JavaScript Essentials – Ein Primer für Database-Beispiele

Willkommen zum JavaScript-Primer! Dieses Tutorial ist speziell für alle, die wenig oder keine JavaScript-Erfahrung haben, aber die Code-Beispiele in unserer Datenbank-Vorlesung verstehen möchten. Sie müssen kein JavaScript-Experte werden – aber Sie sollten die Grundkonzepte kennen, um die Datenbankoperationen nachvollziehen zu können.

🎯 Was Sie hier lernen

Ziel dieses Primers:

> Verstehen Sie JavaScript gut genug, um Database-Code zu lesen, zu verstehen und anzupassen – ohne Frontend-Entwickler zu werden.

Was behandelt wird:

- ✅ Variablen & Datentypen
- ✅ Kontrollstrukturen (if/else, Schleifen)
- ✅ Funktionen & Arrow Functions
- ✅ Objekte & Arrays (mit wichtigen Methoden)
- ✅ Asynchronität (async/await, Promises)
- ✅ JSON & Datenformate
- ✅ Classes (Bonus)
- ✅ Console & Debugging

Was NICHT behandelt wird:

- ❌ DOM-Manipulation (HTML/CSS)
- ❌ Frontend-Frameworks (React, Vue, etc.)
- ❌ Node.js-spezifische APIs
- ❌ Fortgeschrittene Patterns (Closures, Prototypes, etc.)

Geschätzter Zeitaufwand: 60-90 Minuten

JavaScript ist die Sprache des Webs – und damit perfekt für unsere Browser-basierten Datenbank-Demos. Wir nutzen moderne JavaScript-Syntax (ES6+), die Sie in allen aktuellen Browsern ausführen können. Keine Installation nötig – öffnen Sie einfach die Browser DevTools Console und experimentieren Sie!

Kapitel 1: Grundlagen – Variablen & Datentypen

Starten wir mit den absoluten Basics: Wie speichern wir Daten in JavaScript? Welche Arten von Daten gibt es? Variablen sind Container für Werte – wie beschriftete Boxen, in die Sie Dinge legen können. JavaScript ist dynamisch typisiert, das heißt: Sie müssen nicht im Voraus festlegen, welchen Datentyp eine Variable haben wird. Das macht die Sprache flexibel, aber auch fehleranfällig, wenn Sie nicht aufpassen.

1.1 Variablen deklarieren

Drei Arten, Variablen zu deklarieren:

```
console.log("=== Beispiel 1: Variablen-Deklaration ===");

// 1. const - Kann NICHT neu zugewiesen werden (Konstante)
const name = "Alice";
console.log("const name:", name);
// name = "Bob";  // ❌ Fehler: Assignment to constant variable

// 2. let - Kann neu zugewiesen werden (Block-Scope)
let age = 30;
console.log("let age (initial):", age);
age = 31;  // ✅ Funktioniert
console.log("let age (nach Änderung):", age);

// 3. var - ALT, vermeiden! (Function-Scope, Hoisting-Probleme)
var city = "Berlin";  // ⚠️ Nur für Legacy-Code
console.log("var city:", city);

console.log("\n💡 Empfehlung: Nutzen Sie const als Standard, let nur nötig!");
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Variablen-Deklaration ===
const name: Alice
let age (initial): 30
let age (nach Änderung): 31
var city: Berlin

💡 Empfehlung: Nutzen Sie const als Standard, let nur wenn nötig!
--- Ende Beispiel 1 ---
```

Welche sollten Sie nutzen?

Keyword	Verwendung	Empfehlung
`const`	Wert ändert sich nicht	✅ **Standard** – nutzen Sie immer, wenn möglich
`let`	Wert ändert sich	✅ Nur wenn Neuzuweisung nötig
`var`	Legacy	❌ **Nie nutzen** in modernem Code

Faustregel:

> Nutzen Sie **immer** `const` – außer Sie wissen sicher, dass sich der Wert ändern wird. Dann `let`.

Der Unterschied zwischen const und let ist wichtig: const verhindert Neuzuweisungen, aber bei Objekten und Arrays können Sie trotzdem die Inhalte ändern – dazu später mehr. Var sollten Sie komplett vergessen – es hat verwirrende Scoping-Regeln und wurde durch let/const ersetzt.

Beispiel: const vs. let in der Praxis

```
1  console.log("=== Beispiel 2: const vs. let in der Praxis ===");
2
3  // Datenbankverbindung - ändert sich nie
4  const DB_NAME = "products_db";
5  const API_URL = "https://api.example.com";
6  console.log("Konstanten definiert:");
7  console.log("  DB_NAME:", DB_NAME);
8  console.log("  API_URL:", API_URL);
9
10 // Counter, der hochgezählt wird
11 let productCount = 0;
12 console.log("\nCounter initial:", productCount);
13 productCount++;  // ✅ OK mit let
14 console.log("Counter nach Inkrement:", productCount);
15
16 // Wichtig: const verhindert nur Neuzuweisung, nicht Mutation!
17 const products = [];
18 console.log("\nArray initial:", products);
19
20 products.push({ name: "Laptop" });  // ✅ OK - Array wird modifiziert
21 console.log("Array nach push:", products);
22
23 // products = [];  // ❌ Fehler - Neuzuweisung nicht erlaubt
24 console.log("\n💡 const verhindert Neuzuweisung, nicht Mutation von
     Objekten/Arrays!");
25 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: const vs. let in der Praxis ===
Konstanten definiert:
  DB_NAME: products_db
  API_URL: https://api.example.com

Counter initial: 0
Counter nach Inkrement: 1

Array initial: []
Array nach push: [{"name":"Laptop"}]

💡 const verhindert Neuzuweisung, nicht Mutation von Objekten/Arrays!
--- Ende Beispiel 2 ---
```

Live-Test in der Console:

```
1  console.log("=== Live-Test: const Neuzuweisung ===");
2  const x = 10;
3  console.log("x initial:", x);
4
5  try {
6    x = 20;  // Was passiert?
7    console.log("x nach Zuweisung:", x);
8  } catch (error) {
9    console.log("❌ Fehler:", error.message);
10 }
11 console.log("--- Ende Test ---");
```

```
=== Live-Test: const Neuzuweisung ===
x initial: 10
❌ Fehler: Assignment to constant variable.
--- Ende Test ---
```

Dieser Unterschied ist subtil, aber zentral: const bedeutet „die Referenz ändert sich nicht“, nicht „der Inhalt ändert sich nicht“. Bei primitiven Werten wie Zahlen oder Strings ist das egal. Bei Objekten und Arrays können Sie Inhalte ändern, aber nicht das ganze Objekt ersetzen.

1.2 Primitive Datentypen

JavaScript hat 7 primitive Datentypen:

Typ	Beschreibung	Beispiele
`Number`	Ganzzahlen & Dezimalzahlen	`42`, `3.14`, `-100`, `Infinity`
`String`	Text (in `""`, `''` oder Backticks)	`"Hello"`, `'World'`, `` `Hi` ``
`Boolean`	Wahr oder Falsch	`true`, `false`
`null`	Absichtlich leerer Wert	`null`
`undefined`	Variable ohne Wert	`undefined`
`BigInt`	Sehr große Ganzzahlen	`9007199254740991n`
`Symbol`	Eindeutiger Identifier	`Symbol("id")`

Wichtig für unsere Vorlesung: Die ersten 5 reichen völlig! BigInt und Symbol sind Spezialfälle.
Schauen wir uns die wichtigsten Typen im Detail an. Number, String und Boolean werden Sie ständig nutzen. Null und undefined sind oft verwirrend – aber wichtig zu verstehen, weil sie in Datenbank-Queries vorkommen können.

1. Number – Zahlen

```
1  console.log("=== Beispiel 3: Number-Datentyp ===");
2
3  // Ganzzahlen
4  const count = 42;
5  const negative = -17;
6  console.log("Ganzzahlen:");
7  console.log("  count:", count, "| typeof:", typeof count);
8  console.log("  negative:", negative);
9
10 // Dezimalzahlen (Floats)
11 const price = 19.99;
12 const pi = 3.14159;
13 console.log("\nDezimalzahlen:");
14 console.log("  price:", price);
15 console.log("  pi:", pi);
16
17 // Besondere Werte
18 const infinity = Infinity;
19 const notANumber = NaN;
20 console.log("\nBesondere Werte:");
```

```
21  console.log("  Infinity:", infinity);
22  console.log("  NaN:", notANumber, "| typeof:", typeof notANumber);
23
24  // Rechnen
25  const sum = 10 + 5;
26  const product = 10 * 5;
27  const division = 10 / 3;
28  console.log("\nBerechnungen:");
29  console.log("  10 + 5 =", sum);
30  console.log("  10 × 5 =", product);
31  console.log("  10 ÷ 3 =", division);
32
33  // Achtung: Floating Point Probleme!
34  const floatCalc = 0.1 + 0.2;
35  console.log("\n⚠️  Floating Point Problem:");
36  console.log("  0.1 + 0.2 =", floatCalc);
37  console.log("  (erwartet: 0.3, tatsächlich:", floatCalc, ")");
38
39  console.log("\n💡 Bei Geld: Mit Cents (Integer) rechnen!");
40  console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Number-Datentyp ===
Ganzzahlen:
  count: 42 | typeof: number
  negative: -17

Dezimalzahlen:
  price: 19.99
  pi: 3.14159

Besondere Werte:
  Infinity: null
  NaN: null | typeof: number

Berechnungen:
  10 + 5 = 15
  10 × 5 = 50
  10 ÷ 3 = 3.3333333333333335

⚠️  Floating Point Problem:
  0.1 + 0.2 = 0.30000000000000004
  (erwartet: 0.3, tatsächlich: 0.30000000000000004 )

💡 Bei Geld: Mit Cents (Integer) rechnen!
--- Ende Beispiel 3 ---
```

Wichtig:

- Kein Unterschied zwischen Integer und Float (alles `Number`)
- `NaN` ist technisch vom Typ `Number` (paradox!)
- Floating Point Arithmetik ist ungenau → bei Geld mit Cents rechnen!

JavaScript hat nur einen Zahlentyp – das vereinfacht vieles, kann aber bei Präzision Probleme machen. Für Datenbank-Beispiele reicht das völlig, aber merken Sie sich: Bei Geldbeträgen niemals direkt mit Dezimalzahlen rechnen, sondern in Cents umrechnen.

2. String – Text

```
console.log("=== Beispiel 4: String-Datentyp ===");

// Drei Arten, Strings zu schreiben
const single = 'Einfache Anführungszeichen';
const double = "Doppelte Anführungszeichen";
const backtick = `Backticks (Template Literals)`;
console.log("Drei String-Varianten:");
console.log("  single:", single);
console.log("  double:", double);
console.log("  backtick:", backtick);

// String-Verkettung (alt)
const firstName = "Alice";
const lastName = "Smith";
const fullName = firstName + " " + lastName;
console.log("\nString-Verkettung (alt):");
console.log("  firstName + ' ' + lastName =", fullName);

// Template Literals (modern, empfohlen!)
const greeting = `Hello, ${firstName}!`;
const multi = `Zeile 1
Zeile 2`;
console.log("\nTemplate Literals (modern):");
console.log("  greeting:", greeting);
console.log("  multi:", multi);

// String-Eigenschaften & Methoden
console.log("\nString-Methoden:");
console.log("  firstName.length:", firstName.length);
console.log("  firstName.toUpperCase():", firstName.toUpperCase());
console.log("  firstName.toLowerCase():", firstName.toLowerCase());
console.log("  firstName.includes('li'):", firstName.includes("li"));

console.log("\n💡 Template Literals sind modern und lesbar!");
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: String-Datentyp ===
Drei String-Varianten:
  single: Einfache Anführungszeichen
  double: Doppelte Anführungszeichen
  backtick: Backticks (Template Literals)

String-Verkettung (alt):
  firstName + ' ' + lastName = Alice Smith

Template Literals (modern):
  greeting: Hello, Alice!
  multi: Zeile 1
Zeile 2

String-Methoden:
  firstName.length: 5
  firstName.toUpperCase(): ALICE
  firstName.toLowerCase(): alice
  firstName.includes('li'): true

💡 Template Literals sind modern und lesbar!
--- Ende Beispiel 4 ---
```

Template Literals (Backticks) sind modern und mächtig:

```js
console.log("=== Beispiel 5: Template Literals im Detail ===");

const product = "Laptop";
const price = 999;

// Alt (mühsam):
const message1 = "Product: " + product + ", Price: " + price + "€";
console.log("Alt (String-Konkatenation):");
console.log("  ", message1);

// Modern (lesbar):
const message2 = `Product: ${product}, Price: ${price}€`;
console.log("\nModern (Template Literal):");
console.log("  ", message2);

// Mit Berechnungen
const taxRate = 0.19;
const priceWithTax = price * (1 + taxRate);
const message3 = `Product: ${product}, Price: ${price}€, With Tax:
  ${priceWithTax.toFixed(2)}€`;
console.log("\nMit Berechnungen:");
```

```
21  console.log("  ", message3);
22
23  console.log("\n💡 ${} kann beliebige JavaScript-Ausdrücke enthalten!"
24  console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Template Literals im Detail ===
Alt (String-Konkatenation):
   Product: Laptop, Price: 999€

Modern (Template Literal):
   Product: Laptop, Price: 999€

Mit Berechnungen:
   Product: Laptop, Price: 999€, With Tax: 1188.81€

💡 ${} kann beliebige JavaScript-Ausdrücke enthalten!
--- Ende Beispiel 5 ---
```

Template Literals mit Backticks werden Sie in unseren Beispielen ständig sehen – sie machen String-Interpolation so viel einfacher! Statt plus-Zeichen zu jonglieren, schreiben Sie einfach geschweifte Klammern mit Dollarzeichen. Gewöhnen Sie sich das direkt an.

3. Boolean – Wahr oder Falsch

```
1   console.log("=== Beispiel 6: Boolean-Datentyp ===");
2
3   // Nur zwei mögliche Werte
4   const isActive = true;
5   const isDeleted = false;
6   console.log("Boolean-Werte:");
7   console.log("  isActive:", isActive, "| typeof:", typeof isActive);
8   console.log("  isDeleted:", isDeleted);
9
10  // Oft Ergebnis von Vergleichen
11  const age = 25;
12  const stock = 10;
13  const isAdult = age >= 18;
14  const hasStock = stock > 0;
15  const isEqual = (5 === 5);
16  console.log("\nVergleiche (ergeben Boolean):");
17  console.log("  age >= 18:", isAdult);
18  console.log("  stock > 0:", hasStock);
19  console.log("  5 === 5:", isEqual);
20
21  // Boolean-Operatoren
22  const isAdmin = true;
23  const isPremium = false;
24  const canBuy = isAdult && hasStock;
```

```
25 const hasAccess = isAdmin || isPremium;
26 const isInactive = !isActive;
27 console.log("\nBoolean-Operatoren:");
28 console.log("  isAdult && hasStock (AND):", canBuy);
29 console.log("  isAdmin || isPremium (OR):", hasAccess);
30 console.log("  !isActive (NOT):", isInactive);
31
32 console.log("\n💡 Boolean = true/false, Grundlage für Bedingungen!");
33 console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Boolean-Datentyp ===
Boolean-Werte:
  isActive: true | typeof: boolean
  isDeleted: false

Vergleiche (ergeben Boolean):
  age >= 18: true
  stock > 0: true
  5 === 5: true

Boolean-Operatoren:
  isAdult && hasStock (AND): true
  isAdmin || isPremium (OR): true
  !isActive (NOT): false

💡 Boolean = true/false, Grundlage für Bedingungen!
--- Ende Beispiel 6 ---
```

Truthy & Falsy (wichtig!):

JavaScript konvertiert Werte automatisch zu Boolean in Bedingungen:

```
1  console.log("=== Beispiel 7: Truthy & Falsy ===");
2
3  console.log("Falsy-Werte (werden zu false):");
4  console.log("  if (0) →", Boolean(0), "→ nicht ausgeführt");
5  console.log("  if ('') →", Boolean(""), "→ nicht ausgeführt");
6  console.log("  if (null) →", Boolean(null), "→ nicht ausgeführt");
7  console.log("  if (undefined) →", Boolean(undefined), "→ nicht ausge
     );
8  console.log("  if (NaN) →", Boolean(NaN), "→ nicht ausgeführt");
9  console.log("  if (false) →", Boolean(false), "→ nicht ausgeführt");
10
11 console.log("\nTruthy-Werte (werden zu true):");
12 console.log("  if (1) →", Boolean(1), "→ ausgeführt");
13 console.log("  if ('text') →", Boolean("text"), "→ ausgeführt");
14 console.log("  if ([]) →", Boolean([]), "→ ausgeführt (⚠️ auch leere
```

```
  Arrays!)");
console.log("  if ({}) →", Boolean({}), "→ ausgeführt (⚠️ auch leere
  Objekte!)");

console.log("\n⚠️  Wichtig: Leere Arrays/Objekte sind truthy!");
console.log("💡 Boolean(wert) zeigt die Konvertierung");
console.log("--- Ende Beispiel 7 ---");
```

```
=== Beispiel 7: Truthy & Falsy ===
Falsy-Werte (werden zu false):
  if (0) → false → nicht ausgeführt
  if ('') → false → nicht ausgeführt
  if (null) → false → nicht ausgeführt
  if (undefined) → false → nicht ausgeführt
  if (NaN) → false → nicht ausgeführt
  if (false) → false → nicht ausgeführt

Truthy-Werte (werden zu true):
  if (1) → true → ausgeführt
  if ('text') → true → ausgeführt
  if ([]) → true → ausgeführt (⚠️ auch leere Arrays!)
  if ({}) → true → ausgeführt (⚠️ auch leere Objekte!)

⚠️  Wichtig: Leere Arrays/Objekte sind truthy!
💡 Boolean(wert) zeigt die Konvertierung
--- Ende Beispiel 7 ---
```

Truthy und Falsy sind extrem wichtig zu verstehen – viele Bugs entstehen, weil Entwickler vergessen, dass leere Arrays oder Objekte als true gelten! In Datenbank-Code prüfen Sie oft, ob ein Wert existiert – und da müssen Sie wissen, was JavaScript als „wahr“ oder „falsch“ ansieht.

4. null & undefined – Leere Werte

```
console.log("=== Beispiel 8: null & undefined ===");

// undefined - Variable existiert, hat aber keinen Wert
let product;
console.log("let product (ohne Zuweisung):");
console.log("  Wert:", product);
console.log("  typeof:", typeof product);

// null - Absichtlich leerer Wert
let user = null;
console.log("\nlet user = null:");
console.log("  Wert:", user);
console.log("  typeof:", typeof user, "(⚠️ historischer Bug!)");

// Typischer Use Case in Datenbanken
```

```
16 console.log("\nDatabase-Kontext:");
17 console.log("  const result = await db.get('user_123');");
18 console.log("  if (result === null) { /* User nicht gefunden */ }");
19
20 console.log("\n💡 undefined = automatisch, null = explizit leer");
21 console.log("--- Ende Beispiel 8 ---");
```

```
=== Beispiel 8: null & undefined ===
let product (ohne Zuweisung):
  Wert: undefined
  typeof: undefined

let user = null:
  Wert: null
  typeof: object (⚠️ historischer Bug!)

Database-Kontext:
  const result = await db.get('user_123');
  if (result === null) { /* User nicht gefunden */ }

💡 undefined = automatisch, null = explizit leer
--- Ende Beispiel 8 ---
```

Unterschied:

Wert	Bedeutung	Verwendung
`undefined`	„Wert nicht gesetzt“	Automatisch von JavaScript
`null`	„Absichtlich leer“	Explizit von Entwicklern

Häufiger Fehler:

```
1  console.log("=== Beispiel 9: null vs. undefined Vergleich ===");
2
3  // Vorsicht beim Vergleich!
4  console.log("Lose Gleichheit (==):");
5  console.log("  null == undefined:", null == undefined);  // true
6
7  console.log("\nStrikte Gleichheit (===):");
8  console.log("  null === undefined:", null === undefined);  // false
9
10 console.log("\nPraxis-Tipp - beide prüfen:");
11 const result = null;
12 console.log("  result:", result);
```

```
console.log("  result == null:", result == null, "(prüft null UND und
  )");
console.log("  Äquivalent zu: result === null || result === undefined
);

console.log("\n💡 'value == null' ist der akzeptierte Shortcut!");
console.log("--- Ende Beispiel 9 ---");
```

```
=== Beispiel 9: null vs. undefined Vergleich ===
Lose Gleichheit (==):
  null == undefined: true

Strikte Gleichheit (===):
  null === undefined: false

Praxis-Tipp - beide prüfen:
  result: null
  result == null: true (prüft null UND undefined)
  Äquivalent zu: result === null || result === undefined

💡 'value == null' ist der akzeptierte Shortcut!
--- Ende Beispiel 9 ---
```

Null versus undefined verwirrt Anfänger oft. Merken Sie sich: undefined ist JavaScript's Art zu sagen „da ist nichts“, null ist Ihre Art zu sagen „ich will, dass da nichts ist“. In Datenbank-Operationen nutzen wir oft null für fehlende Werte – das ist expliziter und besser nachvollziehbar.

1.3 typeof Operator

Wie finden Sie heraus, welchen Typ ein Wert hat?

```
console.log("=== Beispiel 10: typeof Operator ===");

// typeof gibt den Typ als String zurück
console.log("typeof für verschiedene Datentypen:");
console.log("  typeof 42:", typeof 42);
console.log("  typeof 'Hello':", typeof "Hello");
console.log("  typeof true:", typeof true);
console.log("  typeof undefined:", typeof undefined);
console.log("  typeof null:", typeof null, "(⚠️ Bug!)");
console.log("  typeof {}:", typeof {});
console.log("  typeof []:", typeof []);
console.log("  typeof function(){}:", typeof function(){});

console.log("\n💡 typeof ist nützlich, aber hat Quirks!");
console.log("--- Ende Beispiel 10 ---");
```

```
=== Beispiel 10: typeof Operator ===
typeof für verschiedene Datentypen:
  typeof 42: number
  typeof 'Hello': string
  typeof true: boolean
  typeof undefined: undefined
  typeof null: object (⚠️ Bug!)
  typeof {}: object
  typeof []: object
  typeof function(){}: function

💡 typeof ist nützlich, aber hat Quirks!
--- Ende Beispiel 10 ---
```

Wichtige Quirks:

Expression	Ergebnis	Kommentar
`typeof null`	`"object"`	⚠️ Historischer Bug, kann nicht gefixt werden
`typeof []`	`"object"`	Arrays sind spezielle Objekte
`typeof NaN`	`"number"`	Paradox: „Not a Number“ ist vom Typ Number

Praktischer Check für Arrays:

```
1  console.log("=== Beispiel 11: Arrays richtig prüfen ===");
2
3  const data = [1, 2, 3];
4  console.log("const data = [1, 2, 3]");
5  console.log("\ntypeof data:", typeof data, "(nicht hilfreich für Arra
     );
6  console.log("Array.isArray(data):", Array.isArray(data), "(✅ richtig
     Methode!)");
7
8  const notArray = { length: 3 };
9  console.log("\nconst notArray = { length: 3 }");
10 console.log("typeof notArray:", typeof notArray);
11 console.log("Array.isArray(notArray):", Array.isArray(notArray));
12
13 console.log("\n💡 Für Arrays: Array.isArray() nutzen, nicht typeof!")
14 console.log("--- Ende Beispiel 11 ---");
```

```
=== Beispiel 11: Arrays richtig prüfen ===
const data = [1, 2, 3]

typeof data: object (nicht hilfreich für Arrays!)
Array.isArray(data): true (✅ richtige Methode!)

const notArray = { length: 3 }
typeof notArray: object
Array.isArray(notArray): false

💡 Für Arrays: Array.isArray() nutzen, nicht typeof!
--- Ende Beispiel 11 ---
```

Typeof ist nützlich, aber nicht perfekt. Der größte Stolperstein: typeof null gibt „object“ zurück – ein historischer Fehler, der aus Kompatibilitätsgründen nie gefixt werden kann. Für Arrays nutzen Sie Array.isArray, nicht typeof.

1.4 Type Coercion & Equality

JavaScript konvertiert Typen automatisch – manchmal hilfreich, oft verwirrend!

Implizite Konvertierung:

```
1  console.log("=== Beispiel 12: Type Coercion ===");
2
3  // Zahlen + Strings
4  console.log("Type Coercion bei Operationen:");
5  console.log("  5 + '5' =", 5 + "5", "(Number → String)");
6  console.log("  5 - '2' =", 5 - "2", "(String → Number)");
7  console.log("  '10' * '2' =", "10" * "2", "(beide → Number)");
8
9  // Boolean zu Number
10 console.log("\nBoolean zu Number:");
11 console.log("  true + 1 =", true + 1, "(true → 1)");
12 console.log("  false + 1 =", false + 1, "(false → 0)");
13
14 // Vergleiche
15 console.log("\nVergleiche (lose vs. strikte Gleichheit):");
16 console.log("  5 == '5':", 5 == "5", "(lose, konvertiert!)");
17 console.log("  5 === '5':", 5 === "5", "(strikt, keine Konvertierung)
18
19 console.log("\n⚠️  Type Coercion kann verwirrend sein!");
20 console.log("--- Ende Beispiel 12 ---");
```

```
=== Beispiel 12: Type Coercion ===
Type Coercion bei Operationen:
  5 + '5' = 55 (Number → String)
  5 - '2' = 3 (String → Number)
  '10' * '2' = 20 (beide → Number)

Boolean zu Number:
  true + 1 = 2 (true → 1)
  false + 1 = 1 (false → 0)

Vergleiche (lose vs. strikte Gleichheit):
  5 == '5': true (lose, konvertiert!)
  5 === '5': false (strikt, keine Konvertierung)

⚠️  Type Coercion kann verwirrend sein!
--- Ende Beispiel 12 ---
```

Zwei Arten von Gleichheit:

Operator	Name	Konvertiert Typen?	Empfehlung
`==`	Lose Gleichheit	✅ Ja	❌ Vermeiden
`===`	Strikte Gleichheit	❌ Nein	✅ **Immer nutzen**
`!=`	Lose Ungleichheit	✅ Ja	❌ Vermeiden
`!==`	Strikte Ungleichheit	❌ Nein	✅ **Immer nutzen**

Warum immer `===` nutzen?

```js
console.log("=== Beispiel 13: Verwirrende Fälle mit == ===");

// Verwirrende Fälle mit ==
console.log("Lose Gleichheit (==) - verwirrend:");
console.log("  0 == false:", 0 == false, "😵");
console.log("  '' == false:", "" == false, "😵");
console.log("  null == undefined:", null == undefined, "😵");
console.log("  '0' == false:", "0" == false, "😵");

// Mit === klar und vorhersagbar
console.log("\nStrikte Gleichheit (===) - vorhersagbar:");
console.log("  0 === false:", 0 === false, "✅");
console.log("  '' === false:", "" === false, "✅");
```

```
14 console.log("  null === undefined:", null === undefined, "✅");
15 console.log("  '0' === false:", "0" === false, "✅");
16
17 console.log("\n💡 Nutzen Sie IMMER === (drei Gleichheitszeichen)!");
18 console.log("--- Ende Beispiel 13 ---");
```

```
=== Beispiel 13: Verwirrende Fälle mit == ===
Lose Gleichheit (==) - verwirrend:
  0 == false: true 😵
  '' == false: true 😵
  null == undefined: true 😵
  '0' == false: true 😵

Strikte Gleichheit (===) - vorhersagbar:
  0 === false: false ✅
  '' === false: false ✅
  null === undefined: false ✅
  '0' === false: false ✅

💡 Nutzen Sie IMMER === (drei Gleichheitszeichen)!
--- Ende Beispiel 13 ---
```

Faustregel:

Nutzen Sie **immer** `===` **und** `!==` – außer Sie prüfen explizit auf null/undefined:

```
1  console.log("=== Beispiel 14: Einzige Ausnahme für == ===");
2
3  const value = null;
4  console.log("const value = null");
5
6  // Einzige Ausnahme: Prüfung auf null/undefined
7  console.log("\nvalue == null:", value == null, "(prüft null UND undef
     );
8
9  console.log("\nÄquivalent zu:");
10 console.log("  value === null || value === undefined");
11
12 // Praktischer Test
13 const testValues = [null, undefined, 0, false, ""];
14 console.log("\nTest verschiedener Werte mit '== null':");
15 testValues.forEach(val => {
16   console.log(`  ${String(val).padEnd(10)} == null:`, val == null);
17 });
18
```

```
console.log("\n💡 'value == null' ist der akzeptierte Shortcut!");
console.log("--- Ende Beispiel 14 ---");
```

```
=== Beispiel 14: Einzige Ausnahme für == ===
const value = null

value == null: true (prüft null UND undefined)

Äquivalent zu:
  value === null || value === undefined

Test verschiedener Werte mit '== null':
  null       == null: true
  undefined  == null: true
  0          == null: false
  false      == null: false
             == null: false

💡 'value == null' ist der akzeptierte Shortcut!
--- Ende Beispiel 14 ---
```

Type Coercion ist eine der größten Fehlerquellen in JavaScript. Die goldene Regel: Nutzen Sie immer dreifaches Gleichheitszeichen. Das macht Ihren Code vorhersagbar und vermeidet 90 Prozent der verwirrenden Bugs. Die einzige Ausnahme: Das Prüfen auf null/undefined mit doppeltem Gleichheitszeichen ist ein akzeptierter Shortcut.

1.5 Übung: Datentypen

Probieren Sie diese Aufgaben direkt in der Browser Console aus:

Aufgabe 1: Variablen deklarieren

```
// Deklarieren Sie:
// 1. Eine Konstante 'userName' mit Ihrem Namen
// 2. Eine Variable 'score', die sich ändern kann, Start: 0
// 3. Eine Konstante 'maxScore' mit Wert 100

// Ihre Lösung hier:
```

Aufgabe 2: Datentypen erkennen

```
// Was gibt typeof zurück? Raten Sie erst, dann testen Sie:
typeof "123"
typeof 123
typeof true
```

```
typeof null
typeof undefined
typeof []
typeof {}
```

Aufgabe 3: Vergleiche

```
// Was ist das Ergebnis? Raten Sie, dann testen Sie:
5 == "5"
5 === "5"
0 == false
0 === false
null == undefined
null === undefined
```

Aufgabe 4: Template Literals

```
// Schreiben Sie einen String mit Template Literals:
const product = "Laptop";
const price = 999;
const stock = 5;

// Erstellen Sie: "Product: Laptop, Price: 999€, Stock: 5 units"
// Ihre Lösung hier:
```

Aufgabe 5: Truthy/Falsy

```
// Welche dieser if-Bedingungen werden ausgeführt?
if (0) { console.log("A"); }
if ("") { console.log("B"); }
if ("0") { console.log("C"); }
if ([]) { console.log("D"); }
if (null) { console.log("E"); }
```

```
C
D
```

Nehmen Sie sich Zeit für diese Übungen – sie sind das Fundament für alles Weitere. Wenn Sie die Lösungen nicht sofort wissen, ist das normal! Experimentieren Sie in der Console, machen Sie Fehler, beobachten Sie, was passiert. Das ist der beste Weg, JavaScript zu lernen.

Lösungen:

▶ Lösung Aufgabe 1 (klicken zum Aufklappen)
▶ Lösung Aufgabe 2
▶ Lösung Aufgabe 3
▶ Lösung Aufgabe 4
▶ Lösung Aufgabe 5

Geschafft! Sie haben jetzt die Grundlagen von Variablen und Datentypen verstanden. Das sind die Bausteine für alles Weitere. Wenn etwas unklar ist, kommen Sie zu diesem Kapitel zurück – es ist Ihre Referenz. Bereit für Kapitel 2?

Kapitel 2: Kontrollstrukturen – Entscheidungen & Wiederholungen

Programme müssen Entscheidungen treffen und Aktionen wiederholen. Dafür gibt es Kontrollstrukturen. In Datenbank-Anwendungen nutzen Sie diese ständig: Prüfen Sie, ob ein Datensatz existiert? Iterieren Sie über Suchergebnisse? Validieren Sie Eingaben? All das sind Kontrollstrukturen. Wir zeigen Ihnen jede mit vielen console.log-Ausgaben, damit Sie den Programmfluss verstehen.

Was sind Kontrollstrukturen?

- **Verzweigungen** (`if`, `else`, `switch`): Programme treffen Entscheidungen
- **Schleifen** (`for`, `while`, `for...of`): Programme wiederholen Aktionen
- **Sprünge** (`break`, `continue`): Programme überspringen oder beenden Iterationen

---

2.1 if/else/else if

Die if-Anweisung ist die grundlegendste Kontrollstruktur. Sie prüft eine Bedingung und führt Code aus, wenn diese wahr ist. Mit else und else if können Sie Alternativen definieren. Schauen wir uns an, wie das in der Praxis funktioniert – mit vielen console.log-Ausgaben, um den Fluss zu sehen.

Grundform:

```
1  console.log("=== Beispiel 1: Einfaches if ===");
2  const age = 20;
3  console.log("Alter:", age);
4
5  if (age >= 18) {
6    console.log("✅ Volljährig");
7  }
8  console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Einfaches if ===
Alter: 20
✅ Volljährig
--- Ende Beispiel 1 ---
```

Mit else:

```
1  console.log("=== Beispiel 2: if...else ===");
2  const stock = 0;
3  console.log("Lagerbestand:", stock);
4
5  if (stock > 0) {
6    console.log("✅ Produkt verfügbar");
7  } else {
8    console.log("❌ Ausverkauft");
9  }
10 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: if...else ===
Lagerbestand: 0
❌ Ausverkauft
--- Ende Beispiel 2 ---
```

Mit else if (mehrere Bedingungen):

```
1  console.log("=== Beispiel 3: if...else if...else ===");
2  const score = 75;
3  console.log("Punktzahl:", score);
4
5  if (score >= 90) {
6    console.log("🏆 Note: Sehr gut");
7  } else if (score >= 75) {
8    console.log("👍 Note: Gut");
9  } else if (score >= 60) {
10   console.log("✔️ Note: Befriedigend");
11 } else if (score >= 50) {
12   console.log("⚠️ Note: Ausreichend");
13 } else {
14   console.log("❌ Note: Nicht bestanden");
15 }
16 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: if...else if...else ===
Punktzahl: 75
👍 Note: Gut
--- Ende Beispiel 3 ---
```

Praxis-Beispiel: Datenbank-Validierung

```
1  console.log("=== Beispiel 4: User-Validierung ===");
2  const username = "alice";
3  const password = "1234";
4
5  console.log("Eingabe - Username:", username);
6  console.log("Eingabe - Password:", password);
7
8  if (!username) {
9    console.log("❌ Fehler: Username fehlt");
10 } else if (username.length < 3) {
11   console.log("❌ Fehler: Username zu kurz (min. 3 Zeichen)");
12 } else if (!password) {
13   console.log("❌ Fehler: Password fehlt");
14 } else if (password.length < 8) {
15   console.log("⚠️ Warnung: Schwaches Passwort (min. 8 Zeichen empfohl
16   console.log("✅ Login erlaubt (mit Warnung)");
17 } else {
18   console.log("✅ Login erfolgreich");
19 }
20 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: User-Validierung ===
Eingabe - Username: alice
Eingabe - Password: 1234
⚠️ Warnung: Schwaches Passwort (min. 8 Zeichen empfohlen)
✅ Login erlaubt (mit Warnung)
--- Ende Beispiel 4 ---
```

Verschachtelte if-Statements:

```
1  console.log("=== Beispiel 5: Verschachtelt ===");
2  const user = { name: "Bob", role: "admin", active: true };
3  console.log("User:", user);
4
5  if (user.active) {
6    console.log("→ User ist aktiv");
7
8    if (user.role === "admin") {
9      console.log("   → Admin-Rechte erkannt");
10     console.log("   ✅ Zugriff auf Admin-Panel gewährt");
```

```
    console.log("   ✅ Zugriff auf Admin-Panel gewährt");
  } else {
    console.log("   → Standard-User");
    console.log("   ✅ Zugriff auf User-Panel gewährt");
  }
} else {
  console.log("❌ User ist deaktiviert - kein Zugriff");
}
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Verschachtelt ===
User: {"name":"Bob","role":"admin","active":true}
→ User ist aktiv
  → Admin-Rechte erkannt
  ✅ Zugriff auf Admin-Panel gewährt
--- Ende Beispiel 5 ---
```

2.2 Ternary Operator (Kurzform)

Der Ternary Operator ist eine kompakte Alternative zu if-else für einfache Fälle. Die Syntax lautet: Bedingung Fragezeichen Wert-wenn-wahr Doppelpunkt Wert-wenn-falsch. Sehr praktisch für Zuweisungen oder kurze Ausgaben.

Syntax: `bedingung ? wahr : falsch`

```
console.log("=== Beispiel 1: Ternary Basics ===");
const age = 16;
console.log("Alter:", age);

const status = age >= 18 ? "volljährig" : "minderjährig";
console.log("Status:", status);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Ternary Basics ===
Alter: 16
Status: minderjährig
--- Ende Beispiel 1 ---
```

Vergleich: if vs. Ternary

```
console.log("=== Beispiel 2: if vs. Ternary ===");
const stock = 5;
console.log("Lagerbestand:", stock);

// Mit if/else
let message1;
```

```
if (stock > 0) {
  message1 = "verfügbar";
} else {
  message1 = "ausverkauft";
}
console.log("if/else → Message:", message1);

// Mit Ternary (kürzer)
const message2 = stock > 0 ? "verfügbar" : "ausverkauft";
console.log("Ternary → Message:", message2);

console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: if vs. Ternary ===
Lagerbestand: 5
if/else → Message: verfügbar
Ternary → Message: verfügbar
--- Ende Beispiel 2 ---
```

Ternary in console.log:

```
console.log("=== Beispiel 3: Inline in console.log ===");
const price = 99;
console.log("Preis:", price);
console.log("Bewertung:", price > 100 ? "🔴 Teuer" : "🟢 Günstig");

const discount = 0;
console.log("Rabatt:", discount);
console.log("Aktion:", discount > 0 ? `${discount}% Rabatt!` : "Kein
  Rabatt");

console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Inline in console.log ===
Preis: 99
Bewertung: 🟢 Günstig
Rabatt: 0
Aktion: Kein Rabatt
--- Ende Beispiel 3 ---
```

Verschachtelter Ternary (⚠️ nicht übertreiben!):

```
console.log("=== Beispiel 4: Verschachtelter Ternary ===");
const score = 85;
console.log("Punktzahl:", score);

const grade = score >= 90 ? "A" : score >= 80 ? "B" : score >= 70 ? "C
```

```
  "F";
console.log("Note:", grade);

// ⚠️ Bei mehr als 2 Ebenen besser if/else nutzen!
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Verschachtelter Ternary ===
Punktzahl: 85
Note: B
--- Ende Beispiel 4 ---
```

Praxis: Datenbank-Query-Ergebnis

```
console.log("=== Beispiel 5: DB-Query simuliert ===");
const queryResult = { found: true, data: { id: 123, name: "Laptop" }
console.log("Query-Ergebnis:", queryResult);

const product = queryResult.found
  ? queryResult.data
  : { id: null, name: "Nicht gefunden" };

console.log("→ Produkt:", product);
console.log("→ Name:", product.name);
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: DB-Query simuliert ===
Query-Ergebnis: {"found":true,"data":{"id":123,"name":"Laptop"}}
→ Produkt: {"id":123,"name":"Laptop"}
→ Name: Laptop
--- Ende Beispiel 5 ---
```

2.3 switch/case (Optional)

Switch-Case ist nützlich, wenn Sie einen Wert gegen viele Möglichkeiten prüfen wollen. Statt vieler else-if-Blöcke schreiben Sie switch mit case-Labels. Wichtig: Vergessen Sie nicht break, sonst läuft der Code in den nächsten Case weiter – das nennt man Fall-Through.

Syntax:

```
console.log("=== Beispiel 1: switch Basics ===");
const day = 3;
console.log("Tag-Nummer:", day);

switch (day) {
  case 1:
```

```
    console.log("→ Montag");
    break;
  case 2:
    console.log("→ Dienstag");
    break;
  case 3:
    console.log("→ Mittwoch");
    break;
  case 4:
    console.log("→ Donnerstag");
    break;
  case 5:
    console.log("→ Freitag");
    break;
  case 6:
  case 7:
    console.log("→ Wochenende! 🎉");
    break;
  default:
    console.log("→ Ungültige Eingabe");
}
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: switch Basics ===
Tag-Nummer: 3
→ Mittwoch
--- Ende Beispiel 1 ---
```

Fall-Through demonstriert:

```
console.log("=== Beispiel 2: Fall-Through (ohne break) ===");
const role = "editor";
console.log("User-Rolle:", role);

let permissions = [];
console.log("Sammle Rechte...");

switch (role) {
  case "admin":
    console.log("  → Admin-Rechte");
    permissions.push("delete");
  case "editor":
    console.log("  → Editor-Rechte");
    permissions.push("edit");
  case "viewer":
    console.log("  → Viewer-Rechte");
    permissions.push("read");
    break;
  default:
```

```
    console.log("   → Keine Rechte");
}

console.log("Finale Rechte:", permissions);
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Fall-Through (ohne break) ===
User-Rolle: editor
Sammle Rechte...
  → Editor-Rechte
  → Viewer-Rechte
Finale Rechte: ["edit","read"]
--- Ende Beispiel 2 ---
```

Praxis: HTTP-Status-Codes

```
console.log("=== Beispiel 3: HTTP-Status-Handler ===");
const statusCode = 404;
console.log("HTTP Status:", statusCode);

switch (statusCode) {
  case 200:
    console.log("✅ OK - Anfrage erfolgreich");
    break;
  case 201:
    console.log("✅ Created - Ressource erstellt");
    break;
  case 400:
    console.log("❌ Bad Request - Ungültige Anfrage");
    break;
  case 401:
    console.log("❌ Unauthorized - Keine Berechtigung");
    break;
  case 404:
    console.log("❌ Not Found - Ressource nicht gefunden");
    break;
  case 500:
    console.log("💥 Internal Server Error");
    break;
  default:
    console.log("⚠️ Unbekannter Status:", statusCode);
}
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: HTTP-Status-Handler ===
HTTP Status: 404
❌ Not Found - Ressource nicht gefunden
--- Ende Beispiel 3 ---
```

Vergleich: if vs. switch

```
1  console.log("=== Beispiel 4: if vs. switch ===");
2  const dbType = "postgres";
3  console.log("Datenbank-Typ:", dbType);
4
5  // Mit if/else
6  console.log("\nMit if/else:");
7  if (dbType === "postgres") {
8    console.log("  → PostgreSQL Port: 5432");
9  } else if (dbType === "mysql") {
10   console.log("  → MySQL Port: 3306");
11 } else if (dbType === "mongodb") {
12   console.log("  → MongoDB Port: 27017");
13 } else {
14   console.log("  → Unbekannter Typ");
15 }
16
17 // Mit switch (übersichtlicher bei vielen Cases)
18 console.log("\nMit switch:");
19 switch (dbType) {
20   case "postgres":
21     console.log("  → PostgreSQL Port: 5432");
22     break;
23   case "mysql":
24     console.log("  → MySQL Port: 3306");
25     break;
26   case "mongodb":
27     console.log("  → MongoDB Port: 27017");
28     break;
29   default:
30     console.log("  → Unbekannter Typ");
31 }
32 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: if vs. switch ===
Datenbank-Typ: postgres

Mit if/else:
  → PostgreSQL Port: 5432

Mit switch:
  → PostgreSQL Port: 5432
--- Ende Beispiel 4 ---
```

2.4 for-Schleife

Die for-Schleife ist der Klassiker für Wiederholungen mit Zähler. Sie besteht aus drei Teilen: Initialisierung, Bedingung, Inkrement. Sehr nützlich, wenn Sie wissen, wie oft etwas wiederholt werden soll. Schauen wir uns an, wie der Zähler bei jeder Iteration fortschreitet.

Syntax: `for (init; bedingung; inkrement)`

```
1  console.log("=== Beispiel 1: for-Schleife Basics ===");
2  console.log("Zähle von 1 bis 5:");
3
4  for (let i = 1; i <= 5; i++) {
5    console.log(`  Iteration ${i}`);
6  }
7  console.log("Schleife beendet");
8  console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: for-Schleife Basics ===
Zähle von 1 bis 5:
  Iteration 1
  Iteration 2
  Iteration 3
  Iteration 4
  Iteration 5
Schleife beendet
--- Ende Beispiel 1 ---
```

Array durchlaufen (klassisch):

```
1  console.log("=== Beispiel 2: Array mit Index ===");
2  const products = ["Laptop", "Maus", "Tastatur", "Monitor"];
3  console.log("Produkte:", products);
4  console.log("Anzahl:", products.length);
5  console.log("\nDurchlaufe Array:");
```

```
6
7  for (let i = 0; i < products.length; i++) {
8    console.log(`  [${i}] → ${products[i]}`);
9  }
10 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Array mit Index ===
Produkte: ["Laptop","Maus","Tastatur","Monitor"]
Anzahl: 4

Durchlaufe Array:
  [0] → Laptop
  [1] → Maus
  [2] → Tastatur
  [3] → Monitor
--- Ende Beispiel 2 ---
```

Rückwärts zählen:

```
1  console.log("=== Beispiel 3: Countdown ===");
2  console.log("Starte Countdown:");
3
4  for (let i = 10; i >= 1; i--) {
5    console.log(`  ${i}...`);
6  }
7  console.log("🚀 Start!");
8  console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Countdown ===
Starte Countdown:
  10...
  9...
  8...
  7...
  6...
  5...
  4...
  3...
  2...
  1...
🚀 Start!
--- Ende Beispiel 3 ---
```

Schrittweite ändern:

```
console.log("=== Beispiel 4: Nur gerade Zahlen ===");
console.log("Gerade Zahlen von 0 bis 20:");

for (let i = 0; i <= 20; i += 2) {
  console.log(`  ${i}`);
}
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Nur gerade Zahlen ===
Gerade Zahlen von 0 bis 20:
  0
  2
  4
  6
  8
  10
  12
  14
  16
  18
  20
--- Ende Beispiel 4 ---
```

Praxis: Daten verarbeiten

```
console.log("=== Beispiel 5: Preis-Berechnung ===");
const prices = [19.99, 29.99, 49.99, 99.99];
console.log("Preise:", prices);

let total = 0;
console.log("\nBerechne Summe:");

for (let i = 0; i < prices.length; i++) {
  console.log(`  Produkt ${i + 1}: ${prices[i]}€`);
  total += prices[i];
  console.log(`     → Zwischensumme: ${total.toFixed(2)}€`);
}

console.log(`\n✅ Gesamtsumme: ${total.toFixed(2)}€`);
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Preis-Berechnung ===
Preise: [19.99,29.99,49.99,99.99]

Berechne Summe:
  Produkt 1: 19.99€
    → Zwischensumme: 19.99€
  Produkt 2: 29.99€
    → Zwischensumme: 49.98€
  Produkt 3: 49.99€
    → Zwischensumme: 99.97€
  Produkt 4: 99.99€
    → Zwischensumme: 199.96€

✅ Gesamtsumme: 199.96€
--- Ende Beispiel 5 ---
```

Verschachtelte Schleifen:

```javascript
console.log("=== Beispiel 6: Verschachtelt (Multiplikationstabelle) =
console.log("2x2 Tabelle:");

for (let row = 1; row <= 2; row++) {
  console.log(`\nZeile ${row}:`);
  for (let col = 1; col <= 2; col++) {
    const result = row * col;
    console.log(`  ${row} × ${col} = ${result}`);
  }
}
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Verschachtelt (Multiplikationstabelle) ===
2x2 Tabelle:

Zeile 1:
  1 × 1 = 1
  1 × 2 = 2

Zeile 2:
  2 × 1 = 2
  2 × 2 = 4
--- Ende Beispiel 6 ---
```

2.5 for...of-Schleife

Die for-of-Schleife ist die moderne, elegante Art, Arrays zu durchlaufen. Sie gibt Ihnen direkt die Werte, nicht die Indices. Viel lesbarer als die klassische for-Schleife, wenn Sie keinen Index brauchen. Perfect für Datenbank-Resultate!

Syntax: `for (element of array)`

```
1  console.log("=== Beispiel 1: for...of Basics ===");
2  const fruits = ["🍎 Apfel", "🍌 Banane", "🍇 Trauben"];
3  console.log("Früchte:", fruits);
4  console.log("\nDurchlaufe mit for...of:");
5
6  for (const fruit of fruits) {
7    console.log(`  → ${fruit}`);
8  }
9  console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: for...of Basics ===
Früchte: ["🍎 Apfel","🍌 Banane","🍇 Trauben"]

Durchlaufe mit for...of:
  → 🍎 Apfel
  → 🍌 Banane
  → 🍇 Trauben
--- Ende Beispiel 1 ---
```

Vergleich: for vs. for...of

```
1  console.log("=== Beispiel 2: for vs. for...of ===");
2  const colors = ["rot", "grün", "blau"];
3  console.log("Farben:", colors);
4
5  console.log("\nMit klassischer for-Schleife:");
6  for (let i = 0; i < colors.length; i++) {
7    console.log(`  [${i}] ${colors[i]}`);
8  }
9
10 console.log("\nMit for...of (einfacher!):");
11 for (const color of colors) {
12   console.log(`  ${color}`);
13 }
14 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: for vs. for...of ===
Farben: ["rot","grün","blau"]

Mit klassischer for-Schleife:
  [0] rot
  [1] grün
  [2] blau

Mit for...of (einfacher!):
  rot
  grün
  blau
--- Ende Beispiel 2 ---
```

Praxis: Datenbank-Ergebnisse verarbeiten

```
1  console.log("=== Beispiel 3: DB-Results simuliert ===");
2  const users = [
3    { id: 1, name: "Alice", active: true },
4    { id: 2, name: "Bob", active: false },
5    { id: 3, name: "Charlie", active: true }
6  ];
7  console.log("Users aus Datenbank:", users);
8  console.log("\nVerarbeite jeden User:");
9
10 for (const user of users) {
11   console.log(`\n→ User ${user.id}: ${user.name}`);
12   console.log(`  Status: ${user.active ? "✅ aktiv" : "❌ inaktiv"}`)
13
14   if (user.active) {
15     console.log(`  Aktion: Sende Willkommens-Email an ${user.name}`);
16   }
17 }
18 console.log("\n--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: DB-Results simuliert ===
Users aus Datenbank: [{"id":1,"name":"Alice","active":true},
{"id":2,"name":"Bob","active":false},
{"id":3,"name":"Charlie","active":true}]

Verarbeite jeden User:

→ User 1: Alice
  Status: ✅ aktiv
  Aktion: Sende Willkommens-Email an Alice

→ User 2: Bob
  Status: ❌ inaktiv

→ User 3: Charlie
  Status: ✅ aktiv
  Aktion: Sende Willkommens-Email an Charlie

--- Ende Beispiel 3 ---
```

Strings durchlaufen:

```javascript
console.log("=== Beispiel 4: String-Iteration ===");
const word = "JavaScript";
console.log("Wort:", word);
console.log("\nBuchstabe für Buchstabe:");

let position = 1;
for (const char of word) {
  console.log(`  Position ${position}: '${char}'`);
  position++;
}
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: String-Iteration ===
Wort: JavaScript

Buchstabe für Buchstabe:
  Position 1: 'J'
  Position 2: 'a'
  Position 3: 'v'
  Position 4: 'a'
  Position 5: 'S'
  Position 6: 'c'
  Position 7: 'r'
  Position 8: 'i'
  Position 9: 'p'
  Position 10: 't'
--- Ende Beispiel 4 ---
```

Summen und Aggregationen:

```javascript
console.log("=== Beispiel 5: Aggregation ===");
const orders = [
  { id: 101, amount: 50 },
  { id: 102, amount: 120 },
  { id: 103, amount: 80 }
];
console.log("Bestellungen:", orders);
console.log("\nBerechne Statistiken:");

let totalAmount = 0;
let orderCount = 0;

for (const order of orders) {
  console.log(`  Order #${order.id}: ${order.amount}€`);
  totalAmount += order.amount;
  orderCount++;
  console.log(`     → Laufende Summe: ${totalAmount}€`);
}

const average = totalAmount / orderCount;
console.log(`\n✅ Gesamt: ${totalAmount}€`);
console.log(`✅ Durchschnitt: ${average.toFixed(2)}€`);
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Aggregation ===
Bestellungen: [{"id":101,"amount":50},{"id":102,"amount":120},
{"id":103,"amount":80}]

Berechne Statistiken:
  Order #101: 50€
    → Laufende Summe: 50€
  Order #102: 120€
    → Laufende Summe: 170€
  Order #103: 80€
    → Laufende Summe: 250€

✅ Gesamt: 250€
✅ Durchschnitt: 83.33€
--- Ende Beispiel 5 ---
```

2.6 while-Schleife

Die while-Schleife wiederholt Code, solange eine Bedingung wahr ist. Anders als for hat sie keinen eingebauten Zähler – Sie müssen selbst aufpassen, dass die Bedingung irgendwann falsch wird, sonst läuft die Schleife ewig! Gut für unbekannte Wiederholungszahlen.

Syntax: `while (bedingung)`

```js
console.log("=== Beispiel 1: while Basics ===");
let count = 1;
console.log("Startwert:", count);
console.log("\nZähle bis 5:");

while (count <= 5) {
  console.log(`  Iteration ${count}`);
  count++;
}
console.log("Endwert:", count);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: while Basics ===
Startwert: 1

Zähle bis 5:
  Iteration 1
  Iteration 2
  Iteration 3
  Iteration 4
  Iteration 5
Endwert: 6
--- Ende Beispiel 1 ---
```

Vorsicht: Endlosschleife vermeiden!

```
1  console.log("=== Beispiel 2: Endlosschleife (VORSICHT!) ===");
2  console.log("⚠️ Dieses Beispiel ist deaktiviert, weil es ewig läuft:"
3  console.log(`
4  let i = 0;
5  while (i < 10) {
6    console.log(i);
7    // ❌ FEHLER: i++ fehlt → Endlosschleife!
8  }
9  `);
10
11 console.log("\n✅ Richtig mit Inkrement:");
12 let i = 0;
13 while (i < 5) {
14   console.log(`  ${i}`);
15   i++; // ✅ Wichtig!
16 }
17 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Endlosschleife (VORSICHT!) ===
⚠️ Dieses Beispiel ist deaktiviert, weil es ewig läuft:

let i = 0;
while (i < 10) {
  console.log(i);
  // ❌ FEHLER: i++ fehlt → Endlosschleife!
}


✅ Richtig mit Inkrement:
  0
  1
  2
  3
  4
--- Ende Beispiel 2 ---
```

do...while (führt mindestens 1x aus):

```javascript
console.log("=== Beispiel 3: do...while ===");
let num = 10;
console.log("Startwert:", num);
console.log("\ndo...while läuft MINDESTENS 1x:");

do {
  console.log(`  num ist ${num}`);
  num--;
} while (num > 8);

console.log("Endwert:", num);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: do...while ===
Startwert: 10

do...while läuft MINDESTENS 1x:
  num ist 10
  num ist 9
Endwert: 8
--- Ende Beispiel 3 ---
```

Praxis: Verarbeitung bis Bedingung erfüllt

```
console.log("=== Beispiel 4: Queue-Processing simuliert ===");
const queue = ["Task 1", "Task 2", "Task 3", "Task 4"];
console.log("Queue:", queue);
console.log("Verarbeite Tasks:");

while (queue.length > 0) {
  const task = queue.shift(); // Entfernt erstes Element
  console.log(`   → Verarbeite: ${task}`);
  console.log(`     Verbleibend: ${queue.length} Tasks`);
}

console.log("✅ Queue leer!");
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Queue-Processing simuliert ===
Queue: ["Task 1","Task 2","Task 3","Task 4"]
Verarbeite Tasks:
   → Verarbeite: Task 1
     Verbleibend: 3 Tasks
   → Verarbeite: Task 2
     Verbleibend: 2 Tasks
   → Verarbeite: Task 3
     Verbleibend: 1 Tasks
   → Verarbeite: Task 4
     Verbleibend: 0 Tasks
✅ Queue leer!
--- Ende Beispiel 4 ---
```

Praxis: Paginierung simuliert

```
console.log("=== Beispiel 5: Pagination ===");
let currentPage = 1;
const totalPages = 4;
const results = [];

console.log(`Lade Daten von Seite 1 bis ${totalPages}:`);

while (currentPage <= totalPages) {
  console.log(`\n→ Lade Seite ${currentPage}...`);
  const pageData = `Daten-Seite-${currentPage}`;
  results.push(pageData);
  console.log(`   ✅ Geladen: ${pageData}`);
  console.log(`   Fortschritt: ${currentPage}/${totalPages}`);
  currentPage++;
}

console.log("\n✅ Alle Seiten geladen!");
```

```
18  console.log("Ergebnis:", results);
19  console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Pagination ===
Lade Daten von Seite 1 bis 4:

→ Lade Seite 1...
  ✅ Geladen: Daten-Seite-1
  Fortschritt: 1/4

→ Lade Seite 2...
  ✅ Geladen: Daten-Seite-2
  Fortschritt: 2/4

→ Lade Seite 3...
  ✅ Geladen: Daten-Seite-3
  Fortschritt: 3/4

→ Lade Seite 4...
  ✅ Geladen: Daten-Seite-4
  Fortschritt: 4/4

✅ Alle Seiten geladen!
Ergebnis: ["Daten-Seite-1","Daten-Seite-2","Daten-Seite-3","Daten-
Seite-4"]
--- Ende Beispiel 5 ---
```

2.7 break & continue

Break und continue sind Kontrollbefehle innerhalb von Schleifen. Break bricht die Schleife komplett ab. Continue überspringt nur die aktuelle Iteration und macht mit der nächsten weiter. Sehr nützlich für vorzeitige Abbrüche oder Filter-Logik.

break: Schleife abbrechen

```
1  console.log("=== Beispiel 1: break ===");
2  console.log("Suche Zahl 7 im Array:");
3  const numbers = [2, 5, 7, 9, 12, 15];
4  console.log("Array:", numbers);
5
6  for (const num of numbers) {
7    console.log(`  Prüfe: ${num}`);
8
9    if (num === 7) {
10     console.log(`  ✅ Gefunden! Breche ab.`);
```

```
    break; // Stoppt die Schleife hier
  }
}
console.log("Nach der Schleife");
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: break ===
Suche Zahl 7 im Array:
Array: [2,5,7,9,12,15]
  Prüfe: 2
  Prüfe: 5
  Prüfe: 7
  ✅ Gefunden! Breche ab.
Nach der Schleife
--- Ende Beispiel 1 ---
```

continue: Iteration überspringen

```
console.log("=== Beispiel 2: continue ===");
console.log("Gebe nur gerade Zahlen aus:");

for (let i = 1; i <= 10; i++) {
  console.log(`  Prüfe: ${i}`);

  if (i % 2 !== 0) {
    console.log(`    → Ungerade, überspringe`);
    continue; // Springt zur nächsten Iteration
  }

  console.log(`    ✅ Gerade: ${i}`);
}
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: continue ===
Gebe nur gerade Zahlen aus:
  Prüfe: 1
    → Ungerade, überspringe
  Prüfe: 2
    ✅ Gerade: 2
  Prüfe: 3
    → Ungerade, überspringe
  Prüfe: 4
    ✅ Gerade: 4
  Prüfe: 5
    → Ungerade, überspringe
  Prüfe: 6
    ✅ Gerade: 6
  Prüfe: 7
    → Ungerade, überspringe
  Prüfe: 8
    ✅ Gerade: 8
  Prüfe: 9
    → Ungerade, überspringe
  Prüfe: 10
    ✅ Gerade: 10
--- Ende Beispiel 2 ---
```

Praxis: User-Validierung mit continue

```
1  console.log("=== Beispiel 3: Validierung mit continue ===");
2  const users = [
3    { name: "Alice", email: "alice@example.com" },
4    { name: "Bob", email: "" },
5    { name: "", email: "charlie@example.com" },
6    { name: "David", email: "david@example.com" }
7  ];
8  console.log("Users:", users);
9  console.log("\nValidiere Users:");
10
11 const validUsers = [];
12
13 for (const user of users) {
14   console.log(`\n→ Prüfe User: ${user.name || "(leer)"}`);
15
16   if (!user.name) {
17     console.log(`  ❌ Name fehlt - überspringe`);
18     continue;
19   }
20
```

```
  if (!user.email) {
    console.log(`  ❌ Email fehlt - überspringe`);
    continue;
  }

  console.log(`  ✅ Valid!`);
  validUsers.push(user);
}

console.log("\n✅ Valide Users:", validUsers);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Validierung mit continue ===
Users: [{"name":"Alice","email":"alice@example.com"},
{"name":"Bob","email":""},{"name":"","email":"charlie@example.com"},
{"name":"David","email":"david@example.com"}]

Validiere Users:

→ Prüfe User: Alice
  ✅ Valid!

→ Prüfe User: Bob
  ❌ Email fehlt - überspringe

→ Prüfe User: (leer)
  ❌ Name fehlt - überspringe

→ Prüfe User: David
  ✅ Valid!

✅ Valide Users: [{"name":"Alice","email":"alice@example.com"},
{"name":"David","email":"david@example.com"}]
--- Ende Beispiel 3 ---
```

Praxis: Suche mit break

```
console.log("=== Beispiel 4: Produkt-Suche ===");
const products = [
  { id: 1, name: "Laptop", price: 999 },
  { id: 2, name: "Maus", price: 29 },
  { id: 3, name: "Tastatur", price: 79 }
];
console.log("Produkte:", products);

const searchId = 2;
```

```
const searchId = 2;
console.log(`\nSuche Produkt mit ID ${searchId}:`);

let found = null;

for (const product of products) {
  console.log(`  Prüfe: ${product.name} (ID: ${product.id})`);

  if (product.id === searchId) {
    console.log(`  ✅ Gefunden!`);
    found = product;
    break; // Schleife beenden
  }
}

if (found) {
  console.log(`\nErgebnis:`, found);
} else {
  console.log(`\n❌ Produkt nicht gefunden`);
}
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Produkt-Suche ===
Produkte: [{"id":1,"name":"Laptop","price":999},
{"id":2,"name":"Maus","price":29},
{"id":3,"name":"Tastatur","price":79}]

Suche Produkt mit ID 2:
  Prüfe: Laptop (ID: 1)
  Prüfe: Maus (ID: 2)
  ✅ Gefunden!

Ergebnis: {"id":2,"name":"Maus","price":29}
--- Ende Beispiel 4 ---
```

Kombination: break & continue

```
console.log("=== Beispiel 5: break & continue kombiniert ===");
console.log("Summiere Zahlen, aber:");
console.log("  - Überspringe negative Zahlen");
console.log("  - Stoppe bei Summe > 50\n");

const values = [10, -5, 20, 15, -3, 30, 10];
console.log("Werte:", values);

let sum = 0;

for (const value of values) {
```

```
  console.log(`\n→ Aktueller Wert: ${value}`);

  if (value < 0) {
    console.log(`  ⚠️ Negativ - überspringe (continue)`);
    continue;
  }

  sum += value;
  console.log(`  Addiere: ${value}`);
  console.log(`   → Neue Summe: ${sum}`);

  if (sum > 50) {
    console.log(`  🛑 Summe > 50 erreicht - breche ab (break)`);
    break;
  }
}

console.log(`\n✅ Finale Summe: ${sum}`);
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: break & continue kombiniert ===
Summiere Zahlen, aber:
  - Überspringe negative Zahlen
  - Stoppe bei Summe > 50

Werte: [10,-5,20,15,-3,30,10]

→ Aktueller Wert: 10
  Addiere: 10
  → Neue Summe: 10

→ Aktueller Wert: -5
  ⚠️ Negativ - überspringe (continue)

→ Aktueller Wert: 20
  Addiere: 20
  → Neue Summe: 30

→ Aktueller Wert: 15
  Addiere: 15
  → Neue Summe: 45

→ Aktueller Wert: -3
  ⚠️ Negativ - überspringe (continue)

→ Aktueller Wert: 30
  Addiere: 30
  → Neue Summe: 75
  🛑 Summe > 50 erreicht - breche ab (break)

✅ Finale Summe: 75
--- Ende Beispiel 5 ---
```

2.8 Übung: Kontrollstrukturen

Jetzt sind Sie dran! Diese Übungen testen Ihr Verständnis von if, Schleifen, break und continue. Nutzen Sie console.log ausgiebig, um zu sehen, was passiert.

Aufgabe 1: FizzBuzz (Klassiker!)

```
1  // Schreiben Sie eine for-Schleife von 1 bis 15:
2  // - Bei Zahlen teilbar durch 3: geben Sie "Fizz" aus
3  // - Bei Zahlen teilbar durch 5: geben Sie "Buzz" aus
4  // - Bei Zahlen teilbar durch 3 UND 5: geben Sie "FizzBuzz" aus
```

```
// - Bei Zahlen teilbar durch 3 UND 5: geben Sie "FizzBuzz" aus
// - Sonst: geben Sie die Zahl aus

// Ihr Code hier:
console.log("=== FizzBuzz ===");
```

```
=== FizzBuzz ===
```

Aufgabe 2: Array filtern

```
// Gegeben ist dieses Array:
const numbers = [5, 12, 8, 21, 3, 17, 14, 9];

// Durchlaufen Sie es mit for...of und:
// - Überspringen Sie (continue) alle ungeraden Zahlen
// - Geben Sie gerade Zahlen aus
// - Brechen Sie ab (break), wenn Sie eine Zahl > 15 finden

console.log("=== Filter Array ===");
console.log("Input:", numbers);
// Ihr Code hier:
```

```
=== Filter Array ===
Input: [5,12,8,21,3,17,14,9]
```

Aufgabe 3: Login-Validierung

```
// Schreiben Sie eine Funktion, die Username und Password prüft:
// - Username: min. 3 Zeichen, darf nicht leer sein
// - Password: min. 6 Zeichen, darf nicht leer sein
// Nutzen Sie if/else und console.log für Feedback

console.log("=== Login-Validator ===");
const testUser = "ab";
const testPass = "12345";

console.log("Test:", testUser, testPass);
// Ihr Code hier:
```

```
=== Login-Validator ===
Test: ab 12345
```

Aufgabe 4: Summe bis Schwellwert

```
// Gegeben:
const values = [10, 20, 15, 30, 5, 40];
const threshold = 50;
```

```
const threshold = 50;

// Aufgabe:
// - Summieren Sie die Werte mit einer Schleife
// - Geben Sie bei jedem Schritt die Zwischensumme aus
// - Brechen Sie ab, sobald die Summe >= threshold ist

console.log("=== Summe mit Schwellwert ===");
console.log("Values:", values);
console.log("Threshold:", threshold);
// Ihr Code hier:
```

```
=== Summe mit Schwellwert ===
Values: [10,20,15,30,5,40]
Threshold: 50
```

Aufgabe 5: Status-Mapper

```
// Gegeben sind Bestellstatus-Codes:
const statuses = [1, 2, 3, 4, 99];

// Schreiben Sie eine for...of-Schleife mit switch:
// 1 → "Bestellt"
// 2 → "In Bearbeitung"
// 3 → "Versandt"
// 4 → "Geliefert"
// default → "Unbekannter Status"

console.log("=== Status-Mapper ===");
console.log("Codes:", statuses);
// Ihr Code hier:
```

```
=== Status-Mapper ===
Codes: [1,2,3,4,99]
```

Diese Übungen kombinieren alles aus Kapitel zwei. Experimentieren Sie! Wenn etwas nicht funktioniert, lesen Sie die Fehlermeldung und versuchen Sie zu verstehen, was schief ging. Fehler sind Ihre besten Lehrer.

Lösungen:

- ▶ Lösung Aufgabe 1 (FizzBuzz)
- ▶ Lösung Aufgabe 2 (Array filtern)
- ▶ Lösung Aufgabe 3 (Login-Validierung)
- ▶ Lösung Aufgabe 4 (Summe bis Schwellwert)
- ▶ Lösung Aufgabe 5 (Status-Mapper)

Hervorragend! Sie haben jetzt ein solides Verständnis von Kontrollstrukturen. if-else für Entscheidungen, for und for-of für Schleifen, break und continue für Steuerung. Das sind die Werkzeuge, mit denen Sie komplexe Logik bauen. Bereit für Kapitel drei: Funktionen?

Kapitel 3: Funktionen – Wiederverwendbarer Code

Funktionen sind das Herzstück jeder Programmiersprache. Sie kapseln Logik und machen Code wiederverwendbar. Stellen Sie sich vor, Sie müssen dieselbe Berechnung an zehn Stellen durchführen – mit Funktionen schreiben Sie den Code nur einmal und rufen ihn auf. In Datenbank-Anwendungen nutzen Sie Funktionen ständig: Daten laden, validieren, transformieren, speichern. Alles sind Funktionen. Lernen wir die verschiedenen Arten kennen.

Was sind Funktionen?

- **Wiederverwendbar**: Code einmal schreiben, oft aufrufen
- **Parameter**: Funktionen akzeptieren Eingaben
- **Return**: Funktionen geben Ergebnisse zurück
- **Drei Schreibweisen**: Declaration, Expression, Arrow Function

3.1 Function Declaration

Die klassische Funktionsdeklaration. Sie beginnt mit dem Keyword function, gefolgt vom Namen, Parametern in Klammern und dem Funktionskörper in geschweiften Klammern. Diese Funktionen werden „gehoisted“, das heißt, Sie können sie aufrufen, bevor sie definiert sind.

Syntax: `function name(parameter) { ... }`

```
1  console.log("=== Beispiel 1: Einfache Funktion ===");
2
3  function greet() {
4    console.log("  → Hallo aus der Funktion!");
5  }
6
7  console.log("Vor dem Aufruf");
8  greet(); // Funktion aufrufen
9  console.log("Nach dem Aufruf");
10 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Einfache Funktion ===
Vor dem Aufruf
  → Hallo aus der Funktion!
Nach dem Aufruf
--- Ende Beispiel 1 ---
```

Mit Parameter:

```
1  console.log("=== Beispiel 2: Funktion mit Parameter ===");
2
```

```
function greetUser(name) {
  console.log(`   → Hallo, ${name}!`);
}

console.log("Rufe Funktion mit verschiedenen Namen auf:");
greetUser("Alice");
greetUser("Bob");
greetUser("Charlie");
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Funktion mit Parameter ===
Rufe Funktion mit verschiedenen Namen auf:
   → Hallo, Alice!
   → Hallo, Bob!
   → Hallo, Charlie!
--- Ende Beispiel 2 ---
```

Mit mehreren Parametern:

```
console.log("=== Beispiel 3: Mehrere Parameter ===");

function calculatePrice(basePrice, taxRate) {
  console.log(`  Input: Basispreis=${basePrice}, Steuer=${taxRate}`);
  const totalPrice = basePrice + (basePrice * taxRate);
  console.log(`  Berechnung: ${basePrice} + (${basePrice} × ${taxRate
    ${totalPrice}`);
  console.log(`  Ergebnis: ${totalPrice.toFixed(2)}€`);
}

console.log("\nBerechne Preise:");
calculatePrice(100, 0.19);
console.log();
calculatePrice(50, 0.07);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Mehrere Parameter ===

Berechne Preise:
  Input: Basispreis=100, Steuer=0.19
  Berechnung: 100 + (100 × 0.19) = 119
  Ergebnis: 119.00€

  Input: Basispreis=50, Steuer=0.07
  Berechnung: 50 + (50 × 0.07) = 53.5
  Ergebnis: 53.50€
--- Ende Beispiel 3 ---
```

Hoisting demonstriert:

```javascript
console.log("=== Beispiel 4: Hoisting ===");

console.log("Rufe Funktion auf, BEVOR sie definiert ist:");
sayHello(); // ✅ Funktioniert wegen Hoisting!

function sayHello() {
  console.log("  → Hello from hoisted function!");
}

console.log("Jetzt ist die Funktion definiert");
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Hoisting ===
Rufe Funktion auf, BEVOR sie definiert ist:
  → Hello from hoisted function!
Jetzt ist die Funktion definiert
--- Ende Beispiel 4 ---
```

Praxis: Datenbank-Helper

```javascript
console.log("=== Beispiel 5: DB-Helper ===");

function logQuery(operation, table, id) {
  const timestamp = new Date().toISOString();
  console.log(`[${timestamp}]`);
  console.log(`  Operation: ${operation}`);
  console.log(`  Table: ${table}`);
  console.log(`  ID: ${id || "N/A"}`);
}

console.log("Simuliere DB-Operationen:");
logQuery("SELECT", "users", 123);
```

```
logQuery("SELECT", "users", 123);
console.log();
logQuery("INSERT", "products", null);
console.log();
logQuery("DELETE", "orders", 456);
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: DB-Helper ===
Simuliere DB-Operationen:
[2026-02-13T10:54:35.446Z]
  Operation: SELECT
  Table: users
  ID: 123

[2026-02-13T10:54:35.446Z]
  Operation: INSERT
  Table: products
  ID: N/A

[2026-02-13T10:54:35.447Z]
  Operation: DELETE
  Table: orders
  ID: 456
--- Ende Beispiel 5 ---
```

3.2 Function Expression

Bei einer Function Expression weisen Sie eine Funktion einer Variable zu. Der Unterschied: Diese Funktionen werden NICHT gehoisted, Sie können sie also erst nach der Definition aufrufen. Oft sehen Sie diese Form bei Callbacks oder wenn Funktionen dynamisch zugewiesen werden.

Syntax: `const name = function(parameter) { ... };`

```
console.log("=== Beispiel 1: Function Expression Basics ===");

const sayGoodbye = function() {
  console.log("  → Auf Wiedersehen!");
};

console.log("Funktion definiert");
sayGoodbye(); // Jetzt aufrufen
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Function Expression Basics ===
Funktion definiert
  → Auf Wiedersehen!
--- Ende Beispiel 1 ---
```

Kein Hoisting:

```
1  console.log("=== Beispiel 2: Kein Hoisting ===");
2
3  console.log("Versuche Funktion vor Definition aufzurufen:");
4
5  try {
6    testFunction(); // ❌ Fehler!
7  } catch (error) {
8    console.log("  ❌ Fehler:", error.message);
9  }
10
11 const testFunction = function() {
12   console.log("  → Diese Nachricht sehen Sie nicht");
13 };
14
15 console.log("\nJetzt ist die Funktion definiert:");
16 testFunction(); // ✅ Jetzt funktioniert es
17 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Kein Hoisting ===
Versuche Funktion vor Definition aufzurufen:
  ❌ Fehler: Cannot access 'testFunction' before initialization

Jetzt ist die Funktion definiert:
  → Diese Nachricht sehen Sie nicht
--- Ende Beispiel 2 ---
```

Mit Parametern:

```
1  console.log("=== Beispiel 3: Mit Parametern ===");
2
3  const multiply = function(a, b) {
4    console.log(`  Multipliziere: ${a} × ${b}`);
5    const result = a * b;
6    console.log(`  Ergebnis: ${result}`);
7    return result;
8  };
9
10 console.log("Berechne Produkte:");
11 const r1 = multiply(5, 3);
```

```
console.log( Rückgabewert: ${r1} );
console.log();
const r2 = multiply(10, 7);
console.log(`Rückgabewert: ${r2}`);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Mit Parametern ===
Berechne Produkte:
  Multipliziere: 5 × 3
  Ergebnis: 15
Rückgabewert: 15

  Multipliziere: 10 × 7
  Ergebnis: 70
Rückgabewert: 70
--- Ende Beispiel 3 ---
```

Anonyme Funktionen (häufig bei Callbacks):

```
console.log("=== Beispiel 4: Anonyme Funktion als Callback ===");

const numbers = [1, 2, 3, 4, 5];
console.log("Array:", numbers);
console.log("\nVerdopple jede Zahl:");

const doubled = numbers.map(function(num) {
  console.log(`  ${num} → ${num * 2}`);
  return num * 2;
});

console.log("\nErgebnis:", doubled);
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Anonyme Funktion als Callback ===
Array: [1,2,3,4,5]

Verdopple jede Zahl:
  1 → 2
  2 → 4
  3 → 6
  4 → 8
  5 → 10

Ergebnis: [2,4,6,8,10]
--- Ende Beispiel 4 ---
```

Praxis: Validator-Funktion

```
console.log("=== Beispiel 5: Email-Validator ===");

const validateEmail = function(email) {
  console.log(`\nValidiere: "${email}"`);

  if (!email) {
    console.log("  ❌ Email fehlt");
    return false;
  }

  if (!email.includes("@")) {
    console.log("  ❌ Kein @ gefunden");
    return false;
  }

  if (!email.includes(".")) {
    console.log("  ❌ Kein Punkt gefunden");
    return false;
  }

  console.log("  ✅ Email valide");
  return true;
};

const testEmails = ["user@example.com", "invalid", "test@", "@test.co
console.log("Test-Emails:", testEmails);

for (const email of testEmails) {
  const isValid = validateEmail(email);
  console.log(`   → Return: ${isValid}`);
}
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Email-Validator ===
Test-Emails: ["user@example.com","invalid","test@","@test.com"]

Validiere: "user@example.com"
  ✅ Email valide
  → Return: true

Validiere: "invalid"
  ❌ Kein @ gefunden
  → Return: false

Validiere: "test@"
  ❌ Kein Punkt gefunden
  → Return: false

Validiere: "@test.com"
  ✅ Email valide
  → Return: true
--- Ende Beispiel 5 ---
```

3.3 Arrow Functions (⇒)

Arrow Functions sind die moderne, kompakte Schreibweise seit ES6. Statt function schreiben Sie einen Pfeil. Sie sind kürzer und haben ein spezielles Verhalten bei „this“ – dazu später mehr. In der Praxis sehen Sie Arrow Functions überall, besonders bei Array-Methoden und Callbacks.

Syntax: `(parameter) => { ... }` **oder** `parameter => ...`

```
1  console.log("=== Beispiel 1: Arrow Function Basics ===");
2
3  const greet = () => {
4    console.log("  → Hallo aus Arrow Function!");
5  };
6
7  console.log("Rufe Arrow Function auf:");
8  greet();
9  console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Arrow Function Basics ===
Rufe Arrow Function auf:
  → Hallo aus Arrow Function!
--- Ende Beispiel 1 ---
```

Mit einem Parameter (Klammern optional):

```
console.log("=== Beispiel 2: Ein Parameter ===");

// Mit Klammern
const double1 = (x) => {
  console.log(`  Input: ${x}`);
  return x * 2;
};

// Ohne Klammern (bei genau 1 Parameter erlaubt)
const double2 = x => {
  console.log(`  Input: ${x}`);
  return x * 2;
};

console.log("Mit Klammern:");
console.log("Ergebnis:", double1(5));
console.log("\nOhne Klammern:");
console.log("Ergebnis:", double2(5));
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Ein Parameter ===
Mit Klammern:
  Input: 5
Ergebnis: 10

Ohne Klammern:
  Input: 5
Ergebnis: 10
--- Ende Beispiel 2 ---
```

Implizites Return (ohne Klammern):

```
console.log("=== Beispiel 3: Implizites Return ===");

// Mit explizitem return
const add1 = (a, b) => {
  return a + b;
};

// Ohne Klammern = implizites return (kürzer!)
const add2 = (a, b) => a + b;

console.log("Mit explizitem return:");
console.log("  5 + 3 =", add1(5, 3));
console.log("\nMit implizitem return:");
console.log("  5 + 3 =", add2(5, 3));
```

```
15 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Implizites Return ===
Mit explizitem return:
  5 + 3 = 8

Mit implizitem return:
  5 + 3 = 8
--- Ende Beispiel 3 ---
```

Vergleich: Alle drei Schreibweisen

```
1  console.log("=== Beispiel 4: Vergleich aller Schreibweisen ===");
2
3  // 1. Function Declaration
4  function square1(x) {
5    return x * x;
6  }
7
8  // 2. Function Expression
9  const square2 = function(x) {
10   return x * x;
11 };
12
13 // 3. Arrow Function
14 const square3 = x => x * x;
15
16 console.log("Berechne 7²:");
17 console.log("  Declaration:", square1(7));
18 console.log("  Expression:", square2(7));
19 console.log("  Arrow:", square3(7));
20 console.log("Alle liefern dasselbe Ergebnis!");
21 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Vergleich aller Schreibweisen ===
Berechne 7²:
  Declaration: 49
  Expression: 49
  Arrow: 49
Alle liefern dasselbe Ergebnis!
--- Ende Beispiel 4 ---
```

Praxis: Array-Methoden mit Arrow Functions

```
1  console.log("=== Beispiel 5: Array-Methoden ===");
2
```

```
const products = [
  { name: "Laptop", price: 999 },
  { name: "Maus", price: 29 },
  { name: "Tastatur", price: 79 }
];

console.log("Produkte:", products);

// filter: Nur Produkte > 50€
console.log("\nFilter (Preis > 50€):");
const expensive = products.filter(p => {
  console.log(`  Prüfe: ${p.name} (${p.price}€)`);
  return p.price > 50;
});
console.log("Ergebnis:", expensive);

// map: Nur Namen extrahieren
console.log("\nMap (Namen):");
const names = products.map(p => {
  console.log(`  Extrahiere: ${p.name}`);
  return p.name;
});
console.log("Ergebnis:", names);

// forEach: Ausgabe
console.log("\nforEach (Ausgabe):");
products.forEach(p => {
  console.log(`  → ${p.name}: ${p.price}€`);
});

console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Array-Methoden ===
Produkte: [{"name":"Laptop","price":999},{"name":"Maus","price":29},
{"name":"Tastatur","price":79}]

Filter (Preis > 50€):
  Prüfe: Laptop (999€)
  Prüfe: Maus (29€)
  Prüfe: Tastatur (79€)
Ergebnis: [{"name":"Laptop","price":999},
{"name":"Tastatur","price":79}]

Map (Namen):
  Extrahiere: Laptop
  Extrahiere: Maus
  Extrahiere: Tastatur
Ergebnis: ["Laptop","Maus","Tastatur"]

forEach (Ausgabe):
  → Laptop: 999€
  → Maus: 29€
  → Tastatur: 79€
--- Ende Beispiel 5 ---
```

Mehrzeilige Arrow Functions:

```
1  console.log("=== Beispiel 6: Mehrzeilig ===");
2
3  const processOrder = (orderId, amount) => {
4    console.log(`\n→ Verarbeite Bestellung #${orderId}`);
5    console.log(`  Betrag: ${amount}€`);
6
7    if (amount > 100) {
8      console.log("  🎁 Gratisversand!");
9    }
10
11   const tax = amount * 0.19;
12   console.log(`  Steuer: ${tax.toFixed(2)}€`);
13
14   const total = amount + tax;
15   console.log(`  Gesamt: ${total.toFixed(2)}€`);
16
17   return total;
18 };
19
20 console.log("Verarbeite Bestellungen:");
21 processOrder(101, 150);
```

```
processOrder(102, 50);
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Mehrzeilig ===
Verarbeite Bestellungen:

→ Verarbeite Bestellung #101
  Betrag: 150€
  🎁 Gratisversand!
  Steuer: 28.50€
  Gesamt: 178.50€

→ Verarbeite Bestellung #102
  Betrag: 50€
  Steuer: 9.50€
  Gesamt: 59.50€
--- Ende Beispiel 6 ---
```

3.4 Parameter & Default Values

Funktionen können Parameter haben – Eingabewerte, die Sie beim Aufruf übergeben. Seit ES6 können Sie Default-Werte definieren, die verwendet werden, wenn kein Argument übergeben wird. Das macht Funktionen flexibler und verhindert undefined-Fehler.

Grundlagen:

```
console.log("=== Beispiel 1: Parameter Basics ===");

const greet = (name, greeting) => {
  console.log(`  ${greeting}, ${name}!`);
};

console.log("Mit allen Parametern:");
greet("Alice", "Hallo");

console.log("\nMit fehlenden Parametern:");
greet("Bob"); // greeting ist undefined
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Parameter Basics ===
Mit allen Parametern:
  Hallo, Alice!

Mit fehlenden Parametern:
  undefined, Bob!
--- Ende Beispiel 1 ---
```

Default Values (ES6):

```
1  console.log("=== Beispiel 2: Default Values ===");
2
3  const greet = (name = "Gast", greeting = "Hallo") => {
4    console.log(`  ${greeting}, ${name}!`);
5  };
6
7  console.log("Mit allen Parametern:");
8  greet("Alice", "Guten Tag");
9
10 console.log("\nNur 1 Parameter:");
11 greet("Bob"); // greeting = "Hallo" (default)
12
13 console.log("\nKeine Parameter:");
14 greet(); // name = "Gast", greeting = "Hallo" (defaults)
15
16 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Default Values ===
Mit allen Parametern:
  Guten Tag, Alice!

Nur 1 Parameter:
  Hallo, Bob!

Keine Parameter:
  Hallo, Gast!
--- Ende Beispiel 2 ---
```

Default Values mit Berechnungen:

```
1  console.log("=== Beispiel 3: Berechnete Defaults ===");
2
3  const createUser = (name, role = "user", createdAt = new Date().toISO
     ()) => {
4    console.log("\nErstelle User:");
```

```
  console.log(   Name: ${name} );
  console.log(`  Role: ${role}`);
  console.log(`  Created: ${createdAt}`);
  return { name, role, createdAt };
};

console.log("Alle Werte explizit:");
createUser("Alice", "admin", "2025-01-01T00:00:00Z");

console.log("\n\nMit Defaults:");
createUser("Bob"); // role und createdAt werden generiert

console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Berechnete Defaults ===
Alle Werte explizit:

Erstelle User:
  Name: Alice
  Role: admin
  Created: 2025-01-01T00:00:00Z


Mit Defaults:

Erstelle User:
  Name: Bob
  Role: user
  Created: 2026-02-13T10:54:36.817Z
--- Ende Beispiel 3 ---
```

Rest Parameter (`...args`):

```
console.log("=== Beispiel 4: Rest Parameter ===");

const sum = (...numbers) => {
  console.log("  Erhaltene Argumente:", numbers);
  console.log("  Typ:", Array.isArray(numbers) ? "Array" : "kein Arra

  let total = 0;
  for (const num of numbers) {
    console.log(`    Addiere: ${num}`);
    total += num;
  }

  console.log(`  Summe: ${total}`);
  return total;
```

```
};

console.log("Mit 3 Zahlen:");
sum(10, 20, 30);

console.log("\nMit 5 Zahlen:");
sum(1, 2, 3, 4, 5);

console.log("\nMit 1 Zahl:");
sum(100);
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Rest Parameter ===
Mit 3 Zahlen:
  Erhaltene Argumente: [10,20,30]
  Typ: Array
    Addiere: 10
    Addiere: 20
    Addiere: 30
  Summe: 60

Mit 5 Zahlen:
  Erhaltene Argumente: [1,2,3,4,5]
  Typ: Array
    Addiere: 1
    Addiere: 2
    Addiere: 3
    Addiere: 4
    Addiere: 5
  Summe: 15

Mit 1 Zahl:
  Erhaltene Argumente: [100]
  Typ: Array
    Addiere: 100
  Summe: 100
--- Ende Beispiel 4 ---
```

Praxis: Flexible Query-Builder

```
console.log("=== Beispiel 5: Query Builder ===");

const buildQuery = (table, options = {}) => {
  console.log(`\n→ Baue Query für Tabelle: ${table}`);
  console.log("  Options:", options);

```

```
  const limit = options.limit || 10;
  const offset = options.offset || 0;
  const orderBy = options.orderBy || "id";

  const query = `SELECT * FROM ${table} ORDER BY ${orderBy} LIMIT ${l
    OFFSET ${offset}`;
  console.log(`  ✅ Query: ${query}`);
  return query;
};

console.log("Ohne Options:");
buildQuery("users");

console.log("\n\nMit Options:");
buildQuery("products", { limit: 20, orderBy: "price" });

console.log("\n\nMit teilweisen Options:");
buildQuery("orders", { offset: 100 });
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Query Builder ===
Ohne Options:

→ Baue Query für Tabelle: users
  Options: {}
  ✅ Query: SELECT * FROM users ORDER BY id LIMIT 10 OFFSET 0


Mit Options:

→ Baue Query für Tabelle: products
  Options: {"limit":20,"orderBy":"price"}
  ✅ Query: SELECT * FROM products ORDER BY price LIMIT 20 OFFSET 0


Mit teilweisen Options:

→ Baue Query für Tabelle: orders
  Options: {"offset":100}
  ✅ Query: SELECT * FROM orders ORDER BY id LIMIT 10 OFFSET 100
--- Ende Beispiel 5 ---
```

Destructuring in Parametern:

```
console.log("=== Beispiel 6: Destructuring ===");
```

```
// Statt: function(user) { user.name, user.email }
// Schreiben: function({ name, email })

const displayUser = ({ name, email, role = "user" }) => {
  console.log("\n→ User-Info:");
  console.log(`  Name: ${name}`);
  console.log(`  Email: ${email}`);
  console.log(`  Role: ${role}`);
};

const user1 = { name: "Alice", email: "alice@example.com", role: "adm
const user2 = { name: "Bob", email: "bob@example.com" };

console.log("User 1:");
displayUser(user1);

console.log("\nUser 2 (ohne role):");
displayUser(user2); // role = "user" (default)
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Destructuring ===
User 1:

→ User-Info:
  Name: Alice
  Email: alice@example.com
  Role: admin

User 2 (ohne role):

→ User-Info:
  Name: Bob
  Email: bob@example.com
  Role: user
--- Ende Beispiel 6 ---
```

3.5 Return Values

Funktionen können Werte zurückgeben mit dem return-Keyword. Das Ergebnis können Sie in einer Variable speichern oder direkt weiterverwenden. Ohne return gibt eine Funktion undefined zurück. Return stoppt die Funktion sofort – Code danach wird nicht mehr ausgeführt.

Einfaches Return:

```
console.log("=== Beispiel 1: Return Basics ===");

```

```
const add = (a, b) => {
  console.log(`  Berechne: ${a} + ${b}`);
  const result = a + b;
  console.log(`  Ergebnis: ${result}`);
  return result;
};

console.log("Rufe Funktion auf:");
const sum = add(5, 3);
console.log("Rückgabewert:", sum);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Return Basics ===
Rufe Funktion auf:
  Berechne: 5 + 3
  Ergebnis: 8
Rückgabewert: 8
--- Ende Beispiel 1 ---
```

Ohne Return = undefined:

```
console.log("=== Beispiel 2: Ohne Return ===");

const logMessage = (msg) => {
  console.log(`  → ${msg}`);
  // Kein return!
};

console.log("Rufe Funktion auf:");
const result = logMessage("Hallo");
console.log("Rückgabewert:", result); // undefined
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Ohne Return ===
Rufe Funktion auf:
  → Hallo
Rückgabewert: undefined
--- Ende Beispiel 2 ---
```

Frühes Return (Guard Clauses):

```
console.log("=== Beispiel 3: Frühes Return ===");

const divide = (a, b) => {
  console.log(`\nTeile: ${a} ÷ ${b}`);

```

```
  if (b === 0) {
    console.log("  ❌ Division durch Null!");
    return null; // Früher Abbruch
  }

  // Dieser Code wird nur ausgeführt, wenn b !== 0
  const result = a / b;
  console.log(`  ✅ Ergebnis: ${result}`);
  return result;
};

console.log("Test 1:");
const r1 = divide(10, 2);
console.log("Return:", r1);

console.log("\nTest 2 (Division durch 0):");
const r2 = divide(10, 0);
console.log("Return:", r2);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Frühes Return ===
Test 1:

Teile: 10 ÷ 2
  ✅ Ergebnis: 5
Return: 5

Test 2 (Division durch 0):

Teile: 10 ÷ 0
  ❌ Division durch Null!
Return: null
--- Ende Beispiel 3 ---
```

Return-Objekte:

```
console.log("=== Beispiel 4: Objekte zurückgeben ===");

const calculateStats = (numbers) => {
  console.log("\nBerechne Statistiken für:", numbers);

  const sum = numbers.reduce((acc, num) => acc + num, 0);
  console.log(`  Summe: ${sum}`);

  const average = sum / numbers.length;
  console.log(`  Durchschnitt: ${average.toFixed(2)}`);

  const min = Math.min(...numbers);
```

```
  const min = Math.min(...numbers);
  const max = Math.max(...numbers);
  console.log(`  Min: ${min}, Max: ${max}`);

  return { sum, average, min, max }; // Objekt zurückgeben
};

const data = [10, 20, 30, 40, 50];
const stats = calculateStats(data);
console.log("\nRückgabe-Objekt:", stats);
console.log("Zugriff auf average:", stats.average);
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Objekte zurückgeben ===

Berechne Statistiken für: [10,20,30,40,50]
  Summe: 150
  Durchschnitt: 30.00
  Min: 10, Max: 50

Rückgabe-Objekt: {"sum":150,"average":30,"min":10,"max":50}
Zugriff auf average: 30
--- Ende Beispiel 4 ---
```

Praxis: Validierung mit Boolean-Return

```
console.log("=== Beispiel 5: Boolean-Return ===");

const isValidProduct = (product) => {
  console.log(`\nValidiere Produkt:`, product);

  if (!product.name) {
    console.log("  ❌ Name fehlt");
    return false;
  }

  if (!product.price || product.price <= 0) {
    console.log("  ❌ Ungültiger Preis");
    return false;
  }

  if (product.stock < 0) {
    console.log("  ❌ Negativer Lagerbestand");
    return false;
  }

  console.log("  ✅ Produkt valide");
  return true;
};
```

```
};

const products = [
  { name: "Laptop", price: 999, stock: 5 },
  { name: "", price: 50, stock: 10 },
  { name: "Maus", price: -10, stock: 20 },
  { name: "Tastatur", price: 79, stock: 0 }
];

console.log("Validiere Produkte:");
products.forEach((product, index) => {
  console.log(`\n→ Produkt ${index + 1}:`);
  const valid = isValidProduct(product);
  console.log(`  Return: ${valid}`);
});
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Boolean-Return ===
Validiere Produkte:

→ Produkt 1:

Validiere Produkt: {"name":"Laptop","price":999,"stock":5}
  ✅ Produkt valide
  Return: true

→ Produkt 2:

Validiere Produkt: {"name":"","price":50,"stock":10}
  ❌ Name fehlt
  Return: false

→ Produkt 3:

Validiere Produkt: {"name":"Maus","price":-10,"stock":20}
  ❌ Ungültiger Preis
  Return: false

→ Produkt 4:

Validiere Produkt: {"name":"Tastatur","price":79,"stock":0}
  ✅ Produkt valide
  Return: true
--- Ende Beispiel 5 ---
```

Multiple Returns (verschiedene Ergebnisse):

```
console.log("=== Beispiel 6: Multiple Returns ===");

const getUserStatus = (user) => {
  console.log(`\nPrüfe User:`, user);

  if (!user) {
    console.log("  → User ist null/undefined");
    return "unknown";
  }

  if (!user.active) {
    console.log("  → User ist inaktiv");
    return "inactive";
  }

  if (user.role === "admin") {
    console.log("  → User ist Admin");
    return "admin";
  }

  console.log("  → Standard-User");
  return "user";
};

const users = [
  { name: "Alice", active: true, role: "admin" },
  { name: "Bob", active: false, role: "user" },
  { name: "Charlie", active: true, role: "user" },
  null
];

users.forEach((user, index) => {
  const status = getUserStatus(user);
  console.log(`  ✅ Status: "${status}"\n`);
});
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Multiple Returns ===

Prüfe User: {"name":"Alice","active":true,"role":"admin"}
  → User ist Admin
  ✅ Status: "admin"


Prüfe User: {"name":"Bob","active":false,"role":"user"}
  → User ist inaktiv
  ✅ Status: "inactive"


Prüfe User: {"name":"Charlie","active":true,"role":"user"}
  → Standard-User
  ✅ Status: "user"


Prüfe User: null
  → User ist null/undefined
  ✅ Status: "unknown"

--- Ende Beispiel 6 ---
```

3.6 Scope & Closures (Kurzüberblick)

Scope bestimmt, wo Variablen sichtbar sind. Es gibt Global Scope – überall sichtbar – und Function Scope – nur innerhalb der Funktion. Closures sind ein fortgeschrittenes Konzept: Eine innere Funktion „erinnert“ sich an Variablen aus der äußeren Funktion, selbst wenn die äußere Funktion schon fertig ist. Klingt kompliziert, ist aber sehr mächtig.

Global vs. Function Scope:

```js
console.log("=== Beispiel 1: Scope Basics ===");

const globalVar = "Ich bin global";
console.log("Global Scope:", globalVar);

function testScope() {
  const localVar = "Ich bin lokal";
  console.log("  Inside Function:");
  console.log("    globalVar:", globalVar); // ✅ Zugriff auf global
  console.log("    localVar:", localVar);   // ✅ Zugriff auf local
}

testScope();
```

```
14
15 console.log("\nOutside Function:");
16 console.log("  globalVar:", globalVar); // ✅ Funktioniert
17 // console.log("  localVar:", localVar); // ❌ Error! localVar existi
     hier nicht
18 console.log("  localVar: (nicht zugänglich von außen)");
19 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Scope Basics ===
Global Scope: Ich bin global
  Inside Function:
    globalVar: Ich bin global
    localVar: Ich bin lokal

Outside Function:
  globalVar: Ich bin global
  localVar: (nicht zugänglich von außen)
--- Ende Beispiel 1 ---
```

Block Scope (let/const):

```
1  console.log("=== Beispiel 2: Block Scope ===");
2
3  console.log("Vor dem if-Block:");
4  const x = "außen";
5  console.log("  x =", x);
6
7  if (true) {
8    console.log("\nInside if-Block:");
9    const x = "innen"; // Neue Variable!
10   console.log("  x =", x);
11
12   const y = "nur im Block";
13   console.log("  y =", y);
14 }
15
16 console.log("\nNach dem if-Block:");
17 console.log("  x =", x); // "außen" (äußeres x)
18 // console.log("  y =", y); // ❌ Error! y existiert nur im Block
19 console.log("  y: (nicht zugänglich)");
20 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Block Scope ===
Vor dem if-Block:
  x = außen

Inside if-Block:
  x = innen
  y = nur im Block

Nach dem if-Block:
  x = außen
  y: (nicht zugänglich)
--- Ende Beispiel 2 ---
```

Closure: Innere Funktion erinnert sich:

```javascript
console.log("=== Beispiel 3: Closure ===");

function createCounter() {
  console.log("→ createCounter() wird ausgeführt");
  let count = 0; // Private Variable

  console.log("  Erstelle innere Funktion");

  return function() {
    count++; // Zugriff auf äußere Variable!
    console.log(`    Counter: ${count}`);
    return count;
  };
}

console.log("\nErstelle Counter 1:");
const counter1 = createCounter();
console.log("\nRufe Counter 1 auf:");
counter1(); // 1
counter1(); // 2
counter1(); // 3

console.log("\nErstelle Counter 2:");
const counter2 = createCounter();
console.log("\nRufe Counter 2 auf:");
counter2(); // 1 (eigener count!)
counter2(); // 2

console.log("\nJeder Counter hat seinen eigenen count!");
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Closure ===

Erstelle Counter 1:
→ createCounter() wird ausgeführt
  Erstelle innere Funktion

Rufe Counter 1 auf:
    Counter: 1
    Counter: 2
    Counter: 3

Erstelle Counter 2:
→ createCounter() wird ausgeführt
  Erstelle innere Funktion

Rufe Counter 2 auf:
    Counter: 1
    Counter: 2

Jeder Counter hat seinen eigenen count!
--- Ende Beispiel 3 ---
```

Praxis: Private Variablen simulieren:

```javascript
console.log("=== Beispiel 4: Private Variablen ===");

const createWallet = (initialBalance) => {
  console.log(`\n→ Erstelle Wallet mit ${initialBalance}€`);
  let balance = initialBalance; // Private!

  return {
    deposit: (amount) => {
      console.log(`   → Einzahlung: ${amount}€`);
      balance += amount;
      console.log(`     Neuer Stand: ${balance}€`);
    },

    withdraw: (amount) => {
      console.log(`   → Auszahlung: ${amount}€`);
      if (amount > balance) {
        console.log(`     ❌ Nicht genug Guthaben (${balance}€)`);
        return false;
      }
      balance -= amount;
      console.log(`     Neuer Stand: ${balance}€`);
      return true;
```

```
    },

    getBalance: () => {
      console.log(`   → Kontostand: ${balance}€`);
      return balance;
    }
  };
};

console.log("Erstelle Wallet:");
const myWallet = createWallet(100);

console.log("\nOperationen:");
myWallet.deposit(50);
myWallet.withdraw(30);
myWallet.withdraw(200); // Fehlschlag
myWallet.getBalance();

console.log("\n⚠️ balance ist nicht direkt zugänglich:");
console.log("myWallet.balance =", myWallet.balance); // undefined
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Private Variablen ===
Erstelle Wallet:

→ Erstelle Wallet mit 100€

Operationen:
  → Einzahlung: 50€
    Neuer Stand: 150€
  → Auszahlung: 30€
    Neuer Stand: 120€
  → Auszahlung: 200€
    ❌ Nicht genug Guthaben (120€)
  → Kontostand: 120€

⚠️ balance ist nicht direkt zugänglich:
myWallet.balance = undefined
--- Ende Beispiel 4 ---
```

Closure in Schleifen (häufiger Fehler!):

```
console.log("=== Beispiel 5: Closure in Schleifen ===");

console.log("❌ Falsch mit var:");
const functions1 = [];
for (var i = 0; i < 3; i++) {
```

```
  functions1.push(function() {
    console.log(`  Wert: ${i}`);
  });
}
console.log("Rufe Funktionen auf:");
functions1[0](); // 3 (nicht 0!)
functions1[1](); // 3 (nicht 1!)
functions1[2](); // 3 (nicht 2!)
console.log("Alle zeigen 3, weil var function-scoped ist\n");

console.log("✅ Richtig mit let:");
const functions2 = [];
for (let j = 0; j < 3; j++) {
  functions2.push(function() {
    console.log(`  Wert: ${j}`);
  });
}
console.log("Rufe Funktionen auf:");
functions2[0](); // 0 ✅
functions2[1](); // 1 ✅
functions2[2](); // 2 ✅
console.log("let erzeugt für jede Iteration eigenen Scope");
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Closure in Schleifen ===
❌ Falsch mit var:
Rufe Funktionen auf:
  Wert: 3
  Wert: 3
  Wert: 3
Alle zeigen 3, weil var function-scoped ist

✅ Richtig mit let:
Rufe Funktionen auf:
  Wert: 0
  Wert: 1
  Wert: 2
let erzeugt für jede Iteration eigenen Scope
--- Ende Beispiel 5 ---
```

3.7 Übung: Funktionen

Zeit, Ihr Wissen zu testen! Diese Übungen kombinieren alle Funktionstypen, Parameter, Return-Values und sogar Closures. Nutzen Sie console.log, um Ihre Lösungen zu debuggen.

Aufgabe 1: Temperatur-Konverter

```
// Schreiben Sie eine Arrow Function, die Celsius in Fahrenheit umrec
// Formel: F = C × 9/5 + 32
// Geben Sie Eingabe und Ergebnis mit console.log aus

console.log("=== Temperatur-Konverter ===");
// Ihr Code hier:
// const celsiusToFahrenheit = ...

// Test:
// celsiusToFahrenheit(0);    // 32°F
// celsiusToFahrenheit(100);  // 212°F
// celsiusToFahrenheit(37);   // 98.6°F
```

```
=== Temperatur-Konverter ===
```

Aufgabe 2: Array-Statistiken

```
// Schreiben Sie eine Funktion, die ein Array von Zahlen nimmt und
// ein Objekt mit {min, max, sum, average} zurückgibt
// Nutzen Sie console.log für Zwischenschritte

console.log("=== Array-Statistiken ===");
// Ihr Code hier:
// const getStats = (numbers) => { ... }

// Test:
// const testData = [10, 5, 20, 15, 30];
// const stats = getStats(testData);
// console.log("Ergebnis:", stats);
```

```
=== Array-Statistiken ===
```

Aufgabe 3: Produkt-Filter mit Default Values

```
// Schreiben Sie eine Funktion filterProducts(products, minPrice = 0,
  maxPrice = Infinity)
// Sie soll nur Produkte im Preisbereich zurückgeben
// Nutzen Sie filter() und Arrow Functions

console.log("=== Produkt-Filter ===");
const products = [
  { name: "Laptop", price: 999 },
  { name: "Maus", price: 29 },
  { name: "Tastatur", price: 79 },
  { name: "Monitor", price: 299 }
];

// Ihr Code hier:
```

```
// Ihr Code hier:
// const filterProducts = ...

// Tests:
// filterProducts(products);              // Alle
// filterProducts(products, 50);          // Preis >= 50
// filterProducts(products, 50, 300);     // 50 <= Preis <= 300
```

```
=== Produkt-Filter ===
```

Aufgabe 4: Countdown mit Closure

```
// Schreiben Sie eine Funktion createCountdown(start),
// die eine Funktion zurückgibt, welche bei jedem Aufruf
// den Zähler um 1 reduziert und ausgibt
// Bei 0 soll "🚀 Start!" ausgegeben werden

console.log("=== Countdown ===");
// Ihr Code hier:
// const createCountdown = (start) => { ... }

// Test:
// const countdown = createCountdown(5);
// countdown(); // 5
// countdown(); // 4
// countdown(); // 3
// countdown(); // 2
// countdown(); // 1
// countdown(); // 🚀 Start!
```

```
=== Countdown ===
```

Aufgabe 5: User-Validator mit Guard Clauses

```
// Schreiben Sie eine Funktion validateUser(user),
// die true/false zurückgibt
// Prüfungen:
// - user.name muss existieren (min. 2 Zeichen)
// - user.email muss @ enthalten
// - user.age muss >= 18 sein
// Nutzen Sie frühe returns (Guard Clauses)

console.log("=== User-Validator ===");
// Ihr Code hier:
// const validateUser = (user) => { ... }

// Tests:
const users = [
```

```
  { name: "Alice", email: "alice@test.com", age: 25 },
  { name: "B", email: "bob@test.com", age: 20 },
  { name: "Charlie", email: "charlie.com", age: 30 },
  { name: "David", email: "david@test.com", age: 16 }
];

// users.forEach(user => {
//   console.log(`\n${user.name}:`, validateUser(user) ? "✅" : "❌")
// });
```

```
=== User-Validator ===
```

Diese Übungen fordern Sie heraus! Wenn Sie nicht weiterkommen, schauen Sie sich die Beispiele aus den vorherigen Abschnitten an. Funktionen sind wie Werkzeuge – je mehr Sie üben, desto geschickter werden Sie. Closure ist besonders knifflig, lassen Sie sich nicht entmutigen!

Lösungen:

- ▶ Lösung Aufgabe 1 (Temperatur-Konverter)
- ▶ Lösung Aufgabe 2 (Array-Statistiken)
- ▶ Lösung Aufgabe 3 (Produkt-Filter)
- ▶ Lösung Aufgabe 4 (Countdown mit Closure)
- ▶ Lösung Aufgabe 5 (User-Validator)

Fantastisch! Sie beherrschen jetzt Funktionen – von einfachen Deklarationen über Arrow Functions bis zu fortgeschrittenen Closures. Funktionen sind Ihre Bausteine für wiederverwendbaren, wartbaren Code. Bereit für das Wichtigste: Objekte und Arrays in Kapitel vier?

Kapitel 4: Objekte & Arrays – Strukturierte Daten

Objekte und Arrays sind die fundamentalen Datenstrukturen in JavaScript – und essentiell für Datenbank-Operationen. Datenbanken speichern Dokumente als Objekte, Query-Ergebnisse sind Arrays von Objekten. Fast jede DB-Operation arbeitet mit diesen Strukturen. Dieses Kapitel ist DAS WICHTIGSTE für Ihre Datenbank-Arbeit. Nehmen Sie sich Zeit, die Beispiele zu verstehen!

Warum sind Objekte & Arrays so wichtig?

- **Datenbanken = Objekte**: Jeder Datensatz ist ein Objekt `{ id: 1, name: "Alice" }`
- **Query-Results = Arrays**: `[{user1}, {user2}, {user3}]`
- **Array-Methoden**: `map`, `filter`, `find` sind Ihre täglichen Werkzeuge
- **JSON**: Das universelle Datenformat basiert auf Objekten & Arrays

---

4.1 Objekte: Grundlagen

Objekte sind Sammlungen von Key-Value-Paaren. Denken Sie an eine Tabellzeile in einer Datenbank: Jede Spalte ist ein Property. Objekte werden mit geschweiften Klammern erstellt, Properties mit Doppelpunkt zugewiesen. Sie sind DIE Datenstruktur für strukturierte Informationen.

Syntax: `{ key: value }`

```
console.log("=== Beispiel 1: Objekt erstellen ===");

const user = {
  id: 1,
  name: "Alice",
  email: "alice@example.com",
  age: 28,
  active: true
};

console.log("User-Objekt:", user);
console.log("Typ:", typeof user);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Objekt erstellen ===
User-Objekt:
{"id":1,"name":"Alice","email":"alice@example.com","age":28,"active":true}
Typ: object
--- Ende Beispiel 1 ---
```

Verschachtelte Objekte:

```
console.log("=== Beispiel 2: Verschachtelt ===");

const product = {
  id: 101,
  name: "Laptop",
  price: 999,
  specs: {
    cpu: "Intel i7",
    ram: "16GB",
    storage: "512GB SSD"
  },
  tags: ["electronics", "computers", "new"]
};

console.log("Produkt:", product);
console.log("\nNested Object:");
console.log("  specs:", product.specs);
console.log("  CPU:", product.specs.cpu);
console.log("\nNested Array:");
console.log("  tags:", product.tags);
```

```
21 console.log("  Erster Tag:", product.tags[0]);
22 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Verschachtelt ===
Produkt: {"id":101,"name":"Laptop","price":999,"specs":{"cpu":"Intel
i7","ram":"16GB","storage":"512GB SSD"},"tags":
["electronics","computers","new"]}

Nested Object:
  specs: {"cpu":"Intel i7","ram":"16GB","storage":"512GB SSD"}
  CPU: Intel i7

Nested Array:
  tags: ["electronics","computers","new"]
  Erster Tag: electronics
--- Ende Beispiel 2 ---
```

Objekte dynamisch erstellen:

```
1  console.log("=== Beispiel 3: Dynamisch erstellen ===");
2
3  const createUser = (id, name, role) => {
4    console.log(`\nErstelle User: ${name}`);
5    return {
6      id: id,
7      name: name,
8      role: role,
9      createdAt: new Date().toISOString()
10   };
11 };
12
13 const user1 = createUser(1, "Bob", "admin");
14 console.log("User 1:", user1);
15
16 const user2 = createUser(2, "Charlie", "user");
17 console.log("\nUser 2:", user2);
18 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Dynamisch erstellen ===

Erstelle User: Bob
User 1: {"id":1,"name":"Bob","role":"admin","createdAt":"2026-02-
13T10:54:42.469Z"}

Erstelle User: Charlie

User 2: {"id":2,"name":"Charlie","role":"user","createdAt":"2026-02-
13T10:54:42.469Z"}
--- Ende Beispiel 3 ---
```

Shorthand Property Names (ES6):

```javascript
console.log("=== Beispiel 4: Shorthand Syntax ===");

const id = 123;
const name = "David";
const active = true;

// Alt: { id: id, name: name, active: active }
// Neu (wenn Variablenname = Key):
const user = { id, name, active };

console.log("User:", user);
console.log("\nDies ist identisch mit:");
console.log("{ id: id, name: name, active: active }");
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Shorthand Syntax ===
User: {"id":123,"name":"David","active":true}

Dies ist identisch mit:
{ id: id, name: name, active: active }
--- Ende Beispiel 4 ---
```

Praxis: DB-Dokument simuliert:

```javascript
console.log("=== Beispiel 5: Datenbank-Dokument ===");

const dbDocument = {
  _id: "507f1f77bcf86cd799439011",
  collection: "orders",
  data: {
    orderId: 1001,
```

```
    customer: {
      name: "Alice",
      email: "alice@example.com"
    },
    items: [
      { productId: 1, name: "Laptop", price: 999, quantity: 1 },
      { productId: 2, name: "Maus", price: 29, quantity: 2 }
    ],
    total: 1057,
    status: "paid"
  },
  meta: {
    createdAt: "2025-10-21T10:00:00Z",
    updatedAt: "2025-10-21T10:15:00Z"
  }
};

console.log("DB-Dokument:", dbDocument);
console.log("\nZugriff auf verschachtelte Daten:");
console.log("  Customer Name:", dbDocument.data.customer.name);
console.log("  Anzahl Items:", dbDocument.data.items.length);
console.log("  Erstes Item:", dbDocument.data.items[0].name);
console.log("  Total:", dbDocument.data.total, "€");
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Datenbank-Dokument ===
DB-Dokument:
{"_id":"507f1f77bcf86cd799439011","collection":"orders","data":
{"orderId":1001,"customer":
{"name":"Alice","email":"alice@example.com"},"items":
[{"productId":1,"name":"Laptop","price":999,"quantity":1},
{"productId":2,"name":"Maus","price":29,"quantity":2}],"total":1057,"statu
{"createdAt":"2025-10-21T10:00:00Z","updatedAt":"2025-10-
21T10:15:00Z"}}

Zugriff auf verschachtelte Daten:
  Customer Name: Alice
  Anzahl Items: 2
  Erstes Item: Laptop
  Total: 1057 €
--- Ende Beispiel 5 ---
```

4.2 Property-Zugriff

Es gibt zwei Arten, auf Objekt-Properties zuzugreifen: Dot-Notation mit Punkt und Bracket-Notation mit eckigen Klammern. Dot ist üblicher und lesbarer, aber Brackets sind flexibler – Sie können damit dynamische Keys oder Keys mit Leerzeichen verwenden. Beides ist wichtig für Datenbank-Arbeit!

Dot-Notation vs. Bracket-Notation:

```
console.log("=== Beispiel 1: Zwei Arten des Zugriffs ===");

const user = {
  id: 1,
  name: "Alice",
  email: "alice@example.com",
  "first login": "2025-01-15" // Key mit Leerzeichen
};

console.log("Objekt:", user);

console.log("\nDot-Notation:");
console.log("  user.name =", user.name);
console.log("  user.email =", user.email);

console.log("\nBracket-Notation:");
console.log('  user["name"] =', user["name"]);
console.log('  user["email"] =', user["email"]);

console.log("\nBei Leerzeichen nur Brackets:");
console.log('  user["first login"] =', user["first login"]);
// console.log("  user.first login =", "FEHLER!"); // ❌ Syntaxfehler
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Zwei Arten des Zugriffs ===
Objekt: {"id":1,"name":"Alice","email":"alice@example.com","first
login":"2025-01-15"}

Dot-Notation:
  user.name = Alice
  user.email = alice@example.com

Bracket-Notation:
  user["name"] = Alice
  user["email"] = alice@example.com

Bei Leerzeichen nur Brackets:
  user["first login"] = 2025-01-15
--- Ende Beispiel 1 ---
```

Dynamischer Zugriff mit Variablen:

```
console.log("=== Beispiel 2: Dynamischer Zugriff ===");

const product = {
  id: 101,
  name: "Laptop",
  price: 999,
  stock: 5
};

console.log("Produkt:", product);

const fields = ["name", "price", "stock"];
console.log("\nLese Felder dynamisch:");

for (const field of fields) {
  console.log(`  ${field}: ${product[field]}`); // Bracket mit Variab
}

console.log("\nSuche nach Feld:");
const searchField = "price";
console.log(`  Feld "${searchField}" hat Wert:`, product[searchField]
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Dynamischer Zugriff ===
Produkt: {"id":101,"name":"Laptop","price":999,"stock":5}

Lese Felder dynamisch:
  name: Laptop
  price: 999
  stock: 5

Suche nach Feld:
  Feld "price" hat Wert: 999
--- Ende Beispiel 2 ---
```

Properties hinzufügen & ändern:

```
console.log("=== Beispiel 3: Modify Properties ===");

const user = {
  name: "Bob",
  age: 25
};

console.log("Start:", user);
```

```
9
10 console.log("\nFüge Property hinzu:");
11 user.email = "bob@example.com";
12 console.log("  Nach user.email =", user);
13
14 console.log("\nÄndere existierendes Property:");
15 user.age = 26;
16 console.log("  Nach user.age = 26:", user);
17
18 console.log("\nFüge mit Brackets hinzu:");
19 user["role"] = "admin";
20 console.log("  Nach user['role'] =", user);
21
22 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Modify Properties ===
Start: {"name":"Bob","age":25}

Füge Property hinzu:
  Nach user.email = {"name":"Bob","age":25,"email":"bob@example.com"}

Ändere existierendes Property:
  Nach user.age = 26: {"name":"Bob","age":26,"email":"bob@example.com"}

Füge mit Brackets hinzu:
  Nach user['role'] =
{"name":"Bob","age":26,"email":"bob@example.com","role":"admin"}
--- Ende Beispiel 3 ---
```

Properties löschen:

```
1  console.log("=== Beispiel 4: Delete Properties ===");
2
3  const user = {
4    id: 1,
5    name: "Charlie",
6    email: "charlie@example.com",
7    tempToken: "abc123"
8  };
9
10 console.log("Vorher:", user);
11
12 console.log("\nLösche tempToken:");
13 delete user.tempToken;
14 console.log("Nachher:", user);
15
16 console.log("\nPrüfe ob Property existiert:");
17 console.log("  'name' in user:", "name" in user);
```

```
console.log("  'tempToken' in user:", "tempToken" in user);
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Delete Properties ===
Vorher:
{"id":1,"name":"Charlie","email":"charlie@example.com","tempToken":"abc123

Lösche tempToken:
Nachher: {"id":1,"name":"Charlie","email":"charlie@example.com"}

Prüfe ob Property existiert:
  'name' in user: true
  'tempToken' in user: false
--- Ende Beispiel 4 ---
```

Praxis: Datenbank-Query-Builder:

```
console.log("=== Beispiel 5: Query-Builder ===");

const buildQuery = (table, conditions) => {
  console.log(`\nBaue Query für Tabelle: ${table}`);
  console.log("Conditions:", conditions);

  const whereClauses = [];

  for (const key in conditions) {
    const value = conditions[key];
    console.log(`  → Prüfe: ${key} = ${value}`);

    if (typeof value === "string") {
      whereClauses.push(`${key} = '${value}'`);
    } else {
      whereClauses.push(`${key} = ${value}`);
    }
  }

  const whereString = whereClauses.join(" AND ");
  const query = `SELECT * FROM ${table} WHERE ${whereString}`;

  console.log(`\n✅ Query: ${query}`);
  return query;
};

buildQuery("users", { active: true, role: "admin" });
buildQuery("products", { category: "electronics", price: 999 });
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Query-Builder ===

Baue Query für Tabelle: users
Conditions: {"active":true,"role":"admin"}
  → Prüfe: active = true
  → Prüfe: role = admin

✅ Query: SELECT * FROM users WHERE active = true AND role = 'admin'

Baue Query für Tabelle: products
Conditions: {"category":"electronics","price":999}
  → Prüfe: category = electronics
  → Prüfe: price = 999

✅ Query: SELECT * FROM products WHERE category = 'electronics' AND
price = 999
--- Ende Beispiel 5 ---
```

Optional Chaining (?.):

```javascript
console.log("=== Beispiel 6: Optional Chaining ===");

const users = [
  { name: "Alice", address: { city: "Berlin" } },
  { name: "Bob", address: null },
  { name: "Charlie" } // Kein address-Feld
];

console.log("Users:", users);

console.log("\nOhne Optional Chaining (⚠️ kann crashen):");
for (const user of users) {
  console.log(`\n${user.name}:`);
  // console.log("  Stadt:", user.address.city); // ❌ Crash bei Bob/

  // ✅ Manueller Check:
  if (user.address && user.address.city) {
    console.log("  Stadt:", user.address.city);
  } else {
    console.log("  Stadt: N/A");
  }
}

console.log("\n\nMit Optional Chaining (✅ sicher):");
for (const user of users) {
  console.log(`${user.name}: Stadt =`, user.address?.city || "N/A");
}
```

```
28 console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Optional Chaining ===
Users: [{"name":"Alice","address":{"city":"Berlin"}},
{"name":"Bob","address":null},{"name":"Charlie"}]

Ohne Optional Chaining (⚠️ kann crashen):

Alice:
  Stadt: Berlin

Bob:
  Stadt: N/A

Charlie:
  Stadt: N/A


Mit Optional Chaining (✅ sicher):
Alice: Stadt = Berlin
Bob: Stadt = N/A
Charlie: Stadt = N/A
--- Ende Beispiel 6 ---
```

4.3 Object Methods

Objekte können nicht nur Daten speichern, sondern auch Funktionen – diese nennt man Methoden. Das ist zentral für objektorientierte Programmierung. Besonders wichtig: Die eingebauten Object-Methoden wie Object.keys, Object.values und Object.entries zum Durchlaufen von Objekten.

Methoden in Objekten:

```
1  console.log("=== Beispiel 1: Object Methods ===");
2
3  const user = {
4    name: "Alice",
5    age: 28,
6
7    greet: function() {
8      console.log(`   → Hallo, ich bin ${this.name}!`);
9    },
10
11   // Shorthand Syntax (ES6):
12   getInfo() {
13     console.log(`   → ${this.name}, ${this.age} Jahre alt`);
```

```
  }
};

console.log("User-Objekt:", user);

console.log("\nRufe Methoden auf:");
user.greet();
user.getInfo();
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Object Methods ===
User-Objekt: {"name":"Alice","age":28}

Rufe Methoden auf:
  → Hallo, ich bin Alice!
  → Alice, 28 Jahre alt
--- Ende Beispiel 1 ---
```

Object.keys() - Alle Keys:

```
console.log("=== Beispiel 2: Object.keys() ===");

const product = {
  id: 101,
  name: "Laptop",
  price: 999,
  stock: 5
};

console.log("Produkt:", product);

console.log("\nAlle Keys:");
const keys = Object.keys(product);
console.log("  Keys:", keys);
console.log("  Typ:", Array.isArray(keys) ? "Array" : "kein Array");

console.log("\nIteriere über Keys:");
for (const key of keys) {
  console.log(`  ${key}: ${product[key]}`);
}
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Object.keys() ===
Produkt: {"id":101,"name":"Laptop","price":999,"stock":5}

Alle Keys:
  Keys: ["id","name","price","stock"]
  Typ: Array

Iteriere über Keys:
  id: 101
  name: Laptop
  price: 999
  stock: 5
--- Ende Beispiel 2 ---
```

Object.values() - Alle Werte:

```javascript
console.log("=== Beispiel 3: Object.values() ===");

const scores = {
  Alice: 95,
  Bob: 87,
  Charlie: 92,
  David: 88
};

console.log("Scores:", scores);

console.log("\nAlle Werte:");
const values = Object.values(scores);
console.log("  Values:", values);

console.log("\nBerechne Durchschnitt:");
const sum = values.reduce((acc, val) => acc + val, 0);
const average = sum / values.length;
console.log(`  Summe: ${sum}`);
console.log(`  Durchschnitt: ${average.toFixed(2)}`);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Object.values() ===
Scores: {"Alice":95,"Bob":87,"Charlie":92,"David":88}

Alle Werte:
  Values: [95,87,92,88]

Berechne Durchschnitt:
  Summe: 362
  Durchschnitt: 90.50
--- Ende Beispiel 3 ---
```

Object.entries() - Key-Value-Paare:

```javascript
console.log("=== Beispiel 4: Object.entries() ===");

const config = {
  host: "localhost",
  port: 5432,
  database: "mydb",
  user: "admin"
};

console.log("Config:", config);

console.log("\nAlle Entries:");
const entries = Object.entries(config);
console.log("  Entries:", entries);
console.log("  Format: Array von [key, value] Paaren");

console.log("\nIteriere mit Destructuring:");
for (const [key, value] of entries) {
  console.log(`  ${key}: ${value}`);
}
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Object.entries() ===
Config:
{"host":"localhost","port":5432,"database":"mydb","user":"admin"}

Alle Entries:
  Entries: [["host","localhost"],["port",5432],["database","mydb"],
["user","admin"]]
  Format: Array von [key, value] Paaren

Iteriere mit Destructuring:
  host: localhost
  port: 5432
  database: mydb
  user: admin
--- Ende Beispiel 4 ---
```

Praxis: Objekt-Validierung:

```javascript
console.log("=== Beispiel 5: Validierung ===");

const validateProduct = (product) => {
  console.log("\nValidiere Produkt:", product);

  const requiredFields = ["id", "name", "price"];
  const keys = Object.keys(product);

  console.log("  Erforderlich:", requiredFields);
  console.log("  Vorhanden:", keys);

  for (const field of requiredFields) {
    if (!keys.includes(field)) {
      console.log(`  ❌ Fehlt: ${field}`);
      return false;
    }
  }

  console.log("  ✅ Alle Pflichtfelder vorhanden");
  return true;
};

const products = [
  { id: 1, name: "Laptop", price: 999 },
  { id: 2, name: "Maus" }, // price fehlt
  { name: "Tastatur", price: 79 } // id fehlt
];

products.forEach((p, i) => {
```

```
  console.log(`\n→ Produkt ${i + 1}:`);
  validateProduct(p);
});
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Validierung ===

→ Produkt 1:

Validiere Produkt: {"id":1,"name":"Laptop","price":999}
  Erforderlich: ["id","name","price"]
  Vorhanden: ["id","name","price"]
  ✅ Alle Pflichtfelder vorhanden

→ Produkt 2:

Validiere Produkt: {"id":2,"name":"Maus"}
  Erforderlich: ["id","name","price"]
  Vorhanden: ["id","name"]
  ❌ Fehlt: price

→ Produkt 3:

Validiere Produkt: {"name":"Tastatur","price":79}
  Erforderlich: ["id","name","price"]
  Vorhanden: ["name","price"]
  ❌ Fehlt: id
--- Ende Beispiel 5 ---
```

Object.assign() & Spread Operator:

```
console.log("=== Beispiel 6: Objekte zusammenführen ===");

const defaults = {
  theme: "light",
  notifications: true,
  language: "en"
};

const userSettings = {
  theme: "dark",
  language: "de"
};

console.log("Defaults:", defaults);
console.log("User Settings:", userSettings);
```

```
16
17 console.log("\nMit Object.assign():");
18 const merged1 = Object.assign({}, defaults, userSettings);
19 console.log("  Merged:", merged1);
20
21 console.log("\nMit Spread Operator (moderner):");
22 const merged2 = { ...defaults, ...userSettings };
23 console.log("  Merged:", merged2);
24
25 console.log("\nUser-Werte überschreiben Defaults!");
26 console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Objekte zusammenführen ===
Defaults: {"theme":"light","notifications":true,"language":"en"}
User Settings: {"theme":"dark","language":"de"}

Mit Object.assign():
  Merged: {"theme":"dark","notifications":true,"language":"de"}

Mit Spread Operator (moderner):
  Merged: {"theme":"dark","notifications":true,"language":"de"}

User-Werte überschreiben Defaults!
--- Ende Beispiel 6 ---
```

4.4 Destructuring (Objekte)

Destructuring ist eine elegante Syntax, um Werte aus Objekten zu extrahieren. Statt mehrere Zeilen mit user.name, user.email zu schreiben, extrahieren Sie alles in einer Zeile. Besonders praktisch bei Funktionsparametern und API-Responses. Moderne Datenbank-Libraries nutzen das intensiv!

Grundlagen:

```
1  console.log("=== Beispiel 1: Destructuring Basics ===");
2
3  const user = {
4    id: 1,
5    name: "Alice",
6    email: "alice@example.com",
7    age: 28
8  };
9
10 console.log("User:", user);
11
12 console.log("\nOhne Destructuring:");
13 const name1 = user.name;
```

```
const email1 = user.email;
console.log(`  ${name1}, ${email1}`);

console.log("\nMit Destructuring:");
const { name, email } = user;
console.log(`  ${name}, ${email}`);

console.log("\nBeide sind identisch!");
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Destructuring Basics ===
User: {"id":1,"name":"Alice","email":"alice@example.com","age":28}

Ohne Destructuring:
  Alice, alice@example.com

Mit Destructuring:
  Alice, alice@example.com

Beide sind identisch!
--- Ende Beispiel 1 ---
```

Mit Umbenennung:

```
console.log("=== Beispiel 2: Umbenennen ===");

const product = {
  id: 101,
  name: "Laptop",
  price: 999
};

console.log("Produkt:", product);

console.log("\nDestructuring mit Rename:");
const { name: productName, price: productPrice } = product;
console.log(`  productName: ${productName}`);
console.log(`  productPrice: ${productPrice}`);

console.log("\nOriginal 'name' existiert nicht in diesem Scope:");
// console.log("  name:", name); // ❌ ReferenceError
console.log("  (nur productName ist definiert)");
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Umbenennen ===
Produkt: {"id":101,"name":"Laptop","price":999}

Destructuring mit Rename:
  productName: Laptop
  productPrice: 999

Original 'name' existiert nicht in diesem Scope:
  (nur productName ist definiert)
--- Ende Beispiel 2 ---
```

Mit Default Values:

```javascript
console.log("=== Beispiel 3: Default Values ===");

const user = {
  name: "Bob",
  email: "bob@example.com"
  // role fehlt!
};

console.log("User:", user);

console.log("\nDestructuring mit Defaults:");
const { name, email, role = "user", active = true } = user;
console.log(`  name: ${name}`);
console.log(`  email: ${email}`);
console.log(`  role: ${role} (default)`);
console.log(`  active: ${active} (default)`);
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Default Values ===
User: {"name":"Bob","email":"bob@example.com"}

Destructuring mit Defaults:
  name: Bob
  email: bob@example.com
  role: user (default)
  active: true (default)
--- Ende Beispiel 3 ---
```

Nested Destructuring:

```javascript
console.log("=== Beispiel 4: Verschachtelt ===");

```

```
const order = {
  orderId: 1001,
  customer: {
    name: "Charlie",
    address: {
      city: "Berlin",
      zip: "10115"
    }
  },
  total: 199
};

console.log("Order:", order);

console.log("\nNested Destructuring:");
const {
  orderId,
  customer: {
    name: customerName,
    address: { city }
  }
} = order;

console.log(`  Order: ${orderId}`);
console.log(`  Customer: ${customerName}`);
console.log(`  City: ${city}`);
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Verschachtelt ===
Order: {"orderId":1001,"customer":{"name":"Charlie","address":
{"city":"Berlin","zip":"10115"}},"total":199}

Nested Destructuring:
  Order: 1001
  Customer: Charlie
  City: Berlin
--- Ende Beispiel 4 ---
```

In Funktionsparametern:

```
console.log("=== Beispiel 5: In Funktionen ===");

// Ohne Destructuring:
const displayUser1 = (user) => {
  console.log(`\n→ ${user.name} (${user.email})`);
};

// Mit Destructuring (eleganter):
```

```
const displayUser2 = ({ name, email, role = "user" }) => {
  console.log(`\n→ ${name} (${email}) - Role: ${role}`);
};

const users = [
  { name: "Alice", email: "alice@test.com", role: "admin" },
  { name: "Bob", email: "bob@test.com" }
];

console.log("Ohne Destructuring:");
users.forEach(displayUser1);

console.log("\nMit Destructuring:");
users.forEach(displayUser2);
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: In Funktionen ===
Ohne Destructuring:

→ Alice (alice@test.com)

→ Bob (bob@test.com)

Mit Destructuring:

→ Alice (alice@test.com) - Role: admin

→ Bob (bob@test.com) - Role: user
--- Ende Beispiel 5 ---
```

Praxis: API-Response verarbeiten:

```
console.log("=== Beispiel 6: API-Response ===");

const apiResponse = {
  status: 200,
  data: {
    users: [
      { id: 1, name: "Alice" },
      { id: 2, name: "Bob" }
    ],
    pagination: {
      page: 1,
      totalPages: 5,
      totalItems: 47
    }
  },
```

```
  meta: {
    timestamp: "2025-10-21T10:00:00Z",
    version: "1.0"
  }
};

console.log("API-Response:", apiResponse);

console.log("\nExtrahiere wichtige Daten:");
const {
  status,
  data: {
    users,
    pagination: { page, totalPages }
  }
} = apiResponse;

console.log(`  Status: ${status}`);
console.log(`  Users gefunden: ${users.length}`);
console.log(`  Seite ${page} von ${totalPages}`);

console.log("\nVerarbeite Users:");
users.forEach(({ id, name }) => {
  console.log(`  [${id}] ${name}`);
});
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: API-Response ===
API-Response: {"status":200,"data":{"users":[{"id":1,"name":"Alice"},
{"id":2,"name":"Bob"}],"pagination":
{"page":1,"totalPages":5,"totalItems":47}},"meta":{"timestamp":"2025-
10-21T10:00:00Z","version":"1.0"}}

Extrahiere wichtige Daten:
  Status: 200
  Users gefunden: 2
  Seite 1 von 5

Verarbeite Users:
  [1] Alice
  [2] Bob
--- Ende Beispiel 6 ---
```

4.5 Arrays: Grundlagen

Arrays sind geordnete Listen von Werten. In Datenbanken sind Query-Ergebnisse fast immer Arrays von Dokumenten. Arrays sind nullbasiert – das erste Element hat Index null. Sie können verschiedene Typen mischen, aber in der Praxis enthalten DB-Results meist gleichartige Objekte.

Array erstellen:

```
console.log("=== Beispiel 1: Arrays erstellen ===");

const numbers = [1, 2, 3, 4, 5];
console.log("Numbers:", numbers);
console.log("  Länge:", numbers.length);
console.log("  Typ:", Array.isArray(numbers) ? "Array" : "kein Array"

const mixed = [1, "text", true, null, { key: "value" }];
console.log("\nGemischte Typen:", mixed);

const empty = [];
console.log("\nLeeres Array:", empty);
console.log("  Länge:", empty.length);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Arrays erstellen ===
Numbers: [1,2,3,4,5]
  Länge: 5
  Typ: Array

Gemischte Typen: [1,"text",true,null,{"key":"value"}]

Leeres Array: []
  Länge: 0
--- Ende Beispiel 1 ---
```

Zugriff mit Index:

```
console.log("=== Beispiel 2: Index-Zugriff ===");

const fruits = ["🍎 Apfel", "🍌 Banane", "🍇 Trauben", "🍊 Orange"];
console.log("Fruits:", fruits);

console.log("\nZugriff per Index (0-basiert):");
console.log("  [0]:", fruits[0]);
console.log("  [1]:", fruits[1]);
console.log("  [2]:", fruits[2]);

console.log("\nLetztes Element:");
console.log("  [-1] geht nicht in JS!"); // ❌ Nicht wie Python
console.log("  [length-1]:", fruits[fruits.length - 1]);
```

```
15 console.log("\nUngültiger Index:");
16 console.log("  [99]:", fruits[99]); // undefined
17 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Index-Zugriff ===
Fruits: ["🍎 Apfel","🍌 Banane","🍇 Trauben","🍊 Orange"]

Zugriff per Index (0-basiert):
  [0]: 🍎 Apfel
  [1]: 🍌 Banane
  [2]: 🍇 Trauben

Letztes Element:
  [-1] geht nicht in JS!
  [length-1]: 🍊 Orange

Ungültiger Index:
  [99]: undefined
--- Ende Beispiel 2 ---
```

Elemente hinzufügen & entfernen:

```
1  console.log("=== Beispiel 3: Modify Arrays ===");
2
3  const stack = [1, 2, 3];
4  console.log("Start:", stack);
5
6  console.log("\npush() - Hinten anfügen:");
7  stack.push(4);
8  console.log("  Nach push(4):", stack);
9
10 console.log("\npop() - Letztes entfernen:");
11 const last = stack.pop();
12 console.log(`  Entfernt: ${last}`);
13 console.log("  Nach pop():", stack);
14
15 console.log("\nunshift() - Vorne einfügen:");
16 stack.unshift(0);
17 console.log("  Nach unshift(0):", stack);
18
19 console.log("\nshift() - Erstes entfernen:");
20 const first = stack.shift();
21 console.log(`  Entfernt: ${first}`);
22 console.log("  Nach shift():", stack);
23
24 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Modify Arrays ===
Start: [1,2,3]

push() - Hinten anfügen:
  Nach push(4): [1,2,3,4]

pop() - Letztes entfernen:
  Entfernt: 4
  Nach pop(): [1,2,3]

unshift() - Vorne einfügen:
  Nach unshift(0): [0,1,2,3]

shift() - Erstes entfernen:
  Entfernt: 0
  Nach shift(): [1,2,3]
--- Ende Beispiel 3 ---
```

splice() - Einfügen & Löschen:

```
1  console.log("=== Beispiel 4: splice() ===");
2
3  const letters = ["A", "B", "E", "F"];
4  console.log("Start:", letters);
5
6  console.log("\nFüge 'C' und 'D' bei Index 2 ein:");
7  letters.splice(2, 0, "C", "D"); // Ab Index 2, 0 löschen, 2 einfügen
8  console.log("  Nach splice:", letters);
9
10 console.log("\nLösche 1 Element bei Index 4:");
11 const removed = letters.splice(4, 1); // Ab Index 4, 1 löschen
12 console.log(`  Entfernt: ${removed}`);
13 console.log("  Nach splice:", letters);
14
15 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: splice() ===
Start: ["A","B","E","F"]

Füge 'C' und 'D' bei Index 2 ein:
  Nach splice: ["A","B","C","D","E","F"]

Lösche 1 Element bei Index 4:
  Entfernt: E
  Nach splice: ["A","B","C","D","F"]
--- Ende Beispiel 4 ---
```

Array of Objects (DB-Results!):

```
1  console.log("=== Beispiel 5: DB-Results simuliert ===");
2
3  const users = [
4    { id: 1, name: "Alice", active: true },
5    { id: 2, name: "Bob", active: false },
6    { id: 3, name: "Charlie", active: true }
7  ];
8
9  console.log("DB-Query-Result:", users);
10 console.log("  Anzahl:", users.length);
11
12 console.log("\nDurchlaufe Ergebnisse:");
13 for (let i = 0; i < users.length; i++) {
14   const user = users[i];
15   console.log(`  [${i}] ${user.name} (ID: ${user.id})`);
16   console.log(`      Status: ${user.active ? "✅ aktiv" : "❌ inaktiv
17 }
18 console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: DB-Results simuliert ===
DB-Query-Result: [{"id":1,"name":"Alice","active":true},
{"id":2,"name":"Bob","active":false},
{"id":3,"name":"Charlie","active":true}]
  Anzahl: 3

Durchlaufe Ergebnisse:
  [0] Alice (ID: 1)
      Status: ✅ aktiv
  [1] Bob (ID: 2)
      Status: ❌ inaktiv
  [2] Charlie (ID: 3)
      Status: ✅ aktiv
--- Ende Beispiel 5 ---
```

Spread Operator mit Arrays:

```javascript
console.log("=== Beispiel 6: Spread Operator ===");

const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

console.log("Array 1:", arr1);
console.log("Array 2:", arr2);

console.log("\nKombinieren mit Spread:");
const combined = [...arr1, ...arr2];
console.log("  Combined:", combined);

console.log("\nKopieren:");
const copy = [...arr1];
console.log("  Original:", arr1);
console.log("  Copy:", copy);

console.log("\nSind unterschiedliche Arrays:");
copy.push(999);
console.log("  Nach copy.push(999):");
console.log("    Original:", arr1);
console.log("    Copy:", copy);
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Spread Operator ===
Array 1: [1,2,3]
Array 2: [4,5,6]

Kombinieren mit Spread:
  Combined: [1,2,3,4,5,6]

Kopieren:
  Original: [1,2,3]
  Copy: [1,2,3]

Sind unterschiedliche Arrays:
  Nach copy.push(999):
    Original: [1,2,3]
    Copy: [1,2,3,999]
--- Ende Beispiel 6 ---
```

4.6 Array-Iteration

Es gibt viele Wege, Arrays zu durchlaufen. Die klassische for-Schleife kennen Sie schon. Jetzt lernen Sie die modernen Array-Methoden – forEach ist der einfachste Einstieg. Diese Methoden sind der Schlüssel zu eleganter Datenverarbeitung!

forEach() - Für jedes Element:

```
1  console.log("=== Beispiel 1: forEach() ===");
2
3  const numbers = [10, 20, 30, 40];
4  console.log("Array:", numbers);
5
6  console.log("\nDurchlaufe mit forEach:");
7  numbers.forEach((num, index) => {
8    console.log(`  [${index}] Wert: ${num}, Verdoppelt: ${num * 2}`);
9  });
10
11 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: forEach() ===
Array: [10,20,30,40]

Durchlaufe mit forEach:
  [0] Wert: 10, Verdoppelt: 20
  [1] Wert: 20, Verdoppelt: 40
  [2] Wert: 30, Verdoppelt: 60
  [3] Wert: 40, Verdoppelt: 80
--- Ende Beispiel 1 ---
```

Praxis: DB-Results ausgeben:

```
1  console.log("=== Beispiel 2: forEach mit Objekten ===");
2
3  const products = [
4    { id: 1, name: "Laptop", price: 999, stock: 5 },
5    { id: 2, name: "Maus", price: 29, stock: 150 },
6    { id: 3, name: "Tastatur", price: 79, stock: 0 }
7  ];
8
9  console.log("Produkte:", products);
10 console.log("\nAusgabe formatiert:");
11
12 products.forEach((product, index) => {
13   console.log(`\n[${index + 1}] ${product.name}`);
14   console.log(`    ID: ${product.id}`);
15   console.log(`    Preis: ${product.price}€`);
16   console.log(`    Lager: ${product.stock > 0 ? `${product.stock} Stü
       "❌ Ausverkauft"}`);
17 });
18
19 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: forEach mit Objekten ===
Produkte: [{"id":1,"name":"Laptop","price":999,"stock":5},
{"id":2,"name":"Maus","price":29,"stock":150},
{"id":3,"name":"Tastatur","price":79,"stock":0}]

Ausgabe formatiert:

[1] Laptop
    ID: 1
    Preis: 999€
    Lager: 5 Stück

[2] Maus
    ID: 2
    Preis: 29€
    Lager: 150 Stück

[3] Tastatur
    ID: 3
    Preis: 79€
    Lager: ❌ Ausverkauft
--- Ende Beispiel 2 ---
```

4.7 Array-Methoden: forEach, map, filter, find

Jetzt kommen die WICHTIGSTEN Array-Methoden für Datenbank-Arbeit. Map transformiert Daten, filter selektiert Daten, find sucht einzelne Elemente. Diese Methoden sind funktional – sie ändern das Original-Array nicht, sondern geben ein neues zurück. Das ist sicherer und lesbarer als Schleifen!

map() - Transformieren:

```javascript
console.log("=== Beispiel 1: map() ===");

const numbers = [1, 2, 3, 4, 5];
console.log("Original:", numbers);

console.log("\nVerdopple mit map:");
const doubled = numbers.map(num => {
  console.log(`  ${num} → ${num * 2}`);
  return num * 2;
});

console.log("\nErgebnis:", doubled);
console.log("Original unverändert:", numbers);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: map() ===
Original: [1,2,3,4,5]

Verdopple mit map:
  1 → 2
  2 → 4
  3 → 6
  4 → 8
  5 → 10

Ergebnis: [2,4,6,8,10]
Original unverändert: [1,2,3,4,5]
--- Ende Beispiel 1 ---
```

map() mit Objekten (Praxis!):

```javascript
console.log("=== Beispiel 2: map() - Daten extrahieren ===");

const users = [
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" },
  { id: 3, name: "Charlie", email: "charlie@example.com" }
];

console.log("Users:", users);

console.log("\nExtrahiere nur Namen:");
const names = users.map(user => {
  console.log(`  Extrahiere: ${user.name}`);
  return user.name;
});
console.log("Names:", names);

console.log("\nErstelle Email-Liste:");
const emails = users.map(u => u.email);
console.log("Emails:", emails);

console.log("\nTransformiere zu neuem Format:");
const simplified = users.map(({ id, name }) => ({
  userId: id,
  displayName: name.toUpperCase()
}));
console.log("Simplified:", simplified);
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: map() - Daten extrahieren ===
Users: [{"id":1,"name":"Alice","email":"alice@example.com"},
{"id":2,"name":"Bob","email":"bob@example.com"},
{"id":3,"name":"Charlie","email":"charlie@example.com"}]

Extrahiere nur Namen:
  Extrahiere: Alice
  Extrahiere: Bob
  Extrahiere: Charlie
Names: ["Alice","Bob","Charlie"]

Erstelle Email-Liste:
Emails: ["alice@example.com","bob@example.com","charlie@example.com"]

Transformiere zu neuem Format:
Simplified: [{"userId":1,"displayName":"ALICE"},
{"userId":2,"displayName":"BOB"},{"userId":3,"displayName":"CHARLIE"}]
--- Ende Beispiel 2 ---
```

filter() - Selektieren:

```
1  console.log("=== Beispiel 3: filter() ===");
2
3  const numbers = [1, 5, 10, 15, 20, 25, 30];
4  console.log("Zahlen:", numbers);
5
6  console.log("\nFilter: Nur Zahlen > 15");
7  const filtered = numbers.filter(num => {
8    const keep = num > 15;
9    console.log(`  ${num}: ${keep ? "✅ behalten" : "❌ verwerfen"}`);
10   return keep;
11 });
12
13 console.log("\nErgebnis:", filtered);
14 console.log("Original unverändert:", numbers);
15 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: filter() ===
Zahlen: [1,5,10,15,20,25,30]

Filter: Nur Zahlen > 15
  1: ❌ verwerfen
  5: ❌ verwerfen
  10: ❌ verwerfen
  15: ❌ verwerfen
  20: ✅ behalten
  25: ✅ behalten
  30: ✅ behalten

Ergebnis: [20,25,30]
Original unverändert: [1,5,10,15,20,25,30]
--- Ende Beispiel 3 ---
```

filter() mit Objekten (DB-Query!):

```
1  console.log("=== Beispiel 4: filter() - WHERE clause simuliert ===");
2
3  const products = [
4    { id: 1, name: "Laptop", price: 999, category: "electronics", stock
5    { id: 2, name: "Maus", price: 29, category: "electronics", stock: 1
6    { id: 3, name: "Stuhl", price: 199, category: "furniture", stock: 1
7    { id: 4, name: "Monitor", price: 299, category: "electronics", stoc
8  ];
9
10 console.log("Alle Produkte:", products);
11
12 console.log("\nQuery 1: Elektronik + Preis >= 100 + Lager > 0");
13 const query1 = products.filter(p => {
14   const match = p.category === "electronics" && p.price >= 100 && p.s
       0;
15   console.log(`  ${p.name}: ${match ? "✅" : "❌"}`);
16   return match;
17 });
18 console.log("Ergebnis:", query1);
19
20 console.log("\nQuery 2: Ausverkaufte Produkte");
21 const outOfStock = products.filter(p => p.stock === 0);
22 console.log("Ergebnis:", outOfStock.map(p => p.name));
23 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: filter() - WHERE clause simuliert ===
Alle Produkte:
[{"id":1,"name":"Laptop","price":999,"category":"electronics","stock":5},
{"id":2,"name":"Maus","price":29,"category":"electronics","stock":150},
{"id":3,"name":"Stuhl","price":199,"category":"furniture","stock":10},
{"id":4,"name":"Monitor","price":299,"category":"electronics","stock":0}]

Query 1: Elektronik + Preis >= 100 + Lager > 0
  Laptop: ✅
  Maus: ❌
  Stuhl: ❌
  Monitor: ❌
Ergebnis:
[{"id":1,"name":"Laptop","price":999,"category":"electronics","stock":5}]

Query 2: Ausverkaufte Produkte
Ergebnis: ["Monitor"]
--- Ende Beispiel 4 ---
```

find() - Erstes Element finden:

```javascript
console.log("=== Beispiel 5: find() ===");

const users = [
  { id: 1, name: "Alice", role: "user" },
  { id: 2, name: "Bob", role: "admin" },
  { id: 3, name: "Charlie", role: "user" }
];

console.log("Users:", users);

console.log("\nSuche User mit ID 2:");
const user = users.find(u => {
  console.log(`  Prüfe: ${u.name} (ID: ${u.id})`);
  return u.id === 2;
});
console.log("Gefunden:", user);

console.log("\nSuche ersten Admin:");
const admin = users.find(u => u.role === "admin");
console.log("Admin:", admin.name);

console.log("\nSuche nicht-existierenden User:");
const notFound = users.find(u => u.id === 999);
console.log("Ergebnis:", notFound); // undefined
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: find() ===
Users: [{"id":1,"name":"Alice","role":"user"},
{"id":2,"name":"Bob","role":"admin"},
{"id":3,"name":"Charlie","role":"user"}]

Suche User mit ID 2:
  Prüfe: Alice (ID: 1)
  Prüfe: Bob (ID: 2)
Gefunden: {"id":2,"name":"Bob","role":"admin"}

Suche ersten Admin:
Admin: Bob

Suche nicht-existierenden User:
Ergebnis: undefined
--- Ende Beispiel 5 ---
```

Kombination: map + filter:

```
console.log("=== Beispiel 6: map + filter kombiniert ===");

const orders = [
  { id: 1, customer: "Alice", amount: 150, status: "paid" },
  { id: 2, customer: "Bob", amount: 80, status: "pending" },
  { id: 3, customer: "Charlie", amount: 200, status: "paid" },
  { id: 4, customer: "David", amount: 50, status: "cancelled" }
];

console.log("Orders:", orders);

console.log("\nPipeline: paid orders → amounts → sum");

console.log("\n1. Filter: Nur bezahlte Orders");
const paidOrders = orders.filter(o => {
  console.log(`  ${o.id}: ${o.status} → ${o.status === "paid" ? "✅"
    );
  return o.status === "paid";
});

console.log("\n2. Map: Extrahiere amounts");
const amounts = paidOrders.map(o => {
  console.log(`  Order ${o.id}: ${o.amount}€`);
  return o.amount;
});

console.log("\n3. Reduce: Summiere");
const total = amounts.reduce((sum, amount) => {
```

```
  const newSum = sum + amount;
  console.log(`  ${sum} + ${amount} = ${newSum}`);
  return newSum;
}, 0);

console.log(`\n✅ Gesamtumsatz (paid): ${total}€`);

console.log("\nOder in einer Chain:");
const totalChained = orders
  .filter(o => o.status === "paid")
  .map(o => o.amount)
  .reduce((sum, amt) => sum + amt, 0);
console.log(`Total (chained): ${totalChained}€`);
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: map + filter kombiniert ===
Orders: [{"id":1,"customer":"Alice","amount":150,"status":"paid"},
{"id":2,"customer":"Bob","amount":80,"status":"pending"},
{"id":3,"customer":"Charlie","amount":200,"status":"paid"},
{"id":4,"customer":"David","amount":50,"status":"cancelled"}]

Pipeline: paid orders → amounts → sum

1. Filter: Nur bezahlte Orders
  1: paid → ✅
  2: pending → ❌
  3: paid → ✅
  4: cancelled → ❌

2. Map: Extrahiere amounts
  Order 1: 150€
  Order 3: 200€

3. Reduce: Summiere
  0 + 150 = 150
  150 + 200 = 350

✅ Gesamtumsatz (paid): 350€

Oder in einer Chain:
Total (chained): 350€
--- Ende Beispiel 6 ---
```

4.8 Array-Methoden: reduce, some, every

Reduce ist die mächtigste Array-Methode – sie kann alles, was map und filter können, und mehr. Some und every sind Prüf-Methoden, die true oder false zurückgeben. Diese drei Methoden vervollständigen Ihr Array-Werkzeugkasten für komplexe Datenverarbeitung.

reduce() - Akkumulieren:

```
console.log("=== Beispiel 1: reduce() - Summe ===");

const numbers = [10, 20, 30, 40, 50];
console.log("Zahlen:", numbers);

console.log("\nBerechne Summe:");
const sum = numbers.reduce((accumulator, current) => {
  console.log(`  Acc: ${accumulator}, Current: ${current}`);
  const newAcc = accumulator + current;
  console.log(`    → Neuer Acc: ${newAcc}`);
  return newAcc;
}, 0); // Start bei 0

console.log(`\n✅ Summe: ${sum}`);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: reduce() - Summe ===
Zahlen: [10,20,30,40,50]

Berechne Summe:
  Acc: 0, Current: 10
    → Neuer Acc: 10
  Acc: 10, Current: 20
    → Neuer Acc: 30
  Acc: 30, Current: 30
    → Neuer Acc: 60
  Acc: 60, Current: 40
    → Neuer Acc: 100
  Acc: 100, Current: 50
    → Neuer Acc: 150

✅ Summe: 150
--- Ende Beispiel 1 ---
```

reduce() - Komplexe Aggregation:

```
console.log("=== Beispiel 2: reduce() - Statistiken ===");

```

```
const products = [
  { name: "Laptop", price: 999, sold: 5 },
  { name: "Maus", price: 29, sold: 150 },
  { name: "Tastatur", price: 79, sold: 80 }
];

console.log("Produkte:", products);

console.log("\nBerechne Gesamt-Umsatz:");
const totalRevenue = products.reduce((acc, product) => {
  const revenue = product.price * product.sold;
  console.log(`  ${product.name}: ${product.price}€ × ${product.sold}
    ${revenue}€`);
  console.log(`    Acc: ${acc} → ${acc + revenue}`);
  return acc + revenue;
}, 0);

console.log(`\n✅ Gesamt-Umsatz: ${totalRevenue}€`);
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: reduce() - Statistiken ===
Produkte: [{"name":"Laptop","price":999,"sold":5},
{"name":"Maus","price":29,"sold":150},
{"name":"Tastatur","price":79,"sold":80}]

Berechne Gesamt-Umsatz:
  Laptop: 999€ × 5 = 4995€
    Acc: 0 → 4995
  Maus: 29€ × 150 = 4350€
    Acc: 4995 → 9345
  Tastatur: 79€ × 80 = 6320€
    Acc: 9345 → 15665

✅ Gesamt-Umsatz: 15665€
--- Ende Beispiel 2 ---
```

reduce() - Objekt aufbauen (Group By):

```
console.log("=== Beispiel 3: reduce() - Gruppieren ===");

const users = [
  { name: "Alice", role: "admin" },
  { name: "Bob", role: "user" },
  { name: "Charlie", role: "admin" },
  { name: "David", role: "user" },
  { name: "Eve", role: "moderator" }
];
```

```
10
11 console.log("Users:", users);
12
13 console.log("\nGruppiere nach Rolle:");
14 const grouped = users.reduce((acc, user) => {
15   console.log(`\n  Verarbeite: ${user.name} (${user.role})`);
16
17   if (!acc[user.role]) {
18     console.log(`    Neue Gruppe: ${user.role}`);
19     acc[user.role] = [];
20   }
21
22   acc[user.role].push(user.name);
23   console.log(`    ${user.role}:`, acc[user.role]);
24
25   return acc;
26 }, {});
27
28 console.log("\n✅ Gruppiert:", grouped);
29 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: reduce() - Gruppieren ===
Users: [{"name":"Alice","role":"admin"},{"name":"Bob","role":"user"},
{"name":"Charlie","role":"admin"},{"name":"David","role":"user"},
{"name":"Eve","role":"moderator"}]

Gruppiere nach Rolle:

  Verarbeite: Alice (admin)
    Neue Gruppe: admin
    admin: ["Alice"]

  Verarbeite: Bob (user)
    Neue Gruppe: user
    user: ["Bob"]

  Verarbeite: Charlie (admin)
    admin: ["Alice","Charlie"]

  Verarbeite: David (user)
    user: ["Bob","David"]

  Verarbeite: Eve (moderator)
    Neue Gruppe: moderator
    moderator: ["Eve"]

✅ Gruppiert: {"admin":["Alice","Charlie"],"user":
["Bob","David"],"moderator":["Eve"]}
--- Ende Beispiel 3 ---
```

some() - Mindestens eines:

```
1  console.log("=== Beispiel 4: some() ===");
2
3  const numbers = [1, 3, 5, 7, 10, 11];
4  console.log("Zahlen:", numbers);
5
6  console.log("\nGibt es eine gerade Zahl?");
7  const hasEven = numbers.some(num => {
8    const isEven = num % 2 === 0;
9    console.log(`  ${num}: ${isEven ? "✅ gerade" : "ungerade"}`);
10   if (isEven) console.log("    → some() stoppt hier!");
11   return isEven;
12 });
13
14 console.log(`\nErgebnis: ${hasEven}`);
```

```
15
16 console.log("\nGibt es eine Zahl > 100?");
17 const hasLarge = numbers.some(num => num > 100);
18 console.log(`Ergebnis: ${hasLarge}`);
19 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: some() ===
Zahlen: [1,3,5,7,10,11]

Gibt es eine gerade Zahl?
  1: ungerade
  3: ungerade
  5: ungerade
  7: ungerade
  10: ✅ gerade
    → some() stoppt hier!

Ergebnis: true

Gibt es eine Zahl > 100?
Ergebnis: false
--- Ende Beispiel 4 ---
```

every() - Alle müssen zutreffen:

```
1  console.log("=== Beispiel 5: every() ===");
2
3  const ages = [21, 25, 30, 28, 19];
4  console.log("Alters-Daten:", ages);
5
6  console.log("\nSind alle volljährig (>= 18)?");
7  const allAdults = ages.every(age => {
8    const isAdult = age >= 18;
9    console.log(`  ${age}: ${isAdult ? "✅ volljährig" : "❌ minderjähr
10   if (!isAdult) console.log("    → every() stoppt hier!");
11   return isAdult;
12 });
13
14 console.log(`\nErgebnis: ${allAdults}`);
15
16 console.log("\nSind alle >= 21?");
17 const all21Plus = ages.every(age => age >= 21);
18 console.log(`Ergebnis: ${all21Plus} (19 ist < 21)`);
19 console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 5: every() ===
Alters-Daten: [21,25,30,28,19]

Sind alle volljährig (>= 18)?
  21: ✅ volljährig
  25: ✅ volljährig
  30: ✅ volljährig
  28: ✅ volljährig
  19: ✅ volljährig

Ergebnis: true

Sind alle >= 21?
Ergebnis: false (19 ist < 21)
--- Ende Beispiel 6 ---
```

Praxis: Validierung mit every():

```
1  console.log("=== Beispiel 6: Batch-Validierung ===");
2
3  const products = [
4    { id: 1, name: "Laptop", price: 999, stock: 5 },
5    { id: 2, name: "Maus", price: 29, stock: 150 },
6    { id: 3, name: "", price: 79, stock: 10 }, // ❌ Name fehlt
7    { id: 4, name: "Monitor", price: -50, stock: 20 } // ❌ Negativer P
8  ];
9
10 console.log("Produkte:", products);
11
12 console.log("\nValidiere alle Produkte:");
13 const allValid = products.every(product => {
14   console.log(`\n→ Prüfe: ${product.name || "(leer)"}`);
15
16   if (!product.name) {
17     console.log("  ❌ Name fehlt");
18     return false;
19   }
20
21   if (product.price <= 0) {
22     console.log("  ❌ Ungültiger Preis");
23     return false;
24   }
25
26   console.log("  ✅ Valide");
27   return true;
28 });
29
```

```
30 console.log(`\n${allValid ? "✅ Alle valide" : "❌ Validierung
     fehlgeschlagen"}`);
31 console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Batch-Validierung ===
Produkte: [{"id":1,"name":"Laptop","price":999,"stock":5},
{"id":2,"name":"Maus","price":29,"stock":150},
{"id":3,"name":"","price":79,"stock":10},
{"id":4,"name":"Monitor","price":-50,"stock":20}]

Validiere alle Produkte:

→ Prüfe: Laptop
  ✅ Valide

→ Prüfe: Maus
  ✅ Valide

→ Prüfe: (leer)
  ❌ Name fehlt

❌ Validierung fehlgeschlagen
--- Ende Beispiel 6 ---
```

4.9 Array Destructuring & Spread Operator

Wie bei Objekten gibt es auch bei Arrays Destructuring – extrahieren von Werten basierend auf Position. Der Spread Operator ist extrem nützlich zum Kopieren, Kombinieren und als Funktions-Argumente. Diese modernen Syntax-Features machen Ihren Code kürzer und lesbarer.

Array Destructuring:

```
 1 console.log("=== Beispiel 1: Array Destructuring ===");
 2
 3 const colors = ["rot", "grün", "blau", "gelb"];
 4 console.log("Colors:", colors);
 5
 6 console.log("\nOhne Destructuring:");
 7 const first1 = colors[0];
 8 const second1 = colors[1];
 9 console.log(`  Erste: ${first1}, Zweite: ${second1}`);
10
11 console.log("\nMit Destructuring:");
12 const [first, second, third] = colors;
13 console.log(`  Erste: ${first}`);
```

```
14 console.log(`  Zweite: ${second}`);
15 console.log(`  Dritte: ${third}`);
16
17 console.log("\nÜberspringen mit Kommas:");
18 const [, , blue] = colors; // Erste 2 überspringen
19 console.log(`  Dritte (blau): ${blue}`);
20 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Array Destructuring ===
Colors: ["rot","grün","blau","gelb"]

Ohne Destructuring:
  Erste: rot, Zweite: grün

Mit Destructuring:
  Erste: rot
  Zweite: grün
  Dritte: blau

Überspringen mit Kommas:
  Dritte (blau): blau
--- Ende Beispiel 1 ---
```

Rest Operator in Arrays:

```
1  console.log("=== Beispiel 2: Rest Operator ===");
2
3  const numbers = [1, 2, 3, 4, 5, 6];
4  console.log("Numbers:", numbers);
5
6  console.log("\nErstes + Rest:");
7  const [first, ...rest] = numbers;
8  console.log(`  Erstes: ${first}`);
9  console.log(`  Rest:`, rest);
10
11 console.log("\nErste 3 + Rest:");
12 const [a, b, c, ...remaining] = numbers;
13 console.log(`  a: ${a}, b: ${b}, c: ${c}`);
14 console.log(`  Remaining:`, remaining);
15 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Rest Operator ===
Numbers: [1,2,3,4,5,6]

Erstes + Rest:
  Erstes: 1
  Rest: [2,3,4,5,6]

Erste 3 + Rest:
  a: 1, b: 2, c: 3
  Remaining: [4,5,6]
--- Ende Beispiel 2 ---
```

Spread Operator - Kombinieren:

```
1  console.log("=== Beispiel 3: Arrays kombinieren ===");
2
3  const fruits = ["🍎", "🍌"];
4  const vegetables = ["🥕", "🥦"];
5  const dairy = ["🥛", "🧀"];
6
7  console.log("Fruits:", fruits);
8  console.log("Vegetables:", vegetables);
9  console.log("Dairy:", dairy);
10
11 console.log("\nKombiniere mit Spread:");
12 const allFood = [...fruits, ...vegetables, ...dairy];
13 console.log("All Food:", allFood);
14
15 console.log("\nMit zusätzlichen Elementen:");
16 const shopping = ["🍞", ...fruits, "🥩", ...vegetables];
17 console.log("Shopping:", shopping);
18 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Arrays kombinieren ===
Fruits: ["🍎","🍌"]
Vegetables: ["🥕","🥦"]
Dairy: ["🥛","🧀"]

Kombiniere mit Spread:
All Food: ["🍎","🍌","🥕","🥦","🥛","🧀"]

Mit zusätzlichen Elementen:
Shopping: ["🍞","🍎","🍌","🥩","🥕","🥦"]
--- Ende Beispiel 3 ---
```

Spread Operator - Kopieren (shallow):

```
console.log("=== Beispiel 4: Shallow Copy ===");

const original = [1, 2, 3];
console.log("Original:", original);

const copy = [...original];
console.log("Copy:", copy);

console.log("\nÄndere Copy:");
copy.push(4);
console.log("  Copy nach push:", copy);
console.log("  Original:", original);
console.log("  → Original unverändert!");

console.log("\n⚠️ Bei verschachtelten Arrays/Objekten:");
const nested = [[1, 2], [3, 4]];
const nestedCopy = [...nested];

console.log("Nested:", nested);
console.log("Nested Copy:", nestedCopy);

nestedCopy[0].push(999);
console.log("\nNach nestedCopy[0].push(999):");
console.log("  Nested:", nested); // ❌ Auch geändert!
console.log("  → Innere Arrays sind referenziert!");
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Shallow Copy ===
Original: [1,2,3]
Copy: [1,2,3]

Ändere Copy:
  Copy nach push: [1,2,3,4]
  Original: [1,2,3]
  → Original unverändert!

⚠️ Bei verschachtelten Arrays/Objekten:
Nested: [[1,2],[3,4]]
Nested Copy: [[1,2],[3,4]]

Nach nestedCopy[0].push(999):
  Nested: [[1,2,999],[3,4]]
  → Innere Arrays sind referenziert!
--- Ende Beispiel 4 ---
```

Spread als Funktions-Argumente:

```
console.log("=== Beispiel 5: Spread in Funktionen ===");

const numbers = [5, 12, 3, 18, 7];
console.log("Numbers:", numbers);

console.log("\nMath.max() braucht einzelne Argumente:");
// Math.max([5, 12, 3, 18, 7]) // ❌ Funktioniert nicht
// Math.max(5, 12, 3, 18, 7) // ✅ So muss es sein

const max = Math.max(...numbers); // Spread!
console.log(`  Max: ${max}`);

const min = Math.min(...numbers);
console.log(`  Min: ${min}`);

console.log("\nEigene Funktion:");
const sum = (...nums) => {
  console.log("  Erhaltene Args:", nums);
  return nums.reduce((a, b) => a + b, 0);
};

console.log("  Sum:", sum(...numbers));
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Spread in Funktionen ===
Numbers: [5,12,3,18,7]

Math.max() braucht einzelne Argumente:
  Max: 18
  Min: 3

Eigene Funktion:
  Erhaltene Args: [5,12,3,18,7]
  Sum: 45
--- Ende Beispiel 5 ---
```

Praxis: DB-Results erweitern:

```
console.log("=== Beispiel 6: Results erweitern ===");

const existingUsers = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" }
];

const newUsers = [
```

```
const newUsers = [
  { id: 3, name: "Charlie" },
  { id: 4, name: "David" }
];

console.log("Existing Users:", existingUsers);
console.log("New Users:", newUsers);

console.log("\nKombiniere Results:");
const allUsers = [...existingUsers, ...newUsers];
console.log("All Users:", allUsers);

console.log("\nFüge Meta-Data hinzu:");
const withMeta = allUsers.map(user => ({
  ...user,
  fetchedAt: new Date().toISOString(),
  source: "database"
}));

console.log("With Meta:", withMeta[0]);
console.log("--- Ende Beispiel 6 ---");
```

```
=== Beispiel 6: Results erweitern ===
Existing Users: [{"id":1,"name":"Alice"},{"id":2,"name":"Bob"}]
New Users: [{"id":3,"name":"Charlie"},{"id":4,"name":"David"}]

Kombiniere Results:
All Users: [{"id":1,"name":"Alice"},{"id":2,"name":"Bob"},
{"id":3,"name":"Charlie"},{"id":4,"name":"David"}]

Füge Meta-Data hinzu:
With Meta: {"id":1,"name":"Alice","fetchedAt":"2026-02-
13T10:54:55.677Z","source":"database"}
--- Ende Beispiel 6 ---
```

4.10 Übung: Objekte & Arrays

Das ist DAS zentrale Kapitel! Diese Übungen simulieren echte Datenbank-Szenarien. Nutzen Sie map, filter, find, reduce und console.log ausgiebig. Wenn Sie diese Aufgaben beherrschen, können Sie jede DB-Query in JavaScript verarbeiten!

Aufgabe 1: Daten extrahieren (map)

```
// Gegeben sind User-Daten:
const users = [
  { id: 1, firstName: "Alice", lastName: "Smith", age: 28 },
  { id: 2, firstName: "Bob", lastName: "Jones", age: 34 },
```

```
  { id: 3, firstName: "Charlie", lastName: "Brown", age: 22 }
];

// Aufgabe: Erstellen Sie ein Array mit vollständigen Namen
// Ergebnis: ["Alice Smith", "Bob Jones", "Charlie Brown"]

console.log("=== Aufgabe 1: Namen extrahieren ===");
console.log("Users:", users);

// Ihr Code hier:
// const fullNames = ...

// console.log("Full Names:", fullNames);
```

```
=== Aufgabe 1: Namen extrahieren ===
Users: [{"id":1,"firstName":"Alice","lastName":"Smith","age":28},
{"id":2,"firstName":"Bob","lastName":"Jones","age":34},
{"id":3,"firstName":"Charlie","lastName":"Brown","age":22}]
```

Aufgabe 2: Filtern (filter)

```
// Gegeben sind Produkte:
const products = [
  { id: 1, name: "Laptop", price: 999, category: "electronics", inSto
    true },
  { id: 2, name: "Maus", price: 29, category: "electronics", inStock:
    },
  { id: 3, name: "Stuhl", price: 199, category: "furniture", inStock:
    },
  { id: 4, name: "Monitor", price: 299, category: "electronics", inSt
    true },
  { id: 5, name: "Tisch", price: 399, category: "furniture", inStock:
    }
];

// Aufgabe: Finden Sie alle Elektronik-Produkte, die:
// - in stock sind
// - Preis >= 200
// Geben Sie nur die Namen aus

console.log("=== Aufgabe 2: Filtern ===");
console.log("Produkte:", products);

// Ihr Code hier:
```

```
=== Aufgabe 2: Filtern ===
Produkte:
[{"id":1,"name":"Laptop","price":999,"category":"electronics","inStock":tr
{"id":2,"name":"Maus","price":29,"category":"electronics","inStock":false
{"id":3,"name":"Stuhl","price":199,"category":"furniture","inStock":true},
{"id":4,"name":"Monitor","price":299,"category":"electronics","inStock":tr
{"id":5,"name":"Tisch","price":399,"category":"furniture","inStock":false
```

Aufgabe 3: Summen berechnen (reduce)

```javascript
// Gegeben sind Bestellungen:
const orders = [
  { id: 101, customer: "Alice", items: [
    { product: "Laptop", price: 999, quantity: 1 },
    { product: "Maus", price: 29, quantity: 2 }
  ]},
  { id: 102, customer: "Bob", items: [
    { product: "Monitor", price: 299, quantity: 1 }
  ]},
  { id: 103, customer: "Charlie", items: [
    { product: "Tastatur", price: 79, quantity: 1 },
    { product: "Maus", price: 29, quantity: 1 }
  ]}
];

// Aufgabe: Berechnen Sie den Gesamtumsatz aller Bestellungen
// (Summe aller: price × quantity)
// Nutzen Sie reduce (eventuell verschachtelt)

console.log("=== Aufgabe 3: Gesamtumsatz ===");
console.log("Orders:", orders);

// Ihr Code hier:
```

```
=== Aufgabe 3: Gesamtumsatz ===
Orders: [{"id":101,"customer":"Alice","items":
[{"product":"Laptop","price":999,"quantity":1},
{"product":"Maus","price":29,"quantity":2}]},
{"id":102,"customer":"Bob","items":
[{"product":"Monitor","price":299,"quantity":1}]},
{"id":103,"customer":"Charlie","items":
[{"product":"Tastatur","price":79,"quantity":1},
{"product":"Maus","price":29,"quantity":1}]}]
```

Aufgabe 4: Group By (reduce + Objekt aufbauen)

```
// Gegeben sind Transactions:
const transactions = [
  { id: 1, type: "income", amount: 1000, category: "salary" },
  { id: 2, type: "expense", amount: 50, category: "food" },
  { id: 3, type: "income", amount: 200, category: "freelance" },
  { id: 4, type: "expense", amount: 100, category: "transport" },
  { id: 5, type: "expense", amount: 30, category: "food" },
  { id: 6, type: "income", amount: 500, category: "salary" }
];

// Aufgabe: Gruppieren Sie nach 'type' und berechnen Sie die Summe pr
  Gruppe
// Ergebnis: { income: 1700, expense: 180 }

console.log("=== Aufgabe 4: Group By Type ===");
console.log("Transactions:", transactions);

// Ihr Code hier:
```

```
=== Aufgabe 4: Group By Type ===
Transactions:
[{"id":1,"type":"income","amount":1000,"category":"salary"},
{"id":2,"type":"expense","amount":50,"category":"food"},
{"id":3,"type":"income","amount":200,"category":"freelance"},
{"id":4,"type":"expense","amount":100,"category":"transport"},
{"id":5,"type":"expense","amount":30,"category":"food"},
{"id":6,"type":"income","amount":500,"category":"salary"}]
```

Aufgabe 5: Chain-Operations (map + filter + reduce)

```
// Gegeben sind Students mit Noten:
const students = [
  { name: "Alice", grades: [85, 90, 92, 88] },
  { name: "Bob", grades: [70, 75, 72] },
  { name: "Charlie", grades: [95, 98, 97, 99, 94] },
  { name: "David", grades: [60, 65, 58] }
];

// Aufgabe (mehrstufig):
// 1. Berechnen Sie für jeden Student den Durchschnitt
// 2. Filtern Sie nur Students mit Durchschnitt >= 80
// 3. Extrahieren Sie nur die Namen
// Ergebnis: ["Alice", "Charlie"]

console.log("=== Aufgabe 5: Chain Operations ===");
console.log("Students:", students);

// Ihr Code hier:
```

```
=== Aufgabe 5: Chain Operations ===
Students: [{"name":"Alice","grades":[85,90,92,88]},
{"name":"Bob","grades":[70,75,72]},{"name":"Charlie","grades":
[95,98,97,99,94]},{"name":"David","grades":[60,65,58]}]
```

Diese Übungen sind anspruchsvoll, aber sie spiegeln echte Datenbank-Arbeit wider! Bei Schwierigkeiten: Arbeiten Sie Schritt für Schritt, nutzen Sie console.log nach jedem Schritt, schauen Sie sich die Beispiele aus den vorherigen Abschnitten an. Map, filter und reduce sind Ihre wichtigsten Werkzeuge!

Lösungen:

- ▶ Lösung Aufgabe 1 (map - Namen extrahieren)
- ▶ Lösung Aufgabe 2 (filter - Elektronik)
- ▶ Lösung Aufgabe 3 (reduce - Gesamtumsatz)
- ▶ Lösung Aufgabe 4 (Group By)
- ▶ Lösung Aufgabe 5 (Chain Operations)

Exzellent! Sie haben jetzt die wichtigsten Werkzeuge für Datenverarbeitung gemeistert. Objekte speichern strukturierte Daten, Arrays organisieren Sammlungen, und die Array-Methoden map, filter, reduce sind Ihre täglichen Begleiter in der Datenbank-Programmierung. Dieses Wissen ist fundamental – jede weitere Arbeit mit Datenbanken baut darauf auf!

Kapitel 5: Asynchronität – Der Event Loop

JavaScript ist singlethreaded – aber asynchron! Das ist FUNDAMENTAL für Datenbank-Operationen. Jede DB-Abfrage dauert einige Millisekunden bis Sekunden. Ohne Asynchronität würde Ihre App währenddessen einfrieren. Dieses Kapitel erklärt, wie JavaScript mit Wartezeiten umgeht und wie Sie modernen async/await-Code schreiben.

5.1 Blockierender vs. asynchroner Code

Der Unterschied zwischen synchronem und asynchronem Code ist entscheidend. Synchron bedeutet: Warten, bis eine Operation fertig ist. Asynchron bedeutet: Weitermachen und später das Ergebnis verarbeiten.

=== Beispiel 1: Synchroner Code (blockierend) ===

```
1  console.log("=== Beispiel 1: Synchroner Code ===");
2
3  console.log("1. Start");
4
5  // Simuliere langsame Berechnung (blockierend)
6  function slowCalculation() {
7    console.log("   → Berechnung startet...");
8    const start = Date.now();
9    // Warte 2 Sekunden (blockiert!)
```

```
  while (Date.now() - start < 2000) {
    // Busy waiting - BAD PRACTICE!
  }
  console.log("  → Berechnung fertig (nach 2s)");
  return 42;
}

const result = slowCalculation();
console.log("2. Ergebnis:", result);
console.log("3. Ende");

console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Synchroner Code ===
1. Start
  → Berechnung startet...
  → Berechnung fertig (nach 2s)
2. Ergebnis: 42
3. Ende
--- Ende Beispiel 1 ---
```

Hier wird der gesamte Thread blockiert! Die Zeile „3. Ende“ erscheint erst nach 2 Sekunden. Im Browser würde die UI einfrieren. Das ist problematisch.

=== Beispiel 2: Asynchroner Code (nicht-blockierend) ===

```
console.log("=== Beispiel 2: Asynchroner Code ===");

console.log("1. Start");

// setTimeout ist asynchron
setTimeout(() => {
  console.log("  → Callback nach 2 Sekunden");
  console.log("  → Ergebnis: 42");
}, 2000);

console.log("2. Weiter (ohne zu warten)");
console.log("3. Ende");

console.log("--- Ende Beispiel 2 ---");
// Achtung: Der Callback kommt NACH "Ende"!
```

```
=== Beispiel 2: Asynchroner Code ===
1. Start
2. Weiter (ohne zu warten)
3. Ende
--- Ende Beispiel 2 ---
  → Callback nach 2 Sekunden
  → Ergebnis: 42
```

Jetzt läuft der Code sofort weiter! Die Ausgabe ist: 1, 2, 3, Ende, dann nach 2 Sekunden der Callback. Das ist asynchron: Der Code wartet nicht, sondern registriert einen Callback, der später ausgeführt wird.
=== Beispiel 3: Mehrere asynchrone Operationen ===

```
1  console.log("=== Beispiel 3: Mehrere async Ops ===");
2
3  console.log("Start");
4
5  setTimeout(() => {
6    console.log("  → Timeout 1 (1000ms)");
7  }, 1000);
8
9  setTimeout(() => {
10   console.log("  → Timeout 2 (500ms)");
11 }, 500);
12
13 setTimeout(() => {
14   console.log("  → Timeout 3 (1500ms)");
15 }, 1500);
16
17 console.log("Alle Timeouts registriert");
18 console.log("--- Code läuft weiter ---");
```

```
=== Beispiel 3: Mehrere async Ops ===
Start
Alle Timeouts registriert
--- Code läuft weiter ---
  → Timeout 2 (500ms)
  → Timeout 1 (1000ms)
  → Timeout 3 (1500ms)
```

Die Ausgabe zeigt: Erst die synchronen Zeilen, dann die Callbacks in der Reihenfolge ihrer Verzögerung (500ms, 1000ms, 1500ms). JavaScript merkt sich die Callbacks und ruft sie zur richtigen Zeit auf.

5.2 Event Loop (Konzept)

Der Event Loop ist das Herzstück von JavaScript. Er sorgt dafür, dass asynchrone Operationen funktionieren, obwohl JavaScript single-threaded ist. Verstehen Sie dieses Konzept, und Sie verstehen, warum Ihr Code manchmal „in falscher Reihenfolge“ läuft!

=== Beispiel 1: Call Stack vs. Callback Queue ===

```
console.log("=== Beispiel 1: Event Loop Visualisierung ===");

console.log("1. Synchroner Code (Call Stack)");

setTimeout(() => {
  console.log("3. Callback aus Timeout (Callback Queue → Call Stack)
}, 0); // 0ms Verzögerung!

console.log("2. Noch synchroner Code");

console.log("--- Ende Beispiel 1 ---");
// Warum ist "3." am Ende, obwohl timeout 0ms?
```

```
=== Beispiel 1: Event Loop Visualisierung ===
1. Synchroner Code (Call Stack)
2. Noch synchroner Code
--- Ende Beispiel 1 ---
3. Callback aus Timeout (Callback Queue → Call Stack)
```

Obwohl setTimeout 0ms hat, kommt der Callback NACH dem synchronen Code! Warum? Der Event Loop funktioniert so: 1) Call Stack komplett abarbeiten (synchroner Code), 2) dann Callbacks aus der Queue holen. setTimeout legt den Callback in die Queue, aber die wird erst NACH dem synchronen Code geleert.

=== Beispiel 2: Event Loop-Phasen ===

```
console.log("=== Beispiel 2: Event Loop Phasen ===");

// Phase 1: Synchroner Code
console.log("→ Phase 1: Call Stack (sync)");

// Phase 2: Microtasks (Promises)
Promise.resolve().then(() => {
  console.log("→ Phase 2a: Microtask (Promise)");
});

// Phase 3: Macrotasks (setTimeout)
setTimeout(() => {
  console.log("→ Phase 3: Macrotask (setTimeout)");
}, 0);

// Phase 2 nochmal
Promise.resolve().then(() => {
```

```
  console.log("→ Phase 2b: Noch ein Microtask");
});

console.log("→ Phase 1: Letzter sync Code");
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Event Loop Phasen ===
→ Phase 1: Call Stack (sync)
→ Phase 1: Letzter sync Code
--- Ende Beispiel 2 ---
→ Phase 2a: Microtask (Promise)
→ Phase 2b: Noch ein Microtask
→ Phase 3: Macrotask (setTimeout)
```

Die Reihenfolge ist: 1) Synchroner Code komplett, 2) ALLE Microtasks (Promises), 3) DANN Macrotasks (setTimeout). Promises haben höhere Priorität als setTimeout! Das ist wichtig für Datenbank-Code, denn fast alle DB-Libraries nutzen Promises.

=== Beispiel 3: Visualisierung mit DB-Query ===

```
console.log("=== Beispiel 3: Simulierte DB-Query ===");

function queryDatabase(id) {
  console.log(`   → DB-Query gestartet für ID ${id}`);
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`   → DB antwortet für ID ${id}`);
      resolve({ id, name: `User ${id}` });
    }, 1000);
  });
}

console.log("1. App startet");
console.log("2. Query wird abgeschickt");

queryDatabase(42).then((user) => {
  console.log("4. Daten empfangen:", user);
});

console.log("3. App läuft weiter (ohne zu blockieren)");
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Simulierte DB-Query ===
1. App startet
2. Query wird abgeschickt
   → DB-Query gestartet für ID 42
3. App läuft weiter (ohne zu blockieren)
--- Ende Beispiel 3 ---
   → DB antwortet für ID 42
4. Daten empfangen: {"id":42,"name":"User 42"}
```

Perfekt! Die App startet die Query, läuft SOFORT weiter und verarbeitet das Ergebnis, wenn es ankommt. So funktionieren alle modernen Datenbank-Operationen: async, non-blocking, mit Promises.

5.3 Callbacks (alte Methode)

Callbacks waren die erste Methode für asynchronen Code. Sie funktionieren, führen aber zu verschachteltem „Callback Hell“. Moderne Projekte nutzen Promises und async/await, aber Sie sollten Callbacks verstehen, weil Sie ihnen noch begegnen werden.

=== Beispiel 1: Einfacher Callback ===

```
1  console.log("=== Beispiel 1: Einfacher Callback ===");
2
3  function fetchUser(id, callback) {
4    console.log(`   → Lade User ${id}...`);
5    setTimeout(() => {
6      const user = { id, name: `User ${id}` };
7      console.log(`   → User geladen:`, user);
8      callback(user); // Callback aufrufen
9    }, 1000);
10 }
11
12 console.log("Start");
13
14 fetchUser(1, (user) => {
15   console.log("Callback erhält:", user);
16   console.log(`Willkommen, ${user.name}!`);
17 });
18
19 console.log("Weiter...");
20 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Einfacher Callback ===
Start
  → Lade User 1...
Weiter...
--- Ende Beispiel 1 ---
  → User geladen: {"id":1,"name":"User 1"}
Callback erhält: {"id":1,"name":"User 1"}
Willkommen, User 1!
```

Der Callback wird als Funktion übergeben und später aufgerufen, wenn die Daten da sind. Das Prinzip ist einfach, aber...

=== Beispiel 2: Callback Hell (verschachtelt) ===

```
1  console.log("=== Beispiel 2: Callback Hell ===");
2
3  function getUser(id, callback) {
4    setTimeout(() => callback({ id, name: `User ${id}` }), 500);
5  }
6
7  function getOrders(userId, callback) {
8    setTimeout(() => callback([
9      { orderId: 101, total: 99 },
10     { orderId: 102, total: 149 }
11   ]), 500);
12 }
13
14 function getOrderDetails(orderId, callback) {
15   setTimeout(() => callback({ orderId, items: 3 }), 500);
16 }
17
18 console.log("Start verschachtelte Callbacks:");
19
20 getUser(1, (user) => {
21   console.log("1. User:", user);
22   getOrders(user.id, (orders) => {
23     console.log("2. Orders:", orders);
24     getOrderDetails(orders[0].orderId, (details) => {
25       console.log("3. Details:", details);
26       console.log("Fertig (nach 1.5s)");
27     });
28   });
29 });
30
31 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Callback Hell ===
Start verschachtelte Callbacks:
--- Ende Beispiel 2 ---
1. User: {"id":1,"name":"User 1"}
2. Orders: [{"orderId":101,"total":99},{"orderId":102,"total":149}]
3. Details: {"orderId":101,"items":3}
Fertig (nach 1.5s)
```

Das ist „Callback Hell“ oder „Pyramid of Doom“! Jede asynchrone Operation ist eine Ebene tiefer verschachtelt. Das wird schnell unleserlich und fehleranfällig. Deshalb wurden Promises erfunden.

=== Beispiel 3: Error Handling mit Callbacks ===

```
console.log("=== Beispiel 3: Error Handling ===");

function fetchData(id, callback) {
  console.log(`   → Lade Daten für ID ${id}...`);
  setTimeout(() => {
    if (id < 0) {
      // Error als erstes Argument (Node.js-Konvention)
      callback(new Error("Ungültige ID"), null);
    } else {
      callback(null, { id, data: "Erfolg" });
    }
  }, 500);
}

fetchData(5, (error, data) => {
  if (error) {
    console.log("❌ Fehler:", error.message);
  } else {
    console.log("✅ Daten:", data);
  }
});

fetchData(-1, (error, data) => {
  if (error) {
    console.log("❌ Fehler:", error.message);
  } else {
    console.log("✅ Daten:", data);
  }
});

console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Error Handling ===
  → Lade Daten für ID 5...
  → Lade Daten für ID -1...
--- Ende Beispiel 3 ---
✅ Daten: {"id":5,"data":"Erfolg"}
❌ Fehler: Ungültige ID
```

Die Node.js-Konvention ist: Erster Parameter ist Error (oder null), zweiter ist Data. Sie müssen bei JEDEM Callback prüfen, ob ein Fehler vorliegt. Das ist fehleranfällig und mühsam. Promises machen das eleganter!

5.4 Promises (moderne Methode)

Promises sind das moderne Gegenstück zu Callbacks. Ein Promise ist ein Objekt, das einen zukünftigen Wert repräsentiert. Promises können „pending“, „fulfilled“ (Erfolg) oder „rejected“ (Fehler) sein. Sie verketten sich elegant mit .then() und .catch().

=== Beispiel 1: Promise erstellen ===

```
1  console.log("=== Beispiel 1: Promise erstellen ===");
2
3  const myPromise = new Promise((resolve, reject) => {
4    console.log("  → Promise wird ausgeführt (sofort!)");
5    setTimeout(() => {
6      const success = true;
7      if (success) {
8        resolve("Erfolg! Hier sind die Daten.");
9      } else {
10       reject("Fehler! Etwas ist schief gelaufen.");
11     }
12   }, 1000);
13 });
14
15 console.log("Promise erstellt:", myPromise);
16 console.log("Status: pending (wartend)");
17
18 myPromise.then((data) => {
19   console.log("✅ Promise fulfilled:", data);
20 });
21
22 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Promise erstellen ===
  → Promise wird ausgeführt (sofort!)
Promise erstellt: {}
Status: pending (wartend)
--- Ende Beispiel 1 ---
✅ Promise fulfilled: Erfolg! Hier sind die Daten.
```

Ein Promise wird sofort ausgeführt (der Code im Constructor läuft synchron). Das Ergebnis kommt später. Mit .then() registrieren wir einen Handler für den Erfolgsfall.

=== Beispiel 2: Promise-Verkettung (Chaining) ===

```
1  console.log("=== Beispiel 2: Promise Chaining ===");
2
3  function getUser(id) {
4    return new Promise((resolve) => {
5      setTimeout(() => {
6        console.log(`   → User ${id} geladen`);
7        resolve({ id, name: `User ${id}` });
8      }, 500);
9    });
10 }
11
12 function getOrders(userId) {
13   return new Promise((resolve) => {
14     setTimeout(() => {
15       console.log(`   → Orders für User ${userId} geladen`);
16       resolve([{ orderId: 101, total: 99 }]);
17     }, 500);
18   });
19 }
20
21 console.log("Start Chaining:");
22
23 getUser(1)
24   .then((user) => {
25     console.log("1. User:", user);
26     return getOrders(user.id); // Nächstes Promise!
27   })
28   .then((orders) => {
29     console.log("2. Orders:", orders);
30     console.log("Fertig!");
31   });
32
33 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Promise Chaining ===
Start Chaining:
--- Ende Beispiel 2 ---
  → User 1 geladen
1. User: {"id":1,"name":"User 1"}
  → Orders für User 1 geladen
2. Orders: [{"orderId":101,"total":99}]
Fertig!
```

Kein Callback Hell mehr! Promises verketten sich flach mit .then(). Jedes .then() kann ein neues Promise zurückgeben, und die Kette läuft automatisch weiter. Das ist deutlich lesbarer.

=== Beispiel 3: Error Handling mit .catch() ===

```
1  console.log("=== Beispiel 3: Promise Error Handling ===");
2
3  function fetchData(id) {
4    return new Promise((resolve, reject) => {
5      setTimeout(() => {
6        if (id < 0) {
7          reject(new Error(`Ungültige ID: ${id}`));
8        } else {
9          resolve({ id, data: "Erfolg" });
10       }
11     }, 500);
12   });
13 }
14
15 console.log("Test 1: Gültige ID");
16 fetchData(5)
17   .then((data) => {
18     console.log("  ✅ Daten:", data);
19   })
20   .catch((error) => {
21     console.log("  ❌ Fehler:", error.message);
22   });
23
24 console.log("\nTest 2: Ungültige ID");
25 fetchData(-1)
26   .then((data) => {
27     console.log("  ✅ Daten:", data);
28   })
29   .catch((error) => {
30     console.log("  ❌ Fehler:", error.message);
31   });
32
33 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Promise Error Handling ===
Test 1: Gültige ID

Test 2: Ungültige ID
--- Ende Beispiel 3 ---
  ✅ Daten: {"id":5,"data":"Erfolg"}
  ❌ Fehler: Ungültige ID: -1
```

Mit .catch() fangen Sie alle Fehler in der Promise-Kette ab. Sie müssen nicht bei jedem .then() prüfen – ein .catch() am Ende reicht. Das ist eleganter und weniger fehleranfällig als Callbacks!

=== Beispiel 4: Promise.all() (parallel) ===

```
1  console.log("=== Beispiel 4: Promise.all() ===");
2
3  function fetchUser(id) {
4    return new Promise((resolve) => {
5      setTimeout(() => {
6        console.log(`   → User ${id} geladen`);
7        resolve({ id, name: `User ${id}` });
8      }, Math.random() * 1000);
9    });
10 }
11
12 console.log("Lade 3 Users parallel:");
13
14 const promises = [
15   fetchUser(1),
16   fetchUser(2),
17   fetchUser(3)
18 ];
19
20 Promise.all(promises).then((users) => {
21   console.log("\n✅ Alle Users geladen:");
22   users.forEach(u => console.log(`   ${u.id}: ${u.name}`));
23 });
24
25 console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Promise.all() ===
Lade 3 Users parallel:
--- Ende Beispiel 4 ---
  → User 1 geladen
  → User 2 geladen
  → User 3 geladen

✅ Alle Users geladen:
  1: User 1
  2: User 2
  3: User 3
```

Promise.all() startet mehrere Promises parallel und wartet, bis ALLE fertig sind. Das ist perfekt, wenn Sie mehrere Datenbank-Queries parallel ausführen wollen. Wenn ein Promise fehlschlägt, schlägt das ganze Promise.all() fehl.

=== Beispiel 5: Promise.race() (erster gewinnt) ===

```
1  console.log("=== Beispiel 5: Promise.race() ===");
2
3  function slowServer() {
4    return new Promise((resolve) => {
5      setTimeout(() => {
6        console.log("  → Slow Server antwortet (2s)");
7        resolve("Slow Data");
8      }, 2000);
9    });
10 }
11
12 function fastServer() {
13   return new Promise((resolve) => {
14     setTimeout(() => {
15       console.log("  → Fast Server antwortet (500ms)");
16       resolve("Fast Data");
17     }, 500);
18   });
19 }
20
21 console.log("Race zwischen zwei Servern:");
22
23 Promise.race([slowServer(), fastServer()]).then((result) => {
24   console.log("✅ Erster Antwort:", result);
25 });
26
27 console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Promise.race() ===
Race zwischen zwei Servern:
--- Ende Beispiel 5 ---
  → Fast Server antwortet (500ms)
✅ Erster Antwort: Fast Data
  → Slow Server antwortet (2s)
```

Promise.race() nimmt das Ergebnis des ERSTEN Promises, das fertig wird. Nützlich für Timeouts: Starten Sie eine Query und ein Timeout-Promise parallel, und wenn das Timeout zuerst fertig ist, brechen Sie ab.

5.5 async/await (modernste Methode)

async/await ist syntaktischer Zucker über Promises. Es macht asynchronen Code aussehen wie synchronen Code – aber ohne zu blockieren! Das ist der Standard für moderne JavaScript-Entwicklung und das, was Sie in Ihrem Datenbank-Code verwenden werden.

=== Beispiel 1: async-Funktion basics ===

```
1  console.log("=== Beispiel 1: async/await basics ===");
2
3  // async-Funktion gibt IMMER ein Promise zurück
4  async function fetchData() {
5    console.log("  → Funktion startet");
6    return "Daten"; // Wird automatisch in Promise.resolve("Daten") gew
7  }
8
9  console.log("Vor Aufruf");
10 const promise = fetchData();
11 console.log("Promise:", promise);
12
13 promise.then((data) => {
14   console.log("Daten erhalten:", data);
15 });
16
17 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: async/await basics ===
Vor Aufruf
  → Funktion startet
Promise: {}
--- Ende Beispiel 1 ---
Daten erhalten: Daten
```

Jede Funktion mit async davor gibt automatisch ein Promise zurück. Das return-Statement wird zu resolve(). Das ist die Basis von async/await.

=== Beispiel 2: await (auf Promise warten) ===

```
console.log("=== Beispiel 2: await ===");

function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function demo() {
  console.log("  → Start");

  console.log("  → Warte 1 Sekunde...");
  await delay(1000); // Wartet auf Promise
  console.log("  → 1 Sekunde vorbei");

  console.log("  → Warte noch 1 Sekunde...");
  await delay(1000);
  console.log("  → 2 Sekunden vorbei");

  return "Fertig!";
}

console.log("Vor Aufruf");
demo().then(result => console.log("Ergebnis:", result));
console.log("Nach Aufruf (läuft weiter!)");
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: await ===
Vor Aufruf
  → Start
  → Warte 1 Sekunde...
Nach Aufruf (läuft weiter!)
--- Ende Beispiel 2 ---
  → 1 Sekunde vorbei
  → Warte noch 1 Sekunde...
  → 2 Sekunden vorbei
Ergebnis: Fertig!
```

await pausiert die Funktion, bis das Promise fertig ist – OHNE den Thread zu blockieren! Der Code danach läuft sofort weiter („Nach Aufruf“). Innerhalb von demo() sieht der Code synchron aus, ist aber asynchron.

=== Beispiel 3: DB-Query mit async/await ===

```
console.log("=== Beispiel 3: DB-Query ===");

function queryDB(sql) {
```

```
function queryDB(sql) {
  return new Promise((resolve) => {
    console.log(`   → Query: ${sql}`);
    setTimeout(() => {
      resolve([{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }]);
    }, 1000);
  });
}

async function getUsers() {
  console.log("→ Lade Users...");
  const users = await queryDB("SELECT * FROM users");
  console.log("→ Users geladen:", users.length);
  return users;
}

console.log("Start");
getUsers().then((users) => {
  console.log("✅ Ergebnis:");
  users.forEach(u => console.log(`  ${u.id}: ${u.name}`));
});
console.log("Weiter...");
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: DB-Query ===
Start
→ Lade Users...
  → Query: SELECT * FROM users
Weiter...
--- Ende Beispiel 3 ---
→ Users geladen: 2
✅ Ergebnis:
  1: Alice
  2: Bob
```

Perfekt! Der Code in getUsers() sieht synchron aus: Zeile für Zeile. Aber er blockiert nicht! Das ist der große Vorteil von async/await: Lesbarkeit wie synchroner Code, Verhalten von asynchronem Code.

=== Beispiel 4: Sequentielle Operationen ===

```
console.log("=== Beispiel 4: Sequentiell ===");

function query(name, delay) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`   → ${name} fertig`);
      resolve(`Data from ${name}`);
    }, delay);
  });
```

```
}

async function sequential() {
  console.log("→ Start (sequentiell)");
  const start = Date.now();

  const a = await query("Query A", 1000); // Wartet 1s
  const b = await query("Query B", 1000); // Wartet 1s
  const c = await query("Query C", 1000); // Wartet 1s

  const duration = Date.now() - start;
  console.log(`✅ Fertig nach ${duration}ms`);
  return [a, b, c];
}

sequential().then(results => console.log("Results:", results));
console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Sequentiell ===
→ Start (sequentiell)
--- Ende Beispiel 4 ---
  → Query A fertig
  → Query B fertig
  → Query C fertig
✅ Fertig nach 3090ms
Results: ["Data from Query A","Data from Query B","Data from Query C"]
```

Achtung! Hier dauert es 3 Sekunden, weil jedes await nacheinander wartet. Wenn die Queries unabhängig sind, ist das ineffizient. Besser: parallel ausführen!

=== Beispiel 5: Parallele Operationen ===

```
console.log("=== Beispiel 5: Parallel ===");

function query(name, delay) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`   → ${name} fertig`);
      resolve(`Data from ${name}`);
    }, delay);
  });
}

async function parallel() {
  console.log("→ Start (parallel)");
  const start = Date.now();

  // Queries SOFORT starten (nicht awaiten!)
  const promiseA = query("Query A", 1000);
```

```
  const promiseB = query("Query B", 1000);
  const promiseC = query("Query C", 1000);

  // DANN auf alle warten
  const results = await Promise.all([promiseA, promiseB, promiseC]);

  const duration = Date.now() - start;
  console.log(`✅ Fertig nach ${duration}ms`);
  return results;
}

parallel().then(results => console.log("Results:", results));
console.log("--- Ende Beispiel 5 ---");
```

```
=== Beispiel 5: Parallel ===
→ Start (parallel)
--- Ende Beispiel 5 ---
  → Query A fertig
  → Query B fertig
  → Query C fertig
✅ Fertig nach 1000ms
Results: ["Data from Query A","Data from Query B","Data from Query C"]
```

Jetzt nur 1 Sekunde! Die Queries laufen parallel. Merken Sie sich: Promises SOFORT starten (ohne await), DANN mit Promise.all() auf alle warten. Das ist ein häufiges Pattern in Datenbank-Code.

5.6 Error Handling mit try/catch

Mit async/await können Sie try/catch nutzen – wie bei synchronem Code! Das ist ein großer Vorteil gegenüber Promise-Chaining mit .catch().

=== Beispiel 1: try/catch mit async/await ===

```
console.log("=== Beispiel 1: try/catch ===");

function riskyQuery(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (id < 0) {
        reject(new Error(`Ungültige ID: ${id}`));
      } else {
        resolve({ id, data: "Erfolg" });
      }
    }, 500);
  });
}

```

```
async function fetchData(id) {
  try {
    console.log(`→ Lade Daten für ID ${id}...`);
    const data = await riskyQuery(id);
    console.log("✅ Erfolg:", data);
    return data;
  } catch (error) {
    console.log("❌ Fehler gefangen:", error.message);
    return null;
  }
}

fetchData(5);
fetchData(-1);
console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: try/catch ===
→ Lade Daten für ID 5...
→ Lade Daten für ID -1...
--- Ende Beispiel 1 ---
✅ Erfolg: {"id":5,"data":"Erfolg"}
❌ Fehler gefangen: Ungültige ID: -1
```

Perfekt! try/catch funktioniert mit await wie mit normalem synchronem Code. Wenn das Promise rejected wird, fliegt eine Exception, die Sie mit catch abfangen.

=== Beispiel 2: Multiple Queries mit Error Handling ===

```
console.log("=== Beispiel 2: Multiple Queries ===");

function query(name, shouldFail = false) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldFail) {
        reject(new Error(`${name} failed`));
      } else {
        resolve(`Data from ${name}`);
      }
    }, 500);
  });
}

async function loadAll() {
  try {
    console.log("→ Lade User...");
    const user = await query("User");
    console.log("  ✅", user);

    console.log("→ Lade Orders...");
```

```
    const orders = await query("Orders", true); // Fails!
    console.log("  ✅", orders);

    console.log("→ Lade Profile...");
    const profile = await query("Profile");
    console.log("  ✅", profile);

  } catch (error) {
    console.log("❌ Ein Query ist fehlgeschlagen:", error.message);
    console.log("   Folgende Queries werden nicht ausgeführt!");
  }
}

loadAll();
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Multiple Queries ===
→ Lade User...
--- Ende Beispiel 2 ---
  ✅ Data from User
→ Lade Orders...
❌ Ein Query ist fehlgeschlagen: Orders failed
   Folgende Queries werden nicht ausgeführt!
```

Wichtig! Wenn ein await fehlschlägt, springt der Code sofort in den catch-Block. Die folgenden Queries (Profile) werden nicht ausgeführt. Das ist oft gewollt, aber manchmal wollen Sie weitermachen...

=== Beispiel 3: Einzelne Fehler abfangen ===

```
console.log("=== Beispiel 3: Einzelne Fehler ===");

function query(name, shouldFail = false) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldFail) {
        reject(new Error(`${name} failed`));
      } else {
        resolve(`Data from ${name}`);
      }
    }, 500);
  });
}

async function loadAllRobust() {
  let user, orders, profile;

  try {
    user = await query("User");
    console.log("✅ User:", user);
```

```
  } catch (error) {
    console.log("❌ User fehlgeschlagen:", error.message);
  }

  try {
    orders = await query("Orders", true); // Fails!
    console.log("✅ Orders:", orders);
  } catch (error) {
    console.log("❌ Orders fehlgeschlagen:", error.message);
  }

  try {
    profile = await query("Profile");
    console.log("✅ Profile:", profile);
  } catch (error) {
    console.log("❌ Profile fehlgeschlagen:", error.message);
  }

  console.log("\n→ Ergebnis:");
  console.log("  User:", user || "null");
  console.log("  Orders:", orders || "null");
  console.log("  Profile:", profile || "null");
}

loadAllRobust();
console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Einzelne Fehler ===
--- Ende Beispiel 3 ---
✅ User: Data from User
❌ Orders fehlgeschlagen: Orders failed
✅ Profile: Data from Profile

→ Ergebnis:
  User: Data from User
  Orders: null
  Profile: Data from Profile
```

Jetzt läuft alles durch! Jede Query hat ihr eigenes try/catch. Wenn eine fehlschlägt, geht es mit der nächsten weiter. Das ist robuster, wenn Sie unabhängige Queries haben.

5.7 Übung: Asynchronität

Jetzt sind Sie dran! Diese Übungen simulieren echte Datenbank-Szenarien mit async/await. Nutzen Sie try/catch für Fehler und Promise.all() für parallele Queries.

Aufgabe 1: Sequentielle DB-Queries

```
// Gegeben:
function getUser(id) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ id, name: `User ${id}`, roleId: id * 10 });
    }, 500);
  });
}

function getRole(roleId) {
  return new Promise((resolve) => {
    setTimeout(() => {
      const roles = { 10: "Admin", 20: "Editor", 30: "Viewer" };
      resolve({ roleId, name: roles[roleId] || "Unknown" });
    }, 500);
  });
}

// Aufgabe: Schreiben Sie eine async-Funktion getUserWithRole(id),
// die zuerst den User lädt, dann dessen Role lädt und beides kombini
  zurückgibt.
// Ergebnis: { id: 1, name: "User 1", role: "Admin" }

console.log("=== Aufgabe 1 ===");

// Ihr Code hier:
// async function getUserWithRole(id) { ... }

// Test:
// getUserWithRole(1).then(result => console.log("Result:", result));
```

```
=== Aufgabe 1 ===
```

Aufgabe 2: Parallele Queries mit Promise.all()

```
// Gegeben:
function fetchProduct(id) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ id, name: `Product ${id}`, price: id * 100 });
    }, Math.random() * 1000);
  });
}

// Aufgabe: Laden Sie die Produkte mit IDs [1, 2, 3, 4, 5] PARALLEL
// und geben Sie die Gesamtsumme aller Preise aus.

```

```
13 console.log("=== Aufgabe 2 ===");
14
15 // Ihr Code hier:
```

```
=== Aufgabe 2 ===
```

Aufgabe 3: Error Handling

```
1  // Gegeben:
2  function riskyQuery(id) {
3    return new Promise((resolve, reject) => {
4      setTimeout(() => {
5        if (id === 3) {
6          reject(new Error(`Query für ID ${id} fehlgeschlagen`));
7        } else {
8          resolve({ id, data: `Data ${id}` });
9        }
10     }, 500);
11   });
12 }
13
14 // Aufgabe: Laden Sie Daten für IDs [1, 2, 3, 4] nacheinander.
15 // Wenn eine Query fehlschlägt, loggen Sie den Fehler und machen Sie
     .
16 // Am Ende: Zeigen Sie, welche Queries erfolgreich waren.
17
18 console.log("=== Aufgabe 3 ===");
19
20 // Ihr Code hier:
```

```
=== Aufgabe 3 ===
```

Diese Übungen sind realistisch! Fast jede Datenbank-Interaktion folgt diesem Muster: Queries starten, auf Ergebnisse warten, Fehler behandeln. Wenn Sie das können, sind Sie bereit für echten DB-Code!

Lösungen:

▶ Lösung Aufgabe 1 (Sequentielle Queries)
▶ Lösung Aufgabe 2 (Parallel mit Promise.all)
▶ Lösung Aufgabe 3 (Error Handling)

Hervorragend! Sie haben jetzt das Rüstzeug für asynchrone Programmierung. async/await ist der moderne Standard, und Sie werden es in JEDER Datenbank-Operation nutzen. PouchDB, IndexedDB, REST-APIs – alles basiert auf Promises und async/await!

Kapitel 6: JSON – Das Datenformat des Web

JSON (JavaScript Object Notation) ist DAS Austauschformat für Datenbanken und APIs. Fast jede moderne Datenbank arbeitet mit JSON: REST-APIs senden JSON, NoSQL-Datenbanken speichern JSON, und selbst SQL-Datenbanken haben JSON-Spalten. Dieses Kapitel ist kurz aber absolut essenziell!

6.1 JSON Syntax – Die Regeln

JSON sieht aus wie JavaScript-Objekte, hat aber strengere Regeln. Verstehen Sie die Unterschiede, um Fehler zu vermeiden!

=== Beispiel 1: Gültiges JSON ===

```
console.log("=== Beispiel 1: Gültiges JSON ===");

const validJSON = `{
  "name": "Alice",
  "age": 28,
  "isActive": true,
  "roles": ["admin", "editor"],
  "address": {
    "city": "Berlin",
    "zip": "10115"
  },
  "salary": null
}`;

console.log("JSON String:");
console.log(validJSON);

console.log("\nJSON-Regeln:");
console.log("✅ Property-Namen in doppelten Anführungszeichen");
console.log("✅ Strings in doppelten Anführungszeichen");
console.log("✅ Zahlen, Booleans, null erlaubt");
console.log("✅ Arrays und Objekte verschachtelt erlaubt");

console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Gültiges JSON ===
JSON String:
{
  "name": "Alice",
  "age": 28,
  "isActive": true,
  "roles": ["admin", "editor"],
  "address": {
    "city": "Berlin",
    "zip": "10115"
  },
  "salary": null
}

JSON-Regeln:
✅ Property-Namen in doppelten Anführungszeichen
✅ Strings in doppelten Anführungszeichen
✅ Zahlen, Booleans, null erlaubt
✅ Arrays und Objekte verschachtelt erlaubt
--- Ende Beispiel 1 ---
```

Die wichtigsten Regeln: Property-Namen MÜSSEN in doppelten Anführungszeichen sein. Strings MÜSSEN in doppelten Anführungszeichen sein. Kein trailing comma, keine Kommentare, keine Funktionen, kein undefined.

=== Beispiel 2: Ungültiges JSON (häufige Fehler) ===

```
1  console.log("=== Beispiel 2: Ungültiges JSON ===");
2
3  // Fehler 1: Einfache Anführungszeichen
4  const invalid1 = `{ 'name': 'Alice' }`;
5  console.log("❌ Einfache Quotes:", invalid1);
6
7  // Fehler 2: Keine Quotes bei Property-Namen
8  const invalid2 = `{ name: "Alice" }`;
9  console.log("❌ Keine Quotes bei Key:", invalid2);
10
11 // Fehler 3: Trailing Comma
12 const invalid3 = `{ "name": "Alice", }`;
13 console.log("❌ Trailing Comma:", invalid3);
14
15 // Fehler 4: undefined (nicht erlaubt)
16 const invalid4 = `{ "name": undefined }`;
17 console.log("❌ undefined:", invalid4);
18
19 // Fehler 5: Funktionen (nicht erlaubt)
20 const invalid5 = `{ "getName": function() {} }`;
```

```
console.log("❌ Funktion:", invalid5);

console.log("\n→ Keines dieser Beispiele ist gültiges JSON!");
console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Ungültiges JSON ===
❌ Einfache Quotes: { 'name': 'Alice' }
❌ Keine Quotes bei Key: { name: "Alice" }
❌ Trailing Comma: { "name": "Alice", }
❌ undefined: { "name": undefined }
❌ Funktion: { "getName": function() {} }

→ Keines dieser Beispiele ist gültiges JSON!
--- Ende Beispiel 2 ---
```

Diese Fehler passieren ständig! JavaScript-Objekte sind permissiver als JSON. Wenn Sie JSON.parse() mit diesen Strings aufrufen, gibt es einen SyntaxError.

=== Beispiel 3: Was JSON NICHT kann ===

```
console.log("=== Beispiel 3: JSON Einschränkungen ===");

console.log("JSON kann NICHT darstellen:");
console.log("  ❌ undefined");
console.log("  ❌ Funktionen");
console.log("  ❌ Symbol");
console.log("  ❌ Date-Objekte (werden zu Strings)");
console.log("  ❌ RegExp");
console.log("  ❌ Map, Set");
console.log("  ❌ Zirkuläre Referenzen");
console.log("  ❌ NaN, Infinity (werden zu null)");

console.log("\nJSON kann darstellen:");
console.log("  ✅ Strings");
console.log("  ✅ Numbers");
console.log("  ✅ Booleans (true/false)");
console.log("  ✅ null");
console.log("  ✅ Arrays");
console.log("  ✅ Objects (Plain Objects)");

console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: JSON Einschränkungen ===
JSON kann NICHT darstellen:
  ❌ undefined
  ❌ Funktionen
  ❌ Symbol
  ❌ Date-Objekte (werden zu Strings)
  ❌ RegExp
  ❌ Map, Set
  ❌ Zirkuläre Referenzen
  ❌ NaN, Infinity (werden zu null)

JSON kann darstellen:
  ✅ Strings
  ✅ Numbers
  ✅ Booleans (true/false)
  ✅ null
  ✅ Arrays
  ✅ Objects (Plain Objects)
--- Ende Beispiel 3 ---
```

JSON ist ein Datenaustauschformat, kein vollständiger Ersatz für JavaScript-Objekte. Es ist absichtlich einfach gehalten, damit es sprachunabhängig ist (Python, Java, C# können es auch verarbeiten).

6.2 JSON.parse() und JSON.stringify()

Die beiden wichtigsten Funktionen: JSON.stringify() verwandelt JavaScript-Objekte in JSON-Strings. JSON.parse() verwandelt JSON-Strings zurück in JavaScript-Objekte. Sie werden diese Funktionen ständig nutzen!

=== Beispiel 1: JSON.stringify() – Objekt zu String ===

```
1  console.log("=== Beispiel 1: JSON.stringify() ===");
2
3  const user = {
4    id: 1,
5    name: "Alice",
6    email: "alice@example.com",
7    roles: ["admin", "editor"],
8    settings: {
9      theme: "dark",
10     language: "de"
11   }
12 };
13
14 console.log("JavaScript-Objekt:");
```

```
15 console.log(user);
16
17 const jsonString = JSON.stringify(user);
18 console.log("\nJSON String:");
19 console.log(jsonString);
20
21 console.log("\nTyp:");
22 console.log("  user:", typeof user);
23 console.log("  jsonString:", typeof jsonString);
24
25 console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: JSON.stringify() ===
JavaScript-Objekt:
{"id":1,"name":"Alice","email":"alice@example.com","roles":
["admin","editor"],"settings":{"theme":"dark","language":"de"}}

JSON String:
{"id":1,"name":"Alice","email":"alice@example.com","roles":
["admin","editor"],"settings":{"theme":"dark","language":"de"}}

Typ:
  user: object
  jsonString: string
--- Ende Beispiel 1 ---
```

JSON.stringify() konvertiert das Objekt in einen String. Dieser String kann über das Netzwerk geschickt, in einer Datei gespeichert oder in einer Datenbank abgelegt werden.

=== Beispiel 2: JSON.parse() – String zu Objekt ===

```
1  console.log("=== Beispiel 2: JSON.parse() ===");
2
3  const jsonString = `{
4    "id": 42,
5    "name": "Bob",
6    "isActive": true,
7    "scores": [85, 92, 78]
8  }`;
9
10 console.log("JSON String:");
11 console.log(jsonString);
12 console.log("Typ:", typeof jsonString);
13
14 const obj = JSON.parse(jsonString);
15 console.log("\nGeparst als Objekt:");
16 console.log(obj);
17 console.log("Typ:", typeof obj);
```

```
18
19 console.log("\nZugriff auf Properties:");
20 console.log("  Name:", obj.name);
21 console.log("  Scores:", obj.scores);
22 console.log("  Erster Score:", obj.scores[0]);
23
24 console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: JSON.parse() ===
JSON String:
{
  "id": 42,
  "name": "Bob",
  "isActive": true,
  "scores": [85, 92, 78]
}
Typ: string

Geparst als Objekt:
{"id":42,"name":"Bob","isActive":true,"scores":[85,92,78]}
Typ: object

Zugriff auf Properties:
  Name: Bob
  Scores: [85,92,78]
  Erster Score: 85
--- Ende Beispiel 2 ---
```

JSON.parse() verwandelt den String zurück in ein nutzbares JavaScript-Objekt. Jetzt können Sie mit den Daten arbeiten: Properties lesen, Arrays durchlaufen, etc.

=== Beispiel 3: Prettify mit stringify() ===

```
1  console.log("=== Beispiel 3: Prettify ===");
2
3  const data = {
4    users: [
5      { id: 1, name: "Alice", age: 28 },
6      { id: 2, name: "Bob", age: 34 }
7    ],
8    total: 2
9  };
10
11 console.log("Ohne Formatierung:");
12 console.log(JSON.stringify(data));
13
14 console.log("\nMit Formatierung (Indent 2):");
```

```
15 const pretty = JSON.stringify(data, null, 2);
16 console.log(pretty);
17
18 console.log("\nMit Formatierung (Indent 4):");
19 const prettyWide = JSON.stringify(data, null, 4);
20 console.log(prettyWide);
21
22 console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: Prettify ===
Ohne Formatierung:
{"users":[{"id":1,"name":"Alice","age":28},
{"id":2,"name":"Bob","age":34}],"total":2}

Mit Formatierung (Indent 2):
{
  "users": [
    {
      "id": 1,
      "name": "Alice",
      "age": 28
    },
    {
      "id": 2,
      "name": "Bob",
      "age": 34
    }
  ],
  "total": 2
}

Mit Formatierung (Indent 4):
{
    "users": [
        {
            "id": 1,
            "name": "Alice",
            "age": 28
        },
        {
            "id": 2,
            "name": "Bob",
            "age": 34
        }
    ],
    "total": 2
}
--- Ende Beispiel 3 ---
```

Der dritte Parameter von JSON.stringify() ist der Indent. Mit 2 oder 4 Leerzeichen wird das JSON lesbar formatiert. Das ist nützlich zum Debuggen oder wenn Sie JSON-Dateien per Hand editieren.

=== Beispiel 4: Roundtrip (hin und zurück) ===

```
console.log("=== Beispiel 4: Roundtrip ===");

const original = {
  product: "Laptop",
  price: 999,
  tags: ["electronics", "computers"]
};

console.log("1. Original Object:");
console.log(original);

const json = JSON.stringify(original);
console.log("\n2. Als JSON String:");
console.log(json);

const parsed = JSON.parse(json);
console.log("\n3. Zurück als Object:");
console.log(parsed);

console.log("\n4. Vergleich:");
console.log("  Sind gleich (Werte)?",
  JSON.stringify(original) === JSON.stringify(parsed));
console.log("  Sind gleich (Referenz)?", original === parsed);

console.log("--- Ende Beispiel 4 ---");
```

```
=== Beispiel 4: Roundtrip ===
1. Original Object:
{"product":"Laptop","price":999,"tags":["electronics","computers"]}

2. Als JSON String:
{"product":"Laptop","price":999,"tags":["electronics","computers"]}

3. Zurück als Object:
{"product":"Laptop","price":999,"tags":["electronics","computers"]}

4. Vergleich:
  Sind gleich (Werte)? true
  Sind gleich (Referenz)? false
--- Ende Beispiel 4 ---
```

Wichtig! Nach dem Roundtrip haben Sie ein neues Objekt mit denselben Werten, aber einer anderen Referenz. Das ist kein Problem – die Daten sind identisch.

=== Beispiel 5: API-Response simulieren ===

```
console.log("=== Beispiel 5: API Response ===");

// Simuliere eine API-Response (als String vom Server)
const apiResponse = `{
  "status": "success",
  "data": {
    "users": [
      { "id": 1, "name": "Alice", "email": "alice@example.com" },
      { "id": 2, "name": "Bob", "email": "bob@example.com" }
    ]
  },
  "timestamp": "2024-10-21T10:30:00Z"
}`;

console.log("API Response (String):");
console.log(apiResponse);

const response = JSON.parse(apiResponse);
console.log("\nGeparst:");
console.log("  Status:", response.status);
console.log("  User-Count:", response.data.users.length);

response.data.users.forEach(user => {
  console.log(`   → ${user.name} (${user.email})`);
});

console.log("--- Ende Beispiel 5 ---");
```

Das ist das typische Pattern: Server sendet JSON-String, Sie parsen ihn mit JSON.parse(), verarbeiten die Daten. Später stringify Sie Ihre Antwort und senden sie zurück. Das ist der Standard für REST-APIs!

6.3 JSON vs. JavaScript-Objekte

JSON und JavaScript-Objekte sehen ähnlich aus, sind aber NICHT dasselbe! Dieser Unterschied ist wichtig zu verstehen.

=== Beispiel 1: Unterschiede im Detail ===

```
console.log("=== Beispiel 1: JSON vs. JS-Objekt ===");

// JavaScript-Objekt (permissiv)
const jsObject = {
  name: "Alice",              // Keine Quotes bei Key
  'age': 28,                  // Einfache Quotes OK
  "email": "alice@ex.com",    // Doppelte Quotes OK
  isActive: true,
  greet: function() {         // Funktionen OK
    return "Hello!";
```

```
  },
  metadata: undefined,         // undefined OK
  tags: ["admin", "editor",]  // Trailing comma OK
};

console.log("JavaScript-Objekt:");
console.log(jsObject);
console.log("  Function:", jsObject.greet());

// Zu JSON konvertieren
const json = JSON.stringify(jsObject);
console.log("\nAls JSON:");
console.log(json);

console.log("\n❌ Verloren beim Konvertieren:");
console.log("  - greet() Funktion");
console.log("  - metadata (war undefined)");

console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: JSON vs. JS-Objekt ===
JavaScript-Objekt:
{"name":"Alice","age":28,"email":"alice@ex.com","isActive":true,"tags":
["admin","editor"]}
  Function: Hello!

Als JSON:
{"name":"Alice","age":28,"email":"alice@ex.com","isActive":true,"tags":
["admin","editor"]}

❌ Verloren beim Konvertieren:
  - greet() Funktion
  - metadata (war undefined)
--- Ende Beispiel 1 ---
```

Beim Konvertieren nach JSON gehen Funktionen und undefined verloren! JSON.stringify() ignoriert sie einfach. Das ist kein Bug, sondern beabsichtigt – JSON ist für Daten, nicht für Code.

=== Beispiel 2: Date-Objekte in JSON ===

```
console.log("=== Beispiel 2: Date in JSON ===");

const event = {
  title: "Meeting",
  date: new Date("2024-10-21T10:00:00"),
  location: "Berlin"
};
```

```
console.log("Original:");
console.log("  Date:", event.date);
console.log("  Typ:", typeof event.date);
console.log("  getFullYear():", event.date.getFullYear());

const json = JSON.stringify(event);
console.log("\nAls JSON:");
console.log(json);

const parsed = JSON.parse(json);
console.log("\nZurück geparst:");
console.log("  Date:", parsed.date);
console.log("  Typ:", typeof parsed.date);
console.log("  ❌ getFullYear() existiert nicht mehr!");

console.log("\nLösung: Manuell zurück konvertieren");
parsed.date = new Date(parsed.date);
console.log("  Date:", parsed.date);
console.log("  getFullYear():", parsed.date.getFullYear());

console.log("--- Ende Beispiel 2 ---");
```

```
=== Beispiel 2: Date in JSON ===
Original:
  Date: "2024-10-21T10:00:00.000Z"
  Typ: object
  getFullYear(): 2024

Als JSON:
{"title":"Meeting","date":"2024-10-
21T10:00:00.000Z","location":"Berlin"}

Zurück geparst:
  Date: 2024-10-21T10:00:00.000Z
  Typ: string
  ❌ getFullYear() existiert nicht mehr!

Lösung: Manuell zurück konvertieren
  Date: "2024-10-21T10:00:00.000Z"
  getFullYear(): 2024
--- Ende Beispiel 2 ---
```

Date-Objekte werden zu ISO-Strings konvertiert. Nach dem Parsen sind sie einfache Strings! Sie müssen sie manuell zurück in Date-Objekte konvertieren. Das ist ein häufiger Stolperstein.

=== Beispiel 3: NaN und Infinity ===

```javascript
console.log("=== Beispiel 3: NaN und Infinity ===");

const numbers = {
  normal: 42,
  notANumber: NaN,
  infinite: Infinity,
  negInfinite: -Infinity
};

console.log("Original:");
console.log(numbers);

const json = JSON.stringify(numbers);
console.log("\nAls JSON:");
console.log(json);

const parsed = JSON.parse(json);
console.log("\nZurück geparst:");
console.log(parsed);

console.log("\n❌ NaN und Infinity wurden zu null!");
console.log("  notANumber:", parsed.notANumber);
console.log("  infinite:", parsed.infinite);

console.log("--- Ende Beispiel 3 ---");
```

```
=== Beispiel 3: NaN und Infinity ===
Original:
{"normal":42,"notANumber":null,"infinite":null,"negInfinite":null}

Als JSON:
{"normal":42,"notANumber":null,"infinite":null,"negInfinite":null}

Zurück geparst:
{"normal":42,"notANumber":null,"infinite":null,"negInfinite":null}

❌ NaN und Infinity wurden zu null!
  notANumber: null
  infinite: null
--- Ende Beispiel 3 ---
```

NaN und Infinity gibt es in JSON nicht! Sie werden zu null konvertiert. Wenn Sie diese Werte brauchen, müssen Sie sie anders kodieren (z.B. als Strings).

6.4 Deep Copy mit JSON

Ein cleverer Trick: JSON.stringify() + JSON.parse() erzeugt eine tiefe Kopie eines Objekts. Aber Vorsicht: Funktionen und andere nicht-JSON-Typen gehen verloren!

=== Beispiel 1: Shallow vs. Deep Copy ===

```
console.log("=== Beispiel 1: Shallow vs. Deep Copy ===");

const original = {
  name: "Alice",
  scores: [85, 90, 92]
};

// Shallow Copy (Spread)
const shallow = { ...original };

// Deep Copy (JSON Trick)
const deep = JSON.parse(JSON.stringify(original));

console.log("Original:", original);
console.log("Shallow Copy:", shallow);
console.log("Deep Copy:", deep);

console.log("\nÄndere original.scores[0]:");
original.scores[0] = 999;

console.log("  Original:", original.scores);
console.log("  Shallow:", shallow.scores);  // AUCH geändert! ❌
console.log("  Deep:", deep.scores);        // NICHT geändert! ✅

console.log("--- Ende Beispiel 1 ---");
```

```
=== Beispiel 1: Shallow vs. Deep Copy ===
Original: {"name":"Alice","scores":[85,90,92]}
Shallow Copy: {"name":"Alice","scores":[85,90,92]}
Deep Copy: {"name":"Alice","scores":[85,90,92]}

Ändere original.scores[0]:
  Original: [999,90,92]
  Shallow: [999,90,92]
  Deep: [85,90,92]
--- Ende Beispiel 1 ---
```

Shallow Copy kopiert nur die erste Ebene. Nested Arrays/Objects werden referenziert! Deep Copy mit JSON erzeugt eine komplette Kopie. Änderungen am Original beeinflussen die Deep Copy nicht.

=== Beispiel 2: Deep Copy mit komplexen Daten ===

```
console.log("=== Beispiel 2: Deep Copy komplexer Daten ===");
```

```
2
3  const dbResult = {
4    users: [
5      { id: 1, name: "Alice", meta: { lastLogin: "2024-10-20" } },
6      { id: 2, name: "Bob", meta: { lastLogin: "2024-10-19" } }
7    ],
8    pagination: {
9      page: 1,
10     perPage: 10,
11     total: 2
12   }
13 };
14
15 console.log("Original DB Result:");
16 console.log(dbResult);
17
18 // Deep Copy für Verarbeitung
19 const workingCopy = JSON.parse(JSON.stringify(dbResult));
20
21 console.log("\nModifiziere Working Copy:");
22 workingCopy.users[0].name = "ALICE (Modified)";
23 workingCopy.pagination.page = 2;
24
25 console.log("\nOriginal (unverändert):");
26 console.log("  User 1:", dbResult.users[0].name);
27 console.log("  Page:", dbResult.pagination.page);
28
29 console.log("\nWorking Copy (geändert):");
30 console.log("  User 1:", workingCopy.users[0].name);
31 console.log("  Page:", workingCopy.pagination.page);
32
33 console.log("--- Ende Beispiel 2 ---");
```

Das ist nützlich, wenn Sie mit Datenbank-Resultaten arbeiten! Sie können eine Deep Copy machen, die Daten transformieren, ohne das Original zu beeinflussen. Perfekt für „Dry Run“ Tests oder Vorschau-Funktionen.

=== Beispiel 3: Grenzen des JSON-Copy-Tricks ===

```
1  console.log("=== Beispiel 3: Grenzen des JSON-Tricks ===");
2
3  const complex = {
4    name: "Test",
5    date: new Date("2024-10-21"),
6    regex: /test/gi,
7    func: function() { return "Hello"; },
8    undef: undefined,
9    nan: NaN,
10   inf: Infinity
11 };
12
```

```
13 console.log("Original:");
14 console.log(complex);
15
16 const copy = JSON.parse(JSON.stringify(complex));
17
18 console.log("\nCopy:");
19 console.log(copy);
20
21 console.log("\n❌ Verloren/Verändert:");
22 console.log("  date: Date → String");
23 console.log("  regex: Verschwunden");
24 console.log("  func: Verschwunden");
25 console.log("  undef: Verschwunden");
26 console.log("  nan: NaN → null");
27 console.log("  inf: Infinity → null");
28
29 console.log("\n→ Für Plain Objects (Daten) OK");
30 console.log("→ Für komplexe Objekte: structuredClone() nutzen!");
31
32 console.log("--- Ende Beispiel 3 ---");
```

Der JSON-Trick funktioniert nur für Plain Objects! Für komplexe Objekte mit Dates, RegExp, etc. gibt es seit 2022 structuredClone() im Browser. Aber für typische Datenbank-Daten (Plain Objects) ist der JSON-Trick perfekt.

6.5 Häufige Fehler mit JSON

6.6 Übung: JSON

Kapitel 7: Classes – Objektorientierung (Optional)

7.1 Class Syntax

7.2 Constructor

7.3 Methods

7.4 Getter & Setter

7.5 Static Methods

7.6 Inheritance (Kurzüberblick)

7.7 Übung: Classes

Zusammenfassung & Cheat Sheet

11.1 JavaScript Syntax Cheat Sheet

11.2 Häufige Stolpersteine & Best Practices

11.3 IndexedDB Pattern-Referenz

11.4 Weiterführende Ressourcen