LU01.L10

```
/**
 * ========================================================================
 *  MUSTERLOESUNG: Die Grundrechenarten in JavaScript
 * ========================================================================
 *
 *  Diese Datei ist identisch mit "grundrechenarten.js", enthaelt aber
 *  ausformulierte Musterkommentare an allen 10 Aufgabenstellen.
 *  Nur fuer die Lehrperson bestimmt.
 *
 *  Bewertungshinweis: Erwartet wird sinngemaesse Uebereinstimmung, nicht
 *  Wortgleichheit. Pro Aufgabe genuegen die fett markierten Kernpunkte.
 * ========================================================================
 */

"use strict";

// ========================================================================
//  TEIL 1 - Zwei Zahlen, vier Rechenarten
// ========================================================================

const a = 12;
const b = 5;

// --- AUFGABE 1 ----------------------------------------------------------
// TODO: Erklaere, was die beiden Zeilen oben bewirken. Gehe dabei auf
//       das Schluesselwort "const" ein.
// Antwort:
// Es werden zwei Variablen a und b deklariert und mit den Werten 12
// bzw. 5 initialisiert. "const" erzeugt eine Konstante: Der Name darf
// nach der Zuweisung nicht mehr auf einen anderen Wert zeigen - ein
// spaeteres "a = 20;" fuehrt zu einem TypeError. Mit "let" deklarierte
// Variablen sind dagegen jederzeit neu zuweisbar. Faustregel: immer
// const verwenden, let nur dort, wo sich der Wert wirklich aendern muss
// (z. B. der Zaehler einer Schleife).
// Kernpunkte: Deklaration + Initialisierung / const = nicht neu
// zuweisbar / Abgrenzung zu let.

function addieren(x, y) {
  return x + y;
}

function subtrahieren(x, y) {
  return x - y;
}

function multiplizieren(x, y) {
  return x * y;
```

```
}

// --- AUFGABE 2 ---------------------------------------------------
// TODO: Beschreibe den Aufbau einer Funktion am Beispiel "addieren".
// Antwort:
// "function" leitet die Definition ein, danach folgt der Funktionsname
// "addieren". In den runden Klammern stehen die Parameter x und y -
// Platzhalter fuer die Werte, die beim Aufruf uebergeben werden
// (Argumente). Beim Aufruf addieren(12, 5) wird x zu 12 und y zu 5.
// Zwischen den geschweiften Klammern steht der Funktionsrumpf.
// "return" beendet die Funktion sofort und gibt den berechneten Wert an
// die aufrufende Stelle zurueck, wo er weiterverwendet werden kann.
// Ohne return liefert eine Funktion "undefined".
// Kernpunkte: Parameter sind Platzhalter / return gibt Wert zurueck und
// beendet die Funktion.

function dividieren(x, y) {
  if (y === 0) {
    return "Division durch 0 ist nicht definiert";
  }
  return x / y;
}

// --- AUFGABE 3 ---------------------------------------------------
// TODO: Warum enthaelt "dividieren" eine zusaetzliche if-Abfrage?
// Antwort:
// Die Division durch null ist mathematisch nicht definiert; Addition,
// Subtraktion und Multiplikation haben keinen vergleichbaren
// Sonderfall. JavaScript wirft dabei aber keinen Fehler, sondern
// liefert den speziellen Wert "Infinity" (bzw. -Infinity, und NaN bei
// 0 / 0). Das ist tueckisch, weil das Programm scheinbar normal
// weiterlaeuft und der unsinnige Wert erst spaeter auffaellt. Die
// if-Abfrage faengt den Fall ab und gibt eine verstaendliche Meldung
// zurueck.
// Kernpunkte: 12 / 0 ergibt Infinity, keinen Fehler / Abfangen macht
// den Fehler sichtbar.

console.log("=== TEIL 1: Grundrechenarten ===");
console.log(`${a} + ${b} = ${addieren(a, b)}`);
console.log(`${a} - ${b} = ${subtrahieren(a, b)}`);
console.log(`${a} * ${b} = ${multiplizieren(a, b)}`);
console.log(`${a} / ${b} = ${dividieren(a, b)}`);
console.log(`${a} / 0 = ${dividieren(a, 0)}`);

// --- AUFGABE 4 ---------------------------------------------------
// TODO: Wie heisst die Schreibweise mit Backticks und ${...}?
// Antwort:
// Das sind Template Literals (Vorlagenzeichenketten). Zwischen den
// Backticks darf beliebiger Text stehen, und in ${...} wird ein
```

```
// JavaScript-Ausdruck ausgewertet und sein Ergebnis an dieser Stelle
// eingesetzt - auch ein Funktionsaufruf wie ${addieren(a, b)}.
// Vorteile gegenueber der Verkettung mit +: deutlich besser lesbar,
// weniger Anfuehrungszeichen und Pluszeichen, kein vergessenes
// Leerzeichen, und mehrzeilige Zeichenketten sind direkt moeglich.
// Kernpunkte: Template Literal / Ausdruck in ${} wird eingesetzt /
// Lesbarkeit.

// ========================================================================
//  TEIL 2 - Ganzzahldivision und Rest
// ========================================================================

const zaehler = 17;
const nenner = 5;

const ganzeTeile = Math.floor(zaehler / nenner);
const rest = zaehler % nenner;

console.log("\n=== TEIL 2: Division mit Rest ===");
console.log(`${zaehler} : ${nenner} = ${ganzeTeile} Rest ${rest}`);

// --- AUFGABE 5 ----------------------------------------------------------
// TODO: Was macht Math.floor(), was macht %?
// Antwort:
// a) Math.floor() rundet eine Zahl immer ab, also zur naechstkleineren
//    ganzen Zahl: 17 / 5 ergibt 3.4, Math.floor(3.4) ergibt 3. So
//    erhaelt man aus einer normalen Division die Ganzzahldivision.
//    (Achtung bei negativen Zahlen: Math.floor(-3.4) ist -4.)
// b) Der Modulo-Operator % liefert den Rest einer Ganzzahldivision:
//    17 % 5 ist 2, denn 5 passt dreimal in 17 und es bleiben 2 uebrig.
// Praktische Beispiele: Pruefen, ob eine Zahl gerade ist (n % 2);
// jede dritte Tabellenzeile einfaerben (i % 3 === 0); Sekunden in
// Minuten und Sekunden umrechnen; zyklisch durch eine Liste laufen
// (index % laenge).
// Kernpunkte: floor = abrunden / % = Rest / ein sinnvolles Beispiel.

const zahl = 42;
const istGerade = zahl % 2 === 0;
console.log(`Ist ${zahl} gerade? ${istGerade}`);

// --- AUFGABE 6 ----------------------------------------------------------
// TODO: Unterschied zwischen "=" und "==="? Datentyp von istGerade?
// Antwort:
// "=" ist der Zuweisungsoperator: Der Wert rechts wird in die Variable
// links geschrieben. "===" ist der strikte Vergleichsoperator: Er
// prueft, ob zwei Werte gleich sind, UND ob sie denselben Datentyp
// haben, und liefert true oder false. ("==" vergleicht ebenfalls, wandelt
// die Typen aber vorher um - "5" == 5 ist true, "5" === 5 ist false;
// deshalb verwendet man in der Praxis ===.)
// Ausgewertet wird zuerst zahl % 2 (ergibt 0), dann der Vergleich
// 0 === 0 (ergibt true), und dieses Ergebnis wird zugewiesen.
```

```
// istGerade hat also den Datentyp boolean.
// Kernpunkte: Zuweisung vs. Vergleich / === prueft auch den Typ /
// boolean.

// ========================================================================
//  TEIL 3 - Rangfolge der Operatoren (Punkt vor Strich)
// ========================================================================

const ohneKlammern = 2 + 3 * 4;
const mitKlammern = (2 + 3) * 4;

console.log("\n=== TEIL 3: Rangfolge ===");
console.log(`2 + 3 * 4   = ${ohneKlammern}`);
console.log(`(2 + 3) * 4 = ${mitKlammern}`);

// --- AUFGABE 7 ----------------------------------------------------------
// TODO: Warum liefern beide Zeilen verschiedene Ergebnisse?
// Antwort:
// JavaScript haelt sich an dieselbe Rangfolge wie die Mathematik:
// Punktrechnung vor Strichrechnung. In der ersten Zeile wird deshalb
// zuerst 3 * 4 = 12 berechnet und dann 2 + 12 = 14. Klammern haben die
// hoechste Prioritaet und heben diese Reihenfolge auf: In der zweiten
// Zeile wird zuerst 2 + 3 = 5 gerechnet und dann 5 * 4 = 20.
// Merke: Wenn eine Rechnung laenger wird, setzt man Klammern lieber
// einmal zu viel als zu wenig - das kostet nichts und macht die Absicht
// eindeutig.
// Kernpunkte: 14 vs. 20 / Punkt vor Strich / Klammern haben Vorrang.

// ========================================================================
//  TEIL 4 - Wenn Kommazahlen ungenau werden
// ========================================================================

const summe = 0.1 + 0.2;

console.log("\n=== TEIL 4: Kommazahlen ===");
console.log(`0.1 + 0.2 = ${summe}`);
console.log(`Ist 0.1 + 0.2 === 0.3 ? ${summe === 0.3}`);
console.log(`Gerundet auf 2 Stellen: ${summe.toFixed(2)}`);

// --- AUFGABE 8 ----------------------------------------------------------
// TODO: Warum ueberrascht das Ergebnis von 0.1 + 0.2?
// Antwort:
// Die Konsole zeigt 0.30000000000000004, und der Vergleich mit 0.3
// ergibt false. Grund: JavaScript speichert alle Zahlen als binaere
// Gleitkommazahlen (Standard IEEE 754, 64 Bit). Im Zweiersystem lassen
// sich 0.1 und 0.2 nicht exakt darstellen - aehnlich wie sich 1/3 im
// Zehnersystem nicht exakt als endliche Dezimalzahl schreiben laesst.
// Beide Werte werden also minimal gerundet gespeichert, und beim
// Addieren summiert sich der Fehler zu einer sichtbaren Abweichung.
```

```
// toFixed(2) rundet auf zwei Nachkommastellen und liefert "0.30".
// Der Haken: Das Ergebnis ist ein String, keine Zahl - damit laesst
// sich nicht weiterrechnen ("0.30" + 1 ergibt "0.301"). toFixed eignet
// sich fuer die Ausgabe, nicht fuer die Rechnung. Fuer Vergleiche nimmt
// man Math.abs(x - y) < 1e-9, fuer Geldbetraege rechnet man in Rappen
// bzw. Cent mit ganzen Zahlen.
// Kernpunkte: 0.30000000000000004 / binaere Gleitkommadarstellung /
// toFixed liefert einen String.

// ============================================================================
//  TEIL 5 - Eine Rechnung, viele Male: die Schleife
// ============================================================================

const reihe = 7;

console.log(`\n=== TEIL 5: Einmaleins der ${reihe} ===`);
for (let i = 1; i <= 10; i++) {
  console.log(`${reihe} * ${i} = ${multiplizieren(reihe, i)}`);
}

// --- AUFGABE 9 ---------------------------------------------------------
// TODO: Zerlege den Kopf der for-Schleife in seine drei Bestandteile.
// Antwort:
// 1. "let i = 1"  - Initialisierung: wird genau einmal vor dem ersten
//    Durchlauf ausgefuehrt und legt die Zaehlvariable an. Hier "let"
//    statt "const", weil sich i in jedem Durchlauf aendert.
// 2. "i <= 10"    - Bedingung: wird vor jedem Durchlauf geprueft.
//    Solange sie true ergibt, laeuft die Schleife weiter; ergibt sie
//    false, bricht sie ab.
// 3. "i++"        - Fortschaltung: wird nach jedem Durchlauf
//    ausgefuehrt und erhoeht i um 1 (Kurzform fuer i = i + 1).
// Der Rumpf wird 10-mal ausgefuehrt (i = 1 bis einschliesslich 10).
// Mit "i < 10" waeren es nur 9 Durchlaeufe - ein klassischer
// Fehler um eins.
// Kernpunkte: Initialisierung / Bedingung / Fortschaltung / 10-mal.

// ============================================================================
//  TEIL 6 - Eingabe von aussen
// ============================================================================

const eingabe = process.argv[2];
const eingabeZahl = Number(eingabe);

console.log("\n=== TEIL 6: Eingabe ===");
if (eingabe === undefined) {
  console.log("Keine Zahl uebergeben. Starte z. B. mit: node
grundrechenarten.js 8");
} else if (Number.isNaN(eingabeZahl)) {
  console.log(`"${eingabe}" ist keine gueltige Zahl.`);
} else {
  console.log(`Deine Zahl:        ${eingabeZahl}`);
```

```
  console.log(`Verdoppelt:        ${multiplizieren(eingabeZahl, 2)}`);
  console.log(`Halbiert:          ${dividieren(eingabeZahl, 2)}`);
  console.log(`Um 10 vermindert:  ${subtrahieren(eingabeZahl, 10)}`);
}

// --- AUFGABE 10 ------------------------------------------------------
// TODO: Warum ist Number(eingabe) noetig?
// Antwort:
// process.argv enthaelt die Kommandozeilenargumente, und zwar immer als
// Strings - "8" ist Text, nicht die Zahl 8. Number() wandelt den Text
// in eine Zahl um; laesst sich der Text nicht deuten, entsteht NaN
// ("not a number"), was mit Number.isNaN() geprueft wird.
// Ohne Umwandlung verhaelt sich JavaScript unterschiedlich:
//   "8" + 2  ergibt "82"  - der Operator + ist bei einem String die
//                           Verkettung, die Zahl 2 wird zu "2".
//   "8" * 2  ergibt 16    - fuer * gibt es keine String-Bedeutung,
//                           deshalb wandelt JavaScript automatisch um.
// Genau diese Inkonsistenz macht die explizite Umwandlung noetig:
// Sonst funktionieren drei Rechenarten scheinbar, und die Addition
// liefert stillschweigend Unsinn.
// Kernpunkte: argv liefert Strings / "8" + 2 = "82" / "8" * 2 = 16.

// ========================================================================
//  ZUSATZAUFGABE - Musterloesung
// ========================================================================

function potenzieren(basis, exponent) {
  // Startwert 1, weil 1 das neutrale Element der Multiplikation ist.
  let ergebnis = 1;
  // Der Rumpf laeuft "exponent"-mal; jedes Mal wird einmal mit der
  // Basis multipliziert. Bei exponent = 0 laeuft er nie, das Ergebnis
  // bleibt 1 - genau wie es die Mathematik verlangt.
  for (let i = 0; i < exponent; i++) {
    ergebnis = ergebnis * basis;
  }
  return ergebnis;
}

console.log("\n=== ZUSATZ: Potenzieren ===");
console.log(`2 hoch 10 = ${potenzieren(2, 10)}`);
console.log(`5 hoch 0  = ${potenzieren(5, 0)}`);
// Kuerzer geht es mit dem eingebauten Operator ** :
console.log(`Zur Kontrolle mit ** : ${2 ** 10}`);
```

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/de/modul/m288/learningunits/lu01/loesungen/10?rev=1788357604

Last update: **2026/09/02 16:00**