Tag 7 - JavaScript: Fehler und Debugging

Einleitung

Fehler in Programmen sind unvermeidlich. Sie treten auf, sobald wir eine Anwendung schreiben, ausführen, verändern oder mit anderen Systemen kommunizieren. Sie können aus Syntaxproblemen, logischen Irrtümern, Typfehlern oder falscher Nutzerinteraktion entstehen. Deshalb ist es entscheidend, Fehlerarten zu erkennen, richtig mit ihnen umzugehen und sie gezielt zu beheben. In diesem Kapitel lernst du:

- Welche Arten von Fehlern es gibt (Syntax, Reference, Type, Range usw.)
- Wie man Fehler behandelt (`try...catch...finally`)
- Wie man eigene Fehler mit `throw` erzeugt
- Wie man Debugging-Werkzeuge im Browser nutzt
- Wie man Code performant analysiert

Kapitel 1: JavaScript-Fehlerarten und Ausnahmebehandlung

1.1 Fehlerarten in JavaScript

1. SyntaxError: Tritt auf, wenn der Code den Sprachregeln nicht entspricht. Der Code kann dann nicht interpretiert oder ausgeführt werden.

```
let x = ; // SyntaxError: Unexpected token ';'
```

2. ReferenceError: Wird ausgelöst, wenn auf eine nicht deklarierte Variable oder Funktion zugegriffen wird.

```
console.log(zahl); // ReferenceError: zahl is not defined
```

3. TypeError: Entsteht, wenn eine Operation auf einem ungeeigneten Datentyp versucht wird.

```
let x = 5;
x(); // TypeError: x is not a function
```

4. RangeError: Tritt auf, wenn ein Wert außerhalb eines zulässigen Bereichs liegt.

```
let arr = new Array(-1); // RangeError: Invalid array length
```

5. Logical Error (Logikfehler): Formal korrekter Code, der aber nicht das tut, was er soll:

```
function average(a, b) {
  return a + b / 2; // Fehler! Statt (a + b) / 2
}
```

1.2 Beispielhafte Fehleranalyse mit Kommentar

```
// Ziel: Multiplikation zweier Zahlen
let multiply = (a, b) => a + b; // Logikfehler: sollte * sein
let result = multiply(10, 20);
console.log(result); // Ausgabe: 30 (falsch)
```

Erklärung: Kein Syntaxfehler, aber ein Logikfehler. Der Ausdruck soll multiplizieren, verwendet aber Addition.

Kapitel 2: Fehlerbehandlung mit `try...catch` und `finally`

2.1 Grundstruktur

```
try {
  // Code, der fehlschlagen könnte
} catch (error) {
  // Fehlerbehandlung
} finally {
  // Optional: wird immer ausgeführt
}
```

2.2 Beispiel mit Referenzfehler

Was ist das `error`-Argument im `catch`-Block?

Wenn innerhalb eines `try`-Blocks ein Fehler auftritt, wird ein spezielles **Error-Objekt** erzeugt und an den `catch`-Block übergeben. Du kannst diesem Objekt im `catch`-Block einen beliebigen Namen geben (z. B. `error`, `e`, `err`), in der Praxis wird meist `error` verwendet.

Dieses Objekt enthält wichtige Informationen über den Fehler:

Standard-Eigenschaften und Methoden des Error-Objekts:

- `error.name`
  - Gibt den Typ des Fehlers an (z. B. `ReferenceError`, `TypeError`)
- `error.message`
  - Enthält die vom Fehlerobjekt gelieferte Fehlermeldung als lesbarer Text
- `error.stack`
  - (Optional) Stack-Trace, der zeigt, in welcher Zeile und Funktion der Fehler aufgetreten ist

Beispiel:

```
try {
  notDefinedFunction();
} catch (error) {
  console.log("Name:", error.name);        // ReferenceError
  console.log("Message:", error.message);  // notDefinedFunction is not defined
  console.log("Stack:", error.stack);      // Stacktrace mit Datei und Zeile
}
```

Weitere Hinweise:

- Du kannst mit `instanceof` überprüfen, zu welchem Typ ein Fehlerobjekt gehört (`error instanceof ReferenceError`)
- Das `stack`-Property ist besonders nützlich beim Debugging, um den Ursprung eines Fehlers zu lokalisieren
- Du kannst auch eigene Fehlerobjekte erzeugen mit `throw new Error("...")` oder `throw new TypeError("...")`

```
try {
  let x = y; // y ist nicht deklariert
} catch (err) {
  console.log("Fehler abgefangen:", err.message);
} finally {
  console.log("Aufräumarbeiten durchgeführt.");
}
```

Erklärung: Der `catch`-Block verhindert den Programmabbruch und erlaubt eine benutzerdefinierte Reaktion. `finally` wird immer ausgeführt, selbst wenn kein Fehler vorlag.

2.3 Unterschiedliche Fehler gezielt behandeln

```
try {
  nichtDefiniert();
} catch (e) {
  if (e instanceof ReferenceError) {
    console.log("Referenzproblem erkannt.");
  } else {
    console.log("Anderer Fehler: ", e.message);
  }
}
```

Kapitel 3: Eigene Fehler erzeugen mit `throw`

3.1 Definition

Mit dem `throw`-Statement kannst du bewusst eigene Fehler auslösen, z. B. um ungültige Eingaben abzufangen.

Wenn du dabei `throw new Error("Nachricht")` (oder z. B. `TypeError`, `RangeError`) schreibst, wird die übergebene Zeichenkette als **`message`-Eigenschaft** des Error-Objekts gespeichert.

Das heißt: **Du überschreibst damit die Standardfehlermeldung**.

Beispiel:

```
try {
  throw new Error("Benutzername darf nicht leer sein");
} catch (error) {
  console.log(error.name);    // "Error"
  console.log(error.message); // "Benutzername darf nicht leer sein"
}
```

Diese Zeichenkette kannst du im `catch`-Block über `error.message` auslesen. Mit dem `throw`-Statement kannst du bewusst eigene Fehler auslösen, z. B. um ungültige Eingaben abzufangen.

3.2 Beispiele

```
function teilen(x, y) {
  if (y === 0) {
    throw new Error("Division durch 0 nicht erlaubt");
  }
  return x / y;
}

try {
  console.log(teilen(10, 0));
} catch (err) {
  console.log("FEHLER:", err.message);
}
```

3.3 Vordefinierte Fehlerobjekte

Neben dem allgemeinen `Error`-Objekt gibt es spezifischere Fehlerklassen, die z. B. nach Typ oder Anwendungsfall unterscheiden:

```
throw new ReferenceError("Variable fehlt");
throw new RangeError("Wert außerhalb des Bereichs");
throw new TypeError("Ungültiger Datentyp");
```

Auch hier ist der übergebene String die `message`-Eigenschaft des Fehlerobjekts.

3.4 Eigene Fehlerklassen (Optional für Fortgeschrittene)

Du kannst auch eigene Fehlerklassen definieren, indem du von `Error` erbst:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

try {
  throw new ValidationError("Email-Adresse ist ungültig");
} catch (error) {
  console.log(error.name);    // "ValidationError"
  console.log(error.message); // "Email-Adresse ist ungültig"
}
```

Diese Technik eignet sich, wenn du in komplexeren Programmen verschiedene Fehlerarten unterscheiden willst (z. B. `ValidationError`, `AuthError`, `ApiError`, usw.)

```
throw new ReferenceError("Variable fehlt");
throw new RangeError("Wert außerhalb des Bereichs");
throw new TypeError("Ungültiger Datentyp");
```

Hinweis: Durch `new` wird ein Objekt der entsprechenden Fehlerklasse erzeugt.

Kapitel 4: Debugging mit Developer Tools im Browser (Chrome)

4.1 Vorbereitung

1. HTML-Datei mit Script einbinden

```
<script src="main.js"></script>
```

2. Öffne DevTools mit `Strg+Shift+I` (Windows) oder `Cmd+Opt+I` (Mac)
3. Wechsel zu **Sources** (Chrome) bzw. **Debugger** (Firefox)

4.2 Debugger aktivieren

```
console.log("vorher");
debugger;
console.log("nachher");
```

Ergebnis: Das Programm stoppt an der Stelle `debugger;`. Du kannst die Ausführung analysieren.

4.3 Debugging-Buttons im Überblick (Chrome)

Funktion	Tastenkombination	Beschreibung
Resume	F8	Fortfahren bis zum nächsten Breakpoint
Step Over	F10	Eine Zeile weiter, Funktionen werden als Ganzes ausgeführt
Step Into	F11	Springt in aufgerufene Funktionen hinein
Step Out	Shift + F11	Springt aus der aktuellen Funktion zurück
Set Breakpoint	Klick auf Zeilennummer	Pausiert das Programm an dieser Stelle

4.4 Beispielschrittfolge

```
function a() {
  let x = 10;
  b();
  console.log("Fertig");
}
function b() {
  let y = 20;
  console.log("In Funktion b");
}
a();
```

Vorgehen:

1. Setze Breakpoint bei `let x = 10;`
2. Starte Seite neu (F5)
3. Schrittweise mit F10 durchgehen
4. Mit F11 in Funktion `b()` springen
5. Mit Shift+F11 wieder zurück

4.5 Variablen beobachten

- Nutze **Scope / Lokale Variablen**-Panel
- Füge eigene Beobachtungen über das "Watch"-Panel hinzu
- Verwende die DevTools-Konsole für Live-Auswertung:

```
console.log(meineVariable);
```

4.6 Laufzeit messen

```
console.time("loop");
for (let i = 0; i < 1000000; i++) {}
console.timeEnd("loop");
```

Zweck: Optimierung von Algorithmen, z. B. indem du langsame Schleifen durch bessere Varianten ersetzt.
