1.5 – JSA: Objekterstellung in JavaScript

Einleitung

Die Art und Weise, wie Objekte in JavaScript erstellt werden können, gehört zu den **zentralsten und gleichzeitig schwierigsten Konzepten** der Sprache. Es existieren verschiedene Strategien und syntaktische Möglichkeiten – jede mit eigener Historie, Semantik, Vor- und Nachteilen.

- welche Wege es gibt, um Objekte zu erzeugen
- wie sich diese Wege voneinander unterscheiden
- welche Muster sich für welche Anwendungsfälle eignen

---

1. Objektliteral – Der direkte Weg

Die Literal-Notation ist die **einfachste und am weitesten verbreitete Methode**, ein Objekt in JavaScript zu erstellen.

```
const person = {
  name: "Anna",
  age: 30,
  greet() {
    return `Hallo, ich bin ${this.name}`;
  }
};
```

Erklärung:

- Wir erstellen ein Objekt mit geschweiften Klammern `{}`.
- `name` und `age` sind Eigenschaften (Keys) mit zugewiesenen Werten.
- `greet()` ist eine Methode (Funktion als Wert).
- Das Schlüsselwort `this` bezieht sich auf das Objekt selbst.

Zweites Beispiel:

```
const product = {
  id: 123,
  name: "Tisch",
  price: 199.99,
  discount(percent) {
    return this.price * (1 - percent / 100);
  }
};

console.log(product.discount(10)); // 179.991
```

Vorteile:

- Schnell, lesbar, intuitiv
- Ideal für Konfigurationen, statische Daten, einfache Module

Nachteile:

- Kein Prototypen-Sharing → Methoden werden pro Objekt neu erzeugt
- Keine Wiederverwendbarkeit bei vielen gleichartigen Objekten

---

2. `Object.create()` – Der kontrollierte Weg

Mit `Object.create()` lässt sich **explizit** ein Prototyp angeben. Es ist die sauberste Methode, um Prototypenvererbung direkt zu nutzen.

```
const animal = {
  speak() {
    return `${this.name} macht Geräusche.`;
  }
};

const dog = Object.create(animal);
dog.name = "Bello";
console.log(dog.speak()); // "Bello macht Geräusche."
```

Erklärung:

- `animal` ist das Prototypobjekt.
- `Object.create(animal)` erzeugt ein neues Objekt `dog`, dessen internes `[[Prototype]]` auf `animal` zeigt.
- Das bedeutet: `dog` erbt die Methode `speak` von `animal`.

Weiteres Beispiel mit Property-Deskriptoren:

```
const cat = Object.create(animal, {
  name: {
    value: "Mieze",
    writable: true,
    enumerable: true
  }
});

console.log(cat.speak()); // "Mieze macht Geräusche."
```

Vorteile:

- Kontrolle über die Prototypkette
- Optimal für gezielte Vererbung
- Gut kombinierbar mit Deskriptoren

Nachteile:

- Weniger gebräuchlich, daher schlechter dokumentiert in einfachen Tutorials
- Initialwerte müssen nachträglich gesetzt werden (außer via Deskriptoren)

---

3. `new Object()` – Historisch, aber selten empfohlen

```
const obj = new Object();
obj.name = "Veraltet";
```

Erklärung:

- Funktional gleichwertig zu `{}`.
- Wurde in früheren JavaScript-Versionen häufiger verwendet.
- Intern ruft `new Object()` den Objektkonstruktor auf, was unnötig ist.

➡️ Nicht falsch – aber **veraltet**. In modernen Projekten **vermeiden**.

---

4. Factory Functions – funktionale Objektfabriken

Eine **Factory Function** ist eine Funktion, die ein neues Objekt erzeugt und zurückgibt:

```
function createUser(name, age) {
  return {
    name,
    age,
    greet() {
      return `Hi, ich bin ${this.name}`;
    }
  };
}

const user1 = createUser("Tom", 25);
console.log(user1.greet());
```

Erklärung:

- Kein `new`, kein Prototyp.
- Rückgabe ist ein Literalobjekt.
- Gut kombinierbar mit Closures.

Beispiel: private Werte mit Closure

```
function createCounter() {
  let count = 0;
  return {
    increment() { count++; },
    get value() { return count; }
  };
}

const counter = createCounter();
counter.increment();
console.log(counter.value); // 1
```

Vorteile:

- Klar strukturiert
- Ermöglicht Kapselung privater Zustände
- Testbar und modular

Nachteile:

- Kein Prototypen-Sharing (jeder Aufruf erzeugt neue Methoden)
- Performance bei vielen Instanzen kann schlechter sein als bei Klassen

---

5. Konstruktorfunktion mit `new`

Vor ES6 waren Konstruktorfunktionen der Hauptweg zur objektorientierten Objekterzeugung mit Prototypenanbindung.

```
function Car(brand) {
  this.brand = brand;
  this.honk = function() {
    return `${this.brand} hupt!`;
  };
}

const bmw = new Car("BMW");
console.log(bmw.honk());
```

Erklärung:

- Der Funktionsname wird großgeschrieben (Konvention)
- `this` bezieht sich beim Aufruf mit `new` auf das neu erzeugte Objekt
- Die Rückgabe geschieht implizit, es sei denn, ein anderes Objekt wird explizit zurückgegeben

Verbesserte Version mit Prototyp:

```
function Car(brand) {
  this.brand = brand;
}

Car.prototype.honk = function() {
  return `${this.brand} hupt!`;
};

const audi = new Car("Audi");
console.log(audi.honk());
```

Vorteile:

- Gemeinsames Teilen von Methoden über Prototyp
- Bessere Performance als Factory Functions bei vielen Instanzen
- Klarer OOP-Stil vor Einführung von `class`

Nachteile:

- Kein privater Zustand ohne Closure-Tricks
- Etwas unhandlicher als moderne `class`-Syntax

Wann welche Methode?

Methode	Wann einsetzen?	Beispielanwendung
`{}`	Schnell, einmalige Nutzung, JSON-Daten	Konfiguration in UI-Komponenten
`Object.create()`	Vererbung kontrollieren, gezielte Prototypverbindungen	Lernbeispiele, Spezial-APIs
Factory Function	Closures, private Daten, funktionaler Stil	API-Wrapper, Services, Hooks
Konstruktorfunktion mit `new`	Strukturierte Objekterstellung mit geteilten Methoden	Klassische OOP-Anwendungen
`new Object()`	Veraltet – vermeiden	Nur noch in Altcode oder Lernbeispielen

---

Prototyp in JavaScript?

Bevor man sich mit Konstruktorfunktionen oder `Object.create()` befasst, ist es essenziell zu verstehen, wie das JavaScript-Vererbungssystem funktioniert.

Der Prototypenmechanismus

JavaScript basiert auf **prototypischer Vererbung**. Das bedeutet: Jedes Objekt besitzt intern eine Referenz auf ein weiteres Objekt – seinen **Prototypen**. Diese Verbindung bildet die sogenannte **Prototypenkette** (*Prototype Chain*).

Wenn JavaScript auf eine Eigenschaft oder Methode zugreift, die **nicht im Objekt selbst** vorhanden ist, wird automatisch im zugehörigen Prototyp gesucht – und weiter im nächsten Prototyp, bis zum Ende der Kette (normalerweise `null`).

```
const point = { x: 0, y: 0 };
const coloredPoint = { color: "red" };
```

```
Object.setPrototypeOf(coloredPoint, point);

console.log(coloredPoint.x); // 0 – kommt vom Prototyp
```

Zugriff auf den Prototyp

JavaScript bietet drei Möglichkeiten, um mit dem Prototyp eines Objekts zu arbeiten:

1. `Object.getPrototypeOf(obj)`

Die **empfohlene Methode**, um den Prototyp eines Objekts zu ermitteln:

```
const proto = Object.getPrototypeOf(coloredPoint);
```

2. `Object.setPrototypeOf(obj, prototype)`

Damit kann der Prototyp eines Objekts **nachträglich gesetzt** werden (nicht performant, aber erlaubt):

```
Object.setPrototypeOf(coloredPoint, point);
```

3. `__proto__` **(veraltet, aber oft sichtbar)**

Diese Eigenschaft ist historisch gewachsen, wird jedoch als **veraltet und langsam** betrachtet. Sie ist weiterhin in vielen Browsern nutzbar:

```
coloredPoint.__proto__ = point; // nicht empfohlen in produktivem Code
```

Beispiel mit Methodenvererbung:

```
const figure = {
  getType() {
    return this.type || "unknown";
  }
};

const circle = {
  type: "circle",
  radius: 100
};

Object.setPrototypeOf(circle, figure);
console.log(circle.getType()); // "circle"
```

Auch wenn `getType` im Prototyp definiert ist, kann sie von `circle` genutzt werden – und `this` bezieht sich beim Aufruf auf `circle`.

Vererbung mit `Object.create()`

Statt `setPrototypeOf` kann man direkt bei der Objekterstellung den Prototyp festlegen:

```
const baseFigure = {
  getType() {
    return this.type || "unknown";
```

```
  }
};

const triangle = Object.create(baseFigure);
triangle.type = "triangle";
triangle.sides = 3;

console.log(triangle.getType()); // "triangle"
```

Erweiterung bestehender Prototypen

Da ein Prototyp selbst ein Objekt ist, kann man ihn **auch nachträglich verändern**:

```
baseFigure.describe = function() {
  return `Dies ist ein(e) ${this.type}`;
};

console.log(triangle.describe()); // "Dies ist ein(e) triangle"
```

Diese Änderung wirkt sich auf **alle Objekte** aus, die `baseFigure` als Prototyp verwenden.

Realitätsnahes Beispiel: String erweitern (mit Vorsicht!)

```
String.prototype.hi = function() {
  return "Hi!";
};

console.log("Hallo".hi()); // "Hi!"
```

Obwohl "Hallo" ein primitiver Wert ist, funktioniert der Aufruf, da JavaScript im Hintergrund eine **temporäre Objektbox (Autoboxing)** erstellt.

Achtung: Das Erweitern nativer Prototypen (wie `String.prototype`) sollte in produktivem Code **vermieden** werden.

Übungsaufgaben

1. Literal vs. Factory

Erstelle ein Objekt per Literal und dasselbe per Factory Function. Vergleiche Speicherverhalten und Wiederverwendbarkeit.

2. Vererbung via `Object.create()`

Erzeuge ein Objekt `vehicle` mit einer Methode `start()`. Erstelle ein neues Objekt `car`, das davon erbt.

3. Verschachtelung mit Factory und Closure

Baue eine Factory `createBankAccount` mit privatem Saldo, `deposit()` und `getBalance()`.

4. Konstruktor + Prototyp testen

Erstelle eine Konstruktorfunktion `Book(title)`. Hänge `describe()` an den Prototyp.

5. Methodenweitergabe mit Prototyp erweitern

Ergänze ein Prototypobjekt mit einer neuen Methode und prüfe, ob sie auf allen erbenden Objekten funktioniert.

Micro-Projekt – Objekterstellung

Projekt: Mitarbeiterverwaltungssystem

Implementiere mehrere Varianten zur Erstellung von Mitarbeiterobjekten.

Anforderungen:

- Erzeuge mindestens zwei Mitarbeiter mit Objektliteral
- Erzeuge mindestens zwei Mitarbeiter mit Factory Function (inkl. Methoden)
- Erzeuge mindestens zwei Mitarbeiter per Konstruktorfunktion mit Prototypmethoden

Bonus:

- Vergleiche die Speicherstruktur und Funktionsweise der drei Varianten
- Gib mit einem Tool (z. B. `console.dir()`) die Objektstruktur visuell aus