Übung 04

1 Fortsetzung JavaScript

Eine örtliche Wetterstation sammelt regelmäßig Daten über Temparatur, Windstärke und Luftfeuchtigkeit. Eine tabellarische Übersicht der gesammelten Daten zwischen dem 01.02.2020 und 08.02.2020 ist wie folgt gegeben:

Datum	01.02	02.02	03.02	04.02	05.02	06.02	07.02	08.02
Temparatur(Celsius)	9°	7°	7°	5°	8°	12°	11°	13°
Windstärke(km/h)	61	52	45	49	55	75	80	63
Luftfeuchtigkeit(Prozent)	41%	38%	53%	65%	57%	42%	47%	50%

- Erstellen Sie zunächst eine benannte arrow-Funktion namens `checkDatum`, die zwei Datums-Werte als Parameter entgegen nimmt und `true` zurück gibt, wenn *Datum*1 <= *Datum*2 gilt (also `Datum1` vor `Datum2` liegt). Nutzen Sie dafür die `getTime`-Funktion von `Date`-Objekten. Die Dokumentation zu `Date` finden Sie online[1].

- Überführen Sie nun die Daten der sortierten Tabelle in ein 2D-Array als Datenstruktur. Damit Sie mit einer `for ... of`-Schleife besser über Ihren Datensatz iterieren können, erstellen Sie für den Datensatz anschließend einen Iterator. Nutzen Sie dafür den aus der Vorlesung bekannten `Symbol.iterator`. Bei jedem Iterationsschritt werden Datum, Temperatur, Windstärke und Luftfeuchtigkeit als Objekt zurückgegeben.

- Iterieren Sie mit einer `for ... of`-Schleife über den Datensatz. Finden Sie mit der zuvor erstellten Funktion `checkDatum` alle Daten die vor dem 05.02.2020 aufgezeichnet wurden und Speichern Sie diese in eigene Array-Variablen außerhalb der Iteration. Nutzen Sie die `push`-Funktion um die Arrays dynamisch zu befüllen.

[1] https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Date

- Schreiben Sie nun eine arrow-Funktion namens `calcMittelwert`, welche eine beliebige Anzahl an numerischen Werten übergeben bekommt, den Mittelwert berechnet und zurückgibt. Ermitteln Sie mit dieser Funktion den Mittelwert der Temparatur, Windstärke und Luftfeuchtigkeit zwischen dem 03.02.2020 und den 07.02.2020.

Bevor wir mit der Bearbeitung beginnen, erstellen wir uns zum testen unseres Codes ein Html-Dokument, welches unseren Code ausführt.

```
<!DOCTYPE html>
<html lang="de">
  <head>
    <meta charset="utf-8">
    <title>Fortsetzung JS</title>
  </head>
  <body>
    <script src="wetter.js"></script>
  </body>
</html>
```

Anschließend erstellen wir eine JavaScript Datei im selben Ordner, in dem sich auch unser Html-Dokument befindet. Damit unser Html-Dokument die JavaScript-Datei auch findet, benennen wir die JavaScript-Datei "wetter.js", wie im Html-Dokument innerhalb des script-tags angegeben. Beginnen wir nun mit der Bearbeitung der eigentlichen Aufgabe. In Schritt 1 sollen wir eine arrow-Funktion mit dem Namen `checkDatum` erstellen, welche zwei Datums-Werte als Parameter entgegen nimmt und `true` zurück gibt, wenn $Datum1 <= Datum2$ gilt. Arrow-Functions sind eine verkürzte Schreibweise für den normalen JavaScript Funktionen Syntax. Dies sieht in unserer Aufgabe dann wie folgt aus:

```
const checkDatum =  (datum1, datum2) => {
  return datum1.getTime() <= datum2.getTime();
};
```

Die Methode `getTime` eines `Date`-Objektes gibt uns einen Wert zurück, der die vergangene Zeit in Millisekunden seit dem 01.01.1970 repräsentiert. Um diese Methode nutzen zu können, erstellen wir zum Testen unserer Funktion zwei `Date`-Objekte und führen unsere Funktion aus.

```
let dateA = new Date(2020, 2, 1);
let dateB = new Date(2020, 2, 3);

console.log(checkDatum(dateA, dateB));
```

`console.log` gibt den Wert seines Parameters in der Console aus. Auf diesen Weg können wir also nun sehen, das unsere Funktion den Wert `true` zurückgibt und somit korrekt funktioniert. Im nächsten Schritt wollen wir mithilfe des `Symbol.iterator` über unseren Datensatz iterieren. Hierfür müssen wir zuerst den Datensatz in unseren Code überführen:

```
let wetterDaten = [
```

```
  [new Date(2020, 2, 1), 9, 61, 41],
  [new Date(2020, 2, 2), 7, 52, 38],
  [new Date(2020, 2, 3), 7, 45, 53],
  [new Date(2020, 2, 4), 5, 49, 65],
  [new Date(2020, 2, 5), 8, 55, 57],
  [new Date(2020, 2, 6), 12, 75, 42],
  [new Date(2020, 2, 7), 11, 80, 47],
  [new Date(2020, 2, 8), 13, 63, 50]
];
```

Da wir nun unsere Daten in einem Array gespeichert haben, können wir ein Iterator-Objekt erstellen, mit dem wir leichter über den Datensatz mit einer `for ... of`-Schleife iterieren können. Wir erstellen das Objekt wie folgt:

```
const wetterDatenIterator = {};
```

Mit diesem Code haben wir nun ein leeres wetterDatenIterator-Objekt erstellt. Dieses wollen wir nun mit einem `´Symbol.iterator` füllen. Hierfür benötigen wir folgende Elemente:

- Eine Index-Variable, die den aktuellen Iterationsschritt angibt.
- Einen `Symbol.iterator`
- Eine `next`-Funktion, die angibt, was in jedem Iterationsschritt ausgeführt werden soll
- Eine Abfrage, ob der Iterator am Ende des zu überprüfenden Arrays angekommen ist

```
const wetterDatenIterator = {
currentIndex : 0,
[Symbol.iterator]: () => {
  return{
    next: () =>{
      if (wetterDatenIterator.currentIndex < wetterDaten.length){
        wetterDatenIterator.currentIndex++;
        return {
          value: {
            datum : wetterDaten[wetterDatenIterator.currentIndex-1][0],
            temperatur : wetterDaten[wetterDatenIterator.currentIndex-1][1],
            windstaerke : wetterDaten[wetterDatenIterator.currentIndex-1][2],
            luftfeuchtigkeit :
            ↪  wetterDaten[wetterDatenIterator.currentIndex-1][3]},
            done : false}
        }
      return{ done: true }
    }
  }
}};
```

Betrachten wir das oben dargestellte Objekt nun etwas genauer. Das Objekt besitzt zwei Attribute, zum einen den `currentIndex`, welcher bei 0 startet und später hochgezählt wird, zum anderen den `Symbol.iterator`. Der `Symbol.iterator` ist selber eine Funktion, die

den Wert der `next`-Funktion zurückgibt. Die *next*-Funktion gibt zwei Werte zurück, erstens den eigentlichen Wert, in unserem Fall die Wetterdaten, und zweitens einen Wert *done*, der angibt ob es noch weitere Werte gibt über die Iteriert wird.

Im nächsten Schritt wollen wir unser neues Iterator-Objekt für eine `for ... of`-Schleife nutzen. Hierbei sollen wir die verschiedenen Wetterdaten seit dem 05.02.2020 in eigene Arrays speichern. Erstellen wir also zuerst die Arrays, in denen wir unsere Daten speichern:

```
let subSetTemparatur = [];
let subSetWindstaerke = [];
let subSetLuftfeuchtigkeit = [];
```

Nun können wir unsere `for ... of`-Schleife erstellen. Hierbei nutzen wir auch unsere `checkDatum`-Funktion um alle Daten vor dem 05.02.2020 nicht in unseren Arrays zu speichern. Um die Daten zu speichern verwenden wir die `push`-Funktion von Arrays, die ein neues Element an ein Array anhängt.

```
for(const wetter of wetterDatenIterator){
  if(checkDatum(wetter.datum, new Date(2020, 2, 5))){
    subSetTemparatur.push(wetter.temperatur);
    subSetWindstaerke.push(wetter.windstaerke);
    subSetLuftfeuchtigkeit.push(wetter.luftfeuchtigkeit);
  }
}
```

Zuletzt wollen wir noch eine neue Funktion `calcMittelwert` implementieren, welche eine beliebige Anzahl an numerischen Werten übergeben bekommt, den Mittelwert berechnet und zurückgibt. Hierfür übergeben wir unserer Funktion ein Array mit Werten und nutzen eine `for`-Schleife um den Mittelwert der übergebenen Werte zu errechnen.

```
const calcMittelwert = (werte) => {
  let sum = 0.0;
  for(let i in werte){
    sum += Number.parseFloat(werte[i]);
  }
  return (sum / werte.length);
};
```

Zum Testen unserer neuen Funktion wollen wir uns die Mittelwerte der Temperatur, Windstärke und Luftfeuchtigkeit über den Zeitraum vom 03.02.2020 bis zum 07.02.2020 in der Konsole ausgeben lassen. Hierfür nutzen wir unser bereits erstelltes Arrays und füllen es neu, mithilfe der bereits vorhandenen `for ... of`-Schleife, wir passen allerdings die `if`-Bedingung so an, dass die Wetterdaten dem neuen Zeitraum entsprechen:

```
for(const wetter of wetterDatenIterator){
  if(checkDatum(new Date(2020, 2, 3), wetter.datum) &&
  checkDatum(wetter.datum, new Date(2020, 2, 7))){
    subSetTemparatur.push(wetter.temperatur);
    subSetWindstaerke.push(wetter.windstaerke);
```

```
    subSetLuftfeuchtigkeit.push(wetter.luftfeuchtigkeit);
  }
}
```

Anschließend lassen wir uns die Mittelwerte über die Konsole mit *console.log* ausgeben:

```
console.log("Tem. Mittelwert: " + calcMittelwert(subSetTemparatur));
console.log("Wind. Mittelwert: " + calcMittelwert(subSetWindstaerke));
console.log("Luft. Mittelwert: " + calcMittelwert(subSetLuftfeuchtigkeit));
\end{lstlisting}
```

2 Aufgaben

Aufgabe 1

Die folgenden vier Anweisungen geben `true` aus, wenn an Stelle der `???` ein passendes Objekt übergeben wird. Finden und erstellen Sie die passenden Objekte, sodass alle vier Anweisungen mit `true` beenden.

Listing 1

```
console.log(??? instanceof Window);
console.log(??? instanceof Document);
console.log(??? instanceof Object);
console.log(??? instanceof CustomObject);
```

Aufgabe 2

Erstellen Sie eine Funktion `addTax(amount, taxRate, callback)`, welche einen übergebenen Steuersatz auf einen ebenfalls übergebenen Geldbetrag aufrechnet. Nutzen Sie das Callback-Muster um das Ergebnis der Funktion bei Aufruf auf unterschiedliche Weise zu verarbeiten. So soll ein Aufruf der Funktion das Ergebnis auf die Browser-Konsole ausgeben (`console.log()`), bei einem anderen Aufruf soll das Ergebnis als <p>-Element an den body der Webseite angehangen werden.

Sorgen Sie außerdem dafür, dass der übergebene Callback nur ausgeführt wird, wenn er eine Funktion ist.

Hinweis: Verwenden Sie anonyme Funktionen

Aufgabe 3

Schreiben Sie eine Funktion `scoping()` in deren ersten Anweisungen eine Variable `var v = "Hello"` und eine Konstante `const C = "Function"` deklariert und initialisiert werden. Darauf folgend soll zwei mal (innerhalb der Funktion) `console.log(v, C)` aufgerufen werden. Bei einem Aufruf soll „Hello Function“ ausgegeben werden, ein weiterer Aufruf soll jedoch „Hello Block“ auf der Konsole anzeigen.

Lösungs zu den Aufgabes

Lösung 1

Listing 2

```
var obj = {}; // Objekt erstellt mit Literal-Schreibweise
function CustomObject() {}; // leerer Objektkonstruktor CustomObject
var obj2 = new CustomObject(); // erzeuge Instanz von CustomObject

console.log(this instanceof Window); // true
console.log(this.document instanceof Document); // true
console.log(obj instanceof Object); // true
console.log(obj2 instanceof CustomObject); // true
```

Lösung 2

Listing 3

```
function addTax(amount, taxRate, callback) {
    var taxesAdded = amount + amount * (taxRate / 100);
    // prueft, ob callback eine Funktion ist
    if (typeof callback === "function") {
        callback(taxesAdded);
    }
}
// Callback erzeugt p-Element
addTax(100, 19, function(result) {
    var par = document.createElement("p");
    par.innerText = result; // 119
    document.body.appendChild(par);
});
// Callback mit console.log()
addTax(100, 19, function(result) {
    console.log(result); // 119
});
```

Lösung 3

Listing 4

```
function scoping() {
    var v = "Hello";
    const C = "Function";

    // jede Art von Block innerhalb der Funktion erlaubt
    // 'let C' ohne Konflikt mit der Konstante.
    function magic() {
        let C = "Block";
        console.log(v, C);
```

```
    }
    magic();

    // alternativ auch als anonyme selbst-aufrufende Funktion.
    (function() {
        let C = "Block";
        console.log(v, C);
    })();

    // auch innerhalb von if hat const C keine Gueltigkeit mehr.
    if (true) {
        let C = "Block";
        console.log(v, C);
    }
    console.log(v, C);
}
scoping();
```