Skript 4: Statische Eigenschaften und fortgeschrittene Funktionen in Klassen

Einleitung

Neben Instanzmethoden und -eigenschaften erlaubt JavaScript auch sogenannte **statische** Elemente in Klassen. Diese sind nicht Teil des Objekts, das durch `new` erzeugt wird, sondern bleiben an der Klasse selbst gebunden. Solche statischen Methoden und Eigenschaften werden typischerweise für Werkzeuge, Vergleichslogik oder Factory-Methoden verwendet.

Außerdem betrachten wir fortgeschrittene Nutzungsmuster von Methoden in Klassen – inklusive Arrow Functions, Methodenschatten und dynamischer Methodenbindung.

1. Statische Methoden mit `static`

Statische Methoden werden direkt an der Klasse definiert und aufgerufen – **nicht über Instanzen**.

```
class MathUtils {
  static square(x) {
    return x * x;
  }
}

console.log(MathUtils.square(5)); // 25
```

Eigenschaften:

- Nicht über `this`, sondern direkt über den Klassennamen erreichbar
- Werden meist für Hilfsfunktionen genutzt
- **Kein Zugriff auf Instanzdaten** (z. B. `this.id` ist `undefined`)

Python-Vergleich

```
class MathUtils:
    @staticmethod
    def square(x):
        return x * x

print(MathUtils.square(5))
```

Beide Varianten kapseln Funktionen, die **nicht an die Instanz gebunden** sind.

2. Statische Eigenschaften

Auch Eigenschaften lassen sich statisch definieren. In modernen Browsern kannst du direkt im Klassentext arbeiten:

```
class Config {
  static version = "1.2.0";
}

console.log(Config.version); // "1.2.0"
```

Alternativ (abwärtskompatibler):

```
Config.version = "1.2.0";
```

Statische Eigenschaften eignen sich für Konfigurationswerte, globale Zähler oder Klassenkonstanten.

3. Praxisbeispiel: Vergleichsmethoden

Ein typischer Anwendungsfall ist der Vergleich zweier Objekte **dieselben Klasse**.

```
class Vehicle {
  constructor(id) {
    this.id = id;
  }

  static isSameVehicle(v1, v2) {
    return v1.id === v2.id;
  }
}

const a = new Vehicle("A123");
const b = new Vehicle("A123");
console.log(Vehicle.isSameVehicle(a, b)); // true
```

Diese Logik hätte **keinen Sinn** als Instanzmethode, weil sie zwei Objekte vergleichen soll.

4. Methodenbindung und `this`

In JavaScript ist `this` kontextabhängig. Eine Methode, die z. B. an einen Eventhandler übergeben wird, kann ihren Kontext verlieren.

Problem:

```
class Button {
  constructor(label) {
    this.label = label;
  }

  click() {
    console.log(`Clicked: ${this.label}`);
  }
}

const b = new Button("Save");
setTimeout(b.click, 1000); // "Clicked: undefined"
```

Lösung: Arrow Function oder Bind

```
setTimeout(() => b.click(), 1000); // "Clicked: Save"
// oder:
setTimeout(b.click.bind(b), 1000);
```

Alternativ: Methode direkt als Arrow Function deklarieren

```
class Button {
  label = "Save";
  click = () => {
    console.log(`Clicked: ${this.label}`);
  }
}
```

Diese Arrow Function wird beim Instanziieren gebunden und verliert `this` nicht.

Vergleich zu Python:

In Python ist `self` explizit und bleibt stabil – JavaScript braucht mehr Sorgfalt.

5. Zusammenfassung & Best Practices

- **Statische Methoden** eignen sich für Hilfsfunktionen oder Vergleiche
- **Statische Eigenschaften** für Konstanten oder Konfigurationsdaten
- Verwende `static` bewusst: Nur wenn kein Instanzbezug benötigt wird
- Sei vorsichtig mit `this` in Methoden – ggf. Arrow Function oder `.bind()` verwenden
- Nutze Utility-Klassen wie `DateUtils`, `MathTools`, `IdGenerator` gezielt

6. Übungsaufgaben

Aufgabe 1: Statische Methode

Schreibe eine Klasse `StringTools` mit einer statischen Methode `reverse(str)`, die einen String umkehrt.

Aufgabe 2: Konstante als statische Eigenschaft

Erstelle eine Klasse `AppConfig` mit einer statischen Eigenschaft `DEFAULT_LANGUAGE = "de"`. Greife darauf im Code zu.

Aufgabe 3: Vergleichsmethode

Implementiere eine Klasse `User`, mit `id`, und eine statische Methode `areSame(u1, u2)`, die `true` zurückgibt, wenn beide dieselbe `id` haben.

Aufgabe 4: Methodenbindung testen

Was gibt folgender Code aus?

```
class Test {
  name = "Tester";
  log = () => console.log(this.name);
}

const t = new Test();
const f = t.log;
f();
```

Aufgabe 5: Arrow Function vs. normale Methode

Erkläre den Unterschied in Bezug auf `this` bei:

- `someMethod() { ... }`
- `someMethod = () => { ... }`

7. Micro-Projekt: Utility-Klasse für Fahrzeugdaten

Ziel

Erstelle eine Klasse `FleetUtils`, die **statische Methoden** für eine Fahrzeugliste bereitstellt.

Anforderungen

- `countFreeVehicles(fleet)`: Zählt Fahrzeuge mit `status === "free"`
- `findById(fleet, id)`: Gibt Fahrzeugobjekt mit passender `id` zurück
- `compare(v1, v2)`: Gibt `true` zurück, wenn `id` gleich ist

Beispiel

```
```

```
class FleetUtils {
  static countFreeVehicles(fleet) {
    return fleet.filter(v => v.status === "free").length;
  }

  static findById(fleet, id) {
    return fleet.find(v => v.id === id);
  }

  static compare(v1, v2) {
    return v1.id === v2.id;
  }
}

const fleet = [
  { id: "A", status: "free" },
  { id: "B", status: "busy" },
  { id: "C", status: "free" },
];

console.log(FleetUtils.countFreeVehicles(fleet)); // 2
```

Erweiterungsidee

- Methode `groupByStatus(fleet)` erstellt ein Objekt mit Fahrzeugen nach Status gruppiert