Tag 5 - JavaScript: erweiterte Funktionen

Parameter-Validierung

Warum Parameter validieren?

In JavaScript können Funktionen mit beliebigen Werten aufgerufen werden, unabhängig davon, was sie erwarten. Daher ist es sinnvoll, zu Beginn einer Funktion die **Gültigkeit der Parameter zu überprüfen**. Dies verhindert fehlerhafte Berechnungen, nicht erwartetes Verhalten und kann nützliche Fehlermeldungen liefern.

Beispiel – Temperaturmittelwert nur berechnen, wenn ein Array übergeben wurde:

```
function getMeanTemp(temperatures) {
  if (!(temperatures instanceof Array)) {
    return NaN;
  }
  let sum = 0;
  for (let i = 0; i < temperatures.length; i++) {
    sum += temperatures[i];
  }
  return sum / temperatures.length;
}

console.log(getMeanTemp(10)); // → NaN (Fehlertyp)
console.log(getMeanTemp([10, 30])); // → 20
```

Beispiel – Prüfung auf Array:

```
function getMeanTemp(temperatures) {
  if (!Array.isArray(temperatures)) {
    return "Fehler: Bitte ein Array übergeben.";
  }
  let sum = 0;
  for (let i = 0; i < temperatures.length; i++) {
    sum += temperatures[i];
  }
  return sum / temperatures.length;
}
```

Zusätzlicher Fall: Leeres Array behandeln

```
function getMeanTemp(temperatures) {
  if (!Array.isArray(temperatures) || temperatures.length === 0) {
    return "Fehler: Array ungültig oder leer.";
  }
  let sum = 0;
  for (let i = 0; i < temperatures.length; i++) {
    sum += temperatures[i];
  }
  return sum / temperatures.length;
}
```

Nutzen der Validierung

- Verhindert unerwartetes Verhalten
- Macht Funktionen robuster und fehlerresistenter
- Erhöht die Lesbarkeit und Wartbarkeit durch klares Fehlermanagement

Rekursive Funktionen

Rekursion bedeutet, dass sich eine Funktion **selbst innerhalb ihres eigenen Funktionskörpers aufruft**. Sie wird häufig verwendet, wenn ein Problem in gleichartige, kleinere Teilprobleme zerlegt werden kann.

Zwei Bestandteile:

1. **Abbruchbedingung** (base case): verhindert unendliche Aufrufe
2. **Rekursiver Aufruf**: die Funktion ruft sich mit verändertem Parameter wieder auf

Beispiel – Fakultät (iterativ):

```
function factorial(n) {
  let result = 1;
  while (n > 1) {
    result *= n;
    n--;
  }
  return result;
}
console.log(factorial(6)); // → 720
```

Beispiel – Fakultät (rekursiv):

```
function factorial(n) {
  return n > 1 ? n * factorial(n - 1) : 1;
}
console.log(factorial(6)); // → 720
```

Vorsicht bei Rekursion:

- Es muss **immer eine Abbruchbedingung** geben (hier: `n <= 1`)
- Unbeendete Rekursion führt zu einem **Stack Overflow** (Programmabsturz)
    - Jeder Aufruf verbraucht Speicherplatz auf dem Aufruf-Stack
    - Bei zu vielen Aufrufen (z. B. `fibonacci(1000)`) kann ein Stack Overflow entstehen

Wann Rekursion sinnvoll ist:

- Wenn sich das Problem natürlich rekursiv beschreiben lässt (z. B. bei Bäumen, rekursiven Datenstrukturen, mathematischen Reihen)
- Wenn der Code dadurch lesbarer wird

Funktionen als Werte – First-Class Citizens

In JavaScript sind Funktionen **First-Class Citizens**, d. h.:

- Sie können in **Variablen gespeichert**,
- als **Argumente übergeben**,
- und von anderen Funktionen **zurückgegeben** werden.

Beispiel:

```
function showMessage(message) {
  console.log("Message: " + message);
}
let sm = showMessage;
sm("Hallo!");
console.log(typeof sm); // → function
```

Wichtig: **Ohne Klammern** wird die Referenz gespeichert, **mit Klammern** wird die Funktion aufgerufen.

Funktionen als Parameter – Callbacks

Funktionen können an andere Funktionen übergeben werden – z. B. zur dynamischen Steuerung von Verhalten.

Beispiel – Zwei Rechenfunktionen:

```
function add(a, b) {
  return a + b;
}
function multiply(a, b) {
  return a * b;
}

function operation(func, a, b) {
  return func(a, b);
}

console.log(operation(add, 5, 10));      // → 15
console.log(operation(multiply, 5, 10)); // → 50
```

Funktionsausdrücke (Function Expressions)

Statt eine Funktion mit `function name(...)` zu deklarieren, kann man sie auch **anonym** oder **benannt** direkt einer Variablen zuweisen:

Benannter Ausdruck:

```
let myAdd = function add(a, b) {
  return a + b;
};
```

Anonymer Ausdruck:

```
let myAdd = function(a, b) {
  return a + b;
};
```

Beide Varianten können ganz normal verwendet werden:

```
console.log(myAdd(3, 4)); // → 7
```

Arrow Functions – Kurzschreibweise

Was ist eine Arrow Function?

Arrow Functions (=>) sind eine kompakte Schreibweise für Funktionsausdrücke. Sie sind besonders nützlich für **kurze Funktionen und Callbacks**.

Basisform:

```
let add = (a, b) => {
  return a + b;
};
```

Kompakte Variante (eine Zeile):

```
let add = (a, b) => a + b;
```

Ein Parameter:

```
let quadriere = x => x * x;
console.log(quadriere(4)); // → 16
```

Rekursive Funktion als Arrow:

```
const fakultaet = n => n > 1 ? n * fakultaet(n - 1) : 1;
```

Typischer Anwendungsfall:

```
let namen = ["Anna", "Ben", "Clara"];
namen.forEach(name => console.log("Hallo " + name));
```

→ Gibt alle Elemente der Liste aus

Callbacks

Was ist ein Callback?

Ein **Callback** ist eine Funktion, die **an eine andere Funktion übergeben wird**, um später von dieser Funktion aufgerufen zu werden. Es ist also eine **Funktion als Argument**.

Wofür werden Callbacks verwendet?

- Um Verhalten dynamisch zu definieren (z. B. wie ein Element dargestellt wird)
- Um **asynchrone Reaktionen** zu modellieren (z. B. auf einen Klick, Serverantwort oder Timer)
- Um **Wiederverwendbarkeit und Flexibilität** zu erhöhen

Beispiel: Verarbeitung mit Callback

```
function verarbeiteDaten(daten, callback) {
  console.log("Verarbeite: " + daten);
  callback(daten);
}

verarbeiteDaten("Text", function(d) {
  console.log("In Großbuchstaben: " + d.toUpperCase());
});
```

Vorteile:

- Trennung von Logik und Ablaufsteuerung
- Wiederverwendbare Bausteine
- Grundlage für Event Handling & Libraries wie jQuery, React, Node.js

---

Asynchrone Funktionen und Callbacks

Was bedeutet asynchron?

Asynchrone Funktionen starten einen Prozess, der nicht sofort abgeschlossen ist. Währenddessen wird der restliche Code **weiter ausgeführt**. Ein **Callback** legt fest, was geschehen soll, **wenn die asynchrone Operation fertig ist**.

Synchronous vs. Asynchronous Callbacks

Synchronous:

```
let inner = function() {
  console.log("inner 1");
};
let outer = function(callback) {
  console.log("outer 1");
  callback();
  console.log("outer 2");
};

console.log("test 1");
outer(inner);
console.log("test 2");
```

→ Die Reihenfolge ist vorhersagbar:

```
```

```
test 1
outer 1
inner 1
outer 2
test 2
```

Asynchronous Callbacks mit `setTimeout`

```
let inner = function() {
  console.log("inner 1");
};
let outer = function(callback) {
  console.log("outer 1");
  setTimeout(callback, 1000);
  console.log("outer 2");
};

console.log("test 1");
outer(inner);
console.log("test 2");
```

→ Ausgabe:

```
test 1
outer 1
outer 2
test 2
(inner 1 – verzögert)
```

Beispiel: Zeitverzögerung mit `setTimeout`

```
console.log("Start");
setTimeout(() => {
  console.log("Nach 2 Sekunden");
}, 2000);
console.log("Ende");
```

`setInterval` und `clearInterval`

Wiederholtes Ausführen:

```
let inner = () => console.log("tick");
let timerId = setInterval(inner, 1000);
setTimeout(() => clearInterval(timerId), 5500);
```

→ `tick` wird 5x ausgeführt

Beispiel: Wiederholung mit `setInterval`

```
let counter = 0;
let intervalID = setInterval(() => {
  console.log("Tick: " + counter);
```

```
  counter++;
  if (counter > 4) clearInterval(intervalID);
}, 1000);
```

Beispiel: Reaktion auf Benutzeraktion

```
function frage(callback) {
  let name = prompt("Wie heißt du?");
  callback(name);
}
frage(function(n) {
  alert("Hallo, " + n);
});
```

Benutzerinteraktionen (Einblick)

```
window.addEventListener("click", function() {
  console.log("geklickt!");
});
```

→ Funktion wird **asynchron** bei Klick auf das Fenster aufgerufen

Weitere Anwendungen:

- Daten vom Server empfangen (AJAX, Fetch API)
- Dateioperationen (in Node.js)
- Reaktion auf DOM-Ereignisse (Mausklick, Tastatur)

15 Übungsaufgaben zu Teil 2

Aufgabe 1

Erstelle eine Funktion `summiereBis(n)`, die rekursiv alle Zahlen von n bis 1 aufsummiert.

Aufgabe 2

Schreibe eine Arrow Function `istVokal`, die prüft, ob ein einzelner Buchstabe ein Vokal ist.

Aufgabe 3

Nutze `setTimeout` für eine Countdown-Funktion, die „3", „2", „1", „Los!" mit 1 Sekunde Abstand ausgibt.

Aufgabe 4

Erstelle eine Funktion `wandleUm(text, func)`, die einen übergebenen Text mit einer Callback-Funktion verarbeitet (z. B. rückwärts drehen).

Aufgabe 5

Schreibe eine Funktion `rufeMehrfach(callback, n)`, die eine Callback-Funktion n-mal ausführt.

Aufgabe 6

Erstelle eine Funktion `listeBearbeiten(liste, funktion)`, die ein Array mithilfe einer Callback-Funktion bearbeitet (z. B. verdoppeln).

Aufgabe 7

Schreibe eine rekursive Funktion `zaehle(n)`, die bis zu einer Zahl hochzählt und jeden Wert in der Konsole anzeigt.

Aufgabe 8

Nutze eine Arrow Function mit `map()`, um alle Wörter in einem Array zu Großbuchstaben zu konvertieren.

Aufgabe 9

Simuliere mit `setInterval()` ein einfaches Ladesymbol („...") und stoppe es nach 4 Durchläufen.

Aufgabe 10

Nutze eine anonyme Funktion als Callback in `filter()`, um alle Zahlen unter 10 zu entfernen.

Aufgabe 11

Schreibe eine Funktion `begruesseMehrfach(namen, callback)`, die jeden Namen in einer Liste mit einer übergebenen Callback-Funktion begrüßt.

Aufgabe 12

Nutze `setTimeout`, um eine Liste von Wörtern verzögert nacheinander auszugeben.

Aufgabe 13

Schreibe eine Funktion `operation(a, b, func)`, die zwei Zahlen mit einer beliebigen Rechenoperation kombiniert.

Aufgabe 14

Nutze eine rekursive Funktion `umkehren(text)`, die einen Text Zeichen für Zeichen rückwärts zusammensetzt.

Aufgabe 15

Baue eine kleine Interaktion mit `prompt` und einem benutzerdefinierten Callback, der eine personalisierte Nachricht erzeugt.