Javascript

▼ Inhaltsverzeichnis

1. Programmierung mit Javascript

Wir beginnen mit einem Blick in die Vergangenheit. Die Geschichte von JavaScript reicht zurück bis in die 1990er Jahre, als Brendan Eich die Sprache in nur wenigen Tagen entwickelte. Seitdem hat sich JavaScript rasant entwickelt und ist zu einem unverzichtbaren Bestandteil moderner Webseiten geworden. JavaScript ist überall! Ihr findet es in Webbrowsern, auf Servern und sogar in einigen Haushaltsgeräten. Die weite Verbreitung von JavaScript macht es zu einem wertvollen Werkzeug für Entwickler auf der ganzen Welt. Egal, ob ihr eine interaktive Webseite gestalten oder komplexe Server-Anwendungen entwickeln möchtet, JavaScript bietet euch die Tools, um eure Ziele zu erreichen. Aber warum solltet ihr JavaScript lernen, wenn es doch so viele andere Programmiersprachen gibt? JavaScript hat einige einzigartige Eigenschaften, die es von anderen Sprachen unterscheiden. Zum Beispiel ist es die einzige Sprache, die nativ in Webbrowsern läuft, was es zur Sprache der Wahl für Webentwicklung macht. Außerdem bietet JavaScript eine flexible Syntax und eine dynamische Typisierung, die es sowohl für Anfänger als auch für erfahrene Programmierer attraktiv macht. [3]

1.1. Grammatik und Syntax von JavaScript

Die Grammatik von JavaScript legt fest, wie Anweisungen formuliert und organisiert werden müssen, ähnlich wie die Grammatik in einer menschlichen Sprache. Sie bestimmt, wie verschiedene Elemente wie Variablen, Operatoren und Kontrollstrukturen zusammenarbeiten müssen, um logische und funktionierende Code-Blöcke zu erstellen. Die Syntax hingegen ist die Menge an Regeln, die beschreiben, wie Programme in der Sprache geschrieben werden. Dies umfasst Aspekte wie die Deklaration von Variablen, die Verwendung von Klammern, die Struktur von Schleifen und Funktionen und vieles mehr. Ein Verstoß gegen diese Regeln führt in der Regel zu einem Syntaxfehler, der euch daran hindert, euer Programm auszuführen. Das Verstehen der Grammatik und Syntax von JavaScript ist ähnlich wie das Erlernen der Grundlagen einer Fremdsprache. Sobald ihr die Grundlagen verinnerlicht habt, könnt ihr beginnen, komplexere Ausdrücke und Sätze – oder in diesem Fall Funktionen und Anwendungen – zu formulieren.[2]

1.1.1. Deklarationen, Scope und Hoisting

Das Verständnis dieser Konzepte ist entscheidend für die Entwicklung klarer und effektiver JavaScript-Programme. Deklarationen, Scope und Hoisting bilden die Grundlage, auf der viele weitere Konzepte und Techniken aufbauen, die ihr im Laufe eurer JavaScript-Laufbahn kennenlernen werdet.

Deklaration:

Die Deklaration von Variablen und Funktionen ist ein grundlegender Aspekt in JavaScript. Mit Schlüsselwörtern wie `var`, `let` und `const` könnt ihr Variablen deklarieren und ihnen Werte zuweisen. Zum Beispiel:

```
var name = 'Jacob'; // globale Variable
let age = 26;       // blockbasierte Variable
const pi = 3.141; // blockbasierte Konstante
```

- `var` wird verwendet, um eine Variable zu deklarieren, die global oder innerhalb einer Funktion verfügbar ist, unabhängig davon, ob sie innerhalb einer Schleife oder einer Bedingung deklariert wurde. *Es ist die traditionelle Art, Variablen zu deklarieren.*
- `let` wird verwendet, um eine blockbasierte Variable zu deklarieren, die nur innerhalb des Blocks, der sie enthält, oder in einem Unterblock verfügbar ist. *Es ist eine modernere Art, Variablen zu deklarieren.*
- `const` wird verwendet, um eine blockbasierte Konstante wie z.B. `pi` zu deklarieren, deren Wert nach der ersten Zuweisung nicht mehr geändert werden kann.

Scope: Der Scope einer Variablen bezieht sich darauf, wo sie im Code zugänglich ist. Variablen, die innerhalb einer Funktion deklariert werden, sind nur innerhalb dieser Funktion verfügbar (lokaler Scope), während Variablen, die außerhalb von Funktionen deklariert werden, im gesamten Code zugänglich sind (globaler Scope).

```
function myFunction() {
  var localVariable = 'Ich bin lokal'; // Nur innerhalb dieser Fun
ktion sichtbar
}
var globalVariable = 'Ich bin global'; // Überall im Code sichtbar
```

Hosting: Hoisting ist ein etwas komplexeres Konzept, aber ebenso wichtig zu verstehen. In JavaScript werden Deklarationen an den Anfang ihres Scopes "gehoben". Das bedeutet, dass ihr auf eine Variable zugreifen könnt, bevor ihr sie deklariert habt, aber das kann zu unerwarteten Ergebnissen führen. Zum Beispiel:

```
console.log(x); // undefined
var x = 5;
console.log(x); // 5
```

Bei Verwendung von `let` und `const` tritt ein Fehler auf, wenn ihr versucht, auf die Variable zuzugreifen, bevor sie deklariert ist. Hoisting ist ein Grund, warum viele Entwickler `let` und `const` `var` vorziehen, da sie ein vorhersehbares Verhalten bieten. `let` bietet im Vergleich zu `var` eine feinere Kontrolle über den Variablenzugriff. `let`

ermöglicht es, den Wert einer Variablen nach ihrer Deklaration zu ändern (sofern sie nicht als `const` deklariert wurde). Es schützt jedoch vor einigen der Verwirrungen und Fehlverhalten, die durch die Verwendung von `var` auftreten könnten, wie zum Beispiel beim Hoisting. Hier ist ein Beispiel, das den Unterschied zwischen `let` und `var` nocheinmal genauer verdeutlicht:

```
if (true) {
  let letVar = 'block scope';
  var varVar = 'function scope';
}

console.log(typeof letVar);
// Gibt "undefined" aus, da letVar außerhalb des Blocks nicht sich
tbar ist

console.log(typeof varVar);
 // Gibt "string" aus, da varVar auf die gesamte Funktion sichtbar
ist
```

Die Verwendung von `let` für Variablendeklarationen wird oft als best practice angesehen, da es hilft, unerwartete Verhaltensweisen im Zusammenhang mit Variablensichtbarkeit und Hoisting zu vermeiden.

1.1.2. Datentypen und Literale

In JavaScript, wie auch in vielen anderen Programmiersprachen, ist es wichtig, die Art der Daten zu verstehen, mit denen ihr arbeitet. Daten in JavaScript können in verschiedene Datentypen eingeteilt werden, die bestimmen, welche Werte eine Variable annehmen kann und welche Operationen auf diesen Werten ausgeführt werden können.

Primitive Datentypen

In JavaScript gibt es verschiedene Datentypen, die als "primitiv" bezeichnet werden. Diese sind die Grundbausteine, mit denen alle Informationen und Logiken im Code dargestellt werden können. Primitive Datentypen sind unveränderlich, was bedeutet, dass ihre Werte nicht verändert werden können, sobald sie erstellt wurden.

`Number`**:** Der Datentyp Number kann sowohl Ganzzahlen als auch Fließkommazahlen darstellen. Er wird für alle numerischen Operationen verwendet, die ihr in JavaScript ausführen möchtet.

```
var integer = 42; // Ganzzahl
var floatingPoint = 3.14; // Fließkommazahl
```

`String`**:** Ein String ist eine Sequenz von Zeichen, die Text darstellt. Ihr könnt Strings mit doppelten Anführungszeichen, einfachen Anführungszeichen oder

Rückwärtsschrägstrichen (für Template Literals) erstellen.

```
var singleQuote = 'Hallo Welt!';
var doubleQuote = "Hallo Welt!";
```

`Boolean`: Ein Boolean ist ein logischer Wert, der entweder `true` oder `false` sein kann. Er wird oft in Bedingungen verwendet, um einen Codepfad auszuwählen.

```
var isTrue = true;
var isFalse = false;
```

`Undefined`: Der Typ `undefined` wird einer nicht zugewiesenen Variable zugeordnet. Es zeigt an, dass die Variable deklariert wurde, aber noch keinen Wert hat.

```
var value;
```

`Null`: null repräsentiert die absichtliche Abwesenheit eines Wertes und zeigt an, dass eine Variable absichtlich leer gelassen wurde.

```
var emptyValue = null;
```

`Symbol`: Ein Symbol ist ein einzigartiger und unveränderlicher Wert. Es wird oft verwendet, um einzigartige Eigenschaftsschlüssel innerhalb von Objekten zu erstellen.

```
var uniqueKey = Symbol('description');
```

`BigInt`: `BigInt` ist ein Typ für ganze Zahlen beliebiger Größe, über das hinaus, was mit dem Number-Typ dargestellt werden kann. Es wird verwendet, wenn sehr große Zahlen benötigt werden, die außerhalb des sicheren Integer-Bereichs von JavaScript liegen.

```
var bigNumber = 1234567890123456789012345678901234567890n;
```

Zusammenfassung der Primitive Datentypen

Datentyp	Beschreibung	Beispiel
Number	Ganzzahlen und Fließkommazahlen	let x = 42; let y = 3.14;
String	Sequenz von Zeichen	let name = "Alice";
Boolean	true oder false	let isTrue = true;
Undefined	Nicht zugewiesene Variable	let something;

Null	Absichtliche Abwesenheit eines Wertes	let nothing = null;
Symbol	Einzigartiger und unveränderlicher Datentyp	const uniqueKey = Symbol('description');
BigInt	Ganze Zahlen beliebiger Größe	const bigNumber = 1234567890123456789012345678901234567890n;

Tabelle 1.1: Zusammenfassung der Primitive Datentypen

Literale

Literale in der Programmierung sind feste Werte, die direkt im Quellcode geschrieben werden. Sie repräsentieren konkrete, unveränderliche Werte eines bestimmten Datentyps. In JavaScript gibt es verschiedene Arten von Literalen, um unterschiedliche Werte darzustellen.

🔎 **Ein Beispiel zum Verständnis:**

- Der primitive Datentyp könnte “Number” sein, der alle möglichen numerischen Werte umfasst.
- Ein numerisches Literal wäre eine konkrete Zahl in Ihrem Code, wie 42.

Hier sind einige Beispiele für Literale in JavaScript:

```
let myNumber = 42; // 42 ist ein numerisches Literal
let myString = "Hallo Welt!"; // "Hallo Welt!" ist ein String-Lite
ral
let myBoolean = true; // true ist ein boolesches Literal
let myArray = [1, 2, 3]; // [1, 2, 3] ist ein Array-Literal
let myObject = { name: 'Jacob', age: 26 }; // { name: 'Jacob', ag
e: 26 } ist ein Objekt-Literal
```

In diesem Beispiel sind die Werte auf der rechten Seite der Zuweisungen Literale. Sie repräsentieren konstante, fest codierte Werte im Code. Die Variablen (`myNumber`, `myString`, etc.) speichern diese Werte, aber die Literale selbst sind die direkten Darstellungen der Werte.

🧠 **Was ist der Unterschied zwischen den beiden Konzepten?**

- **Primitiver Datentyp:** Beschreibt die Art eines Wertes. Diese Datentypen sind "primitiv", weil sie keine Eigenschaften oder Methoden haben und sie nicht weiter unterteilt werden können.
- **Literal:** Ist eine konkrete Darstellung dieses Wertes im Code. Literale hingegen sind feste Werte, die ihr im Quellcode verwenden könnt. Sie sind die tatsächliche Repräsentation eines Wertes eines primitiven Datentyps oder eines zusammengesetzten Datentyps (wie ein Objektliteral).

1.2. Kontrollstrukturen

In der Programmierung ist es häufig notwendig, den Fluss des Programms basierend auf bestimmten Bedingungen oder Kriterien zu steuern. Kontrollstrukturen sind essentielle Bausteine, die es Entwicklern ermöglichen, den Ablauf eines Programms zu kontrollieren. Sie können dazu verwendet werden, Code in unterschiedlichen Wegen zu steuern, abhängig von den Eingabebedingungen und den spezifischen Anforderungen der Aufgabe. In diesem Abschnitt werden wir uns die verschiedenen Arten von Kontrollstrukturen in JavaScript genauer ansehen und wie sie dazu verwendet werden können, komplexe Logik und Funktionalitäten in Ihren Programmen zu implementieren.

1.2.1. Bedingte Anweisungen: if-else, switch

Bedingte Anweisungen sind ein zentraler Bestandteil der meisten Programmiersprachen, einschließlich JavaScript. Mit ihnen kann der Code auf der Grundlage bestimmter Bedingungen in unterschiedliche Richtungen gelenkt werden. In JavaScript gibt es zwei Hauptarten von bedingten Anweisungen: if-else und switch.

if-else

Die if-else-Struktur ermöglicht es, einen Codeblock auszuführen, wenn eine Bedingung erfüllt ist und optional einen anderen Codeblock, wenn sie nicht erfüllt ist.

```
let wetter = "sonnig";

if (wetter === "sonnig") {
  console.log("Vergiss die Sonnenbrille nicht!");
} else {
  console.log("Du brauchst wohl einen Regenschirm.");
}
```

In diesem Beispiel wird die Nachricht über die Sonnenbrille nur dann angezeigt, wenn die Variable `wetter` den Wert "sonnig" hat. Andernfalls wird die Nachricht über den Regenschirm angezeigt.

switch

Die switch-Anweisung ist eine weitere Möglichkeit, mehrere Bedingungen zu überprüfen. Sie vergleicht einen Ausdruck mit mehreren möglichen Werten und führt den entsprechenden Codeblock aus.

```
let tag = 3;

switch (tag) {
  case 1:
    console.log("Montag");
    break;
  case 2:
    console.log("Dienstag");
    break;
  case 3:
    console.log("Mittwoch");
    break;
  // Weitere Fälle...
  default:
    console.log("Ungültiger Tag!");
}
```

In diesem Beispiel wird der Wert der Variable `tag` mit den Werten in den einzelnen `case`-Anweisungen verglichen. Wenn eine Übereinstimmung gefunden wird, wird der entsprechende Codeblock ausgeführt. Der `break`-Befehl verhindert, dass der Code in den nächsten `case`-Block übergeht. Wenn keine Übereinstimmung gefunden wird, wird der `default`-Block ausgeführt.

1.2.2. Fehlerbehandlung: try/catch/throw, Error, Promise

In der Programmierung ist es wichtig, auf unerwartete Situationen oder Fehler vorbereitet zu sein, die während der Ausführung eines Programms auftreten können. Fehler können aus einer Vielzahl von Gründen auftreten, wie z.B. ungültigen Eingaben, fehlerhaften externen Ressourcen oder unerwarteten Programmzuständen. In JavaScript bietet die Fehlerbehandlung die Mittel, um auf solche unerwarteten Ereignisse zu reagieren und das Programm so robust wie möglich zu gestalten.

Try/Catch/Throw

Das try/catch/throw-Konstrukt in JavaScript ermöglicht es Ihnen, Codeblöcke auszuführen, die potenziell zu einem Fehler führen könnten und diesen Fehler dann auf elegante Weise zu behandeln.

```
try {
  // Code, der einen Fehler werfen könnte
```

```
  throw new Error('Ein Fehler ist aufgetreten!');
} catch (error) {
  // Fehlerbehandlung
  console.error(error.message);
}
```

- `try`: Dieser Block enthält Code, der möglicherweise einen Fehler werfen könnte. Im Beispiel wird ein Fehler mit der Nachricht "Ein Fehler ist aufgetreten!" sofort mit dem throw-Statement geworfen.
- `catch`: Wenn innerhalb des try-Blocks ein Fehler auftritt, wird der Code im catch-Block ausgeführt. Das Fehlerobjekt wird als Error gefangen und seine Nachricht wird auf der Konsole ausgegeben.

Error

Der Error-Konstruktor in JavaScript wird verwendet, um eine Fehlerinstanz zu erstellen, die dann mit `throw` geworfen werden kann. Es gibt auch spezifische Unterarten von Error, wie `TypeError`, `ReferenceError` usw., um bestimmte Arten von Fehlern zu kennzeichnen.

```
throw new TypeError('Das ist ein Typfehler!');
```

Dieses Beispiel zeigt, wie ein spezifischer Fehler vom Typ `TypeError` mit einer zugehörigen Nachricht geworfen wird. Es wird verwendet, um auf Fehler hinzuweisen, die aufgrund eines inkompatiblen Datentyps auftreten.

Promise

Promises sind ein leistungsfähiges Konzept in JavaScript, das die Arbeit mit asynchronen Operationen erleichtert. Sie können auch verwendet werden, um Fehler in asynchronen Abläufen zu behandeln.

```
const myPromise = new Promise((resolve, reject) => {
  if (/* irgendeine Bedingung */) {
    resolve('Erfolg!');
  } else {
    reject(new Error('Etwas ist schiefgelaufen.'));
  }
});

myPromise
  .then(result => {
    console.log(result);
  })
  .catch(error => {
```

```
    console.error(error.message);
  });
```

Ein Promise wird erstellt, um asynchrone Operationen darzustellen. Wenn die Bedingung innerhalb des Promise erfüllt ist, wird es mit 'Erfolg!' aufgelöst; wenn nicht, wird es mit einem Fehler abgelehnt.

- `.then(result => {...})`: Wenn das Versprechen erfüllt ist (d.h., mit `resolve` aufgelöst), wird dieser Code mit dem Resultatwert "Erfolg!" ausgeführt.
- `.catch(error => {...})`: Wenn das Versprechen abgelehnt wird (d.h., mit `reject` und einem Fehler), wird dieser Code ausgeführt. Der gefangene Fehler wird auf der Konsole ausgegeben.

1.3. Schleifen programmieren

Schleifen sind ein unverzichtbares Konzept in der Programmierung, das es ermöglicht, bestimmte Blöcke von Code wiederholt auszuführen. Diese Wiederholung kann genutzt werden, um über eine Datenstruktur wie ein Array zu iterieren, eine bestimmte Bedingung zu erfüllen oder einfach eine Aufgabe mehrmals auszuführen. In JavaScript gibt es verschiedene Arten von Schleifen, die jeweils in unterschiedlichen Situationen verwendet werden können. Die Verwendung von Schleifen hilft nicht nur, den Code zu verkürzen und wiederholte Aufgaben zu automatisieren, sondern trägt auch zur Lesbarkeit und Wartbarkeit des Codes bei.

1.3.1. for, while, do-while, break/continue, for-in, for-of

In diesem Abschnitt beschäftigen wir uns intensiv mit dem Programmieren von Schleifen in JavaScript. Wir werden verschiedene Arten von Schleifen kennenlernen, darunter die gängigen for, while und do...while Schleifen, sowie einige speziellere Schleifenkonstrukte wie for...in und for...of. Jede dieser Schleifen bietet unterschiedliche Möglichkeiten und kann je nach Bedarf und Kontext eingesetzt werden.

for-Schleife

Die for-Schleife ist eine der grundlegendsten Kontrollstrukturen in JavaScript und wird verwendet, um einen Block von Code mehrmals auszuführen. Die Struktur einer for-Schleife besteht aus drei Hauptteilen: Initialisierung, Bedingung und Iteration (Aktualisierung). Hier ist die allgemeine Syntax einer for-Schleife:

```
for (Initialisierung; Bedingung; Iteration) {
  // Code, der wiederholt ausgeführt wird
}
```

- **Initialisierung:** Dies ist der erste Ausdruck in der Schleife und er wird nur einmal zu Beginn der Schleife ausgeführt. Er wird normalerweise verwendet, um einen Zähler

zu initialisieren.

- **Bedingung:** Dieser Ausdruck wird vor jedem Durchlauf der Schleife überprüft. Wenn die Bedingung wahr (true) ist, wird der Code innerhalb der Schleife ausgeführt. Wenn die Bedingung falsch (false) ist, wird die Schleife beendet.
- **Iteration:** Dieser Ausdruck wird am Ende eines jeden Durchlaufs ausgeführt und wird normalerweise verwendet, um den Zähler zu aktualisieren.

Hier ist ein einfaches Beispiel, das die for-Schleife in Aktion zeigt:

```
for (let i = 0; i < 5; i++) {
  console.log(i); // Gibt 0 bis 4 aus
}
```

Diese Schleife verwendet eine Zählvariable `i`, die von 0 beginnt und bei jedem Durchlauf um 1 erhöht wird, solange `i` kleiner als 5 ist. In jeder Iteration wird der Wert von `i` ausgegeben, also 0, 1, 2, 3, 4.

- **Initialisierung:** `let i = 0` setzt einen Zähler `i` auf 0.
- **Bedingung:** `i < 5` überprüft, ob `i` kleiner als 5 ist.
- **Iteration**: `i++` erhöht `i` um 1 bei jedem Durchlauf.

Die for-Schleife wird oft verwendet, wenn die Anzahl der Iterationen im Voraus bekannt ist, wie z.B. beim Durchlaufen eines Arrays:

```
const fruits = ['apple', 'banana', 'cherry'];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]); // Gibt jedes Element im Array aus
}
```

In JavaScript gibt es auch Variationen der for-Schleife, wie die for-of-Schleife, die das Durchlaufen von iterierbaren Objekten wie Arrays vereinfacht und die for-in-Schleife, die für das Durchlaufen von Eigenschaften eines Objekts verwendet wird.

for-of-Schleife: Wird verwendet, um durch die Werte von iterierbaren Objekten wie Arrays zu iterieren.

```
const array = [1, 2, 3];
for (const value of array) {
  console.log(value); // Gibt 1, 2, 3 aus
}
```

Die for-of-Schleife iteriert durch die Werte des Arrays `array` und gibt jeden Wert nacheinander aus.

for-in-Schleife: Wird verwendet, um durch die Eigenschaften eines Objekts zu iterieren.

```
const obj = { a: 1, b: 2, c: 3 };
for (const key in obj) {
  console.log(key, obj[key]); // Gibt "a 1", "b 2", "c 3" aus
}
```

Diese Schleife iteriert durch die Schlüssel des Objekts `obj` und gibt jeden Schlüssel zusammen mit seinem zugehörigen Wert aus.

while-Schleife

Die while-Schleife ist eine der grundlegenden Schleifenstrukturen in JavaScript und vielen anderen Programmiersprachen. Sie ermöglicht es, einen Codeblock so lange auszuführen, wie eine bestimmte Bedingung wahr ist (d.h., so lange die Bedingung den Wert `true` zurückgibt). Die Syntax einer while-Schleife ist einfach und übersichtlich:

```
while (Bedingung) {
  // Code, der ausgeführt wird, solange die Bedingung wahr ist
}
```

Ein typisches Beispiel könnte sein, eine Zählvariable von 1 bis 10 zu inkrementieren:

```
let count = 1;

while (count <= 10) {
  console.log("Zählung: " + count);
  count++; // Inkrementiert die Zählvariable um 1
}
```

In diesem Fall wird die while-Schleife solange durchlaufen, bis die Variable `count` den Wert 10 überschreitet. Nach jedem Durchlauf wird die Variable um eins erhöht und die Ausgabe zeigt die Zählung von 1 bis 10.

Die while-Schleife wird oft verwendet, wenn die Anzahl der Durchläufe im Voraus nicht bekannt ist. Sie eignet sich besonders, wenn die Bedingung auf einer dynamischen Berechnung oder einer externen Eingabe basiert. Eine wichtige Vorsichtsmaßnahme bei der Verwendung von while-Schleifen ist das Vermeiden einer Endlosschleife. Wenn die Bedingung immer wahr bleibt und es keinen Code gibt, der die Bedingung irgendwann falsch macht, wird die Schleife unbegrenzt weiterlaufen. Dies kann zu einem Absturz oder Einfrieren des Programms oder der Umgebung führen, in der der Code ausgeführt wird. Deshalb ist es entscheidend, sicherzustellen, dass die Bedingung in der Schleife irgendwann falsch wird, damit die Schleife ordnungsgemäß beendet werden kann.

do-while-Schleife: Ähnlich wie die while-Schleife, aber der Codeblock wird mindestens einmal ausgeführt, auch wenn die Bedingung von Anfang an falsch ist.

```
let i = 5;
do {
  console.log(i); // Gibt 5 aus
} while (i < 5);
```

Hier wird der Codeblock einmal ausgeführt, auch wenn die Bedingung (`i < 5`) falsch ist. `i` wird zu 5 gesetzt und die 5 wird gedruckt.

break-Statement

Das break-Statement in JavaScript wird verwendet, um eine Schleife vollständig zu beenden oder aus einem Schalter (`switch`) herauszuspringen. Sobald ein break-Statement ausgeführt wird, wird die Ausführung des Codes innerhalb der Schleife oder des Schalters sofort beendet und das Programm fährt mit dem Code fort, der nach der Schleife oder dem Schalter steht.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) {
    break; // Unterbricht die Schleife, wenn i gleich 5 ist
  }
  console.log(i); // Gibt 0, 1, 2, 3, 4 aus
}
```

In diesem Beispiel wird die for-Schleife verwendet, um die Zahlen von 0 bis 9 zu zählen. Sobald `i` den Wert 5 erreicht, wird das break-Statement ausgeführt, das die Schleife sofort unterbricht und den Code nach der Schleife fortsetzt. Daher werden nur die Werte 0, 1, 2, 3 und 4 in der Konsole ausgegeben. Das `break`-Statement ist nützlich, um eine Schleife vorzeitig zu beenden, wenn eine bestimmte Bedingung erfüllt ist.

continue-Statement

Das `continue`-Statement wird innerhalb einer Schleife verwendet, um den aktuellen Durchlauf der Schleife zu beenden und unmittelbar mit dem nächsten Durchlauf fortzufahren. Im Gegensatz zum break-Statement, das die gesamte Schleife beendet, überspringt `continue` nur den Rest des aktuellen Schritt und fährt mit dem nächsten Durchgang fort.

```
for (let i = 0; i < 5; i++) {
  if (i === 3) continue;
  console.log(i); // Gibt 0, 1, 2, 4 aus
}
```

Diese Schleife überspringt die Iteration, in der `i` gleich 3 ist, aufgrund des `continue`-Statements. Daher werden die Zahlen 0, 1, 2, 4 gedruckt.

1.4. Operatoren und Ausdrücke

In diesem Abschnitt werden wir uns auf die spezifischen Operatoren und Ausdrücke konzentrieren, die in JavaScript verwendet werden. Dabei werdet ihr erfahren, wie verschiedene Operatoren wie arithmetische, logische und vergleichende Operatoren funktionieren und wie sie kombiniert werden können, um komplexe Ausdrücke zu bilden.

1.4.1. Zuweisungen und Vergleiche

Die Zuweisungs- und Vergleichsoperationen sind fundamentale Konzepte in der Programmierung mit JavaScript. Hier wird ein Überblick über diese Operationen gegeben und wie sie in verschiedenen Kontexten verwendet werden können.

Zuweisungen

Zuweisungen sind in der Programmierung dazu da, Werte in Variablen zu speichern. In JavaScript gibt es verschiedene Zuweisungsoperatoren. Der bekannteste davon ist der Gleichheitsoperator `=`:

```
let x = 10; // Zuweisung des Wertes 10 zur Variable x
```

Es gibt auch kombinierte Zuweisungsoperatoren wie `+=`, `-=`, `*=` und `/=` die eine Operation und eine Zuweisung gleichzeitig ausführen. Kombinierte Zuweisungsoperatoren in JavaScript sind eine nützliche und effiziente Möglichkeit, einfache arithmetische Operationen mit einer Variablen auszuführen und das Ergebnis in der gleichen Variablen zu speichern. Diese Operatoren kombinieren eine arithmetische Operation wie Addition, Subtraktion, Multiplikation oder Division mit einer Zuweisung, wodurch der Code kompakter und lesbarer wird.

```
let x = 10;
x += 5; // Addiert 5 zu x, entspricht x = x + 5, Ergebnis: x = 15
x -= 3; // Subtrahiert 3 von x, entspricht x = x - 3, Ergebnis: x
= 12
x *= 2; // Multipliziert x mit 2, entspricht x = x * 2, Ergebnis:
x = 24
x /= 4; // Dividiert x durch 4, entspricht x = x / 4, Ergebnis: x
= 6
```

Vergleiche

In JavaScript können Vergleichsoperatoren dazu verwendet werden, Werte auf Gleichheit, Ungleichheit und andere Beziehungen zu überprüfen. Die Ergebnisse solcher Vergleiche werden als boolesche Werte (`true` oder `false`) zurückgegeben.

`==` **(Gleichheitsoperator):** Vergleicht zwei Werte auf Gleichheit und gibt `true` zurück, wenn sie gleich sind, `false`, wenn sie unterschiedlich sind. Es findet eine Typumwandlung statt, wenn die Werte unterschiedliche Typen haben.

```
5 == '5'; // Gibt true zurück, weil die Werte gleich sind, obwohl die Typen unterschiedlich sind
```

`===` **(Strikter Gleichheitsoperator):** Vergleicht zwei Werte auf strikte Gleichheit. Gibt `true` zurück, wenn sie gleich sind und `false`, wenn sie unterschiedlich sind, ohne Typumwandlung.

```
5 === '5'; // Gibt false zurück, weil die Typen unterschiedlich sind
```

`!=` **(Ungleichheitsoperator):** Gibt `true` zurück, wenn die Werte ungleich sind, sonst `false`. Es findet eine Typumwandlung statt, wenn die Werte unterschiedliche Typen haben.

```
"5" != 6; // Gibt true zurück
```

`!==` **(Strikter Ungleichheitsoperator):** Gibt `true` zurück, wenn die Werte ungleich sind, ohne Typumwandlung.

```
"5" !== 5; // Gibt true zurück
```

`<`, `<=`, `>`, `>=` **Relationale Vergleiche:** Diese Operatoren vergleichen zwei Werte und prüfen, ob der eine kleiner, kleiner oder gleich, größer oder größer oder gleich als der andere ist.

```
5 < 10; // Gibt true zurück
5 <= 5; // Gibt true zurück
10 > 5; // Gibt true zurück
10 >= 10; // Gibt true zurück
```

1.4.2. Arithmetische, bitwise, logische und bedingte (ternär) Operatoren

In der Programmierung sind Operatoren Symbole, die verwendet werden, um bestimmte Berechnungen oder Operationen durchzuführen. JavaScript bietet eine breite Palette von Operatoren, die in die Kategorien arithmetisch, bitweise, logisch und bedingt (ternär) unterteilt werden können. Diese Operatoren ermöglichen es den Entwicklern, komplexere Berechnungen und Logik in ihren Skripten zu implementieren. Die Verwendung dieser Operatoren reicht von einfachen mathematischen Berechnungen bis

hin zu komplizierten logischen Verknüpfungen und Bitmanipulationen. Wir wollen uns in diesem Abschnitt die vier Typen von Operatoren im Detail mit Beispielen genau ansehen, um einen klaren Überblick über ihre Funktion und Anwendung in JavaScript zu bekommen.

Arithmetische Operatoren

Diese Operatoren führen grundlegende mathematische Berechnungen aus:

Addition (`+`): Addiert zwei Werte.

```
5 + 3; // Gibt 8 zurück
```

Subtraktion (`-`): Subtrahiert den rechten Wert vom linken Wert.

```
5 - 3; // Gibt 2 zurück
```

Multiplikation (`*`): Multipliziert zwei Werte.

```
5 * 3; // Gibt 15 zurück
```

Division (`/`): Dividiert den linken Wert durch den rechten Wert.

```
5 / 2; // Gibt 2.5 zurück
```

Bitweise Operatoren

Bitweise Operatoren arbeiten auf der Ebene der einzelnen Bits in einer Zahl. Sie können verwendet werden, um komplexe Manipulationen auf der binären Ebene durchzuführen, was in bestimmten Anwendungsfällen, wie beispielsweise in der Low-Level-Programmierung oder bei der Optimierung von Code, sehr nützlich sein kann. Bitweise Operatoren sind leistungsstarke Werkzeuge, die zwar nicht häufig im alltäglichen JavaScript-Code verwendet werden, aber dennoch wichtige Möglichkeiten für bestimmte spezialisierte Anwendungsfälle bieten. Es ist ratsam, diese Operatoren mit Vorsicht zu verwenden und die Auswirkungen auf der Bit-Ebene zu verstehen, um unerwartete Ergebnisse zu vermeiden.

Bitweise UND (`&`): Vergleicht die Bits zweier Zahlen und gibt eine neue Zahl zurück, bei der jedes Bit auf 1 gesetzt ist, wenn die entsprechenden Bits in beiden Eingangszahlen 1 sind.

```
let result = 5 & 3; // result wird 1 sein
```

Bitweise ODER (`|`): Vergleicht die Bits zweier Zahlen und gibt eine neue Zahl zurück, bei der jedes Bit auf 1 gesetzt ist, wenn mindestens eines der entsprechenden Bits in den Eingangszahlen 1 ist.

```
let result = 5 | 3; // result wird 7 sein
```

Bitweise XOR (`^`): Vergleicht die Bits zweier Zahlen und gibt eine neue Zahl zurück, bei der jedes Bit auf 1 gesetzt ist, wenn genau eines der entsprechenden Bits in den Eingangszahlen 1 ist.

```
let result = 5 ^ 3; // result wird 6 sein
```

Bitweise NICHT (`~`): Kehrt die Bits einer Zahl um.

```
let result = ~5; // result wird -6 sein
```

Linksverschiebung (`<<`): Verschiebt die Bits einer Zahl nach links um die angegebene Anzahl von Stellen.

```
let result = 5 << 2; // result wird 20 sein
```

Rechtsverschiebung (`>>`): Verschiebt die Bits einer Zahl nach rechts um die angegebene Anzahl von Stellen.

```
let result = 5 >> 1; // result wird 2 sein
```

Ungesicherte Rechtsverschiebung (`>>>`): Verschiebt die Bits einer Zahl nach rechts um die angegebene Anzahl von Stellen, wobei führende Nullen eingefügt werden.

```
let result = -5 >>> 1; // result wird 2147483645 sein
```

Logische Operatoren

Logische Operatoren werden in JavaScript verwendet, um Werte auf Basis ihrer Wahrheitswerte zu vergleichen oder zu kombinieren. Sie sind besonders nützlich, um komplexere Bedingungen in Kontrollstrukturen wie if-Anweisungen zu erstellen. Die folgenden logischen Operatoren stehen zur Verfügung:

Logisches UND (`&&`): Gibt `true` zurück, wenn beide Operanden wahr sind und `false` sonst.

```
let result = true && false; // result wird false sein
```

Logisches ODER (`||`): Gibt `true` zurück, wenn mindestens einer der Operanden wahr ist und `false` sonst.

```
let result = true || false; // result wird true sein
```

Logisches NICHT (`!`): Kehrt den Wahrheitswert des Operanden um.

```
let result = !true; // result wird false sein
```

Diese Operatoren arbeiten mit den Wahrheitswerten `true` und `false`, aber sie können auch mit anderen Werten verwendet werden, die als `truthy` oder `falsy` betrachtet werden. In JavaScript wird fast alles als `truthy` betrachtet, außer den folgenden `falsy` Werten: `false`, 0, '' (leerer String), `null` `undefined` und `NaN`.

Diese Beispiele zeigen, wie diese Operatoren mit `truthy` und `falsy` Werten funktionieren:

```
let result1 = 'text' && 123; // result1 wird 123 sein
let result2 = 0 || 'text'; // result2 wird 'text' sein
let result3 = !null; // result3 wird true sein
```

Bedingter (Ternärer) Operator

Der bedingte (ternäre) Operator ist ein spezieller Operator in JavaScript, der drei Operanden nimmt. Dieser Operator ist sehr nützlich, um kurze und klare Zuweisungen zu schreiben, die auf einer Bedingung basieren und kann dabei helfen, den Code sauberer und kompakter zu machen. Er wird oft als eine verkürzte Form einer if-else-Anweisung verwendet. Die Syntax dafür ist:

```
bedingung ? ausdruckWennWahr : ausdruckWennFalsch
```

Der Operator überprüft die Bedingung (normalerweise eine Vergleichsoperation) und wertet den ersten Ausdruck (ausdruckWennWahr) aus, wenn die Bedingung wahr ist, oder den zweiten Ausdruck (ausdruckWennFalsch), wenn die Bedingung falsch ist.

Ein Beispiel dafür könnte wie folgt aussehen:

```
let age = 18;
let status = age >= 18 ? 'Erwachsener' : 'Minderjähriger';
console.log(status); // Ausgabe: "Erwachsener"
```

In diesem Beispiel überprüft der ternäre Operator, ob das Alter 18 oder älter ist. Wenn dies der Fall ist, wird der String 'Erwachsener' der Variablen `status` zugewiesen, andernfalls wird der String 'Minderjähriger' zugewiesen.

1.5. Funktionen

In der Welt der Programmierung sind Funktionen ein bisschen wie kleine Maschinen oder Werkzeuge, die ihr euch vorstellen könnt, um spezifische Aufgaben zu erledigen. Sie gehören zu den Grundbausteinen, die euch helfen, Code zu schreiben, der leicht zu verstehen, zu organisieren und wiederzuverwenden ist. Funktionen tragen dazu bei,

euren Code sauber und effizient zu gestalten. Stellt euch vor, ihr hättet eine Maschine, die jedes Mal, wenn ihr einen Knopf drückt, einen Apfel in Apfelsaft verwandelt. Ihr müsstet nicht jedes Mal verstehen, wie die Maschine funktioniert; ihr müsstet nur wissen, wie man den Knopf drückt. Funktionen in der Programmierung funktionieren ähnlich. Sie nehmen Eingaben (Befehle oder Daten), führen eine bestimmte Aufgabe aus und geben dann etwas zurück.

🧠 **Warum und wann brauchen wir Funktionen?**

- **Modularisierung:** Funktionen ermöglichen es euch, den Code in kleinere, handhabbare Teile zu unterteilen.
- **Wiederverwendung:** Wenn ihr einen Codeblock mehrmals benötigt, könnt ihr ihn in eine Funktion packen und diese immer wieder aufrufen.
- **Lesbarkeit:** Funktionen können Namen haben, die beschreiben, was sie tun. Das macht den Code leichter zu verstehen.
- **Fehlerbehandlung:** Durch die Organisation des Codes in Funktionen ist es einfacher, Fehler zu finden und zu beheben

1.5.1. Deklarieren, definieren und aufrufen von Funktionen

Die Verwendung von Funktionen in JavaScript beginnt mit der Deklaration einer Funktion. Hier legt ihr den Namen der Funktion fest und definiert, welche Aktionen sie ausführen soll. Die Definition einer Funktion umfasst den eigentlichen Codeblock, der die Aufgaben der Funktion erfüllt. Sobald die Funktion deklariert und definiert ist, könnt ihr sie jederzeit in eurem Programm aufrufen.

Deklaration und Definition

Das Deklarieren einer Funktion wird erreicht, indem ihr das Schlüsselwort `function`, gefolgt von einem Namen, einer Liste von Parametern und dem Codeblock verwendet, der die Aufgabe der Funktion erfüllt.

```
function gruss(name) {
  return "Hallo, " + name + "!";
}
```

In diesem Beispiel haben wir eine Funktion namens `gruss` deklariert, die einen Parameter `name` akzeptiert. Der Codeblock der Funktion nimmt den Namen und gibt einen String zurück.

Aufrufen einer Funktion

Sobald eine Funktion definiert ist, könnt ihr sie in eurem Code aufrufen, indem ihr den Namen der Funktion, gefolgt von Klammern `()` mit den gewünschten Argumenten

verwendet. Im Fall unserer `gruss` Funktion wäre das:

```
let begruessung = gruss("Jacob");
console.log(begruessung); // Ausgabe: "Hallo, Jacob!"
```

In diesem Beispiel rufen wir die Funktion `gruss()` mit dem Argument "Jacob" auf und sie gibt den String zurück, den wir dann in der Konsole ausgeben.

1.5.2. Funktions-Scope und Argumente

Wenn ihr mit Funktionen in JavaScript arbeitet, ist es wichtig, die Konzepte von Funktions-Scope und Argumenten zu verstehen. Sie helfen euch dabei, wie Variablen innerhalb von Funktionen funktionieren und wie ihr Daten zwischen verschiedenen Teilen eures Programms übertragen könnt.

Funktions-Scope

In JavaScript hat jede Funktion ihren eigenen Scope. Das bedeutet, dass Variablen, die innerhalb einer Funktion deklariert werden, nur innerhalb dieser Funktion zugänglich sind. Dies wird als Funktions-Scope bezeichnet. Schaut euch das folgende Beispiel an:

```
function meineFunktion() {
  let innerhalb = "Ich bin innerhalb der Funktion";
  console.log(innerhalb); // Ausgabe: "Ich bin innerhalb der Funkt
ion"
}

meineFunktion();
console.log(innerhalb); // Fehler, da 'innerhalb' außerhalb des Fu
nktions-Scopes nicht definiert ist
```

Der Funktions-Scope hilft euch, euren Code organisiert und fehlerfrei zu halten, indem er verhindert, dass Variablen außerhalb ihres beabsichtigten Bereichs verwendet werden.

Argumente

Argumente sind Werte, die ihr einer Funktion übergeben könnt, wenn ihr sie aufruft. Die Parameter, die ihr in der Funktionsdeklaration festlegt, legen fest, welche Argumente die Funktion akzeptieren kann. Argumente ermöglichen es euch, Funktionen flexibel und wiederverwendbar zu machen, indem ihr ihnen verschiedene Eingabewerte übergebt.

```
function addiere(zahl1, zahl2) {
  return zahl1 + zahl2;
}
```

```
let summe = addiere(5, 3);
console.log(summe); // Ausgabe: 8
```

In diesem Beispiel haben wir eine Funktion addiere, die zwei Parameter `zahl1` und `zahl2` akzeptiert. Wenn wir die Funktion aufrufen, übergeben wir die Argumente 5 und 3 und die Funktion gibt ihre Summe zurück.

1.5.3 Funktionen höherer Ordnung

In JavaScript können Funktionen als Argumente an andere Funktionen übergeben werden und Funktionen können auch als Ergebnis aus anderen Funktionen zurückgegeben werden. Wenn man eine Funktion an eine andere Funktion übergibt, nennt man dies „höhere Ordnung Funktion“ (Higher-order function). Die übergebene Funktion wird oft als „Callback-Funktion“ bezeichnet, weil sie später in der höheren Ordnung Funktion zurückgerufen wird.

Funktionen als Argumente übergeben

```
function greet(name) {
  console.log('Hallo ' + name);
}

function processPerson(greetingFunction, name) {
  greetingFunction(name);
}

processPerson(greet, 'Jacob');
// Ausgabe: Hallo Jacob
```

In diesem Beispiel wird die `greet` Funktion als Argument an die `processPerson` Funktion übergeben, die sie dann mit dem zweiten Argument „name“ aufruft.

Funktionen als Ergebnis zurückgeben

```
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

let doubler = multiplier(2);
console.log(doubler(5));
// Ausgabe: 10
```

Hier gibt die `multiplier` Funktion eine neue Funktion zurück, die eine Nummer nimmt und sie mit dem `factor` multipliziert, der der `multiplier` Funktion übergeben wurde.

1.6. Arbeit mit Zahlen und Daten

In JavaScript ist die Arbeit mit Zahlen und Daten ein zentraler Bestandteil vieler Programme. Ihr könnt nicht nur grundlegende arithmetische Operationen durchführen, sondern auch komplexere mathematische Funktionen nutzen und mit Daten und Uhrzeiten arbeiten.

1.6.1 Math

JavaScript bietet das Math-Objekt, das eine Sammlung von Eigenschaften und Methoden für mathematische Konstanten und Funktionen enthält. Damit könnt ihr eine breite Palette von mathematischen Aufgaben erledigen, wie z. B. das Runden von Zahlen oder die Berechnung von Sinus und Kosinus.

Runden von Zahlen: Ihr könnt Zahlen auf verschiedene Arten runden.

```
let aufgerundet = Math.ceil(5.2); // 6
let abgerundet = Math.floor(5.7); // 5
let gerundet = Math.round(5.5); // 6
```

Konstanten: Das Math-Objekt enthält auch wichtige mathematische Konstanten.

```
let pi = Math.PI; // 3.141592653589793
let e = Math.E; // 2.718281828459045
```

Trigonometrische Funktionen: Für trigonometrische Berechnungen stellt Math Funktionen wie Sinus, Kosinus und Tangens zur Verfügung.

```
let sinus = Math.sin(Math.PI / 2); // 1
let kosinus = Math.cos(Math.PI); // -1
```

Zufallszahlen: Das Generieren von Zufallszahlen ist ebenfalls möglich.

```
let zufallszahl = Math.random();
// Gibt eine Zufallszahl zwischen 0 (einschließlich) und 1 (aussch
ließlich) zurück
```

1.6.2 Date

Das Date-Objekt in JavaScript ist ein leistungsfähiges Hilfsmittel zur Handhabung von Datum und Zeit. Hiermit könnt ihr Daten erstellen, bearbeiten und abrufen, was besonders nützlich ist, wenn ihr mit Zeitstempeln in euren Anwendungen oder

Webseiten arbeitet. Werfen wir einen genaueren Blick auf einige der Möglichkeiten, die das Date-Objekt bietet.

Ein neues Datum erstellen: Das Date-Objekt kann auf verschiedene Arten instanziiert werden.

```
let heute = new Date(); // Aktuelles Datum und Uhrzeit
let bestimmtesDatum = new Date('2023-08-16'); // Ein bestimmtes Da
tum
let datumMitZeit = new Date(2023, 7, 16, 14, 30); // Jahr, Monat
(0-basiert), Tag, Stunden, Minuten
```

Datum abrufen und setzen: Das Date-Objekt stellt verschiedene Methoden zur Verfügung, um Teile eines Datums abzurufen oder zu setzen.

```
let jahr = heute.getFullYear(); // Gibt das Jahr zurück
let monat = heute.getMonth(); // Gibt den Monat zurück (0-basiert,
d. h. Januar = 0)
heute.setDate(20); // Setzt den Tag des Monats
```

Zeitdifferenz berechnen: Mit dem Date-Objekt könnt ihr auch die Zeitdifferenz zwischen zwei Daten berechnen.

```
let start = new Date();
// Irgendeine Aktion, die Zeit in Anspruch nimmt
let ende = new Date();
let differenz = ende - start; // Differenz in Millisekunden
```

Formatierung: Das Date-Objekt kann in verschiedene Formate konvertiert werden, um es menschenlesbar zu machen.

```
let datumAlsString = heute.toString(); // Gibt das Datum als lesba
ren String zurück
let datumAlsUTCString = heute.toUTCString(); // Gibt das Datum als
UTC-String zurück
```

Zeitzone: Das Date-Objekt berücksichtigt auch die Zeitzone des Benutzers. Ihr könnt die lokale Zeit oder die koordinierte Weltzeit (UTC) abrufen und bearbeiten.

```
let lokaleZeit = heute.toLocaleTimeString(); // Gibt die lokale Ze
it zurück
let utcZeit = heute.getUTCHours(); // Gibt die UTC-Stunde zurück
```

1.7. Formatierung und einfache Auswertung von Zeichenketten

In der modernen Web-Entwicklung spielt die Behandlung von Textdaten eine zentrale Rolle. JavaScript bietet eine Reihe von Möglichkeiten, um Zeichenketten (Strings) zu formatieren, zu analysieren und zu manipulieren. Nachfolgend werden einige dieser Techniken vorgestellt.

1.7.1. String

In JavaScript ist ein String eine Datenstruktur, die eine Sequenz von Zeichen darstellt. Strings können durch einfache Anführungszeichen ('), doppelte Anführungszeichen (") oder Backticks (`) eingeschlossen werden. Die Verwendung von Backticks ermöglicht die Erstellung von Template-Literalen, die es erlauben, Ausdrücke direkt im String einzufügen.

String-Literale

Ein String-Literal ist eine Sequenz von Zeichen, die in Anführungszeichen eingeschlossen ist.

```
let str1 = "Das ist ein String";
let str2 = 'Das ist auch ein String';
let str3 = `Das ist ein Template-Literal`;
```

String-Methoden

Es gibt zahlreiche Methoden, die ihr mit Strings verwenden könnt, um sie zu manipulieren und zu analysieren. Strings in JavaScript sind unveränderlich, was bedeutet, dass sie nach ihrer Erstellung nicht verändert werden können. Alle Methoden, die eine Änderung vornehmen, geben einen neuen String zurück, anstatt den ursprünglichen zu ändern. Schauen wir uns einmal einige wichtige an.

`length`: Gibt die Länge eines Strings zurück.

```
let len = str1.length; // 18
```

`toUpperCase()` **und** `toLowerCase()`: Konvertiert den String in Groß- bzw. Kleinbuchstaben.

```
let upper = str1.toUpperCase();
let lower = str1.toLowerCase();
```

`indexOf()` **und** `lastIndexOf()`: Findet die Position eines Teils des Strings.

```
let position = str1.indexOf('ein'); // 8
```

`substring()` **und** `slice()`**:** Schneidet einen Teil des Strings heraus.

```
let teil = str1.substring(8, 11); // 'ein'
```

`replace()`**:** Ersetzt einen Teil des Strings.

```
let ersetzt = str1.replace('ein', 'kein'); // 'Das ist kein Strin
g'
```

`split()`**:** Teilt den String an einem bestimmten Zeichen und gibt ein Array zurück.

```
let teile = str1.split(' '); // ['Das', 'ist', 'ein', 'String']
```

`trim()`**:** Entfernt Leerzeichen am Anfang und am Ende des Strings.

```
let sauber = "   Extra Leerzeichen    ".trim(); // 'Extra Leerzeic
hen'
```

1.7.2. Template-Literale

Template-Literale sind eine erweiterte Art von String-Literalen in JavaScript, die es ermöglichen, Ausdrücke in den String einzufügen. Sie werden mit Backticks (`) anstelle von einfachen oder doppelten Anführungszeichen definiert. Mit Template-Literalen könnt ihr Variablen und Ausdrücke direkt in einen String einbetten, ohne die Konkatenation mit dem Plus-Operator (+) verwenden zu müssen. Dies kann den Code oft übersichtlicher machen.

Grundlegende Verwendung

Hier ist ein einfaches Beispiel für die Verwendung eines Template-Literals:

```
let name = "Jacob";
let begrüßung = `Hallo ${name}! Willkommen zurück.`;
console.log(begrüßung); // "Hallo Jacob! Willkommen zurück."
```

Die Syntax `${...}` innerhalb des Template-Literals ermöglicht es, einen Ausdruck einzubetten, der ausgewertet wird und dessen Ergebnis im resultierenden String erscheint.

Mehrzeilige Strings

Ein weiterer Vorteil von Template-Literalen ist die Unterstützung von mehrzeiligen Strings. Ihr könnt einfach einen Zeilenumbruch in das Template-Literal einfügen, ohne Escape-Sequenzen zu verwenden.

```
let mehrzeilig = `Dies ist ein mehrzeiliger String.
Es geht auf die nächste Zeile ohne Probleme.`;
console.log(mehrzeilig);
```

Verschachtelte Template-Literale

Ihr könnt auch Template-Literale innerhalb von Template-Literale verwenden, um komplexe Strukturen zu erstellen:

```
let item = "Apfel";
let anzahl = 3;
let nachricht = `Sie haben ${anzahl} ${anzahl === 1 ? `${item}` :
`${item}s`} in Ihrem Warenkorb.`;
console.log(nachricht); // "Sie haben 3 Äpfel in Ihrem Warenkorb."
```

1.7.3. I18n

I18n steht für "Internationalisierung" und ist ein wichtiger Aspekt der modernen Programmierung, insbesondere in einer globalisierten Welt. Es geht darum, Anwendungen so zu gestalten, dass sie leicht an verschiedene Sprachen, Regionen und Kulturen angepasst werden können. In JavaScript gibt es verschiedene Methoden und Bibliotheken, die euch bei der Internationalisierung unterstützen können. Dies ist essentiell, um sicherzustellen, dass eure Anwendung in verschiedenen Märkten und von Nutzern aus unterschiedlichen Kulturkreisen verwendet werden kann.

Grundlegende Verwendung

JavaScript bietet das `Intl`-Objekt, das Schnittstellen für String-Vergleich, Number-Formatierung und Datums- und Zeitformatierung in verschiedenen Sprachen enthält.

Datums- und Zeitformatierung: Hier ist ein Beispiel, wie ihr das `Intl.DateTimeFormat`-Objekt verwenden könnt.

```
let datum = new Date();
let deutschesFormat = new Intl.DateTimeFormat('de-DE').format(datu
m);
let amerikanischesFormat = new Intl.DateTimeFormat('en-US').format
(datum);

console.log(deutschesFormat); // z.B. "31.12.2023"
console.log(amerikanischesFormat); // z.B. "12/31/2023"
```

Zahlenformatierung: Ebenso könnt ihr das `Intl.NumberFormat`-Objekt verwenden, um Zahlen entsprechend der lokalen Konventionen zu formatieren.

```
let zahl = 1234567.89;
let deutschesFormat = new Intl.NumberFormat('de-DE').format(zahl);
let amerikanischesFormat = new Intl.NumberFormat('en-US').format(z
ahl);

console.log(deutschesFormat); // "1.234.567,89"
console.log(amerikanischesFormat); // "1,234,567.89"
```

Die Unterstützung für verschiedene Sprachen und Kulturen kann eure Anwendung für ein breiteres Publikum zugänglich machen. Die Verwendung von I18n-Praktiken und -Tools in JavaScript ermöglicht es, diese globale Reichweite auf eine standardisierte und effiziente Weise zu erreichen. Es hilft euch, eure Software benutzerfreundlich zu gestalten, indem sie in der Sprache und den Formaten präsentiert wird, die eure Nutzer kennen und verstehen.

1.7.4. RegEx

RegEx, oder reguläre Ausdrücke, sind ein mächtiges Werkzeug in der Programmierung, mit dem ihr Textmuster suchen, abgleichen, ersetzen und manipulieren könnt. In JavaScript könnt ihr RegEx verwenden, um mit Zeichenketten (Strings) auf flexible und effiziente Weise zu arbeiten. Ein regulärer Ausdruck besteht aus einer Kombination von Zeichen, die ein Muster bilden. Dieses Muster wird verwendet, um eine Übereinstimmung in einem String zu finden oder zu prüfen. Ihr könnt einen regulären Ausdruck in JavaScript auf zwei Arten erstellen:

- **Literal-Syntax:** `/muster/flags`
- **Konstruktor-Syntax:** `new RegExp('muster', 'flags')`

Übereinstimmung finden: In diesem Beispiel verwenden wir den regulären Ausdruck `/a/gi` zum Finden aller Vorkommen des Buchstabens "a" in dem gegebenen String "Ananas und Bananen".

```
let regex = /a/gi;
let string = "Ananas und Bananen";
let result = string.match(regex);
console.log(result); // Ausgabe: ["A", "a", "a", "a"]
```

- `/a/`: das Muster, das den Buchstaben "a" repräsentiert
- `g`: steht für "global", sodass alle Übereinstimmungen im gesamten Text gefunden werden
- `i`: steht für "ignoriere die Groß-/Kleinschreibung", sodass sowohl "A" als auch "a" gefunden werden

Die Methode `match()` gibt ein Array mit allen gefundenen Übereinstimmungen zurück.

Ersetzen: Hier verwenden wir den Ausdruck `/Katze/g` zum Ersetzen aller Vorkommen von "Katze" durch "Hund" im gegebenen String.

```
let regex = /Katze/g;
let string = "Katze im Sack. Katze auf dem Baum.";
let result = string.replace(regex, 'Hund');
console.log(result); // Ausgabe: "Hund im Sack. Hund auf dem Bau
m."
```

- `/Katze/`: das Muster, das das Wort "Katze" repräsentiert
- `g`: steht für "global", sodass alle Übereinstimmungen im gesamten Text ersetzt werden

Die Methode `replace()` nimmt den regulären Ausdruck und den Ersatzstring als Argumente und gibt den neuen String mit den Änderungen zurück.

Überprüfen: Dieses Beispiel verwendet den regulären Ausdruck `\\S+@\\S+\\.\\S+` zum Überprüfen, ob ein String eine E-Mail-Adresse enthält.

```
let regex = /\\S+@\\S+\\.\\S+/;
let string = "Email me at example@example.com";
let result = regex.test(string);
console.log(result); // Ausgabe: true
```

- `\\S+`: steht für "ein oder mehrere Nicht-Leerzeichen-Zeichen"
- `@`: steht für das At-Zeichen in der E-Mail-Adresse
- `\\.`: steht für den Punkt in der E-Mail-Adresse (der Backslash ist notwendig, da der Punkt in RegEx eine besondere Bedeutung hat und hier als wörtliches Zeichen behandelt werden soll)

Die Methode `test()` gibt `true` zurück, wenn das Muster im String gefunden wird und `false`, wenn nicht.

1.8. Datenstrukturen

In der Programmierung ist die effiziente Handhabung und Organisation von Daten von entscheidender Bedeutung. Die Verwaltung großer Mengen von Informationen erfordert Strukturen, die sowohl flexibel als auch leistungsfähig sind. In JavaScript gibt es spezifische Datenstrukturen, die zur Sammlung und Verwaltung von Daten in verschiedenen Formaten und mit unterschiedlichen Anforderungen eingesetzt werden. Dieser Abschnitt wird sich mit den Hauptstrukturen in JavaScript befassen, darunter Arrays, Maps und Sets. Jede dieser Strukturen hat ihre eigenen Eigenschaften und

Verwendungszwecke, von der Speicherung geordneter Listen bis hin zur Verwaltung eindeutiger Schlüssel-Wert-Paare. Indem ihr die Grundlagen dieser Datenstrukturen verstehen und meistern lernt, werdet ihr in der Lage sein, leistungsstarke und effiziente Lösungen für eine Vielzahl von Problemen zu entwickeln. Die folgenden Abschnitte werden euch durch die Besonderheiten von Arrays, Maps und Sets führen, einschließlich ihrer Deklaration, Verwendung und der Methoden, die sie bieten, um mit den darin gespeicherten Daten zu interagieren.

🧠 **Was ist der Unterschied zwischen Arrays, Maps und Sets?**

- **Arrays** werden für geordnete Listen mit möglichen Duplikaten und numerischen Indizes verwendet.
- **Sets** sind für geordnete Sammlungen ohne Duplikate und ohne Schlüssel-Wert-Zugriff gedacht.
- **Maps** eignen sich für geordnete Schlüssel-Wert-Paare, wobei die Schlüssel einzigartig sein müssen und von beliebigem Typ sein können.

1.8.1. Arrays

Arrays sind eine der grundlegendsten und vielseitigsten Datenstrukturen in JavaScript. Sie ermöglichen es euch, eine geordnete Sammlung von Elementen zu speichern, auf die über einen numerischen Index zugegriffen werden kann. Arrays sind besonders nützlich, wenn ihr eine Liste von Dingen wie Zahlen, Strings oder sogar anderen Arrays und Objekten speichern möchtet.

- **Reihenfolge:** Arrays speichern Elemente in einer geordneten Liste.
- **Zugriff:** Der Zugriff auf Elemente erfolgt über einen numerischen Index, der bei 0 beginnt.
- **Duplikate:** Erlaubt Duplikate.
- **Schlüsseltyp:** Die Indizes sind immer numerisch.
- **Verwendung:** Gut geeignet, wenn die Reihenfolge der Elemente wichtig ist oder wenn die Daten als Liste strukturiert sind.

Erstellen eines Arrays

Ihr könnt ein Array mit der Array-Literal-Notation erstellen, indem ihr die Elemente in eckigen Klammern `[]` auflistet.

```
let fruechte = ['Apfel', 'Birne', 'Kirsche'];
```

Zugriff auf Elemente

Ihr könnt auf die einzelnen Elemente eines Arrays zugreifen, indem ihr den Index des gewünschten Elements in eckigen Klammern nach dem Array-Namen angebt.

```
let ersteFrucht = fruechte[0]; // Gibt 'Apfel' zurück
```

Ändern von Elementen

Ihr könnt auch die Werte in einem Array ändern, indem ihr einen Index verwendet.

```
fruechte[2] = 'Banane'; // Ändert das dritte Element zu 'Banane'
```

Länge eines Arrays

Die Eigenschaft `length()` gibt euch die Anzahl der Elemente in einem Array.

```
let anzahl = fruechte.length; // Gibt 3 zurück
```

Hinzufügen und Entfernen von Elementen

Mit Methoden wie `push()`, `pop()`, `shift()` und `unshift()` könnt ihr Elemente am Anfang oder Ende des Arrays hinzufügen oder entfernen.

```
fruechte.push('Traube'); // Fügt 'Traube' am Ende hinzu
fruechte.shift(); // Entfernt das erste Element
```

1.8.2. Map

Die Map-Objekte in JavaScript sind eine Sammlung von Schlüssel-Wert-Paaren. Im Gegensatz zu normalen Objekten, bei denen die Schlüssel Strings sein müssen, könnt ihr in einer Map nahezu jeden Wert als Schlüssel verwenden. Das ermöglicht eine höhere Flexibilität bei der Zuordnung von Werten.

- **Reihenfolge:** Maps speichern Schlüssel-Wert-Paare in der Reihenfolge ihres Einfügens.
- **Zugriff:** Der Zugriff auf Elemente erfolgt über einen eindeutigen Schlüssel, der sowohl numerisch als auch nicht numerisch sein kann.
- **Duplikate:** Erlaubt keine doppelten Schlüssel; die Werte können jedoch dupliziert werden.
- **Schlüsseltyp:** Kann beliebige Datentypen als Schlüssel haben, einschließlich Objekte und Funktionen.
- **Verwendung:** Ideal, wenn eine Zuordnung von Schlüsseln zu Werten benötigt wird, insbesondere wenn die Schlüssel nicht numerisch sind.

Erstellen einer Map

Ihr könnt eine Map mit `new` und dem Map-Konstruktor erstellen.

```
let meineMap = new Map();
```

Hinzufügen von Schlüssel-Wert-Paaren

Ihr könnt Schlüssel-Wert-Paare mit der Methode `set()` hinzufügen.

```
meineMap.set('Name', Jacob);
meineMap.set('Alter', 26);
```

Zugriff auf Werte

Um auf einen Wert in der Map zuzugreifen, verwendet ihr die Methode `get()`, indem ihr den Schlüssel übergebt.

```
let name = meineMap.get('Name'); // Gibt 'Jacob' zurück
```

Überprüfung, ob ein Schlüssel vorhanden ist

Ihr könnt überprüfen, ob ein bestimmter Schlüssel in der Map vorhanden ist, mit der Methode `has()`.

```
let hatAlter = meineMap.has('Alter'); // Gibt true zurück
```

Entfernen von Schlüssel-Wert-Paaren

Ihr könnt ein Schlüssel-Wert-Paar mit der Methode `delete()` entfernen.

```
meineMap.delete('Alter'); // Entfernt den Schlüssel 'Alter' und se
inen Wert
```

Größe der Map

Die Eigenschaft `size()` gibt euch die Anzahl der Schlüssel-Wert-Paare in der Map.

```
let groesse = meineMap.size; // Gibt 1 zurück, nachdem 'Alter' ent
fernt wurde
```

1.8.3. Set

Ein Set in JavaScript ist eine Sammlung von Werten, wobei jeder Wert nur einmal vorkommen darf. Das bedeutet, dass ein Set keine Duplikate enthält. Dies kann besonders nützlich sein, wenn ihr eine Liste von Werten erstellen wollt, bei der jeder Wert eindeutig sein muss.

- **Reihenfolge:** Sets speichern Elemente ebenfalls in einer geordneten Weise, aber die Reihenfolge kann für die Programmlogik irrelevant sein.
- **Zugriff:** Kein direkter Zugriff über einen Index wie bei Arrays. Stattdessen bietet das Set Methoden wie `has()` zum Prüfen von Elementen.
- **Duplikate:** Erlaubt keine Duplikate; jedes Element ist eindeutig.
- **Schlüsseltyp:** Keine Schlüssel; das Set speichert nur Werte.
- **Verwendung:** Gut geeignet, wenn Duplikate vermieden werden müssen und der Schlüssel-Wert-Zugriff nicht erforderlich ist.

Erstellen eines Sets

Ein Set wird mit `new` und dem Set-Konstruktor erstellt.

```
let meinSet = new Set();
```

Ihr könnt auch ein Array oder ein anderes iterierbares Objekt als Argument übergeben, um ein Set mit diesen Werten zu initialisieren.

```
let zahlenSet = new Set([1, 2, 3, 4, 5]);
```

Hinzufügen von Werten

Werte könnt ihr mit der Methode `add()` hinzufügen.

```
meinSet.add('Apfel');
meinSet.add('Birne');
```

Wenn ihr denselben Wert noch einmal hinzufügt, wird nichts passieren, da die Werte in einem Set eindeutig sein müssen.

```
meinSet.add('Apfel'); // 'Apfel' wird nur einmal im Set gespeicher
t
```

Überprüfung, ob ein Wert vorhanden ist

Ihr könnt überprüfen, ob ein Wert im Set vorhanden ist, mit der Methode `has()`.

```
let hatApfel = meinSet.has('Apfel'); // Gibt true zurück
```

Entfernen von Werten

Einen Wert könnt ihr mit der Methode `delete()` entfernen.

```
meinSet.delete('Birne'); // Entfernt 'Birne' aus dem Set
```

Größe des Sets

Die Eigenschaft `size()` gibt euch die Anzahl der Werte im Set.

```
let groesse = meinSet.size; // Gibt 1 zurück, nachdem 'Birne' entf
ernt wurde
```

Durchlaufen eines Sets

Ihr könnt mit der Methode `forEach()` durch ein Set iterieren und dabei eine Funktion auf jeden Wert anwenden.

```
meinSet.forEach(value => {
  console.log(value);
});
```

1.9. Quellcode-Formatierung und externe Dateien

In der Softwareentwicklung sind klare und konsistente Formatierungskonventionen entscheidend, um den Code verständlich und wartbar zu halten. Egal, ob ihr alleine arbeitet oder in einem Team, das Festlegen und Befolgen von gemeinsamen Formatierungsrichtlinien kann die Lesbarkeit des Codes erheblich verbessern und die Zusammenarbeit vereinfachen.

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."* - Martin Fowler

1.9.1. Konventionen zur Formatierung

Formatierungskonventionen beziehen sich auf die Art und Weise, wie der Code strukturiert ist, einschließlich Aspekten wie Einrückung, Zeilenumbrüche, Leerzeichen und Kommentar-Stil. In JavaScript, wie in vielen anderen Programmiersprachen, gibt es keine festen Regeln dafür, aber es gibt weit verbreitete Best Practices und Stile, die von der Gemeinschaft angenommen werden.

- **Einrückung:** Verwenden von Tabs oder Leerzeichen (üblicherweise 2 oder 4) zur Einrückung von Code-Blöcken, um die Hierarchie der Anweisungen zu verdeutlichen.
- **Zeilenumbrüche:** Verwendung von Zeilenumbrüchen zwischen Funktions- und Kontrollstrukturen, um den Code übersichtlich zu gestalten.
- **Leerzeichen:** Das Platzieren von Leerzeichen um Operatoren und nach Kommata trägt zur Klarheit des Codes bei.

- **Kommentare:** Kommentare sollten klar und informativ sein, um den Code zu erklären, ohne überflüssig zu sein.
- **Benennung:** Verwenden von klaren und aussagekräftigen Variablennamen, die dem CamelCase-Stil folgen.

Indem ihr auf eine konsistente Formatierung achtet, erleichtert ihr anderen Entwicklern (und auch euch selbst in der Zukunft) das Lesen und Verstehen eures Codes. Es zeigt auch eine Professionalität in eurer Arbeitsweise und erleichtert die Zusammenarbeit mit anderen, da jeder die gleichen Regeln befolgt.

Einrückung

Die Verwendung von Einrückungen macht die Struktur des Codes klarer. Hier ist ein Beispiel, das die Verwendung von 4 Leerzeichen zur Einrückung zeigt.

```
if (x > 10) {
    console.log('x ist größer als 10');
    if (y > 20) {
        console.log('y ist größer als 20');
    }
}
```

In diesem Beispiel sind die Einrückungen konsistent auf 4 Leerzeichen festgelegt. Die Einrückungen veranschaulichen die Verschachtelung der Bedingungen und ermöglichen es, die Struktur des Codes auf einen Blick zu erkennen. Ohne Einrückungen wäre die Hierarchie der Bedingungen weniger offensichtlich und es könnte schwieriger sein, den Code schnell zu verstehen.

Zeilenumbrüche

Zeilenumbrüche tragen dazu bei, den Code lesbar zu machen, indem sie visuelle Trennung zwischen verschiedenen Blöcken bieten.

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

function subtract(a, b) {
    return a - b;
}
```

Zeilenumbrüche trennen verschiedene Codeblöcke, wie in diesem Fall zwei verschiedene Funktionen. Durch die visuelle Trennung wird der Code übersichtlicher und die Leser können leichter erkennen, wo ein Abschnitt endet und der nächste beginnt. Es fördert die Lesbarkeit und Verständlichkeit des Codes.

Leerzeichen

Leerzeichen um Operatoren und nach Kommata machen den Code angenehmer zu lesen.

```
var sum = a + b;
var myList = [1, 2, 3, 4];
```

Leerzeichen um Operatoren und nach Kommata schaffen Luft im Code und vermeiden das Gedränge von Symbolen. In diesem Beispiel machen die Leerzeichen den Code angenehmer für das Auge und erleichtern das schnelle Scannen und Verstehen des Ausdrucks.

Kommentare

Gut platzierte Kommentare helfen, den Code zu verstehen.

```
// Diese Funktion addiert zwei Zahlen
function add(a, b) {
    return a + b; // Gibt die Summe zurück
}
```

Kommentare in diesem Beispiel erklären, was die Funktion tut, ohne überflüssig zu sein. Sie bieten Kontext und helfen jedem, der später den Code liest oder bearbeitet, zu verstehen, was der ursprüngliche Programmierer beabsichtigte.

Benennung

Die Verwendung von klaren und aussagekräftigen Variablennamen erleichtert das Verständnis des Codes.

```
var grundgehalt = 2000;
var bonusGehalt = 500;
var gesamtGehalt = grundgehalt + bonusGehalt;
```

Die Benennung in diesem Beispiel folgt dem CamelCase-Stil und verwendet aussagekräftige Namen, die sofort klar machen, was die Variablen repräsentieren. Durch das Verständnis der Bedeutung der Variablen kann der Leser den Ausdruck leichter interpretieren, ohne zusätzliche Kommentare oder Dokumentation.

1.9.2. JavaScript-Quellcode in externe Dateien auslagern

Das Auslagern von JavaScript-Quellcode in externe Dateien ist eine übliche Praxis, die dazu dient, den Code besser zu organisieren und die Wartbarkeit zu erhöhen. Es ermöglicht euch, den gleichen Code auf mehreren Seiten oder sogar in verschiedenen Projekten zu verwenden. Im Folgenden erkläre ich euch, wie ihr dies machen könnt und warum es sinnvoll sein kann.

Auslagern in externe Dateien

Um JavaScript-Code in eine externe Datei auszulagern, schreibt ihr euren JavaScript-Code in eine separate Datei mit der Erweiterung `.js`, z.B. `meinSkript.js`. Danach verknüpft ihr diese Datei mit eurem HTML-Dokument, indem ihr das `<script>`-Tag mit dem `src`-Attribut verwendet.

```
<!DOCTYPE html>
<html>
<head>
    <title>Eure Webseite</title>
    <script src="meinSkript.js"></script>
</head>
<body>
    <!-- Euer HTML-Code hier -->
</body>
</html>
```

Vorteile der Auslagerung

- **Wiederverwendbarkeit:** Durch das Auslagern des Codes in eine externe Datei könnt ihr den gleichen Code auf mehreren Seiten verwenden, ohne ihn jedes Mal kopieren zu müssen.
- **Lesbarkeit:** Das Trennen von JavaScript von HTML kann die Lesbarkeit beider Dateien verbessern, besonders wenn euer Projekt wächst.
- **Wartbarkeit:** Die Änderung des Codes an einem Ort spiegelt sich auf allen Seiten wider, die die Datei verwenden. Das erleichtert das Beheben von Fehlern und das Aktualisieren von Funktionen.
- **Caching:** Browser können externe JavaScript-Dateien zwischenspeichern, was die Ladezeiten für wiederkehrende Besucher verbessern kann.