Lernskript: `abstract class` vs. `interface`

Beide Konzepte dienen der **Abstraktion**, können nicht direkt mit `new` instanziiert werden und definieren Verträge für andere Klassen. Der Unterschied liegt in ihrer Absicht und ihren Fähigkeiten.

Teil 1: Die grundlegende Analogie - Das "Warum"

- `abstract class` **(Die unvollständige Vorlage):**
  - **Analogie:** Ein detaillierter, aber bewusst unvollständiger **Grundbauplan** für ein "Fahrzeug". Er definiert bereits **gemeinsame Eigenschaften** (Zustand, z.B. `int anzahlRaeder`) und **gemeinsames, funktionierendes Verhalten** (implementierte Methoden, z.B. `transportieren()`). Er kann aber auch `abstract`-Methoden ohne Implementierung enthalten, die Subklassen ausfüllen müssen.
  - **Kernidee:** Ideal für eine **"ist-ein"-Beziehung** (`Hund` ist ein `Tier`), bei der Subklassen viel **gemeinsamen Code und Zustand** teilen. Eine Klasse kann nur von **einer** abstrakten Klasse erben (`extends`).
- `interface` **(Der reine Fähigkeits-Vertrag):**
  - **Analogie:** Ein **Zertifikat für eine Fähigkeit** (z.B. "Fliegbar"). Es schreibt nur vor, *was* eine Klasse können muss (z.B. `starten()`, `fliegen()`), aber nicht *wie*.
  - **Kernidee:** Ideal für eine **"kann-ein"-Beziehung**. Es beschreibt, *was* eine Klasse tun kann, nicht *was* sie ist. Völlig unterschiedliche Klassen (`Vogel`, `Flugzeug`) können dieselbe Fähigkeit implementieren. Eine Klasse kann **viele** Interfaces implementieren (`implements`).

Teil 2: Praktische Anwendungsfälle - Das "Wofür"

Use Case 1: Die Tierhierarchie (`abstract class`)

- **Problem:** Eine Hierarchie von Tieren modellieren, die alle einen gemeinsamen Zustand (z.B. `name`) und gemeinsames Verhalten (z.B. `essen()`) haben.
- **Lösung:** Eine `abstract class Tier`, die diese gemeinsamen Mitglieder bereits implementiert und nur die tier-spezifischen Methoden (z.B. `geraeuschMachen()`) als `abstract` deklariert.

Use Case 2: Speicherbare Objekte (`interface`)

- **Problem:** Verschiedene, unzusammenhängende Klassen (`Benutzer`, `Rechnung`) sollen alle die Fähigkeit erhalten, sich zu speichern.
- **Lösung:** Ein `interface Speicherbar` mit der Methode `speichern()`, das von allen relevanten Klassen implementiert wird.

Use Case 3: Moderne Erweiterbarkeit (`default`-Methoden)

- **Problem (Klassisch):** Einem Interface eine neue Methode hinzuzufügen, würde erfordern, alle implementierenden Klassen anzupassen.

- **Lösung (Modern, seit Java 8):** Eine `default`-Methode im Interface deklarieren. Sie bietet eine Standard-Implementierung, sodass bestehender Code nicht bricht.

Teil 3: Wichtige Regeln & Vertiefung (Profi-Tipps)

- **Faustregel:** Beginne, wenn möglich, immer mit einem `interface`. Wähle eine `abstract class` nur dann, wenn du merkst, dass du viel gemeinsamen Code und **Zustand (Felder)** zwischen den Subklassen teilen musst.
- **"Marker Interfaces":** Leere Interfaces (z.B. `java.io.Serializable`) dienen nur als "Etikett", um einer Klasse eine bestimmte Eigenschaft zu geben.
- **Private Methoden in Interfaces (seit Java 9):** Erlauben es, gemeinsame Logik innerhalb von `default`-Methoden in eine private Hilfsmethode auszulagern, um das Interface selbst sauber zu halten.