

Programmieren lernen mit JavaScript

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

Über dieses Buch

Programmieren lernen – das Richtige für dich?

Für wen ist dieses Buch geeignet – und für wen ist es ganz besonders geeignet? Natürlich geht es ums Programmieren, um echtes Programmieren mit Spielen und einer ganzen Menge Spaß. Ist dieses Buch also das Richtige für dich?

Beschäftigt man sich zum ersten Mal mit dem Thema »Programmieren«, scheint es sich eher um eine geheime Kunst zu handeln, die nur wenige Eingeweihte beherrschen. Mysteriöse Begriffe schwirren umher, das meiste scheint kaum verständlich. Vielleicht brummt dir der Kopf schon nach den ersten Versuchen? Zeit für dieses Buch!

Bin ich hier richtig?

Es geht in diesem Buch ums **Programmieren und darum, wie du deine eigenen Ideen in Programme umsetzen kannst**. Du lernst also nicht nur die Befehle und Anweisungen einer Programmiersprache, sondern ich zeige dir in diesem Buch auch, wie du deine Ideen sinnvoll formulierst und dann als Computerprogramm schreibst.

In (fast) jedem Kapitel stelle ich ein kurzes, meist einfaches Spiel vor. Wie musst du vorgehen, um dieses Spiel in ein Programm umzusetzen? In jedem Kapitel lernst du außerdem ein paar neue Befehle und Möglichkeiten der Programmierung kennen. Büffeln musst du dabei eigentlich gar nichts – wichtiger ist der Spaß daran.

Wenn du also ein Buch für einen durchaus **ernsthaften Einstieg** in die Welt der Programmierung suchst, aber auch dem Spaß nicht vollends abgeneigt bist, dann bist du hier richtig. Du willst das Programmieren für dich selbst lernen? Du suchst Hilfe für den Informatikunterricht oder die letzte Rettung für begleitende Informatikkurse im Studium oder in der Ausbildung? Wenn es dabei ums Programmieren geht und es JavaScript sein darf, bist du hier richtig.

Komplexe UIs unter Verwendung komplexer Frameworks?

Du kannst bereits programmieren? Du willst ein komplexes User Interface (UI) für das Frontend deiner Website bauen? Du suchst ein Buch über Oberflächen- und Webseitenprogrammierung mit JavaScript, jQuery und anderen Frameworks wie AngularJS oder

React? Du willst per AJAX asynchrone Datenübertragung realisieren? Dann ist dieses Buch nicht die erste Wahl – es sei denn, du hast gerade erst angefangen zu programmieren und möchtest noch etwas üben, ohne viele Themen auf einmal zu beginnen. Denn die üblichen Themen klassischer JavaScript-Bücher wie Web- und Oberflächenentwicklung, die sich an **erfahrene Entwickler** richten, aber nicht zu einem Einstieg in die Programmierung taugen, werden hier nur am Rande behandelt. Hier geht es um deinen **Einstieg** in die Programmierung.

Programmieren ist doch Mathematik, viel Mathematik?

Es ist ein Irrglaube, dass gute (oder sogar sehr gute) Mathematikkenntnisse eine zwingende Voraussetzung seien, um programmieren zu können. Sicher, ein gesundes Grundverständnis von Zahlen und allgemeinen Rechenoperationen schadet zumindest nicht. Aber das ist es eigentlich auch schon. Berechnungen führt der Computer durch. Notwendige Formeln findest du zumeist im Internet. Auch wenn du kein mathematisches Genie sein solltest, wirst du dennoch in der Lage sein, Programme zu schreiben. Viel wichtiger sind gute Ideen, etwas Ausdauer und Spaß an kniffligen Rätseln und Aufgaben.

Und wenn du erst einmal erlebt hast, wie der Computer mathematische Probleme und Schwierigkeiten erleichtert und knackt, macht dir Mathematik gleich doppelt so viel Spaß.

Was du brauchst

Du benötigst nicht viel. Hast du einen Computer, der schnell genug ist, um damit im Internet zu surfen, oder mit dem du Textverarbeitung machen kannst? Du hast einen Browser wie Chrome, Safari oder Firefox? Du weißt, wie du am Computer Texte schreibst und speicherst? Dann hast du eigentlich schon alles, was du brauchst. Es ist übrigens egal, ob du einen PC mit Windows oder Linux verwendest oder einen Mac. Was ich dir hier zeigen werde, kannst du tatsächlich **überall einsetzen**.

Warum JavaScript?

Natürlich ist es nicht egal, mit welcher Programmiersprache du beginnst. Wir nehmen deshalb eine Sprache, die Ähnlichkeiten mit den Großen der Szene hat, also mit C/C++, Java, C# und PHP, eine Programmiersprache, die selbst eine bekannte Größe ist und ursprünglich für das Internet entwickelt wurde.

Aber warum ist JavaScript denn so toll, um das Programmieren zu lernen?

Keine Installation – keine Konfiguration – keine Sorgen

JavaScript ist auf nahezu **jedem Computer** verfügbar, denn ein Browser, der JavaScript ausführen kann, gehört heute zur Grundausstattung aller gängigen Betriebssysteme.

So toll andere Programmiersprachen auch sind: Du musst erst eine Laufzeitumgebung oder einen Compiler installieren, einen leistungsfähigen Editor oder noch besser eine komplexe IDE. Eine Menge Wissen und **Vorarbeit** sind notwendig, um nur die kleinsten Dinge auf den Bildschirm zu zaubern. Und solange du beispielsweise nicht weißt, was `public static void main(String[] args){}` bedeutet, kannst du nur wenig machen. JavaScript ist da anders, ganz anders: Du schreibst deine JavaScript-Programme in einfache Webseiten (und die sind eigentlich auch nur einfache Textdateien) und schaust dir das Resultat direkt im Browser (also in Google Chrome, Edge, Firefox oder Safari) an. Die Browser haben einen eingebauten Übersetzer, einen *Interpreter* für JavaScript, der deine Programme wie von Zauberhand übersetzt und ausführt.

Das heißt für dich: Keine Suche nach Software, keine Installation, keine Ärgernisse, weil irgendwelche Systemeinstellungen oder Berechtigungen gesetzt werden müssen.

JavaScript ist sicher

Du nutzt den teuren (oder treuen) Laptop der Familie oder das Arbeitsgerät von Onkel Oswald? Kein Problem: JavaScript kann **nichts kaputtmachen**. JavaScript kann keine Dateien löschen oder verändern, die sich auf dem Computer befinden. Alles läuft in einer sicheren (sogenannten) *Sandbox* ab. Damit kein schädliches Programm aus dem Internet Schaden am Computer anrichten kann, hat man bei der Entwicklung entschieden, dass JavaScript bestimmte Dinge eben nicht machen darf – dazu gehört das Löschen oder Verändern von Dateien. Der einfachste Weg, solche Aktionen zu verhindern, war, diese Fähigkeiten erst gar nicht einzubauen. JavaScript »lebt« quasi innerhalb der Browser – und nur und **ausschließlich** dort. Natürlich kannst du trotzdem Daten speichern, aber eben geschützt innerhalb des Browsers.

JavaScript ist eine dynamisch typisierte Sprache

JavaScript gehört zu den dynamisch typisierten Sprachen. Kling kompliziert, ist aber ganz einfach: Eine Variable (das ist ein kleiner Speicher) kann beliebige Werte aufnehmen – egal, ob es eine Zahl ist, ein Wort oder ein Wahrheitswert (0 oder 1 bzw. `true` oder `false`). Treffen solche unterschiedlichen Werte aufeinander, kümmert sich JavaScript darum, alles korrekt zusammenzubringen. Du brauchst dir über solche Dinge keine Gedanken zu machen – du kannst dich gleich ins Programmieren stürzen.

Andere Sprachen wie Java oder C# sind da wesentlich strenger: Eine Variable muss vor der Verwendung dauerhaft auf eine Art von Wert festgelegt werden, sonst gibt es einen Fehler. Willst du eine Zahl und einen Text in irgendeiner Form zusammenbringen, musst du die Werte erst konvertieren. Das ist statische Typisierung. Das bringt bei (sehr) großen Programmen mehr Sicherheit, die Entwicklung ist aber aufwendiger.

Entwickler*innen führen immer wieder lange Diskussionen, ob die dynamische oder die strenge Typisierung der Weg zum Glück sind. Das braucht dich aber nicht zu kümmern.



Falls du es genauer wissen willst: Interpretiert oder kompiliert

Mit Programmen, wie du sie schreibst, kann ein Computer noch nicht viel anfangen – sie müssen erst in sogenannte *Maschinensprache* übersetzt werden. Manche Sprachen werden *interpretiert*, manche *kompiliert*. Der Unterschied ist einfach: Mit Sprachen, die kompiliert werden, können direkt ausführbare Programme generiert werden. Unter Windows sind das die klassischen *.exe*-Dateien, die mit einem Doppelklick gestartet werden. Interpreter-Sprachen werden jedes Mal aufs Neue aus dem Quellcode (also der Textdatei, in der du dein Programm geschrieben hast) übersetzt und dann ausgeführt. Man hat keine (echten) ausführbaren Dateien, sondern benötigt immer einen *Interpreter* bzw. eine sogenannte *Laufzeitumgebung* für die Umsetzung und Ausführung. Für JavaScript übernimmt der Browser diese Aufgabe: Er hat einen solchen JavaScript-Interpreter an Bord.

Eine klassische Compilersprache ist C/C++. Interpreter-Sprachen sind PHP, Python, Ruby und natürlich JavaScript. Und es gibt Sprachen, die zwar kompiliert werden – in sogenannten *Bytecode* –, aber erst bei der Ausführung des Programms in Maschinensprache interpretiert werden. Dazu gehören Java, C# oder auch VB.NET.

Früher war es einfach: Kompilierte Programme waren viel schneller. Es war gar nicht sinnvoll, Programme **nicht** zu kompilieren. Heute hat sich das geändert. Die Prozessoren sind derart schnell, dass sie sich meistens sowieso langweilen und eine Warteschleife nach der anderen drehen. Muss ein Programm vor dem Start erst noch übersetzt werden? Was soll's, der Prozessor macht das nebenbei. Im Moment der Übersetzung kann das Programm unter Umständen sogar »just in time« noch einmal etwas besser optimiert werden. Interpretierte Sprachen gelten als flexibler, die Entwicklungszeit ist kürzer. Auch sind diese Sprachen weniger streng. Der Nachteil: Man muss immer einen Interpreter oder eine entsprechende Laufzeitumgebung haben – im Fall von JavaScript ist das problemlos, da jeder Browser einen Interpreter zur Verfügung stellt. Ihre Berechtigung haben alle Programmiersprachen, und den besten Weg gibt es gar nicht – wie so oft.

Für Interessierte: Die Geschichte

Anfangs wurde JavaScript nur für kleine, sogar sehr kleine Aufgaben verwendet: die Überprüfung von Formulareingaben oder die Ausgabe von Meldungen und Warnhinweisen. Und tatsächlich fristete JavaScript anfangs ein Schattendasein als langsames und hässliches Entlein.

Es war einmal eine dunkle Zeit

Das war die Zeit der Browserkriege. Ohne dabei in die neuere Geschichtsschreibung einsteigen zu wollen: Die Firmen Netscape und Microsoft stritten dereinst erbittert um die Vorherrschaft im Internet und die Nutzerzahlen ihrer Browser *Netscape Navigator* und *Internet Explorer*. Netscape führte JavaScript ein, um Webseiten flexibler zu gestalten. Als Retourkutsche brachte Microsoft VBScript ins Spiel. Natürlich war alles inkompatibel miteinander. Da sich aber recht früh abzeichnete, dass niemand an JavaScript vorbeikam, hatte auch Microsoft JavaScript in seinen Browsern integriert. Leider mit genügend Eigenheiten und Besonderheiten, um doch wieder etwas Eigenes zu haben. Das führte dazu, dass Entwickler diese Besonderheiten bei den jeweiligen Browsern mit sogenannten »Browserweichen« berücksichtigen mussten. Für die eigenen Programme führte das an einigen Stellen zu doppeltem Code mit besonderen Anpassungen. Das war nervig, das war unschön und es trug natürlich nicht gerade dazu bei, den Ruf von JavaScript zu verbessern.

Und irgendwann wurde es hell, aber so was von

Das sollte aber nicht immer so bleiben. JavaScript wurde **schneller**, viel schneller, rasend schnell. Die Browser, die JavaScript in den Webseiten ausführen, erreichen heute Geschwindigkeiten, die im Vergleich zu früher unglaublich erscheinen. Komplexe Oberflächen, Animationen, dreidimensionale Darstellungen werden rasend schnell ausgeführt. JavaScript wurde auch **genormt** – unter dem seltsam klingenden Namen ECMAScript bzw. ECMA-262. Was langweilig und öde klingt, ist für Entwickler eine super Sache: Kein Hersteller eines Browsers kann es sich noch leisten, diesen Standard zu missachten, um sein eigenes Süppchen zu kochen.

Und dann kamen schließlich die **Frameworks**, die JavaScript für Entwickler endgültig vom hässlichen Entlein zum strahlend weißen Schwan mutieren ließen: Frameworks (das sind so etwas wie fertige Programmsammlungen für wiederholt auftretende Aufgaben) wie jQuery oder Angular sorgen dafür, dass die Entwicklung noch schneller geht und die (geringen) Unterschiede zwischen den Browsern kaum noch erwähnenswert sind.

JavaScript ist heute eine der am weitesten verbreiteten und am meisten verwendeten Programmiersprachen. Kein Wunder, denn JavaScript ist **praktisch überall**: Jeder Browser kann JavaScript ausführen und Browser sind bekannterweise auf jedem PC, Mac, iPhone, Smartphone oder Tablet vorhanden. Selbst auf Servern kommt JavaScript als Node.js zum Einsatz.

JavaScript und Java – nicht einmal verwandt

JavaScript und Java sind übrigens weder verwandt noch verschwägert. Erst recht ist Java nicht der große Bruder. Als JavaScript in den 1990er Jahren bei Netscape entwickelt wurde, hieß es noch *LiveScript*. Zu dieser Zeit war Java von der Firma Sun Microsystems eine ganz heiße Sache. An diesen Erfolg wollte man sich hängen (auch im Rahmen einer Kooperation mit Sun) und wählte einen bewusst ähnlichen Namen: **JavaScript**, kurz JS. Eine gewisse Verwandtschaft haben beide Sprachen aber dennoch: Sie haben als gemeinsamen Ursprung die Sprache C bzw. C++.

So (oder ähnlich) sieht eine Schleife aus, die zehnmal durchlaufen wird:

```

10-mal [ for ( i=0; i<10; i++ )
von hier ... [ {
                //Hier passiert etwas
... bis hier [ }
```

Alle Programmiersprachen, die sich an der Schreibweise der (fast schon uralten) Sprache C orientieren, sehen ähnlich aus. Natürlich gibt es auch andere Sprachen, zum Beispiel BASIC und seine »Verwandten«:

```

FOR i = 1 TO 10 STEP 1
    REM Hier passiert etwas
NEXT i
```

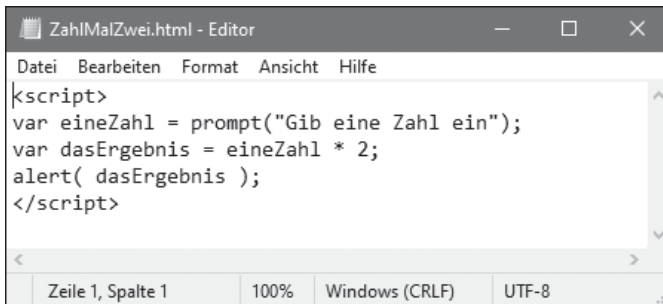
Man kann sich natürlich trefflich darüber streiten, welche *Syntax* (so nennt man die Schreibweisen und Regeln der Programmiersprachen) schöner oder besser ist – und tatsächlich wird auch immer wieder darüber gestritten. Gute Entwickler und Programmierinnen sollten aber mehrere Sprachen kennen und vorbehaltlos damit umgehen können.

Die Wahl der Waffen: Editor, Entwicklungsumgebung oder IDE?

Ähnlich sieht es bei der Frage aus, womit du deine JavaScript-Programme schreibst. In fast jeder Programmiersprache wird der sogenannte Quellcode in recht einfache Textdateien geschrieben.

Was war noch mal der Quellcode?

Quellcode sind die in einer Programmiersprache geschriebenen Befehle, die zusammen ein Programm ausmachen (vereinfacht gesagt). Das sieht zum Beispiel so aus:



```

Datei Bearbeiten Format Ansicht Hilfe
<script>
var eineZahl = prompt("Gib eine Zahl ein");
var dasErgebnis = eineZahl * 2;
alert( dasErgebnis );
</script>
Zeile 1, Spalte 1    100%    Windows (CRLF)    UTF-8
  
```

Abbildung 1 Ein einfaches Programm in JavaScript als Quellcode – du kannst alle Befehle sehen und lesen.

Das ist ein einfaches Programm in JavaScript, das nach einer Zahl fragt, sie mit 2 multipliziert und das Ergebnis in einem kleinen Fenster ausgibt. Quellcode wird in ganz einfachen Textdateien gespeichert. Die meisten Programmiersprachen benutzen dafür jeweils eigene Dateierendungen. Bei PHP zum Beispiel *dateiname.php*, bei Java ist es *dateiname.java* und bei JavaScript *dateiname.html* oder *dateiname.js*. Diese Dateien sind die Grundlage für jedes laufende Programm. Bei manchen Programmiersprachen wird dieser Quellcode in ausführbare Programmdateien (in sogenannte *Maschinensprache*) übersetzt und dauerhaft gespeichert (kompiliert). Bei manchen Programmiersprachen ist der Quellcode immer auch das Programm, das ausgeführt (interpretiert) wird. Dazu gehört auch JavaScript.

Wichtig ist: Du brauchst einen Texteditor, um deine Programme bzw. den Quellcode zu schreiben und zu bearbeiten. Für den Anfang reicht ein ganz einfaches Exemplar. So ein Editor ist – genauso wie ein Browser – auf jedem Rechner vorhanden (von Smartphones und Tablets einmal abgesehen). Unter Linux kann das *Leafpad*, auf einem Mac *TextEdit* und unter Windows *Editor* oder *Notepad* sein. Allen Editoren gemeinsam ist, dass es sich um standardmäßig vorhandene Programme handelt, um einfache und vor allem leicht zu bedienende Programme.

Was du **nicht verwenden** solltest, sind **Textverarbeitungsprogramme** wie *Microsoft Word*, *LibreOffice Writer* oder *Write*. Der Quellcode muss in einfache Textdateien geschrieben werden. Es dürfen **keine Formatierungen** wie fett, kursiv, Schriftgrößen oder Schriftarten in der Datei vorhanden sein. Passt du beim Speichern nicht auf, werden diese Merkmale mitgespeichert. Versucht der Computer nun, das zu lesen, kann er mit diesen zusätzlichen Informationen nichts anfangen – im Gegenteil, er »stolpert« darüber und kann gar nicht mehr weiterarbeiten.

Editoren, IDEs und ein zauberhafter Geany

Natürlich gibt es auch spezielle Editoren, die mehr als nur die absoluten Grundfunktionen bieten. Viele programmieren oft mit komplexen Entwicklungsumgebungen, sogenannten IDEs (Integrated Development Environment). Dabei handelt es sich um recht komplizierte Programme, die dafür sehr viele Funktionen und Komfort bieten. Gerade für den Anfang kann es aber frustrierend sein, gleichzeitig das Programmieren an sich, eine Programmiersprache **und** die Bedienung einer komplexen IDE lernen zu wollen. Du solltest deshalb anfangs einen ganz einfachen Editor verwenden oder zu einem einfach zu bedienenden (besseren) Editor wie Geany greifen.

Geany ist ein leistungsfähiger Texteditor, den du von der Site www.geany.org kostenlos herunterladen kannst. Geany ist für Linux, Mac und natürlich auch Windows verfügbar. Es spielt also keine Rolle, was für einen Rechner du zur Verfügung hast.



Abbildung 2 Gestatten? Geany. Recht schick für einen einfachen Texteditor.

Geany ist fast so leicht zu bedienen wie ein einfacher Editor. Die zahlreichen zusätzlichen Funktionen sind recht unaufdringlich, die Menüs nicht überladen. Verstellen kannst du auch nicht allzu viel – es sei denn, du gibst dir richtig Mühe. Geany ist natürlich auch auf Deutsch verfügbar. Das ist in der Welt der Editoren und Entwicklungsumgebungen nicht selbstverständlich. Ach, schnell (sehr schnell) ist Geany übrigens auch.

Viele entwickeln übrigens mit einer großen IDE und haben für alles, was »mal eben schnell« erledigt werden muss, noch einen einfachen Editor in Verwendung. Manchmal ist man mit diesen Editoren schon fertig, bevor die oft doch etwas träge IDE überhaupt gestartet ist und die jeweilige Datei geladen hat. Geany ist so ein treuer Begleiter, der dich auch zukünftig begleiten könnte.

Where's the beef?

So sieht eine Webseite mit JavaScript in einem einfachen Editor aus:

```

*BubbleSort.html - Editor
Datei Bearbeiten Format Ansicht Hilfe
<body>
  <p id="notiz">Hallo Welt</p>
  <script>
    document.getElementById("notiz").innerHTML =
      "Die Ergebnisse von Bubblesort<br>";
    var mannschaft =
      ["Xaver", "Hans", "Darth Berti", "Helge", "Bärbel", "Andrea"];
    sortiere(mannschaft);
    function sortiere(liste, position, geaendert) {
      if (position == undefined) {
        position = 0;
      }
      if (geaendert == undefined) {
        geaendert = false;
      }
      for (var i = position; i < liste.length - 1; i++) {
        if (liste[i] > liste[i + 1]) {
          kurzMerken = liste[i];
          liste[i] = liste[i + 1];
          liste[i + 1] = kurzMerken;
          geaendert = true;
        }
      }
    }
  </script>
</body>
Zeile 14, Spalte 2 100% Windows (CRLF) UTF-8

```

Abbildung 3 Eine ganz einfache Seite mit ganz einfachem JavaScript in einem ganz einfachen Standardeditor

Und so wie in Abbildung 4 sieht die gleiche Seite mit Geany aus.

Für Abbildung 4 habe ich bewusst diesen etwas abschreckenden, aber prägnant schwarzen Hintergrund gewählt. Tatsächlich gibt es eine Vielzahl von **Farbschemata**, aus denen du dir eine passendere Darstellung auswählen kannst.

Davon abgesehen bietet Geany eine Vielzahl an Funktionen:

- Das Interessanteste ist sicherlich die farbliche Darstellung des Quellcodes. Das nennt man *Syntax-Highlighting*. Befehle der Sprache, Variablen und Sonderzeichen werden jeweils in einer anderen Farbe präsentiert. Der Code ist übersichtlicher und Schreibfehler sind schnell zu erkennen (falsch geschriebene Befehle werden nämlich erst gar

nicht farblich markiert). Dabei erkennt Geany eine Vielzahl von Programmiersprachen automatisch an der Dateieindung und verwendet eine korrekte Darstellung.

- ▶ Du kannst **beliebig viele Dateien** gleichzeitig geöffnet haben und mithilfe von Karteitern (Tabs) über dem Textfenster aufrufen.
- ▶ Geany beherrscht *Code Folding*. Das heißt, zusammengehörende Programmteile erkennt Geany problemlos. Und über kleine quadratische Kästchen am Rand kannst du diese Teile zusammenklappen und auch wieder aufklappen. Damit wird auch das längste Programm wieder **übersichtlich**.
- ▶ Über fertige *Plug-ins* lässt sich Geany um weitere Fähigkeiten erweitern. Bei der normalen Installation ist bereits ein ganzer Schwung gleich mit dabei, beispielsweise ein Dateibrowser, und sogar HTML-Sonderzeichen kannst du dir dank eines Plug-ins automatisch einfügen lassen.
- ▶ Eine Webseite mit JavaScript kannst du mit einem Klick auf das Icon AUSFÜHREN direkt im Browser öffnen lassen. Einfacher geht es kaum.
- ▶ Geany kann aber noch **wesentlich mehr**, ein Versuch lohnt sich auf jeden Fall.

```

1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <title>Bubblesort</title>
6    </head>
7    <body>
8      <p id="notiz">Hallo Welt</p>
9      <script>
10     document.getElementById("notiz").innerHTML =
11       "Die Ergebnisse von Bubblesort<br>";
12     var mannschaft =
13       ["Xaver", "Hans", "Darth Berti", "Helge", "Bärbel", "Andrea"];
14     sortiere(mannschaft);
15     function sortiere(liste, position, geaendert) {
16       if (position == undefined) {
17         position = 0;
18       }
19       if (geaendert == undefined) {
20         geaendert = false;
21       }
22       for (var i = position; i < liste.length - 1; i++) {
23         if (liste[i] > liste[i + 1]) {

```

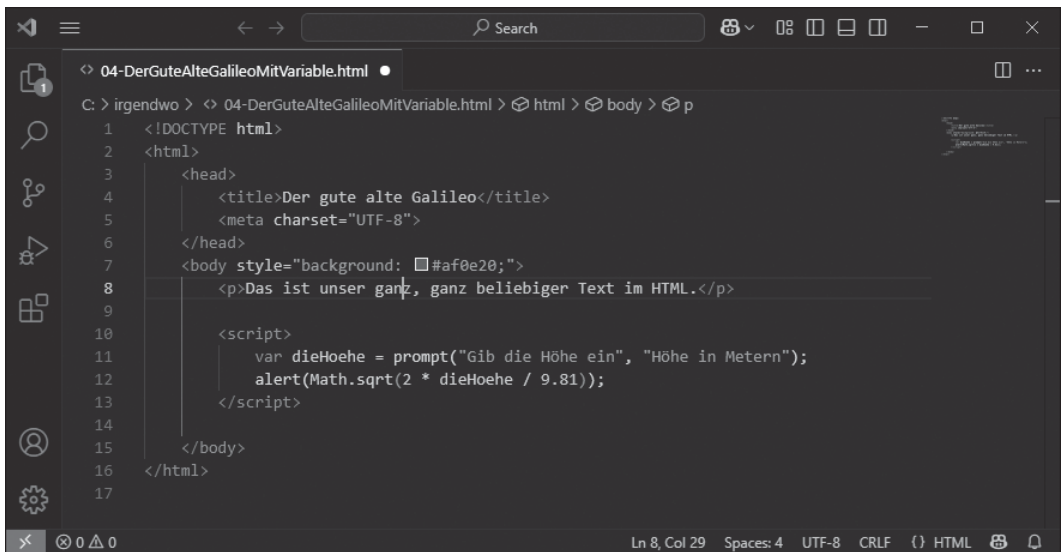
Zeile: 1 / 41 Spa: 0 Aus: 0 EINFUG Tab EOL: CRLF Kodierung: UTF-8 Dateityp: HTML Kontext: unbekannt

Abbildung 4 Immer noch die gleiche, einfache Seite mit einfachem JavaScript, hier extra mit einem auffälligem Farbschema, aus der großen Auswahl von Geany

Aber keine Sorge: Auch mit einem ganz einfachen Editor wirst du alles machen können und das Programmieren in JavaScript erlernen.

Gibt es sonst noch was?

Wie gesagt gibt es eine Vielzahl sehr guter Editoren für jedes Betriebssystem. Eines der Programme, die für Windows, Linux und Mac verfügbar sind, ist *Visual Studio Code* von Microsoft. Dieser (wie Geany kostenfrei verfügbare) Editor ist einfach zu bedienen, bietet aber zahlreiche Funktionen und eine Menge praktischer Gimmicks. Kryptische Farbangaben im HTML-Code werden beispielsweise mit einem passenden Farbkästchen im Quelltext dargestellt. Syntax-Highlighting und Code Folding kennt Visual Studio Code natürlich auch.



```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Der gute alte Galileo</title>
5     <meta charset="UTF-8">
6   </head>
7   <body style="background: #af0e20;">
8     <p>Das ist unser ganz, ganz beliebiger Text im HTML.</p>
9
10    <script>
11      var dieHoehe = prompt("Gib die Höhe ein", "Höhe in Metern");
12      alert(Math.sqrt(2 * dieHoehe / 9.81));
13    </script>
14  </body>
15 </html>
16
17

```

Abbildung 5 Schwarz ist das neue Weiß: Du findest Visual Studio Code zum Herunterladen unter »<https://code.visualstudio.com>«.

Welchen PC-Boliden brauche ich zur Entwicklung?

Jeder halbwegs aktuelle PC mit einer aktuellen Version von Windows oder Linux ist für die Entwicklung geeignet – genauso natürlich ein Mac. Wenn der Rechner für die normalen Alltagsaufgaben schnell genug ist, dann genügt er mit Sicherheit für die Entwicklung mit JavaScript. Sogar mit einem Tablet oder einem Smartphone wäre das Erlernen von JavaScript (und das Programmieren) möglich. Du brauchst allerdings eine App, die dir als Texteditor dient. Dafür eine Empfehlung zu geben, ist etwas problematisch, da

sich hier erfahrungsgemäß recht schnell etwas ändern kann. Eine empfohlene App ist unter Umständen mit Erscheinen dieses Buches gar nicht mehr verfügbar. Zum Vergleich: Geany ist seit 2005 erhältlich und wird wohl auch noch anno 2035 aktuell sein.

So einfach kann der Einstieg sein

Willst du anfangen zu programmieren, brauchst du also nur **dieses Buch**, einen **Computer** und das grundlegende Wissen, wie du an einem Computer Texte schreibst, bearbeitest (editierst), speicherst und wieder öffnest. Es gibt **keine Installationsorgien** und keine Probleme mit Rechten bei der Ausführung (oder einer notwendigen Installation). Und du musst dich nicht in irgendwelche Frameworks oder Entwicklungsumgebungen einarbeiten, bevor du überhaupt nur eine Zeile Code schreiben kannst. Einfacher geht es nicht.

Die Codebeispiele in diesem Buch

Den Code für die Spiele aus diesem Buch kannst du auf dieser Webseite herunterladen: www.rheinwerk-verlag.de/6107. Scrolle etwas herunter zu den MATERIALIEN; über den Button geht es zum Download.

Im Buch sind die Programme oft nicht am Stück abgedruckt, sondern Codezeilen und Erklärungen wechseln sich ab. Falls du sie lieber am Stück anschauen möchtest, nimm dir den heruntergeladenen Code vor. Um Programmieren zu lernen, ist es aber von Vorteil, Programme selbst zu einzutippen.

Nimm noch ein paar Weisheiten mit auf den Weg

Du wirst sie mit Sicherheit auch erleben: mehr oder (oft) weniger ernsthafte Diskussionen über die Programmierung. Und natürlich wird immer der eine oder die andere dabei sein, die dich überzeugen wollen, dass seine Programmiersprache, sein Editor oder ihre Programmieretechnik eben doch die bessere, schnellere oder schönere ist.

Programmieren ist Pragmatismus

Zum Programmieren gehört eine gehörige Portion Pragmatismus. Oder um es anders zu sagen: Schönheit löst keine Probleme. Wenn eine Lösung in der Programmierung funktioniert und sie keine wesentlichen Nachteile mit sich bringt, dann ist sie in Ordnung. Das bedeutet natürlich nicht, dass man unsauber oder schlampig programmieren darf oder sollte – es heißt nur, dass eine funktionierende Lösung eine gute Lösung ist und durchaus von gewohnten Lösungen abweichen darf.

Das KISS-Prinzip – Keep It Simple, Stupid

Die amerikanische Navy hat es sich in den 1960er Jahren zum erfolgreichen Grundsatz gemacht, der in ähnlicher Form auch in der Programmierung Anwendung findet: Eine einfache Lösung ist einer komplexen, umständlichen Lösung vorzuziehen – KISS (*Keep It Simple, Stupid*). Das hat einen ganz plausiblen Grund: Einfacher Code hat meist weniger versteckte Fehler, ist einfacher zu lesen und einfacher zu warten. Die Kunst besteht nicht darin, komplexe, schwer lesbare Programme zu schreiben, um anderen zu zeigen, wo der syntaktische Hammer hängt. Gute Programmierer*innen schaffen es, schwierige Aufgaben mit einfachen, gut lesbaren Programmen zu lösen – das ist die wahre Kunst.

Welche Sprache ist die beste?

Die Frage, welche Programmiersprache die beste ist, ist zwar nicht so alt wie die Menschheit, aber sicherlich so alt wie die Programmierung. Keine Programmiersprache ist für alle Anwendungsfälle gleichermaßen geeignet. C# oder Java eignen sich für umfangreiche Business-Software. PHP und JavaScript sind Sprachen, mit denen du sehr schnell entwickeln kannst. Für viele kleinere Programme und Anwendungen im Web, für ausgefeilte Oberflächen im Browser ist JavaScript aber alternativlos – es gibt keine andere Sprache, mit der Weboberflächen im Browser realisiert werden können. JavaScript ist hier der röhrende Platzhirsch.

Aber lass uns anfangen. Wir starten mit Webseiten und etwas HTML: Das ist das notwendige Zuhause von JavaScript.

Kapitel 4

CodeBreaker

Knack den Code von Mr. JS

Du hast jetzt schon einige Programme geschrieben und eine Menge Befehle kennengelernt. Mit diesem Wissen machst du jetzt den Computer zum gerissenen Superhirn, dessen Codes vom Spieler in kürzester Zeit geknackt werden müssen. Und da der Mensch nicht nur von Luft und fertigen Programmen leben will, wirst du das fertige Programm danach noch weiter verbessern.

In diesem Kapitel ...

... wirst du einige Abwandlungen und Besonderheiten der bereits bekannten Anweisungen kennenlernen. Es gibt nicht nur `if` und ein zugehöriges `else` – mit einem `else if` kann dein Programm noch komplexere Entscheidungen fällen. Du lernst Funktionen kennen, mit denen du Teile deines Programms aufteilen und einfach und beliebig oft wiederverwenden kannst. Und du beginnst auch, dein Programm mithilfe von HTML (und CSS) etwas aufzuhübschen. Denn HTML, das sind nicht nur schnöde Webseiten, sondern das ist auch die grafische Oberfläche für JavaScript.

Kennst du Spiele wie »Mastermind« oder »Superhirn«? Nenn es einfach »CodeBreaker«. Jemand denkt sich eine **mehrstellige Zahl** oder einen Code aus, der mit farbigen Stiftchen verdeckt abgelegt wird. Der Spieler (oder der Gegner, wie man es auch sehen möchte) muss nun diesen **Code erraten**, der vom mysteriösen »Mister JS« erstellt wurde. Und auch wenn unser Mister JavaScript eigentlich gar nicht so mysteriös ist, wird es gar nicht so leicht, den geheimen Code zu knacken.

Schweiß tropfte von ihrer Stirn. Sie musste den Code knacken und das Rätsel lösen – koste es, was es wolle. Diese Sache war zu wichtig. Mit zitternden Händen tippte sie den Code in die Tastatur und wartete auf das Ergebnis: Na, das sah doch schon mal nicht schlecht aus. Rasend schnell tippte sie eine neue, leicht veränderte Kombination von Zahlen. Sie musste den Computer besiegen, und zwar schnell.

Blieben wir bei den Zahlen: Der Code ist also mehrstellig und jede Stelle kann eine Zahl von 1 bis 9 sein – ganz klassisch wie in einem Agentenfilm. Die Null lassen wir einfach außen vor; schließlich ist es so schon schwer genug, das Rätsel zu lösen.

Du gibst also einen Tipp ab – rätst einfach. Der geheimnisvolle Rätselgeber sagt dir nun, wie viele Zahlen du richtig hast und wie viele Zahlen zwar in dem Code vorkommen, aber **an einer falschen Stelle** stehen. Hast du alle Zahlen richtig **und** an der richtigen Stelle, ist das Rätsel gelöst, idealerweise natürlich möglichst schnell.

Hast du bei einem vierstelligen Code beispielsweise 3 – 8 – 5 – 9 ausprobiert und ist **keine Zahl** davon an der richtigen Stelle und kommt keine Zahl davon überhaupt in dem Code vor, dann weißt du bereits, dass du diese Zahlen **komplett streichen** kannst. Erfährst du bei einem Tipp von 1 – 4 – 2 – 7, dass alle Zahlen richtig sind, aber keine an der richtigen Stelle, dann musst du die Zahlen komplett umstellen, und zwar so lange, bis alle Stellen richtig sind und du den Code geknackt hast. Herzlichen Glückwunsch!

Lass uns den ersten Schritt für die Programmierung machen, beschreiben wir, was gemacht werden soll:

- ▶ Der Computer denkt sich einen Code aus. Damit es nicht zu schwer wird, sollen es **drei Stellen** mit Zahlen von **1 bis 9** sein.
- ▶ Wir raten dann eine Zahl.
- ▶ Der Computer soll uns jetzt ausgeben, wie viele unserer Zahlen an der richtigen Stelle stehen und wie viele unserer Zahlen in dem unbekanntem Code (an falscher Stelle) vorkommen.
- ▶ Solange wir den Code nicht vollständig geknackt haben, wollen wir erneut raten.

Die geheime Zahl

Eine nicht ganz unwichtige Frage, die wir uns selbst beantworten müssen, lautet: Wollen wir als geheimen Code **drei einzelne Zahlen** (und damit drei einzelne Stellen und Variablen) haben? Oder nehmen wir **eine große Zahl** (also eine Variable), die dann ebenso viele Stellen hat? Im Zweifelsfall (das lehrt die Erfahrung) ist es oft besser, Informationen möglichst **klein und einfach** zu halten, also jede Stelle für sich selbst zu erzeugen und in einer eigenen Variablen abzuspeichern.

Natürlich wäre es nicht falsch, eine große Zahl mit entsprechend vielen Stellen im Programm zu verwenden. In der Programmierung gibt es oft verschiedene Lösungen für ein Problem oder eine Aufgabe. Und es ist schwer zu sagen, was besser oder schlechter wäre. Hast du bereits eine andere, ähnliche Lösung aus einem anderen Programm, die du als funktionierende Vorlage nehmen kannst, dann solltest du das ruhig machen. Überlegst du dir, das Programm später zu erweitern, dann kann es sinnvoll sein, eine vermeintlich schwierigere (oder umständlichere) Lösung zu wählen. Das sind Erfahrungen, die du ganz schnell machen wirst – umso schneller, je mehr du programmierst.

Für eine Lösung müssen wir uns entscheiden: Wir nehmen die etwas einfachere mit mehreren einzelnen Zahlen und jeweils einer eigenen Variablen für jede Stelle des Codes.

Einfache Lösungen

Es ist in der Programmierung nicht verpönt, einfache Lösungen zu wählen. Gerade einfache Lösungen sind oft die besseren. Es zählt mehr, wie schnell etwas umgesetzt und auch von anderen Programmierer*innen gelesen und verstanden werden kann.

Anders sieht es mit der Zahl aus, die als **Tipp abgegeben** werden soll. Hier ist es tatsächlich sinnvoller (vor allem bequemer), mit nur einer (mehrstelligen) Zahl zu arbeiten – so können die Spieler ihren Tipp mit einer einzigen Eingabe abgeben. Andernfalls müssten sie die Eingabe dreimal machen und bestätigen – das wäre vielleicht einfacher zu programmieren, benutzerfreundlich ist aber etwas anderes.

Fangen wir an und nehmen als Grundlage für unser Programm wieder unsere Beschreibung, die wir wieder direkt in JavaScript umsetzen.

Von der Beschreibung zum Programm

Der Computer denkt sich also einen Code aus. Damit es nicht zu schwer wird, sollen es (erst einmal) drei Stellen mit Zahlen von 1 bis 9 sein.

Wir brauchen also drei Zufallszahlen, die wir jeweils in einer eigenen Variablen speichern. Das kann für Zahlen von 1 bis 9 dann so aussehen:

```
var zahl1 = Math.round( Math.random() * 9 + 0.5);
var zahl2 = Math.round( Math.random() * 9 + 0.5);
var zahl3 = Math.round( Math.random() * 9 + 0.5);
```

Wir haben wieder unsere klassische Funktion `Math.round` für Zufallszahlen, die wir wieder etwas aufbohren. Da wir in dem Spiel die Zufallszahlen nur einmalig am Anfang festlegen, können wir die Deklaration mit `var` direkt hier erledigen. Auch in diesem Fall gilt natürlich: Anders machen ist ausdrücklich erlaubt (zumindest, solange es funktioniert).

Da sich diese drei Zufallszahlen im Spiel nicht mehr ändern, wären sie auch gute Kandidaten, um als Konstanten *deklariert* zu werden:

```
const zahl1 = Math.round( Math.random() * 9 + 0.5);
const zahl2 = Math.round( Math.random() * 9 + 0.5);
const zahl3 = Math.round( Math.random() * 9 + 0.5);
```

Willst du später aber mehrere Spiele hintereinander ermöglichen – natürlich mit jeweils neuen Zahlen, bleibt die Deklaration mit `var` die bessere Lösung.

Wir raten dann eine Zahl.

```
var meinVersuch = prompt("Gib einen Tipp ab", "Zahl von 111 bis 999");
```

Wir deklarieren direkt bei der Verwendung und fragen mit einem `prompt` den Tipp des Spielers ab. Natürlich könnten wir die Deklaration getrennt und nur einmal machen – außerhalb der späteren Schleife. Das sähe dann so aus:

```
var meinVersuch;
//Hier fängt irgendeine Schleife an, und noch einiges an Code ist hier
meinVersuch = prompt("Gib einen Tipp ab", "Zahl von 111 bis 999");
```



Wenn du es genauer wissen willst: »var« vor oder in der Schleife?

Werden Variablen (wie hier) in Schleifen verwendet, kannst du dich (zu Recht) fragen, ob die Deklaration mit `var` getrennt **vor** der Schleife oder **in** der Schleife erfolgen sollte – direkt bei der ersten Verwendung. Eindeutige Antwort: Es spielt eigentlich keine Rolle. Die Deklaration erfolgt in jedem Fall nur einmal – es ist egal, ob `var` einmalig im Code vorkommt oder innerhalb einer Schleife immer wieder mit durchlaufen wird. Der JavaScript-Interpreter ist so intelligent, dass er das problemlos erkennt und sich unnötige Arbeit spart. Allein durch eine erneute Deklaration mit `var` (ohne eine Zuweisung) ändert sich eine Variable deshalb auch nicht mehr.

Anders verhält es sich bei `const`. Da sich eine einmal als Konstante deklarierte Variable nicht mehr ändern darf (sie ist ja schließlich eine Konstante), kann und darf diese Deklaration auch nur einmal erfolgen.

Der Computer soll uns dann ausgeben, wie viele unserer Zahlen an der richtigen Stelle stehen und wie viele unserer Zahlen überhaupt in dem unbekanntem Code vorkommen.

So eine Prüfung auf Übereinstimmung der Zahlen ist nicht schwer, wird aber etwas umfangreicher. Schließlich muss der Computer genau wissen, was er machen soll. Und die Antwort auf unsere Rateversuche sollte ja auch korrekt sein.

Zuerst benötigen wir zwei Variablen, in denen wir das Ergebnis der aktuellen Runde speichern. Irgendwo muss der Computer ja speichern, wie viele Zahlen richtig sind.

```
var richtigeStelle = 0;
var richtigeZahl = 0;
```

Wir müssen in jeder Runde erneut prüfen, wie viele Zahlen an der richtigen Stelle stehen und wie viele der angegebenen Zahlen richtig sind, aber an der falschen Stelle stehen. Deshalb deklarieren wir die Zahlen und setzen sie bei jedem Durchlauf durch die Zuweisung mit 0 auch wieder zurück. Ansonsten würde das Ergebnis doch »etwas« verfälscht, wenn unsere Variablen noch Werte aus der Runde davor hätten. Die dazugehörige Logik kommt gleich – erst brauchen wir ja einen aktuellen Tipp in Form der Eingabe des Spielers.

Wenn du es genauer wissen willst: implizite und explizite Deklaration



Ja, sie sind kein Mythos, es gibt sie tatsächlich, die **Fachbegriffe**. Manche werden dir häufiger begegnen, sodass ich ab und zu einige besprechen möchte.

Variablen können in JavaScript *implizit* und *explizit* deklariert werden. Was seltsam und kompliziert nach Nerd-Sprech klingt, ist einfach: Machst du **selbst** die Deklaration mit `var`, dann ist das *explizit*. Deklarierst du eine Variable **nicht**, dann kümmert sich der Interpreter von JavaScript **stillschweigend** darum, wenn du dieser Variablen einen Wert zuweist. Das nennt man dann *implizite Deklaration*.

Das ist so wie die Bestellung einer Kugel Eis: Auch wenn du nichts sagst, bekommst du eine Waffel zu der Eiskugel, auch **ohne explizit** darum zu bitten (ja, das Beispiel hinkt natürlich etwas, schließlich könntest du auch einen Becher bekommen).

Zahlen spalten einfach gemacht

Der Spieler gibt seinen Tipp – seine drei Zahlen – als **eine Zahl mit drei Stellen** ein. So muten wir ihm nicht zu, alle Zahlen einzeln einzugeben und abzuschicken. So eine nervige Kleinigkeit kann bei einem längeren Spiel schnell den Spaß verderben.

Nur, irgendwie müssen wir jetzt die **einzelnen Stellen** aus der eingegebenen Zahl in der Variablen `meinVersuch` herausbekommen. Wir müssen sie ja mit den einzelnen Zahlen (und Stellen) im Code vergleichen.

Einfache Lösung mit Hausmitteln

Wir könnten die einzelnen Stellen mithilfe komplizierter mathematischer Verfahren aus der ganzen Zahl herausrechnen. Oder aber wir sehen uns an, was JavaScript zu bieten hat. Und JavaScript hat tatsächlich einige sehr mächtige Funktionen und Möglichkeiten für die Arbeit mit Texten.

Moment, ich gebe doch eine Zahl ein und keinen Text!

Alles, was über `prompt` eingegeben wird, ist für JavaScript erst einmal ein Text. Selbst wenn der Inhalt eine Zahl ist – aus der Sicht von JavaScript ist eben auch das (erst einmal) ein Text. Was sich jetzt seltsam anhören mag, ist ein großer Vorteil: JavaScript bietet sehr viele Funktionen, mit denen gerade Text bearbeitet werden kann. So können wir sehr einfach Zeichen an einer angegebenen Stelle abfragen.

Der Befehl dazu heißt `charAt`, was so viel bedeutet wie **das Zeichen an Stelle** (wobei in den Klammern die gewünschte Stelle als Zahl angegeben wird). Du lernst damit auch eine neue Schreibweise kennen. Dieser Befehl wird nämlich mit einem Punkt an die jeweilige Variable angehängt:

```
var eineVariableMitText = "HALLO";
alert( eineVariableMitText.charAt(1) );
```

Das Ergebnis sieht so aus:



Abbildung 4.1 Moment, ein A? Sollte das nicht ein H sein? Nein, denn für den Computer ist die erste Stelle immer 0, nicht 1.



Falls du es genauer wissen willst: 0 ist die neue 1

Nein, neu ist das nicht. Bei Elementen, die der Computer **selbst nummeriert**, beginnt diese Nummerierung **in der Regel mit 0**, nicht mit 1. Die erste Stelle – zumindest, wenn der Computer das Sagen hat – ist also immer die Stelle 0. Deshalb liefert der Befehl `eineVariableMitText.charAt(1)` den zweiten Buchstaben zurück. Willst du den ersten Buchstaben, gibst du also einfach die 0 an: `eineVariableMitText.charAt(0)`. Das erscheint (aus menschlicher Sicht) nicht unbedingt logisch, ist es eigentlich auch gar nicht. Aus Sicht des Computers ist es aber einfach ökonomischer, die 0 nicht unbenutzt zu lassen, es soll ja nichts umkommen.

Jetzt weißt du also, wie du **eine bestimmte Stelle** aus der Eingabe herausbekommst:

1. `meinVersuch.charAt(0)` für die erste Stelle
2. `meinVersuch.charAt(1)` für die zweite Stelle
3. `meinVersuch.charAt(2)` für die dritte Stelle

Auf diese Art kannst du **jede Stelle** und damit jede Zahl des eingegebenen Tipps herausfinden und in einer Ausgabe mit `alert`, in einer Zuweisung oder in einem Vergleich verwenden.

Du kannst diese Anweisungen exakt so jedes Mal im Programm verwenden, wenn du eine der Stellen benötigst. Du könntest dir aber auch **alle** Stellen einmal holen und in **eigenen Variablen speichern**. Das macht für das Programm (und den Computer) keinen nennenswerten Unterschied. Das Programm wird dadurch aber besser lesbar. Deshalb werden wir das machen.

Wir holen uns einmalig die einzelnen Stellen des eingegebenen Tipps und speichern sie jeweils in eigenen Variablen ab:

```
var tipp1 = meinVersuch.charAt(0);
var tipp2 = meinVersuch.charAt(1);
var tipp3 = meinVersuch.charAt(2);
```

Wie erfolgreich war das Raten?

Wir haben jetzt alle Zahlen zur Verfügung. Es wird also Zeit, sich zu überlegen, wie wir eine Prüfung machen können: Was ist an der richtigen Stelle und was ist zwar an falscher Stelle, aber als Zahl richtig? Schwierig daran wird, sicherzustellen, dass Zahlen nicht mehrfach gewertet werden.

Du kannst auf jeden Fall die erste Stelle des geheimen Codes, also `zahl1`, mit der **ersten Stelle** aus deinem **Tipp**, `tipp1`, (oder auch mit `meinVersuch.charAt(0)`) vergleichen. Stimmen beide überein, vermerkst du eine richtige Stelle, indem du die Variable `richtigeStelle` um eins hochzählst:

```
if( tipp1 == zahl1 ){
    richtigeStelle++;
}
```

Nach dem gleichen Muster könntest du für die beiden anderen Stellen vorgehen.

Problematisch wird die Sache mit der Überprüfung, ob eine Zahl an einer falschen Stelle sitzt. Grundsätzlich ist so eine Prüfung ja relativ einfach: Du brauchst dafür nur nachzusehen, ob die Zahl jeweils an den beiden anderen Stellen vorkommt und damit **zwar vorkommt, aber nicht an der richtigen Stelle**. Das kannst du in der Variablen `richtigeZahl` vermerken.

Das ist an sich leicht zu programmieren:

```

if ( tipp1 == zahl2 ){
    richtigeZahl++;
}
if ( tipp1 == zahl3 ){
    richtigeZahl++;
}

```

Das müsstest du für die anderen Zahlen mit den jeweils anderen Stellen genauso machen.

Nur der Teufel steckt so tief im Detail und alle, die programmieren – »else if«

Die obige Lösung hat ein kleines, aber **höllisches Problem**: Deine Tipps können mehrfach gezählt werden. Ist der Code beispielsweise 747 und die erste Zahl deines Tipps 7, dann würde in diesem Fall **einmal** für die richtige Stelle und **zusätzlich** für eine richtige Zahl an falscher Stelle gezählt. Das würde jeden Spieler verwirren.

Wäre der **Code 277**, dann würde deine 7 **zweimal** als richtige Zahl an der falschen Stelle gewertet werden. Hättest du als Tipp für diesen Code 727 angegeben, hättest du dadurch (für alle Zahlen ausgewertet) eine Zahl an richtiger Stelle und immerhin **vier richtige Zahlen an falscher Stelle** – ein ganz imposanter Wert bei nur drei Stellen.

Was kann ich also machen?

Keine Zahl deines Tipps darf mehrfach gewertet werden.

Du musst also zum einen sicherstellen, dass für eine Zahl, die an der richtigen Stelle steht, nicht weitergesucht wird.

Zweitens darf eine Zahl an falscher Stelle nur einmal gezählt werden. Bei einem **Tipp** von 712 und dem **Code 977** würden ja ansonsten als Ergebnis zwei richtige Zahlen an falscher Stelle ausgegeben. Das wäre irreführend.

Drittens solltest du sicherstellen, dass eine Stelle, an der schon eine richtige Zahl gefunden wurde, nicht noch einmal gewertet wird. Bei einem **Tipp** von 177 und einem **Code** von 237 würde ja sonst eine 7 als richtige Zahl an falscher Stelle und die andere 7 als an richtiger Stelle gewertet.

Wie immer in der Programmierung gibt es mehrere Lösungen. Die einfachste Lösung hast du mit `else` und etwas zusätzlicher Logik bei der Überprüfung. Das einfache `else` hast du ja bereits kennengelernt: Mit `else` legst du fest, was passieren soll, wenn die Be-

dingung im davorstehenden `if` nicht zutrifft. Aber `else` kann **nicht nur allein**, sondern auch zusammen mit einem weiteren `if` in Erscheinung treten.

»else if« – ein starkes »ansonsten« mit einer weiteren Bedingung

Wie sieht das in unserem konkreten Fall aus?

Wenn deine erste Zahl mit der ersten Stelle des Codes übereinstimmt, dann soll das als richtige Stelle gezählt werden. Ansonsten (und nur ansonsten) soll überprüft werden, ob die Zahl an der zweiten oder dritten Stelle vorkommt – um dann einmalig als richtige Zahl an falscher Stelle gezählt zu werden.

Setzen wir das alles in JavaScript um. Hier für die erste Zahl:

Wenn ...	<pre>if(tipp1 == zahl1)</pre>
dann soll ...	<pre>{ richtigeStelle++; }</pre>
Ansonsten (und nur ansonsten!) soll geprüft werden, ob ...	<pre>else if (tipp1 == zahl2 tipp1 == zahl3)</pre>
um dann (und nur dann!) ...	<pre>{ richtigeZahl++; }</pre>

Das erste, ursprüngliche `if` ist unverändert. Nur steht hinter den geschweiften Klammern jetzt noch ein `else` (das hast du ja schon kennengelernt) und dabei ein komplettes `if` mit einer eigenen Bedingung.

Der dahinter folgende Teil in geschweiften Klammern `{ }` wird nur ausgeführt, wenn die **erste Bedingung nicht zutrifft** und wenn die **Bedingung hinter dem zweiten `if` zutrifft**.

Sehen wir uns noch die Bedingung bei dem zweiten `if` an:

```
if ( tipp1 == zahl2 || tipp1 == zahl3 )
```

Die beiden Striche `||` stehen für ein »Oder«. Wir überprüfen damit, ob unsere erste Zahl an der zweiten **oder** dritten Stelle des Codes zu finden ist. Dabei ist es in unserem Fall egal, ob sie an der zweiten oder dritten Stelle gefunden wird – **oder an beiden**: Der Ausdruck innerhalb der folgenden geschweiften Klammern wird nur einmal ausgeführt, wenn irgendetwas oder alles davon zutrifft.

Um die beiden Striche `||` für das »Oder« in eine Bedingung zu schreiben, musst du bei einem Windows-Rechner die Taste `[Alt]` drücken und halten und dann auf die Taste mit `>` und `<` tippen. Auf dem Mac sind es die Tasten `[Alt]` und `[7]`.

Könnte ich das nicht auch mit einem weiteren »else if« machen?

Natürlich, das würde genauso gut funktionieren:

```
if( tipp1 == zahl1 ){
    richtigeStelle++;
}else if ( tipp1 == zahl2 ){
    richtigeZahl++;
}else if ( tipp1 == zahl3 ){
    richtigeZahl++;
}
```

Da in beiden Fällen im `else if` das Gleiche passiert (die Variable `richtigeZahl` wird um 1 erhöht), ist es durchaus sinnvoll, alles mit einem **Oder** (`||`) zusammenzufassen.



Falls du es genauer wissen willst: »else« und »else if«

Das `else`, also unser **ansonsten**, ist ziemlich **ausschließlich**: Wenn die Bedingung in dem ersten `if` zutrifft, dann wird das, was zum `else` gehört, **nicht mehr ausgeführt**. Ganz genauso funktioniert das bei einem `else if`. Egal, was vielleicht in der Bedingung des zweiten `if` stehen mag, es wird dann überhaupt nicht mehr berücksichtigt.

Ach ja: In einigen Sprachen gibt es eine alternative Schreibweise von `else if`, nämlich **elseif**. Dort werden die beiden Schlüsselwörter zusammengeschrieben. JavaScript kennt und erlaubt hingegen nur die eine, getrennte Schreibweise.

Was jetzt noch fehlt – die anderen Zahlen, eine Ausgabe und 'ne tolle Schleife

Jetzt musst du noch die **zweite** und **dritte** Zahl deines Tipps überprüfen. Das ist nicht schwer, genau genommen musst du nur das gesamte »Konstrukt« mit `if` und `else if`

kopieren, zweimal wieder einfügen und die Variable `tipp1` passend in `tipp2` und `tipp3` umbenennen. **Aber Vorsicht:** Gerade beim vermeintlich so schnellen Arbeiten mit Copy and Paste passieren schnell nervige Flüchtigkeitsfehler. Du übersiehst eine Kleinigkeit, es kommt zu keiner Fehlermeldung, aber irgendetwas stimmt beim Ergebnis nicht. Also schau bitte genau nach, ob du alle notwendigen Umbenennungen richtig vorgenommen hast.

So sieht unser Programm bis jetzt aus:

```
//Der Computer denkt sich einen Code aus. Damit es nicht zu schwer wird,
//sollen es drei Stellen mit Zahlen von 1 bis 9 sein.
var zahl1 = Math.round( Math.random() * 9 + 0.5);
var zahl2 = Math.round( Math.random() * 9 + 0.5);
var zahl3 = Math.round( Math.random() * 9 + 0.5);
```

Hier wäre genau richtige Stelle für den **Beginn einer Schleife**, nachdem der Computer sich eine Zahl ausgedacht hat und bevor wir unseren Tipp abgeben.

```
//Wir raten dann eine Zahl
var meinVersuch = prompt("Gib einen Tipp ab", "Zahl von 111 bis 999");
var tipp1 = meinVersuch.charAt(0);
var tipp2 = meinVersuch.charAt(1);
var tipp3 = meinVersuch.charAt(2);

//Der Computer soll uns dann ausgeben, wie viele unserer Zahlen
//an der richtigen Stelle stehen und wie viele unserer Zahlen ueberhaupt
//in dem unbekanntem Code vorkommen.

var richtigeStelle = 0;
var richtigeZahl = 0;

//Das == sind zwei Gleichheitszeichen ohne Leerzeichen dazwischen
if( tipp1 == zahl1 ){
    richtigeStelle++;
}else if ( tipp1 == zahl2 || tipp1 == zahl3 ){
    richtigeZahl++;
}

if( tipp2 == zahl2 ){
    richtigeStelle++;
}else if ( tipp2 == zahl1 || tipp2 == zahl3 ){
```

```

    richtigeZahl++;
}

if( tipp3 == zahl3 ){
    richtigeStelle++;
}else if ( tipp3 == zahl1 || tipp3 == zahl2 ){
    richtigeZahl++;
}

```

Nachdem wir oben unseren Tipp abgegeben haben und der Computer überprüft hat, wie wir damit liegen, wäre **hier** die richtige Stelle für eine **Ausgabe**. Und auch das Ende der Schleife – mit einer passenden Bedingung – wäre an dieser Stelle goldrichtig. Wenn der Code geknackt wurde, sollte an dieser Stelle die Schleife verlassen werden und es sollte natürlich auch noch eine entsprechende Ausgabe zum Spielende erfolgen.

Dann wollen wir mal den Rest machen

So könnte unsere Schleife dazu aussehen:

```

do{

    //hier kommt alles hin,
    //was immer wieder gemacht werden soll

}while( richtigeStelle < 3 )

```

Die Bedingung ist recht einfach. Solange (*while*) nicht alle Stellen des Codes richtig getippt wurden, geht es oben (bei dem *do*) noch einmal von vorn los. Eine *while*-Schleife wird ja immer wieder durchlaufen, solange (bzw. während) eine angegebene Bedingung zutrifft. Anfangs mag es nicht ganz leicht sein, solch eine Bedingung richtig zu formulieren. Mit der Zeit bekommst du aber ein gutes Gefühl dafür – das nennt man manchmal auch Erfahrung.

```

alert("Du hast gewonnen. Super!");

```

Am **Ende des Spiels** wollen wir noch eine Ausgabe machen, um den Sieg gebührend zu feiern. Ein einfaches `alert` soll an dieser Stelle genügen.

Jetzt das ganze Programm **in einem Stück**. Um etwas Platz zu sparen, entfallen hier die Kommentare, die du im weiter oben dargestellten Code findest.

```

<script>
var zahl1 = Math.round( Math.random() * 9 + 0.5);
var zahl2 = Math.round( Math.random() * 9 + 0.5);
var zahl3 = Math.round( Math.random() * 9 + 0.5);

do{
    var meinVersuch = prompt("Gib einen Tipp ab", "Zahl von 111 bis 999");
    var tipp1 = meinVersuch.charAt(0);
    var tipp2 = meinVersuch.charAt(1);
    var tipp3 = meinVersuch.charAt(2);

    var richtigeStelle = 0;
    var richtigeZahl = 0;

    if( tipp1 == zahl1 ){
        richtigeStelle++;
    }else if ( tipp1 == zahl2 || tipp1 == zahl3 ){
        richtigeZahl++;
    }

    if( tipp2 == zahl2 ){
        richtigeStelle++;
    }else if ( tipp2 == zahl1 || tipp2 == zahl3 ){
        richtigeZahl++;
    }

    if( tipp3 == zahl3 ){
        richtigeStelle++;
    }else if ( tipp3 == zahl1 || tipp3 == zahl2 ){
        richtigeZahl++;
    }

    alert( richtigeStelle + " Zahlen an der richtigen Stelle, " +
        richtigeZahl + " Zahlen kommen im Code vor" );

}while( richtigeStelle < 3 )
alert("Du hast gewonnen. Super!");
</script>

```

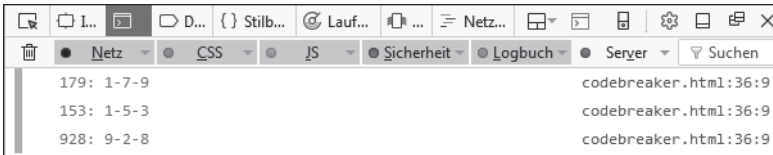


Abbildung 4.2 In drei Runden wurde 179, 153 und 928 eingegeben und korrekt in die Variablen »tipp1«, »tipp2« und »tipp3« aufgeteilt. Es scheint ja zu funktionieren!



Falls du es genauer wissen willst: Kontrolle über die Konsole

Das soll es tatsächlich geben: Dein Programm läuft zwar, macht aber irgendwie nicht so richtig, was es eigentlich soll? Kein Problem, denn der Browser gibt dir über die **Konsole** der integrierten Entwicklertools Hinweise auf mögliche (oder echte) Fehler. Und falls das Programm zwar fehlerfrei läuft, nur eben nicht so, wie du es vorgesehen hast, dann ist es Zeit, dir die Variablen und ihre Werte einmal genauer anzusehen. Sind die Zufallszahlen wirklich so, wie du es erwartest? Stimmt die bearbeitete Eingabe? Und was kommt bei den diversen Vergleichen der Werte heraus? Du kannst dir die Werte natürlich mit `alert()` ausgeben lassen, das ist aber nur für einzelne Werte sinnvoll. Ständig öffnen sich neue Fenster und müssen bestätigt werden. Einfacher ist die Ausgabe mit `console.log()`. Das funktioniert exakt so wie bei `alert()`: In den Klammern übergibst du einen Wert, eine Berechnung oder auch einen Vergleich. Nur öffnet sich nicht jedes Mal ein Fenster, das geschlossen werden müsste – die Ausgabe erfolgt nebenbei in der Konsole. Ist die Konsole nicht geöffnet, hat dies keinen Einfluss auf dein Programm. Öffnest du die Konsole, wird dir das Ergebnis im Konsolenfenster angezeigt – ohne eine entnervende Klickorgie, gerade bei Schleifen.

So könntest du dir beispielsweise deine Eingabe und die daraus erzeugten einzelnen Stellen ausgeben lassen:

```
console.log(meinVersuch + ": " + tipp1 + "-" +
  tipp2 + "-" + tipp3);
```

Das muss sinnvollerweise natürlich hinter der Eingabe und der Berechnung der einzelnen Stellen stehen. Auch kannst du jeden benötigten (oder gerade interessanten) Wert einzeln mit `console.log()`; ausgeben lassen, das geht genauso gut.

Tunen, tieferlegen, lackieren und Locken eindrehen

Jetzt haben wir unser Programm. Es sollte funktionieren und fehlerfrei arbeiten. Aber das genügt uns nicht. Denn bei jedem Programm gibt es auch nach einer ersten lauffähigen Version **noch genug zu tun**. Und eigentlich ist so eine erste lauffähige Version auch immer nur der Anfang, denn dann folgt die Kür.

Aufgaben

In diesem Abschnitt geht es darum, was du **noch besser machen kannst**.

Einige der Aufgaben kannst du ohne weiteres Wissen lösen, zu einigen Aufgaben werden wir noch ein paar Dinge in JavaScript besprechen. Ich werde dir dabei nur ein paar Tipps und Hinweise geben. Den Rest schaffst du allein, das bekommst du hin.

Also: Was kannst du noch verbessern?

- ▶ Zähl die gespielten Runden, und begrenze das Spiel auf eine festgelegte Anzahl von Runden.
- ▶ Ermögliche ein vorzeitiges Beenden des Spiels. In diesem Fall verliert der Spieler, erfährt aber den geheimen Code.
- ▶ Das Spiel soll einen Einleitungstext erhalten und erst durch einen Klick des Spielers gestartet werden.

Zähl die gespielten Runden und begrenze das Spiel auf eine festgelegte Anzahl von Runden

Um etwas zu zählen, brauchst du eine Variable, die du nicht nur deklarierst, sondern auch **initialisiert** hast: Du musst ihr also **einen ersten Wert** geben – sinnvollerweise 0, um hochzuzählen. Du erinnerst dich: Erst wenn eine Variable einen Wert hat, kann sie selbst Teil einer Berechnung werden (darf also rechts von einem = stehen). Ansonsten ist ihr Wert `undefined`, und du bekommst als Ergebnis bei jeder Berechnung mit dieser Variablen nur einen abstrusen Wert `NaN` (*not a number* – das ist keine Zahl) als Ergebnis der jeweiligen Berechnung. Auch JavaScript kann streng sein ...

Dann solltest du noch aufpassen, **wo** du die Variable auf 0 setzt – wenn du es in der Schleife machst, wird sie bei jedem Durchlauf wieder auf 0 zurückgesetzt. Zählen geht anders.

```
var meinZaehler = 0;
// Beginn der Schleife
    meinZaehler = meinZaehler + 1;

    //oder (beides wäre etwas zu viel des Guten)

    meinZaehler++;
//Ende der Schleife
```

Natürlich kannst du auch in jeder Runde ausgeben, wie viele Runden bereits gespielt wurden. Das könnte dann so aussehen, wenn du es in die bestehende Ausgabe einbaust:

```
alert( meinZaehler + ".Runde: " + richtigeStelle +
      " Zahlen an der richtigen Stelle, " + richtigeZahl +
      " Zahlen kommen im Code vor" );
```

Wenn du die Runden nicht nur rein informativ mitzählen möchtest, kannst du das Spiel nach einer bestimmten **Zahl von Runden als verloren** beenden.

Der Spieler hat zu lange gebraucht – die Falle des mysteriösen Mister JS schnappt zu ...

Du musst in diesem Fall dafür sorgen, dass die **Schleife verlassen wird**. Auch die fest programmierte Ausgabe für den Gewinn ergibt dann keinen Sinn mehr.

Füge zuerst dem `while` eine entsprechende Bedingung hinzu, die zutrifft (also wahr ist), solange das Spiel in die nächste Runde gehen soll. In JavaScript »gedacht«, könnte das lauten: Mach diesen Teil so lange, wie die Anzahl der richtigen Stellen kleiner als 3 ist **und** die Anzahl der Runden kleiner als (beispielsweise) 12 ist.

```
do{
    //hier passiert alles Mögliche
}while( richtigeStelle < 3 && meinZaehler < 12 )
```

Bis jetzt war es immer klar: Wenn Die Schleife verlassen wurde, dann hatte der Spieler den Code geknackt **und gewonnen**. Darauf können wir uns nicht mehr verlassen, denn jetzt kann der Spieler das Spiel auch verloren haben.

```
//Wir sind hier hinter der Schleife
if( richtigeStelle == 3 && meinZaehler < 12 ) {
    alert("Du hast gewonnen. Super!");
}else{
    alert("Du konntest den Code nicht rechtzeitig knacken!\nDer mysteriöse
        Mr. JS hat gewonnen");
}
```

Mit einem `if` und einem zugehörigen `else` schaffst du das alles ohne Probleme. Da es nur **zwei Möglichkeiten** gibt, müssen wir nur eine Bedingung schreiben, die den Sieg überprüft, und können alle anderen Möglichkeiten mit dem `else` behandeln.

Vielleicht ist dir das `\n` im Text unseres `alert` aufgefallen? Der **Text** in einem `alert` wird manchmal recht lang. Irgendwann (und meist an einer unpassenden Stelle) wird langer Text von manchen Browsern automatisch in eine neue Zeile umbrochen. Im Gegensatz

zu anderen Anweisungen darfst du **keinen Umbruch im Text** machen. Die einzige Ausnahme davon ist die Möglichkeit, den Text in Backticks zu schreiben, um Zeilenumbrüche direkt in den Text schreiben zu können. Mit dem `\n` hingegen legst du auch in normalen Texten fest, wo ein Umbruch im Fenster von alert erfolgen soll. Etwas Kosmetik schadet doch nie.

Strings mit Backticks – Template Literals

Manche Dinge klingen kompliziert. Dazu gehören ohne Zweifel *Template Literals* bzw. die »Strings mit Backticks«. Kurz und umso einfacher erklärt: Damit können auf sehr einfache Art mehrzeilige Strings, also Texte, geschrieben werden. Dazu werden die Strings nicht mit einfachen oder doppelten Anführungszeichen, sondern mit Backtick-Zeichen umschlossen, also so:

```
meinString = `Das ist ein Text mit Backticks`;
```

Mit den Backticks kannst du Texte über beliebig viele Zeilen schreiben, ohne irgendwelche Sonderzeichen oder andere Konstrukte bemühen zu müssen.

```
mehrzeiligerString = `Zeile 1 und...
```

```
  Zeile 2 und...
```

```
  Zeile 3. `;
```

Die Zeilenumbrüche und auch die Leerzeichen bleiben exakt so, wie du sie in deinen String geschrieben hast. Das kannst du sehr gut sehen, wenn du so einen String mit alert aus gibst.

Aber das ist noch **nicht alles**. Du kannst Variablen und sogar ganze Rechenoperationen (fast) direkt in diese Art von Strings schreiben. Das machst du mit einem Dollarzeichen und geschweiften Klammern, in die du dann deine Variablen schreibst:

```
alert( `${meinZaehler}.Runde:  
  ${richtigeStelle} an der richtigen Stelle,  
  ${richtigeZahl} richtige Zahlen.` );
```

Bei der Ausgabe werden weder das Dollarzeichen, noch die zugehörigen geschweiften Klammern dargestellt – dafür wird deren aktueller Inhalt dargestellt. Das mag so im Programm erst mal nicht besonders schick aussehen – ist aber dafür sehr praktisch!

Wenn's dann doch mal reicht – das Spiel selbst beenden

Manchmal steckt einfach der **Wurm drin**. Man glaubt, alles versucht zu haben, und es will einfach nicht klappen – keine Angst, ich meine nicht das Programmieren: Wir wol-

len im Spiel eine *Abbruchbedingung* einbauen, falls der Spieler verzweifelt **aufgeben** will. Wird ein bestimmter Wert eingegeben, soll dies als ein »Ich will aufhören, ich gebe auf« gewertet werden. Wir könnten Werte festlegen, die als Aufgabe (im Sinne von »aufgeben«) gewertet würden, zum Beispiel alles, was kleiner als 111 oder größer als 999 ist. Du könntest jetzt versucht sein, das auf die Schnelle als Bedingung so zu schreiben:

```
meinVersuch < 111 || meinVersuch > 999
```

while arbeitet aber so, dass es in die nächste Runde geht, wenn die angegebenen Bedingungen **stimmen**. Du musst es also genau umgekehrt formulieren:

```
meinVersuch >= 111 && meinVersuch <= 999
```

Solange das zutrifft, die Zahl also zwischen 111 und 999 (einschließlich) liegt, solange **geht es weiter**. Ist der eingegebene Wert kleiner (oder größer), **endet das Spiel**.

So könntest du es dann in die *do-while*-Schleife einbauen:

```
do{
    //hier passiert alles Mögliche
}while(richtigeStelle < 3 && meinVersuch >= 111 && meinVersuch <= 999)
```

Das ist schon etwas lang, und es besteht die (recht konkrete) Gefahr, dass es unübersichtlich wird. Die Lösung: Du kannst Bedingungen auch vorher – also vor der *while*-Schleife – ausführen lassen und das **Ergebnis** einfach in einer **Variablen ablegen**. Diese Variable baust du dann – quasi ersatzweise – in die Bedingung der *while*-Schleife ein.

```
do{
    //hier passiert alles Mögliche

    var nichtAufgegeben = meinVersuch >= 111 && meinVersuch <= 999;
}while(richtigeStelle < 3 && nichtAufgegeben)
```

Der Vorteil liegt sowohl klar auf der Hand als auch gut **sichtbar im Programmcode**: Es ist übersichtlicher, **viel übersichtlicher**. Nützlicher Nebeneffekt: Es ist oft nicht leicht, die unterschiedlichsten Bedingungen, die wiederum mit `||` (also »oder«) und `&&` (»und«) verknüpft sind, korrekt zusammenzufügen. Teilst du solche komplexen Bedingungen aber auf und schreibst sie in eigene Variablen, hast du solche Probleme gar nicht.

Nicht vergessen – wie war denn jetzt der Code?

Wenn das **Spiel beendet wird**, soll dem Spieler angezeigt werden, wie der bis dahin geheime Code ausgesehen hat, an dem er (hoffentlich nicht du) fast verzweifelt ist:

```
alert("Die Lösung war " + zahl1 + " "+ zahl2 + " "+ zahl3);
```

Ein paar Zeilen als Einleitung

Das Spiel soll auch einen **Einleitungstext** erhalten und erst durch einen Klick des Spielers gestartet werden.

Der Einleitungstext ist schnell gemacht – ganz einfach in der Webseite, im HTML-Code. Dafür müssen wir JavaScript nicht bemühen. Schwieriger ist es, einen ansprechenden Text zu schreiben, der den Spieler motiviert, das Spiel zu spielen. Du kannst versuchen, auch mit wenig Text eine kurze Geschichte zu erzählen – vielleicht ist der Spieler ein Geheimagent, der sich auf seine Einsätze vorbereitet:

```
<h1>CodeBreaker</h1>
<p>Dies ist das Trainingsprogramm für Geheimagenten.
  Es bereitet dich auf deine Einsätze vor und bringt dir bei,
  Geheimcodes zu entschlüsseln.</p>
<p>Gib dein Bestes, um den geheimnisvollen Mr. JS zu besiegen!</p>
```

So ist das gleich richtig in HTML geschrieben. Du kannst natürlich gerne mehr daraus machen. Pass nur auf, dass du HTML und JavaScript nicht vermischst.

JavaScript über Klicks auf HTML-Elemente aufrufen

Der Spieler soll das Programm jetzt **selbst starten** – nicht durch das Öffnen der Webseite, sondern gezielt durch einen **Klick** auf ein bestimmtes Element im HTML-Code, innerhalb der Webseite. HTML und JavaScript arbeiten hier zum Glück gut zusammen. Das Element kann ein Text oder auch eine Grafik sein. Wir nehmen einen **Text**: den Namen unseres Programms und der Optik wegen ein schickes, passendes **Symbol** dazu. Wir haben ja nicht nur die normalen Zahlen und Buchstaben, sondern auch etliche Sonderzeichen, die uns dank **Unicode** zur Verfügung stehen. Das Ganze versehen wir noch mit einer stattlichen Größe, etwas Farbe und einem malerischen Schatten.

Im HTML-Code, also außerhalb unseres `script`-Tags, schreiben wir einen knackigen Titel und dazu ein passendes **Symbol**, ein Schloss mit einem Schlüssel:

```
<p style="font-size:42pt;color: black; text-shadow:
grey 0.05em 0.05em 0.1em;">CodeBreaker...&#128272;</p>
```

Wenn dir das Grau als Schatten zu langweilig ist, versuch es doch einmal mit Rot, `text-shadow:red`.

Was bedeutet das »🔐«, und was war noch mal Unicode?

Unicode ist ein **Standard**, der alle möglichen und unmöglichen **Zeichen** enthält. Geordnet nach Art und Sprache sind dort die verschiedensten Zeichensätze festgehalten. Jedes Zeichen hat eine eindeutige Nummer, die im HTML-Code in der obigen Form geschrieben werden kann. Was es alles gibt, findest du in entsprechenden Listen im Internet. Suche einfach einmal im Internet nach »Unicode-Liste«.

Leider sind nicht alle Zeichen aus dem Unicode auch direkt im Browser verfügbar. Das hängt damit zusammen, dass zwar alle Zeichen definiert sind, aber nur für einen relativ **kleinen Teil darstellbarer Zeichen** im Browser vorgehalten werden. Natürlich könntest du auch passende Schriften nachladen, aber wir haben ja schon etwas Passendes:









Abbildung 4.3 So sieht es dann aus – gar nicht so schlecht.



Falls du es genauer wissen willst: Das Erbe von C und komische Tastenkombinationen

Wie viele erfolgreiche Programmiersprachen orientiert sich JavaScript an der C-Syntax. C ist eine sehr erfolgreiche Programmiersprache, die in den 1970er Jahren entstand. Den spektakulären Namen C hat die Sprache, weil sie Nachfolger einer Programmiersprache namens B war (ehrlich!). C ist extrem schnell. Noch heute wird C eingesetzt, wenn Programme besonders schnell ablaufen müssen oder die verwendeten Rechner nur eine geringe Rechenleistung haben (kleine mobile Geräte oder Einplatinencomputer).

Wer in einer Sprache mit C-Syntax programmiert, freut sich immer wieder über die **teils abstrusen Tastenkombinationen** und Verrenkungen, die notwendig sind, um ständig solche Zeichen wie `{ }` oder `||` zu tippen. Vielleicht kommt sogar die Frage auf, warum ausgerechnet diese Art von Syntax so erfolgreich ist. Die Antwort ist ganz einfach:

Wer auf einer **amerikanischen** oder englischen **Tastatur** schreibt – und das machen die meisten Erfinder von Programmiersprachen –, kann all diese schönen Zeichen entweder direkt oder mit der -Taste verwenden. Dort, wo sich unsere Umlaute befinden, sind die Klammern / und /. Das häufig verwendete Semikolon ; kann direkt getippt werden – dort, wo bei uns das  sitzt.

Die **Verbindung** zwischen **HTML** und **JavaScript** ist sehr eng, was wir jetzt zum ersten Mal richtig nutzen werden: Es ist problemlos möglich, JavaScript direkt aus HTML zu starten, zum Beispiel **durch einen Klick** auf einen Text. So muss JavaScript nicht zwangsläufig beim Öffnen der Seite ausgeführt werden, sondern kann artig warten, bis es über einen **eigenen Namen aufgerufen** wird.

Was müssen wir dazu machen?

Wir müssen nur einem **von uns bestimmten Element** im HTML-Code sagen, dass es bei einem **Klick** JavaScript aufrufen soll. Natürlich müssen wir in diesem Aufruf auch angeben, **was** denn gestartet werden soll. Wir nehmen als Element in HTML einfach ein `div`, ein recht neutrales Element, das für solche Spielereien wie gemacht ist. Ein `div` selbst hat nämlich praktisch gar keine eigenen Eigenschaften, es ist vielmehr so etwas wie ein Container für Eigenschaften oder um andere Tags zusammenzufassen. So ein `div` setzen wir also um unseren Text, der dadurch genauso **klickbar wird**. Denn die Eigenschaften eines Tags (und dazu gehört das »Anklickbarsein«) vererben sich auf das, was sich innerhalb dieses Tags befindet – andere Tags und deren Inhalte eingeschlossen.

Der Befehl im Tag lautet `onclick` und hat als Wert den **Namen einer** von dir festgelegten *Funktion* (den Begriff erkläre ich gleich). Diesen Befehl musst du in das Tag wie ein Attribut bzw. eine Eigenschaft einbauen, etwa wie folgt:

```
<div onclick="codeBreaker();">
  <p style="font-size:42pt; color:black; text-shadow:
grey 0.05em 0.05em 0.1em;">CodeBreaker...&#128272;</p>
</div>
```

Du könntest den Aufruf auch direkt in das verwendete Tag `<p ...>` setzen, aber wir wollen es ja auch etwas **übersichtlich** halten – und da ist ein eigenes Tag ganz hilfreich. Schließlich sind die Kosten für ein paar zusätzliche Zeilen Quelltext nicht besonders hoch.

Aber was passiert da eigentlich?

JavaScript kann über sogenannte *Ereignisse* gestartet werden. JavaScript kennt einige Ereignisse, wie das **Anklicken mit der Maus**, das **Drücken einer Taste** oder das **Absenden eines Formulars**. Sogar das Laden oder das Verlassen der Webseite sind Ereignisse.

Vereinfacht kannst du dir das so vorstellen: Nachdem eine Webseite geöffnet wurde, passt der **Browser** die ganze Zeit auf, ob irgendeines dieser **Ereignisse eintritt** oder durch irgendeine Aktion *ausgelöst* wird. Wenn das passiert, führt der Browser die Befehle (oder die Funktion), die zu diesem Ereignis hinterlegt sind, einfach aus. Du musst nur die Anweisung im HTML-Code geben, den Rest macht der Browser.

Und wo finde ich diese Funktion namens »codeBreaker«?

Das ist jetzt deine Aufgabe. **Du selbst** kannst in deinem Programm jederzeit beliebige Funktionen schreiben und ihnen (fast) beliebige Namen geben. In diesen Funktionen kannst du alles Mögliche programmieren.

Funktionen?

Funktionen – besonders dick und saugfähig

Nun, eigentlich sind *Funktionen* weder dick noch saugfähig. Aber sie sind tatsächlich unglaublich praktisch und man braucht sie für alles Mögliche und Unmöglichliche.

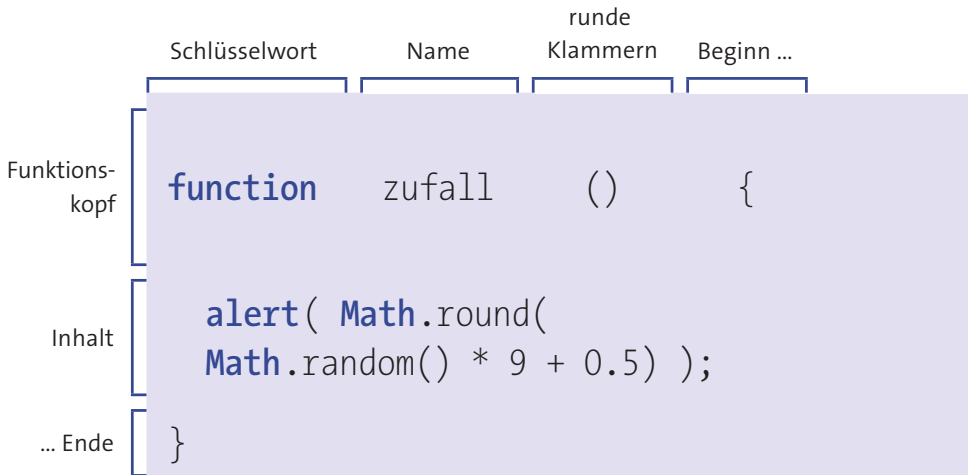
Während Variablen dafür verwendet werden, Werte zu speichern, können **Funktionen** ganze **Programmteile aufnehmen**. So wie du den Wert einer Variablen jederzeit aufrufen kannst, kannst du den Programmcode einer Funktion jederzeit über deren Namen aufrufen.

Eine Funktion zu schreiben ist recht einfach. Wenn **bisher** etwas im Programm passieren sollte, hast du das als entsprechende Anweisungen direkt geschrieben, einfach so in das `script`-Tag. Nehmen wir als kurzes Beispiel die Ausgabe unserer Zufallszahl von 1 bis 9 mit einem `alert`:

```
alert( Math.round( Math.random() * 9 + 0.5) );
```

So etwas in der Art kennst du ja schon. Wenn du diese Ausgabe jetzt dreimal an unterschiedlichen Stellen bräuchtest, müsstest du das **dreimal schreiben** – oder eben per Copy and Paste kopieren. Das ist aufwendig, erzeugt viel Code, und wenn du hier etwas ändern musst, musst du das an jeder Stelle machen. Das ist kein Problem, aber nicht umsonst ist gesunde Faulheit eine sehr geschätzte Eigenschaft beim Programmieren.

Und so kommt jetzt die Magie der *Funktionen* ins Spiel:



`function` ist das Schlüsselwort. JavaScript weiß dadurch: Hier wird eine Funktion definiert. Und das Wort **nach** dem Schlüsselwort `function` ist der **Name der Funktion**. Diesen Namen legst du selbst fest. Hier gelten ähnliche Namensregeln wie bei Variablen: keine Zahlen am Anfang, keine Leerzeichen usw. Der Name darf natürlich auch kein bereits vorhandenes Schlüsselwort von JavaScript sein – du könntest deine Funktion also zum Beispiel nicht `alert` nennen. JavaScript besteht hier auf seinen älteren Rechten und es käme zu einem Fehler. Die runden Klammern gehören zur Funktion, später wirst du darüber auch Werte **an Funktionen übergeben**.

Zwischen die geschweiften Klammern schreibst du den Inhalt, also alles, was die Funktion machen soll, eigentlich ganz einfach. Das Besondere kommt erst noch: Wenn du das alles so geschrieben hast – passiert erst einmal gar nichts. Die Funktion wird tatsächlich nur definiert. Sie ist vorhanden, nicht mehr und nicht weniger. **Sie lauert** in Habtachtstellung **auf ihren Auftritt**. Brauchst du deinen Code jetzt irgendwo im Programm, dann rufst du so deine Funktion auf:

```
zufall();
```

Du schreibst einfach den Namen (natürlich ohne das Schlüsselwort `function`) und dahinter die (leeren) runden Klammern und dahinter (ja, optional) ein Semikolon.

Es funktioniert ein bisschen wie bei einer Variablen: Der aktuelle Wert (bei einer Funktion eben der hinterlegte Programmcode) wird an dieser Stelle verwendet oder dort quasi »eingesetzt«. So, wie eine Variable der Platzhalter für einen Wert ist, ist eine Funktion vereinfacht ausgedrückt der Platzhalter für Programmcode.

Über den Namen wird also der Inhalt der Funktion an dieser Stelle abgerufen, **beliebig oft** und überall, wo du es möchtest.

Und wie geht das jetzt bei unserem Programm?

Das funktioniert so einfach wie in unserem Beispiel mit alert. Wir haben etwas mehr Code, nämlich unser gesamtes Programm:

```
<div onClick="codeBreaker();">
  <p style="font-size:42pt;color: red; text-shadow:
    red 0.05em 0.05em 0.15em">&#9200;</p>
</div>
<script>
function codeBreaker(){
//hier ist das gesamte Programm
}
</script>
```

Die Änderungen sind eigentlich minimal. Es kommt eine Funktion dazu, und das Programm wird – so, wie es ist – in die Funktion verschoben.

Und jetzt alles

```
<html>
  <head>
    <meta charset="utf-8">
  </head>
<body>

<h1>CodeBreaker</h1>
<p>Dies ist das Trainingsprogramm für Geheimagenten.
  Es bereitet dich auf deine Einsätze vor und bringt dir bei,
  Geheimcodes zu entschlüsseln.</p>
<p>Gib dein Bestes, um den geheimnisvollen Mr. JS zu besiegen!</p>

<div onClick="codeBreaker();">
  <p style="font-size:42pt; color:black; text-shadow:
grey 0.05em 0.05em 0.1em;">CodeBreaker...&#128272;</p>
</div>
```

```

<script>
function codeBreaker(){

var zahl1 = Math.round( Math.random() * 9 + 0.5);
var zahl2 = Math.round( Math.random() * 9 + 0.5);
var zahl3 = Math.round( Math.random() * 9 + 0.5);
var meinZaehler = 0;

do{
    meinZaehler = meinZaehler + 1;

    var meinVersuch = prompt("Gib einen Tipp ab", "Zahl von 111 bis 999");
    var tipp1 = meinVersuch.charAt(0);
    var tipp2 = meinVersuch.charAt(1);
    var tipp3 = meinVersuch.charAt(2);

    var richtigeStelle = 0;
    var richtigeZahl = 0;

    if( tipp1 == zahl1 ){
        richtigeStelle++;
    }else if ( tipp1 == zahl2 || tipp1 == zahl3 ){
        richtigeZahl++;
    }

    if( tipp2 == zahl2 ){
        richtigeStelle++;
    }else if ( tipp2 == zahl1 || tipp2 == zahl3 ){
        richtigeZahl++;
    }

    if( tipp3 == zahl3 ){
        richtigeStelle++;
    }else if ( tipp3 == zahl1 || tipp3 == zahl2 ){
        richtigeZahl++;
    }

    alert( meinZaehler + ".Runde: " + richtigeStelle +
        " Zahlen an der richtigen Stelle, " + richtigeZahl +
        " Zahlen kommen im Code vor" );
}
}

```

```

var nichtAufgegeben = meinVersuch >= 111 && meinVersuch <= 999;
}while( richtigeStelle < 3 && nichtAufgegeben && meinZaehler < 12 )

if( richtigeStelle == 3 && meinZaehler < 12 ) {
    alert("Du hast gewonnen. Super!");
}
if( meinZaehler >= 12 ) {
    alert("Zu viele Versuche!\nDer mysteriöse Mr. JS hat gewonnen");
}
if( nichtAufgegeben == false ) {
    alert("Du hast aufgegeben. Die Lösung ist " +
        zahl1 + " " + zahl2 + " " + zahl3);
}
}
}
</script>
</body>
</html>

```

Das Programm in (s)einer endgültigen Form mit allen **Verbesserungen**, die ich vorgeschlagen hatte. Wie immer speicherst du die Änderungen im Texteditor und lädst danach die Seite im Browser neu. Klickst du jetzt auf den Titel oder das Symbol daneben, dann startet dein Programm.

Es gibt noch mehr Möglichkeiten, das Programm zu verbessern, es schöner und eleganter zu machen: mit etwas Farbe, mit ein bisschen mehr Text und Formatierungen. Schließlich kannst du das Programm noch auf einen **vierstelligen Code erweitern**. Vielleicht wäre es ja auch ganz schick, wenn jede Zahl nur einmal im gesuchten Code vorkäme und wenn die Eingabe noch besser überprüft würde? Es gibt immer viel zu tun – viel Spaß beim Ausprobieren.



Abbildung 4.4 Viel Spaß beim Ausprobieren und Verbessern!