3.6 – JSA: Utility-Objekte – JSON, Math & RegExp

Einleitung

Neben klassischen Datentypen und Strukturen bietet JavaScript eine Reihe eingebauter Utility-Objekte, die systemweit verwendet werden:

- **JSON** zur Umwandlung von Objekten in textbasierte Formate (z. B. zur Speicherung oder Übertragung über Netzwerke).
- **Math** zur Durchführung mathematischer Operationen – z. B. Runden, Potenzen, Zufallszahlen.
- **RegExp** (Regular Expressions) zur Suche, Prüfung und Manipulation von Zeichenketten auf Basis von Mustern.

Diese Objekte sind **nicht instanziierbar** wie etwa Klassen, sondern global verfügbare „Werkzeuge“ mit vielen nützlichen Funktionen.

---

1. JSON – Daten serialisieren und rekonstruieren

1.1 `JSON.stringify()` – Objekt → JSON-String

Funktion: Wandelt Objekte, Arrays oder gemischte Datenstrukturen in einen **gültigen JSON-Text** um, der als String gespeichert oder übertragen werden kann.

Syntax:

```
JSON.stringify(value[, replacer[, space]])
```

Beispiel:

```
const vehicle = {
  id: "AK12113",
  latitude: 59.3586,
  longitude: 17.9476,
  getId: function() { return this.id; }
};

const json = JSON.stringify(vehicle);
console.log(json);
// → {"id":"AK12113","latitude":59.3586,"longitude":17.9476}
```

Erklärung:

- Methoden (`getId`) werden **nicht** serialisiert.
- JSON erlaubt nur Daten (keine Funktionen oder Prototypen).

1.2 `JSON.parse()` – JSON-String → Objekt

Funktion: Wandelt einen JSON-Text (z. B. aus dem Netzwerk) zurück in ein JavaScript-Objekt oder Array.

Syntax:

```
JSON.parse(text[, reviver])
```

Beispiel:

```
const json = '{"id":"AK12113","latitude":59.3586,"longitude":17.9476}';
const obj = JSON.parse(json);
console.log(obj.latitude); // 59.3586
```

Hinweise:

- JSON verlangt **doppelte Anführungszeichen** ("), keine einfachen ('').
- Nur ein **Top-Level-Objekt oder Array** erlaubt.
- Zirkuläre Objekte (z. B. self-referencing) führen zu Fehlern.

---

2. Math – mathematische Hilfsfunktionen

2.1 Konstanten

```
console.log(Math.PI); // Kreiszahl π ≈ 3.1415
console.log(Math.E);  // Eulersche Zahl ≈ 2.718
```

2.2 Runden

```
Math.round(10.5); // → 11: Mathematisch korrekt gerundet
Math.floor(10.5); // → 10: Immer nach unten
Math.ceil(10.2);  // → 11: Immer aufrunden
```

2.3 Zufallszahlen

```
Math.random(); // → z. B. 0.572839 – zwischen 0 (inkl.) und 1 (exkl.)

// eigene Zufallszahlfunktion:
const randomInt = (min, max) =>
  Math.floor(Math.random() * (max - min + 1)) + min;

randomInt(1, 6); // → Würfel: 1–6
```

2.4 Weitere Methoden

```
Math.abs(-5);              // → 5 (Betrag)
Math.min(3, 1, 7);          // → 1
Math.max(...[1, 2, 3]);     // → 3 (Spread nötig bei Arrays)
Math.pow(2, 3);             // → 8 (2 hoch 3)
Math.sqrt(16);              // → 4 (Wurzel)
Math.log10(1000);           // → 3 (Zehnerlogarithmus)
```

2.5 Trigonometrie (in Radiant!)

```
Math.cos(Math.PI / 3);   // → 0.5
Math.tan(Math.PI / 4);   // → 1
```

---

3. RegExp – Reguläre Ausdrücke

3.1 Konstruktion

```
let re = new RegExp("c.t"); // Pattern via String
let re2 = /c.t/;             // Literalnotation
```

3.2 `test()` und `exec()`

```
re.test("cat");      // → true (passt auf c–t)
re.exec("haircut");  // → ["cut", index: 4, input: "haircut", ...]
```

3.3 String-Integration

```
let str = "dog and cat";
str.match(/c.t/);              // → ["cat"]
str.search(/c.t/);             // → 8 (Position im Text)
str.replace(/c.t/, "fox");     // → "dog and fox"
```

3.4 Wichtige Metazeichen (Auswahl)

Symbol	Bedeutung
`.`	ein beliebiges Zeichen
`+`	mindestens 1 Wiederholung
`*`	0 oder mehr Wiederholungen
`?`	optional (0 oder 1x)
`^`	Anfang der Zeichenkette
`$`	Ende der Zeichenkette
`[abc]`	a **oder** b **oder** c
`[^abc]`	nicht a, b oder c
`\d`	Ziffer (0–9)
`\s`	Whitespace (Leerzeichen, Tab etc.)

Beispiel:

```
/\d{3}/.test("abc123"); // → true (drei Ziffern)
```

---

4. Übungsaufgaben

1. Serialisiere ein Objekt mit verschachteltem Array nach JSON.
2. Parse einen JSON-String und greife auf ein verschachteltes Feld zu.
3. Vergleiche das Verhalten von `Math.round`, `floor`, `ceil` mit Werten >0 und <0.
4. Schreibe eine Funktion `rollDice()` mit Zufallswert 1–6.
5. Verwende `Math.max(...array)` – erkläre, warum `...` nötig ist.
6. Erstelle eine RegExp, die einfache E-Mail-Adressen prüft (z. B. `/\S+@\S+\.\S+/`).
7. Ersetze in einem Text mit `replace()` z. B. Namen durch Platzhalter.

---

5. Micro-Projekt: Validierungs- und Speicher-Utility für ein Formular

Szenario: Ein Kontaktformular soll validiert, formatiert und gespeichert werden.

Anforderungen:

- Felder: Name, E-Mail, Nachricht
- E-Mail muss einem einfachen RegExp-Muster entsprechen
- Erfolgreiche Eingaben sollen als JSON serialisiert und gespeichert (z. B. in `localStorage` oder einer Datei) ausgegeben werden

Beispielimplementierung:

```
const isEmail = email => /\S+@\S+\.\S+/.test(email);

function saveFormData(name, email, message) {
  if (!isEmail(email)) {
    console.error("Ungültige E-Mail-Adresse");
    return;
  }
  const data = { name, email, message };
  const json = JSON.stringify(data);
  console.log("Gespeichert:", json);
  // z. B. localStorage.setItem("formData", json);
}
```

Ziel: Verbindung von Eingabevalidierung, regulären Ausdrücken, Datenstrukturierung und Serialisierung in einem realistischen Szenario.