdrücken können Sie nach bestimmten Mustern in Strings suchen. Damit Sie sehen, wie das funktioniert, möchte ich mit einem einfachen Beispiel anfangen, in dem die Funktion `test()` den Inhalt eines Textfelds überprüft:

reg01.html

```
<html>
<head>
<title>RegExp</title>
<meta http-equiv="Content-Script-Type"
content="text/javascript" />
```

```

<script type="text/javascript">
function test(eing) {
  var reg = /@/;
  if (reg.exec(eing))
    alert("Gueltige E-Mail-Adresse");
  else alert("Ungueltige E-Mail-Adresse");
}
</script>
</head>
<body>

  Bitte geben Sie eine E-Mail-Adresse ein:<br />
  <form>
    <p>
      <input type="text" name="eingabe" size="30" />
      <input type="button" value="Test"
        onclick="test(this.form.eingabe.value)" />
    </p>
  </form>

</body>
</html>

```

Zuerst wird der reguläre Ausdruck `/@/` mit

```
var reg = /@/;
```

definiert. Danach wird durch `exec()` festgestellt, ob dieses Muster `/@/` im String `eing` vorkommt. Die beiden Schrägstriche sind dabei nur die Begrenzungen, die den regulären Ausdruck definieren (ähnlich wie die Anführungsstriche eines Strings), d.h., mit `exec()` wird in unserem Fall nur überprüft, ob das Zeichen `@` in dem String `eing` vorkommt. Unser Skript erkennt anhand dieses Zeichens, ob eine gültige E-Mail-Adresse eingegeben wurde. `reg.exec(eing)` liefert `null` zurück, wenn das Zeichen `@` in dem String `eing` nicht vorkommt. Falls dieses Zeichen vorkommt, wird, wie wir später sehen werden, ein Objekt mit weiteren Informationen zurückgeliefert. Wir können in unserem Beispiel eine einfache `if`-Abfrage verwenden, da der Wert `null` als `false` interpretiert wird und das zurückgelieferte Objekt als `true`.

Sie werden schnell feststellen, dass unsere Prüfroutine nicht besonders gut ist, denn man kann sehr einfach ungültige E-Mail-Adressen finden, die unser Skript als gültig erkennen würde, z.B. einen String, der nur das `@`-Zeichen beinhaltet.

Unser Skript könnte etwas verbessert werden, wenn wir überprüfen würden, ob vor und nach dem `@`-Zeichen irgendwelche anderen Zeichen erscheinen. Dazu müssen Sie den regulären Ausdruck `reg=/@/` in `reg01.html` wie folgt ändern:

```
reg02.html      var reg = /.+@.+/;
```

(Auszug)

Das wird Ihnen vielleicht nicht sofort einleuchten, aber wenn Sie das Skript ausprobieren, werden Sie feststellen, dass weder "@abc" noch "abc@" als Eingabe angenommen werden. Die Eingabe "abc@abc" wird jedoch akzeptiert. Dieses Skript zum Überprüfen von E-Mail-Adressen ist etwas besser als die erste Version. Jedoch werden Sie auch jetzt feststellen, dass noch sehr viele ungültige Adressen akzeptiert werden.

Besondere Zeichen

Was bedeutet das kryptische `/.+@.+/`? JavaScript versucht, das Muster, das im regulären Ausdruck angegeben ist, im Eingabestring wiederzufinden. Die Zeichen `.` und `+` haben innerhalb von regulären Ausdrücken eine besondere Bedeutung. Der Punkt passt auf jedes beliebige Zeichen (außer auf den Zeilenumbruch `\n`). Das folgende Skript würde *gefunden* ausgeben, da das Muster `/a.c/` auf die Zeichenkette "abc" passt. Der String "abbc" hingegen würde nicht passen, da zwischen a und c zwei Zeichen stehen.

```
reg03.html
```

(Auszug)

```
var str = "abc";
var reg = /a.c/;
if (reg.exec(str))
    alert("gefunden");
else alert("nicht gefunden");
```

Das `+`-Zeichen bedeutet in regulären Ausdrücken, dass das vorhergehende Zeichen mindestens einmal vorkommen muss. Würden wir im letzten Beispiel den regulären Ausdruck in `/a.+c/` ändern, würde nicht nur "abc", sondern auch "abbc" passen.

In dem regulären Ausdruck `/.+@.+/` geht dem `+`-Zeichen ein Punkt voraus. Damit wird nach einem beliebigen Zeichen gesucht, das mindestens einmal vorkommt. Das heißt, mit dem regulären Ausdruck `/.+@.+/` wird überprüft, ob das Muster *ein @-Zeichen mit mindestens einem Zeichen davor und danach* in dem Eingabestring gefunden werden kann.

14.1.1 Kurzschreibweise

Wem reguläre Ausdrücke noch nicht kryptisch genug sind, für den gibt es eine Kurzschreibweise. Statt

```
var reg = /a.c/;
var found = reg.exec("abc");
```

kann man Folgendes schreiben:

```
var found = /a.c/.exec("abc");
```

Auch Modifikatoren, die wir im nächsten Abschnitt kennenlernen werden, können angegeben werden, z.B.:

```
var found = /a.c/i.exec("abc");
```

14.1.2 Modifikatoren

Oft spielt es keine Rolle, ob der Anwender Groß- oder Kleinschreibung für eine Eingabe verwendet. Das Problem wurde bereits am Anfang dieses Kapitels im Zusammenhang mit einem Skript, das unterschiedliche Webseiten für verschiedene Personen laden soll, erwähnt. Der reguläre Ausdruck

```
/Stefan/
```

passt zwar auf den String "Stefan", aber nicht auf "stefan" oder "STEFAN", da der Computer zwischen Groß- und Kleinschreibung unterscheidet. Der folgende reguläre Ausdruck umgeht dieses Problem:

```
/stefan/i
```

Das angehängte *i* bezeichnet man als Modifikator. Es bezweckt, dass in dem regulären Ausdruck zwischen Groß- und Kleinschreibung nicht unterschieden wird.

JavaScript kennt außerdem den Modifikator *g*. Die Angabe dieses Modifikators bedeutet, dass eine globale Suche durchgeführt werden soll. Dieser Modifikator kommt im Zusammenhang mit den Methoden `match()` und `replace()`, die wir später kennenlernen werden, zur Geltung.

Weiterhin kann der Modifikator *m* verwendet werden, wenn sich ein String über mehrere Zeilen erstreckt. Darauf werden wir später im Zusammenhang mit Ankern zurückkommen (siehe *Anker*, S. 249).

14.1.3 Besondere Zeichen

Ähnlich wie `.` und `+` gibt es in regulären Ausdrücken weitere Zeichen mit besonderer Bedeutung. Diese Zeichen werden auch Metazeichen genannt. Die folgenden Zeichen haben eine besondere Bedeutung in regulären Ausdrücken:

```
\ / | ( ) [ { ^ $ * + ? .
```

Wenn Sie nach diesen Zeichen in einem String suchen wollen (statt sie als besondere Zeichen zu verwenden), müssen Sie einen Backslash (`\`) davorsetzen. Mit `*` suchen Sie beispielsweise nach einem `*`. Um einen Backslash zu finden, muss `\\` benutzt werden.

*Groß- und
Kleinschreibung*

Modifikator i

Modifikator g

Modifikator m

Metazeichen

Der Backslash wird auch dazu verwendet, einigen Buchstaben und Ziffern eine besondere Bedeutung zu geben. So findet man mit `\n` beispielsweise nicht einen Backslash, gefolgt von dem Buchstaben `n`, sondern die Kombination `\n` wird benutzt, um einen Zeilenumbruch zu finden.

Die folgende Tabelle listet besondere Zeichen auf, die für unterschiedliche Buchstaben, Ziffern und/oder Sonderzeichen stehen:

Tab. 14-1
Besondere Zeichen

Zeichen	Bedeutung
<code>\n</code>	Zeilenvorschub
<code>\r</code>	Wagenrücklauf
<code>\t</code>	Tabulator
<code>\v</code>	Vertikaltabulator
<code>\f</code>	Seitenvorschub
<code>\d</code>	Eine Ziffer (gleichbedeutend mit <code>[0-9]</code>)
<code>\D</code>	Ein Zeichen, das keine Ziffer ist (gleichbedeutend mit <code>[^0-9]</code>)
<code>\w</code>	Ein alphanumerisches Zeichen (gleichbedeutend mit <code>[a-zA-Z_0-9]</code>)
<code>\W</code>	Ein nicht alphanumerisches Zeichen (gleichbedeutend mit <code>[^a-zA-Z_0-9]</code>)
<code>\s</code>	Ein Whitespace-Zeichen (gleichbedeutend mit <code>[\t\v\n\r\f]</code>)
<code>\S</code>	Ein Zeichen, das kein Whitespace ist (gleichbedeutend mit <code>[^\t\v\n\r\f]</code>)
<code>.</code>	Jedes beliebige Zeichen (außer <code>\n</code>)

Whitespaces

Die Zeichen Leerzeichen, Zeilenvorschub, Wagenrücklauf, Tabulator, Vertikaltabulator, Seitenvorschub werden allgemein als Whitespaces bezeichnet.

Das folgende Beispiel gibt z.B. an, ob sich ein String über mehrere Zeilen erstreckt:

`reg04.html`
(Auszug)

```
var str = "Dieser String erstreckt sich\n" +
         "ueber mehrere Zeilen";

var reg = /\n/;
if (reg.exec(str)) alert("Mehrere Zeilen.");
else alert("Nur eine Zeile.");
```

14.1.4 Multiplikatoren

Mithilfe von Multiplikatoren (auch Vervielfacher oder Quantifikatoren genannt) können Sie festlegen, wie oft das vorhergehende Zeichen vorkommen soll. Wir haben bereits das +-Zeichen kennengelernt, das zu dieser Gruppe gehört. Die Tabelle 14–2 zeigt, welche Multiplikatoren JavaScript kennt.

Zeichen	Bedeutung
$x\{m,n\}$	x soll mindestens m -mal, aber nicht mehr als n -mal vorkommen.
$x\{n,\}$	x soll mindestens n -mal vorkommen.
$x\{n\}$	x soll genau n -mal vorkommen.
x^*	x soll 0-mal oder öfter vorkommen (gleichbedeutend mit $x\{0,\}$).
x^+	x soll 1-mal oder öfter vorkommen (gleichbedeutend mit $x\{1,\}$).
$x?$	x soll 0- oder 1-mal vorkommen (gleichbedeutend mit $x\{0,1\}$).

Tab. 14–2

Multiplikatoren

Um ein Verständnis für reguläre Ausdrücke zu bekommen, würde ich Ihnen empfehlen, mit diesen Tabellen zu versuchen, eigene reguläre Ausdrücke zu erstellen. Dazu können Sie das Skript in `reg03.html` oder `reg05.html` verwenden. Ein Beispiel wäre der reguläre Ausdruck `/a{1,3}bc/`. Dieser Ausdruck passt auf:

```
abc
aabc
aaabc
aaaabc
aaabcbc
aaaaaxyzaaabc
```

aber nicht auf:

```
bc
aaaxbc
aaabxyzc
```

Mit dem folgenden Skript können Sie diese Liste überprüfen:

```
<html>
<head>
  <title>RegExp</title>
</head>
<body>
  <div id="ausgabe"></div>
  <script type="text/javascript">
    var ausg = document.getElementById("ausgabe");
    var str = ["abc", "aabc", "aaabc", "aaaabc", "aaabcbc",
              "aaaaaxyzaaabc", "bc", "aaaxbc", "aaabxyzc"];
```

`reg05.html`

```

var reg = /a{1,3}bc/;
for (i in str) {
  var found = reg.exec(str[i]);
  if (found)
    ausg.innerHTML += str[i] + " - gefunden: " + found;
  else ausg.innerHTML += str[i] + " - nicht gefunden";
  ausg.innerHTML += "<br />";
}
</script>
</body>
</html>

```

Dieses Beispiel gibt neben jedem String des Arrays `str` an, ob das Muster `/a{1,3}bc/` in diesem String gefunden wurde oder nicht. Wurde das Muster gefunden, wird außerdem ausgegeben, welcher Teil des Strings gepasst hat. Dafür wird die Variable `found` definiert, die den Rückgabewert von `exec()` entgegennimmt:

```
var found = reg.exec(str[i]);
```

Die Methode `exec()` gibt den Teil des Strings `str[i]` zurück, auf den der reguläre Ausdruck gepasst hat. Wie wir später sehen werden, liefert `exec()` eigentlich ein Objekt mit verschiedenen Informationen zurück. Da wir nur ein Ergebnis erwarten, können wir hier so tun, als ob `exec()` einen String zurückliefert. Konnte JavaScript das Muster nicht wiederfinden, wird der Wert `null` zurückgeliefert. Da der Wert `null` in einer `if`-Abfrage als `false` interpretiert wird, können wir an dieser Stelle eine einfache `if`-Abfrage verwenden.

Sie werden sich vielleicht wundern, warum der reguläre Ausdruck `/a{1,3}bc/` auf `aaaabc` passt, denn schließlich kommt hier der Buchstabe `a` viermal vor. Wenn Sie sich aber anschauen, welchen Wert die Variable `found` in diesem Fall annimmt, sehen Sie, dass nur der Teil `aaabc` gefunden wird, d.h., im gefundenen String kommt nur dreimal der Buchstabe `a` vor. Mit dem regulären Ausdruck haben Sie also nur festgelegt, dass der Buchstabe `a` ein- bis dreimal vorkommen soll, gefolgt von `bc`. Wir haben keine Aussage darüber gemacht, was vor oder nach dem gefundenen Teilstring stehen darf.

14.1.5 Mehrere Zeichen zusammenfassen

Wenn Sie `/abc+/` schreiben, bezieht sich das `+`-Zeichen nur auf das `c` und nicht auf die komplette Zeichenkette `abc`. Wenn Sie das `+`-Zeichen für die gesamte Zeichenkette `abc` verwenden möchten, müssen Sie Klammern benutzen, also beispielsweise:

```
/(abc)+/
```

Runde Klammern

Dieses Muster passt etwa auf:

```
abc
aabcc
abcabcabc
xyzabcabcabc
```

14.1.6 Das Oder-Zeichen

Das Oder-Zeichen | kann dazu verwendet werden, mehrere Alternativen zuzulassen. Beispielsweise passt

```
/Stefan|Stephan/
```

sowohl auf "Stefan" als auch auf "Stephan". Man kann auch Folgendes schreiben:

```
/Ste(f|ph)an/
```

Jetzt bezieht sich das Oder-Zeichen nur auf den Teil, der innerhalb der Klammern steht.

14.1.7 Anker

Das folgende Skript zeigt, wie wir eine Login-Seite erstellen können. Wird der Username "Stefan" (ohne Rücksicht auf Groß- und Kleinschreibung) eingegeben, wird die Seite stefan.html aufgerufen. Ansonsten soll die Seite default.html geladen werden.

```
<html>
<head>
<title>RegExp</title>
<meta http-equiv="Content-Script-Type"
  content="text/javascript" />
<script type="text/javascript">
function test(eing) {
  var reg = /stefan/i;
  if (reg.exec(eing))
    location.href = "stefan.html";
  else location.href = "default.html";
}
</script>
</head>
```

reg06.html

```

<body>
  Bitte geben Sie Ihren Namen ein:<br />
  <form onsubmit="test(this.eing.value);return false;">
    <p>
      <input type="text" id="eing" name="eing" size="30" />
      <input type="button" value="Login"
        onclick="test(this.form.eing.value)" />
    </p>
  </form>
</body>
</html>

```

Es ist vielleicht noch relativ einleuchtend, dass die Seite stefan.html selbst dann geladen wird, wenn der eingegebene Name "Stefanie" ist. Aber wer hätte gedacht, dass auch "Gloria Estefan" als der Benutzer Stefan durchgeht? Um dies zu vermeiden, müssen wir sicherstellen, dass sich am Anfang und Ende des Strings "stefan" eine Wortgrenze befindet. Dafür gibt es das besondere Zeichen \b. Wenn wir den regulären Ausdruck im vorhergehenden Beispiel durch

reg07.html
(Auszug)

```
/\bstefan\b/i
```

ersetzen, werden sowohl "Stefanie" als auch "Gloria Estefan" an die default.html-Seite weitergeleitet. Die Eingabe "Stefan Koch" wird jedoch weiterhin akzeptiert, da *Stefan* hier ein separates Wort ist.

Tabelle 14–3 zeigt einige besondere Zeichen, sogenannte Anker, die ähnlich wie \b eingesetzt werden können:

Tab. 14–3
Anker

Zeichen	Bedeutung
^	Passt auf den Anfang eines Strings.
\$	Passt auf das Ende eines Strings.
\b	Passt auf eine Wortgrenze (zwischen \w und \W).
\B	Passt auf eine Stelle, an der keine Wortgrenze ist.

Mehrzeilige Strings

Wie die Tabelle zeigt, findet man mit \$ das Ende eines Strings. Hierbei ist zu beachten, dass bei einem mehrzeiligen String unter Verwendung des Modifikators m auch das Ende einer Zeile gefunden wird. Der String "yz\nzz" enthält durch \n einen Zeilenumbruch. Sucht man in diesem String mit dem regulären Ausdruck

```
/.z$/m
```

so ist das Ergebnis "yz", da der Modifikator m verwendet wurde und das \$-Zeichen auf den Zeilenumbruch passt. Ohne das m am Ende des regulären Ausdrucks lautet das Ergebnis "zz".

14.1.8 Zeichenklassen

Innerhalb eckiger Klammern können Sie Zeichenklassen definieren. So definiert `[0-9]` die Menge aller Ziffern. Der reguläre Ausdruck `/xyz[0-9]/` passt beispielsweise auf die Strings `xyz0`, `xyz1`, `xyz2`, ..., `xyz9`.

Eckige Klammern

Der folgende reguläre Ausdruck überprüft, ob der String nur Ziffern enthält:

```
/^[0-9]*$/
```

Wie im letzten Abschnitt erklärt wurde, wird mit `^` und `$` der Anfang bzw. das Ende eines Strings gefunden. Da zwischen dem `^` und `$` nur die Menge `[0-9]` im Zusammenhang mit dem Multiplikator `*` angegeben ist, werden in dem String lediglich Ziffern akzeptiert.

Menge aller Zahlen

Es wurde bereits erwähnt, dass man auch `\d` verwenden kann, um Ziffern zu finden, d.h., der reguläre Ausdruck

```
/^\d*$/
```

ist gleichbedeutend mit dem vorherigen.

Sie können in Zeichenklassen auch einzelne Zeichen angeben. Beispielsweise findet man mit der Zeichenklasse `[aeiou]` einen der Vokale. Es wird in diesem Fall nicht nach dem String `"aeiou"` gesucht, sondern es bedeutet, dass an dieser Stelle im Prüfstring ein einzelner Vokal stehen muss. Zum Beispiel würden durch

```
/h[euo]y/
```

die Teilstrings `"hey"`, `"hoy"` oder `"huy"` gefunden werden, jedoch nicht `"heuo"`.

Das ^-Zeichen in Zeichenklassen

Mit dem `^`-Zeichen kann innerhalb einer Zeichenklasse festgelegt werden, dass die beinhalteten Zeichen an dieser bestimmten Stelle *nicht* vorkommen dürfen. `^[0-9]` bedeutet z.B., dass dort keine Ziffer stehen darf (d.h., es ist gleichbedeutend mit `\D`). Das `^`-Zeichen muss dabei als erstes Zeichen in der Zeichenklasse erscheinen.

Bitte beachten Sie, dass das `^`-Zeichen hier eine andere Bedeutung hat als vorher, da es innerhalb einer Zeichenklasse verwendet wird. Außerhalb einer Zeichenklasse wird damit der Anfang eines Strings gefunden. Die meisten anderen besonderen Zeichen verlieren innerhalb von Zeichenklassen ebenfalls ihre Bedeutung und werden als normale Zeichen behandelt.

Die Menge aller Kleinbuchstaben lässt sich mit `[a-z]` angeben. Wenn Sie auch die Großbuchstaben akzeptieren wollen, können Sie `[a-zA-Z]` schreiben. Dies umfasst aber keine Umlaute.

Menge aller Buchstaben

14.1.9 Klammern

Die Verwendung von Klammern wurde bereits demonstriert. Klammern haben jedoch eine weitaus größere Bedeutung, als nur Zeichenketten zusammenzufassen. Klammern in regulären Ausdrücken fungieren als Zwischenspeicher für gefundene Teilstrings. Auf den Teilstring, der durch die erste Klammer gefunden wurde, kann man innerhalb desselben regulären Ausdrucks mit \1 zugreifen. Um den Inhalt der zweiten Klammer zu erfahren, wird \2 verwendet usw. Dieser reguläre Ausdruck soll dies verdeutlichen:

```
/(ab*c) xyz \1/
```

Der Teilstring, der durch (ab*c) gefunden wird, wird in einer besonderen Variablen gespeichert, damit darauf später wieder zugegriffen werden kann. In unserem Beispiel greifen wir mit \1 auf den vorher gefundenen Teilstring zu. Unser regulärer Ausdruck stellt also sicher, dass vor und nach dem "xyz" jeweils der gleiche Teilstring erscheint. Auf folgende Strings passt dieses Muster:

```
abc xyz abc
abbc xyz abbc
abbbc xyz abbbc
```

Im Gegensatz dazu wird in den folgenden Strings keine Übereinstimmung mit dem verwendeten Muster gefunden, da die Teilstrings vor und nach dem xyz nicht identisch sind:

```
abc xyz
abc xyz abbc
```

Mit dem folgenden regulären Ausdruck können Sie einfache HTML-Tags finden (z.B. <center>Irgendein Text</center>):

```
/<(.*?)>.*<\/\1>/
```

14.2 Die RegExp-Klasse

14.2.1 Der RegExp()-Konstruktor

Wie bereits erwähnt, können Sie auch den RegExp()-Konstruktor einsetzen, um einen regulären Ausdruck zu definieren. Statt

```
reg = /abc/;
```

können Sie genauso

```
reg = new RegExp("abc");
```

schreiben. Bitte beachten Sie, dass Sie in diesem Fall keine Schrägstriche, sondern Anführungszeichen verwenden müssen, um den regulären Ausdruck abzugrenzen. Die Modifikatoren `g`, `i` und `m` werden optional als zweites Argument angegeben:

```
reg = new RegExp("abc", "gi");
```

Beide gezeigten Möglichkeiten machen Gebrauch von der `RegExp`-Klasse, d.h., Sie können die Methoden, die wir später anschauen werden, in beiden Fällen anwenden.

Außer der Schreibweise gibt es zwischen den beiden Möglichkeiten jedoch noch einen weiteren Unterschied. Bevor ein regulärer Ausdruck verwendet werden kann, muss er kompiliert werden. Der Programmierer muss sich darum nicht selbst kümmern, da dies automatisch geschieht. Der Unterschied zwischen den beiden Varianten, wie man einen regulären Ausdruck definieren kann, liegt im Zeitpunkt der Kompilierung.

*Kompilierung eines
regulären Ausdrucks*

Sie sollten einen regulären Ausdruck in Schrägstrichen angeben, wenn Sie den regulären Ausdruck vor Ausführung des Programms kennen und sich dieser im Verlauf des Programms nicht ändert, da in diesem Fall der reguläre Ausdruck beim Start des Programms kompiliert wird. Verwenden Sie dagegen den `RegExp()`-Konstruktor, wenn Sie den regulären Ausdruck vor Ausführung des Skripts nicht kennen oder sich dieser im Verlauf des Programms ändern soll. Der `RegExp()`-Konstruktor bietet eine Laufzeit-Kompilierung, d.h., der reguläre Ausdruck wird erst kompiliert, wenn er benötigt wird. Die Kompilierung geht insgesamt sehr schnell, sodass dies für den Anwender in den meisten Fällen nicht spürbar sein dürfte.

14.2.2 Methoden der RegExp-Klasse

Die `RegExp`-Klasse kennt die Methoden:

- `exec(str)`
- `test(str)`

exec(str)

Die Methode `exec()` haben wir bereits im Einsatz gesehen. Damit wird in dem String `str` nach dem Muster gesucht, das im regulären Ausdruck angegeben ist.

Wenn das Muster in dem String `str` nicht wiedergefunden werden konnte, liefert `exec()` den Wert `null` zurück. Ansonsten wird ein Objekt mit den folgenden Elementen zurückgegeben:

Tab. 14-4

Elemente des durch `exec()` zurückgegebenen Objekts

Element	Bedeutung
<code>index</code>	Die Position des ersten gefundenen Zeichens im String <code>str</code>
<code>input</code>	Der String <code>str</code> , auf den der reguläre Ausdruck angewendet wurde
<code>[0]</code>	Der Teilstring, der als Letztes gefunden wurde
<code>[1],[2],..., [n]</code>	Die Teilstrings, die mithilfe von Klammern gefunden wurden (d.h. die Werte, auf die innerhalb des regulären Ausdrucks mit <code>\1</code> , <code>\2</code> usw. zugegriffen werden kann)

Das folgende Beispiel demonstriert, wie die gezeigten Eigenschaften ausgelesen werden können:

```
reg08.html    <html>
              <head>
                <title>RegExp</title>
              </head>
              <body>

                <div id="ausgabe"></div>

                <script type="text/javascript">

                  var ausg = document.getElementById("ausgabe");
                  var str = "Dilbert Dogbert Catbert Ratbert";
                  var reg = /\br(\w*)bert/i;

                  var found = reg.exec(str);

                  if (found) {
                    ausg.innerHTML +=found.index + "<br />" +
                      found.input + "<br />" +
                      found[0] + "<br />" +
                      found[1];
                  }

                </script>
              </body>
            </html>
```

Dieses Skript produziert die folgende Ausgabe:

```
24
Dilbert Dogbert Catbert Ratbert
Ratbert
at
```

Bei der Ausführung von `exec()` werden außerdem folgende Eigenschaften des `RegExp`-Objekts verändert:

Element	Bedeutung
global	Gibt an, ob der Modifikator <code>g</code> verwendet wurde.
ignoreCase	Gibt an, ob der Modifikator <code>i</code> verwendet wurde.
lastIndex	Die Position, an der die nächste Suche gestartet wird
multiline	Gibt an, ob der Modifikator <code>m</code> verwendet wurde.
source	Der Inhalt des regulären Ausdrucks

Tab. 14-5

Veränderte Eigenschaften eines `RegExp`-Objekts nach dem Aufruf von `exec()`

Das folgende Beispiel gibt `true` aus, da der Modifikator `i` benutzt wurde:

```
var str = "ABC";
var reg = /ab/i;
var found = reg.exec(str);
alert(reg.ignoreCase);
```

reg09.html
(Auszug)

test(str)

Die Methode `test()` prüft, ob das Muster im String `str` vorkommt. Ist dies der Fall, wird `true` zurückgeliefert, ansonsten `false`. Diese Methode ist also ähnlich wie `exec()`, mit dem Unterschied, dass `exec()` wesentlich mehr Informationen zurückliefert. Dafür ist `test()` schneller.

14.2.3 Die String-Klasse im Zusammenhang mit regulären Ausdrücken

Die folgenden Methoden der `String`-Klasse können im Zusammenhang mit regulären Ausdrücken verwendet werden:

- `match(reg)`
- `replace(regexp, replaceStr)`
- `search(reg)`
- `split(reg)`

match(reg)

Der Ausdruck `str.match(reg)` ist gleichbedeutend mit `reg.exec(str)` (`str` ist ein beliebiger String). Der Unterschied liegt darin, dass `exec()` eine Methode der `RegExp`-Klasse ist, während `match()` zur `String`-Klasse gehört.

replace(regexp, replaceStr)

Mit `replace()` können Sie einen Teil eines Strings mit einem anderen String ersetzen. `replace()` wird als Methode eines `String`-Objekts aufgerufen. Das folgende Beispiel ersetzt `bbb` durch `xyz`:

```
reg10.html      var str = "aaa bbb ccc";
(Auszug)        var newstr = str.replace(/bbb/, "xyz");
                // newstr ist jetzt "aaa xyz ccc"
```

Das oben gezeigte Beispiel ersetzt bbb durch xyz nur einmal, auch wenn bbb mehrmals in dem String vorkommt. Wenn Sie bbb an allen Stellen ersetzen möchten, können Sie den Modifikator g verwenden:

```
var str = "bbb bbb bbb";
var newstr = str.replace(/bbb/g, "xyz");
// newstr ist jetzt "xyz xyz xyz"
```

Interessant ist, dass Sie im zweiten Argument mit \$1, \$2 usw. auf die Teilstrings, die Sie mit Klammerausdrücken gefunden haben, zugreifen können. Das folgende Beispiel dreht die Reihenfolge der ersten drei Wörter um:

```
reg11.html      var str = "eins zwei drei";
(Auszug)        var newstr = str.replace(/^(.+) (.+) (.+)/, "$3 $2 $1");
                // newstr ist jetzt "drei zwei eins"
```

search(reg)

Der Ausdruck `str.search(reg)` ist gleichbedeutend mit `reg.test(str)`. Der Unterschied liegt darin, dass `test()` eine Methode des `RegExp`-Objekts ist, während `search()` zur `String`-Klasse gehört.

split(reg)

Die `split()`-Methode der `String`-Klasse haben wir bereits kennengelernt (siehe *split()*, S. 106). Als Argument kann nicht nur ein einfacher String angegeben werden, sondern auch ein regulärer Ausdruck. Das folgende Beispiel trennt den String an jedem Whitespace:

```
reg12.html      var str = "Dies ist ein String,\n" +
(Auszug)        "der sich ueber mehrere\n" +
                "Zeilen erstreckt";

                var erg = str.split(/\s+/);

                // Ausgabe des Arrays
                alert(erg.join(" - "));
```

Die Teilstrings, die in diesem Beispiel auf den regulären Ausdruck passen (d.h. die Whitespaces), sind nicht im Array `erg` enthalten. Wenn Sie innerhalb des regulären Ausdrucks jedoch Klammerausdrücke verwenden, werden die mit diesen Klammerausdrücken gefundenen Teilstrings im zurückgelieferten Array mit einbezogen.

14.3 Alternativen

Reguläre Ausdrücke sind ein hilfreiches Werkzeug. Zugegebenermaßen ist es nicht immer ganz einfach, einen regulären Ausdruck zu definieren. In diesem Abschnitt soll kurz gezeigt werden, wie sich Überprüfungsrouitinen und ähnliche Skripte ohne Verwendung von regulären Ausdrücken schreiben lassen.

14.3.1 Nach einem bestimmten Teilstring suchen

Wenn es nur darum geht zu überprüfen, ob ein bestimmtes Zeichen oder eine bestimmte Zeichenfolge in einem String vorkommt, kann die Methode `indexOf()` verwendet werden. Zum Beispiel kann man mit

```
if (eingabe.indexOf("@") != -1) alert("OK!")
else alert("Keine gueltige E-Mail-Adresse.");
```

pruefen1.html
(Auszug)

feststellen, ob in dem String `eingabe` das `@`-Zeichen vorkommt. Ist dies nicht der Fall, liefert `indexOf()` den Wert `-1` zurück. Mit dieser `if`-Abfrage kann man also eine einfache Überprüfungsroutine für E-Mail-Adressen realisieren.

14.3.2 Einen String auf bestimmte Zeichen beschränken

Als Nächstes soll ein allgemeines Skript gezeigt werden, das kontrolliert, ob ein String ausschließlich aus bestimmten Zeichen besteht. Mit solch einem Skript kann man beispielsweise überprüfen, ob eine Telefonnummer ungültige Zeichen enthält. Normalerweise besteht eine Telefonnummer aus den Ziffern 0 bis 9. Neben dem Leerzeichen sind außerdem die folgenden Sonderzeichen `- + / . , ()` üblich.

```
<html>
<head>
<title>indexOf()</title>
<meta http-equiv="Content-Script-Type"
  content="text/javascript" />
<script type="text/javascript">
function pruefen(eingabe, erlaubt) {
  var korrekt = true;
  for (var i = 0; i < eingabe.length; i++) {
    var zeichen = eingabe.charAt(i);
    if (erlaubt.indexOf(zeichen) == -1)
      korrekt = false;
  }
  return korrekt;
}
```

pruefen2.html

```
function test(eingabe) {
  if (!pruefen(eingabe, "0123456789 -+/,()")) {
    alert("Eingabe nicht korrekt.");
  } else {
    alert("Eingabe ok!");
  }
}

</script>
</head>

<body>
  <form>
    <p>
      Telefon:
      <input type="text" name="Telefon" value="" />
      <input type="button" value="Ueberpruefen"
        onclick="test(this.form.Telefon.value)" />
    </p>
  </form>
</body>
</html>
```

Die Funktion `pruefen()` ist dafür zuständig, den Eingabestring auf ungültige Zeichen zu überprüfen. Als erstes Argument wird der Funktion der Eingabestring übergeben. Das zweite Argument ist ein String der erlaubten Zeichen. Dort sehen Sie die Auflistung der Ziffern 0 bis 9 und die Sonderzeichen, die wir zulassen möchten. Die Funktion `pruefen()` verwendet nun `indexOf()`, um festzustellen, ob die einzelnen Zeichen des Eingabestrings in dem String der zugelassenen Zeichen vorkommen. Wird kein unerlaubtes Zeichen gefunden, liefert `pruefen()` den Wert `true` zurück, ansonsten `false`.