Tag 2 JavaScript: Variablen, Datentypen und Typumwandlung

1. Variablen in JavaScript

Was ist eine Variable?

Eine Variable ist ein benannter Speicherplatz für einen Wert. In JavaScript können Variablen nahezu jeden Datentyp aufnehmen – auch mehrfach hintereinander. Die Sprache ist **dynamisch typisiert**: Man muss den Typ also nicht vorher festlegen.

Deklaration und Initialisierung (Instanziierung)

In JavaScript unterscheidet man **Deklaration** und **Initialisierung (auch: Instanziierung)** explizit.

- **Deklaration**: Die Variable wird angelegt, aber es wird noch kein Wert zugewiesen.
- **Initialisierung / Instanziierung**: Es wird ein konkreter Wert zugewiesen – die Variable erhält damit ihre erste „Instanz".

Beispiel:

```
let name;         // Deklaration
name = "Anna";    // Initialisierung (Instanziierung)
```

Oder beides zusammen:

```
let stadt = "Berlin"; // Deklaration + Initialisierung
```

Dieser Unterschied existiert in Python so **nicht** explizit – dort wird durch eine Zuweisung automatisch auch deklariert:

```
name = "Anna"  # automatisch deklariert und initialisiert
```

In JavaScript hingegen kann man eine Variable deklarieren, ohne ihr sofort einen Wert zu geben. Der Wert ist dann automatisch `undefined`.

```
let a;
console.log(a); // undefined
```

Dieser Aspekt ist besonders relevant für das Verständnis von `undefined` und Hoisting (siehe unten).

Drei Arten der Deklaration

JavaScript kennt drei Schlüsselwörter zur Deklaration:

Schlüsselwort	Gültigkeitsbereich (Scope)	Wiederzuweisung erlaubt?	Besonderheit
`var`	Funktional (Function Scope)	Ja	Wird gehoisted, veraltet
`let`	Blockbasiert	Ja	Modern, Standard
`const`	Blockbasiert	Nein (Wert fix)	Muss sofort initialisiert werden

Beispiel

```
let stadt = "Berlin";
const land = "Deutschland";
```

```
stadt = "Hamburg"; // erlaubt
land = "Frankreich"; // ❌ Fehler: const-Wert kann nicht überschrieben werden
```

Wichtig zu `const`

`const` bedeutet nicht, dass der Inhalt eines Objekts oder Arrays unveränderlich ist – nur die **Referenz** darf nicht neu zugewiesen werden:

```
const person = { name: "Anna" };
person.name = "Maria"; // erlaubt
person = { name: "Tom" }; // ❌ Fehler
```

Scope – Gültigkeitsbereich von Variablen

Der Scope entscheidet, **wo** eine Variable gültig und sichtbar ist. Das ist in JavaScript entscheidend, insbesondere wegen der Unterschiede zwischen `var`, `let` und `const`.

1. Globaler Scope

Eine Variable, die **außerhalb aller Funktionen und Blöcke** definiert wird, ist global verfügbar.

```
let globalVar = "Ich bin global";
```

2. Funktionsscope (nur bei `var`)

Mit `var` deklarierte Variablen sind **innerhalb einer Funktion** gültig – selbst wenn sie in einem Block `{}` stehen.

```
function test() {
  if (true) {
    var x = 42;
  }
  console.log(x); // 42 – `var` lebt in der ganzen Funktion
}
test();
```

3. Blockscope (bei `let` und `const`)

`let` und `const` sind **auf den Block beschränkt**, in dem sie definiert wurden.

```
{
  let y = 99;
  const z = 100;
  console.log(y, z); // 99 100
}
console.log(y); // ❌ ReferenceError
```

Variable Shadowing

Shadowing tritt auf, wenn eine Variable **im inneren Scope denselben Namen hat wie im äußeren** – die innere „verdeckt" dann die äußere.

Beispiel:

```
let wert = 10;
{
  let wert = 20;
  console.log(wert); // 20 (innere Variable überschattet die äußere)
}
console.log(wert); // 10 (außerhalb wieder sichtbar)
```

Bei `var` besonders kritisch

Da `var` funktionsbasiert ist, kann Shadowing schnell zu Fehlern führen:

```
function test() {
  var wert = 5;
  if (true) {
    var wert = 10;
    console.log(wert); // 10
  }
  console.log(wert); // auch 10 – gleiche Variable!
}
test();
```

Deshalb sollte man moderne JavaScript-Projekte **immer mit `let` oder `const` schreiben**, um diese Fallstricke zu vermeiden.

Vergleich zu Python

```
name = "Anna" # dynamisch typisiert, wie in JS
```

In Python gibt es keine Schlüsselwörter für Deklaration – Variablen sind automatisch global oder lokal, je nach Einbettung. Konzepte wie **Hoisting**, **Blockscope** und **Shadowing** gibt es dort nicht in dieser Form.

JavaScript verlangt eine **bewusste Wahl des Kontexts und der Sichtbarkeit** – ein häufiger Stolperstein für Anfänger.

---

2. Hoisting

JavaScript „zieht" Deklarationen mit `var` intern nach oben. Das nennt man **Hoisting**.

```
console.log(a); // undefined (kein Fehler)
var a = 5;
```

- Bei `let` und `const` funktioniert das nicht:

```
console.log(b); // ❌ ReferenceError
let b = 10;
```

Merksatz: Nur `var` wird „gehoisted" – aber ohne Wert. Der Code wird dadurch schwer lesbar und fehleranfällig.

3. Primitive Datentypen

JavaScript kennt sieben primitive Datentypen. Diese Typen sind **nicht veränderbar (immutable)** und werden **nicht als Objekte** gespeichert. Jeder Wert in JavaScript ist entweder ein primitiver Wert oder ein Objekt.

Übersicht der primitiven Typen:

Datentyp	Beschreibung
`string`	Zeichenkette, z. B. Namen, Wörter, Text
`number`	Ganzzahlen und Fließkommazahlen
`bigint`	Sehr große Ganzzahlen (ab ES2020)
`boolean`	Wahrheitswerte: `true` oder `false`
`undefined`	Eine deklarierte, aber nicht initialisierte Variable
`null`	Bedeutet explizit „kein Wert“
`symbol`	Einzigartige und unveränderliche Werte (für Objekte als Schlüssel)

`string`

```
let name = "Anna";
let begruessung = `Hallo, ${name}!`;
console.log(begruessung); // "Hallo, Anna!"
```

- Strings können in `'...'`, `"..."` oder `` `...` `` geschrieben werden
- Template Literals (mit Backticks) erlauben **Interpolation** (`${}`)
- Strings sind **immutable** – Methoden wie `.toUpperCase()` geben neue Strings zurück

Fallstrick:

```
let str = "Test";
str[0] = "B";
console.log(str); // "Test" – Änderung schlägt fehl
```

`number`

```
let zahl = 42;
let preis = 19.99;
```

- Es gibt nur einen Zahlentyp in JS: `number` (egal ob ganz oder mit Nachkommastellen)
- Intern sind alle Zahlen **64-bit Gleitkommazahlen (IEEE 754)**

Fallstrick: Rundungsungenauigkeiten:

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

`bigint`

```
let grosseZahl = 12345678901234567890123456789012345678901234567890n;
console.log(typeof grosseZahl); // "bigint"
```

- Für sehr große Ganzzahlen außerhalb des `number`-Limits
- Muss mit `n` enden

Wichtig: `bigint` und `number` **dürfen nicht gemischt** werden:

```
console.log(10n + 5); // ❌ TypeError
```

`boolean`

```
let aktiv = true;
let istLeer = false;
```

- Nur zwei Werte möglich: `true` und `false`
- Häufig Ergebnis von Vergleichen:

```
let istVolljaehrig = alter >= 18;
```

Coercion: Viele Werte verhalten sich in Bedingungen wie `true` oder `false` → siehe auch `Boolean()`-Konvertierung

---

`undefined`

```
let a;
console.log(a); // undefined
```

- Variable wurde deklariert, aber **nicht initialisiert**
- Typ: `undefined`

Fallstrick:

```
function test() {}
let result = test();
console.log(result); // undefined (keine Rückgabe)
```

`null`

```
let b = null;
console.log(b); // null
```

- Wert für „bewusst kein Wert“
- Wird oft zur **Initialisierung** verwendet, wenn ein Wert absichtlich leer sein soll

Fallstrick:

```
console.log(typeof null); // "object" – bekanntes JS-Fehlverhalten
```

- Technisch kein Objekt, aber dieser Bug bleibt aus Kompatibilitätsgründen erhalten

---

`symbol` (fortgeschritten)

```
const s1 = Symbol("id");
const s2 = Symbol("id");
console.log(s1 === s2); // false
```

- Wird für **einzigartige, unveränderliche Bezeichner** verwendet
- Vor allem in komplexeren Objektstrukturen, Libraries oder Frameworks (z. B. React intern)
- Zwei `Symbol("text")` mit gleichem Wert sind **nicht gleich** – sie sind **immer einzigartig**

`typeof` – den Typ eines Wertes herausfinden

Die Funktion `typeof` gibt den Datentyp eines Wertes oder einer Variablen als String zurück.

Beispiele:

```
console.log(typeof "Hallo");       // "string"
console.log(typeof 42);            // "number"
console.log(typeof true);          // "boolean"
console.log(typeof undefined);     // "undefined"
console.log(typeof null);          // "object" (!) historischer Fehler
console.log(typeof 10n);           // "bigint"
console.log(typeof Symbol("id")); // "symbol"
```

Tipp: `typeof` ist besonders hilfreich beim Debuggen oder um z. B. Benutzereingaben zu prüfen.

---

4. Strings (Zeichenketten)

Ein **String** ist eine Zeichenkette – also eine Folge von Zeichen (Buchstaben, Zahlen, Sonderzeichen usw.). Strings sind einer der am häufigsten verwendeten Datentypen in JavaScript und tauchen in fast jedem Programm auf.

Schreibweise

Strings können in JavaScript mit:

- einfachen Anführungszeichen `'Text'`
- doppelten Anführungszeichen `"Text"`
- oder Backticks (Template Literals) `` `Text` `` geschrieben werden.

```
let a = 'Hallo';
let b = "Welt";
let c = `Hallo, ${b}`; // Template Literal mit Platzhalter
```

Die Verwendung von Template Literals mit `${...}` zur Variablen-Einbettung ist besonders leserfreundlich.

Länge eines Strings

```
let name = "Anna";
console.log(name.length); // 4
```

- `.length` ist keine Methode, sondern eine Eigenschaft

Zeichenposition und Zugriff

```
let wort = "JavaScript";
console.log(wort[0]);      // J
console.log(wort.charAt(1)); // a
```

- Der Zugriff erfolgt wie bei einem Array über einen Index (beginnend bei 0)

Häufig genutzte Methoden

Methode	Funktion	Beispiel
`toUpperCase()`	Wandelt alles in Großbuchstaben um	`'hallo'.toUpperCase()` → `HALLO`
`toLowerCase()`	Wandelt alles in Kleinbuchstaben um	`'WELT'.toLowerCase()` → `welt`
`includes()`	Prüft, ob ein Teilstring enthalten ist	`'JavaScript'.includes('Script')` → `true`
`startsWith()`	Prüft, ob der String mit einem bestimmten Teil beginnt	`'Hallo Welt'.startsWith('Hallo')` → `true`
`endsWith()`	Prüft, ob der String mit einem bestimmten Teil endet	`'Hallo Welt'.endsWith('Welt')` → `true`
`slice(start, end)`	Schneidet Teilstring heraus	`'Hallo'.slice(1, 4)` → `'all'`
`replace(a, b)`	Ersetzt erstes Vorkommen von `a` durch `b`	`'test'.replace('t', 'b')` → `'best'`
`trim()`	Entfernt Leerzeichen vorn und hinten	`' test '.trim()` → `'test'`
`split(separator)`	Wandelt String in Array um	`'a,b,c'.split(',')` → `['a','b','c']`

Strings sind immutable

Strings können nicht direkt verändert werden – alle Methoden geben **einen neuen String zurück**:

```
let text = "hallo";
text[0] = "H";
console.log(text); // "hallo"

let neuerText = text.toUpperCase();
console.log(neuerText); // "HALLO"
```

Sonderzeichen und Escape-Sequenzen

Sequenz	Bedeutung
`\n`	Zeilenumbruch
`\t`	Tabulator
`\"`	Anführungszeichen
`\\`	Backslash selbst

```
console.log("Zeile 1\nZeile 2");
```

Vergleich zu Python

Auch in Python sind Strings unveränderlich und bieten viele Methoden. Unterschiede:

- In JS gibt es zusätzlich Template Literals
- `length` ist in JS eine Eigenschaft, in Python eine Funktion: `len(text)`
- Methoden-Namen und Verhalten können leicht abweichen

Typ mit `typeof` prüfen

```
let s = "Hallo";
console.log(typeof s); // "string"
```

---

5. Komplexe Datentypen: Objekte & Arrays

In JavaScript sind **komplexe Datentypen** alle Datentypen, die nicht zu den primitiven Typen zählen. In diesem Kurs fokussieren wir uns auf die zwei wichtigsten: **Objekte** und **Arrays**.

5.1 Objekte

Ein **Objekt** ist eine Sammlung von Schlüssel-Wert-Paaren. Die Schlüssel werden auch **Properties** genannt. Objekte dienen dazu, zusammengehörige Daten sinnvoll zu strukturieren.

Objekt erstellen:

```
let person = {
  name: "Calvin",
  age: 66,
  email: "calvin@example.com"
};
```

Zugriff auf Eigenschaften:

```
console.log(person.name);      // "Calvin"
console.log(person["email"]); // "calvin@example.com"
```

Eigenschaften ändern und hinzufügen:

```
person.age = 67;
person.phone = "123-456";
```

Eigenschaften löschen:

```
delete person.phone;
```

Vorteile:

- Klar strukturierte Daten
- Eigenschaften sind durch Namen leicht verständlich
- Beliebige Typen als Werte möglich

Beispiel: Benutzer ohne Objekt (unpraktisch)

```
let name1 = "Calvin";
let age1 = 66;
let email1 = "calvin@example.com";
```

Mit Objekt (empfohlen):

```
let user1 = {
  name: "Calvin",
  age: 66,
  email: "calvin@example.com"
};
```

5.2 Arrays

Ein **Array** ist eine geordnete Sammlung von Werten. Die einzelnen Werte sind über **Indizes** zugänglich (beginnend bei 0).

Array erstellen:

```
let tage = ["So", "Mo", "Di", "Mi", "Do", "Fr", "Sa"];
```

Zugriff und Änderung:

```
console.log(tage[0]); // "So"
tage[0] = "Sonntag";
```

Leeres Array:

```
let leer = [];
```

Mischung von Datentypen:

```
let mixed = ["Text", 42, true, null];
```

Verschachtelte Arrays:

```
let namen = [["Olivia", "Emma"], ["James", "Daniel"]];
console.log(namen[0][1]); // "Emma"
```

Array mit Objekten (z. B. Benutzerliste):

```
let users = [
  { name: "Calvin", age: 66 },
  { name: "Mateus", age: 21 }
```

```
];
console.log(users[1].name); // "Mateus"
```

Typprüfung:

```
console.log(typeof users);         // "object"
console.log(users instanceof Array); // true
```

5.3 Array-Eigenschaften und Methoden

`length` – Anzahl der Elemente

```
let namen = ["Olivia", "Emma"];
console.log(namen.length); // 2
```

`indexOf()` – Index eines Werts

```
let arr = ["a", "b", "c"];
console.log(arr.indexOf("b")); // 1
console.log(arr.indexOf("x")); // -1
```

`push()` – ans Ende hinzufügen

```
arr.push("d");
```

`unshift()` – an den Anfang hinzufügen

```
arr.unshift("z");
```

`pop()` – letztes Element entfernen

```
let letztes = arr.pop();
```

`shift()` – erstes Element entfernen

```
let erstes = arr.shift();
```

`reverse()` – Reihenfolge umkehren

```
arr.reverse();
```

`slice(start, end)` – Teil kopieren

```
let teil = arr.slice(1, 3);
```

`concat()` – Arrays verketten

```
let alle = arr.concat(["x", "y"]);
```

5.4 Warum sind Arrays so wichtig?

- Dynamisch erweiterbar
- Sammlung beliebiger Werte unter einem Namen
- Grundlage für viele JS-Techniken: Loops, Funktionen, Datenverarbeitung

Arrays sind „der Einstieg" in echte Datenverarbeitung. Deshalb werden sie in späteren Kapiteln nochmals vertieft – insbesondere bei Schleifen, Funktionen, Methoden wie `map`, `filter`, `forEach` usw.

5.5 Vergleich Objekt vs. Array

Kriterium	Objekt	Array
Schlüssel	benannt ("name", "email")	nummeriert (0, 1, 2, ...)
Zugriff	`obj.key` oder `obj["key"]`	`arr[0]`
Inhalt	strukturierte Eigenschaften	Liste gleichartiger Werte
Reihenfolge	nicht garantiert	geordnet
Erweiterung	dynamisch	dynamisch

---

6. Typumwandlung (Type Conversion)

JavaScript ist dafür bekannt, Datentypen automatisch zu konvertieren – das nennt man **implizite Typumwandlung** oder auch **Type Coercion**. Manchmal ist das nützlich, manchmal sehr verwirrend.

Implizite Umwandlung (coercion)

Bei vielen Operatoren versucht JavaScript automatisch, passende Typen herzustellen. Das passiert z. B. bei `+`, `-`, Vergleichen oder logischen Ausdrücken.

Beispiel 1: Addition mit String

```
const str1 = 42 + "1";
console.log(str1);       // "421"
console.log(typeof str1); // string
```

Erklärung: Der `+`-Operator steht auch für **String-Verkettung**. Wenn einer der Operanden ein String ist, wandelt JavaScript **alle Operanden zu Strings** um. Aus `42 + "1"` wird also die Zeichenkette "421".

Beispiel 2: Subtraktion mit String

```
const str2 = 42 - "1";
console.log(str2);        // 41
console.log(typeof str2); // number
```

Erklärung: `-` ist ein **arithmetischer Operator**, und JavaScript kann keine Zeichenketten subtrahieren. Daher wird der String "1" **in eine Zahl konvertiert**. `42 - 1 = 41`.

Das Verhalten hängt also vom **Operator** ab – bei `+` kann coercion zu String-Verkettung führen, bei `-`, `*`, `/` wird versucht, die Werte als Zahlen zu behandeln.

Weitere Beispiele:

```
true + 1      // 2   (true wird zu 1)
"5" - 2       // 3   ("5" → 5, dann 5 - 2)
false + "7"   // "false7" (false → "false")
null + 1      // 1   (null → 0)
undefined + 1 // NaN (undefined kann nicht in Zahl gewandelt werden)
```

Explizite Umwandlung (type casting)

Man kann Typen auch gezielt umwandeln:

```
Number("5")   // 5
String(123)   // "123"
Boolean(0)    // false
```

Merksätze:

- Bei `+` mit String: **Alles wird zum String!** (z. B. "5" + 1 = "51")
- Bei `-`, `*`, `/`: **Alles wird zur Zahl**, wenn möglich (z. B. "5" - 1 = 4)
- `null` wird zu 0, `true` zu 1, `false` zu 0, `undefined` zu NaN

Typ prüfen mit `typeof`

```
let x = "Hallo";
console.log(typeof x); // "string"
```

7. Übungsaufgaben: Variablen, Datentypen und Typumwandlung

Aufgabe 1: Variablen deklarieren und initialisieren

Erstelle folgende Variablen:

- `vorname` (String)
- `alter` (Number)
- `istStudent` (Boolean)
- Gib alle Werte mit `console.log()` aus.

Aufgabe 2: Unterschied `let` und `const`

Deklariere eine Variable `stadt` mit `let` und ändere später ihren Wert. Deklariere eine weitere Variable `land` mit `const` und versuche sie zu verändern. Was passiert?

Aufgabe 3: Hoisting verstehen

Führe folgenden Code aus:

```
console.log(x);
var x = 5;
```

Erkläre das Ergebnis. Wiederhole den Test mit `let` statt `var`.

Aufgabe 4: Primitive Datentypen anwenden

Erstelle je eine Variable für jeden primitiven Datentyp. Nutze `typeof`, um den Typ jeder Variable in der Konsole auszugeben.

Aufgabe 5: Strings bearbeiten

Erstelle einen String `satz = " JavaScript ist toll! "`

- Entferne Leerzeichen mit `.trim()`
- Ersetze „toll“ mit „fantastisch“
- Wandelt den Satz in Großbuchstaben um
- Gib die Anzahl der Zeichen mit `.length` aus

Aufgabe 6: Objekt erstellen und manipulieren

Erstelle ein Objekt `auto` mit den Eigenschaften:

- `marke`
- `modell`
- `baujahr` Füge eine neue Eigenschaft `farbe` hinzu. Ändere `baujahr`. Lösche `farbe` wieder. Gib das Objekt nach jedem Schritt in der Konsole aus.

Aufgabe 7: Array mit Wochentagen

Erstelle ein Array `tage` mit den sieben Wochentagen (Abkürzungen).

- Ersetze „Mo“ durch „Montag“
- Füge „Feiertag“ am Ende hinzu
- Entferne den ersten Tag mit `.shift()`
- Gib das Ergebnis aus

Aufgabe 8: Array mit gemischten Datentypen

Erstelle ein Array `mix` mit folgenden Elementen:

- ein String
- eine Zahl
- ein Boolean
- `null`
- ein Objekt mit einer Eigenschaft `wert` Greife auf den `wert` im Objekt im Array zu.

Aufgabe 9: Typumwandlung verstehen

Was ergibt folgender Code? Schreibe ihn ab und erkläre die Ausgabe:

```
let a = "5" + 2;
let b = "5" - 2;
let c = true + 1;
let d = null + 1;
let e = undefined + 1;
console.log(a, b, c, d, e);
```

Aufgabe 10: Array von Objekten

Erstelle ein Array `users`, das drei Objekte enthält. Jedes Objekt soll einen Namen und ein Alter enthalten.

- Gib den Namen des zweiten Benutzers aus
- Ändere das Alter des dritten Benutzers
- Füge einen vierten Benutzer hinzu
- Gib das vollständige Array aus