



CODERS.BAY

FUNKTIONEN

- Eine **Funktion** ist ein benannter Codeblock, der vom Rest des Programms weitestgehend unabhängig ist. Sie erfüllt eine bestimmte Aufgabe und kann mehrmals und von verschiedenen Stellen aus **aufgerufen** werden, um diese Aufgabe zu erfüllen. Beim Aufruf können ein oder mehrere **Parameter** übergeben werden, Werte, die die Funktion zum Erfüllen ihrer Aufgabe benötigt. Eine Funktion kann einen **Rückgabewert** haben, einen Wert, der dem Aufrufer als Ergebnis der Funktion zurückgegeben wird.
 - Das Schlüsselwort **function** leitet die Funktionsdefinition ein
 - Funktionsnamen folgen denselben Regeln wie Variablennamen: An erster Stelle dürfen Buchstaben, \$ und _ stehen, weiter hinten auch Ziffern. Auch für Funktionen gilt die Konvention, ihren Namen mit einem Kleinbuchstaben beginnen zu lassen.
 - In runden Klammern werden die Parameter angegeben, die diese Funktion erwartet. Hat die

Funktion keine Parameter, bleiben die Klammern leer, aber sie müssen da sein. Mehrere Parameter werden durch Kommas getrennt. Auch für Parameternamen gelten dieselben Regeln wie für Variablennamen.

- Der Code der Funktion, der sogenannte Funktionsrumpf, steht zwischen geschweiften Klammern.
- Innerhalb der Funktion verhält sich ein Parameter genau wie eine Variable. Wird dem Parameter wie hier ein neuer Wert zugewiesen, dann ändert das nichts am Wert außerhalb der Funktion.
- Das Schlüsselwort **return** gibt einen Wert zum Aufrufer der Funktion zurück.
- Der mit **return** zurückgegebene Wert kann vom Aufrufer benutzt werden wie jeder andere Wert und zum Beispiel einer Variablen zugewiesen werden.

```
function quersumme(zahl){
    zahl = zahl.toString();
    var ergebnis = parseInt(zahl.charAt(0))+parseInt(zahl.charAt(1));
    return ergebnis;
}
var quersumme1 = quersumme(33);
var quersumme2 = quersumme(17);
```



- Der Bereich, in dem eine Variable sichtbar ist, heißt der **Scope** der Variablen. In JavaScript gibt es nur zwei Scopes: **Globale Variablen** sind überall sichtbar, das betrifft alle Variablen, die außerhalb von Funktionen deklariert werden. Variablen, die innerhalb einer Funktion deklariert werden, sind nur innerhalb der Funktion sichtbar und heißen **lokale Variablen**.
- Gibt es eine globale Variable und eine lokale Variable mit demselben Namen, dann wird innerhalb ihres Scopes die lokale Variable verwendet. Auf die globale Variable kann dann nicht zugegriffen werden.
- Nimmt eine Funktion Änderungen an globalen Variablen vor, nennt man das einen Seiteneffekt.
- Soweit möglich sollte man Funktionen ohne Seiteneffekte schreiben. Und es ist immer möglich, alle Seiteneffekte sind vermeidbar. Sie machen den Programmablauf schwer nachvollziehbar und führen gern zu schwer zu findenden Fehlern.

*1 Hier wird zuallererst geprüft, ob überhaupt ein Parameter übergeben wurde, ...

*2 ... und hier dann, dass der Parameter wirklich zweistellig ist.

```
function quersumme(zahl){  
  if (!zahl)*1 return NaN;*3  
  zahl = zahl.toString();  
  if (zahl && zahl.length == 2){*2  
    var ergebnis = parseInt(zahl.charAt(0)) + parseInt(zahl.charAt(1));  
    return ergebnis;  
  } else {  
    return NaN;*3  
  }  
}
```

*3 Falls etwas nicht stimmt, muss ich trotzdem etwas zurückgeben. Mit **NaN** signalisiere ich, dass kein richtiges Ergebnis berechnet werden konnte. Viel besser wäre es, eine **Fehlermeldung** zu erzeugen. Jetzt wo ich es sage, Fehler sind noch ein wichtiges Thema. Dazu erzähl ich dir besser gleich noch was.

PARSEINT(String, RADIX);

- Parameter
 - string
 - Umzuwandelnder Wert. Wenn string kein String ist, wird er zu einem String konvertiert (durch die abstrakte Operation ToString). Führende Leerzeichen im String werden ignoriert.
 - radix
 - Eine ganze Zahl zwischen 2 und 36, die die Basis eines mathematischen Zahlensystems ist, in der der String geschrieben ist. 10 steht für das gebräuchliche Dezimalsystem.
 - Rückgabewert
 - Eine ganze Zahl des übergebenen Strings. Wenn das erste Zeichen nicht zu einer Zahl konvertiert werden kann, wird NaN zurückgegeben.

- Variablen, die innerhalb einer Funktion definiert werden, sind auch nur innerhalb der Funktion sichtbar. Nicht nur das, die gelten auch nur für diesen Aufruf der Funktion. Beim nächsten Aufruf ist die Variable wieder komplett neu.
- Der Bereich, in dem eine Variable sichtbar ist, heißt der **Scope** der Variablen. In JavaScript gibt es nur zwei Scopes: **Globale Variablen** sind überall sichtbar, das betrifft alle Variablen, die außerhalb von Funktionen deklariert werden. Variablen, die innerhalb einer Funktion deklariert werden, sind nur innerhalb der Funktion sichtbar und heißen **lokale Variablen**.
- Gibt es eine globale Variable und eine lokale Variable mit demselben Namen, dann wird innerhalb ihres Scopes die lokale Variable verwendet. Auf die globale Variable kann dann nicht zugegriffen werden.
- Nimmt eine Funktion Änderungen an globalen Variablen vor, nennt man das einen Seiteneffekt. Soweit möglich sollte man Funktionen ohne Seiteneffekte schreiben. Und es ist immer möglich, alle Seiteneffekte sind vermeidbar. Sie machen den Programmablauf schwer nachvollziehbar und führen gern zu schwer zu findenden Fehlern

RETURN NAN

*1 Vorbedingung für `zahl1`: Der Parameter muss übergeben werden. Gibt es schon keinen ersten Wert, dann ist der Durchschnitt nicht berechenbar. Die etwas unschöne Und-Verknüpfung ist nötig, weil bei einem Wert von 0 weitergerechnet werden soll, obwohl 0 ebenfalls falsy ist. Der Unterschied zwischen `undefined` und 0 ist also wichtig.

*2 Jetzt solltest du mit Variablen arbeiten, sonst wird es sehr unübersichtlich.

```
function durchschnitt(zahl1, zahl2){  
  if (!zahl1 && zahl1 !== 0) return NaN;*1  
  var summe = zahl1;*2  
  var zaehler = 1;*2  
  if (zahl2 || zahl2 === 0){*3  
    summe += zahl2;*4  
    zaehler++;*5  
  }  
  return summe / zaehler;  
}
```

*4 Wenn es den zweiten Parameter gibt, dann wird er zur Summe addiert.

*3 Die Oder-Verknüpfung hat die gleiche Funktion wie die Und-Verknüpfung vorher und ist nur andersherum geschrieben. `undefined` ist ein Falsy-Wert. Blödsinn ist 0 aber auch falsy, und eine 0 dürfte ganz legitim übergeben werden. Deshalb die Oder-Verknüpfung: Der `if`-Block wird genau dann ausgeführt, wenn der Parameter die Zahl 0 oder ein beliebiger Truthy-Wert ist.

*5 Es muss jetzt auch gezählt werden, wie viele Werte eigentlich übergeben wurden, am Ende muss ja durch die Anzahl geteilt werden. `zaehler++` ist eine **Kurzschreibweise für die Kurzschreibweise** `zaehler += 1`. Ja, Entwickler sind **so** schreibfaul.



ARRAYS



- Arrays geben dir eine zusätzliche Dimension, du kannst Objekte damit stapeln und (fast) beliebig viele in eine Variable packen. Wenn es eine Liste gleichartiger Daten ist, dann kann es in einem Array stehen.
- Es ist zwar technisch möglich, in ein und demselben Array Strings, Zahlen und andere Objekte unterzubringen. **Eine gute Idee ist es aber nicht**, ein Array sollte nur gleichartige Daten enthalten, zum Beispiel ein Array von Zahlen oder ein Array von Strings.

ARRAYS

- In Einzelfällen gibt es gute Gründe, ein Array von Arrays von Arrays zu haben oder sogar noch tiefer zu verschachteln. Meistens aber nicht. Tu es nicht ohne einen wirklich guten Grund.

*1 ein leeres Array, zwei eckige Klammern mit nichts dazwischen

*2 Mehrere Werte werden durch Kommas getrennt. Achte bei Zahlen besonders darauf, dass das **Komma** die Elemente des Arrays voneinander trennt und der **Punkt** das Dezimaltrennzeichen ist.

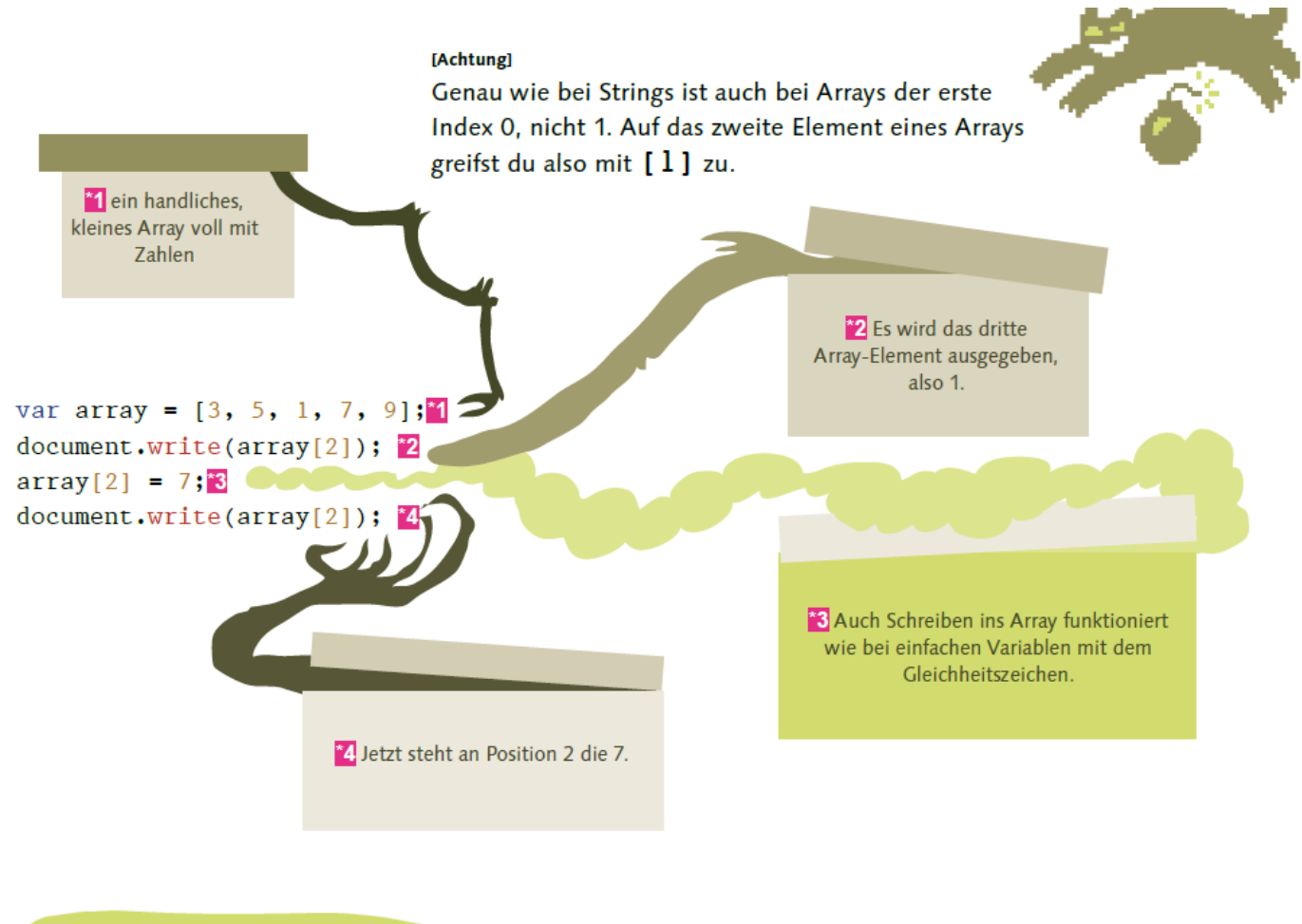
```
//Ein leeres Array. Ja, es gibt gute Gründe, ein leeres Array anzugeben  
var leer = [];*1  
//Ein Array von Zahlen, zum Beispiel um ihren Durchschnitt zu berechnen  
var zahlen = [7, 4, 3.8, 19];*2  
//Ein Array von Strings: Schrödingers World-of-Warcraft-Charaktere  
var charaktere = ["Schrökiller", "Schrödinator", "Schrödirella"];  
//Ein Array von Arrays: GPS-Koordinaten der Chinesischen Mauer*3  
var koordinaten = [[40.11666, 124.38333], [40.21966, 124.50915]]*4;
```

*3 Natürlich nicht der ganzen Mauer, nur zweier Punkte ganz im Osten. Den kompletten Verlauf in Koordinaten anzugeben, würde das Kapitel wohl sprengen.

*4 Arrays sind, du hast es schon erraten, auch selbst wieder Objekte. Und Objekte können Elemente in einem Array sein. In Fällen wie diesem ist es durchaus sinnvoll, Arrays zu verschachteln: Die inneren Arrays stellen jeweils ein Koordinatenpaar dar, das äußere Array ist eine Liste dieser Koordinatenpaare. Arrays in Arrays werden auch als mehrdimensionale Arrays bezeichnet, weil genau das ein häufiger Fall ist, bei dem sie eingesetzt werden: um mehrdimensionale Strukturen abzubilden. Denk zum Beispiel an ein Schachbrett mit seinen 8×8 Feldern. Ein äußeres Array enthält die Reihen, jede Reihe ist wiederum ein Array und enthält die Felder. Und schon kannst du mit `schachbrett[0][0]` deinen Turm finden.

ARRAYS

- Auch an Array-Positionen, denen kein Wert zugewiesen ist, steht **undefined**. Du kannst aus einem Array mit drei Elementen die 200. Stelle auslesen, ohne dass ein Fehler auftritt.



[Achtung]
Genau wie bei Strings ist auch bei Arrays der erste Index 0, nicht 1. Auf das zweite Element eines Arrays greifst du also mit [1] zu.

*1 ein handliches, kleines Array voll mit Zahlen

*2 Es wird das dritte Array-Element ausgegeben, also 1.

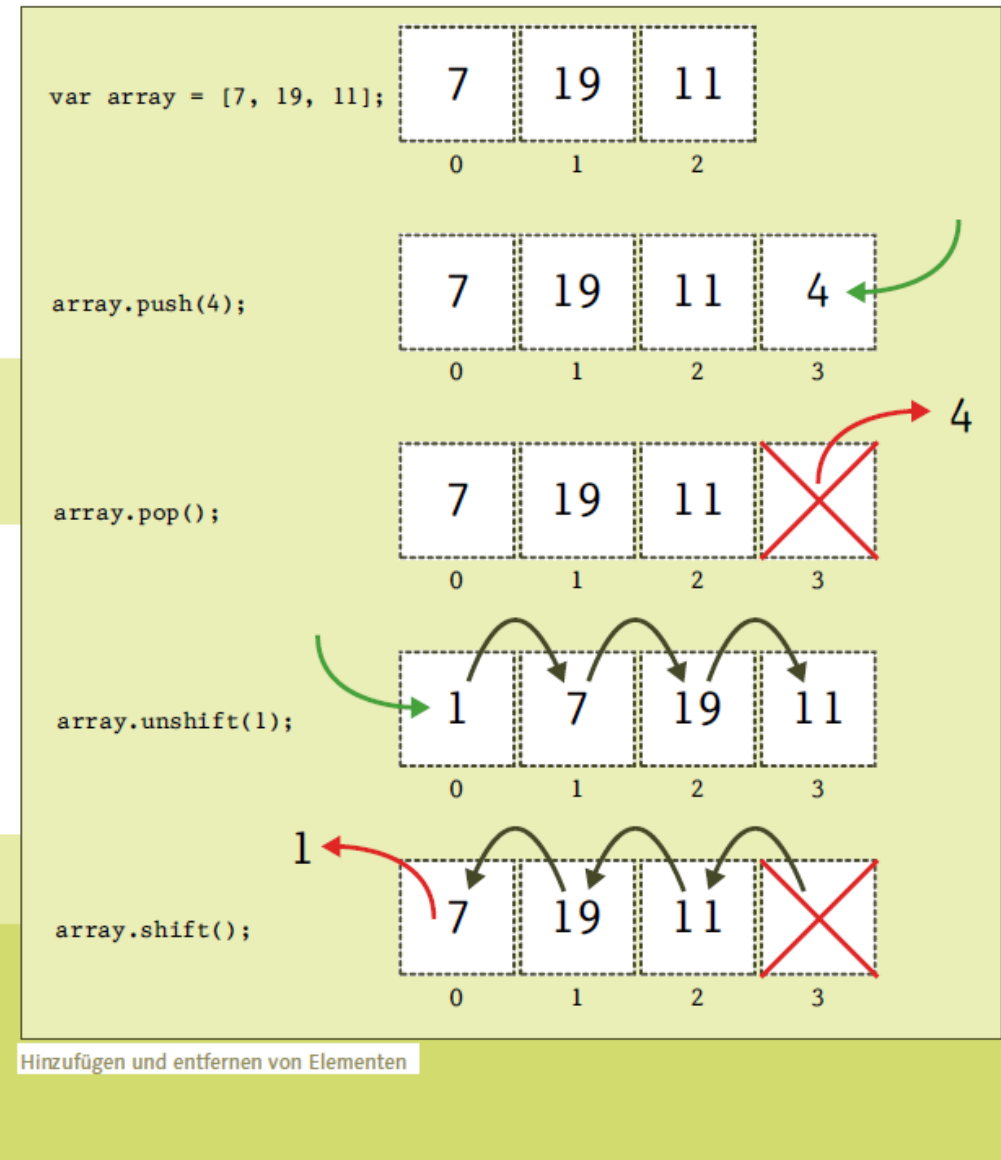
*3 Auch Schreiben ins Array funktioniert wie bei einfachen Variablen mit dem Gleichheitszeichen.

*4 Jetzt steht an Position 2 die 7.

```
var array = [3, 5, 1, 7, 9]; *1
document.write(array[2]); *2
array[2] = 7; *3
document.write(array[2]); *4
```

METHODEN EINES ARRAY-OBJEKTS

- Die meisten Methoden eines Array-Objekts fügen entweder Elemente hinzu oder entfernen sie. **push()** und **pop()** fügen ein Element am Ende hinzu und entfernen es wieder. **unshift()** und **shift()** tun dasselbe am Anfang und verschieben dabei alle weiteren Elemente um eine Position.
- Die beiden Methoden zum Entfernen von Elementen, **pop()** und **shift()**, geben auch das entfernte Element zurück. Außerdem interessant sind noch die Methoden **reverse()**, die ein Array umdreht, und **concat()**, die ein neues Array erzeugt, indem sie zwei Arrays zusammenfügt.



FOR

- **Anweisung 1** wird (einmalig) vor der Ausführung des Codeblocks ausgeführt.
- **Anweisung 2** definiert die Bedingung für die Ausführung des Codeblocks.
- **Anweisung 3** wird (jedes Mal) ausgeführt, nachdem der Codeblock ausgeführt wurde.

```
for (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```

```
for (i = 0; i < 5; i++) {  
    document.write(i);*5  
}
```

Der Variablenname **i** ist zwar nicht sehr aussagekräftig, aber er wird für Zählvariablen benutzt, seit es **for**-Schleifen gibt. Der Name ist aussagekräftig aus Tradition.

- Schreibe eine Funktion, die ein Array von Zahlen als Parameter erhält und die größte Zahl im Array zurückgibt
- Nein, keine Magie, aber ein Detail von JavaScript, das viele nicht kennen. Wenn eine Funktion mehr Parameter erhält, als sie erwartet, dann stehen diese im **arguments**-Objekt. Sind keine Parameter deklariert, dann landen alle übergebenen Parameter in **arguments**. So muss für den Aufruf der Funktion nicht jedes Mal ein Array erzeugt werden, man übergibt einfach so viele Zahlen, wie man möchte.
- Genau wie Arrays sollte man **arguments** nur dann benutzen, wenn alle Werte den gleichen Typ und die gleiche Bedeutung haben, so wie die Zahlen, deren Durchschnitt berechnet werden soll. Benutze es auf keinen Fall, wenn deine Parameter verschiedene Bedeutungen haben, so wie Name, Alter, Postleitzahl und Geschlecht. All das im **arguments**-Objekt zu übergeben, ist der sicherste Weg, Code vollkommen unlesbar zu machen.
- Man kann auf den Inhalt von **arguments** zwar mit dem Index in eckigen Klammern zugreifen wie bei einem Array, aber **arguments ist kein echtes Array**. Es kennt die eckigen Klammern und die Eigenschaft **length**, aber keine der Array-Funktionen.

- Andersherum kann man auch mehr Argumente übergeben, als die Funktion an Parametern deklariert. Das **arguments**-Array hatte schon immer etwas von Voodoo, und hat dazu noch den Nachteil, dass alle Argumente noch mal drinstehen und man die ersten Einträge überspringen muss. Dank der neuen Restparameter ist auch das eine Sache der Vergangenheit. Außer im Internet Explorer.

Diese Funktion verkettet eine beliebige Anzahl an Strings mit einem Trennzeichen zu einem langen String. Aber das ist nicht so wichtig. Wichtig ist der Restparameter. Wie gezeigt, gibst du ihn mit drei vorangestellten Punkten an.

(Funktioniert in)
Restparameter funktionieren in Edge, Firefox, Chrome und Safari ab Version 10.

```
function concat(trenner, ...strings){  
  if (strings.length == 0) return "";  
  let ergebnis = strings[0];  
  for (let i = 1; i < strings.length; i++){  
    ergebnis += trenner;  
    ergebnis += strings[i];  
  }  
  return ergebnis;  
}  
concat(",", "Tick", "Trick", "Track");
```

1 Mit drei vorangestellten Punkten machst du den letzten Parameter einer Funktion zum Restparameter. Er sammelt alle verbleibenden Argumente.

2 In der Funktion ist der Restparameter ein Array.

3 Mit diesem Aufruf kommt in `concat` das Komma als Parameter `trenner` an und das Array `["Tick", "Trick", "Track"]` als `strings`.

UNTERSCHIEDE ZWISCHEN REST PARAMETERN UND DEM ARGUMENTS OBJEKT

- Es gibt drei Hauptunterschiede zwischen Rest Parametern und dem **arguments** Objekt:
 - Rest Parameter sind nur diejenigen, die zu einem Namen gehören (z. B. in Funktionsausdrücken formal definierte Parameter), während das **arguments** Objekt alle übergebenen Argumente einer Funktion enthält.
 - Das **arguments** Objekt ist kein echtes Array, während Rest Parameter eine Array sind, was bedeutet, dass Methoden wie **sort**, **map**, **forEach** oder **pop** direkt angewendet werden können.
 - Das arguments Objekt hat zusätzliche, spezielle Funktionalität (wie die **callee** Eigenschaft).

- Fehler werden **geworfen**, nicht zurückgegeben. Dadurch ist klar, dass die Funktion auf eine andere Art verlassen wird, als bei einem regulären Ergebnis. Ein Ergebnis übergibt sie dir in die Hand, einen Fehler wirft sie dir vor die Füße.
- Benutze die **throw**-Anweisung, um einen Fehler zu werfen, wenn die übergebene Zahl nicht zweistellig ist.
- Fehler, die mit **throw** geworfen werden, werden auch als Ausnahmefehler oder **Exceptions** bezeichnet.

```
function quersumme(zahl){
  zahl = zahl.toString();
  if (zahl.length != 2) throw "Nur zweistellige Zahlen
erlaubt";*1
  var ergebnis = parseInt(zahl.charAt(0)) +
  parseInt(zahl.charAt(1));
  return ergebnis;
}
```

```
try {
  ...*2
} catch (error ){
  alert("Fehler: " + error);*5
}
```

Das Schlüsselwort **try** leitet einen Codeblock ein, in dem Fehler auftreten könnten, die behandelt werden sollen. Mit **catch** wird der Fehler gefangen. Falls im **try-Block** ein Fehler auftrat, und nur dann, wird der **catch-Block** ausgeführt.

AUFGABE – SPIEL HANDYWEITWURF

ANGABE 1

- In diesem Spiel sollst du mit deinem Handy ein Monster treffen
- Du musst dazu den Winkel und die Kraft eingeben mit der du das Handy wirfst um das Monster zu treffen
- Die Entfernung zum Monster darf nicht immer gleich sein
- Drei Versuche darf es geben
 - für Fortgeschrittene: die Anzahl an Versuchen kann vom User eingegeben werden
- Definiere die Fallbeschleunigung als Konstante
 - Für Fortgeschrittene: Lass den User eingeben ob er auf der Erde, dem Mond, Mars oder Jupiter ist. Dementsprechend wird die Fallbeschleunigung zur Berechnung genommen
- Gib dem User aus ob er getroffen hat oder wie weit weg das Monster noch ist und wie weit er geworfen hat
- Achte auf ein angenehmes Styling der Seite

AUFGABE – SPIEL HANDYWEITWURF

ANGABE 2

- `g` steht für die Fallbeschleunigung
der Wert `g` beträgt auf der
 - Erde 9.81 m/s²
 - Mond 1.62 m/s²
 - Mars 3.69 m/s²
 - Jupiter 24.79 m/s²
- `var v0` ist die Anfangsgeschwindigkeit und ist mit einem Formular und einem Inputfeld vom User einzugeben
- `var derWinkel` ist der Winkel in dem der User wirft. Es darf nur eine Zahl zwischen 1 und 90 sein. Dieser Wert ist auch mit einem Inputfeld in einem Formular vom User einzugeben
 - Die Berechnung die du hier brauchst:
`derWinkel = derWinkel * (Math.PI / 180)`
- `var wurfweite` berechnet sich aus dem Wurfwinkel und der Anfangsgeschwindigkeit
 - Die Berechnung sieht folgendermaßen aus:
`wurfweite = ((v0 * v0) * Math.sin(2 * derWinkel)) / g`
Achtung: Runde das Ergebnis
- `var i` ist die Zählervariable für die for Schleife. Sie muss vor der Schleife deklariert werden. Die for-Schleife brauchst du um die Versuche darzustellen
 - `for (i = 1; i < 4; i++) { ... }`
- `var entfernungZumMonster` speichert den Wert der zufällig gewählt wird. Er stellt die Entfernung zum Monster dar
 - `entfernungZumMonster = Math.random() * 90 + 10;`
Mit dieser Zeile generierst du eine zufällige Zahl
 - `entfernungZumMonster = Math.round(entfernungZumMonster)`
 - Hier wird der Wert noch gerundet um eine ganze Zahl zu erhalten

Weitere hilfreiche Angaben

- `Math.round()`
rundet eine Zahl
- `alert()`
gibt ein Meldungsfenster aus
- `if(wurfweite == entfernungZumMonster) { ... }`
fragt ab ob der User getroffen hat
- `prompt()`
gibt ein Fenster aus das zu einer Eingabe auffordert
- `Math.PI`
gibt die Zahl PI an
- `Math.sin`
berechnet den Sinus



CODERS.BAY

ENDE