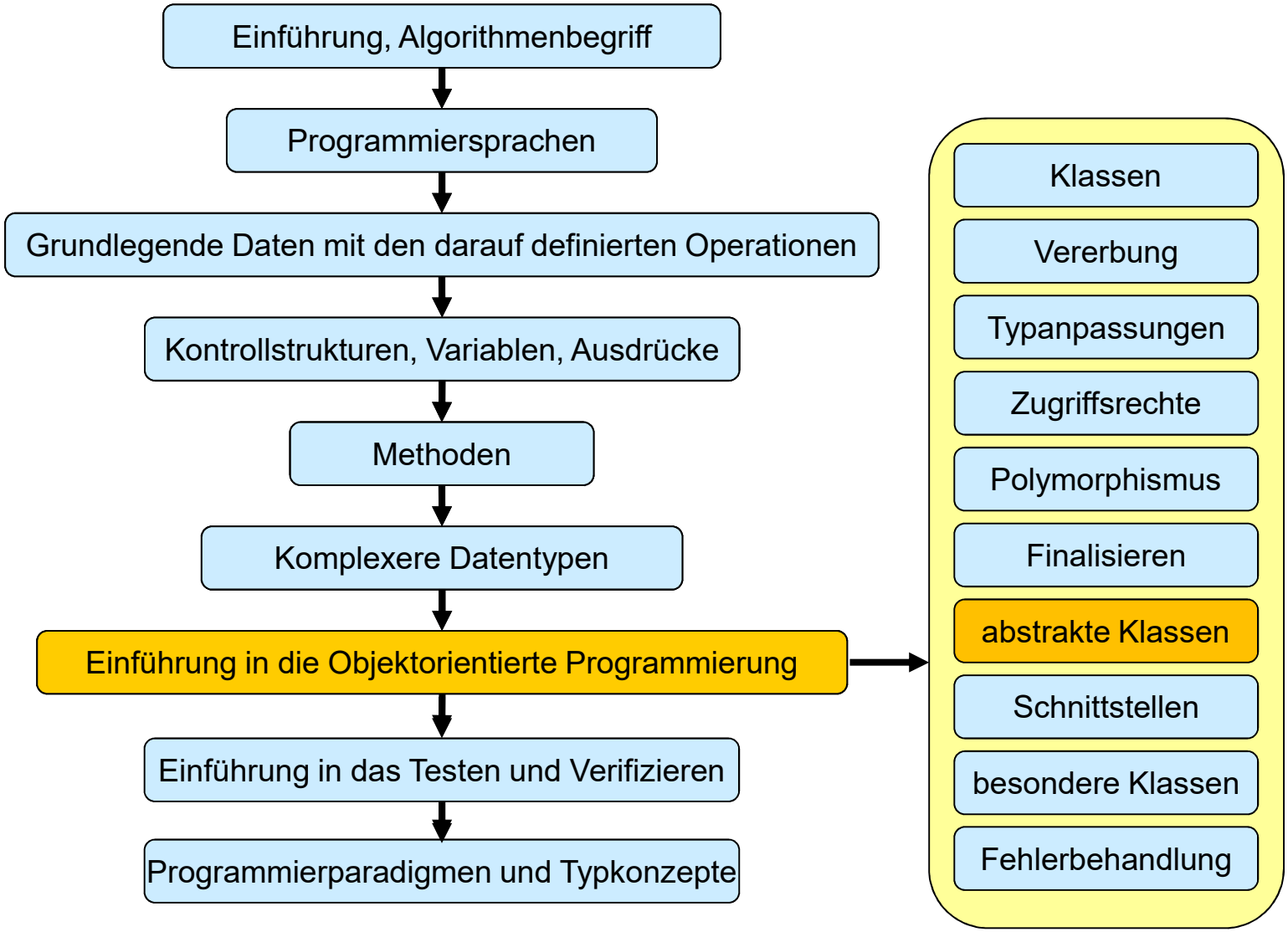
Inhalt dieser Veranstaltung



Notwendigkeit abstrakter Klassen

- Tiere sollen weiter differenziert werden:
  - Ameisen gehören genau einer Kaste an: Königin, Männchen, Arbeiterin
  - Eulen können fliegen
- Es gibt keine Tiere außer Ameisen und Eulen
- Die Tierklasse dient nur noch als gemeinsame Basisklasse für alle Tierarten und macht für sich alleine keinen Sinn

- Realisierung V1:

```
class Tier {… }  // gemeinsame Basisklasse
class Ameise extends Tier { … }
class Eule   extends Tier { … }
```

- Nachteil dieser Lösung: mit `new Tier()` lässt sich nach wie vor auch ein Tier erzeugen. Diese Möglichkeit möchte man aber verhindern.

- Lösung: Markierung der Tier-Klasse, so dass sich davon keine Instanzen mehr erzeugen lassen

Abstrakte Klassen

- In der Klassendefinition kann man durch das Schlüsselwort `abstract` eine Klasse als nicht instanziierbar deklarieren
- Sie dient dann nur noch als (gemeinsame) Oberklasse, um z.B. gemeinsame Funktionalität dort bereit zu stellen

- Realisierung V2:

```
abstract class Tier {… }  // gemeinsame Basisklasse
class Ameise extends Tier { … }
class Eule   extends Tier { … }
```

- Ein `new Tier()` würde jetzt zu einem Compiler-Fehler führen

Eigenschaften abstrakter Klassen

- Eine mit `abstract` gekennzeichnete Klasse nennt man abstrakte Klasse, eine instanziierbare Klasse eine konkrete Klasse

- Abstrakte Klassen können selbst von abstrakten oder konkreten Klassen erben und umgekehrt auch

- In komplexen Ableitungshierarchien kann es Sinn machen Konstruktoren auch in abstrakten Klassen zu definieren, um so (geeignete) Konstruktoren in Oberklassen aufrufen zu können

- Auch wenn zu einer abstrakten Klasse sich keine Instanz erzeugt lässt, ist dies ein Referenztyp und es lassen sich z.B. Referenzvariablen dieses Typs anlegen (z.B. zur Nutzung des Substitutionsprinzips)
- Beispiel:

```
Tier[] meineTiere = new Tier[2];
meineTiere[0] = new Ameise();
meineTiere[1] = new Eule();
System.out.println(meineTiere[0].toString());
System.out.println(meineTiere[1].toString());
```

Notwendigkeit abstrakter Methoden

- Man kann auch eine oder mehrere Methoden in einer Klasse als `abstract` markieren
- Dann **muss** man auch die Klasse als `abstract` kennzeichnen (und diese ist damit wiederum nicht instanziierbar)

- Notation für abstrakte Methoden: im Methodenkopf zusätzlich `abstract` und statt des Methodenrumpfs (ein Block) lediglich ein Semikolon

- Beispiel: `abstract void bewege();`

- Bedeutung und Nutzung (**deutlich anders** als bei rein abstrakten Klassen!!!):
  - Hiermit wird spezifiziert, dass man eine Methode mit genau dieser Signatur **vorgibt**
  - In einer konkreten Klasse, die von der abstrakten Klasse direkt oder indirekt abgeleitet ist, **muss** irgendwo in der Ableitungshierarchie die Methode realisiert sein

Beispiel

- Jedes Tier (Ameise, Eule) soll sich bewegen können
- Wir wollen einheitlich in der Basisklasse **vorgeben**, wie diese Funktionalität angesprochen werden kann: `void bewegen()`
- Damit **zwingen** wir jede konkrete abgeleitete Klasse dazu, dass diese Methode mit genau dieser Signatur auch existiert
- Substitutionsprinzip: wir garantieren damit, dass sich jedes Tier jetzt bewegen kann, über eine einheitlich definierte Methode

```
// Klasse muss abstract sein
abstract class Tier {
  …
  // Vorgabe fuer Methode
  abstract void bewegen();
}
```

```
// konkrete Klasse (Eule analog)
class Ameise extends Tier {
  …
  // Methode nach Vorgabe realisiert
  void bewegen() { … }
}
```

```
Tier[] meineTiere = new Tier[2];
meineTiere[0] = new Ameise();
meineTiere[1] = new Eule();

/* hier ist jetzt garantiert, dass
   jedes konkrete Tier eine einheitlich
   anzusprechende bewege-Methode
   besitzt, die aber nicht in Tier
   realisiert ist / sein muss
*/
meineTiere[0].bewegen();
meineTiere[1].bewegen();
```

Eigenschaften abstrakter Methoden

- Unterschied in der praktischen Anwendung von abstrakten Klassen / Methoden:
  - nur abstrakte Klasse: in diese Oberklasse gemeinsame Funktionalität für (viele) abgeleitete Klassen
  - auch abstrakte Methode: Vorgabe für Methoden in abgeleiteten Klassen, wenn man in der Basisklasse z.B. keine sinnvolle gemeinsame Realisierung hat

- Jede Klasse, die aufgrund ihrer Vererbungshierarchie mindestens eine nicht realisierte abstrakte Methode besitzt, ist nicht instanziierbar (also eine abstrakte Klasse)

- Man kann eine abstrakte Klasse X, die selbst eine abstrakte Methode definiert, ableiten von einer konkreten Klasse Y. Y ist instanziierbar, X ist es dann nicht.

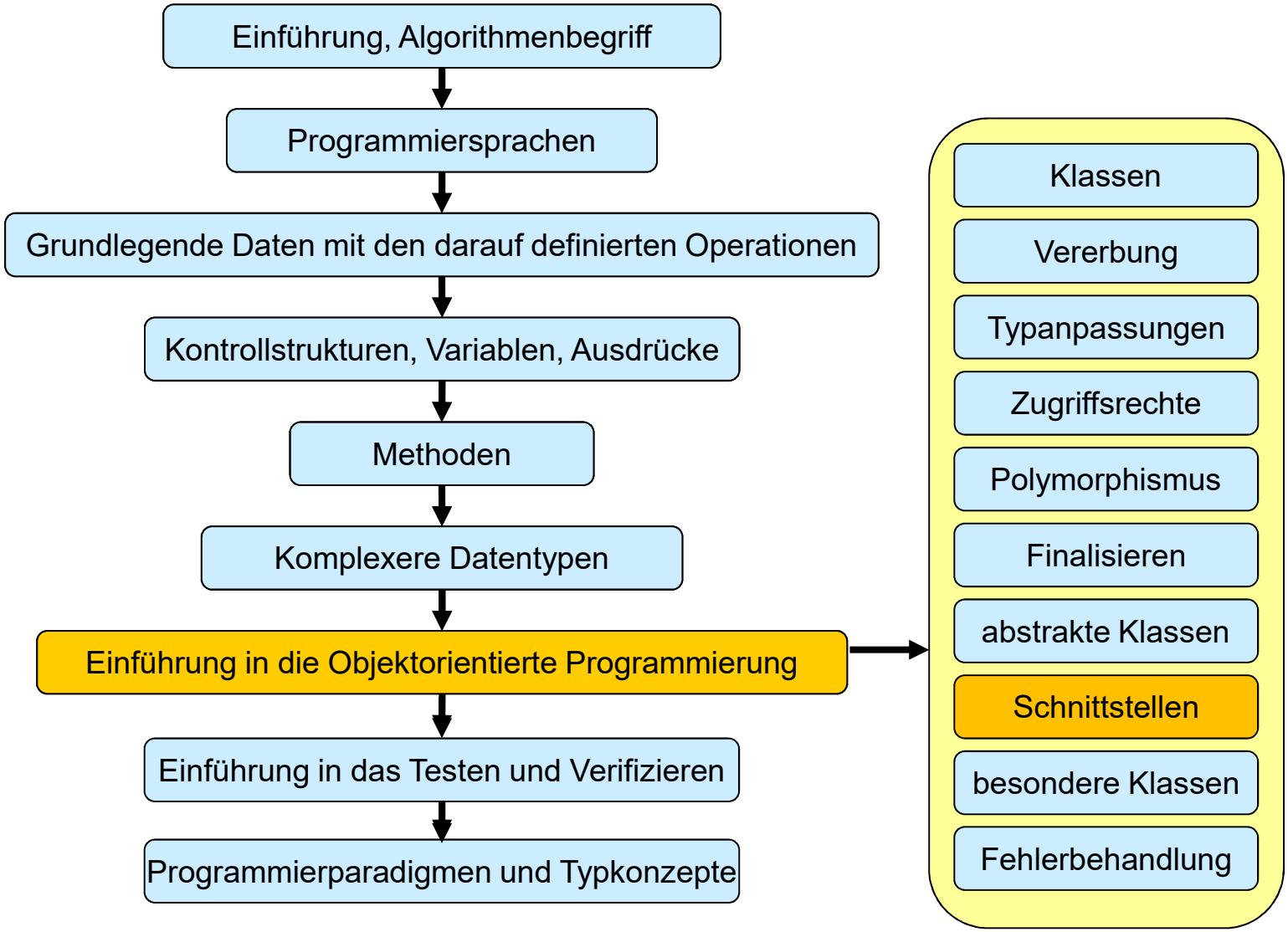
Zwischenstand

- Abstrakte Klassen stellen gemeinsame Funktionalität für mehrere davon abgeleitete Klassen bereit, sollen selbst aber nicht instanziierbar sein
- Abstrakte Methoden machen eine Vorgabe für abgeleitete konkrete Klassen, um eine einheitlich definierte Methode zu garantieren

Reflektion

- Kann man `abstract` mit `final` bei Klassen mischen? Wenn ja, was ist dann die Bedeutung einer solchen Klasse? Wenn nein, wieso nicht?

Inhalt dieser Veranstaltung



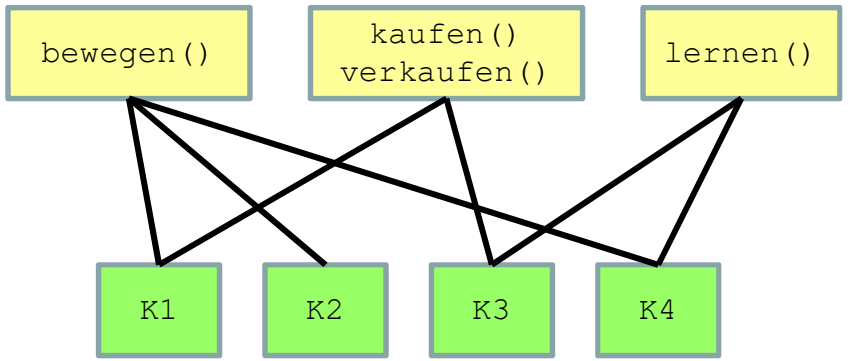
Was geht bis jetzt, was geht nicht?

- Abstrakte Klassen können als Basis weiterer konkreter Klassen dienen
- Abstrakte Methoden geben eine funktionale Sicht vor (beschreiben, was wie zwingend vorhanden sein muss)

- Situation in (großen) Software-Projekten:

Ebene der funktionalen Beschreibung, wäre derzeit über abstrakte Methoden (in abstrakten Klassen) möglich

konkrete Klassen, die jeweils **Teile** der Gesamtfunktionalität realisieren



- Manche Klasse müssen mehrere funktionale (Teil-)Sichten realisieren
- Java kennt aber nur Einfachvererbung → also alles in eine große abstrakte Klasse? Unsinn!
- Denn dann müsste z.B. `K2` auch `lernen()` realisieren, was keinen Sinn macht

Schnittstellen

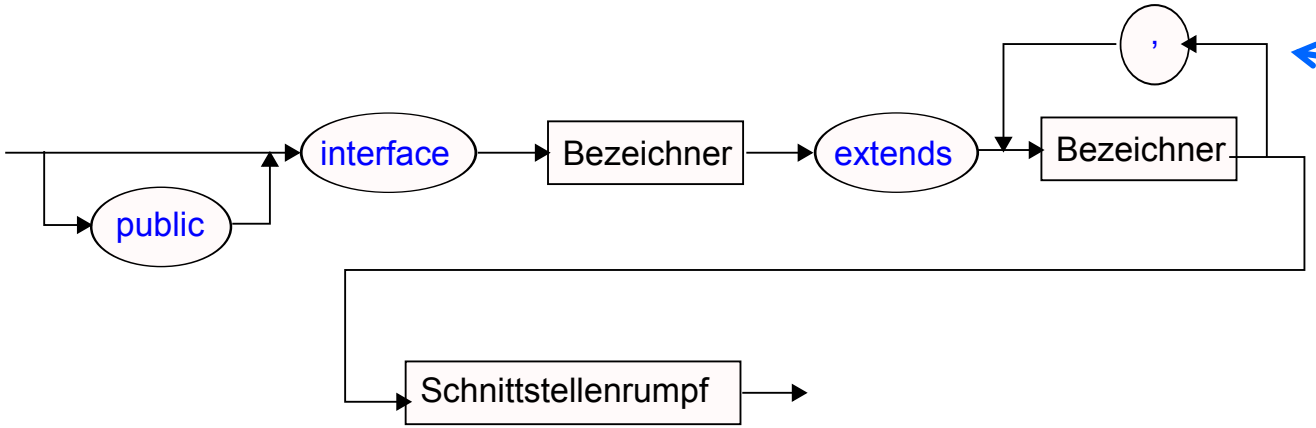
- Schnittstellen in Java beschreiben ausschließlich eine Funktionalität
- Sie enthalten keinerlei Realisierungen (was in abstr. Klasse möglich wäre)
- Sie ähneln damit einer abstrakten Klasse mit ausschließlich abstrakten Methoden

- Aber im Gegensatz zu abstrakten Klassen:
  - Eine Klasse kann mehrere Schnittstellen realisieren
  - Schnittstellen erlauben Mehrfachvererbung bei der Definition von Schnittstellen (nach wie vor gibt es die nicht bei Klassen)
- Schnittstellen ermöglichen eine strikte Trennung der Spezifikation und der Implementierung

- Namenskonvention:
  - Schnittstellen beschreiben eine Funktionalität, wozu jemand in der Lage ist
  - Namen hören deshalb konventionsgemäß mit -able auf (ist in der Lage…)
  - Beispiel: Payable

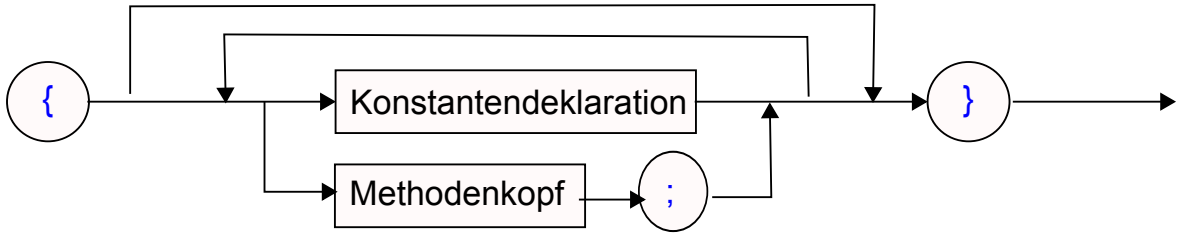
Syntax Schnittstellendefinition

Schnittstellendefinition

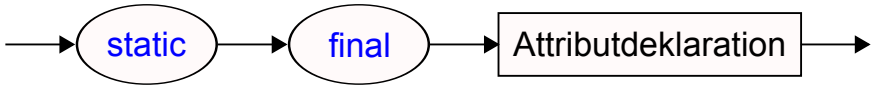


Mehrfachvererbung bei Schnittstellen möglich

Schnittstellenrumpf



Konstantendeklaration



Es sind nur Konstanten erlaubt

Beispiel

- Tiere (und Menschen, Pflanzen,…) können geimpft werden (sind impfbar)

```
/**
 * Schnittstelle Impfable
 */
interface Impfeable {
    public void impfen();
}
```

- Zu Tieren (und Laptops, Autos,…) lässt sich ein Preis/Wert ermitteln

```
/**
 * Schnittstelle zur Ermittlung eines Preises
 */
interface PreisErmitteable {
    double preisErmitteln();
}
```

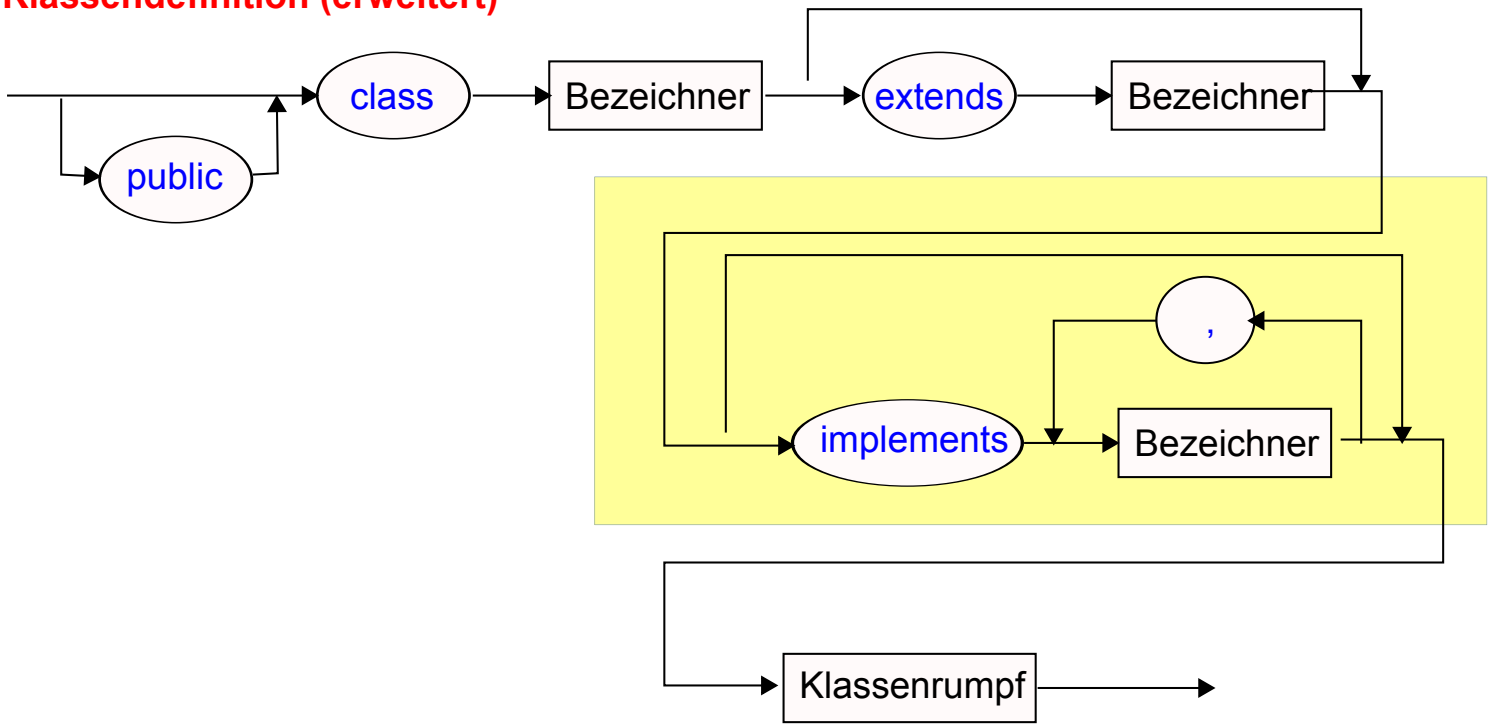
Definition von Schnittstellen

- Eine Schnittstelle ohne `public` hat eine Sichtbarkeit nur innerhalb des eigenen Pakets (siehe Klassen)
- Alle Methoden in einer Schnittstellendefinition haben automatisch die Eigenschaft `abstract` und `public`
- Variablen mit `static final` lassen sich ebenfalls in einer Schnittstelle definieren (und sind somit Konstanten)
- Im Gegensatz zu einer Klasse machen Konstruktoren in Schnittstellen keinen Sinn und sind auch nicht erlaubt
- Eine (`public`) Schnittstelle mit Namen `X` muss wie bei Klassen gewohnt in einer Datei mit dem Namen `X.java` abgelegt sein

Nutzung von Schnittstellen

- Eine Klasse kann jetzt angeben, dass sie eine oder mehrere Schnittstellen realisiert / implementiert
- Dann muss sie auch alle in den Schnittstellen geforderten Methoden realisieren oder die Klasse muss mit `abstract` markiert werden
- Syntax:

Klassendefinition (erweitert)



Beispiel

- Ameise
  - ist von der abstrakten Klasse `Tier` abgeleitet und realisiert deren abstrakte Methode `bewegen()`
  - und implementiert die Schnittstellen zum Impfen und die Schnittstelle zur Preisermittlung

```
public class Ameise extends Tier implements Impfeable,PreisErmitteable {
    …
    // realisiere abstrakte Methode der Oberklasse
    void bewegen() { System.out.println("bewege 6 Beine"); }

    // Schnittstellenmethode realisieren
    public void impfen() { System.out.println("ich werde geimpft"); }

    // Schnittstellenmethode realisieren
    public double preisErmitteln() { return 0.01; } // Ameisen sind billig
}
```

Vererbung bei Schnittstellen

- Schnittstellen können analog zu Klassen auch von anderen Schnittstellen erben
- Die Bedeutung ist ebenfalls analog zu sehen (die abgeleitete Schnittstelle fordert die Realisierung aller eigenen und geerbten Methoden)
- Beispiel:
  - Impfen gibt es als Schluckimpfung und Nadelimpfung
  - Definiere für Nadelimpfungen Schnittstelle `Pieksable`

```
// Impfung mit pieken (Nadel). Vorher muss die Stelle desinfiziert werden.
public interface Piekseable extends Impfeable {
    // vorher muss desinfiziert werden
    public void desinfizieren();
}
```

```
public class Eule extends Tier implements Piekseable,PreisErmitteable {
    // realisiere abstrakte Methode der Oberklasse
    void bewegen() { System.out.println("bewege Fluegel"); }
    public void impfen() { System.out.println("hier mit Nadel"); }
    public void desinfizieren() { System.out.println("wir desinfizieren"); }
    public double preisErmitteln() { return 5000.0; } // Eulen sind teuer
}
```

Mehrfachvererbung bei Schnittstellen

- Besonderheit bei Schnittstellenvererbung: Mehrfachvererbung erlaubt
- Prinzipielle Probleme, die dabei auftreten können, werden hier nicht behandelt

- Beispiel:

```
public interface Verkaeufer {
    public void verkaufen();
}
```

```
public interface MaengelVerschweigen {
    public void verschweigen();
}
```

```
public interface Autoverkaeufer extends Verkaeufer,MaengelVerschweigen {
    public void beSmart();
}
```

Zur Realisierung der Schnittstelle `AutoVerkaeufer` müsste man also insgesamt 3 Methoden realisieren.

Schnittstellen sind Typen

- Mit der Definition einer Schnittstelle wird ein neuer Typ definiert
- Eine Schnittstelle kann zwar nicht instanziiert werden, aber kann wie ein ganz normaler Referenztyp verwendet werden
- Es lassen sich z.B. Referenzvariablen dieses Typs anlegen und es ist ein Cast auf diesen Typ möglich

- Beispiel:

```
Ameise a = new Ameise(Ameise.KOENIGIN);
PreisErmitteable p = (PreisErmitteable)a;
System.out.println(p.preisErmitteln());
```

- Eine Ameise realisiert die Schnittstelle `PreisErmitteable`
- Das Objekt, das über `p` angesprochen wird, hat nach dem Cast nur die Funktionalität, die über die Schnittstelle `PreisErmittable` definiert ist.

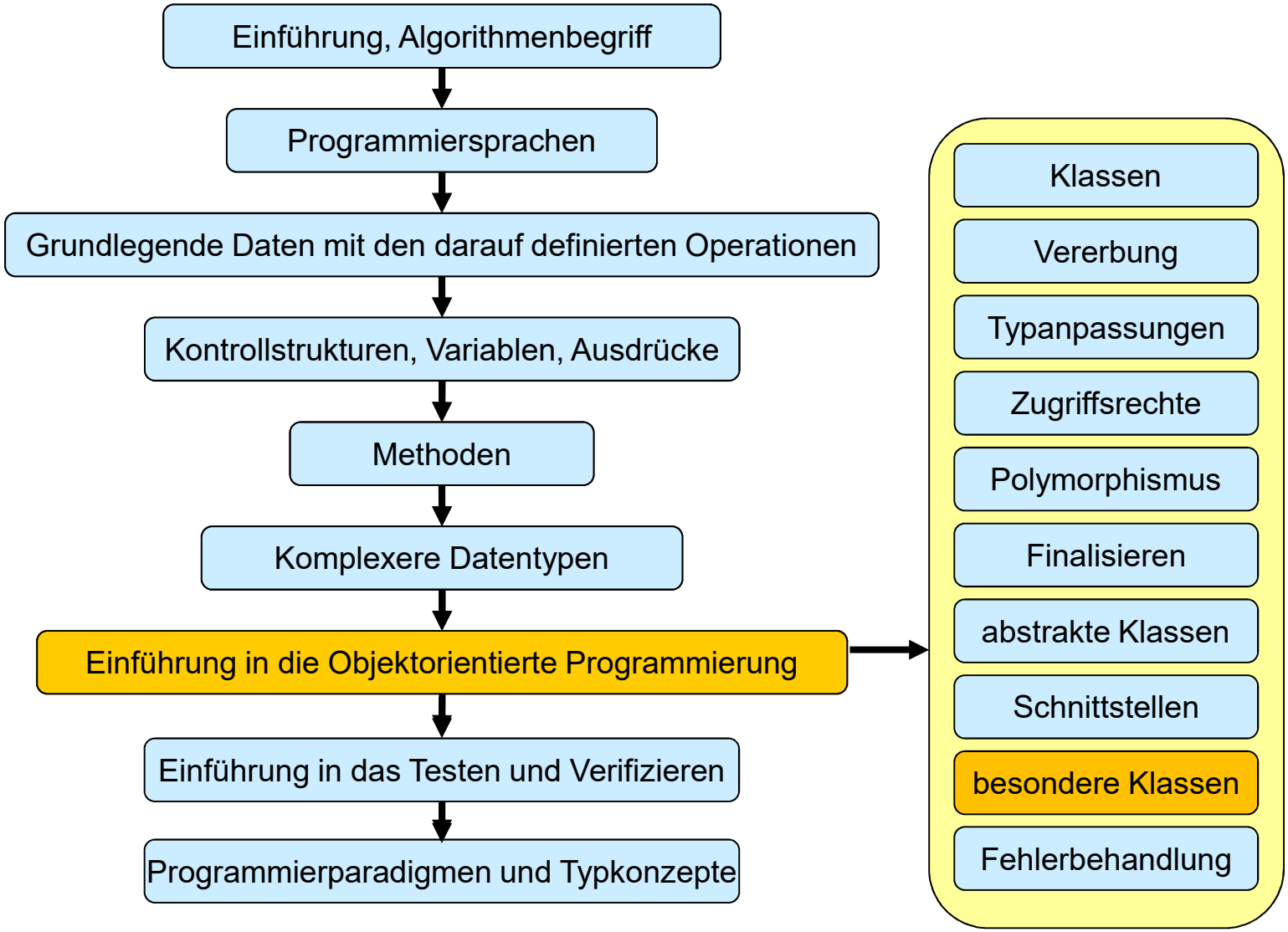
Zwischenstand

- Schnittstellen dienen der Spezifikation von Funktionalität
- Klassen können mehrere Schnittstellen realisieren
- Schnittstellen kennen Mehrfachvererbung
- Schnittstellen sind Referenztypen

Reflektion

- Kann man eine Schnittstelle durch eine abstrakte Klasse mit ausschließlich abstrakten Methoden ersetzen? Immer? Nie? Wenn nur manchmal, wann?
- Diskutieren Sie die Unterschiede / Vorteile von Schnittstellen gegenüber abstrakten Klassen.

Inhalt dieser Veranstaltung



Kleine Hilfsklassen

- In manchen Situationen benötigt man evtl. nur kleine "Hilfsklassen" um z.B. Daten zu kapseln
- Beispiel:

```
class Statistikwerte {
    private double[] werte;
    Statistikwerte(double[] werte) {
        this.werte = werte.clone();
    }
    double mittelwert() {…   }
}
```

- Dazu gibt es drei Möglichkeiten, wie man solche Hilfsklassen definieren kann:
  - innere Klasse
  - lokale Klasse
  - anonyme Klasse

- Die Angabe ist für jede Variante jeweils nur an bestimmten Stellen in einer Programmstruktur möglich

Innere Klasse

- Innerhalb einer Klasse auf Methodenniveau lassen sich innere Klassen definieren
- Einschränkung: in einer inneren Klasse sind Klassenvariablen und Klassenmethoden nicht erlaubt
- In einer solchen inneren Klasse sind alle Attribute und Methoden der umgebenden Klasse bekannt, und auch umgekehrt

Beispiel

```
public class BeispielInnereKlasse {

    private MeinInteger wert;
    private double wert2 = 2;

    // innere Klasse
    class MeinInteger {
        private int meinWert;
        public MeinInteger(int wert)  {
            this.meinWert = wert;
            wert2 = 2*wert;   // hier Zugriff auf Attribut der aeusseren Klasse
        }
        public int getMeinWert()           { return meinWert; }
        public void setMeinWert(int wert) { this.meinWert = wert; }
    }

    // Konstruktor aeussere Klasse
    BeispielInnereKlasse(int wert) {
        // Nutzung innere Klasse
        this.wert = new MeinInteger(wert);
        this.wert.meinWert = 4711;   // nicht schoen, aber prinzipiell moeglich
    }
}
```

Lokale Klasse

- Lokale Klassen lassen sich innerhalb von Blöcken (also damit innerhalb von Methoden) definieren

- In einer lokalen Klasse hat man Zugriff auf:
  - blocklokale Variablen des umschließenden Blocks, die mit `final` deklariert sind
  - innerhalb von Instanzmethoden auf Instanzvariablen und Klassenvariablen der umgebenden Klasse
  - innerhalb von Klassenmethoden auf Klassenvariablen der umgebenden Klasse

Beispiel

```
public class BeispielLokaleKlasse {

  // Methode der aeusseren Klasse
  public void untersuche() {
    // final ist noetig, um Variable in der lokalen Klasse nutzen zu koennen
    final int wert2 = 0;

    // lokale Klasse (Teil des Methodenrumpfs)
    class MeinInteger {
      private int meinWert;
      public MeinInteger(int wert)  {
        this.meinWert = wert * wert2;  // Zugriff auf blocklokale Variable
      }
      public int getMeinWert()           { return meinWert; }
      public void setMeinWert(int wert) { this.meinWert = wert; }
    }

    // weiter im Methodenrumpf
    MeinInteger wert = new MeinInteger(5);
    wert.setMeinWert(4711);
  }
}
```

Anonyme Klasse

- Anonyme Klassen haben keinen Namen und keinen `class` Prefix
- Sie können in einer Deklarationsanweisung oder innerhalb einer Ausdrucksanweisung definiert werden
- Es kann keine Konstruktoren darin geben (weil kein Klassenname vorhanden)
- An der Programmstelle der Klassendefinition wird bereits genau ein Objekt erzeugt (ohne Klassenname wäre dies an anderen Stellen ja auch nicht möglich)

- Einsatzgebiete anonymer Klassen:
  - Existierende Klasse in der Funktionalität erweitern, z.B. durch Überschreiben einer Methode
  - Abstrakte Methoden / Schnittstellen realisieren, ohne eine eigene benannte Klasse angeben zu müssen

Beispiel 1

```
public class BeispielAnonymeKlasse {

 public static void main(String[] args) {
    // Ausdruck
    new MeinInteger(5) {
      // hier ist schon die anonyme Klasse
      // ueberschreibe Methode der Ursprungsklasse,
      // z.B. fuer einen besondere Zweck
      public String toString() { return "anonym: " + this.getMeinWert(); }
    };
  }
}

// weitere Klasse
class MeinInteger {
  private int meinWert;
  public MeinInteger(int wert)       {
    meinWert = wert;
    System.out.println(this);
  }
  public int getMeinWert()           { return meinWert; }
  public void setMeinWert(int wert)  { this.meinWert = wert; }
  public String toString()           { return "Wert ist " + meinWert; }
}
```

Beispiel 2

```
// fuege einen ActionListener zu einem Menu-Eintrag hinzu
menu.addActionListener(new ActionListener() {
        // Durch Druecken des Menu-Punktes wird actionPerformed aufgerufen
        public void actionPerformed(ActionEvent e) {
           ...   // was immer hier zu tun ist
        }
   });
```

- `ActionListener` ist eine Schnittstelle des JDK im Zusammenhang mit der Programmierung grafischer Benutzeroberflächen
- Diese Schnittstelle fordert genau eine Methode: `actionPerformed`
- In der anonymen Klasse wird die geforderte Methode definiert, ohne eine eigene benannte Klasse schreiben zu müssen
- Wiederholung: Schnittstellen könne nicht instanziiert werden, Klassen sehr wohl. Alleine `new ActionListener()` würde also zu einem Fehler führen.

Zwischenstand

- Kleine Hilfsklassen lassen sich als innere, lokale oder anonyme Klassen definieren
- Damit verringert man einerseits gezielt die Sichtbarkeit dieser Klassen
- Oft werden sie aber auch eingesetzt um kleine Sachen "vor Ort" angeben zu können
- (Nicht behandelt: statische Klassen und Initialisierer)

Reflektion

- Erarbeiten Sie sich drei eigene Beispiele, wo jeweils einer der drei Typen von geschachtelten Klassen jeweils Sinn macht