1.4 JSA: Property Attributes – Eigenschaften von Eigenschaften

Einleitung

In JavaScript ist eine Objekteigenschaft nicht einfach nur ein Key mit einem Wert. Jede Eigenschaft hat zusätzlich eine Reihe von **internen Attributen**, die festlegen, **wie** diese Eigenschaft funktioniert: Ob sie überschreibbar ist, ob sie sichtbar ist, ob sie gelöscht werden darf usw.

- Metainformationen einer Eigenschaft
- Property Attributes sichtbar machen
- Unterschied von `data properties` und `accessor properties`
- Objekte gegen Veränderung schützen (`Object.preventExtensions`, `Object.seal`, `Object.freeze`)

Grundlagen: Was sind Property Attributes?

Jede Eigenschaft in einem JavaScript-Objekt besitzt **unsichtbare Kontrollinformationen**. Dazu gehören:

Attribut	Bedeutung
`value`	Der tatsächliche Wert der Eigenschaft (nur bei Data Properties)
`writable`	Kann der Wert geändert werden?
`enumerable`	Wird die Eigenschaft bei Iterationen wie `for...in` oder `Object.keys()` berücksichtigt?
`configurable`	Kann die Eigenschaft gelöscht oder erneut konfiguriert werden?
`get` / `set`	Zugriffsfunktionen bei Accessor Properties

Property-Typen: Data vs. Accessor Properties

1. Data Property

Ein normaler Schlüssel-Wert-Paar-Eintrag:

```
const user = {
  name: "Anna"
};
```

Hier liegt eine Data Property `name` mit dem Wert "Anna" vor.

2. Accessor Property

Hierbei handelt es sich um Eigenschaften mit `get` und/oder `set`:

```
const user = {
  get name() {
    return "Anna";
  }
};
```

Accessors haben **keinen `value` und `writable`**, sondern `get` und `set` Funktionen.

Property Attributes anzeigen

Verwende `Object.getOwnPropertyDescriptor()`:

```
const obj = { name: "Anna" };
const desc = Object.getOwnPropertyDescriptor(obj, "name");
console.log(desc);
```

Ausgabe:

```
{
  value: "Anna",
  writable: true,
  enumerable: true,
  configurable: true
}
```

Property Attributes ändern

Verwende `Object.defineProperty()`:

```
const obj = {};
Object.defineProperty(obj, "secret", {
  value: 1234,
  writable: false,
  enumerable: false,
  configurable: false
});

console.log(obj.secret);       // 1234
obj.secret = 9999;
console.log(obj.secret);       // bleibt 1234
```

Kombination mit Accessor Properties

```
const person = {};

Object.defineProperty(person, "fullName", {
  get() {
    return this.first + " " + this.last;
  },
  set(value) {
    [this.first, this.last] = value.split(" ");
  },
  enumerable: true,
  configurable: true
});

person.fullName = "Tom Becker";
console.log(person.first);     // Tom
console.log(person.last);      // Becker
console.log(person.fullName);  // Tom Becker
```

Muss man alle Property Attributes gleichzeitig angeben?

Beim Einsatz von `Object.defineProperty()` ist es nicht erforderlich, alle möglichen Attribute (`writable`, `enumerable`, `configurable`) gleichzeitig zu definieren. Wichtig ist jedoch: **Attribute, die nicht explizit angegeben werden, erhalten automatisch Standardwerte** – und diese sind oft restriktiver als erwartet.

Wird beispielsweise nur ein `value` gesetzt, gelten implizit folgende Einstellungen:

```
{
  value: 123,
  writable: false,
  enumerable: false,
  configurable: false
}
```

Das bedeutet, dass die Eigenschaft:

- **nicht beschreibbar** ist (`writable: false`)
- **nicht aufgezählt** wird in `for...in` oder `Object.keys()`
- **nicht gelöscht** oder umkonfiguriert werden kann

Konsequenzen von `configurable: false`

Wird eine Eigenschaft mit `configurable: false` definiert, lässt sie sich **nicht mehr löschen** und ihre Attributstruktur **kann nicht mehr verändert** werden. Das gilt insbesondere für:

- Änderungen an `writable`, `enumerable` oder `get`/`set`
- spätere Versuche, `Object.defineProperty` erneut aufzurufen

Beispiel:

```
Object.defineProperty(obj, "locked", {
  value: 42,
  configurable: false
});

Object.defineProperty(obj, "locked", { writable: true }); // TypeError
```

Hinweis: Das einmalige Setzen von `configurable: false` ist **irreversibel**. Diese Eigenschaft wird dauerhaft geschützt und eingefroren.

Objekte absichern: Erweiterbarkeit und Schutzmechanismen

JavaScript bietet drei Methoden, um Objekte gegen Veränderungen zu schützen:

1. `Object.preventExtensions()`

Verhindert das **Hinzufügen neuer Eigenschaften**, erlaubt aber weiterhin Änderungen und Löschen vorhandener Eigenschaften.

```
const obj = { a: 1 };
Object.preventExtensions(obj);

obj.b = 2; // Ignoriert im Strict Mode: TypeError
console.log(obj.b); // undefined
```

Test:

```
console.log(Object.isExtensible(obj)); // false
```

2. `Object.seal()`

- Verhindert das Hinzufügen **und Löschen** von Eigenschaften

- Bestehende Eigenschaften können aber noch **verändert** werden (wenn `writable: true`)

```
const obj = { x: 10 };
Object.seal(obj);

delete obj.x;        // funktioniert nicht
obj.x = 20;          // funktioniert
console.log(obj.x);  // 20
```

Test:

```
console.log(Object.isSealed(obj)); // true
```

3. `Object.freeze()`

- Komplettschutz: keine neuen Eigenschaften, keine Löschung, keine Änderungen

```
const obj = { y: 5 };
Object.freeze(obj);

obj.y = 100;
obj.z = 200;
delete obj.y;

console.log(obj); // { y: 5 }
```

Test:

```
console.log(Object.isFrozen(obj)); // true
```

Hinweis: Diese Methoden sind besonders nützlich in sicherheitsrelevanten Anwendungen oder um APIs „einzufrieren“.

Warum sind Property Attributes wichtig?

- Du kannst so Objekte **vor versehentlichen Änderungen schützen**.
- Du kannst **interne APIs kapseln** und steuern, was sichtbar ist.
- Viele Frameworks (z. B. Vue, Angular) nutzen diese Mechanismen gezielt.
- Auch für Prüfungsaufgaben (z. B. „Was passiert bei Zugriff auf eine nicht writable property?“) ist das Verständnis entscheidend.
- Erweiterte Methoden wie `Object.freeze` sind **praktisch relevant** in der Alltagsentwicklung (z. B. Redux State, Immutability Patterns).

Praxisbeispiele

```
const data = {};
Object.defineProperty(data, "id", {
  value: 1001,
  writable: false,
  enumerable: true,
  configurable: false
});

console.log(data.id);    // 1001

for (let key in data) {
```

```
  console.log(key);      // "id"
}

data.id = 2000;
console.log(data.id);    // bleibt 1001
```

```
const config = { env: "dev" };
Object.freeze(config);
config.env = "prod";
console.log(config.env); // "dev"
```

Übungsaufgaben

1. Property Descriptor anzeigen

Erzeuge ein Objekt `book` mit einer Eigenschaft `title`. Verwende `Object.getOwnPropertyDescriptor()` um die Attribute anzuzeigen.

2. Nicht beschreibbare Eigenschaft

Erstelle eine Eigenschaft `isbn` im Objekt `book`, die nicht beschreibbar (`writable: false`) ist. Teste, ob der Wert geändert werden kann.

3. Nicht konfigurierbare Eigenschaft

Erstelle eine Eigenschaft `internalId`, die nicht gelöscht werden kann. Versuche sie mit `delete` zu entfernen.

4. Unsichtbare Eigenschaft

Erstelle eine Eigenschaft `hiddenField`, die nicht bei Iterationen erscheint. Nutze eine Schleife, um dies zu überprüfen.

5. Accessor Property erstellen

Definiere im Objekt `user` eine Accessor Property `password`, die intern zwei Variablen `salt` und `hash` verwaltet.

6. `preventExtensions`, `seal` und `freeze`

Erzeuge ein Objekt `config`, das du nacheinander mit allen drei Schutzmechanismen versiehst. Teste jeweils durch Hinzufügen, Ändern und Löschen von Eigenschaften.

Micro-Projekt – Property Attributes

Projekt: Sicheres Benutzerobjekt

Baue ein Benutzerobjekt `userAccount`, bei dem bestimmte Eigenschaften geschützt oder eingeschränkt sind.

Anforderungen:

- Verwende `Object.defineProperty()` für folgende Eigenschaften:
  - `id`: nicht veränderbar, nicht löschbar
  - `password`: nicht sichtbar bei Iteration
- Ergänze Getter/Setter für `passwordMasked`, z. B. um „******“ zurückzugeben

Bonus:

- Verwende `Object.preventExtensions()` oder `Object.freeze()` auf dem Objekt
- Schreibe eine kleine Prüf-Funktion `isProtected(obj)`