

Unit Tests 2

### **Best Practices**

 Für jedes Modul/jede Datei im Quellcode erstellen wir eine zugehörige Test-Datei Beispiel: apfel.js + apfel.test.js

Aussagekräftige TestNamen!



```
test("Test 433"), () => {};
test("Error"), () => {};
test("Leerer Array"), () => {};
```



```
test("Summe von 100 und 333 ist 433"), () =>{ }
test("Erstellen von User ohne Passwort wirft Error"),
test("Leerer Array ist sortiert"), () =>{ }
```



## **Best Practices**

#### Lesbare Test

- Aussagekräftige Namen (auch für Variablen)
- Lesbarkeit > Codedichte
   => Code lieber aufteilen als
   alles in eine Zeile zu quetschen
- Gute Praxis: Arrange Act Assert

```
test("adds 1 + 2 to equal 3", () => {
  //Arrange
  const [a, b] = [1, 2];
  //Act
  const result = calc.sum(a, b);
  //Assert (bzw. Expect)
  expect(result).toBe(3);
```



## Arrange Act Assert

- Arrange: Test Setup (z.B. Daten vorbereiten)
- Act:
   Das getestete Verhalten ausführen.
- Assert (in Jest: Expect)
   Erwartetes Verhalten überprüfen

```
test("adds 1 + 2 to equal 3", () => {
  //Arrange
  const [a, b] = [1, 2];
  //Act
  const result = calc.sum(a, b);
  //Assert (bzw. Expect)
  expect(result).toBe(3);
```



## Jest - it und test

Die Funktionen it und test sind synonym.

Heißt also:

```
test("Die Summe von 2 + 5 ist 7", () \Rightarrow {
```

Ist gleichbedeutend mit:

```
it("Die Summe von 2 + 5 ist 7", () => {
```



## Jest - it und test

#### Warum gibt es solche Synonyme?

Dazu können wir diese zwei Tests betrachten, die exakt dasselbe testen sollen:

=> Funktionsnamen sollen sich möglichst gut mit dem restlichen Code einfügen, um lesbaren Code zu erzeugen.



Siehe auch: expect(result).toEqual(7);

## Jest - describe

- Die describe-Funktion lässt uns mehrere Tests gruppieren
- Nützlich um eine Test-Datei mit vielen Tests zu organisieren
- describe darf beliebig oft verschachtelt in anderen describe-Aufrufen sein



### Jest - describe

#### Beispiel:

Wir haben folgende verschachtelte Struktur in unserer \*.test.js:

```
describe("describe nr 1", () => {
  describe("describe nr 2", () => {
    test("test name", () => {
```

Der Test schlägt fehl; wir erhalten folgende Ausgabe von *Jest*:

```
    describe nr 1 > describe nr 2 > test name
```

Bietet sich wunderbar an um verschiedene Test-Cases für dieselbe Funktion zu schachteln (z.B. "Berechnung", "Input-Validation", ...)

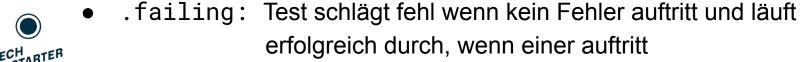


```
describe("sumOfNumbers", () => {
  describe("Berechnung", () => {
    test("2 + 5 = 7", () => {
```

## Jest - Suffixe

Wir können an test und describe einige Suffixe anhängen in der Form: test.<suffix>("testname", () => {}) describe.<suffix>("testname", () => {})

- .only: Führt nur diesen Test / die Test in diesem Block aus
- .skip: Führt diesen Test / Block nicht aus
- . todo: Platzhalter mit Namen für zukünftig implementierten Test.





## Jest - *before* und *after*

Testing Frameworks wie *Jest* bieten in der Regel Funktionen, die wir vor und nach Tests ausführen können. Dies hilft unter Anderem beim Isolieren der Tests!

- Vor oder nach einzelnen Tests "aufräumen" oder Vorbereitungen treffen Mit: beforeEach, afterEach
   => z.B. Datenstrukturen leeren / wieder füllen
- Vor oder nach allen Tests "aufräumen" oder Vorbereitungen treffen Mit: beforeAll, afterAll
  - =>z.B. Datenbankverbindung aufbauen, Datenbank säubern



```
afterEach(() => {
  cleanUpDatabase(globalDatabase);
});
```

## Wann schlägt ein Test fehl?

- Wenn ein Fehler fliegt
  - Jest führt trotzdem die weiteren Tests noch aus!
     (Programm stürzt nicht ab)
- In Javascript heißen Fehler: Error
   In anderen Programmiersprachen oft auch: Exception



## Wie gut kennt ihr euch schon aus?

# Wozu dienen die folgenden Begriffe?:

- Try
- Catch
- Finally
- Throw



# Welche JavaScript-Fehler gibt es?

#### • SyntaxError:

Syntaktischer Fehler im Code (z.B. Klammer vergessen)

#### • TypeError:

Anderen Datentyp verwendet als erwartet

#### RangeError:

Wenn ein Parameter nicht in den vordefinierten Bereich einer Funktion passt



#### • ReferenceError:

Wenn eine Variable (die Referenz) nicht definiert ist

# Wie geh ich mit Errors um?

- Unbehandelte Errors bringen das Programm zum abstürzen
- Mit try, catch und finally können wir unseren Code davor bewahren, abzustürzen
  - => wir erwarten eventuelle Fehler und geben an, wie mit diesen umgegangen werden soll!
- Try: enthält den Code der fehlschlagen kann
- Catch: enthält, was im Falle des fehlschlags passiert



• Finally: enthält Code der so oder so ausgeführt wird (clean up)

## Wie erzeuge ich manuell Errors?

- Wir können mit dem Keyword throw selbst Fehler werfen
- Hinter throw schreiben wir entweder nur eine Nachricht, oder den Fehlertyp (mit oder ohne Nachricht)
- => Diese (und sonst alle anderen) Fehler können wir auch in Tests erwarten!



## Wie teste ich Fehlerverhalten?

- Jest bietet beim expecten die Funktion toThrow()
- Wir müssen hier dem Expect eine Funktion mitgeben und können dann mit .toThrow(TypeError) beispielsweise testen, ob die Funktion einen TypeError wirft

```
expect(() => {
   sumOfNumbers("Hallo", 5);
}).toThrow(TypeError);
```

