JS Moderne Syntax

Lernziele

- **JS Moderne Syntax**
- Verstehen von JS als eine sich entwickelnde Programmiersprache
- Destrukturierende Zuweisung
- Rest- und Spread-Syntax
- Optionale Verkettung
- Nullish Coalescing

Eine sich entwickelnde Sprache: JavaScript

JavaScript ist noch nicht abgeschlossen – es wird weiterhin entwickelt. Die tc39-Gruppe trifft sich alle zwei Monate, um Vorschläge bezüglich der ECMAScript-Spezifikationen zu diskutieren, auf denen JavaScript basiert. Hier findest du eine Liste aktiver Vorschläge, die noch geprüft werden. Einige der abgeschlossenen Vorschläge kannst du hier einsehen.

Destrukturierende Zuweisung

Die destrukturierende Zuweisungssyntax ist ein JavaScript-Ausdruck, der es ermöglicht, Werte aus Arrays oder Eigenschaften aus Objekten in einzelne Variablen zu entpacken. Die Destrukturierung mutiert weder das ursprüngliche Array noch das Objekt.

Destrukturierung von Arrays

```
const griechischeBuchstaben = ["alpha", "beta", "gamma", "delta"];
```

Wenn wir die Werte von `griechischeBuchstaben` als einzelne Variablen deklarieren wollen, kann der Zugriff darauf ziemlich mühsam werden.

Ohne Destrukturierung

```
const ersterBuchstabe = griechischeBuchstaben[0];
const zweiterBuchstabe = griechischeBuchstaben[1];
const dritterBuchstabe = griechischeBuchstaben[2];
const vierterBuchstabe = griechischeBuchstaben[3];
// ersterBuchstabe → 'alpha'
// zweiterBuchstabe → 'beta'
// usw.
```

Mit Destrukturierung

```
const [ersterBuchstabe, zweiterBuchstabe, dritterBuchstabe,
vierterBuchstabe] =
  griechischeBuchstaben;
// ersterBuchstabe → 'alpha'
// zweiterBuchstabe → 'beta'
// usw.
```

Das Ergebnis der beiden Codebeispiele ist dasselbe, aber die Destrukturierung des Arrays sorgt für lesbareren und prägnanteren Code.

Falls du den zweiten und vierten Wert in unserem ursprünglichen Array `griechischeBuchstaben` nicht benötigst oder möchtest, gibt es die Möglichkeit, Werte während des Destrukturierungsprozesses zu überspringen.

```
const [ersterBuchstabe, , dritterBuchstabe] = griechischeBuchstaben;
// ersterBuchstabe → 'alpha'
// dritterBuchstabe → 'gamma'
```

Jetzt enthält `dritterBuchstabe` den *dritten* Wert von `griechischeBuchstaben`. Das zusätzliche Komma überspringt den Wert an der jeweiligen Position in `griechischeBuchstaben`. Der vierte Wert wird ignoriert, da die Destrukturierungszuweisung endet.

Das Überspringen von Werten mit zusätzlichen Kommas kann schnell unleserlich werden. Stell dir vor, du möchtest fünf Werte überspringen: `const [a, , , , , , g] = x;`. Allgemeiner Rat ist, mit diesem Feature sehr vorsichtig umzugehen.

Destrukturierung von Objekten

```
const coachObjekt = {
  name: "Sam",
  stimmung: "großartig",
  fähigkeiten: "erstaunlich",
  punktzahl: 9999,
};
```

Du kannst die destrukturierende Zuweisung wie folgt verwenden:

```
const { name, stimmung, fähigkeiten, punktzahl } = coachObjekt;
// name → 'Sam'
// stimmung → 'großartig'
// usw.
```

Jetzt hast du `name`, `stimmung`, `fähigkeiten` und `punktzahl` als einzelne Variablen verfügbar.

Im Gegensatz zur Array-Destrukturierung sind die Variablennamen hier nicht willkürlich und hängen nicht von der Reihenfolge ab. Die Namen der Variablen müssen hier den Schlüsseln der Objekteigenschaften entsprechen.

Du kannst die Variable während der Destrukturierung auch umbenennen:

```
const { name: vorname } = coachObjekt;
// vorname → 'Sam'
// name → undefined
```

Du kannst auch einen Standardwert für eine Eigenschaft festlegen, falls diese nicht existiert:

```
const { istAdmin = true } = coachObjekt;
// istAdmin → true
```

Rest- und Spread-Syntax

Die Rest- und Spread-Syntax sieht täuschend ähnlich aus. Beide werden durch drei Punkte identifiziert: `...`. Sie erfüllen jedoch je nach Kontext, in dem `...` verwendet wird, unterschiedliche Aufgaben.

Rest-Syntax (`...`)

Die Rest-Syntax erlaubt es dir zu sagen: "Pack den Rest in diese Variable" beim Verwenden der destrukturierenden Zuweisung oder beim Deklarieren von Funktionsparametern.

Du kannst sie mit Arrays verwenden:

```
const griechischeBuchstaben = ["alpha", "beta", "gamma", "delta"];
const [ersterBuchstabe, ...alleAnderenBuchstaben] = griechischeBuchstaben;
// ersterBuchstabe → "alpha"
// alleAnderenBuchstaben → ["beta", "gamma", "delta"]
```

Oder mit Objekten:

```
const coachObjekt = {
  name: "Sam",
  stimmung: "großartig",
  fähigkeiten: "erstaunlich",
  punktzahl: 9999,
};

const { name, punktzahl, ...derRestDesCoachObjekts } = coachObjekt;
// name → "Sam"
// punktzahl → 9999
```

```
// derRestDesCoachObjekts → { stimmung: 'großartig', fähigkeiten:
'erstaunlich' }
```

Und sogar mit Funktionsparametern:

```
function buchstabenLoggen(ersterBuchstabe, ...mehrBuchstaben) {
  console.log("der erste Buchstabe ist", ersterBuchstabe);
  console.log("noch mehr Buchstaben", mehrBuchstaben);
}

buchstabenLoggen("alpha", "beta", "gamma", "delta");
// logs:
// der erste Buchstabe ist alpha
// noch mehr Buchstaben (3) ['beta', 'gamma', 'delta']
```

Spread-Syntax (`...`)

Die Spread-Syntax erlaubt es dir zu sagen: "Verteile alles in dieser Variablen hier" beim Deklarieren von Array- oder Objekt-Literalen oder beim Aufrufen von Funktionen.

Es funktioniert so bei Array-Literal-Deklarationen:

```
const griechischeBuchstaben = ["alpha", "beta", "gamma", "delta"];
const mehrGriechischeBuchstaben = [...griechischeBuchstaben, "epsilon",
"zeta"];
// mehrGriechischeBuchstaben → ['alpha', 'beta', 'gamma', 'delta',
'epsilon', 'zeta']
```

Du kannst zwei (oder mehr) Arrays in ein anderes Array spreaden...

```
const roteFarben = ["karminrot", "rosa", "lila"];
const blaueFarben = ["marineblau", "türkis", "himmelblau"];
const gemischteFarben = [...roteFarben, ...blaueFarben];
// gemischteFarben → ['karminrot', 'rosa', 'lila', 'marineblau', 'türkis',
'himmelblau']
```

... und es ist wichtig, wo du jeweils ein Array in ein anderes spreadest:

```
const katzen = ["katze", "katze", "katze"];
const hunde = ["hund", "hund", "hund"];

const katzenUndHunde = [...katzen, ...hunde];
// katzenUndHunde → ['katze', 'katze', 'katze', 'hund', 'hund', 'hund']

const hundeUndKatzen = [...hunde, ...katzen];
```

```
// hundeUndKatzen → ['hund', 'hund', 'hund', 'katze', 'katze', 'katze']

const katzenZwischenVögeln = ["vogel", ...katzen, "vogel"];
// katzenZwischenVögeln → ['vogel', 'katze', 'katze', 'katze', 'vogel']
```

So funktioniert die Spread-Syntax bei Objekterklärungen:

```
const kreis = { radius: 5, form: "kreis" };

const grünerKreis = { ...kreis, farbe: "grün" };
// grünerKreis → { radius: 5, form: 'kreis', farbe: 'grün' }
```

Beachte, dass die Reihenfolge der Spread-Operationen wichtig ist, da du Eigenschaften überschreiben kannst:

```
const kreis = { radius: 5, form: "kreis" };

const großerKreis = { ...kreis, radius: 20 };
// großerKreis → { radius: 20, form: 'kreis' }

const keinGroßerKreis = { radius: 20, ...kreis };
// keinGroßerKreis → { radius: 5, form: 'kreis' }
```

Die Spread-Syntax ist sehr hilfreich, um eine flache Kopie von Arrays und Objekten zu erstellen:

```
const buch = { titel: "Ulysses", autor: "James Joyce" };
const flacheKopieDesBuchs = { ...buch };
flacheKopieDesBuchs.jahr = "1920";

// buch → { titel: 'Ulysses', autor: 'James Joyce' }
// flacheKopieDesBuchs → { titel: 'Ulysses', autor: 'James Joyce', jahr:
'1920' }
```

```
const griechischeBuchstaben = [
  "alpha",
  "beta",
  "gamma",
  "delta",
  "epsilon",
  "zeta",
];
const sortierteGriechischeBuchstaben = [...griechischeBuchstaben].sort((a,
b) =>
  a.localeCompare(b)
);
```

```
// dies ist eine Alternative zu griechischeBuchstaben.slice(), die auch
eine flache Kopie erstellt

// griechischeBuchstaben → ['alpha', 'beta', 'gamma', 'delta', 'epsilon',
'zeta']
// sortierteGriechischeBuchstaben → ['alpha', 'beta', 'delta', 'epsilon',
'gamma', 'zeta']
```

Du kannst die Spread-Syntax auch beim Aufrufen von Funktionen verwenden:

```
const zahlen = [4534, 3411, 2455, 4952];
const kleinsteZahl = Math.min(...zahlen);
// kleinsteZahl → 2455
```

Optionale Verkettung

Der Operator für optionale Verkettung `?.` funktioniert ähnlich wie der Verkettungsoperator `.`, mit dem Unterschied, dass anstatt einen Fehler zu werfen, wenn ein Verweis *nullish* (null oder undefined) ist, der Ausdruck mit einem Wert von `undefined` "kurzgeschlossen" wird. Dies ist nützlich, wenn du dir nicht sicher bist, ob eine Eigenschaft, Methode oder ein Index existiert oder ob etwas aufrufbar ist.

```
function füttereKatzen(katzen) {
  return katzen.forEach((katze) => console.log(katze, "sagt miau"));
}

füttereKatzen(["Garfield", "Tom", "Grumpy Cat"]);
// logs:
// Garfield sagt miau
// Tom sagt miau
// Grumpy Cat sagt miau

füttereKatzen();
// wirft:
// Uncaught TypeError: Cannot read properties of undefined (reading
'forEach')
```

```
function füttereKatzen(katzen) {
  return katzen?.forEach((katze) => console.log(katze, "sagt miau"));
}

füttereKatzen();
// logs nichts, wirft aber auch keinen Fehler
```

Der Operator `?.` kurzschließt den Ausdruck und bewertet die Rückgabeanweisung zu `undefined`. Auf diese Weise wird der Fehler vermieden, ohne dass explizit if-Anweisungen oder ternäre Operatoren zur

Überprüfung der Werte verwendet werden müssen.

Optionale Verkettung ist nützlich, wenn du auf eine Eigenschaft in einem Objekt mit einer tief verschachtelten Struktur abzielst.

```
const person = {
  name: "Sam",
  fähigkeiten: [
    {
      name: "HTML",
      level: 9999,
      kategorie: {
        name: "coding",
      },
    },
    {
      name: "Agile",
      level: 1337,
      kategorie: {
        name: "projects",
      },
    },
  ],
};

console.log(person.fähigkeiten[1].kategorie.name);
// logs: projects

console.log(person.fähigkeiten[2].level);
// wirft: Uncaught TypeError: Cannot read properties of undefined (reading
'level')

console.log(person.fähigkeiten?.[2]?.level);
// logs: undefined

console.log(person.fähigkeiten[0].partner.name);
// wirft: Uncaught TypeError: Cannot read properties of undefined (reading
'name')

console.log(person.fähigkeiten[0].partner?.name);
// logs: undefined
```

Nullish Coalescing

Der Nullish-Coalescing-Operator `??` ist ein logischer Operator, der seinen rechten Operand zurückgibt, wenn sein linker Operand null oder undefined ist, und ansonsten seinen linken Operand zurückgibt.

```
const schokolade = true;
```

```
function schokoladenCheck() {
  return schokolade ?? "Keine Schokolade :(";
}

const ergebnis = schokoladenCheck();
```

In diesem Beispiel wird unsere Funktion `schokoladenCheck` den Wert von `schokolade` zurückgeben (in diesem Fall: `true`), da der Wert von `schokolade` `weder` `null` `noch` `undefined` ist.

```
const schokolade = undefined;

function schokoladenCheck() {
  return schokolade ?? "Keine Schokolade :(";
}

const ergebnis = schokoladenCheck();
```

In diesem Beispiel würde unsere Funktion `'Keine Schokolade :('` zurückgeben, da der Wert von `schokolade` `undefined` ist. Dasselbe würde passieren, wenn wir die Variable `schokolade` mit dem Wert `null` deklarieren.

Geschrieben als `if/else`-Anweisung, würde der Code so aussehen:

```
const schokolade = true;

function schokoladenCheck() {
  if (schokolade === null || schokolade === undefined) {
    return "Keine Schokolade :(";
  }

  return true;
}

const ergebnis = schokoladenCheck();
```

Ressourcen

- TC39 - Specifying JavaScript
- async function (MDN)
- Destructuring assignment (MDN)
- Spread Syntax (MDN)
- Rest parameters (MDN)