JS-Funktionen 2

Lernziele

- Was ein `return statement` einer Funktion ist und wie man es in JavaScript-Funktionen verwendet
- Was ein `early return` ist
- Wie man Funktionen mit der `fat arrow notation` schreibt

Return Statements

Funktionen sind ein unglaublich vielseitiges und zentrales Werkzeug in den meisten Programmiersprachen. Wir haben bereits gelernt, wie man Werte mit Eingabeparametern in eine Funktion übergibt. Aber eine Funktion kann auch einen Wert zurückgeben, an die Stelle, an der sie aufgerufen wurde. Dies geschieht über ein `return statement`.

```
function add3Numbers(first, second, third) {
  const sum = first + second + third;
  return sum;
}
```

Das `return statement` beginnt mit dem Schlüsselwort `return`, gefolgt von einem Ausdruck. In diesem Fall ist der Ausdruck die Variable sum. Ihr Wert wird von der Funktion zurückgegeben und kann gespeichert werden, wenn die Funktion aufgerufen wird:

```
const firstSum = add3Numbers(1, 2, 3);
// der Rückgabewert wird in "firstSum" gespeichert, nämlich 6

const secondSum = add3Numbers(4, 123, 33);
// der Rückgabewert wird nun in "secondSum" gespeichert, nämlich 160
```

> 💡 Ein Ausdruck ist alles, was einen Wert erzeugt: eine Variable, ein fest kodierter Wert wie `true` oder `6`, eine mathematische Operation wie `2 + 3` oder sogar ein anderer Funktionsaufruf! Dieser Artikel erklärt dies ausführlicher.

Auf diese Weise können wir Berechnungen und/oder Entscheidungsprozesse auslagern und den zurückgegebenen Wert im Programm weiterverwenden.

Eine Funktion kann nur einen Ausdruckswert zurückgeben, kann jedoch mehrere return statements in Kombination mit if else-Anweisungen haben:

```
function checkInputLength(inputString) {
  if (inputString.length > 3) {
    return true;
```

```
  } else {
    return false;
  }
}
```

Early Return Statements

Sobald ein return statement in einem Funktionsaufruf erreicht wird, endet die Ausführung der Funktion. Die nachfolgende `console.log()`-Anweisung wird daher nie erreicht:

```
function testFunction() {
  return "a returned string";

  console.log("Ich werde niemals in der Konsole ausgegeben.");
}
```

Dieses Verhalten kann zu unserem Vorteil als early return statements genutzt werden. Manchmal möchten wir bestimmte Teile unseres Codes nur ausführen, wenn eine Bedingung zutrifft. Wir können dies mit einer if else-Anweisung(statement) überprüfen. Wenn mehrere Bedingungen vorliegen, wird der Code jedoch schwerer lesbar und zu verstehen:

```
function setBackgroundColor(color) {
  if (typeof color === "String") {
    if (color.startsWith("#")) {
      if (color.length >= 7) {
        document.body.style.backgroundColor = color;
      }
    }
  }
}
```

Ein alternativer Ansatz besteht darin, die Funktion mit early return statements zu beenden:

```
function setBackgroundColor(color) {
  // erste Bedingung
  if (typeof color !== "String") {
    return;
  }

  // zweite Bedingung
  if (!color.startsWith("#")) {
    return;
  }

  // dritte Bedingung
  if (color.length < 7) {
```

```
    return;
  }

  // nur wenn alle 3 Bedingungen erfüllt sind, wird die letzte Zeile des
Codes ausgeführt.
  document.body.style.backgroundColor = color;
}
```

Diese Schreibweise macht den Code besser lesbar.

💡 Hinweis: Ein return statement kann leer gelassen werden, der zurückgegebene Wert ist dann `undefined`.

Arrow Function Expressions

Neben der klassischen Funktionsdeklaration gibt es in JavaScript eine zweite Möglichkeit, Funktionen als `arrow function expressions` zu schreiben:

```
const addNumbers = (first, second) => {
  return first + second;
};
```

Die Funktion wird wie eine Variable mit dem Schlüsselwort `const` gespeichert. Die Parameter werden wie gewohnt in runden Klammern geschrieben, gefolgt von einem fat arrow `=>`. Dann wird der Funktionskörper in geschweiften Klammern geschrieben.

Implizite Return Statements

Der Vorteil von arrow functions besteht in kürzeren Notationen, wenn bestimmte Kriterien erfüllt sind:

1. Runden Klammern um die Parameter weglassen: Dies ist möglich, wenn nur ein Eingabewert vorliegt:

```
const addOne = (number) => {
  return number + 1;
};
```

2. Implizite return statements: Wenn die Funktion nur aus einem return statement besteht, können die geschweiften Klammern und das return-Schlüsselwort weggelassen werden:

```
const addNumbers = (first, second) => {
  return first + second;
};
```

kann umgeschrieben werden als:

```
const addNumbers = (first, second) => first + second;
```

> 💡 Diese Kurzschreibweise ist besonders nützlich, wenn wir in ein paar Tagen mit callback functions arbeiten. Versuche, dir diese Funktion zu merken.

> 💡 Vielleicht erinnerst du dich an die Syntax der `addEventListener`-Methode. Wir haben diese arrow functions dort bereits kennengelernt!

```
button.addEventListener('click',() => {
    ...
})
```

Ressourcen

- Statements vs Expressions by Josh Comeau

Bonusmaterial

Hier ist ein zusätzliches Material zu **callback functions**. Während dieses Wissen für die heutigen Herausforderungen nicht erforderlich ist, wird es als Referenz angeboten, falls du auf diesen Begriff während deines Lernwegs stößt.

Callback Functions

Eine callback function ist eine Funktion, die **als Argument** in eine andere Funktion übergeben wird.

Die äußere Funktion kann diese callback function zum richtigen Zeitpunkt oder mehrmals ausführen, beispielsweise:

- wenn ein Ereignis ausgelöst wird
- wenn die abgerufenen Daten auf deinem Computer angekommen sind
- für jedes Element in einem Array.

Callback functions werden verwendet, wenn das Programm selbst entscheiden muss, **wann** oder **wie oft** die Funktion ausgeführt werden soll. Wir haben bereits callback functions in **Event Listeners** verwendet:

```
button.addEventListener("click", () => {
  console.log("Innerhalb der Callback-Funktion.");
});
```

Hier ist die Struktur wie folgt:

- äußere Funktion: `addEventListener()`
- erstes Argument: `'click'`

- zweites Argument: callback function

```
() => {
  console.log("Innerhalb der Callback-Funktion.");
};
```

Dieser Funktionstyp wird als **anonyme Funktion** bezeichnet, da sie ohne Namen deklariert wird.

Named Callback Functions

Jede Funktion kann als callback function verwendet werden. Sie muss nur an eine andere Funktion übergeben werden. Du kannst eine normale Funktion deklarieren und dann den **Namen der Funktion** verwenden, um sie an eine andere Funktion zu übergeben:

```
function sayHello() {
  console.log("Hey Dude!");
}

button.addEventListener("click", sayHello);
```

> ! Beachte, dass wir die Funktion hier nicht aufrufen (wir haben `sayHello` geschrieben statt `sayHello()`). Wir übergeben die Funktion nur an den Event Listener. Die Funktion wird nur aufgerufen, wenn das Ereignis eintritt.

Parameter in Callback Functions

Eine callback function kann Parameter akzeptieren. Die Werte für die Parameter werden von der Funktion bereitgestellt, die die callback function aufruft (die "higher order function").

In diesem Beispiel kann die callback function einen Parameter akzeptieren, um Informationen über das aufgetretene Ereignis zu erhalten:

```
button.addEventListener("click", (event) => {
  console.log("Dieser Button wurde geklickt:", event.target);
});
```

Ressourcen

- MDN Callback Functions