# Einführung in die Objektorientierte Programmierung
# Vorlesung 11: Abstrakte Klassen & Interfaces

Sebastian Küpper

# Abstrakte Einheiten

- Nicht alle Klassen sind notwendigerweise sinnvoll instanziierbar
- Beispiel: Klasse `Tier` hat Methode `macheLaut()`, die den charakteristischen Laut des Tiers ausgibt. Welchen Laut macht ein (allgemeines) Tier?
- Außerdem können auch zwei abstrakte Prototypen logisch einer Klasse vorangestellt sein: Eine Schnabeltier ist z.B. sowohl ein Eierleger als auch ein Säugetier

# Abstrakte Einheiten

- Nicht alle Klassen sind notwendigerweise sinnvoll instanziierbar
- Beispiel: Klasse `Tier` hat Methode `macheLaut()`, die den charakteristischen Laut des Tiers ausgibt. Welchen Laut macht ein (allgemeines) Tier?
- Außerdem können auch zwei abstrakte Prototypen logisch einer Klasse vorangestellt sein: Eine Schnabeltier ist z.B. sowohl ein Eierleger als auch ein Säugetier
- Java bietet zwei Optionen für abstrakte Einheiten:
  - abstrakte Klassen
  - Schnittstellen (Interfaces)

# Wozu abstrakte Einheiten?

- Implementation gemeinsamer Methoden und Deklaration konzeptuell gleicher Methoden in Oberklasse
- Verwaltung in typsicheren Datenstrukturen wie Feldern (statt `Object`-Array)

# Abstrakte Klasse

- Eine Klasse ist abstrakt, wenn ihr das Schlüsselwort `abstract` vorangestellt wird.
- Abstrakte Klassen sind nicht instanziierbar

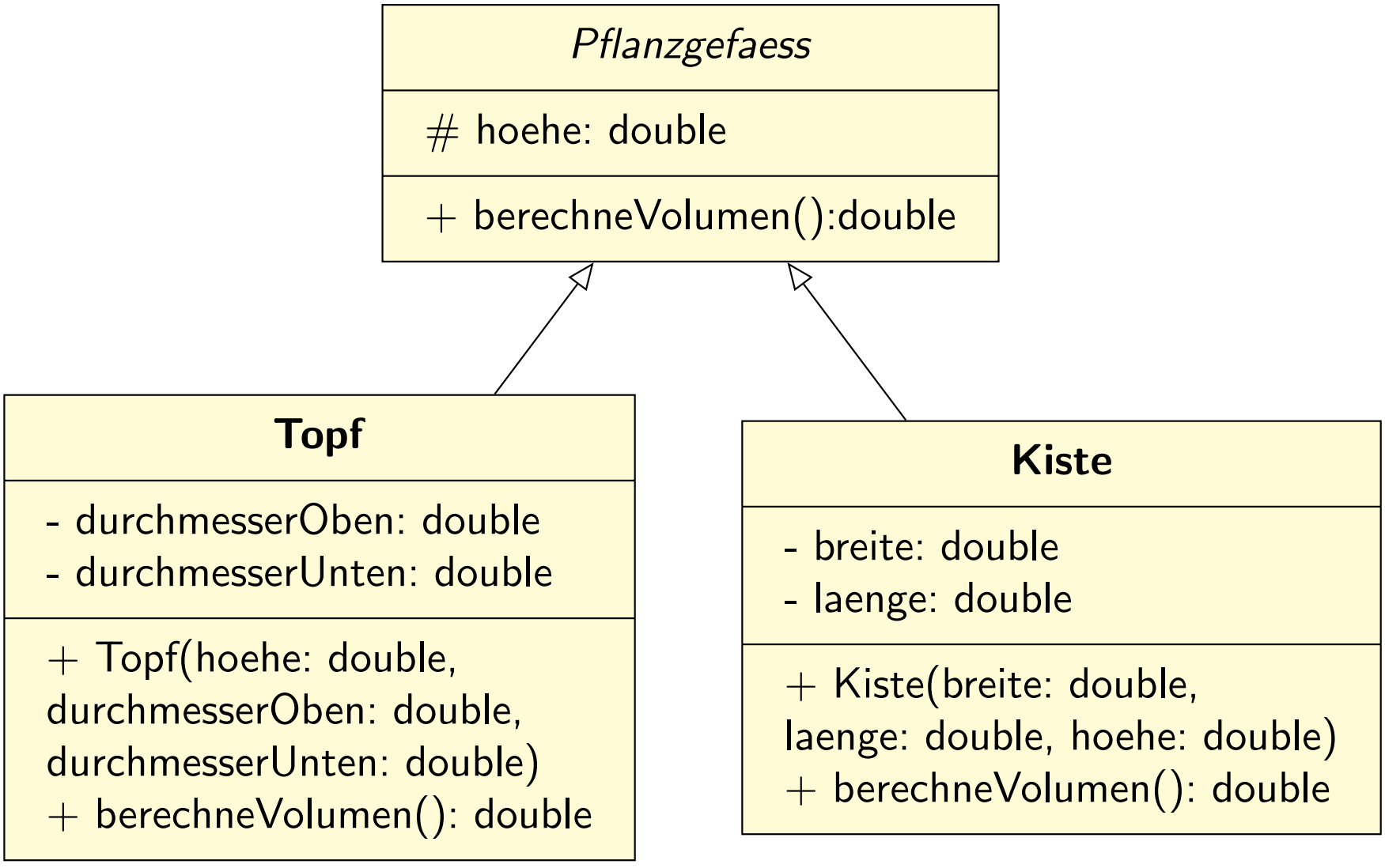
# Abstrakte Klasse

- Eine Klasse ist abstrakt, wenn ihr das Schlüsselwort `abstract` vorangestellt wird.
- Abstrakte Klassen sind nicht instanziierbar
- Abstrakte Klassen können abstrakte Methoden enthalten, die nur aus dem Schlüsselwort `abstract` und einer Signatur bestehen
- Konkrete Kindklassen von abstrakten Klassen müssen alle abstrakten Methoden implementieren.

# Abstrakte Klasse

- Eine Klasse ist abstrakt, wenn ihr das Schlüsselwort `abstract` vorangestellt wird.
- Abstrakte Klassen sind nicht instanziierbar
- Abstrakte Klassen können abstrakte Methoden enthalten, die nur aus dem Schlüsselwort `abstract` und einer Signatur bestehen
- Konkrete Kindklassen von abstrakten Klassen müssen alle abstrakten Methoden implementieren.
- In UML mit Stereotyp <<`abstract`>> oder durch kursive Schrift gekennzeichnet

# Beispiel: Abstrakte Klasse in UML



# Beispiel abstrakte Klasse in Java

Wir betrachten eine mögliche Implementierung der abstrakten Klasse:

```
public abstract class Pflanzgefaess extends Artikel {
   protected double hoehe;
   protected final double PI = 3.14159265;
   public abstract double berechneVolumen();
}
```

# Beispiel abstrakte Klasse in Java

So kann beispielsweise die hiervon abgeleitete Klasse Topf aussehen:

```
public class Topf extends Pflanzgefaess {
   private double durchmesserOben;
   private double durchmesserUnten;
   public Topf (double h, double dOben, double dUnten) {
      this.hoehe = h;
      this.durchmesserOben = dOben;
      this.durchmesserUnten = dUnten;
   }
   public double berechneVolumen() {
      double radOben = durchmesserOben / 2;
      double radUnten = durchmesserUnten / 2;
      return (radOben * radOben + radOben * radUnten +
          radUnten * radUnten) * hoehe * PI / 3;
   }
}
```

# Beispiel abstrakte Klasse in Java

und die Klasse `Kiste`:

```
public class Kiste extends Pflanzgefaess {
   private double breite;
   private double laenge;
   public Kiste(double hoehe, double breite, double
       laenge) {
      this.hoehe = hoehe;
      this.breite = breite;
      this.laenge = laenge;
   }
   public double berechneVolumen() {
      return laenge * breite * hoehe;
   }
}
```

# Anwendung abstrakter Klassen

- Abstrakte Klassen können nicht instanziiert werden, die Zeile `Pflanzgefaess p = new Pflanzgefaess();` würde also nicht kompilieren. Meldung: `error: Pflanzgefaess is abstract; cannot be instantiated`

# Anwendung abstrakter Klassen

- Abstrakte Klassen können nicht instanziiert werden, die Zeile `Pflanzgefaess p = new Pflanzgefaess();` würde also nicht kompilieren. Meldung: `error: Pflanzgefaess is abstract; cannot be instantiated`
- Dennoch können abstrakte Klassen Konstruktoren enthalten – diese können nicht abstrakt sein. Hilfreich für den Aufruf mit `super` von Oberklasse aus.

# Anwendung abstrakter Klassen

```
public abstract class Pflanzgefaess extends Artikel{
   ...
   public Pflanzgefaess(double hoehe) {
      this.hoehe = hoehe;
   }
   ...
}
public class Kiste extends Pflanzgefaess {
   ...
   public Kiste(double hoehe, double breite, double
       laenge) {
      super(hoehe);
      this.breite = breite;
      this.laenge = laenge;
   }
   ...
}
```

# Anwendung abstrakter Klassen

- Hingegen kann man Pflanzgefaess als Typ verwenden, was besonders für Sammlungen hilfreich ist:

```
Pflanzgefaess[] p = new Pflanzgefaess[2];
p[0] = new Topf(1,2,3);
p[1] = new Kiste(1,2,3);
for(int i = 0; i < p.length(); ++i)
   System.out.println(p[i].berechneVolumen());
```

ergibt beispielsweise die Ausgabe:

```
4.9741883625000005
6.0
```

# Einschränkungen abstrakter Klassen

- Methode kann nicht gleichzeitig abstrakt und final sein
- Klasse kann nicht gleichzeitig abstrakt und final sein
- Methode kann nicht privat und abstrakt sein

Grund?

# Einschränkungen abstrakter Klassen

- Methode kann nicht gleichzeitig abstrakt und final sein
- Klasse kann nicht gleichzeitig abstrakt und final sein
- Methode kann nicht privat und abstrakt sein

Grund?

Eine solche abstrakte Klasse könnte in keinem Programm sinnvoll verwendet werden.

# Einschränkungen abstrakter Klassen

- Methode kann nicht gleichzeitig abstrakt und final sein
- Klasse kann nicht gleichzeitig abstrakt und final sein
- Methode kann nicht privat und abstrakt sein

Grund?

Eine solche abstrakte Klasse könnte in keinem Programm sinnvoll verwendet werden.

- Jede Klasse kann nur von einer abstrakten Klasse erben

Restriktion für Klassen im Allgemeinen.

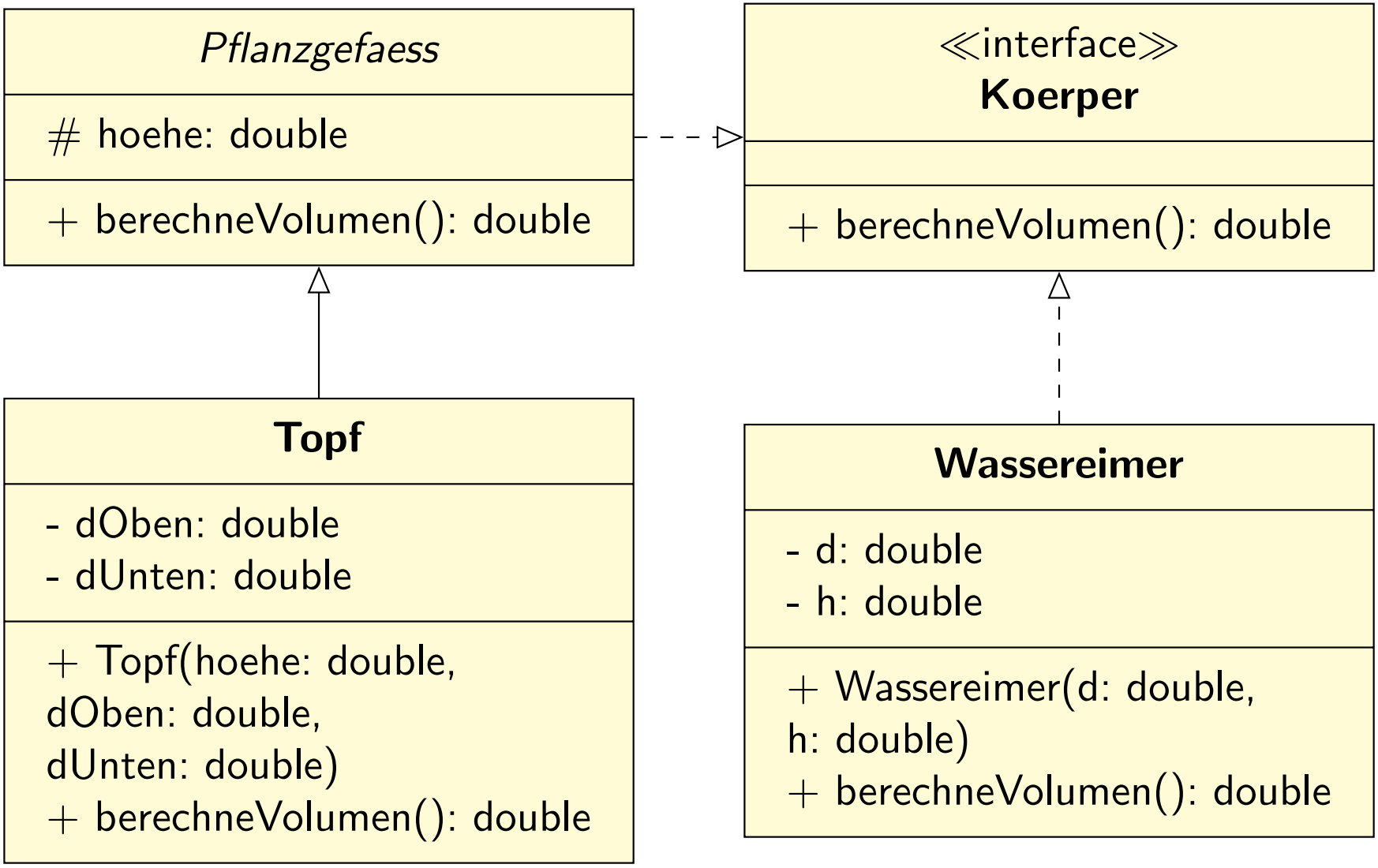
# Interfaces / Schnittstellen

- Alternative zu abstrakten Klassen
- Vorteil: Mehrere Interfaces können von einer Klasse implementiert werden
- Nachteil: Keine Objektattribute, d.h. Implementationen von Methoden sind eingeschränkt.
- Aber: Durch Vorsehen von Gettern und Settern kann das umgangen werden

# Schlüsselwörter für Interfaces

- Werden definiert wie Klassen; statt `class`-Schlüsselwort wird `interface` verwendet.
- Statt `extends` wird `implements` verwendet; bei mehreren Interfaces: Durch Komma getrennt auflisten.
- Vererbung zwischen Interfaces mit `extends` (auch mehrere Interfaces)
- Standardfall ist dass Methode abstrakt ist, bedarf keines Schlüsselwortes
- Möchte man hingegen Implementation angeben, muss Schlüsselwort `default` vorangesetzt werden.
- In UML: Mit Stereotyp <<`interface`>> gekennzeichnet; Implementation mit gestricheltem Ableitungspfeil

# Beispiel: Interfaces in UML



# Anwendung von Interfaces

Es ergibt sich der folgende Code für `Koerper`:

```
public interface Koerper {
   double berechneVolumen();
}
```

Oder, falls wir eine einfache Standardimplementierung wünschen:

```
public interface Koerper {
   default double berechneVolumen(){
      return 0;
   }
}
```

# Anwendung von Interfaces

Dann ist eine mögliche Implementation für `Wassereimer`:

```
public class Wassereimer implements Koerper {
   private double d;
   private double h;
   public Wassereimer(double d, double h){
      this.d = d;
      this.h = h;
   }
   public double berechneVolumen(){
      return 1.57079633*d*h;
   }
}
```

# Fragen zur Vorlesungseinheit

1. Was ist der Unterschied zwischen abstrakten Klassen und Schnittstellen?
2. Wozu gibt es abstrakte Methoden?
3. Wie geht Java mit Methoden um, die von zwei verschiedenen, unabhängigen Interfaces übernommen wurden und jeweils eine Standard-Implementation bieten?