TRY, CATCH, FINALLY UND THROW

try und catch

Bei try und catch handelt es sich um die Fehlerbehandlung. Ein Fehler beendet den Code und führt damit auch den darauffolgenden Code nicht mehr aus. Um dem entgegenzuwirken, kann man diese Fehler „abfangen“ (catch).

Man verwendet dafür einen try-catch-Block. In den try-Block wird der Code geschrieben, der tatsächlich zu einem Fehler führen könnte. Der catch-Block fängt den Fehler ab und gibt diesen aus. Anschließend lässt er das Programm weiterlaufen.

finally

In den Codeblock von **finally** wird der Code geschrieben, der am Ende auf jeden Fall noch ausgeführt werden soll.

Beispiel:

```
try {
    console.log(age)
} catch(e) {
    console.error(“Error abgefangen: “ + e)
} finally {
    console.log(“Nutzereingaben entfernen”)
}

console.log(“Hello World“)
```

throw

Mit throw besteht die Möglichkeit, selbst „Errors zu werfen“.

Beispiel:

```
try {

    var n = 500;

    if (n > 0 && < 1000) {
         console.log(“Number between 0 and 1000”)
    } else {
        throw new RangeError(“Number too small/big”)
    }

} catch(e) {

     if (e instanceof ReferenceError)
          console.log(“Referenz Fehler”);
     else if (e instanceof RangeError)
          console.log(“Range Fehler”);
}
```

WICHTIG Nicht jeder einzelne Aufruf muss in einen try-catch-Block gepackt werden. Das verschlechtert u. a. die Lesbarkeit.

PROMISES

Ein **Promise** (Versprechen) kann wie ein Objekt erstellt werden und zwei verschiedene Zustände haben:

1. Das Promise wurde erfüllt (resolve)
2. Das Promise wurde **nicht** erfüllt (reject)

Wenn das Versprechen erfüllt wird, der Code also funktioniert, ist dessen Ergebnis ein Wert. Andernfalls wird ein Fehler-Objekt zurückgegeben. Über ein Promise kann man nach dem „Erfüllen“ über die **then**-Funktion an das Ergebnis gelangen. Mit einem **catch**-Block kann ein Fehler im Promise abgefangen werden.

Beispiel:

```
let promise = new Promise((resolve, reject) => {

    let n = 50;

    if (n < 100) {
        resolve(“Versprechen gehalten”)
    } else {
        reject(“Versprechen NICHT gehalten”)
    }
})

   promise.then((res) => {
       console.log(res)
   }).catch((e) => {
       console.log(“Error: “ + e)
   })
```

setTimeout UND setInterval

Timeouts werden verwendet, um nach einer gewissen Zeit Code auszuführen. Der folgende Code wird nach 3 Sekunden (3000 Millisekunden) ausgeführt:

```
setTimeout(() => {
    console.log(“Zeige nach 3 Sekunden”);
}, 3000);
```

WICHTIG Code, der unter die setTimeout-Methode geschrieben wurde, wird bereits **vorher** ausgeführt.

Intervalle wiederholen sich immer wieder in gewissen Zeitabständen. Der folgende Code erhöht die Variable ticker jede Sekunde um 1:

```
var ticker = 0;

setInterval(() => {
    ticker++;
    console.log(ticker);
}, 1000);
```

CALLBACKS

Eine **Callback Funktion** ist eine Funktion, die einer anderen Funktion als Parameter übergeben und von dieser erst später mit definierten Argumenten aufgerufen wird.

Beispiel:

```
function main(){

     frisieren(abgeschlossen)

}

function abgeschlossen(preis){
     console.log(“Finaler Preis: “ + preis);
}

function frisieren(callback){
     console.log(”Hund fertig frisiert”);
     callback(10);
}
```

//Ausgabe:
Hund fertig frisiert
Finaler Preis:
10

ARRAY FILTER UND MAP

Mithilfe der **filter-Funktion** kann durch ein Array iteriert und bestimmte Elemente herausgefiltert werden. Dabei wird ein neues Array erstellt. Im folgenden Beispiel sollen nur die Nummern behalten werden, die kleiner als 10 sind:

```
let numbers = [1, 5, 3, 29, 83, 52, 10, 8, 4]

let smallNumbers = numbers.filter(n =>{
    return n < 10
});

console.log(smallNumbers);
```

//Ausgabe: [1, 5, 3, 8, 4]

Bei der **map-Funktion** wird auch über jedes Element des Arrays iteriert. Dabei kann man die Elemente **verändern**. Auch bei dieser Funktion wird ein neues Array erstellt. Das folgende Beispiel ist eine Ergänzung an das vorherige Beispiel:

```
let doubledNumbers = numbers.map(n => n * 2);
console.log(doubledNumbers);
```

//Ausgabe: [2, 10, 6, 58, 166, 104, 20, 16, 8]