JavaScript 05

OBJEKTORIENTIERTE PROGRAMMIERUNG

OBJEKTORIENTIERUNG

- In Objekten können verschiedene Felder definiert werden
- Feldern können Werte zugewiesen werden
- Mit `new Object()` wird ein neues Objekt erstellt

```
let meineKaffeemaschine = new Object();
meineKaffeemaschine.Farbe = "schwarz";
alert(meineKaffeemaschine.Farbe);

// alternative Schreibweise
meineKaffeemaschine["Farbe"] = "schwarz";
alert(meineKaffeemaschine["Farbe"]);
```

FOR-IN-SCHLEIFE

- Ähnlich wie for-of Schleife
- Gibt aber Bezeichnung für alle Eigenschaften des Objekts aus
- In diesem Fall ist nur Schreibweise mit einer eckigen Klammer zulässig

```
"use strict";
let meineKaffeemaschine = new Object();
meineKaffeemaschine.Farbe = "schwarz";
meineKaffeemaschine.Verbrauch = 500;
meineKaffeemaschine["max. Menge(1)"] = 0.75;

for (let i in meineKaffeemaschine) {
    document.write("<strong>" + i + "</strong>: " + meineKaffeemaschine[i] + "<br>");
}
```

Farbe: schwarz
Verbrauch: 500
max. Menge(1): 0.75

OBJEKTORIENTIERUNG

- Objekte erlauben Vorgabe genauer Strukturen = immer gleiche Eigenschaften in einem Objekt
- In meisten OOP Sprachen kommen hierfür Klassen zum Einsatz
- Früher in JS nicht möglich, deshalb klassenlose objektorientierte Programmiersprache
- Seit ECMAScript 6 gibt es in JavaScript ebenfalls Klassen

Funktionen als Konstruktor verwenden

- Wenn man ein Objekt erzeugt, kann man dafür einen Konstruktor verwenden = spezielle Funktion – dem neuen Objekt werden direkt die gewünschten Werte zugewiesen
- Um Konstruktor zu erzeugen => gewöhnliche Funktion verwenden, die Eigenschaften des Objekts als Übergabewerte erhält => entsprechende Variablen definieren
- Das `this` bezieht sich immer auf das aktuelle Objekt das durch Funktion erzeugt wird

```
// Konstruktor gestalten
function Kaffeemaschine(farbe, preis, verbrauch, typ) {
    this.Farbe = farbe;
    this.Preis = preis;
    this.Verbrauch = verbrauch;
    this.Typ = typ;
}
// Objekt erzeugen
let meineKaffeemaschine = new Kaffeemaschine("silber", 39.99, 600, "Kaffeevollautomat");
// Eigenschaften des Objektes ausgeben
for (let i in meineKaffeemaschine) {
    document.write("<strong>" + i + "</strong>: " + meineKaffeemaschine[i] + "<br>");
}
```

Vererbung durch Prototypen

- Wesentliches Grundprinzip der OOP = Vererbung
- Ermöglicht es, verschiedene Objekttypen in über- und untergeordnete Kategorien einzuteilen
- BSP: Kaffeemaschine => übergeordnete Kategorie „elektrische Küchengeräte“
- Vorteile:
  - Vorhandene Funktionen lassen sich für beliebig viele untergeordnete Objekttypen als Prototyp verwenden
  - Klare Strukturen
  - Macht Verbindung zwischen einzelnen Objekten deutlich

```
"use strict";
// übergeordnete Funktion definieren
function ElekGeraete(farbe, preis, verbrauch) {
    this.Farbe = farbe;
    this.Preis = preis;
    this.Verbrauch = verbrauch;
}
// untergeordnetes Objekt ableiten
// Konstruktor-Funktion
function Kaffeemaschine (farbe, preis, verbrauch, typ) {
    // Konstruktor-Funktion für ElekGeraete als Attribut einfügen
    // base ist üblich da es sich um Basis-Funktion des Objektes handelt
    this.base = ElekGeraete;
    // entsprechende Werte aus Konstruktorfunktion übergeben
    this.base(farbe, preis, verbrauch);
    this.Typ = typ;
}
Kaffeemaschine.prototype = new ElekGeraete;
let meineKaffeemaschine = new Kaffeemaschine("silber", 39.99, 600, "Kaffeevollautomat");

for (let i in meineKaffeemaschine) {
    document.write("<strong>" + i + "</strong>: " + meineKaffeemaschine[i] + "<br>");
}
```

KLASSEN IN JAVASCRIPT

- Klassen in JS basieren auf Funktionen
- Um Struktur für entsprechendes Objekt vorzugeben muss ein Konstruktor erstellt werden

```
// Klasse erstellen
class eGeraete {
    constructor(farbe, preis, verbrauch) {
        this.Farbe = farbe;
        this.Preis = preis;
        this.Verbrauch = verbrauch;
    }
}
// abgeleitete Klasse erstellen (extends => verweist auf Basisklasse
class Kaffeemaschine extends eGeraete {
    // Konstruktor erstellen
    constructor(farbe, preis, verbrauch, typ) {
        // Konstruktor der Basisklasse aufrufen
        super(farbe, preis, verbrauch);
        this.Typ = typ;
    }
}
let meineKaffeemaschine = new Kaffeemaschine("silber", 39.99, 600, "Kaffeevollautomat");

for (let i in meineKaffeemaschine) {
    document.write("<strong>" + i + "</strong>: " + meineKaffeemaschine[i] + "<br>");
}
```

Methoden erstellen und anwenden mit Funktionen

- Objekte erlauben es, bestimmte Aktionen mit ihnen durchzuführen => Methoden zum Einsatz
- Ähnlich aufgebaut wie Funktionen
- Beziehen sich stets auf ein Objekt

```
"use strict";
// Methode erstellen
function Rabatt(wert) {
    this.Preis *= 1 - wert/100;
}
function Kaffeemaschine(farbe, preis, verbrauch, typ) {
    this.Farbe = farbe;
    this.Preis = preis;
    this.Verbrauch = verbrauch;
    this.Typ = typ;
    // in Konstruktor-Funktion einbinden => wird zu einer Methode des Objekts
    this.Rabatt = Rabatt;
}

let meineKaffeemaschine = new Kaffeemaschine("silber", 39.99, 600, "Kaffeevollautomat");
meineKaffeemaschine.Rabatt(10);
for (let i in meineKaffeemaschine) {
    document.write("<strong>" + i + "</strong>: " + meineKaffeemaschine[i] + "<br>");
}
```

Farbe: silber
Preis: 35.991
Verbrauch: 600
Typ: Kaffeevollautomat
Rabatt: function Rabatt(wert) { this.Preis *= 1 - wert/100; }

Methoden erstellen und anwenden mit Klassen

- Etwas einfacher mit Klassen

```
class Kaffeemaschine {
    constructor(farbe, preis, verbrauch, typ) {
        this.Farbe = farbe;
        this.Preis = preis;
        this.Verbrauch = verbrauch;
        this.Typ = typ;
    }
    Rabatt(wert) {
        this.Preis *= 1 - wert/100;
    }
}

let meineKaffeemaschine = new Kaffeemaschine("silber", 39.99, 600, "Kaffeevollautomat");
meineKaffeemaschine.Rabatt(10);
for (let i in meineKaffeemaschine) {
    document.write("<strong>" + i + "</strong>: " + meineKaffeemaschine[i] + "<br>");
}
```

DATENKAPSELUNG IN JS

- Nutzen: Im Objekt gespeicherte Daten schützen
- Methoden als öffentlich deklarieren = ermöglichen Zugriff aus jedem beliebigen Teil des Programms
- Vorteil:
  - in der Klassendefinition vorgeben, welche Werte aus dem Hauptprogramm zugänglich sein sollen
  - Genaue Definition, ob und auf welche Weise Werte verändert werden können
- JS erlaubt keine private Attribute zu deklarieren => Zugang aus Hauptprogramm immer möglich
  - Konvention: Namen und vorangestellter Unterstrich bedeuten das die Variable nur für internen Gebrauch innerhalb der Klasse bestimmt ist
- Die getter-Methode dient dazu, Wert abzufragen
- Die setter-Methode dient dazu, Wert zu verändern.

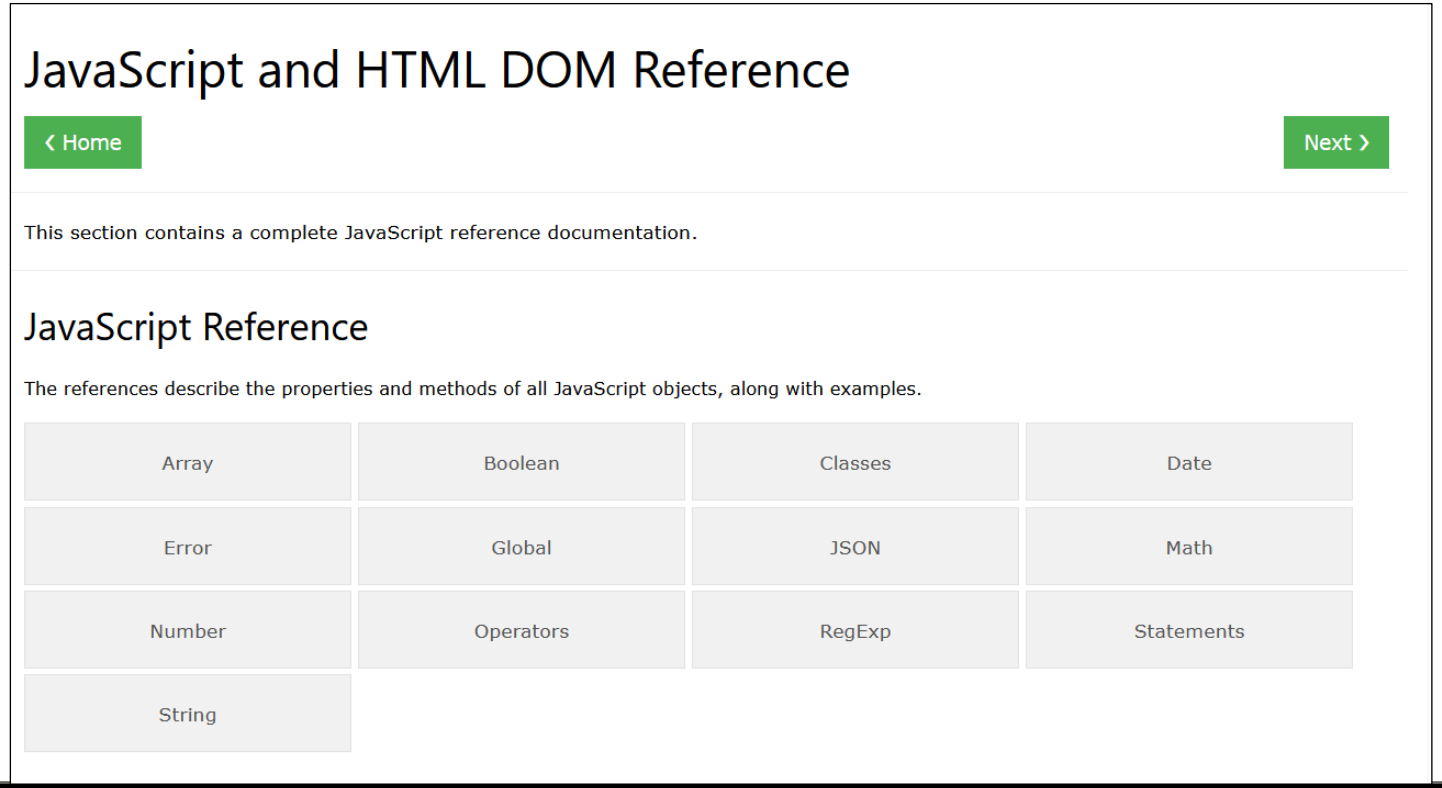
DATENKAPSELUNG IN JS

```
class Kaffeemaschine {
    constructor(farbe, preis, verbrauch, typ) {
        this._Farbe = farbe;
        this._Preis = preis;
        this._Verbrauch = verbrauch;
        this._Typ = typ;
    }

    // getter-Methoden
    get Farbe() {
        return this._Farbe;
    }
    get Preis() {
        return this._Preis;
    }
    get Verbrauch() {
        return this._Verbrauch;
    }
    get Typ() {
        return this._Typ;
    }
    // setter-Methoden
    set Farbe(wert) {
        this._Farbe = wert;
    }
    // bestimmte Vorgaben unterbinden
    set Preis(wert) {
        if(wert >= 0) {
            this._Preis = wert;
        } else {
            alert("Preis darf nicht negativ sein!");
        }
    }
}

let meineKaffeemaschine = new Kaffeemaschine("silber", 39.99, 600,
"Kaffeevollautomat");
document.write("Farbe: " + meineKaffeemaschine.Farbe + "<br>");
meineKaffeemaschine.Preis = 42.99;
document.write("Preis: " + meineKaffeemaschine.Preis);
```

Vorgefertigte Objekte und Methoden

- JS besitzt bereits eine Menge vorgefertigter Objekte die viele verschiedene Methoden enthalten
  https://www.w3schools.com/jsref/

```
// Math.random(): Zufälliger Wert zwischen 0 und (exklusive) 1 mal 1000
// Math.floor(): gibt Zahl zurück die vor dem Komma steht
let zufall = Math.floor(Math.random() * 1000);
document.write("Die Wurzel aus " + zufall + " ist " + Math.sqrt(zufall) + ".");
```



AUFGABE

1. Erzeuge mit einer Funktion die Struktur für ein Objekt, das einen Spieler in einem Computerspiel repräsentiert. Dieses soll als Attribut den Namen des Spielers enthalten. Außerdem soll eine Zahl vorhanden sein, die das Feld repräsentiert, auf dem er sich befindet. Erstelle ein Programm, das ein entsprechendes Objekt erzeugt und dessen Werte ausgibt.

```
"use strict";
// Ausgabe der Attribute des Objekts
function Spieler(name, feld) {
    this.Name = name;
    this.Feld = feld;
}
let Spieler1 = new Spieler("Katharina", 5);
for (let i in Spieler1) {
    document.write(i + ": " + Spieler1[i] + "<br>");
}
```

Name: Katharina
Feld: 5

AUFGABE

1. Erstelle nun eine Methode. Diese soll dazu dienen, einen weiteren Zug durchzuführen. Sie erhält als Übergabewert die Anzahl der Felder, die der Spieler in diesem Zug vorrücken soll. Verwende hierfür ebenfalls eine Funktion. Wende die Methode auf das Spieler-Objekt an

```
"use strict";
// Methode hat den Spielstand verändert und wird nun als Attribut angezeigt
function Zug(wert) {
    this.Feld += wert;
}
function Spieler(name, feld) {
    this.Name = name;
    this.Feld = feld;
    this.Zug = Zug;
}
let Spieler1 = new Spieler("Katharina", 5);
Spieler1.Zug(3);
for (let i in Spieler1) {
    document.write(i + ": " + Spieler1[i] + "<br>");
}
```

Name: Katharina
Feld: 8
Zug: function Zug(wert) { this.Feld += wert; }

AUFGABE

1. Erstelle ein neues Programm, das genau die gleiche Aufgabe erfüllt wie in der vorherigen Übungsaufgabe. Verwende jedoch dieses Mal für die Erstellung eine Klasse.

```
// Methode verändert ebenfalls den Wert, wird aber nicht mehr als Attribut angezeigt
class Spieler {
    constructor(name, feld) {
        this.Name = name;
        this.Feld = feld;
    }
    Zug(wert) {
        this.Feld += wert;
    }
}

let Spieler1 = new Spieler("Katharina", 5);
Spieler1.Zug(3);
for (let i in Spieler1) {
    document.write(i + ": " + Spieler1[i] + "<br>");
}
```

Name: Katharina
Feld: 8

ENDE

QUELLE: JAVASCRIPT
PROGRAMMIEREN FÜR EINSTEIGER
ISBN: 978-3-96645-016-4