JavaScript

#ITP

[[JavaScript#[redacted] - (Destructuring / Klassen / Array-Funktionen)[redacted] - Destructuring / Klassen / Array-Funktionen]] [[JavaScript#[redacted] - (Block-Scope, let und const / Template Strings) [redacted] - Block Scope, let-const, Template Strings]] [[JavaScript#[redacted] - (Default- & Rest-Parameter / Spread Operator für Arrays)[redacted] - Default- & Rest-Parameter / Spread Operator für Arrays]] [[JavaScript#[redacted] - (Import-Export / Arrow Functions)[redacted] - Import-Export / Arrow Functions]]

[redacted] - (Block-Scope, let und const / Template Strings)

Block Scope Man kann auf eine in einem if, switch, for- oder while loops deklarierte Variable nicht außerhalb von ihnen zugreifen. Bsp.:

```
{
  let b   10;
  console.log(b);
}
console.log(b);
```

Die Erste Ausgabe funktioniert, die nächste führt zu einem Fehler.

Function Scope Man kann auf eine in einer Funktion deklarierte Variable nicht außerhalb von der Funktion zugreifen.

`let` verwendet block scope var verwendet function scope

Let und Const let ist eine veränderbare Variable Bsp.:

```
let a = 10;
a = 20;
console.log(a);
```

Funktioniert einwandfrei 😎

const ist eine nicht veränderbare Variable Bsp.:

```
// Hier tritt absichtlich ein Fehler auf
const a   10;
a   20;
console.log(a);
```

Template Strings

Multiline String Jeder Zeilenumbruch wird mitgezählt

```
const address = `Eduard Müller
Anton Ehrenfried-Straße 10
2020 Hollabrunn`;

console.log(address);
```

Ausgabe:

```
Eduard Müller
Anton Ehrenfried Straße 10
2020 Hollabrunn
```

String Interpolation Variablen in einen String eingeben Bsp1.:

```
const firstName   "Eduard";
const lastName   "Müller";
const street   "Anton Ehrenfried Straße";
const houseNr   10;
const zipCode   2020;
const city   "Hollabrunn";

const address    ${firstName} ${lastName}
${street} ${houseNr}
${zipCode} ${city} ;

console.log(address);
```

Ausgabe:

```
Eduard Müller
Anton Ehrenfried-Straße 10
2020 Hollabrunn
```

Bsp2.:

```
const add = `1 + 1 = ${1 + 1}`;
console.log(add);
```

Ausgabe: ``1 + 1 = 2

[redacted] - (Destructuring / Klassen / Array-Funktionen)

Destructuring

Destructuring in JavaScript ermöglicht es, Werte aus Objekten und Arrays zu extrahieren und in separaten Variablen zu speichern.

Destructuring von Objekten:

```
const person = { name: 'Alice', age: 30 };

// Destructuring von Objektattributen
const { name:firstName, age } = person;

console.log(firstName); // Gibt 'Alice' aus
        console.log(age);  // Gibt 30 aus
```

Destructuring von Arrays:

```
const colors = ['Rot', 'Grün', 'Blau'];

// Destructuring von Array-Elementen
const [firstColor, secondColor, thirdColor] = colors;

console.log(firstColor);  // Gibt 'Rot' aus
console.log(secondColor); // Gibt 'Grün' aus
console.log(thirdColor);  // Gibt 'Blau' aus
```

Durch Destructuring kannst du einfach auf Werte in Objekten und Arrays zugreifen und sie in benannten Variablen speichern. Dies erleichtert das Arbeiten mit komplexen Datenstrukturen.

Klassen

JavaScript-Klassen werden verwendet, um Objekte mit Methoden und Eigenschaften zu erstellen. Hier ist ein Beispiel, das eine Klasse mit einem Konstruktor, einer schreibgeschützten Eigenschaft und einem Getter zeigt:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  // Getter für das Alter
  get age() {
    return this.age;
  }

  // Eine schreibgeschützte Eigenschaft
  get isAdult() {
    return this.age >= 18;
  }
}

const alice = new Person('Alice', 30);
console.log(alice.name); // undefined, da _name nicht direkt zugänglich ist
console.log(alice.age); // 30, über den Getter
console.log(alice.isAdult); // true, über den Getter
```

In diesem Beispiel:

- Die `Person`-Klasse hat einen Konstruktor, der beim Erstellen eines `Person`-Objekts aufgerufen wird und `name` und `age` initialisiert.
- Der Getter `age` ermöglicht den Zugriff auf das Alter über `alice.age`.
- Der Getter `isAdult` gibt zurück, ob die Person volljährig ist oder nicht.
- `_name` und `_age` sind als Konvention mit einem Unterstrich versehen, um sie als private Felder zu kennzeichnen. Sie sollten normalerweise nicht direkt zugänglich sein. Der Zugriff erfolgt über die Getter-Methode.

Array-Funktionen

JavaScript bietet nützliche Array-Funktionen wie `forEach()`, `map()`, `filter()`, `sort()`, `find()` um die Verarbeitung von Arrays zu erleichtern. Hier ist eine kurze Erklärung und Beispiele für jede Funktion:

forEach()

Diese Funktion führt eine bereitgestellte Funktion einmal für jedes Element im Array aus.

```
const numbers   [1, 2,  , 4];
numbers.forEach((number, i)    {
  console.log(number);
});
// Gibt 1, 2,  , 4 in der Konsole aus.

// Syntax:
array.forEach(function(currentValue, index, arr), thisValue)
```

map()

Die `map()`-Funktion erstellt ein neues Array, indem sie eine bereitgestellte Funktion auf jedes Element im Ausgangsarray anwendet.

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
// Gibt [2, 4, 6, 8] aus.

// Syntax:
array.map(function(currentValue, index, arr), thisValue)
```

filter()

Mit `filter()` kannst du ein neues Array erstellen, das nur die Elemente enthält, die einer bestimmten Bedingung entsprechen.

```
const numbers = [1, 2, 3, 4];
const evenNumbers = numbers.filter((number) => number % 2 === 0);
console.log(evenNumbers);
// Gibt [2, 4] aus.

// Syntax:
array.filter(function(currentValue, index, arr), thisValue)
```

sort()

Mit `sort()` kannst du die Elemente im Array sortieren. Beachte, dass es das ursprüngliche Array ändert.

```
const fruits = ['Banane', 'Apfel', 'Dattel', 'Kirsche'];
fruits.sort();
console.log(fruits);
// Gibt ["Apfel", "Banane", "Dattel", "Kirsche"] aus.

// Syntax:
array.sort(compareFunction)
```

find()

`find()` gibt das erste Element im Array zurück, das einer bestimmten Bedingung entspricht.

```
const numbers = [1, 2, 3, 4];
const found = numbers.find((number) => number > 2);
console.log(found);
// Gibt 3 aus (das erste Element, das größer als 2 ist).

// Syntax:
array.find(function(currentValue, index, arr),thisValue)
```

[redacted] - (Import-Export / Arrow Functions)

import/export

In JavaScript werden "import" und "export" verwendet, um Code zwischen Dateien aufzuteilen und zu teilen. Mit "export" markierst du, was du in einer Datei freigeben möchtest, und mit "import" kannst du auf diese freigegebenen Teile in einer anderen Datei zugreifen. "export" sagt, was exportiert wird, und "import" sagt, was importiert wird.

Beispiel:

mathFunctions.js:

```
export function add(a, b) {
  return a + b;
}

export const pi = 3.14159265;
```

main.js:

```
import { add, pi } from './mathFunctions';

console.log(add(2, 3)); // Ausgabe: 5

console.log(pi); // Ausgabe: 3.14159265
```

Arrow Funktionen

Arrow-Funktionen in JavaScript sind eine kompakte Möglichkeit, Funktionen zu definieren. Sie haben eine kürzere Syntax und erben das „this“ des umgebenden Kontexts. Hier ist ein einfaches Beispiel:

```
// Herkömmliche Funktion zur Addition
function add(a, b) {
  return a + b;
}
console.log(add(1, 2)); // Ausgabe: "3"

// Arrow-Funktion zur Addition
// zwei Parameter (runde Klammer nötig)
const addArrow = (a, b) => a + b;
console.log(addArrow(2,3)) // Ausgabe: "5"



// ein Parameter (keine runde Klammer nötig)
const double = zahl => zahl * 2;
console.log(double(5)); // Ausgabe: "10"



// null Parameter (runde Klammer nötig)
const helloWorld = () => "Hello World";
console.log(helloWorld()); // Ausgabe: "Hello World"



// Arrow-Funktion mit größerem body
const multiplyAndAdd = (zahl1, zahl2) => {
  const product = zahl1 * zahl2;
  const sum = zahl1 + zahl2;
  return `${zahl1} * ${zahl2} = ${product} \n${zahl1} + ${zahl2} = ${sum}.`;
};
console.log(multiplyAndAdd(5, 3)); // Ausgabe: "5 * 3 = 15" \n "5 + 3 = 8"
```

[redacted] - (Default- & Rest-Parameter / Spread Operator für Arrays)

Default-Parameter in ES6

Default-Parameter werden verwendet, um Parametern in Funktionen Standartwerte zu geben. Diese werden in der Funktion verwendet, falls kein anderer Wert beim Methodenaufruf angegeben wird.

```
function sayHello(name = 'User') {
        console.log(`Hello, ${name}.`)
}

sayHello();
sayHello('Austin');
```

Der erste Aufruf gibt "Hello, User." aus, der zweite "Hello, Austin". In diesem Fall ist User der Default-Parameter.

Rest-Parameter

Rest-Parameter werden verwendet, um eine nicht-sichere Anzahl an Argumente in einem Array zu speichern, wodurch man diese verarbeiten kann, obwohl man die genaue Anzahl nicht von Anfang an weiß.

```
function add(...numbers) {
        let sum = 0;
        for (let number of numbers) {
                sum += number;
        }
        return sum;
}

console.log(add(3,6,9));
```

Die Ausgabe in diesem Fall ist 18, man kann hier durch die for-each Schleife jedoch auch mehr oder weniger Parameter angeben.

Spread Operator für Arrays

Mit Spread Operators "..." kann man Arrays:

1. weitere Elemente hinzufügen

```
const startArray = [1, 2, 3];
const newArray = [...startArray, 4];

console.log(newArray);
```

2. kopieren

```
const startArray = [1, 2, 3];
const copiedArray = [...startArray];

console.log(copiedArray);
```

3. zusammenfügen

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const combined = [...array1, ...array2];

console.log(combined);
```

4. Als Parameter in einer Funktion verwenden

```
function add(a, b, c) {
  return a + b + c;
}

const numbers = [1, 2, 3];
const sum = add(...numbers);

console.log(sum);
```

Sie sind sehr nützlich um javascript übersichtlicher und einfacher zu schreiben zu machen.

React

![[React.svg]]