Entscheidungen und Wiederholungen

Programme sollen auf unterschiedliche Situationen reagieren und Aufgaben wiederholt ausführen können. Dieses Kapitel zeigt dir, wie du mit Bedingungen Entscheidungen formulierst und mit Schleifen Listen, Zahlenfolgen oder wiederkehrende Abläufe verarbeitest. Dabei übst du verständlichen Programmcode.

Bedingungen mit `if` und `else`

Lernziel

Mit `if` und `else` kann dein Programm **Entscheidungen treffen**. Es prüft eine Bedingung und führt je nach Ergebnis unterschiedlichen Code aus.

Beispielsweise:

- Ist ein Nutzer angemeldet?
- Ist eine Zahl größer als 10?
- Ist das Passwort korrekt?
- Regnet es gerade?

Am Ende dieser Lektion kannst du Bedingungen formulieren und damit den Ablauf deines Programms steuern.

Die Grundidee einer Bedingung

Eine Bedingung hat immer zwei mögliche Ergebnisse:

- `true` bedeutet: Die Aussage ist wahr.
- `false` bedeutet: Die Aussage ist falsch.

Schau dir dieses Beispiel an:

```
let alter = 20;

if (alter >= 18) {
  console.log("Du bist volljährig.");
}
```

JavaScript prüft hier:

> Ist `alter` größer oder gleich 18?

Da `alter` den Wert `20` hat, ist die Bedingung wahr. Deshalb wird die Ausgabe innerhalb der geschweiften Klammern ausgeführt:

```
Du bist volljährig.
```

Aufbau einer `if`-Anweisung

Eine `if`-Anweisung besteht aus drei Teilen:

```
if (bedingung) {
  // Dieser Code läuft nur bei true.
}
```

Dabei gilt:

1. Nach `if` folgt die Bedingung in runden Klammern.
2. Der auszuführende Code steht in geschweiften Klammern.
3. Der Codeblock wird nur ausgeführt, wenn die Bedingung `true` ergibt.

Ein weiteres Beispiel:

```
let punktzahl = 85;


if (punktzahl >= 50) {
  console.log("Du hast bestanden.");
}
```

Die Punktzahl ist mindestens 50. Die Bedingung ist also wahr und die Meldung erscheint in der Konsole.

Was passiert bei `false`?

Wenn eine Bedingung falsch ist, überspringt JavaScript den Codeblock.

```
let temperatur = 12;

if (temperatur > 20) {
  console.log("Es ist warm.");
}

console.log("Die Temperatur wurde geprüft.");
```

Die Ausgabe lautet:

```
Die Temperatur wurde geprüft.
```

`temperatur > 20` ist falsch. Deshalb erscheint „Es ist warm.“ nicht. Der Code *nach* dem `if`-Block wird jedoch ganz normal ausgeführt.

Alternative mit `else`

Mit `else` legst du fest, was passieren soll, wenn die Bedingung falsch ist.

```
let alter = 16;

if (alter >= 18) {
  console.log("Du darfst wählen.");
} else {
  console.log("Du bist noch nicht wahlberechtigt.");
}
```

Da `alter` kleiner als 18 ist, lautet die Ausgabe:

```
Du bist noch nicht wahlberechtigt.
```

Der Aufbau sieht allgemein so aus:

```
if (bedingung) {
  // Code für true
} else {
  // Code für false
}
```

Bei einer `if`-`else`-Anweisung wird **genau einer** der beiden Codeblöcke ausgeführt.

Vergleiche in Bedingungen

Für Bedingungen nutzt du häufig Vergleichsoperatoren.

Operator	Bedeutung	Beispiel
`===`	genau gleich	`farbe === "blau"`
`!==`	nicht gleich	`farbe !== "blau"`
`>`	größer als	`zahl > 10`
`<`	kleiner als	`zahl < 10`
`>=`	größer oder gleich	`alter >= 18`
`<=`	kleiner oder gleich	`punkte <= 100`

Beispiel mit einem Textwert:

```
let lieblingsfarbe = "grün";

if (lieblingsfarbe === "grün") {
  console.log("Grün ist auch eine schöne Farbe.");
} else {
  console.log("Du magst eine andere Farbe.");
}
```

> **Wichtig:** Verwende für Vergleiche normalerweise `===` statt `==`.
> `===` prüft sowohl den Wert als auch den Datentyp und vermeidet dadurch viele unerwartete Fehler.

Der Unterschied zwischen `=` und `===`

Diese beiden Zeichenfolgen sehen ähnlich aus, haben aber völlig unterschiedliche Aufgaben:

```
let name = "Mira";
```

Ein einzelnes Gleichheitszeichen `=` **weist einen Wert zu**.

```
name = "Sam";
```

Nun enthält die Variable `name` den Wert `"Sam"`.

Drei Gleichheitszeichen `===` **vergleichen Werte**:

```
if (name === "Sam") {
  console.log("Hallo Sam!");
}
```

Ein häufiger Anfängerfehler ist dieser:

```
if (name = "Sam") {
  console.log("Hallo!");
}
```

Das ist kein Vergleich. JavaScript versucht hier, `"Sam"` erneut zuzuweisen. Verwende in Bedingungen daher immer `===`, wenn du auf Gleichheit prüfen möchtest.

Mit Boolean-Werten arbeiten

Eine Variable vom Typ Boolean enthält bereits `true` oder `false`. Du kannst sie direkt in einer Bedingung verwenden.

```
let istEingeloggt = true;

if (istEingeloggt) {
  console.log("Willkommen zurück!");
} else {
  console.log("Bitte melde dich an.");
}
```

Du musst nicht schreiben:

```
if (istEingeloggt === true) {
  console.log("Willkommen zurück!");
}
```

Das funktioniert zwar, ist aber unnötig lang. Die kürzere Variante ist üblicher und besser lesbar.

Eine Bedingung verneinen

Mit `!` drehst du einen Boolean-Wert um:

```
let istEingeloggt = false;

if (!istEingeloggt) {
  console.log("Bitte melde dich an.");
}
```

`!istEingeloggt` bedeutet: „ist **nicht** eingeloggt“.

Mehrere Anweisungen in einem Block

In geschweiften Klammern können beliebig viele Anweisungen stehen.

```
let kontostand = 120;

if (kontostand >= 50) {
  console.log("Zahlung möglich.");
  kontostand = kontostand - 50;
  console.log("Neuer Kontostand: " + kontostand);
} else {
  console.log("Nicht genug Guthaben.");
}
```

Die Ausgabe ist:

```
Zahlung möglich.
Neuer Kontostand: 70
```

Die geschweiften Klammern machen deutlich, welche Anweisungen zu `if` oder `else` gehören. Auch wenn ein Block nur aus einer Zeile besteht, solltest du sie verwenden. Das macht deinen Code übersichtlicher und sicherer.

Praktisches Beispiel: Rabatt prüfen

Stell dir einen kleinen Online-Shop vor. Ab einem Bestellwert von 50 erhält ein Kunde kostenlosen Versand.

```
let bestellwert = 64;

if (bestellwert >= 50) {
  console.log("Der Versand ist kostenlos.");
} else {
  console.log("Es fallen Versandkosten an.");
}
```

Ändere den Wert von `bestellwert` und beobachte, wie sich die Ausgabe verändert:

```
let bestellwert = 35;
```

Nun wird der `else`-Block ausgeführt.

Praktisches Beispiel: Passwort prüfen

Auch Texte lassen sich vergleichen:

```
let passwort = "sonne123";

if (passwort === "sonne123") {
  console.log("Zugang erlaubt.");
```

```
} else {
  console.log("Passwort ist falsch.");
}
```

In einer echten Anwendung würdest du Passwörter selbstverständlich nicht so direkt im Browser-Code speichern oder vergleichen. Für das Lernen zeigt dieses Beispiel aber gut, wie eine Text-Bedingung funktioniert.

Häufige Fehler

Semikolon direkt nach `if`

Dieses Semikolon beendet die Anweisung zu früh:

```
if (alter >= 18); {
  console.log("Volljährig");
}
```

Der Codeblock wird dadurch immer ausgeführt — unabhängig von der Bedingung.

Richtig ist:

```
if (alter >= 18) {
  console.log("Volljährig");
}
```

Text ohne Anführungszeichen

Textwerte müssen in Anführungszeichen stehen:

```
let land = "Deutschland";

if (land === "Deutschland") {
  console.log("Willkommen!");
}
```

Ohne Anführungszeichen würde JavaScript `Deutschland` als Variablennamen interpretieren:

```
if (land === Deutschland) {
  console.log("Willkommen!");
}
```

Das führt zu einem Fehler, wenn keine Variable namens `Deutschland` existiert.

Groß- und Kleinschreibung beachten

JavaScript unterscheidet genau zwischen Groß- und Kleinbuchstaben:

```
let benutzername = "Mira";

if (benutzername === "mira") {
  console.log("Gefunden");
} else {
  console.log("Nicht gefunden");
}
```

Die Bedingung ist hier falsch: `"Mira"` und `"mira"` sind unterschiedliche Texte.

Übung 1: Wetterhinweis

Erstelle eine Variable `regnetEs` mit dem Wert `true`. Gib bei Regen „Nimm einen Regenschirm mit.“ aus. Andernfalls soll „Heute brauchst du keinen Regenschirm.“ erscheinen.

Lösung anzeigen

```
let regnetEs = true;

if (regnetEs) {
  console.log("Nimm einen Regenschirm mit.");
} else {
  console.log("Heute brauchst du keinen Regenschirm.");
}
```

Übung 2: Altersprüfung

Erstelle eine Variable `alter`. Wenn die Person mindestens 16 Jahre alt ist, soll „Du darfst dich anmelden.“ ausgegeben werden. Andernfalls soll „Du bist noch zu jung.“ erscheinen.

Lösung anzeigen

```
let alter = 14;

if (alter >= 16) {
  console.log("Du darfst dich anmelden.");
} else {
  console.log("Du bist noch zu jung.");
}
```

Übung 3: Lieblingszahl

Erstelle eine Variable `lieblingszahl`. Prüfe, ob sie genau `7` ist.

- Bei `7`: „Das ist auch meine Lieblingszahl!“
- Bei jeder anderen Zahl: „Eine interessante Wahl.“

Lösung anzeigen

```
let lieblingszahl = 7;

if (lieblingszahl === 7) {
  console.log("Das ist auch meine Lieblingszahl!");
} else {
  console.log("Eine interessante Wahl.");
}
```

Zusammenfassung

- Mit `if` führst du Code nur aus, wenn eine Bedingung wahr ist.
- Mit `else` definierst du eine Alternative für den Fall, dass sie falsch ist.
- Bedingungen ergeben immer `true` oder `false`.
- Für Gleichheitsvergleiche verwendest du `===`.
- `=` weist Werte zu und ist kein Vergleich.
- Geschweifte Klammern zeigen klar, welcher Code zu einer Bedingung gehört.

Im nächsten Schritt lernst du, wie du mit `else if` **mehr als zwei Fälle** unterscheiden kannst.

Mehrere Fälle mit `else if`

Wenn eine Entscheidung mehr als zwei Möglichkeiten hat

Mit `if` und `else` kannst du zwischen **zwei Fällen** unterscheiden:

- Bedingung ist wahr
- Bedingung ist nicht wahr

Oft gibt es aber mehr Möglichkeiten. Zum Beispiel soll ein Programm je nach Punktzahl eine andere Rückmeldung geben:

- ab 90 Punkten: „Sehr gut“
- ab 75 Punkten: „Gut“
- ab 50 Punkten: „Bestanden“
- darunter: „Leider nicht bestanden“

Dafür verwendest du `else if`.

Die Grundstruktur

```
if (bedingung1) {
  // Dieser Code läuft, wenn bedingung1 wahr ist.
} else if (bedingung2) {
  // Dieser Code läuft, wenn bedingung1 falsch
  // und bedingung2 wahr ist.
} else {
  // Dieser Code läuft, wenn keine Bedingung wahr ist.
}
```

JavaScript prüft die Bedingungen **von oben nach unten**:

1. Ist die erste Bedingung wahr? Dann wird ihr Code ausgeführt.
2. Andernfalls prüft JavaScript die nächste `else if`-Bedingung.

3. Sobald eine Bedingung wahr ist, werden die übrigen Fälle übersprungen.
4. Wenn keine Bedingung zutrifft, läuft der Code im `else`-Block.

> **Wichtig:** Von einer `if`-`else if`-`else`-Kette wird immer nur **ein** Block ausgeführt.

Beispiel: Eine Punktzahl auswerten

```
const punkte = 82;

if (punkte >= 90) {
  console.log("Sehr gut!");
} else if (punkte >= 75) {
  console.log("Gut gemacht!");
} else if (punkte >= 50) {
  console.log("Bestanden.");
} else {
  console.log("Leider nicht bestanden.");
}
```

Die Variable `punkte` enthält den Wert `82`.

JavaScript prüft nun nacheinander:

1. Ist `82 >= 90`? Nein.
2. Ist `82 >= 75`? Ja.

Deshalb erscheint in der Konsole:

```
Gut gemacht!
```

Die folgenden Fälle werden nicht mehr geprüft, weil JavaScript bereits einen passenden Fall gefunden hat.

Die Reihenfolge ist entscheidend

Bei mehreren Bedingungen muss die **genaueste oder höchste Bedingung zuerst** stehen.

Dieses Beispiel ist korrekt:

```
const alter = 17;

if (alter >= 18) {
  console.log("Du bist volljährig.");
} else if (alter >= 16) {
  console.log("Du darfst bestimmte Angebote nutzen.");
} else {
  console.log("Du bist noch unter 16.");
}
```

Für `alter = 17` ist die erste Bedingung falsch, die zweite aber wahr. Die Ausgabe lautet:

```
Du darfst bestimmte Angebote nutzen.
```

Ein häufiger Fehler

Schau dir diese vertauschte Reihenfolge an:

```
const punkte = 95;

if (punkte >= 50) {
  console.log("Bestanden.");
} else if (punkte >= 75) {
  console.log("Gut gemacht!");
} else if (punkte >= 90) {
  console.log("Sehr gut!");
}
```

Die Ausgabe ist:

```
Bestanden.
```

Das ist zwar technisch korrekt, aber vermutlich nicht gewollt. Denn `95` ist bereits größer oder gleich `50`. JavaScript führt deshalb den ersten Block aus und prüft die restlichen Bedingungen nicht mehr.

Besser: Beginne bei solchen Wertebereichen mit dem höchsten Wert.

```
const punkte = 95;

if (punkte >= 90) {
  console.log("Sehr gut!");
} else if (punkte >= 75) {
  console.log("Gut gemacht!");
} else if (punkte >= 50) {
  console.log("Bestanden.");
} else {
  console.log("Leider nicht bestanden.");
}
```

Jetzt lautet die Ausgabe:

```
Sehr gut!
```

Mehrere konkrete Werte prüfen

`else if` eignet sich auch, wenn eine Variable verschiedene feste Werte annehmen kann.

```
const wochentag = "Samstag";

if (wochentag === "Montag") {
  console.log("Die Woche beginnt.");
} else if (wochentag === "Freitag") {
  console.log("Fast Wochenende!");
} else if (wochentag === "Samstag" || wochentag === "Sonntag") {
  console.log("Wochenende!");
} else {
  console.log("Ein normaler Wochentag.");
}
```

Hier wird zusätzlich der logische Operator `||` verwendet. Er bedeutet **„oder“**.

Die Bedingung

```
wochentag === "Samstag" || wochentag === "Sonntag"
```

ist wahr, wenn mindestens eine der beiden Teilbedingungen wahr ist.

`else` ist optional

Ein abschließendes `else` musst du nicht immer schreiben. Es ist sinnvoll, wenn dein Programm auch für alle übrigen Fälle eine Reaktion zeigen soll.

Ohne `else`:

```
const temperatur = 24;

if (temperatur < 0) {
  console.log("Es friert.");
} else if (temperatur < 15) {
  console.log("Es ist kühl.");
} else if (temperatur > 30) {
  console.log("Es ist heiß.");
}
```

Bei `24` erscheint keine Ausgabe, denn keine Bedingung trifft zu.

Mit einem abschließenden `else` kannst du diesen normalen Bereich auffangen:

```
const temperatur = 24;

if (temperatur < 0) {
  console.log("Es friert.");
} else if (temperatur < 15) {
  console.log("Es ist kühl.");
} else if (temperatur > 30) {
  console.log("Es ist heiß.");
} else {
  console.log("Die Temperatur ist angenehm.");
}
```

Verschachtelte Bedingungen vermeiden

Du könntest mehrere Bedingungen auch ineinander schreiben:

```
const rolle = "admin";

if (rolle === "admin") {
  console.log("Du hast alle Rechte.");
} else {
  if (rolle === "mitglied") {
    console.log("Du bist angemeldet.");
  } else {
    console.log("Du bist ein Gast.");
  }
}
```

Das funktioniert, ist aber unnötig unübersichtlich. Mit `else if` ist derselbe Code klarer:

```
const rolle = "admin";

if (rolle === "admin") {
  console.log("Du hast alle Rechte.");
} else if (rolle === "mitglied") {
  console.log("Du bist angemeldet.");
} else {
  console.log("Du bist ein Gast.");
}
```

Verwende `else if`, wenn die Fälle **alternativ** sind: Es soll genau eine passende Antwort ausgewählt werden.

Typische Fehlerquellen

`=` statt `===` verwenden

Für Vergleiche brauchst du meistens `===`.

```
const farbe = "blau";

if (farbe === "blau") {
  console.log("Die Farbe ist blau.");
}
```

Ein einzelnes `=` weist dagegen einen Wert zu:

```
farbe = "rot";
```

In einer Bedingung führt das oft zu Fehlern oder unerwartetem Verhalten.

Überlappende Bedingungen falsch sortieren

Bei Zahlenbereichen überschneiden sich Bedingungen häufig. Zum Beispiel trifft bei einer Punktzahl von `90` sowohl `punkte >= 50` als auch `punkte >= 75` und `punkte >= 90` zu.

Darum gilt:

- höchste Grenze zuerst
- niedrigste Grenze zuletzt
- einen Restfall mit `else` ergänzen, wenn nötig

`else if` nach `else` schreiben

Ein `else` muss immer am Ende stehen. Das wäre falsch:

```
if (alter >= 18) {
  console.log("Volljährig");
} else {
  console.log("Minderjährig");
} else if (alter >= 16) {
  console.log("16 oder älter");
}
```

Sobald JavaScript bei `else` angekommen ist, gibt es keinen weiteren Bedingungsfall mehr. Schreibe `else if` daher immer **vor** `else`.

Praxisbeispiel: Versandkosten bestimmen

In einem Online-Shop hängen die Versandkosten vom Bestellwert ab:

- ab 50: kostenloser Versand
- ab 25: Versand kostet 2,99
- darunter: Versand kostet 4,99

```
const bestellwert = 32;

if (bestellwert >= 50) {
  console.log("Der Versand ist kostenlos.");
} else if (bestellwert >= 25) {
  console.log("Der Versand kostet 2,99 Euro.");
} else {
  console.log("Der Versand kostet 4,99 Euro.");
}
```

Für einen Bestellwert von `32` wird ausgegeben:

```
Der Versand kostet 2,99 Euro.
```

Ändere den Wert von `bestellwert` und beobachte, wie sich die Ausgabe verändert.

Übung: Ticketpreis berechnen

Schreibe ein Programm mit einer Variable `alter`.

Die Regeln lauten:

- Kinder unter 6 Jahren fahren kostenlos.
- Personen unter 18 Jahren zahlen den Jugendpreis.

- Personen ab 65 Jahren erhalten einen Seniorenpreis.
- Alle anderen zahlen den normalen Preis.

Gib jeweils einen passenden Text in der Konsole aus.

Teste zum Beispiel diese Werte: `4`, `15`, `30` und `70`.

Eine mögliche Lösung

```
const alter = 15;


if (alter < 6) {
  console.log("Dein Ticket ist kostenlos.");
} else if (alter < 18) {
  console.log("Du zahlst den Jugendpreis.");
} else if (alter >= 65) {
  console.log("Du zahlst den Seniorenpreis.");
} else {
  console.log("Du zahlst den normalen Preis.");
}
```

Beachte die Reihenfolge:

- Zuerst wird geprüft, ob die Person noch ein Kind ist.
- Danach folgt der Jugendpreis.
- Anschließend wird der Seniorenpreis geprüft.
- Das abschließende `else` gilt für alle Erwachsenen zwischen 18 und 64 Jahren.

Merke dir

- Mit `else if` kannst du **mehrere alternative Fälle** formulieren.
- JavaScript prüft die Bedingungen von **oben nach unten**.
- Sobald eine Bedingung wahr ist, wird nur ihr Codeblock ausgeführt.
- Bei überlappenden Zahlenbereichen steht die **höchste Grenze zuerst**.
- Ein `else` fängt alle Fälle auf, die zu keiner vorherigen Bedingung passen.

Schleifen mit `for`

Lernziele

Nach dieser Lektion kannst du:

- eine `for`-Schleife schreiben und lesen,
- Code eine bestimmte Anzahl von Malen wiederholen,
- einen Zähler mit `let` verwenden,
- typische Fehler wie eine Endlosschleife oder einen falschen Startwert erkennen.

Was ist eine `for`-Schleife?

Eine Schleife wiederholt einen Codeblock. Eine `for`-Schleife ist besonders praktisch, wenn du schon weißt, *wie oft* etwas passieren soll.

Stell dir vor, du möchtest fünfmal „Hallo!“ in der Konsole ausgeben. Ohne Schleife müsstest du denselben Befehl fünfmal schreiben:

```
console.log("Hallo!");
console.log("Hallo!");
console.log("Hallo!");
console.log("Hallo!");
console.log("Hallo!");
```

Mit einer `for`-Schleife geht das kürzer und übersichtlicher:

```
for (let i = 1; i <= 5; i++) {
  console.log("Hallo!");
}
```

Die Konsole zeigt:

```
Hallo!
Hallo!
```

```
Hallo!
Hallo!
Hallo!
```

Der Aufbau einer `for`-Schleife

Eine `for`-Schleife besteht aus drei Teilen:

```
for (Startwert; Bedingung; Veränderung) {
  // Code, der wiederholt wird
}
```

Ein konkretes Beispiel:

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

Die drei Teile bedeuten:

Teil	Beispiel	Bedeutung
Startwert	`let i = 1`	Die Zählvariable `i` beginnt bei `1`.
Bedingung	`i <= 5`	Die Schleife läuft, solange `i` kleiner oder gleich `5` ist.
Veränderung	`i++`	Nach jedem Durchlauf wird `i` um `1` erhöht.

`i` ist ein üblicher Name für einen Zähler. Du darfst aber auch einen verständlicheren Namen wählen:

```
for (let nummer = 1; nummer <= 5; nummer++) {
  console.log(nummer);
}
```

Schritt für Schritt: Was passiert in der Schleife?

Sieh dir diese Schleife an:

```
for (let i = 1; i <= 3; i++) {
  console.log("Durchlauf: " + i);
}
```

JavaScript arbeitet sie so ab:

1. `let i = 1` erstellt den Zähler mit dem Wert `1`.
2. JavaScript prüft: Ist `i <= 3`?
3. Ja. Der Code im Block wird ausgeführt.
4. `i++` erhöht `i` auf `2`.
5. Die Bedingung wird erneut geprüft.
6. Sobald `i` den Wert `4` hat, ist `i <= 3` falsch. Die Schleife endet.

Die Ausgabe lautet:

```
Durchlauf: 1
Durchlauf: 2
Durchlauf: 3
```

> **Merke:** Der Code innerhalb der geschweiften Klammern wird bei jedem Durchlauf ausgeführt.

Zählen von 1 bis 10

Ein häufiger Anwendungsfall ist das Hochzählen:

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

Ausgabe:

```
1
2
3
4
5
6
7
8
9
10
```

Achte auf den Vergleichsoperator `<=`:

- `i <= 10` bedeutet: Die `10` wird noch ausgegeben.
- `i < 10` bedeutet: Die Schleife endet bereits bei `9`.

```
for (let i = 1; i < 10; i++) {
  console.log(i);
}
```

Diese Schleife gibt nur die Zahlen von `1` bis `9` aus.

Bei 0 beginnen

In JavaScript beginnt man häufig bei `0` zu zählen. Das ist später besonders wichtig bei Arrays.

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

Ausgabe:

```
0
1
2
3
4
```

Die Schleife läuft insgesamt fünfmal:

- beim Wert `0`,
- beim Wert `1`,
- beim Wert `2`,
- beim Wert `3`,
- beim Wert `4`.

Obwohl die letzte Zahl `4` ist, gibt es fünf Durchläufe.

Rückwärts zählen

Du kannst einen Zähler auch verkleinern. Dafür verwendest du `--`.

```
for (let i = 5; i >= 1; i--) {
  console.log(i);
}
```

Ausgabe:

```
5
4
3
2
1
```

Hier gilt:

- Start bei `5`
- Wiederholung, solange `i` größer oder gleich `1` ist
- Nach jedem Durchlauf wird `i` um `1` kleiner

Das Gegenstück zu `i++` ist `i--`:

```
i++; // erhöht um 1
i--; // verringert um 1
```

In größeren Schritten zählen

Die Veränderung muss nicht immer genau `1` sein. Du kannst beispielsweise in Zweierschritten zählen:

```
for (let i = 0; i <= 10; i += 2) {
  console.log(i);
}
```

Ausgabe:

```
0
2
4
6
8
10
```

`i += 2` bedeutet: Addiere bei jedem Durchlauf `2` zu `i`.

Auch andere Schrittgrößen sind möglich:

```
for (let i = 10; i >= 0; i -= 2) {
  console.log(i);
}
```

Ausgabe:

```
10
8
6
4
2
0
```

Texte mit einem Zähler ausgeben

Der Zähler kann innerhalb der Schleife verwendet werden. So kannst du beispielsweise nummerierte Begrüßungen erzeugen:

```
for (let i = 1; i <= 3; i++) {
  console.log("Begrüßung Nummer " + i);
}
```

Ausgabe:

```
Begrüßung Nummer 1
Begrüßung Nummer 2
Begrüßung Nummer 3
```

Mit einem Template Literal ist die Schreibweise oft gut lesbar:

```
for (let i = 1; i <= 3; i++) {
  console.log(`Begrüßung Nummer ${i}`);
}
```

Berechnungen in einer Schleife

Schleifen sind nützlich, um wiederholt Berechnungen durchzuführen.

Dieses Beispiel gibt die Einmaleins-Reihe für die Zahl `4` aus:

```
for (let i = 1; i <= 10; i++) {
  let ergebnis = 4 * i;
  console.log(`4 mal ${i} ist ${ergebnis}`);
}
```

Ausgabe:

```
4 mal 1 ist 4
4 mal 2 ist 8
4 mal 3 ist 12
...
4 mal 10 ist 40
```

Die Variable `ergebnis` wird bei jedem Durchlauf neu berechnet.

Einen Wert aufsummieren

Eine Schleife kann auch Werte sammeln. Zum Beispiel möchtest du alle Zahlen von `1` bis `5` addieren.

```
let summe = 0;

for (let i = 1; i <= 5; i++) {
  summe = summe + i;
}

console.log(summe);
```

Ausgabe:

```
15
```

So verändert sich `summe`:

Durchlauf	`i`	Neue Summe
Start	–	`0`
1	`1`	`1`
2	`2`	`3`
3	`3`	`6`
4	`4`	`10`
5	`5`	`15`

Die Zeile

```
summe = summe + i;
```

kann auch kürzer geschrieben werden:

```
summe += i;
```

Beide Varianten haben dieselbe Bedeutung.

Der Gültigkeitsbereich des Zählers

Wird der Zähler mit `let` in der `for`-Schleife erstellt, existiert er nur innerhalb der Schleife:

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}

console.log(i);
```

Die letzte Zeile führt zu einem Fehler, weil `i` außerhalb der Schleife nicht bekannt ist.

Das ist sinnvoll: Der Zähler wird meist nur für die Schleife benötigt. Verwende deshalb in der Regel `let` direkt im Schleifenkopf.

Häufige Fehler

Die Bedingung endet zu früh

```
for (let i = 1; i < 5; i++) {
  console.log(i);
}
```

Diese Schleife gibt `1` bis `4` aus, nicht bis `5`.

Wenn die `5` dazugehören soll, brauchst du:

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

Der Zähler wird in die falsche Richtung verändert

```
for (let i = 1; i <= 5; i--) {
  console.log(i);
}
```

Hier wird `i` immer kleiner: `1`, `0`, `-1` und so weiter. Gleichzeitig bleibt die Bedingung `i <= 5` wahr. Das kann zu einer **Endlosschleife** führen.

Korrekt ist:

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

> **Achtung:** Wenn eine Webseite oder die Konsole scheinbar nicht mehr reagiert, könnte eine Endlosschleife die Ursache sein. Prüfe dann besonders die Bedingung und die Veränderung des Zählers.

Ein Semikolon vergessen

Im Kopf einer `for`-Schleife stehen zwei Semikolons:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

Falsch wäre:

```
for (let i = 0 i < 5 i++) {
  console.log(i);
}
```

JavaScript kann die drei Bereiche dann nicht mehr voneinander unterscheiden.

Geschweifte Klammern weglassen

Bei nur einer Anweisung kann JavaScript zwar ohne Klammern arbeiten:

```
for (let i = 1; i <= 3; i++)
  console.log(i);
```

Für Einsteiger ist es aber besser, die Klammern immer zu schreiben:

```
for (let i = 1; i <= 3; i++) {
  console.log(i);
}
```

So ist klar erkennbar, welcher Code zur Schleife gehört. Außerdem kannst du später leichter weitere Anweisungen ergänzen.

Praktisches Beispiel: Countdown

Ein Countdown lässt sich mit einer rückwärts laufenden Schleife umsetzen:

```
for (let sekunden = 5; sekunden >= 1; sekunden--) {
  console.log(`${sekunden} ...`);
}

console.log("Los!");
```

Ausgabe:

```
5 ...
4 ...
3 ...
2 ...
1 ...
Los!
```

Wichtig: Die Ausgaben erscheinen hier sehr schnell hintereinander. Ein echter Countdown mit einer Sekunde Pause benötigt später zusätzliche Funktionen wie `setInterval()`.

Übungen

Übung 1: Zahlen ausgeben

Schreibe eine `for`-Schleife, die alle Zahlen von `1` bis `20` in der Konsole ausgibt.

Lösung ansehen

```
for (let i = 1; i <= 20; i++) {
  console.log(i);
}
```

Übung 2: Gerade Zahlen

Gib alle geraden Zahlen von `2` bis `20` aus.

Lösung ansehen

```
for (let i = 2; i <= 20; i += 2) {
  console.log(i);
}
```

Übung 3: Rückwärts zählen

Erstelle einen Countdown von `10` bis `0`.

Lösung ansehen

```
for (let i = 10; i >= 0; i--) {
  console.log(i);
}
```

Übung 4: Kleine Rechenreihe

Gib die Ergebnisse der 7er-Reihe von `7 mal 1` bis `7 mal 10` aus.

Lösung ansehen

```
for (let i = 1; i <= 10; i++) {
  let ergebnis = 7 * i;
  console.log(`7 mal ${i} ist ${ergebnis}`);
}
```

Zusammenfassung

Eine `for`-Schleife wiederholt Code, solange ihre Bedingung wahr ist:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

Dabei gilt:

- Der **Startwert** legt fest, wo gezählt wird.
- Die **Bedingung** entscheidet, ob ein weiterer Durchlauf erfolgt.
- Die **Veränderung** passt den Zähler nach jedem Durchlauf an.
- `i++` erhöht einen Wert um `1`, `i--` verringert ihn um `1`.
- Achte darauf, dass Bedingung und Veränderung zusammenpassen, damit keine Endlosschleife entsteht.

Im nächsten Schritt lernst du die `while`-Schleife kennen. Sie eignet sich besonders für Wiederholungen, bei denen die Anzahl der Durchläufe vorher noch nicht feststeht.

Schleifen mit `while`

Wiederholen, solange eine Bedingung wahr ist

Eine `while`-Schleife wiederholt Code, **solange** eine Bedingung `true` ergibt. Sie ist besonders nützlich, wenn du vorher nicht genau weißt, wie oft sich etwas wiederholen muss.

Zum Beispiel: Eine Zahl soll so lange erhöht werden, bis sie 5 erreicht hat.

```
let zahl = 1;

while (zahl <= 5) {
  console.log(zahl);
  zahl = zahl + 1;
}
```

Die Konsole zeigt:

```
1
2
3
4
5
```

Der Aufbau einer `while`-Schleife

Eine `while`-Schleife besteht aus drei wichtigen Teilen:

```
while (bedingung) {
  // Code, der wiederholt wird
}
```

- `while` bedeutet auf Englisch „solange“.
- In den runden Klammern steht eine Bedingung.
- Der Code im Block `{ ... }` wird ausgeführt, solange die Bedingung wahr ist.

Schau dir dieses Beispiel Schritt für Schritt an:

```
let punktzahl = 0;

while (punktzahl < 3) {
  console.log("Noch nicht genug Punkte.");
  punktzahl = punktzahl + 1;
}

console.log("Ziel erreicht!");
```

So läuft das Programm ab:

1. `punktzahl` startet bei `0`.
2. Die Bedingung `punktzahl < 3` ist wahr.
3. Der Text wird ausgegeben.
4. `punktzahl` wird um `1` erhöht.
5. Die Bedingung wird erneut geprüft.
6. Sobald `punktzahl` den Wert `3` hat, ist die Bedingung falsch.
7. Die Schleife endet.

Die Ausgabe lautet:

```
Noch nicht genug Punkte.
Noch nicht genug Punkte.
Noch nicht genug Punkte.
Ziel erreicht!
```

> **Merke:** Die Bedingung wird *vor jedem Durchlauf* geprüft. Ist sie schon am Anfang `false`, wird der Schleifenblock kein einziges Mal ausgeführt.

Einen Zähler verändern

Damit eine `while`-Schleife irgendwann endet, muss sich meist eine Variable innerhalb der Schleife verändern. Diese Variable nennt man oft **Zähler**.

```
let zaehler = 1;

while (zaehler <= 4) {
  console.log("Durchlauf Nummer: " + zaehler);
  zaehler++;
}
```

`zaehler++` ist eine Kurzschreibweise für:

```
zaehler = zaehler + 1;
```

Die Schleife gibt Folgendes aus:

```
Durchlauf Nummer: 1
Durchlauf Nummer: 2
Durchlauf Nummer: 3
Durchlauf Nummer: 4
```

Du kannst einen Zähler auch rückwärts laufen lassen:

```
let countdown = 5;

while (countdown > 0) {
  console.log(countdown);
  countdown--;
}

console.log("Los!");
```

`countdown--` bedeutet:

```
countdown = countdown - 1;
```

Ausgabe:

```
5
4
3
2
1
```

```
Los!
```

Achtung vor Endlosschleifen

Wenn die Bedingung immer wahr bleibt, endet eine `while`-Schleife nie. Das nennt man **Endlosschleife**. Sie kann dazu führen, dass der Browser nicht mehr reagiert.

Dieses Beispiel ist problematisch:

```
let zahl = 1;

while (zahl <= 5) {
  console.log(zahl);
}
```

Die Variable `zahl` wird nie verändert. Sie bleibt immer `1`, deshalb ist `zahl <= 5` immer wahr.

Die richtige Version erhöht `zahl` innerhalb der Schleife:

```
let zahl = 1;

while (zahl <= 5) {
  console.log(zahl);
  zahl++;
}
```

> **Prüffrage:** Wird mindestens eine Variable verändert, sodass die Bedingung irgendwann `false` werden kann?

Falls du versehentlich eine Endlosschleife startest, kannst du das Laden der Seite im Browser abbrechen oder den Tab schließen. Speichere deinen Code am besten regelmäßig.

Ein praktisches Beispiel: Guthaben abbauen

Stell dir vor, ein Spielzug kostet 2 Punkte. Solange genügend Punkte vorhanden sind, darf weitergespielt werden.

```
let guthaben = 10;

while (guthaben >= 2) {
  console.log("Eine Runde wird gespielt.");
  guthaben = guthaben - 2;
  console.log("Verbleibendes Guthaben: " + guthaben);
}

console.log("Nicht genug Guthaben für eine weitere Runde.");
```

Die Bedingung lautet `guthaben >= 2`. Eine Runde startet also nur, wenn noch mindestens 2 Punkte verfügbar sind.

Werte in einem Array mit `while` ausgeben

Auch Arrays kannst du mit einer `while`-Schleife durchgehen. Dafür benötigst du einen Index, der bei `0` beginnt.

```
const farben = ["Rot", "Blau", "Grün"];
let index = 0;

while (index < farben.length) {
  console.log(farben[index]);
  index++;
}
```

Ausgabe:

```
Rot
Blau
Grün
```

Dabei bedeutet:

- `farben.length`: Anzahl der Elemente im Array

- `farben[index]`: Element an der aktuellen Stelle
- `index++`: Zum nächsten Element wechseln

Die Bedingung `index < farben.length` ist wichtig. Der größte gültige Index liegt immer **eins unter** der Länge des Arrays.

Bei drei Elementen sind die gültigen Indizes `0`, `1` und `2`.

`while` oder `for`?

Viele Aufgaben lassen sich sowohl mit `for` als auch mit `while` lösen.

Eine `for`-Schleife passt gut, wenn die Anzahl der Wiederholungen von Anfang an klar ist:

```
for (let zahl = 1; zahl <= 5; zahl++) {
  console.log(zahl);
}
```

Eine `while`-Schleife passt gut, wenn eine Bedingung entscheidet, wann Schluss ist:

```
let temperatur = 18;

while (temperatur < 22) {
  temperatur++;
  console.log("Temperatur: " + temperatur);
}
```

Hier ist nicht die Anzahl der Durchläufe entscheidend, sondern das Ziel: Die Temperatur soll 22 erreichen.

Verwende …	Wenn …
`for`	die Anzahl der Durchläufe klar feststeht
`while`	eine Bedingung bestimmt, wann die Wiederholung endet

Häufige Fehler

Die Zählvariable wird nicht verändert

```
let alter = 10;

while (alter < 18) {
  console.log("Noch nicht volljährig.");
}
```

Diese Schleife endet nie. Die Lösung:

```
let alter = 10;

while (alter < 18) {
  console.log("Noch nicht volljährig.");
  alter++;
}
```

Die Bedingung ist von Beginn an falsch

```
let versuche = 5;

while (versuche < 3) {
  console.log("Neuer Versuch");
  versuche++;
}
```

Hier erscheint keine Ausgabe, denn `5 < 3` ist direkt am Anfang falsch.

Falscher Vergleichsoperator

```
let nummer = 1;

while (nummer < 5) {
  console.log(nummer);
  nummer++;
}
```

Diese Schleife gibt nur `1` bis `4` aus. Soll auch die `5` erscheinen, brauchst du `<=`:

```
let nummer = 1;

while (nummer <= 5) {
  console.log(nummer);
  nummer++;
}
```

Übung: Gerade Zahlen ausgeben

Gib alle geraden Zahlen von 2 bis 10 mit einer `while`-Schleife aus.

Erwartete Ausgabe:

```
2
4
6
8
10
```

```
let zahl = 2;

while (zahl <= 10) {
  console.log(zahl);
  zahl = zahl + 2;
}
```

Übung: Begrüßungen zählen

Schreibe eine Schleife, die dreimal „Hallo!“ in der Konsole ausgibt. Verwende dafür eine Variable namens `anzahl`.

```
let anzahl = 1;

while (anzahl <= 3) {
  console.log("Hallo!");
  anzahl++;
}
```

Das Wichtigste auf einen Blick

- Eine `while`-Schleife wiederholt Code, solange ihre Bedingung `true` ist.
- Die Bedingung wird vor jedem Durchlauf geprüft.
- Eine Zählvariable muss sich meistens innerhalb der Schleife ändern.
- Ohne Veränderung kann eine Endlosschleife entstehen.
- Verwende `while`, wenn nicht die feste Anzahl der Wiederholungen, sondern eine Bedingung im Mittelpunkt steht.

Im nächsten Schritt kannst du Bedingungen und Schleifen zusammen einsetzen, um Programme flexibel auf unterschiedliche Situationen reagieren zu lassen.