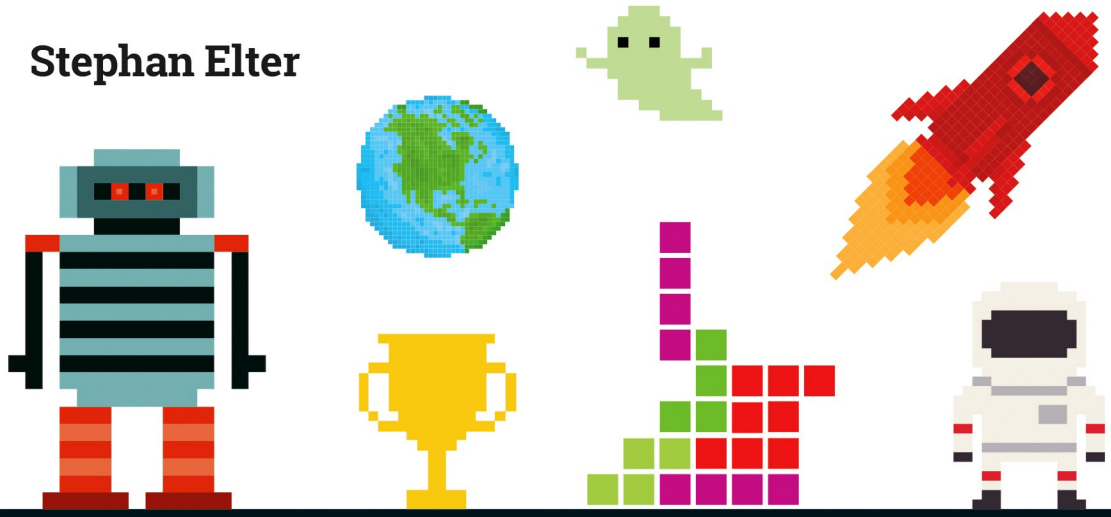
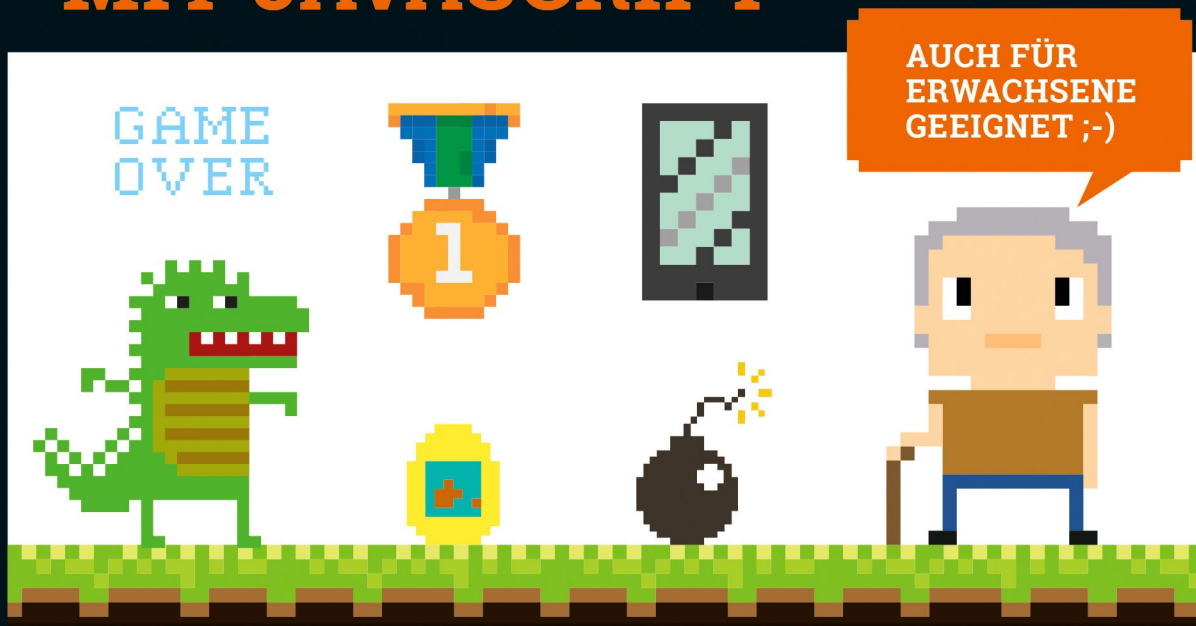
Stephan Elter



PROGRAMMIEREN LERNEN MIT JAVASCRIPT



- Einfach starten – du brauchst nur deinen Computer
- Programmiere Spiele wie Tetris und bekannte Klassiker
- Lerne Schritt für Schritt die Sprache des Web

2. AUFLAGE

Rheinwerk Computing

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 hübscher zu machen. 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 seiner Stirn. Er musste den Code knacken und das Rätsel lösen – koste es, was es wolle. Diese Sache war zu wichtig. Mit zitternden Händen tippte er den Code auf der Tastatur und wartete auf das Ergebnis: Na, das sieht doch nicht schlecht aus. Rasend schnell tippte er eine neue, leicht veränderte Kombination von Zahlen. Er musste den Computer besiegen – und zwar schnell.

Bleiben 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: 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 sind und wie viele unserer Zahlen in dem unbekannten 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: Wollen wir als geheimen Code **drei einzelne Zahlen** (und damit drei einzelne Stellen und Variable) 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 Programmierern 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 kann der Spieler seinen Tipp mit einer einzigen Eingabe abgeben. Andernfalls müsste er die Eingabe dreimal machen und bestätigen – das wäre vielleicht einfacher zu programmieren, »benutzerfreundlich« sieht aber anders aus.

Fangen wir an und nehmen wir 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 machen. Auch in diesem Fall gilt natürlich: Anders machen ist ausdrücklich erlaubt (zumindest, solange es funktioniert).

Wir raten dann eine Zahl.

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

Wir machen die Deklaration 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 Variable (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.

Der Computer soll uns dann ausgeben, wie viele unserer Zahlen an der richtigen Stelle sind und wie viele unserer Zahlen überhaupt in dem unbekannten 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 Variable, 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 sind und wie viele der angegebenen Zahlen richtig, aber an der falschen Stelle sind. 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.

Variable 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. 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 Programmierer: »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 **4 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 ist, 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 Bedingung 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 ...,

```
if( tipp1 == zahl1 )
```

dann soll ...

```
{
  richtigeStelle++;
}
```

Ansonsten (und nur ansonsten!) soll geprüft werden, ob ...

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

um **dann** (und nur dann!) ...

```
{
  richtigeZahl++;
}
```

Das erste, ursprüngliche `if` ist unverändert. Nur steht hinter der geschweiften Klammer 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**: Die folgende geschweifte Klammer wird nur einmal ausgeführt, wenn irgendetwas oder alles davon zutrifft.

Um die beiden Striche || für das »Oder« in einer 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 eins 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 gesamtes »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 & 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 die 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 sind und wie viele unserer Zahlen ueberhaupt
//in dem unbekannten Code vorkommen.

var richtigeStelle = 0;
var richtigeZahl = 0;

//Das == sind zwei Ist-Zeichen 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 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 siehst du 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, sich 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 Wertevergleichen? 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 da die Ausgabe mit **`console.log()`**. Das funktioniert exakt so wie bei `alert`: In der Klammer 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 Klick-Orgie, 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 kommt 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 eine entsprechende Bedingung zu dem `while` hinzu, eine Bedingung, 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. Anders als bei anderen Anweisungen darfst du **keinen Umbruch im Text** machen. So etwas (mit einem echten Zeilenumbruch im Quelltext) geht also **nicht**:

```
alert('So eine Zeile kann schon ziemlich lang werden. ↵
   Und ohne Zeilenumbruch sieht das dann schon etwas schräg aus.');
```


Mit **\n** legst du selbst fest, wann ein Umbruch im Fenster von `alert` auf jeden Fall erfolgen soll. Etwas Kosmetik schadet doch nie.

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 wollen 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
```

So lange 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 Variable, hast du solche Probleme gar nicht.

Nicht vergessen – wie war denn jetzt der Code?

Wenn das **Spiel beendet wird**, soll dem Spieler am Spielende angezeigt werden, wie der bis dahin geheime Code ausgesehen hat, an dem der Spieler (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. Mehr kannst du natürlich gerne machen. Pass bitte auf, dass du das HTML und das 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**, also 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 setzen ein dazu 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 darstellbare Zeichen** im Browser vorgehalten werden. Natürlich könnte man auch passende Schriften nachladen, aber wir haben ja schon etwas Passendes:

Abbildung 4.3 So sieht es dann aus – fast noch etwas zu dezent. Aber es ist ja Platz für ein paar auffälligere Farben: Rot ist auch einen Versuch wert.

Wer mit einem Mac arbeitet, bei dem ist das Ganze sogar von Haus aus gleich etwas schicker.

Abbildung 4.4 Tatsächlich noch etwas schicker – die Darstellung der gleichen Seite auf einem Mac.

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 70er-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, sitzen auf einer US-Tastatur die Klammern [/] und {/}. Das häufig verwendete Semikolon ; kann direkt getippt werden – dort, wo bei uns das Ö 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 dem 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 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). Das 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 einfach die Befehle (oder die Funktion) aus, die zu diesem Ereignis hinterlegt sind (bzw. ist). 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 Unmögliche.

Während Variable dafür verwendet werden, Werte zu speichern, können **Funktionen** ganze **Programmteile aufnehmen**. So wie du den Wert einer Variablen jederzeit aufru-

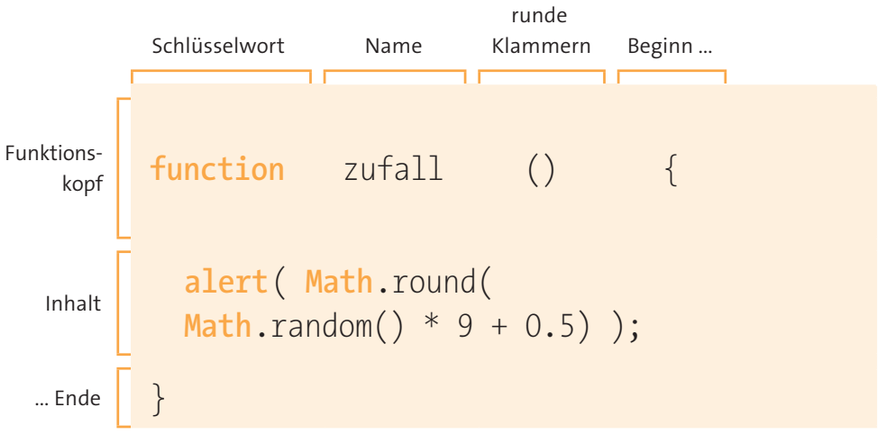
fen kannst, kannst du den Programmcode einer Funktion jederzeit über deren Namen aufrufen.

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 & 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 bei Programmierern.

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 ist auch: 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 Hab-acht-Stellung **auf ihren Auftritt**. Brauchst du deinen Code jetzt irgendwo im Programm, dann rufst du so deine Funktion auf:

```
zufall();
```

Du schreibst also einfach ihren 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 damit der Platzhalter für Programmcode – vereinfacht ausgedrückt.

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 f&uuml;r Geheimagenten.
       Es bereitet dich auf deine Eins&auml;tze vor und bringt dir bei,
       Geheimcodes zu entschl&uuml;sseln.</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!\n Der 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.

Achtung: Keine Zeilenumbrüche

Falls du längere Texte verwendest: Jeder Text in JavaScript, insbesondere innerhalb von `alert`, muss für sich in einer Zeile stehen – ganz im Gegensatz zu Texten in HTML. Ein echter **Zeilenumbruch** in einem Text ist nicht erlaubt und führt zu einem **Fehler**.

Das Folgende löst also einen **Fehler** aus, da in den Texten ein Zeilenumbruch steht (erkennbar hier im Buch am ˣ):

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

So geht es:

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

Eine Möglichkeit gibt es aber trotzdem, Texte über mehrere Zeilen zu schreiben: Du musst am Ende der jeweiligen Zeile (im Text selbst) ein »\« einfügen:

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

Auf diese Art darfst Du auch **Texte umbrechen**. Leider kommen einige Editoren bei der Darstellung etwas durcheinander, und besonders übersichtlich ist das auch nicht. (Du musst in diesem Fall mit den Einrückungen aufpassen, die dann Teil des Textes werden.) Deshalb werden wir diese Möglichkeit nicht weiter nutzen – erlaubt ist es jedenfalls.

Wie immer speicherst du die Änderungen im Text-Editor und lädst danach die Seite im Browser neu. Klickst du jetzt auf den Titel oder das Symbol daneben, dann startet dein Programm.

Wie immer gibt es noch mehr Möglichkeiten, das Programm zu verbessern, es schöner und eleganter zu machen: mit etwas Farbe, ein bisschen mehr Text und Formatierungen. Schließlich kannst du das Programm noch auf einen **4-stelligen Code erweitern**. Wäre es vielleicht 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 4.5 Viel Spaß beim Ausprobieren und Verbessern!

Auf einen Blick

Inhalt