Lernskript: JUnit 5 & Grundlagen des Testens

Automatisierte Tests sind der Grundpfeiler moderner, robuster Softwareentwicklung. Sie geben schnelles Feedback, schaffen Vertrauen und dienen als Sicherheitsnetz bei Code-Änderungen. JUnit 5 ist das Standard-Framework für das Testen von Java-Anwendungen.

Teil 1: Die grundlegende Analogie - Das "Warum"

- **Konzept:** Anstatt eine komplexe Anwendung manuell zu testen (wie eine ganze Uhr zusammenzubauen), testet man jede kleinste Komponente (jedes Zahnrad) isoliert mit einer präzisen Test-Maschine (einem JUnit-Test).
- **Vorteile:**
    1. **Schnelles Feedback:** Fehler in kleinen Einheiten werden sofort gefunden.
    2. **Vertrauen (Confidence):** "Grüne" Tests geben die Sicherheit, dass der Code funktioniert.
    3. **Sicherheitsnetz (Regression Testing):** Nach Änderungen kann man alle Tests erneut ausführen und sofort sehen, ob man versehentlich etwas Bestehendes kaputt gemacht hat.

Teil 2: Praktische Anwendungsfälle - Das "Wofür"

Use Case 1: Der Standard-Test (Arrange, Act, Assert)

Ein gut strukturierter Test folgt immer dem **Arrange, Act, Assert**-Muster.

- `@Test`**:** Die Annotation, die eine Methode als ausführbaren Testfall markiert.
- **Code-Beispiel:**

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class RechnerTest {
    @Test
    void testAddition() {
        // 1. Arrange (Vorbereiten): Objekte und Daten erstellen.
        Rechner rechner = new Rechner();
        int erwartetesErgebnis = 8;

        // 2. Act (Aktion): Die zu testende Methode aufrufen.
        int tatsaechlichesErgebnis = rechner.add(5, 3);

        // 3. Assert (Überprüfen): Das Ergebnis mit der Erwartung
vergleichen.
        assertEquals(erwartetesErgebnis, tatsaechlichesErgebnis);
    }
}
```

Use Case 2: Testen von Grenzfällen (Edge Cases)

Ein guter Entwickler testet nicht nur den "Happy Path", sondern auch Sonderfälle wie negative Zahlen, Null, leere Strings etc.

```
Lösung: Für jeden wichtigen Grenzfall wird eine eigene, dedizierte @Test-Methode
geschrieben.
```

Use Case 3: Testen auf erwartete Fehler (Exceptions)

```
Problem: Wie testet man, ob eine Methode bei ungültiger Eingabe wie erwartet einen
Fehler (eine Exception) wirft?

Lösung: JUnit 5 bietet die assertThrows-Methode.

Code-Beispiel:
Generated java
```

@Test void testDivisionDurchNullWirftException() { Taschenrechner rechner = new Taschenrechner(); // Der Test ist ERFOLGREICH, wenn der Code im Lambda eine Exception wirft. assertThrows(IllegalArgumentException.class, () -> { rechner.dividiere(10, 0); }); }

Teil 3: Vertiefung (JavaMasta's Profi-Tipps)

```
Test-Benennung: Testmethoden sollten lange, beschreibende Namen haben, die den
Testfall klar kommunizieren (z.B. add_sollteZweiPositiveZahlenKorrektAddieren).

Setup & Teardown Annotationen:

    @BeforeEach: Wird vor jedem einzelnen Test ausgeführt. Ideal für wiederholtes
Setup.

    @AfterEach: Wird nach jedem einzelnen Test ausgeführt. Ideal zum Aufräumen.

    @BeforeAll / @AfterAll: Werden nur einmal pro Klasse ausgeführt. Müssen static
sein.

Assertions: Die Assertions-Klasse ist dein wichtigstes Werkzeug. Wichtige Methoden
sind assertEquals, assertTrue, assertFalse, assertNotNull.

Was man NICHT testet: Du testest deine eigene Logik, nicht die von Java. Du musst
nicht prüfen, ob ArrayList.add() funktioniert, sondern wie dein Code die ArrayList
benutzt.
```