4.2 - JSA: Generatoren und Iteratoren

Inhaltsübersicht Abschnitt 4.2

---

1. Einführung zu Iterables und Iteration

In JavaScript gibt es viele Typen von Objekten, die eine Sammlung von Daten darstellen, beispielsweise Arrays, Strings, Maps oder Sets. Oftmals besteht die Notwendigkeit, die Elemente solcher Sammlungen nacheinander zu durchlaufen oder zu verarbeiten.

1.1 Was sind iterable Objekte?

Ein Objekt wird als **iterierbar** (iterable) bezeichnet, wenn es ein Standardverfahren implementiert, um seine Elemente einzeln zugänglich zu machen. Genauer gesagt muss ein iterierbares Objekt eine Methode mit dem Schlüssel `Symbol.iterator` besitzen. Diese Methode ist dafür verantwortlich, ein **Iterator**-Objekt zurückzugeben.

Standardmäßig sind in JavaScript folgende eingebaute Typen iterierbar:

- Arrays (`Array`)
- Strings (`String`)
- Maps (`Map`)
- Sets (`Set`)
- `arguments`-Objekt (in Funktionen)
- NodeList und HTMLCollection (im Browser-DOM)

1.2 Standard-Iterationsmechanismen

JavaScript bietet einige eingebaute Sprachkonstrukte, die mit iterierbaren Objekten arbeiten:

1. **`for...of`-Schleife:** Die `for...of`-Schleife ist die gebräuchlichste Methode, um über die Elemente eines iterierbaren Objekts zu iterieren.

```
const zahlenArray = [10, 20, 30];
for (const zahl of zahlenArray) {
  console.log(zahl); // Gibt 10, dann 20, dann 30 aus
}

const text = "JS";
for (const buchstabe of text) {
  console.log(buchstabe); // Gibt "J", dann "S" aus
}
```

2. **Spread-Operator (`...`):** Der Spread-Operator kann verwendet werden, um die Elemente eines iterierbaren Objekts an Stellen zu expandieren, an denen mehrere Elemente oder Argumente erwartet werden.

```
const buchstaben = ['a', 'b', 'c'];
const neuesArray = ['x', ...buchstaben, 'y']; // neuesArray ist ['x', 'a', 'b', 'c', 'y']
console.log(neuesArray);

function summe(x, y, z) { return x + y + z; }
const werte = [1, 2, 3];
console.log(summe(...werte)); // Äquivalent zu summe(1, 2, 3) -> Ausgabe: 6
```

Diese Mechanismen funktionieren, weil sie intern das Iterator-Protokoll verwenden, das wir uns jetzt genauer ansehen.

---

2. Das Iterator-Protokoll und Iteratoren im Detail

Die Fähigkeit, über Objekte zu iterieren, basiert auf zwei Protokollen: dem **Iterable-Protokoll** und dem **Iterator-Protokoll**.

- **Iterable-Protokoll:** Ein Objekt ist iterierbar, wenn es eine Methode namens `[Symbol.iterator]` implementiert. Diese Methode muss ein Objekt zurückgeben, das das Iterator-Protokoll erfüllt.
- **Iterator-Protokoll:** Ein Objekt ist ein Iterator, wenn es eine Methode namens `next()` implementiert.

2.1 Definition und Zweck eines Iterators (Edube 4.2.2)

Ein **Iterator** ist ein Objekt, das weiß, wie es auf die Elemente einer Sammlung zugreifen und diese nacheinander zurückgeben kann. Man kann es sich als einen Zeiger oder Cursor vorstellen, der sich durch die Sammlung bewegt. Der Iterator verwaltet den aktuellen Stand der Iteration.

2.2 Die `next()`-Methode

Die `next()`-Methode ist das Herzstück des Iterator-Protokolls. Bei jedem Aufruf führt sie zwei Aufgaben aus:

1. Sie bewegt den Iterator zum nächsten Element der Sammlung (falls vorhanden).
2. Sie gibt ein Objekt mit zwei Eigenschaften zurück:
    - `value`: Der Wert des aktuellen Elements in der Iteration. Wenn die Iteration beendet ist und keine weiteren Werte vorhanden sind, ist `value` oft `undefined` (kann aber auch ein spezifischer Rückgabewert sein).
    - `done`: Ein Boolean-Wert. `false`, wenn der Iterator ein weiteres Element produzieren konnte. `true`, wenn die Iteration abgeschlossen ist und keine weiteren Elemente mehr vorhanden sind.

Beispiel: Manuelle Verwendung eines Array-Iterators

```
const farben = ["rot", "grün", "blau"];

// 1. Iterator-Objekt vom iterierbaren Objekt (Array) erhalten:
const farbenIterator = farben[Symbol.iterator]();

// 2. Die next()-Methode des Iterators aufrufen:
let schritt1 = farbenIterator.next();
console.log(schritt1); // Ausgabe: { value: "rot", done: false }

let schritt2 = farbenIterator.next();
console.log(schritt2); // Ausgabe: { value: "grün", done: false }

let schritt3 = farbenIterator.next();
console.log(schritt3); // Ausgabe: { value: "blau", done: false }

let schritt4 = farbenIterator.next();
console.log(schritt4); // Ausgabe: { value: undefined, done: true }
// 'done' ist true, die Iteration ist beendet.

// Jeder weitere Aufruf von next() gibt ebenfalls { value: undefined, done: true } zurück.
```

```
let schritt5 = farbenIterator.next();
console.log(schritt5); // Ausgabe: { value: undefined, done: true }
```

Die `for...of`-Schleife macht genau das intern: Sie ruft `Symbol.iterator` auf, um den Iterator zu bekommen, und ruft dann wiederholt `next()` auf, bis `done` `true` ist.

2.3 Manuelle Erstellung eines iterierbaren Objekts (Schritt-für-Schritt)

Obwohl viele eingebaute Objekte bereits iterierbar sind, können wir auch eigene, benutzerdefinierte Objekte iterierbar machen. Dies ist besonders nützlich für eigene Datenstrukturen. (Vergleich Edube 4.2.2, `almostIterable`)

Szenario: Wir haben ein Objekt, das eine einfache Liste von Aufgaben verwaltet, und wir möchten es mit `for...of` durchlaufen können.

Schritt 1: Das Basisobjekt (noch nicht iterierbar)

```
const meineAufgabenliste = {
  titel: "Wichtige Aufgaben",
  aufgaben: ["Einkaufen gehen", "Rechnungen bezahlen", "Code schreiben"],
  zeigeTitel: function() {
    console.log(`Listentitel: ${this.titel}`);
  }
};

// Versuch, darüber zu iterieren (führt zu einem Fehler):
try {
  for (const aufgabe of meineAufgabenliste) {
    console.log(aufgabe);
  }
} catch (e) {
  console.error("Fehler beim Iterieren:", e.message);
  // Ausgabe: Fehler beim Iterieren: meineAufgabenliste is not iterable
}
```

Der Fehler tritt auf, weil `meineAufgabenliste` keine `[Symbol.iterator]`-Methode besitzt.

Schritt 2: Implementierung der `[Symbol.iterator]`-Methode Um das Objekt iterierbar zu machen, fügen wir die Methode `[Symbol.iterator]` hinzu. Diese Methode muss ein Iterator-Objekt zurückgeben, das wiederum eine `next()`-Methode hat.

```
const meineAufgabenlisteIterierbar = {
  titel: "Wichtige Aufgaben",
  aufgaben: ["Einkaufen gehen", "Rechnungen bezahlen", "Code schreiben"],
  zeigeTitel: function() {
    console.log(`Listentitel: ${this.titel}`);
  },

  // Die [Symbol.iterator]-Methode
  [Symbol.iterator]: function() {
    console.log("[Symbol.iterator] wurde aufgerufen für:", this.titel);
    let aktuellerIndex = 0;       // Zustand des Iterators: aktueller Index
    const aufgabenArray = this.aufgaben; // Referenz auf das interne Array

    // Das Iterator-Objekt mit der next()-Methode wird zurückgegeben
    return {
      next: function() {
        console.log("Iterator: next() wurde aufgerufen, aktuellerIndex:", aktuellerIndex);
        if (aktuellerIndex < aufgabenArray.length) {
          // Es gibt noch Elemente
          return { value: aufgabenArray[aktuellerIndex++], done: false };
        } else {
          // Keine Elemente mehr, Iteration beendet
          console.log("Iterator: Iteration beendet.");
```

```
          return { value: undefined, done: true };
        }
      }
    };
  }
};

// Jetzt können wir mit for...of darüber iterieren:
console.log("Iteriere über meineAufgabenlisteIterierbar:");
for (const aufgabe of meineAufgabenlisteIterierbar) {
  console.log(`- ${aufgabe}`);
}
// Erwartete Ausgabe (mit Logs):
// Iteriere über meineAufgabenlisteIterierbar:
// [Symbol.iterator] wurde aufgerufen für: Wichtige Aufgaben
// Iterator: next() wurde aufgerufen, aktuellerIndex: 0
// - Einkaufen gehen
// Iterator: next() wurde aufgerufen, aktuellerIndex: 1
// - Rechnungen bezahlen
// Iterator: next() wurde aufgerufen, aktuellerIndex: 2
// - Code schreiben
// Iterator: next() wurde aufgerufen, aktuellerIndex: 3
// Iterator: Iteration beendet.

// Der Spread-Operator funktioniert jetzt auch:
const aufgabenAlsArray = [...meineAufgabenlisteIterierbar];
console.log("Aufgaben als Array (Spread):", aufgabenAlsArray);
// Erwartete Ausgabe (mit Logs für einen neuen Iterator):
// [Symbol.iterator] wurde aufgerufen für: Wichtige Aufgaben
// Iterator: next() wurde aufgerufen, aktuellerIndex: 0
// Iterator: next() wurde aufgerufen, aktuellerIndex: 1
// Iterator: next() wurde aufgerufen, aktuellerIndex: 2
// Iterator: next() wurde aufgerufen, aktuellerIndex: 3
// Iterator: Iteration beendet.
// Aufgaben als Array (Spread): [ 'Einkaufen gehen', 'Rechnungen bezahlen', 'Code schreiben' ]
```

Erläuterung:

- `[Symbol.iterator]` ist ein sogenanntes "Well-known Symbol". JavaScript sucht nach einer Methode mit genau diesem symbolischen Namen, um ein Objekt als iterierbar zu erkennen.
- Die `[Symbol.iterator]`-Methode wird einmal zu Beginn der Iteration (z.B. durch `for...of` oder Spread) aufgerufen.
- Sie gibt ein neues Iterator-Objekt zurück. Es ist wichtig, dass bei jedem Aufruf von `[Symbol.iterator]` ein *neuer* Iterator erstellt wird, damit das Objekt mehrfach iteriert werden kann.
- Das Iterator-Objekt enthält die `next()`-Methode. Der Zustand der Iteration (hier `aktuellerIndex`) wird innerhalb des Scopes der `[Symbol.iterator]`-Methode gespeichert und ist durch die Closure für die `next()`-Funktion zugänglich.

Beispiel: Ein iterierbares Objekt für einen Zahlenbereich Dieses Objekt repräsentiert nicht eine fixe Datensammlung, sondern generiert seine Werte dynamisch.

```
function erstelleZahlenBereich(start, ende, schritt = 1) {
  if (schritt === 0) throw new Error("Schrittweite darf nicht null sein.");

  return {
    [Symbol.iterator]: function() {
      let aktuellerWert = start;
      let ersterDurchlauf = true; // Um sicherzustellen, dass 'start' auch bei schritt < 0
korrekt behandelt wird

      return {
        next: function() {
          if (ersterDurchlauf) {
            ersterDurchlauf = false;
            // Prüfen, ob der Startwert bereits außerhalb des Bereichs liegt
            if ((schritt > 0 && aktuellerWert > ende) || (schritt < 0 && aktuellerWert < ende))
```

```
{
              return { value: undefined, done: true };
            }
            return { value: aktuellerWert, done: false };
          }

          aktuellerWert += schritt;

          if ((schritt > 0 && aktuellerWert <= ende) || (schritt < 0 && aktuellerWert >= ende))
{
            return { value: aktuellerWert, done: false };
          } else {
            return { value: undefined, done: true };
          }
        }
      };
    }
  };
}

console.log("Zahlenbereich von 2 bis 8 (Schritt 2):");
for (const zahl of erstelleZahlenBereich(2, 8, 2)) {
  console.log(zahl); // Ausgabe: 2, 4, 6, 8
}

console.log("Zahlenbereich von 5 bis 1 (Schritt -1):");
const bereichAbsteigend = erstelleZahlenBereich(5, 1, -1);
for (const zahl of bereichAbsteigend) {
  console.log(zahl); // Ausgabe: 5, 4, 3, 2, 1
}

console.log("Leerer Bereich:");
for (const zahl of erstelleZahlenBereich(10, 1, 1)) { // Start > Ende mit positivem Schritt
  console.log(zahl); // Keine Ausgabe
}
```

Die manuelle Implementierung von Iteratoren kann, wie man sieht, recht detailliert werden, besonders das korrekte Zustandsmanagement. Hier kommen Generatoren ins Spiel.

Referenzen:

- MDN Web Docs: Iterationsprotokolle
- MDN Web Docs: Symbol.iterator

---

3. Generatoren (`function*` und `yield`)

Generatoren, eingeführt in ES6, bieten eine wesentlich elegantere und einfachere Syntax zur Erstellung von Iteratoren. (Edube 4.2.3)

3.1 Problemstellung mit manuellen Iteratoren

Wie im vorherigen Abschnitt gesehen, erfordert die manuelle Implementierung eines Iterators die explizite Verwaltung eines internen Zustands (z.B. `aktuellerIndex`, `aktuellerWert`) und die sorgfältige Implementierung der `next()`-Methode, um `{ value: ..., done: ... }`-Objekte korrekt zurückzugeben. Bei komplexeren Iterationslogiken kann dies schnell unübersichtlich und fehleranfällig werden.

3.2 Was ist eine Generatorfunktion?

Eine **Generatorfunktion** ist eine spezielle Art von Funktion, die ihre Ausführung an bestimmten Stellen **anhalten** (pausieren) und später an derselben Stelle **wieder aufnehmen** kann. Ihr Zustand (inklusive lokaler Variablen) bleibt über diese Pausen hinweg erhalten.

Syntax: Generatorfunktionen werden durch ein Sternchen `*` nach dem `function`-Keyword (oder vor dem Funktionsnamen bei bestimmten Schreibweisen) gekennzeichnet:

```
// Deklaration einer Generatorfunktion
function* meineGeneratorFunktion() {
  // ... Code mit yield
}

// Als Funktionsausdruck
const andererGenerator = function*() {
  // ... Code mit yield
};

// Als Methode in einem Objektliteral
const meinObj = {
  *generatorMethode() {
    // ... Code mit yield
  }
};

// Als Methode in einer Klasse
class MeineKlasse {
  *klassenGeneratorMethode() {
    // ... Code mit yield
  }
}
```

3.3 Der `yield`-Operator

Innerhalb einer Generatorfunktion wird der `yield`-Operator verwendet, um einen Wert zu produzieren und die Ausführung der Funktion anzuhalten.

- `yield ausdruck;`: Die Generatorfunktion pausiert an dieser Stelle. Der Wert von `ausdruck` wird als `value`-Eigenschaft des vom Iterator zurückgegebenen Objekts gesendet. Die `done`-Eigenschaft ist `false`.
- Wenn die `next()`-Methode des Iterators erneut aufgerufen wird, setzt die Generatorfunktion ihre Ausführung direkt *nach* dem letzten `yield`-Statement fort.

3.4 Wie Generatorfunktionen Iteratoren erzeugen

Der entscheidende Punkt ist: Wenn eine Generatorfunktion aufgerufen wird, wird ihr Code **nicht sofort ausgeführt**. Stattdessen gibt der Aufruf ein spezielles **Generator-Objekt** zurück. Dieses Generator-Objekt ist selbst ein **Iterator**. Es implementiert das Iterator-Protokoll, d.h., es hat eine `next()`-Methode.

Jeder Aufruf von `next()` auf diesem Generator-Objekt führt den Code der Generatorfunktion bis zum nächsten `yield`-Statement aus (oder bis zum Ende der Funktion, falls kein `yield` mehr folgt oder ein `return` erreicht wird).

Beispiel 1: Einfacher Generator mit mehreren `yield`-Anweisungen (Vergleich Edube 4.2.3, `testGenerator`)

```
function* einfacheSequenzGenerator() {
  console.log("Generator: Start oder Fortsetzung – vor yield 'alpha'");
  yield 'alpha';
  console.log("Generator: Fortsetzung – vor yield 'beta'");
  yield 'beta';
  console.log("Generator: Fortsetzung – vor yield 'gamma'");
  yield 'gamma';
  console.log("Generator: Ausführung beendet.");
  return 'Fertig!'; // Der return-Wert wird value, wenn done: true ist
}

// 1. Generator-Objekt (Iterator) erstellen:
const sequenzIterator = einfacheSequenzGenerator();
console.log("Iterator erstellt.");

// 2. Werte mit next() abrufen:
console.log("Aufruf 1: sequenzIterator.next()");
let ergebnis1 = sequenzIterator.next();
```

```
console.log(ergebnis1); // { value: 'alpha', done: false } (und Log von innerhalb des
Generators)

console.log("Aufruf 2: sequenzIterator.next()");
let ergebnis2 = sequenzIterator.next();
console.log(ergebnis2); // { value: 'beta', done: false } (und Log)

console.log("Aufruf 3: sequenzIterator.next()");
let ergebnis3 = sequenzIterator.next();
console.log(ergebnis3); // { value: 'gamma', done: false } (und Log)

console.log("Aufruf 4: sequenzIterator.next()");
let ergebnis4 = sequenzIterator.next();
console.log(ergebnis4); // { value: 'Fertig!', done: true } (und Log)

console.log("Aufruf 5: sequenzIterator.next()");
let ergebnis5 = sequenzIterator.next();
console.log(ergebnis5); // { value: undefined, done: true } (nachdem return erreicht wurde)
```

Erläuterung:

- Beim ersten `sequenzIterator.next()`: Die Funktion startet, gibt "Generator: Start..." aus, trifft auf `yield 'alpha'`, pausiert und gibt `{ value: 'alpha', done: false }` zurück.
- Beim zweiten `sequenzIterator.next()`: Die Funktion setzt *nach* `yield 'alpha'` fort, gibt "Generator: Fortsetzung - vor yield 'beta'" aus, trifft auf `yield 'beta'`, pausiert und gibt `{ value: 'beta', done: false }` zurück.
- Dies setzt sich fort, bis die Funktion endet. Wenn ein `return`-Statement vorhanden ist, wird dessen Wert als `value` im letzten Schritt (wo `done: true` ist) verwendet. Ohne explizites `return` wäre `value` dann `undefined`.

3.5 Verwendung von Generatoren zur Implementierung von `Symbol.iterator`

Generatoren machen die Implementierung der `[Symbol.iterator]`-Methode drastisch einfacher. (Vergleich Edube 4.2.3, `iterable` mit Generator)

```
const meineAufgabenlisteMitGenerator = {
  titel: "Optimierte Aufgabenliste",
  aufgaben: ["Dokumentation lesen", "Tests schreiben", "Refactoring"],

  [Symbol.iterator]: function*() { // Die Methode ist jetzt eine Generatorfunktion
    console.log("Generator für Aufgabenliste gestartet.");
    for (let i = 0; i < this.aufgaben.length; i++) {
      console.log(`Generator: yield Aufgabe Nr. ${i + 1}`);
      yield this.aufgaben[i];
    }
    // Alternativ und oft kürzer für Arrays oder andere Iterables:
    // yield* this.aufgaben; // Siehe Abschnitt zu yield*
    console.log("Generator für Aufgabenliste beendet.");
  }
};

console.log("Iteriere über Aufgabenliste mit Generator:");
for (const aufgabe of meineAufgabenlisteMitGenerator) {
  console.log(`-> ${aufgabe}`);
}
// Ausgabe:
// Iteriere über Aufgabenliste mit Generator:
// Generator für Aufgabenliste gestartet.
// Generator: yield Aufgabe Nr. 1
// -> Dokumentation lesen
// Generator: yield Aufgabe Nr. 2
// -> Tests schreiben
// Generator: yield Aufgabe Nr. 3
// -> Refactoring
// Generator für Aufgabenliste beendet.
```

Der gesamte Code für das Zustandsmanagement (`aktuellerIndex`) und die Erstellung der `{value, done}`-Objekte entfällt. Der Generator kümmert sich darum automatisch.

3.6 Generatoren für dynamische oder "unendliche" Sequenzen

Generatoren sind ideal, um Sequenzen von Werten zu erzeugen, die nicht notwendigerweise in einer festen Datenstruktur gespeichert sind, oder sogar potenziell unendliche Sequenzen darzustellen. (Vergleich Edube 4.2.3, `factorialGenerator` und `fibonacci`)

Beispiel: Quadratzahlen-Generator

```
function* quadratGenerator(limit) {
  for (let i = 1; i <= limit; i++) {
    yield i * i;
  }
}

const quadrateBis5 = quadratGenerator(5);
console.log("Quadratzahlen bis 5:");
for (const q of quadrateBis5) {
  console.log(q); // 1, 4, 9, 16, 25
}

// Da der Iterator verbraucht ist, gibt eine erneute Iteration keine Werte:
console.log("Erneuter Versuch, über quadrateBis5 zu iterieren:");
for (const q of quadrateBis5) {
  console.log(q); // Keine Ausgabe
}
// Um erneut zu iterieren, muss ein neuer Iterator erstellt werden:
const neuerQuadratIterator = quadratGenerator(3);
console.log("Quadratzahlen bis 3 (neuer Iterator):", ...neuerQuadratIterator); // 1, 4, 9
```

Beispiel: Fibonacci-Sequenz (potenziell unendlich)

```
function* fibonacciGenerator(maxWerte = Infinity) {
  let a = 0;
  let b = 1;
  let anzahlProduziert = 0;

  while (anzahlProduziert < maxWerte) {
    yield a;
    anzahlProduziert++;
    if (anzahlProduziert >= maxWerte) break;

    [a, b] = [b, a + b]; // Nächste Fibonacci-Zahl berechnen
  }
}

const fibGen = fibonacciGenerator(10); // Die ersten 10 Fibonacci-Zahlen (0-9)
console.log("Die ersten 10 Fibonacci-Zahlen:");
for (const fibZahl of fibGen) {
  console.log(fibZahl); // 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
}

// Ein "unendlicher" Fibonacci-Generator (Vorsicht bei der Verwendung!)
const unendlicherFibGen = fibonacciGenerator();
// console.log(unendlicherFibGen.next().value); // 0
// console.log(unendlicherFibGen.next().value); // 1
// ... und so weiter. Eine for...of-Schleife würde hier ohne Abbruchbedingung endlos laufen.
```

3.7 Der `yield*`-Operator: Delegation an einen anderen Iterator/Generator

Der `yield*`-Operator (yield-star) wird verwendet, um die Iteration an ein anderes iterierbares Objekt oder einen anderen Generator zu **delegieren**. Er iteriert über den Operanden und `yield`et jeden Wert, den dieser Operand produziert, einzeln weiter. (Vergleich Edube 4.2.3, `test` mit `yield* cities()`)

Syntax: `yield* <iterableOderGenerator>;`

Beispiel 1: Delegation an ein Array und einen anderen Generator

```
function* praefixGenerator() {
  yield "A";
  yield "B";
}

function* suffixGenerator() {
  yield "Y";
  yield "Z";
}

function* kombinierterGenerator() {
  yield "Start";
  yield* praefixGenerator(); // Delegiert an praefixGenerator
  yield* ["Mittel1", "Mittel2"]; // Delegiert an ein Array
  yield* suffixGenerator();   // Delegiert an suffixGenerator
  yield "Ende";
}

const kombiIt = kombinierterGenerator();
console.log("Kombinierter Generator:");
for (const wert of kombiIt) {
  console.log(wert);
}
// Ausgabe:
// Start
// A
// B
// Mittel1
// Mittel2
// Y
// Z
// Ende
```

Erläuterung:

- `yield* praefixGenerator()`: Der `kombinierterGenerator` pausiert, und der `praefixGenerator` wird ausgeführt. Jeder Wert, den `praefixGenerator` `yield`et (`"A"`, `"B"`), wird direkt vom `kombinierterGenerator` weitergegeben, als ob er selbst diese `yield`-Anweisungen hätte.
- `yield* ["Mittel1", "Mittel2"]`: Iteriert über das Array und `yield`et `"Mittel1"` und dann `"Mittel2"`.
- Dies ist nützlich, um Generatorfunktionen modular aufzubauen oder Code aus verschiedenen iterierbaren Quellen zusammenzuführen.

Referenzen:

- MDN Web Docs: Generator
- MDN Web Docs: function*
- MDN Web Docs: yield
- MDN Web Docs: yield*

---

4. Übungsaufgaben zu Iteratoren und Generatoren

Aufgabe 4.1: Benutzerdefinierter Iterator für ein Wort Erstellen Sie ein Objekt `WortSequenz`, das ein einzelnes Wort als String im Konstruktor annimmt. Machen Sie dieses Objekt iterierbar, sodass es bei jeder Iteration den nächsten Buchstaben des Wortes in Großbuchstaben zurückgibt. Implementieren Sie `[Symbol.iterator]` manuell (ohne Generator). Beispiel: `const meinWort = new`

`WortSequenz("Hallo"); for (const buchstabe of meinWort) { console.log(buchstabe); } // Sollte H, A, L, L, O ausgeben`

Aufgabe 4.2: Umgekehrter Array-Iterator Erstellen Sie eine Funktion `erstelleUmgekehrtenIterator(arr)`, die ein Array entgegennimmt und einen Iterator zurückgibt, der die Elemente des Arrays in umgekehrter Reihenfolge liefert. Implementieren Sie dies manuell.

Aufgabe 4.3: Generator für Potenzen Schreiben Sie eine Generatorfunktion `potenzGenerator(basis, maxExponent)`. Diese soll nacheinander die Potenzen der `basis` von `basis^0` bis `basis^maxExponent` liefern. Beispiel: `for (const p of potenzGenerator(2, 4)) { console.log(p); } // 1, 2, 4, 8, 16`

Aufgabe 4.4: Objekt-Eigenschaften-Generator Schreiben Sie eine Generatorfunktion `objektEigenschaftenGenerator(obj)`, die über die Schlüssel-Wert-Paare eines gegebenen Objekts iteriert und bei jedem Schritt ein Array `[schluessel, wert]` zurückgibt. (Hinweis: `Object.keys()`, `Object.values()`, oder `Object.entries()` könnten hier nützlich sein, aber versuchen Sie, es primär mit einer Schleife über Schlüssel zu lösen und `yield` zu verwenden.)

Aufgabe 4.5: Generator mit `yield*` für verschachtelte Arrays Gegeben sei ein verschachteltes Array, z.B. `const daten = [1, [2, 3], 4, [5, [6]], 7];`. Schreiben Sie eine Generatorfunktion `flacherGenerator(arr)`, die `yield*` verwendet, um rekursiv alle Zahlen aus diesem verschachtelten Array einzeln zu liefern (also eine "flache" Sequenz der Zahlen). Beispiel: `[...flacherGenerator(daten)]` sollte `[1, 2, 3, 4, 5, 6, 7]` ergeben.

Aufgabe 4.6: Protokollierender Iterator-Wrapper (manuell) Schreiben Sie eine Funktion `logIterator(iterable)`, die ein iterierbares Objekt entgegennimmt. Diese Funktion soll ein neues iterierbares Objekt zurückgeben. Wenn über dieses neue Objekt iteriert wird, soll jede `next()`-Operation protokolliert werden (z.B. mit `console.log("next() aufgerufen, Ergebnis:", result)`), bevor das Ergebnis `{value, done}` zurückgegeben wird. Die eigentliche Iterationslogik soll vom ursprünglichen `iterable` stammen.

Aufgabe 4.7: Palindrom-Tester mit Iterator (Analyse) Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen dasselbe ergibt (z.B. "anna"). Wie könnte man das Iterator-Protokoll (oder Generatoren) konzeptionell nutzen, um zu prüfen, ob ein String ein Palindrom ist, indem man ihn von vorne und von hinten gleichzeitig "iteriert" und die Zeichen vergleicht? Beschreiben Sie den Ansatz; eine vollständige Implementierung ist optional, aber Bonus.

---

5. Micro-Projekt: Utility-Funktionen - Phase 3 (Iteratoren/Generatoren)

Wir erweitern unsere `textUtils.js` Bibliothek.

Aufgabe für das Micro-Projekt (Phase 3):

Entwickeln Sie eine Generatorfunktion `generiereZeilen(mehrzeiligerText)` in `textUtils.js`. Diese Funktion soll einen String als Eingabe nehmen, der mehrere Zeilen enthalten kann (getrennt durch `\n`). Der Generator soll bei jeder Iteration (also bei jedem `yield`) eine einzelne Zeile des Textes zurückgeben. Leere Zeilen sollen ebenfalls als (leere) Strings zurückgegeben werden.

Beispielaufrufe und erwartete Ergebnisse:

```
// In textUtils.js:
// export function* generiereZeilen(mehrzeiligerText) {
//   // Ihre Implementierung hier
// }

// In einer anderen Datei zum Testen:
// import { generiereZeilen } from './textUtils.js'; // (Modul-Syntax)

const textBlock = `Zeile eins.
Zeile zwei ist etwas länger.

Zeile vier nach einer Leerzeile.`;

const zeilenGenerator = generiereZeilen(textBlock);

for (const zeile of zeilenGenerator) {
  console.log(`"${zeile}"`);
}
// Erwartete Ausgabe:
```

```
// "Zeile eins."
// "Zeile zwei ist etwas länger."
// ""
// "Zeile vier nach einer Leerzeile."

// Alternativ mit Spread (erzeugt ein Array aller Zeilen):
// const alleZeilen = [...generiereZeilen(textBlock)];
// console.log(alleZeilen);
// Erwartet: [ 'Zeile eins.', 'Zeile zwei ist etwas länger.', '', 'Zeile vier nach einer
Leerzeile.' ]
```

Hinweis: Die String-Methode `split('\n')` könnte hierfür eine gute Basis sein, um die Zeilen zu erhalten, über die der Generator dann iteriert.
