JavaScript Grundlagen 2

Übersicht

- Kommentare
- Verzweigungen
  - If / else
  - Switch case
  - Ternäre Ausdrücke
- Funktionen
  - Einfache Funktionen
  - Parameter
  - Rückgabewerte
  - Anonyme Funktionen
  - Arrow Functions

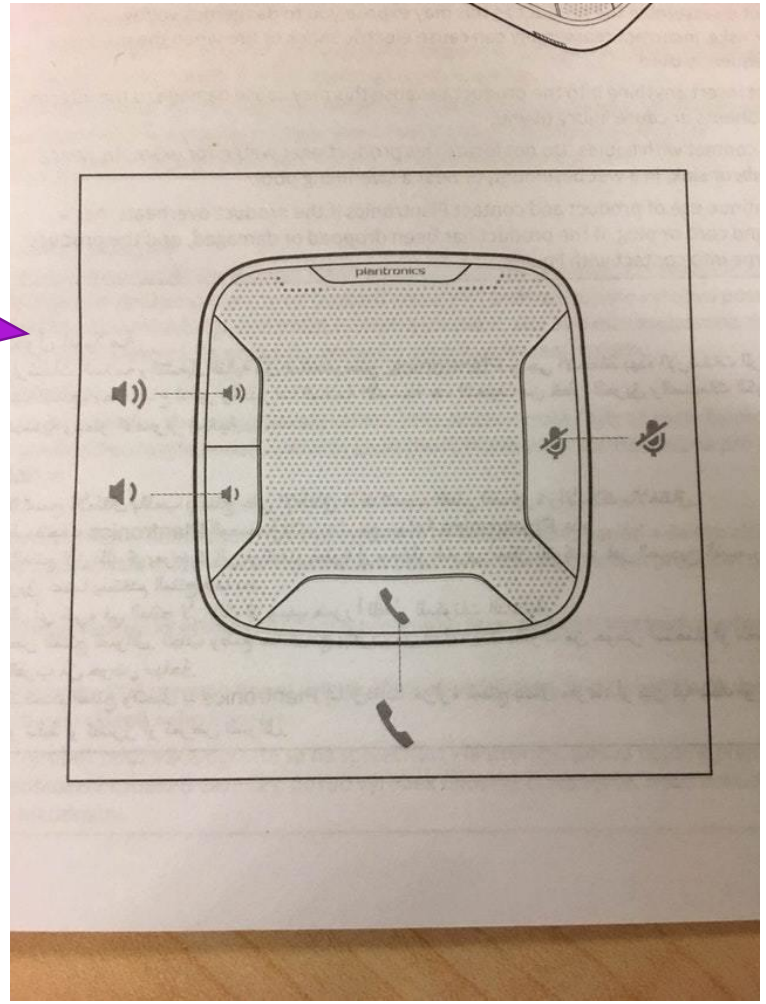
Kommentare

Kommentare im Code

- Man kann Kommentare im Code schreiben, die nicht als Code interpretiert werden
- Mit // wird der Rest der Zeile als Kommentar interpretiert.
- Mit /* wir alles bis zum folgenden */ als Kommentar interpretiert, auch mehrzeilig.

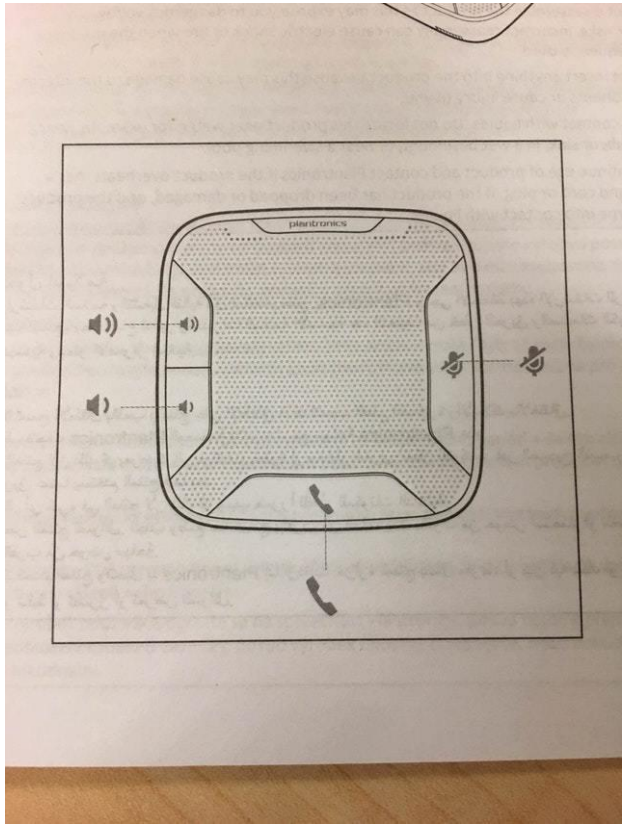
```
1 const myValue = 'Hello'; // dies ist ein Kommentar
2
3 const myOtherValue = /* auch ein Kommentar */ 'World';
4
5 /*
6 Mehrzeiliger Kommentar
7 console.log('Das hier siehst du nicht, es ist "Auskommentiert"');
8 */
9 console.log(myValue, myOtherValue);
```

Wie Entwickler häufig ihren Code kommentieren



Top 5 der worst practices für Kommentare

- **Irreführend**
  - Kommentare, die falsche oder irreführende Informationen enthalten und den Leser in die Irre führen können.
  - closeSocketWithTimeout(500); // close socket after 500ms
- **Überflüssig**
  - Kommentare, die offensichtlichen Code beschreiben oder Informationen enthalten, die bereits durch den Code selbst klar sind.
  - waitForEvent(buttonClick); // wait for button event
- **Veraltet**
  - Kommentare, die nicht mit dem aktualisierten Code synchronisiert sind und somit falsche oder verwirrende Informationen enthalten.
- **Unverständlich**
  - Kommentare, die unklar oder schlecht formuliert sind, und daher keinen klaren Kontext oder Sinn für den Code liefern.
- **Auskommentierter Code**
  - //sendMessage(„hello“);



Best practices für Kommentare

- Kommentare sollten…
  - das „Warum“ und nicht das „Was“ beschreiben
  - // Ergebnis der Berechnung wird verwendet um die Farben der Balken des Diagramms zu bestimmen
  - // Hier wird eine Berechnung durchgeführt bei der ein hsl-Wert durch die Anzahl der Schritte dividiert wird und so ein neuer Farbenwert erzeugt wird

```
var hslNumber = 120;
var hslNumberMax = 140;
while(hslnumber >= 0){
    chartcolors.push("hsl("+hslNumber+", 75%, 50%, 0.4)");
    bordercolors.push("hsl("+hslNumber+", 75%, 50%, 1)")
    hslNumber = hslNumber - Math.round(hslNumberMax/colorSteps);
}
```

- Überlege 2 mal
  - ob ein Kommentar wirklich notwendig ist
  - ob der Code verständlicher formuliert werden kann

“Code is like humor. When you have to explain it, it’s bad.”

Verzweigungen

Verzweigungen - if

- Möchte man abhängig von einer Bedingung entscheiden, wie es weiter geht, so helfen if (wenn) else (sonst) Konstrukte.
- Auf das Schlüsselwort if folgt eine Bedingung in Klammern ( ), und ein anschließender Code-Block (Scope) in Geschwungenen Klammern { }, der nur dann ausgeführt wird, wenn die Bedingung erfüllt ist

```
1 const testValue = 3; // ändere diesen Wert und überprüfe die Ausgabe
2
3 if (testValue < 5) {
4     console.log('Diesen Log siehst du, wenn der Wert kleiner 5 ist');
5 }
6
7 console.log('Diesen Log siehst du immer');
```

Verzweigungen - else

- Möchte man auf die nicht-Erfüllung der Bedingung anders reagieren, nutzt man das Schlüsselwort else (sonst).
- Es folgt auf den Code-Block des vorhergehenden if. Auf das Schlüsselwort else folgt keine Bedingung, wiederrum aber ein Code-Block in Geschwungenen Klammern { }, der ausgeführt wird, wenn die vorherige Bedingung nicht erfüllt ist

```
1 const testValue = 3; // ändere diesen Wert und überprüfe die Ausgabe
2
3 if (testValue < 5) {
4     console.log('Diesen Log siehst du, wenn der Wert kleiner 5 ist');
5 } else {
6     console.log('Sonst siehst du diesen Log')
7 }
8
9 console.log('Diesen Log siehst du immer');
```

Verzweigungen - else if

- Möchte man mehr als nur zwei Fälle unterscheiden, kann man mit else if eine weitere Bedingung prüfen, wenn die vorherige nicht erfüllt wurde.
- Auf das Schlüsselwort else if folgt eine Bedingung, und ein Code-Block.
- Es können mehrere else if folgen. Ein else ohne if kann nur am Ende einer Verzweigung stehen

```
1  const testValue = 3;
2
3  if (testValue < 5) {
4      console.log('kleiner als 5');
5  } else if (testValue <= 8) {
6      console.log('5 - 8');
7  } else if (testValue <= 12) {
8      console.log('8. - 12');
9  } else {
10     console.log('größer als 12');
11 }
```

Verzweigungen - switch case

- Möchte man zwischen vielen verschiedenen Exakten Werten unterscheiden, bietet sich das switch case an.
- Auf das Wort switch folgt in Klammern ( ) der Wert, der zu überprüfen ist.
- Es folgt ein Block { } in dem jeweils nach dem Wort case der auf Gleichheit zu prüfende Wert steht.
- Es folgt ein Doppelpunkt : und danach der Code für diesen Fall.
- Jeder Fall wird mit break; beendet bevor der nächste beginnt.

```
let fruit = 'Banane';

switch(fruit) {
  case 'Apfel':
    console.log('Äpfel sind rot.');
    break;
  case 'Banane':
    console.log('Bananen sind gelb.');
    break;
  case 'Orange':
    console.log('Orangen sind orange.');
    break;
  default:
    console.log('Ich kenne diese Frucht nicht.');
}
```

Verzweigungen - default case

- Auch bei switch case gibt es eine art "else" Teil
- ein Teil der ausgeführt wird, wenn keine der anderen Bedingungen zutrifft.
- Dieser wird mit dem Schlüsselwort **default** bezeichnet.
- Er steht immer als letzter Fall im switch.

```
let fruit = 'Banane';

switch(fruit) {
  case 'Apfel':
    console.log('Äpfel sind rot.');
    break;
  case 'Banane':
    console.log('Bananen sind gelb.');
    break;
  case 'Orange':
    console.log('Orangen sind orange.');
    break;
  default:
    console.log('Ich kenne diese Frucht nicht.');
}
```

Verzweigungen - fall-through

- Vergisst man das „break;“ am Ende eines Falls, so wird auch der Code des nächsten Falls ausgeführt, auch wenn dessen Bedingung nicht erfüllt wird.
- Man spricht dann von fall-through.
- Man kann es aber auch gewollt einsetzen, um mehrere Fälle gleich zu behandeln, wobei man dann für gewöhnlich die cases ohne Abstand untereinander schreibt

```
1  const value = 1;
2
3  switch (value) {
4      case 1:
5          console.log("1 oder...");
6          // code aber kein break
7          // -> vermeiden!
8      case 2:
9          console.log("2!");
10         break;
11     case 3: // das ist OK!
12     case 4:
13         console.log("3 oder 4");
14         break;
15 }
```

Verzweigungen - Ternäre Ausdrücke

- Möchte man einer Variable abhängig von einer Bedingung den einen oder anderen Wert zuweisen, so kann man dies mit einem Ternären Ausdruck machen.
- Dazu schreibt man auf der rechten Seite einer Zuweisung eine Bedingung
- gefolgt von einem ?
- gefolgt von dem Wert wenn die Bedingung zutrifft
- gefolgt von einem :
- gefolgt von dem Wert wenn die Bedingung nicht zutrifft

```
1 const myNumber = -5;
2
3 // Variable = Bedingung ? wenn wahr : wenn falsch;
4 const isPositive = myNumber >= 0 ? 'positiv' : 'negativ';
5
6 console.log(`Die Zahl war ${isPositive}`);
```

Vergleichsoperatoren

- Werte können in JavaScript mit bestimmten Operatoren verglichen werden Für Gleichheit (XNOR) nutzt man ===
- Für Ungleichheit (XOR) nutzt man !==
- Für den und (AND) Vergleich nutzt man &&
- Für den oder (OR) Vergleich nutzt man ||
- Für das Verneinen (NOT) nutzt man ein vorangestelltes !
- Bei Zahlen kann man Größer/-gleich (> / >=) und Kleiner/-gleich (< / <=) prüfen.

```
1 === 3;
1 !== 3;
true && false;
(1 !== 3) || (1 === 3);
!true;
!false;
3 > 3;
3 <= 3;
```

Typensicherheit beim Vergleich

- Wir haben gesehen, dass man für (Un-)Gleichheit die === bzw. !== Zeichen verwendet.
- Es gibt diese Operatoren aber auch ohne das letzte = Zeichen, als == bzw. !=
- Bei der Nutzung von == bzw. != wird nicht sichergestellt, dass beide Variablen auch den selben Typen haben
- Mit Hilfe der Funktion **typeof** kann der Datentyp einer Variable bestimmt werden
- Ressourcen: 7. Javascript/datatypes.js

```
let stringValue = "0";
let numberValue = 0;
stringValue == numberValue;
stringValue === numberValue;
stringValue != numberValue;
stringValue !== numberValue;
typeof(stringValue) //String
typeof(numberValue) //Number
```

```
do {
    // ... (weitere Code-Abschnitte)
    for (let p = buffer, i = 0; i < last; i++, p += 32) {
        // ... (weitere Code-Abschnitte)
        if (dirent.flags & ATTR_DIRECTORY) {
            // ... (weitere Code-Abschnitte)
            if (dirent.name === "..") {
                if (dir.parent) { /* XXX */
                    if (!dir.parent.parent) {
                        if (dirent.head) {
                            // ... (weitere Code-Abschnitte)
                            if (ask(1, "Correct")) {
                                // ... (weitere Code-Abschnitte)
                                if (boot.ClustMask === CLUST32_MASK)
                                    p[20] = p[21] = 0;
                                    mod |= THISMOD | FSDIRMOD;
                                }
                            } else {
                                mod |= FSERROR;
                            }
                        }
                    }
                    continue;
                }
                // ... (weitere Code-Abschnitte)
            } else {
                // ... (weitere Code-Abschnitte)
            }
        }
        // ... (weitere Code-Abschnitte)
    }
} while (cl = fat[cl].next >= CLUST_FIRST && cl < boot.NumClusters);
```

Nesting Depth

- Vermeide eine Verschachtelung von mehr als 3 Ebenen
- Code mit mehr als 3 Ebenen wird unverständlich und unleserlich
- Erschwert es Fehler zu finden
- Code ist nur mehr schwer wartbar

```
do {
    // ... (weitere Code-Abschnitte)
    for (let p = buffer, i = 0; i < last; i++, p += 32) {
        // ... (weitere Code-Abschnitte)
        if (dirent.flags & ATTR_DIRECTORY) {
            // ... (weitere Code-Abschnitte)
            if (dirent.name === "..") {
                if (dir.parent) { /* XXX */
                    if (!dir.parent.parent) {
                        if (dirent.head) {
                            // ... (weitere Code-Abschnitte)
                            if (ask(1, "Correct")) {
                                // ... (weitere Code-Abschnitte)
                                if (boot.ClustMask === CLUST32_MASK)
                                    p[20] = p[21] = 0;
                                    mod |= THISMOD | FSDIRMOD;
                                }
                            } else {
                                mod |= FSERROR;
                            }
                        }
                    }
                    continue;
                }
                // ... (weitere Code-Abschnitte)
            } else {
                // ... (weitere Code-Abschnitte)
            }
        }
        // ... (weitere Code-Abschnitte)
    }
} while (cl = fat[cl].next >= CLUST_FIRST && cl < boot.NumClusters);
```

4 Übungen zu Verzweigungen

1. Lies vom Nutzer eine Temperatur ein, nutze ein if/else-if/else um auszugeben, ob es kalt, angenehm oder heiß ist.
2. Lies einen einzelnen Buchstaben ein und gib aus, ob es ein Vokal oder Konsonant ist. Nutze dafür ein switch-case mit default case und fall-through.
3. Lese 3 Zahlen vom Benutzer ein, und gib die größte davon zurück. Nutze dabei Ternäre Ausdrücke.
4. Lies eine Zahl vom Benutzer ein, und gebe aus, ob diese gerade, ungerade oder eine Kommazahl ist

Funktionen

Grundlagen zu Funktionen

- Eine Funktion ist ein wiederverwendbarer Block Code.
- Eine Funktion beginnt mit dem Wort function, gefolgt vom Funktionsnamen, Klammern ( ) (in diesen können Parameter definiert werden) und dem Code-Block { }.
- Ein Wert kann mit dem Schlüsselwort return zurückgegeben werden.
- Ein Aufruf erfolgt über den Funktionsnamen gefolgt von Klammern ( ), in denen Werte (die den definierten Parametern zugeordnet werden) übergeben werden können.

```
function sumUp(a,b){
    const sum = a+b;
    console.log(sum)
    return sum;
}

const mySum= sumUp(5,3);
```

Einfache Funktionen

- Im einfachsten Fall hat eine Funktion weder Parameter, noch einen Rückgabewert.
- Hier genügt das Schlüsselwort function, der (beliebig wählbare) Funktionsname, Runde Klammern () und der Code-Block (Scope) in geschwundenen Klammern { ... } zur Definition.
- Beim Aufruf der Funktion mit dem Funktionsnamen und runden Klammern () wird der Code innerhalb der Funktion ausgeführt.
- Die Funktion kann beliebig oft ausgeführt werden..

```
2 function hello() {
3     console.log('Hallo');
4 }
5
6 // Benutzung
7 hello();
8 hello();
```

Funktionen - Parameter

- Man kann einer Funktion auch Werte übergeben.
- Dazu gibt man in der Funktionsdefinition innerhalb der Klammern ( ) an, welchen Variablennamen die Werte innerhalb der Funktion haben sollen (ein let ist hier nicht nötig!).
- Beim Aufruf der Funktion übermittelt man in den Klammern ( ) die gewünschten Werte, die auf die Variablen in der Funktion übertragen werden sollen.
- Die Reihenfolge der Parameter ist dabei sehr wichtig. Mehrere Parameter können mit Komma getrennt angegeben werden.

```
1 // Definition
2 function hello(name, country) {
3     console.log(`Hallo ${name} aus ${country}!`);
4 }
5
6 // Benutzung
7 hello('Anton', 'Österreich');
```

Funktionen - Defaultwert für Parameter

- Man muss nicht alle von der Funktion definierten Parameter beim Aufruf mit einem Wert befüllen
- Befüllt man einen Wert nicht, so hat dieser in der Funktion den Wert undefined.
- Man kann einen Standardwert für Parameter vergeben, dazu schreibt man hinter den Parameter in der Funktion = und den gewünschten Wert.

```
1 function test(a, b = 'Hallo') {
2     console.log(`${a}, ${b}`);
3 }
4
5 test('Anton', 'Berta');
6 test('Caesar');
7 test();
```

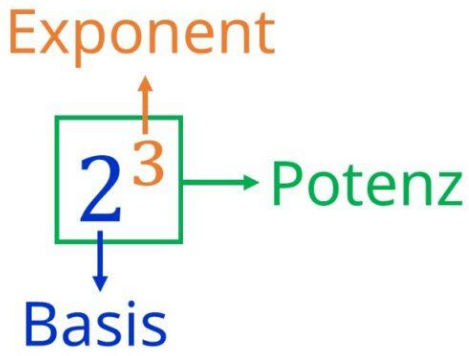
Funktionen - Rückgabewert

- Ein Rückgabewert wird nach Ausführung der Funktion an Stelle des Aufrufs eingesetzt, man kann es also beispielsweise einer Variable zuweisen.
- Mit return und einem darauf folgendem Wert gibt man in der Funktion einen Wert zurück.
- Die Funktion stoppt ihre Ausführung, sobald sie auf ein return stößt, Code der innerhalb der Funktion danach folgt, wird nicht mehr ausgeführt.

```
1 // Definition
2 function add(a, b) {
3     const sum = a + b;
4     return sum;
5 }
6
7 // Benutzung
8 const mySum = add(3, 5);
9 console.log(`Summe: ${mySum}`);
```

Übung zu Funktionen

- Erstelle eine Funktion „pow“ die 2 Zahlenwerte als Parameter übergibt
- Die Funktion soll den ersten Wert mit dem zweiten Wert potenzieren
- Wurde kein zweiter Wert mitgegeben, dann wird automatisch die erste Zahl quadriert



Anonyme Funktionen

- Funktionen können auch namenlos sein, man nennt sie dann "Anonyme Funktion„
- Man kann Funktionen genauso wie Texte und Zahlen in Variablen speichern, und sie dann über den Variablennamen ansprechen.

```
1 const a = function() {
2     console.log('Hello');
3 }
4
5 a();
6
7 const b = a;
8
9 b();
```

Arrow Functions

- ArrowFunctions verzichten auf das Schlüsselwort function und schreiben stattdessen => zwischen die Parameter-Klammern () und den Code-Block { ... }.
- Wird lediglich eine Berechnung angestellt, die sofort zurückgegeben wird, kann bei Arrow-Functions auf den Code-Block { ... } und das return verzichtet werden, man schreibt dann hinter den Pfeil => sofort den Rückgabewert.
- Hat eine ArrowFunction genau einen Parameter, kann sogar auf die () verzichtet werden.

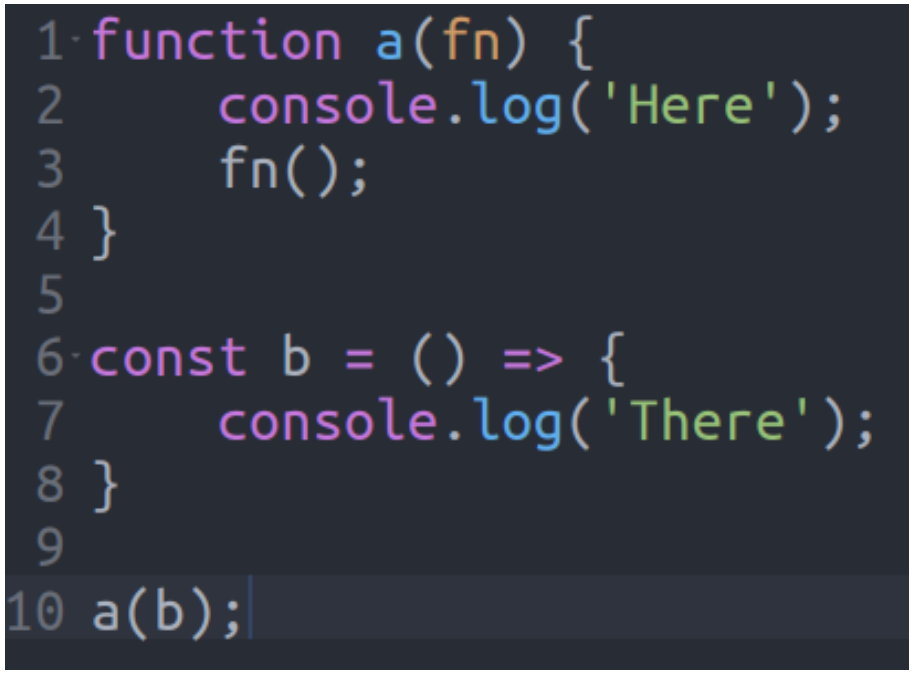
```
// Herkömmliche Funktion
function add(a, b) {
  return a + b;
}

// Arrow-Funktion mit implizitem Return
const addArrow = (a, b) => a + b;

// Aufruf der Funktionen
console.log(add(5, 3)); // Output: 8
console.log(addArrow(5, 3)); // Output: 8
```

Funktionen an Funktionen übergeben

- Da man Funktionen in Variablen speichern kann, kann man sie auch anderen Funktionen als Parameter übergeben.



Funktionen – Call by value

- In JavaScript werden primitive Datentypen wie Zahlen, Strings und Booleans per Wert übergeben
- Das bedeutet, dass beim Aufruf einer Funktion eine Kopie des Werts der Variable erstellt wird und die Funktion mit dieser Kopie arbeitet, ohne die ursprüngliche Variable zu ändern

```
// Definieren einer Funktion, die den Wert einer Variable ändert
function changeValue(val) {
  val = 5; // Ändert den Wert der lokalen Variable val
  console.log("Innerhalb der Funktion:", val);
}

// Definieren einer Variablen und Aufruf der Funktion
let num = 10;
console.log("Vor der Funktion:", num);
changeValue(num);
console.log("Nach der Funktion:", num);
```

Clean Functions

Clean Functions - Länge

- Wie groß darf eine Funktion sein?
  - Nicht größer als notwendig
  - Weniger als 100 Zeilen
- Wie viele Parameter darf eine Funktion haben
  - Im Idealfall < 3
  - Nicht mehr als wirklich notwendig
- Verwende Flag Parameter mit Bedacht
  - buyCar(brand, isElectric)
  - Alternative: in 2 Funktionen aufspalten
  - Bsp: buyElectricCar(brand), buyFuelCar(brand)
  - Bsp Funktionsaufruf: buyCar(„Audi“, true) vs buyElectricCar(„Audi“)

Clean Functions – Single Responsibility

- Schreibe Funktionen so, dass sie **spezifische und wiederverwendbare** Aufgaben erfüllen, die unabhängig voneinander funktionieren können
- Behandle nicht mehrere Aspekte in einer Funktion - eine Funktion sollte immer nur eine bestimmte Aufgabe haben
- Merkmal das eine Funktion nicht dem Single Responsibility Prinzip folgt
  - „Meine Funktion macht das … UND das … UND das …“
  - Funktionsname: ReadAndScaleTemperature

Übung zu Funktionen & Verzweigungen

Schreibe ein Programm, das herausfindet, ob ein durch den Benutzer eingegebenes Jahr ein Schaltjahr ist.

So lässt sich herausfinden, ob ein bestimmtes Jahr ein Schaltjahr ist:

- Ein Jahr ist ein Schaltjahr, wenn es durch 4 ohne Rest teilbar ist;
- außer wenn dieses Jahr auch durch 100 ohne Rest teilbar ist;
- es sei denn, dieses Jahr ist auch durch 400 ohne Rest teilbar.

Verwende dafür einen Funktionsaufruf indem das eingegebene Jahr als Parameter der Funktion übergeben wird

JavaScript Übungen als Wiederholung

- https://www.w3schools.com/js/exercise_js.asp?filename=exercise_js_functions1
- Functions
- Conditions
- Switch

Viel Erfolg beim Entwickeln!