1.1 JSA: Grundlagen der Objekte

Einleitung

Objekte gehören zu den zentralen Bausteinen der Programmiersprache JavaScript. Fast alles in JavaScript ist entweder ein Objekt oder kann wie ein Objekt behandelt werden – einschließlich Funktionen, Arrays und selbst viele primitive Werte im erweiterten Kontext.

In dieser Lektion geht es um die Grundlagen:

- Was ist ein Objekt in JavaScript?
- Wie erstellt man Objekte?
- Wie greift man auf Eigenschaften zu?
- Was sind typische Fehler und Besonderheiten beim Umgang mit Objekten?
- Was passiert im Speicher (Garbage Collection)?
- Wie prüft man, ob Eigenschaften existieren?

Was ist ein Objekt?

Ein Objekt in JavaScript ist eine **ungeordnete Sammlung von Schlüssel-Wert-Paaren**. Jeder Schlüssel (Key) ist entweder ein **String** oder ein **Symbol** und verweist auf einen zugehörigen Wert (Value), der jeglichen Typ annehmen kann – von primitiven Werten über Funktionen bis hin zu weiteren Objekten.

Objektliteral – die gebräuchlichste Form

Ein sogenanntes **Object Literal** ist die häufigste und direkteste Art, ein Objekt zu erstellen. Es verwendet geschweifte Klammern `{}`:

```
const person = {
  name: "Anna",
  age: 28,
  isStudent: false
};
```

`person` ist ein Objekt mit drei Eigenschaften:

- `name` (String)
- `age` (Number)
- `isStudent` (Boolean)

Im Hintergrund erzeugt JavaScript eine Referenzstruktur im Speicher. Diese kann automatisch vom sogenannten **Garbage Collector** bereinigt werden, sobald keine Referenz mehr auf das Objekt existiert.

```
let user = { name: "Max" };
user = null; // Das ursprüngliche Objekt ist nun "unreachable" und wird vom Garbage Collector entsorgt.
```

Zugriff auf Eigenschaften

Es gibt zwei grundlegende Möglichkeiten, auf Eigenschaften eines Objekts zuzugreifen:

1. Dot-Notation

```
console.log(person.name); // "Anna"
```

Die Dot-Notation ist leserlich und intuitiv. Sie funktioniert nur, wenn der Eigenschaftsname ein valider Bezeichner ist (keine Leerzeichen, beginnt nicht mit Zahl, keine Sonderzeichen).

2. Bracket-Notation

```
console.log(person["age"]); // 28
```

Die Bracket-Notation ist flexibler und erlaubt auch den Zugriff mit Variablen oder dynamischen Schlüsseln:

```
const key = "isStudent";
console.log(person[key]); // false
```

Wann ist die Bracket-Notation erforderlich?

Wenn der Schlüssel kein gültiger Bezeichner ist (z. B. Leerzeichen oder Sonderzeichen enthält), **muss** die Bracket-Notation verwendet werden:

```
const employee = {
  "full name": "Maria Schmidt",
  "job-title": "Designer"
};

console.log(employee["full name"]); // "Maria Schmidt"
```

Der Zugriff per Dot-Notation würde hier zu einem Syntaxfehler führen:

```
console.log(employee.full name); // ❌ Fehler
```

Eigenschaften hinzufügen, ändern und löschen

Hinzufügen neuer Eigenschaften

```
person.city = "Berlin";
```

Ändern bestehender Eigenschaften

```
person.age = 29;
```

Löschen einer Eigenschaft

```
delete person.isStudent;
```

Eigenschaftsnamen als Symbole

Neben Strings können in JavaScript auch **Symbole (`Symbol`)** als Eigenschaftsnamen verwendet werden. Symbole erzeugen garantiert eindeutige Schlüssel und verhindern unbeabsichtigte Namenskonflikte.

```
const id = Symbol('id');
const user = {
  [id]: 1234
```

```
};

console.log(user[id]); // 1234
```

Wichtig: Symbole werden bei normalen Iterationen über Objekte (`for...in`, `Object.keys()`) nicht angezeigt und müssen speziell behandelt werden.

Symbole sind hauptsächlich in fortgeschrittenen Anwendungen wichtig (z. B. interne Objektmarkierungen oder spezielle Protokolle wie `Symbol.iterator`).

Typische Stolperfallen

1. Zugriff auf nicht existierende Eigenschaften

```
console.log(person.height); // undefined
```

Der Zugriff erzeugt **keinen Fehler**, sondern liefert `undefined`, wenn die Eigenschaft nicht existiert.

2. Unterschied: Eigenschaft fehlt vs. Wert ist `undefined`

```
const obj = { height: undefined };

console.log("height" in obj); // true
console.log(obj.height);      // undefined
console.log("weight" in obj); // false
```

Praxisbeispiel

```
const product = {
  name: "Schreibtisch",
  price: 199.99,
  inStock: true
};

product.category = "Büro";
product.price = 179.99;
delete product.inStock;

console.log(product);
```

Ergebnis:

```
{
  name: "Schreibtisch",
  price: 179.99,
  category: "Büro"
}
```

Iteration über Objekte mit `for...in`

Ein zentraler Bestandteil beim Umgang mit Objekten ist die Fähigkeit, **alle Eigenschaften** eines Objekts zu durchlaufen. Dafür gibt es in JavaScript die Schleife `for...in`.

Syntax:

```
for (let key in object) {
  // Zugriff über object[key]
}
```

Diese Schleife iteriert über **alle aufzählbaren (``) Eigenschaften** eines Objekts **sowie geerbte Eigenschaften über die Prototypenkette**.

Beispiel:

```
const person = {
  name: "Anna",
  age: 28,
  city: "Berlin"
};

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

Ausgabe:

```
name: Anna
age: 28
city: Berlin
```

- **Nicht für Arrays geeignet** → stattdessen `for`, `forEach`, `for...of` verwenden
- Keine garantierte Reihenfolge

Existenzprüfung von Eigenschaften

Es gibt mehrere Möglichkeiten, um zu testen, ob ein Objekt eine bestimmte Eigenschaft besitzt:

1. Zugriff und Vergleich mit `undefined`

Du kannst versuchen, auf eine Eigenschaft zuzugreifen und prüfen, ob der Wert `undefined` ist.

```
const car = {
  brand: "Toyota",
  year: 2020
};

console.log(car.color === undefined); // true
```

Problem:

- Diese Methode unterscheidet nicht zwischen einer fehlenden Eigenschaft und einer existierenden Eigenschaft, deren Wert `undefined` ist.

Beispiel:

```
const car = { color: undefined };

console.log(car.color === undefined); // true (obwohl `color` existiert!)
```

2. Der `in`-Operator

Der `in`-Operator prüft, ob eine Eigenschaft **existiert** – unabhängig davon, ob der Wert `undefined` oder etwas anderes ist.

```
console.log("brand" in car); // true
console.log("color" in car); // true (auch wenn color undefined ist)
```

Syntax:

```
"propertyName" in object
```

3. `hasOwnProperty` Methode

Diese Methode prüft, ob die Eigenschaft **direkt** auf dem Objekt existiert, und ignoriert geerbte Eigenschaften aus der Prototypenkette.

```
console.log(car.hasOwnProperty("brand")); // true
console.log(car.hasOwnProperty("toString")); // false
```

Warum wichtig?

- Viele eingebaute Methoden und Eigenschaften stammen vom Prototyp und sind nicht direkt Teil des Objekts.

Unterschiede zwischen den Prüfmethoden

Methode	Erkennt direkt eigene Eigenschaft	Erkennt geerbte Eigenschaft	Risiko bei undefined Werten
Zugriff + `=== undefined`	✅	❌	Hoch (siehe Problem oben)
`in` Operator	✅	✅	Sicher
`hasOwnProperty`	✅	❌	Sicher

Typische Stolperfallen

1. Unterschied zwischen `undefined` und fehlender Eigenschaft

Nicht jede `undefined`-Rückgabe bedeutet, dass die Eigenschaft fehlt!

2. Geerbte Eigenschaften beachten

Der `in`-Operator findet auch Eigenschaften, die vom Prototyp vererbt wurden. Wenn du nur eigene Eigenschaften prüfen willst, nutze `hasOwnProperty`.

Praxisbeispiel

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

console.log("age" in user); // true
console.log(user.hasOwnProperty("age")); // true

console.log("toString" in user); // true (vom Prototyp geerbt)
console.log(user.hasOwnProperty("toString")); // false
```

Übungsaufgaben

1. Objekt erstellen und Eigenschaften prüfen

Erstelle ein Objekt `book` mit den Eigenschaften `title`, `author` und `pages`. Greife auf die Eigenschaften zu (Dot-Notation, Bracket-Notation) und teste ihre Existenz mit `in` und `hasOwnProperty`.

2. Unterschied zwischen undefined und fehlender Eigenschaft

Erzeuge ein Objekt `animal` mit einer Eigenschaft `species: undefined`. Vergleiche Zugriff und Existenzprüfung.

3. Iteration mit `for...in` + Eigenschaftstest

Erstelle ein Objekt `settings` mit mindestens vier Eigenschaften. Durchlaufe es mit `for...in` und gib nur **eigene Eigenschaften** mit `hasOwnProperty()` aus.

4. Geerbte Eigenschaften erkennen

Prüfe bei einem leeren Objekt `{}`, ob `toString` existiert, und analysiere mit `in` vs. `hasOwnProperty`.

5. Eigene Funktion

Schreibe eine Funktion `checkProperty(obj, prop)`, die `true` zurückgibt, wenn die Eigenschaft direkt auf dem Objekt existiert.

Mini-Mini-Projekt – Grundlagen der Objekte

Projekt: Produktkarte

Erstelle ein Objekt `productCard`, das die Daten eines Onlineshop-Artikels enthält.

Anforderungen:

- Mindestens 5 Eigenschaften (z. B. `title`, `price`, `stock`, `category`, `rating`)
- Mindestens eine verschachtelte Eigenschaft (z. B. `supplier: { name, country }`)
- Mindestens eine Methode, z. B. `printDetails()`, die die Daten als formatierten Text zurückgibt

Bonus:

- Verwende dynamische Eigenschaftsnamen oder Sonderzeichen (z. B. `"in stock"`)
- Nutze Bracket-Notation bewusst für den Zugriff

Mini-Mini-Projekt – Eigenschaften prüfen

Projekt: Zugangskontrolle

Simuliere ein Zugriffssystem, das prüft, ob ein Nutzer bestimmte Rechte hat.

Anforderungen:

- Ein Objekt `userRights` mit verschiedenen Boolean-Flags (z. B. `canEdit`, `canDelete`, `canView`)
- Eine Funktion `checkRight(obj, rightName)`, die prüft, ob das Recht existiert und `true` ist
- Verwende `in` und `hasOwnProperty` gezielt

Bonus:

- Gib aussagekräftige Fehlermeldungen aus, je nachdem ob das Recht fehlt oder `false` ist
- Ermögliche die Übergabe mehrerer Rechte zur Prüfung
