Skript 5: Klassen vs. Konstruktorfunktionen – Hinter dem syntaktischen Zucker

Einleitung

JavaScript kennt seit jeher mehrere Wege zur Erstellung von Objekten:

- Objektliterale (`{}`)
- Konstruktorfunktionen
- Prototypenverkettung
- Seit ES6: **Klassen als syntaktischer Zucker**

Dieses Skript stellt Klassen und Konstruktorfunktionen gegenüber, erklärt Gemeinsamkeiten, Unterschiede sowie interne Mechanismen wie die Prototypenkette. Es wird außerdem gezeigt, wann welche Methode sinnvoll ist – und wie sich Klassenmethoden (inkl. `static`) mit der klassischen Funktionsweise abbilden lassen.

1. Konstruktorfunktionen – das alte Modell

Vor ES6 wurden sogenannte Konstruktorfunktionen verwendet:

```
function Vehicle(id) {
  this.id = id;
  this.status = "offline";
}

Vehicle.prototype.setStatus = function(status) {
  this.status = status;
};

const car = new Vehicle("V123");
car.setStatus("ready");
```

Merkmale:

- Funktionen, die mit `new` aufgerufen werden
- Gemeinsame Methoden über das `prototype`-Objekt
- Kein spezielles Sprachkonstrukt für Klassen

2. Klassen – modernes Konstrukt ab ES6

Seit ES6 kann das gleiche Verhalten durch die `class`-Syntax eleganter und strukturierter ausgedrückt werden:

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

  setStatus(status) {
    this.status = status;
  }
}

const car = new Vehicle("V123");
car.setStatus("ready");
```

3. Vergleich: Gemeinsamkeiten

- Beide erzeugen Objekte mit eigener Instanz
- Beide unterstützen Methoden über Prototypen
- Beide nutzen `new` zur Instanzierung
- Beide erlauben das Erweitern per Prototyp (auch bei Klassen, intern)

4. Vergleich: Unterschiede

Aspekt	Konstruktorfunktion	Klasse (ES6+)
Syntax	Funktion mit `this`, `prototype`	`class`-Syntax mit `constructor`
Klarheit	Weniger intuitiv	Klar strukturiert
Methodendeklaration	Am `prototype`	Direkt im Klassenrumpf
`super()`-Unterstützung	Manuell über Aufruf	Eingebaut, verpflichtend bei Vererbung
Statische Methoden	Manuell via `FunctionName.method = ...`	`static`-Schlüsselwort im Rumpf
Fehlerbehandlung	Weniger strikt	Syntaxfehler bei typischen Stolperfallen

5. Prototypenmechanik im Hintergrund

Auch bei Klassen bleibt der **Prototypenmechanismus erhalten**. Der folgende Code zeigt, dass eine Klasseninstanz ebenfalls Zugriff auf Methoden über die Prototypenkette hat:

```
class Tool {
  doSomething() {
    console.log("done");
  }
}

const t = new Tool();
console.log(Object.getPrototypeOf(t) === Tool.prototype); // true
```

Auch mit Konstruktorfunktion:

```
function Tool() {}
Tool.prototype.doSomething = function() { console.log("done"); };
const t = new Tool();
console.log(Object.getPrototypeOf(t) === Tool.prototype); // true
```

Beide Mechanismen greifen auf denselben Unterbau von JavaScript zurück.

6. Klassenmethoden: Instanz vs. static in beiden Varianten

Klasse mit `static`

```
class MathUtil {
  static add(a, b) {
    return a + b;
  }
}

console.log(MathUtil.add(2, 3));
```

Konstruktorfunktion mit "statischer" Methode

```
function MathUtil() {}
MathUtil.add = function(a, b) {
  return a + b;
};

console.log(MathUtil.add(2, 3));
```

Beide Varianten erlauben Methoden, die **nicht instanzgebunden** sind. Die Klassensyntax ist jedoch semantisch klarer.

7. Wann was verwenden?

Szenario	Empfehlung
Objektorientiertes Design	`class`
Legacy-Kompatibilität	Konstruktorfunktion
Dynamische Klassenerzeugung	Klassenausdruck
Utility-Funktionen oder Namespaces	`static` in Klassen oder Funktionsobjekte

Anmerkung

Viele moderne Frameworks (z. B. React) setzen Klassen oder Funktionsobjekte gezielt ein. In modernen Projekten ist `class` die bevorzugte Schreibweise.

8. Übungsaufgaben

Aufgabe 1: Konstruktorfunktion → Klasse

Wandle folgende Funktion in eine Klasse um:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function() {
  console.log("Hi, I'm " + this.name);
};
```

Aufgabe 2: Klasse → Konstruktorfunktion

Wandle folgende Klasse in eine Konstruktorfunktion um:

```
class Book {
  constructor(title) {
    this.title = title;
  }
  info() {
    console.log(this.title);
  }
}
```

Aufgabe 3: Typen vergleichen

Was ergibt folgender Code?

```
function A() {}
class B {}
console.log(typeof A); // ?
console.log(typeof B); // ?
```

Aufgabe 4: Prototypentest

Beweise mit Code, dass Methoden in Klassen ebenfalls im Prototyp gespeichert sind.

Aufgabe 5: static in beiden Varianten

Erstelle eine Utility-Funktion für Temperaturumrechnung als:

- Statische Methode in einer Klasse
- Eigenschaft an einer Funktion

9. Micro-Projekt: Duales Modell – Klasse und Konstruktorfunktion im Vergleich

Ziel

Modellierung einer einfachen `User`-Struktur in beiden Varianten.

Anforderungen

- Attribute: `id`, `username`
- Methode: `describe()` gibt String zurück
- Statische Methode `compare(u1, u2)` vergleicht IDs

Umsetzung

Variante 1: Klasse

```
class User {
  constructor(id, username) {
    this.id = id;
    this.username = username;
  }
  describe() {
    return `${this.username} (${this.id})`;
  }
  static compare(u1, u2) {
    return u1.id === u2.id;
  }
}
```

Variante 2: Konstruktorfunktion

```
function User(id, username) {
  this.id = id;
  this.username = username;
}
User.prototype.describe = function() {
  return `${this.username} (${this.id})`;
};
User.compare = function(u1, u2) {
  return u1.id === u2.id;
};
```

Beide Varianten sind voll funktionsfähig und unterstreichen die konzeptionelle Nähe.