4.1 - JSA: Fortgeschrittene Funktionen und Decorators

Inhaltsübersicht Abschnitt 4.1

- **Block A: Erweiterte Parameterbehandlung und Closures**
  - Erweiterte Parameterbehandlung (Default-Werte, Rest-Parameter, Spread-Operator, Simulierte benannte Parameter)
  - Closures (Konzept, Lexikalisches Scoping, Praktische Anwendungsbeispiele)
  - Übungsaufgaben zu Block A
  - Micro-Projekt: Utility-Funktionen - Phase 1
- **Block B: IIFE und `this`-Kontextsteuerung**
  - IIFE (Immediately Invoked Function Expression)
  - Das `this`-Keyword in Funktionen (Überblick, Strict-Mode, Arrow Functions)
  - Weiterleitung von Aufrufen und `this`-Bindung (`call`, `apply`, `bind`)
  - Übungsaufgaben zu Block B
- **Block C: First-Class Functions und Decorators**
  - First-Class Functions (Konzept und Bedeutung)
  - Decorating Functions (Higher-Order Functions, Wrappers, Anwendungsbeispiele wie Caching)
  - Übungsaufgaben zu Block C
  - Micro-Projekt: Utility-Funktionen - Phase 2

Block A: Erweiterte Parameterbehandlung und Closures

A.1 Erweiterte Parameterbehandlung

ECMAScript 2015 (ES6) hat einige sehr nützliche Verbesserungen für den Umgang mit Funktionsparametern eingeführt. Diese helfen, Funktionen robuster und flexibler zu gestalten.

A.1.1 Default-Parameterwerte

Ermöglichen die Definition von Standardwerten für Parameter, falls beim Aufruf kein Argument übergeben wird oder `undefined` übergeben wird.

Syntax:

```
function funktionsName(parameter1 = standardwert1, parameter2 = standardwert2) {
  // Funktionskörper
}
```

Beispiel: (Vergleich Edube 4.1.1)

```
function gruss(name = "Gast") {
  console.log(`Hallo, ${name}!`);
}
gruss("Alice"); // Ausgabe: Hallo, Alice!
gruss();        // Ausgabe: Hallo, Gast!
```

Referenz: MDN Web Docs: Default parameters

A.1.2 Der Rest-Parameter (`...`)

Erlaubt das Sammeln einer unbestimmten Anzahl von Argumenten in einem Array.

Syntax und Regeln:

```
function funktionsName(param1, ...restlicheArgumente) { /* ... */ }
```

- Nur ein Rest-Parameter pro Funktion.
- Muss der letzte Parameter sein. (Edube 4.1.2)

Beispiel: (Vergleich Edube 4.1.2)

```
function summiereAlle(...zahlen) {
  return zahlen.reduce((summe, zahl) => summe + zahl, 0);
}
console.log(summiereAlle(1, 2, 3)); // Ausgabe: 6
console.log(summiereAlle(10, 20));  // Ausgabe: 30
```

Referenz: MDN Web Docs: Rest parameters

A.1.3 Der Spread-Operator (`...`) bei Funktionsaufrufen

Verteilt Elemente eines Arrays (oder eines anderen iterierbaren Objekts) als einzelne Argumente für einen Funktionsaufruf. (Edube 4.1.3)

Syntax:

```
const meineArgumente = [arg1, arg2, arg3];
funktionsName(...meineArgumente);
```

Beispiel: (Vergleich Edube 4.1.3)

```
function addiere(x, y, z) {
  return x + y + z;
}
const zahlenArray = [10, 20, 30];
console.log(addiere(...zahlenArray)); // Ausgabe: 60
```

Referenz: MDN Web Docs: Spread syntax (...)

A.1.4 Simulierung von benannten Parametern durch Objekt-Destrukturierung

Ermöglicht die Übergabe eines Objekts als Argument und die Destrukturierung dieses Objekts in der Parameterliste, um die Reihenfolge der Argumente irrelevant zu machen und die Lesbarkeit zu erhöhen. (Edube 4.1.4)

Syntax:

```
function funktionsName({ param1, param2, param3 = defaultValue }) { /* ... */ }
```

Beispiel: (Vergleich Edube 4.1.4)

```
function erstellePunkt({ x, y, farbe = "rot", beschriftung }) {
  console.log(`Punkt: (${x}, ${y}), Farbe: ${farbe}${beschriftung ? ', Label: ' + beschriftung :
''}`);
}
erstellePunkt({ y: 200, x: 100, beschriftung: "Mittelpunkt" });
// Ausgabe: Punkt: (100, 200), Farbe: rot, Label: Mittelpunkt
```

Referenz: MDN Web Docs: Destructuring assignment - Function parameter destructuring

A.2 Closures

Eine Closure ermöglicht einer Funktion den Zugriff auf Variablen aus ihrem äußeren (umschließenden) Gültigkeitsbereich, selbst nachdem die äußere Funktion ihre Ausführung beendet hat. (Edube 4.1.5)

A.2.1 Konzept und Lexikalisches Scoping

JavaScript verwendet lexikalisches Scoping. Der Gültigkeitsbereich wird durch die Position im Quellcode bestimmt. Innere Funktionen haben Zugriff auf Variablen äußerer Funktionen. Die Closure "erinnert" sich an diesen Scope.

Beispiel: (Vergleich Edube 4.1.5, `getTick` function)

```
function initialisiereZaehler() {
  let zaehler = 0;
  return function() { // Diese innere Funktion ist eine Closure
    zaehler++;
    return zaehler;
  };
}
const meinZaehler = initialisiereZaehler();
console.log(meinZaehler()); // Ausgabe: 1
console.log(meinZaehler()); // Ausgabe: 2
```

A.2.2 Praktische Anwendungsbeispiele

- **Datenkapselung / Private Variablen:** (Vergleich Edube 4.1.5, `getPoint` function)

  ```
  function erstellePerson(name) {
    let _alter = 0; // "Private" Variable
    function _altern() { _alter++; } // "Private" Methode
    return {
      getName: () => name,
      getAlter: () => { _altern(); return _alter; }
    };
  }
  const personA = erstellePerson("Max");
  console.log(personA.getAlter()); // Ausgabe: 1
  ```

- **Funktionsfabriken:**

  ```
  function erstelleMultiplikator(faktor) {
    return (zahl) => zahl * faktor;
  }
  const doppelt = erstelleMultiplikator(2);
  console.log(doppelt(5)); // Ausgabe: 10
  ```

- **Event Handler und Callbacks (z.B. `setTimeout`)**

Referenz: MDN Web Docs: Closures

Übungsaufgaben zu Block A (Erweiterte Parameterbehandlung & Closures)

(Die bereits erstellten Übungsaufgaben 1-7 von der vorherigen Antwort würden hier stehen. Ich wiederhole sie hier zur Vollständigkeit.)

Aufgabe A.1: Default-Werte gestalten (Anwendung) Erstelle eine Funktion `erstelleRechteckBeschreibung`, die ein Objekt mit den Eigenschaften `breite`, `hoehe` und `einheit` entgegennimmt. `breite` und `hoehe` sind Pflicht, `einheit` hat den Default-Wert `"px"`. Die Funktion soll einen String zurückgeben, z.B.: `"Rechteck: 100px breit und 50px hoch."`. Nutze Objekt-Destrukturierung.

Aufgabe A.2: Argumente summieren (Rest-Parameter) Schreibe eine Funktion `multipliziereUndSummiere`, die als ersten Parameter einen `multiplikator` (Zahl) erwartet und danach eine beliebige Anzahl weiterer Zahlen (Rest-Parameter). Die Funktion soll jede der weiteren Zahlen mit dem `multiplikator` multiplizieren und dann die Summe dieser multiplizierten Zahlen zurückgeben.

Aufgabe A.3: Array-Elemente nutzen (Spread-Operator) Gegeben ist `function erstelleCssTransform(x, y, rotation, scale) { /* ... */ }`. Ein Array `einstellungen` enthält `[10, 20, 45, 1.5]`. Rufe die Funktion mit dem Spread-Operator auf.

Aufgabe A.4: Konfigurations-Objekt (Simulierte benannte Parameter) Entwickle eine Funktion `logNachricht`, die ein Konfigurationsobjekt akzeptiert (`text` (Pflicht), `level` (Default: `"INFO"`), `timestamp` (Boolean, Default: `false`), `benutzer` (optional)). Formatiere die Ausgabe entsprechend.

Aufgabe A.5: Closure für privaten Zähler (Analyse & Implementierung) Implementiere `function geheimerZaehler() { /* ... */ }`, die ein Objekt mit Methoden `inkrementiere()` und `wert()` zurückgibt, um einen internen, privaten Zähler zu managen.

Aufgabe A.6: Fehlerfindung - Rest & Spread Korrigiere den folgenden Code:

```
// function zeigeProdukte(hauptprodukt, ...weitereProdukte, kategorie = "Allgemein") { /* ... */
} // Fehler 1
// const produkte = ["Laptop", "Maus", "Tastatur"];
// zeigeProdukte(produkte); // Fehler 2
```

Aufgabe A.7: Funktionsfabrik mit Closure (Eigene Implementierung) Erstelle `erstelleBegruessungsfunktion(sprache)`, die eine Funktion zurückgibt, die einen Namen nimmt und sprachspezifisch grüßt (z.B. "de" -> "Hallo Max", "en" -> "Hello Max").

Micro-Projekt: Utility-Funktionen - Phase 1 (Erweiterte Parameterbehandlung)

Entwickle eine Funktion `formatText` in einer Datei `textUtils.js`. Parameter (Objekt-Destrukturierung):

- `text` (String, Pflicht).
- `uppercase` (Boolean, Default: `false`).
- `prefix` (String, Default: `""`).
- `suffix` (String, Default: `""`). Die Funktion soll den formatierten Text zurückgeben.

Beispiel:

```
// function formatText({ text, uppercase = false, prefix = "", suffix = "" }) {
//   let result = text;
//   if (uppercase) {
//     result = result.toUpperCase();
//   }
//   return `${prefix}${result}${suffix}`;
// }
// console.log(formatText({ text: "Hallo Welt", uppercase: true, prefix: "INFO: " }));
// Erwartete Ausgabe: INFO: HALLO WELT
```

Block B: IIFE und `this`-Kontextsteuerung

B.1 IIFE (Immediately Invoked Function Expression)

Eine IIFE ist ein Design-Pattern in JavaScript, bei dem eine Funktion definiert und sofort danach ausgeführt wird. Dies ist nützlich, um einen eigenen Scope zu erzeugen und so den globalen Namespace nicht zu "verschmutzen". (Edube 4.1.6)

Syntax:

```
(function() {
  // Code hier ist in einem eigenen Scope
  let lokaleVariable = "Ich bin lokal";
  console.log(lokaleVariable);
```

```
})(); // Die äußeren Klammern und die aufrufenden Klammern am Ende sind entscheidend

// Oder mit Arrow Function:
(() => {
  let andereLokaleVariable = "Auch lokal";
  console.log(andereLokaleVariable);
})();
```

Vorteile:

- **Vermeidung von globalen Variablen:** Variablen, die innerhalb der IIFE deklariert werden, sind nicht außerhalb sichtbar. Dies verhindert Namenskonflikte, besonders in größeren Anwendungen oder beim Einbinden von Drittanbieter-Bibliotheken.
- **Modularität:** Kann helfen, Code-Blöcke logisch zu isolieren.
- **Parameterübergabe:** Einer IIFE können auch Argumente übergeben werden:

  ```
  (function(global, doc) {
    // global ist hier window, doc ist document
    // console.log("Fenstertitel:", global.document.title);
  })(window, document);
  ```

Beispiel: (Vergleich Edube 4.1.6)

```
(function() {
  let a = 10;
  let b = 20;
  let summe = (x, y) => x + y;
  let ergebnis = summe(a, b);
  console.log(`Das Ergebnis (intern): ${ergebnis}`); // Ausgabe: Das Ergebnis (intern): 30
})();

// console.log(ergebnis); // Fehler: ergebnis is not defined (außerhalb des IIFE-Scopes)
```

Referenz: MDN Web Docs: IIFE

B.2 Das `this`-Keyword in Funktionen

Das `this`-Keyword in JavaScript ist ein häufiger Quell für Verwirrung, da sein Wert davon abhängt, *wie* eine Funktion aufgerufen wird (Aufrufkontext). (Edube 4.1.7)

Grundregeln für `this`:

1. **Globaler Kontext:** Außerhalb jeder Funktion im globalen Scope zeigt `this` auf das globale Objekt (`window` im Browser, `global` in Node.js).

   ```
   console.log(this === window); // true (im Browser)
   ```

2. **Einfacher Funktionsaufruf (Non-Strict Mode):** Wenn eine Funktion "einfach so" aufgerufen wird (nicht als Methode eines Objekts), zeigt `this` ebenfalls auf das globale Objekt.

   ```
   function zeigeThis() {
     console.log(this);
   }
   zeigeThis(); // Window (im Browser, Non-Strict Mode)
   ```

3. **Einfacher Funktionsaufruf (Strict Mode):** Im Strict Mode (`'use strict';`) ist `this` bei einfachen Funktionsaufrufen `undefined`. Dies hilft, unbeabsichtigtes Verändern des globalen Objekts zu vermeiden.

```
'use strict';
function zeigeThisStrict() {
  console.log(this);
}
zeigeThisStrict(); // undefined
```

4. **Als Objektmethode:** Wenn eine Funktion als Methode eines Objekts aufgerufen wird, zeigt `this` auf das Objekt, zu dem die Methode gehört.

```
const meinObjekt = {
  wert: 42,
  getWert: function() {
    console.log(this.wert); // this zeigt auf meinObjekt
    console.log(this === meinObjekt); // true
  }
};
meinObjekt.getWert(); // Ausgabe: 42, true
```

5. **Als Konstruktorfunktion (mit `new`):** Wenn eine Funktion mit dem `new`-Keyword aufgerufen wird, um ein neues Objekt zu erstellen, wird `this` innerhalb der Funktion an das neu erstellte Objekt gebunden.

```
function Auto(marke) {
  this.marke = marke; // this zeigt auf das neue Auto-Objekt
}
const meinAuto = new Auto("Tesla");
console.log(meinAuto.marke); // Ausgabe: Tesla
```

6. **Arrow Functions (`=>`):** Arrow Functions haben **kein eigenes `this`-Binding**. Sie erben den `this`-Wert von ihrem umschließenden (lexikalischen) Kontext. Das macht sie oft einfacher handhabbar, besonders in Callbacks.

```
const meinObjektArrow = {
  wert: 100,
  ladeDaten: function() {
    console.log("this in ladeDaten:", this); // meinObjektArrow
    setTimeout(() => {
      // Arrow Function erbt 'this' von ladeDaten
      console.log("this in setTimeout (Arrow):", this); // meinObjektArrow
      console.log("Wert:", this.wert); // 100
    }, 100);

    setTimeout(function() {
      // Reguläre Funktion hat ihr eigenes 'this' (Window oder undefined)
      console.log("this in setTimeout (Regular):", this); // Window (non-strict) oder
undefined (strict)
      // console.log("Wert:", this.wert); // Fehler oder undefined
    }, 200);
  }
};
meinObjektArrow.ladeDaten();
```

7. **Explizite Bindung (`call`, `apply`, `bind`):** Diese Methoden erlauben es, den Wert von `this` explizit zu setzen (siehe nächster Abschnitt).

Referenz: MDN Web Docs: this

B.3 Weiterleitung von Aufrufen und `this`-Bindung (`call`, `apply`, `bind`)

JavaScript bietet drei Methoden, die auf jeder Funktion verfügbar sind, um den Kontext (`this`-Wert) einer Funktion beim Aufruf explizit zu steuern und Argumente zu übergeben: `call()`, `apply()`, und `bind()`. (Edube 4.1.7)

B.3.1 `function.call(thisArg, arg1, arg2, ...)`

- Ruft die Funktion sofort auf.
- Der erste Parameter `thisArg` setzt den Wert von `this` innerhalb der Funktion.
- Nachfolgende Argumente (`arg1, arg2, ...`) werden einzeln als Argumente an die Funktion übergeben.

```
function gruss(grussformel, zusatz) {
  console.log(`${grussformel}, mein Name ist ${this.name}. ${zusatz}`);
}

const person1 = { name: "Alice" };
const person2 = { name: "Bob" };

gruss.call(person1, "Hallo", "Wie geht es dir?"); // Ausgabe: Hallo, mein Name ist Alice. Wie
geht es dir?
gruss.call(person2, "Guten Tag", "Schöner Tag heute."); // Ausgabe: Guten Tag, mein Name ist
Bob. Schöner Tag heute.

const zahlen = [1, 2, 3];
// Math.max.call(null, 1, 2, 3); // Alternative, aber Spread ist hier schöner:
Math.max(...zahlen)
```

B.3.2 `function.apply(thisArg, [argsArray])`

- Ruft die Funktion sofort auf.
- Der erste Parameter `thisArg` setzt den Wert von `this`.
- Der zweite Parameter `[argsArray]` ist ein Array (oder ein Array-ähnliches Objekt), dessen Elemente als einzelne Argumente an die Funktion übergeben werden.

```
function berechneSumme(a, b, c) {
  console.log(`${this.kontextInfo}: ${a + b + c}`);
}

const kontext = { kontextInfo: "Summe" };
const werte = [10, 20, 5];

berechneSumme.apply(kontext, werte); // Ausgabe: Summe: 35

// Ein häufiger Anwendungsfall (vor ES6 Spread Operator) war das Anwenden von Array-Methoden auf
Array-ähnliche Objekte
// oder das Übergeben von Array-Elementen an Funktionen wie Math.max
console.log(Math.max.apply(null, [1, 5, 2])); // Ausgabe: 5
// Mit ES6 Spread: console.log(Math.max(...[1, 5, 2]));
```

Der Hauptunterschied zwischen `call` und `apply` liegt darin, wie die Argumente übergeben werden (einzeln vs. als Array).

B.3.3 `function.bind(thisArg, arg1, arg2, ...)`

- Erstellt eine **neue Funktion** (eine gebundene Funktion), bei der der `this`-Wert fest auf `thisArg` gesetzt ist.
- Die ursprüngliche Funktion wird **nicht sofort aufgerufen**.
- Argumente (`arg1, arg2, ...`), die an `bind` übergeben werden (nach `thisArg`), werden als "voreingestellte" oder "partiell angewendete" Argumente für die neue Funktion verwendet. Später übergebene Argumente werden angehängt.

```
const modul = {
  x: 42,
  getX: function() {
```

```
    return this.x;
  }
};

const ungebundenesGetX = modul.getX;
// console.log(ungebundenesGetX()); // Fehler im Strict Mode (this ist undefined) oder Window.x
im Non-Strict Mode

const gebundenesGetX = modul.getX.bind(modul);
console.log(gebundenesGetX()); // Ausgabe: 42 (this ist korrekt an modul gebunden)

// Beispiel für partielle Anwendung mit bind:
function multipliziere(a, b) {
  return a * b;
}
const verdopple = multipliziere.bind(null, 2); // this ist hier irrelevant, 2 wird als erstes
Argument 'a' festgesetzt
console.log(verdopple(5));  // Ausgabe: 10 (5 wird als zweites Argument 'b' übergeben)
console.log(verdopple(10)); // Ausgabe: 20

const zeigeKoordinaten = gruss.bind(person1, "Koordinaten");
zeigeKoordinaten("Hier sind sie."); // Ausgabe: Koordinaten, mein Name ist Alice. Hier sind sie.
```

`bind` ist besonders nützlich, um den `this`-Kontext für Callback-Funktionen oder Event-Handler sicherzustellen, die später aufgerufen werden.

Wichtig für Arrow Functions: `call`, `apply` und `bind` haben **keine Auswirkung auf den `this`-Wert von Arrow Functions**, da diese ihren `this`-Wert lexikalisch erben und dieser nicht geändert werden kann. Man kann sie aber nutzen, um Argumente für Arrow Functions vorab zu binden.

Referenzen:

- MDN Web Docs: Function.prototype.call()
- MDN Web Docs: Function.prototype.apply()
- MDN Web Docs: Function.prototype.bind()

Übungsaufgaben zu Block B (IIFE & `this`-Kontextsteuerung)

Aufgabe B.1: IIFE zur Kapselung Erstelle eine IIFE, die eine Variable `geheimeNachricht` enthält und eine Funktion `zeigeNachricht`, die diese Nachricht auf der Konsole ausgibt. Rufe `zeigeNachricht` innerhalb der IIFE auf. Stelle sicher, dass `geheimeNachricht` von außerhalb der IIFE nicht zugänglich ist.

Aufgabe B.2: `this` im Objektkontext Erstelle ein Objekt `rechner` mit den Eigenschaften `a` und `b` (Zahlen) und einer Methode `summe()`, die die Summe von `this.a` und `this.b` zurückgibt. Rufe die Methode auf und gib das Ergebnis aus. Füge eine weitere Methode `zeigeWerte()` hinzu, die eine interne (reguläre) Funktion verwendet, die versucht, `this.a` und `this.b` auszugeben. Was passiert? Wie könntest du das Problem mit `bind` oder einer Arrow Function lösen?

Aufgabe B.3: `call` zur Methoden"ausleihe" Erstelle zwei Objekte, `personA` mit einer Eigenschaft `name` und `personB` ebenfalls mit einer Eigenschaft `name`. Erstelle eine separate Funktion `stellSichVor(stadt)`, die `console.log(\Hallo, ich bin ${this.name} aus ${stadt}.`)ausführt. Nutzecall, um stellSichVorfürpersonAundpersonB` mit unterschiedlichen Städten aufzurufen.

Aufgabe B.4: `apply` für variable Argumente Gegeben sei ein Array von Zahlen, z.B. `const zahlen = [5, 2, 8, 1, 9];`. Nutze `Math.max.apply(null, zahlen)` um die größte Zahl im Array zu finden und auszugeben. Vergleiche dies mit der Spread-Operator-Syntax.

Aufgabe B.5: `bind` für Event-Handler (Simulation) Simuliere einen Event-Handler. Erstelle ein Objekt `button` mit einer Eigenschaft `buttonText = "Klick mich"` und einer Methode `handleClick()`, die `console.log(this.buttonText + " wurde geklickt!")` ausführt. Erstelle eine Variable `onClickHandler = button.handleClick.bind(button);`. Rufe `onClickHandler()` auf. Was passiert, wenn du `bind` nicht verwenden würdest und `onClickHandler = button.handleClick; onClickHandler();` schreibst (simuliere, dass `onClickHandler` in einem anderen Kontext aufgerufen wird)?

Aufgabe B.6: `this` mit Arrow Functions in Callbacks Erstelle ein Objekt `timer` mit einer Eigenschaft `sekunden = 0` und einer Methode `start()`. Innerhalb von `start()` soll `setInterval` verwendet werden, um die `sekunden`-Eigenschaft jede Sekunde um 1 zu

erhöhen und `console.log(this.sekunden)` auszugeben. Implementiere dies einmal mit einer regulären Funktion als Callback für `setInterval` und einmal mit einer Arrow Function. Erkläre die Unterschiede im Verhalten von `this`. (Hinweis: `setInterval` stoppen nach einigen Sekunden mit `clearInterval`).

Block C: First-Class Functions und Decorators

C.1 First-Class Functions

In JavaScript sind Funktionen "First-Class Citizens" (Bürger erster Klasse). Das bedeutet, Funktionen können wie jeder andere Wert behandelt werden:

- Sie können Variablen zugewiesen werden.
- Sie können als Argumente an andere Funktionen übergeben werden (Callbacks, Higher-Order Functions).
- Sie können von anderen Funktionen zurückgegeben werden (Closures, Funktionsfabriken).
- Sie können in Datenstrukturen (Arrays, Objekte) gespeichert werden. (Edube 4.1.8)

Beispiele:

```
// Zuweisung an Variable (Function Expression)
const grussFunktion = function(name) {
  return `Hallo, ${name}!`;
};
console.log(grussFunktion("Welt"));

// Übergabe als Argument
function fuehreAus(fn, wert) {
  return fn(wert);
}
console.log(fuehreAus(grussFunktion, "Alice")); // Ausgabe: Hallo, Alice!

// Rückgabe aus Funktion (Funktionsfabrik)
function erstelleAddierer(addWert) {
  return function(zahl) { // Diese innere Funktion ist eine Closure
    return zahl + addWert;
  };
}
const addiere5 = erstelleAddierer(5);
console.log(addiere5(10)); // Ausgabe: 15
```

Dieses Konzept ist die Grundlage für viele fortgeschrittene Patterns in JavaScript, einschließlich funktionaler Programmierung und Decorators.

Referenz: MDN Web Docs: First-class Function

C.2 Decorating Functions (Wrappers, Higher-Order Functions)

Ein Decorator ist ein spezielles Pattern, bei dem eine Funktion (der Decorator) eine andere Funktion als Argument nimmt und eine *neue* Funktion zurückgibt. Diese neue Funktion "umhüllt" (wraps) die ursprüngliche Funktion und fügt ihr zusätzliches Verhalten hinzu, ohne die ursprüngliche Funktion direkt zu verändern. Decorators sind selbst Higher-Order Functions. (Edube 4.1.9)

Grundkonzept:

```
function originalFunktion(a, b) {
  console.log(`Originalfunktion mit ${a} und ${b}`);
  return a + b;
}

// Decorator-Funktion
function loggingDecorator(fn) {
  return function(...args) { // Die zurückgegebene Funktion ist der eigentliche Decorator-
Wrapper
```

```
    console.log(`Aufruf von ${fn.name || 'anonymer Funktion'} mit Argumenten: ${args.join(',
')}`);
    const startZeit = performance.now(); // Performance-Messung (Beispiel für Zusatzverhalten)

    const ergebnis = fn.apply(this, args); // Ruft die originale Funktion auf, Kontext und
Argumente weiterleiten

    const endZeit = performance.now();
    console.log(`Ergebnis: ${ergebnis}`);
    console.log(`${fn.name || 'Anonyme Funktion'} ausgeführt in ${(endZeit -
startZeit).toFixed(4)} ms.`);
    return ergebnis;
  };
}

const dekorierteFunktion = loggingDecorator(originalFunktion);
dekorierteFunktion(5, 3);
// Mögliche Ausgabe:
// Aufruf von originalFunktion mit Argumenten: 5, 3
// Originalfunktion mit 5 und 3
// Ergebnis: 8
// originalFunktion ausgeführt in X.XXXX ms.

originalFunktion(1,2); // Bleibt unverändert
// Ausgabe:
// Originalfunktion mit 1 und 2
```

Anwendungsbeispiele für Decorators:

- **Logging:** Protokollieren von Funktionsaufrufen, Argumenten, Ergebnissen oder Ausführungszeiten.
- **Caching / Memoization:** Speichern der Ergebnisse von teuren Funktionsaufrufen, um bei gleichen Argumenten das Ergebnis direkt aus dem Cache zurückzugeben, anstatt die Funktion erneut auszuführen. (Vergleich Edube 4.1.9, `factorial` caching)

  ```
  function teureBerechnung(n) {
    console.log(`Berechne für ${n}...`);
    // Simulierte langsame Berechnung
    for (let i = 0; i < 1e8; i++) {}
    return n * 2;
  }

  function memoizeDecorator(fn) {
    const cache = new Map();
    return function(arg) {
      if (cache.has(arg)) {
        console.log(`Aus Cache für ${arg} geholt.`);
        return cache.get(arg);
      }
      const result = fn.call(this, arg); // this Kontext beibehalten
      cache.set(arg, result);
      return result;
    };
  }

  const memoizedBerechnung = memoizeDecorator(teureBerechnung);
  console.log(memoizedBerechnung(5)); // Berechne für 5..., Ausgabe: 10
  console.log(memoizedBerechnung(5)); // Aus Cache für 5 geholt., Ausgabe: 10
  console.log(memoizedBerechnung(10)); // Berechne für 10..., Ausgabe: 20
  ```

- **Validierung:** Überprüfung von Argumenten vor dem Aufruf der originalen Funktion.
- **Formatierung:** Umwandlung von Eingabe- oder Ausgabewerten.
- **Debounce und Throttle:** (Kurze Erwähnung, Edube 4.1.9) Steuern, wie oft eine Funktion aufgerufen werden kann, z.B. bei häufigen Events wie Scrollen oder Tastatureingaben.

    - `debounce`: Führt die Funktion erst aus, nachdem eine bestimmte Zeitspanne seit dem letzten Aufrufversuch vergangen ist. Nützlich z.B. für Suchvorschläge.
    - `throttle`: Stellt sicher, dass die Funktion höchstens einmal innerhalb eines bestimmten Zeitintervalls ausgeführt wird. Nützlich z.B. für Scroll-Events.

Decorators sind ein mächtiges Werkzeug, um Funktionen wiederverwendbar zu erweitern und das Single Responsibility Principle zu wahren.

Referenz:

- MDN Web Docs: Higher-order function
- Decorator Pattern (allgemeines Softwareentwurfsmuster)

Übungsaufgaben zu Block C (First-Class Functions & Decorators)

Aufgabe C.1: Funktionen in Array Erstelle ein Array, das drei verschiedene Funktionen speichert:

1. Eine Funktion, die eine Zahl quadriert.
2. Eine Funktion, die eine Zahl verdoppelt.
3. Eine Funktion, die von einer Zahl 1 subtrahiert. Schreibe dann eine Funktion `verarbeiteZahl(zahl, funktionenArray)`, die die gegebene `zahl` nacheinander durch jede Funktion im `funktionenArray` verarbeitet und das Endergebnis zurückgibt. Beispiel: `verarbeiteZahl(3, [quadriere, verdopple, subtrahiereEins])` sollte `(3*3 = 9) -> (9*2 = 18) -> (18-1 = 17)` ergeben.

Aufgabe C.2: Einfacher Logging-Decorator Schreibe einen Decorator `einfacherLogger`, der eine Funktion `fn` entgegennimmt. Die zurückgegebene (dekorierte) Funktion soll vor dem Aufruf von `fn` `"Funktion [Name von fn] wird aufgerufen..."` und nach dem Aufruf `"Funktion [Name von fn] beendet."` auf der Konsole ausgeben. Das Ergebnis von `fn` soll normal zurückgegeben werden. Teste ihn mit einer einfachen Additionsfunktion.

Aufgabe C.3: Argument-Validierungs-Decorator Erstelle einen Decorator `validierePositiveZahlen`, der eine Funktion `fn` entgegennimmt, die eine beliebige Anzahl von Argumenten erwartet. Die dekorierte Funktion soll prüfen, ob alle übergebenen Argumente positive Zahlen sind. Wenn ja, wird `fn` aufgerufen. Wenn nicht, soll eine Fehlermeldung ausgegeben werden (z.B. `"Fehler: Alle Argumente müssen positive Zahlen sein."`) und `undefined` zurückgegeben werden. Teste dies mit einer Funktion `summierePositive(...zahlen)`, die nur positive Zahlen addieren soll.

Aufgabe C.4: Zeitmessungs-Decorator Erweitere den `loggingDecorator` aus Aufgabe C.2 oder erstelle einen neuen `zeitMessungsDecorator`. Dieser soll zusätzlich die Ausführungszeit der dekorierten Funktion messen (z.B. mit `performance.now()` oder `Date.now()`) und nach dem Aufruf ausgeben: `"Funktion [Name] benötigte [Zeit] ms."`.

Aufgabe C.5: Decorator für Ergebnistransformation Schreibe einen Decorator `ergebnisZuStringDecorator`, der eine Funktion `fn` entgegennimmt. Die dekorierte Funktion soll `fn` aufrufen und deren Ergebnis (egal welchen Typs) immer in einen String umwandeln, bevor es zurückgegeben wird. Teste mit einer Funktion, die eine Zahl zurückgibt und einer, die ein Objekt zurückgibt.

Micro-Projekt: Utility-Funktionen - Phase 2 (Anwendung von `bind` oder Decorators)

Wir erweitern unsere `textUtils.js` Bibliothek.

Aufgabe für das Micro-Projekt (Phase 2):

Wähle **eine** der folgenden Optionen, um die `formatText` Funktion oder eine neue Funktion in `textUtils.js` zu erweitern:

Option 1: Vorkonfigurierte Formatierer mit `bind` Erstelle in `textUtils.js` zwei neue Funktionen, die auf `formatText` basieren, aber vorkonfigurierte Optionen verwenden, indem sie `bind` nutzen:

- `formatLogNachricht(text)`: Soll `formatText` intern so aufrufen, dass immer `prefix: "[LOG]: "` und `suffix: "..."` verwendet wird. `text` wird als normales Argument übergeben.
- `formatFehlerNachricht(text)`: Soll `formatText` intern so aufrufen, dass immer `uppercase: true`, `prefix: "[FEHLER]: "` verwendet wird.

Option 2: Logging-Decorator für Utility-Funktionen Erstelle einen Decorator `logUtilityAufruf` in einer Datei `decorators.js`. Dieser Decorator soll den Namen der aufgerufenen Utility-Funktion und ihre Argumente loggen, bevor die Funktion ausgeführt wird. Wende diesen Decorator auf die `formatText` Funktion (oder eine andere Utility-Funktion deiner Wahl in `textUtils.js`) an.

Beispiel für Option 1 (Konzept):

```
// In textUtils.js
// Angenommen formatText ist definiert wie in Phase 1
// export function formatText({ text, uppercase = false, prefix = "", suffix = "" }) { /* ... */
}

// const basisFormatText = ({ textIn, ...optionen }) => formatText({ text: textIn, ...optionen
}); // Hilfsfunktion für bind

// export const formatLogNachricht = basisFormatText.bind(null, { prefix: "[LOG]: ", suffix:
"..." });
// export const formatFehlerNachricht = basisFormatText.bind(null, { uppercase: true, prefix: "
[FEHLER]: "});

// Aufruf in main.js:
// console.log(formatLogNachricht({textIn: "System gestartet"})); // Erwartet: [LOG]: System
gestartet...
// console.log(formatFehlerNachricht({textIn: "Kritischer Fehler"})); // Erwartet: [FEHLER]:
KRITISCHER FEHLER
```

(Hinweis: Die `bind`-Nutzung mit Objekt-Destrukturierung als erstem Parameter ist etwas knifflig. Eventuell ist eine kleine Anpassung an `formatText` oder eine Wrapper-Funktion für `bind` nötig, damit die Argumente richtig durchgereicht werden, falls `formatText` weiterhin ein Objekt erwartet. Das gezeigte `basisFormatText` ist ein Ansatz dafür.)

Wähle die Option, die dir mehr zusagt, um die Konzepte zu üben.