2.2 - JSA: Eigenschaften und Zugriff in Klassen (JavaScript)

Einleitung

- Die klassische Initialisierung über den **Konstruktor**
- Die direkte Deklaration **außerhalb des Konstruktors**, im Rumpf der Klasse (neu seit ES2022)
- **Private Eigenschaften** mit `#`
- Den Einsatz von **Gettern und Settern** zur kontrollierten Abfrage und Manipulation

---

1. Eigenschaften im Konstruktor definieren

Dies ist die "klassische" Methode, die mit der Einführung von Klassen übernommen wurde.

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

const bike = new Vehicle("B123");
console.log(bike.status); // "offline"
```

- `this.status = ...` erzeugt beim Instanziieren ein Feld `status` auf dem konkreten Objekt (`bike`).
- Diese Felder sind **Instanzeigenschaften**.

Python-Vergleich

```
class Vehicle:
    def __init__(self, id):
        self.id = id
        self.status = "offline"

bike = Vehicle("B123")
print(bike.status)  # "offline"
```

In beiden Sprachen erzeugt die Initialisierung innerhalb des Konstruktors **Instanzattribute**. Bis hier ist alles analog.

---

2. Eigenschaften im Klassenrumpf deklarieren

Seit ES2022 können Eigenschaften **direkt im Rumpf** der Klasse deklariert werden:

```
class Vehicle {
  status = "offline";

  constructor(id) {
    this.id = id;
  }
}

const car = new Vehicle("C001");
console.log(car.status); // "offline"
```

Wichtig: Das ist **kein** Klassenattribut!

Diese Eigenschaft wird **bei jedem Aufruf von `new Vehicle(...)` erneut auf das Objekt kopiert**.

Das heißt: `status` ist ebenfalls eine **Instanzeigenschaft**, obwohl sie **außerhalb des Konstruktors** steht.

Python-Vergleich: Unterschiedliches Verhalten

```
class Vehicle:
    status = "offline"  # Klassenattribut

    def __init__(self, id):
        self.id = id

car1 = Vehicle("C1")
car2 = Vehicle("C2")

Vehicle.status = "maintenance"
print(car1.status)  # "maintenance"
print(car2.status)  # "maintenance"
```

In Python wird `status` **einmalig für die Klasse definiert** und von allen Instanzen gemeinsam genutzt – **sofern es nicht überschrieben wird**.

In JavaScript dagegen wird `status = "offline"` **in jede neue Instanz kopiert**.

Beweis: Vergleich der Property-Inhaber

```
console.log(car.hasOwnProperty("status")); // true
console.log(Vehicle.status); // undefined
```

Best Practice:

Eigenschaften im Rumpf der Klasse deklarieren, wenn:

- der Wert für **alle Instanzen gleich beginnt**
- kein Zugriff auf `constructor`-Argumente erforderlich ist

---

3. Private Eigenschaften mit `#`

Mit `#` werden Eigenschaften als **privat** markiert – d. h. sie sind **nicht von außen sichtbar**.

```
class Vehicle {
  #latitude;
  #longitude;

  constructor(id, latitude, longitude) {
    this.id = id;
    this.setPosition(latitude, longitude);
  }

  setPosition(lat, lon) {
    this.#latitude = lat;
    this.#longitude = lon;
  }

  getPosition() {
    return {
      latitude: this.#latitude,
      longitude: this.#longitude
    };
  }
```

```
}

const v = new Vehicle("A100", 59.33, 18.06);
console.log(v.getPosition());
console.log(v.#latitude); // SyntaxError
```

- Nur Methoden innerhalb der Klasse haben Zugriff auf `#latitude` und `#longitude`.
- Diese Felder sind nicht über `this` zugänglich.

Vergleich zu Python

```
class Vehicle:
    def __init__(self, id, lat, lon):
        self.__latitude = lat
        self.__longitude = lon

    def get_position(self):
        return (self.__latitude, self.__longitude)
```

- In Python sind Attribute mit `__` nur **eingeschränkt zugänglich** (Name mangling).
- In JavaScript ist `#` **echte Kapselung auf Sprachebene**.

---

4. Getter & Setter

Getter und **Setter** ermöglichen Zugriff wie auf eine Eigenschaft, obwohl intern Methoden genutzt werden:

```
class Vehicle {
  constructor(id, lat, lon) {
    this.id = id;
    this._latitude = lat;
    this._longitude = lon;
  }

  get position() {
    return {
      latitude: this._latitude,
      longitude: this._longitude
    };
  }

  set position({latitude, longitude}) {
    this._latitude = latitude;
    this._longitude = longitude;
  }
}

const v = new Vehicle("V100", 59.33, 18.06);
console.log(v.position); // getter
v.position = { latitude: 59.36, longitude: 18.10 }; // setter
```

Vorteile:

- Einheitliche Zugriffssyntax: `obj.property` statt `obj.getProperty()`
- Validierung, Formatierung oder Logging möglich

Einschränkung:

- Setter darf nur **ein Argument** akzeptieren
- Getter **darf kein Argument** erwarten

Python-Vergleich (mit `@property`):

```python
class Vehicle:
    def __init__(self, id, lat, lon):
        self._latitude = lat
        self._longitude = lon

    @property
    def position(self):
        return (self._latitude, self._longitude)

    @position.setter
    def position(self, pos):
        self._latitude, self._longitude = pos
```

---

5. Zusammenfassung & Best Practices

- Verwende den **Konstruktor**, wenn Eigenschaften vom Aufruf abhängen
- Deklariere konstante Defaultwerte direkt im **Klassenrumpf**
- Nutze `#private`, um Daten zu kapseln
- Nutze `get/set`, um kontrollierten Zugriff zu ermöglichen
- In JavaScript ist **jede Eigenschaft instanzspezifisch**, auch wenn sie im Klassenrumpf steht
- In Python sind solche Felder standardmäßig **Klassenattribute**

---

6. Übungsaufgaben

Aufgabe 1: Begriffsabgrenzung

- Erkläre den Unterschied zwischen Instanz- und Klassenattribut in JS und Python.
- Was bewirkt das `#` in einer JavaScript-Klasse?

Aufgabe 2: Fehlerdiagnose

Was ist an folgendem Code falsch?

```javascript
class Device {
  #name = "Default";

  constructor(name) {
    #name = name;
  }
}
```

Aufgabe 3: Getter & Setter

Erstelle eine Klasse `Rectangle`, die folgende Eigenschaften hat:

- `width`, `height`
- Getter `area`, der die Fläche zurückgibt
- Setter `dimensions`, der beide Eigenschaften gleichzeitig setzt

Aufgabe 4: Unterschied zeigen

Beweise mit Code, dass ein direkt im Klassentext deklariertes Feld in JS kein Klassenattribut ist.

Aufgabe 5: Eigene Property-Strategie

Schreibe eine Klasse `User`, die `username` über einen Getter liefert, aber das interne Attribut `#name` kapselt.

---

7. Micro-Projekt: Positionstracker mit Zugriffskontrolle

Ziel

Entwickle eine Klasse `Tracker`, die Geopositionen verwaltet. Verwende **private Eigenschaften**, sowie **Getter und Setter**, um auf Positionen zuzugreifen.

Anforderungen

- Konstruktor mit `id`, `latitude`, `longitude`
- Private Felder für `latitude`, `longitude`
- Getter `position`, Setter `position`
- Methode `resetPosition()` setzt Position auf `(0,0)`

Beispiel

```
const t = new Tracker("T001", 59.32, 18.06);
console.log(t.position); // { latitude: 59.32, longitude: 18.06 }
t.position = { latitude: 0, longitude: 0 };
t.resetPosition();
```

Erweiterungsidee

- Positionshistorie als Array privat speichern
- Methode `getHistory()` gibt alle gespeicherten Positionen zurück

Nächster Schritt: In Skript 3 lernst du, wie du Klassen vererbst und Methoden überschreibst. Dabei vergleichen wir auch mit Python-Konstrukten wie `super()` und Mehrfachvererbung.