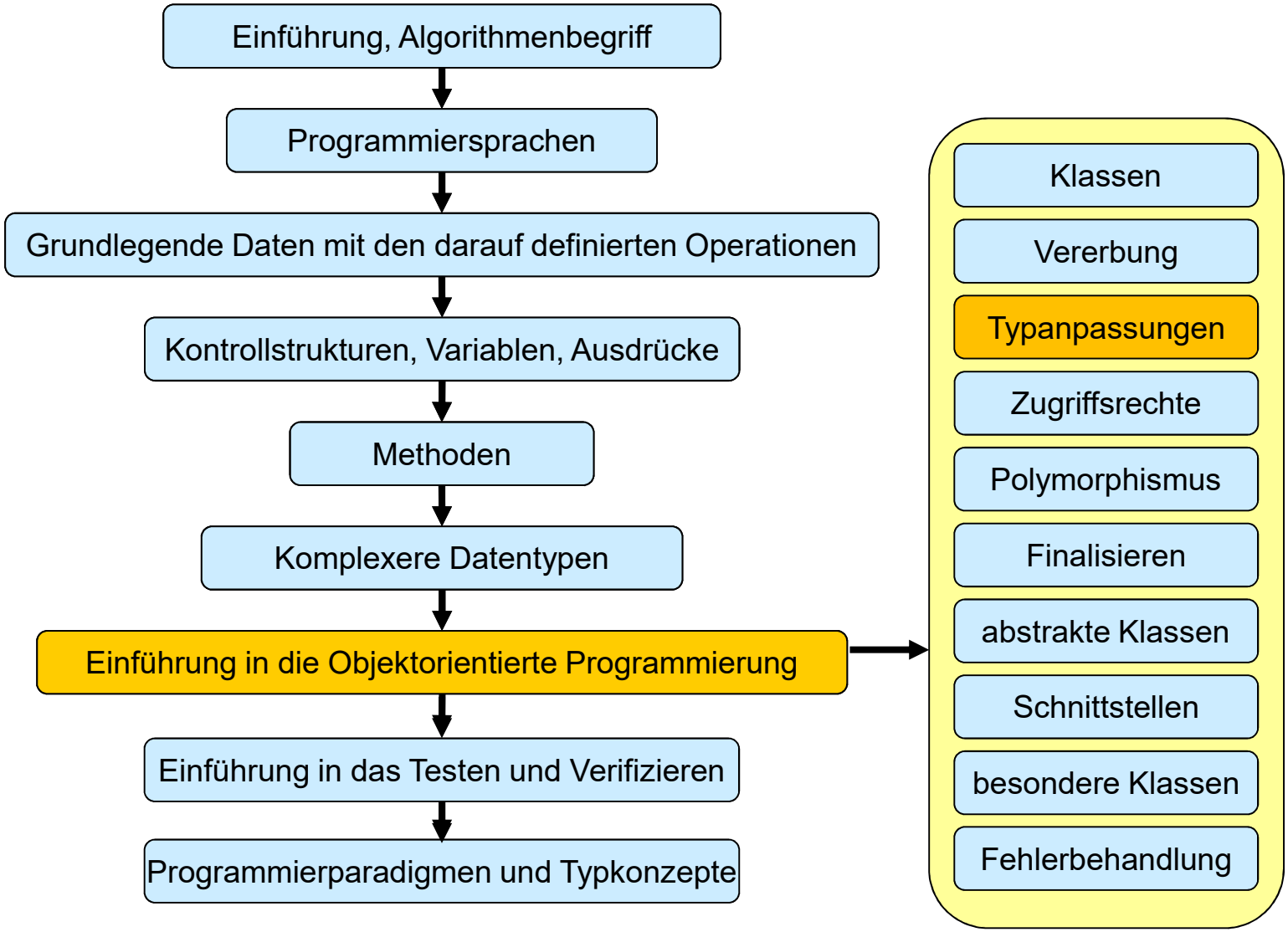
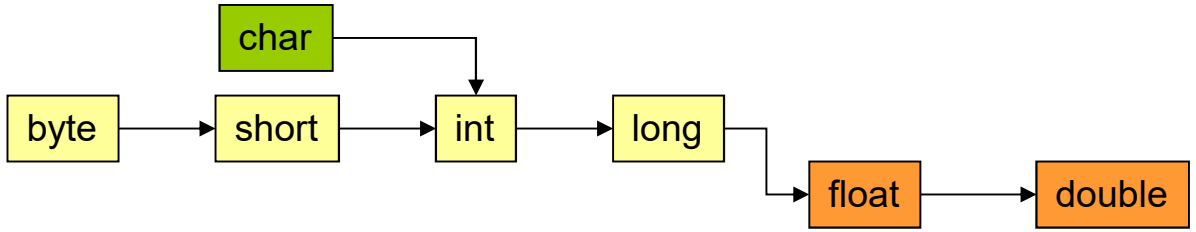
Inhalt dieser Veranstaltung



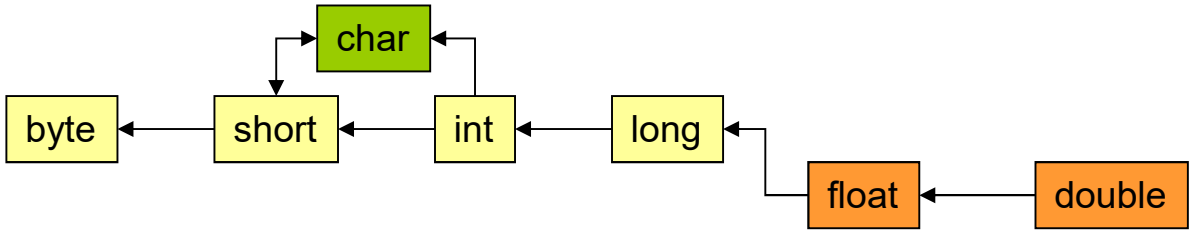
Wiederholung: Typanpassung bei primitiven Typen

- Erweiternde Typumwandlung: implizit oder explizit über cast-Operator
- Bis auf den Übergang ganzzahlig / Fließkomma keine Probleme mit Werterhalt
- Beispiel: `long l = 7;`



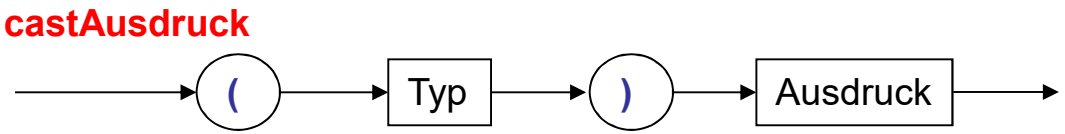
- Einengende Typumwandlung: nur explizit über cast-Operator möglich
- Erhebliche Probleme bzgl. des Werterhalts möglich
- Beispiele:

```
int i = (int) 7L;
int i = (int) (1L << 33);
```



Wiederholung: cast-Operator

- Cast-Operator in einem Ausdruck:



- Es wird im Folgenden an diesen Zusammenhängen keine Änderung geben, nur die Anwendung wird erweitert auf Referenztypen.

Der Sinn von Typanpassungen

- Was will man mit einer Typanpassung bei primitiven Typen bezwecken?
- Der Wert in einem Typ x soll unter Werterhalt (bzw. genau definierter "geringer" Wertabweichung) in einem (ähnlichen) Typ y dargestellt werden, z.B. um im Typ y Operationen ausführen zu können

- Was will man mit einer Typanpassung bei Referenztypen bezwecken?
- Der Wert in einem Typ x soll unter Werterhalt (bzw. genau definierter "geringer" Wertabweichung) in einem (ähnlichen) Typ y dargestellt werden, z.B. um im Typ y Operationen ausführen zu können

Sinn von Typanpassungen bei Referenztypen

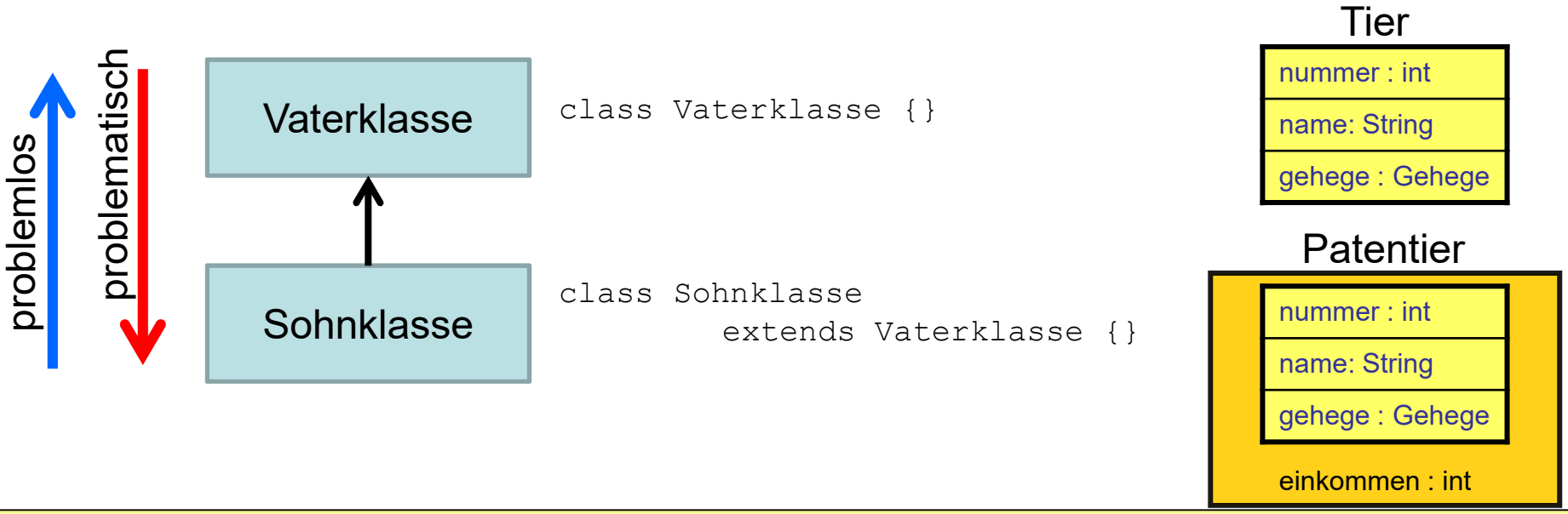
- Typanpassungen bei Referenztypen machen nur Sinn innerhalb einer Vererbungshierarchie (später noch Erweiterung auf Schnittstellen), weil dort überhaupt nur "ähnliche" Werte vorkommen

- Gegenbeispiele, wo es keinen Sinn machen würde:
  - 3.999 in `boolean` umwandeln
  - Objekt vom Typ `Tier` in ein Objekt vom Typ `Math` umzuwandeln. `Math.random()` macht Sinn. Aber was sollte `((Math)tier).random()` bedeuten?

Richtung der Typanpassungen bei Referenztypen

- Ebenso wie bei primitiven Typen
  - gibt es eine unproblematische Richtung
  - gibt es eine problematische Richtung
  - macht es manchmal überhaupt keinen Sinn (siehe 3.999 → boolean)

- Ebenso wie bei primitiven Typen sind bei Referenztypen implizite und explizite Typumwandlungen möglich

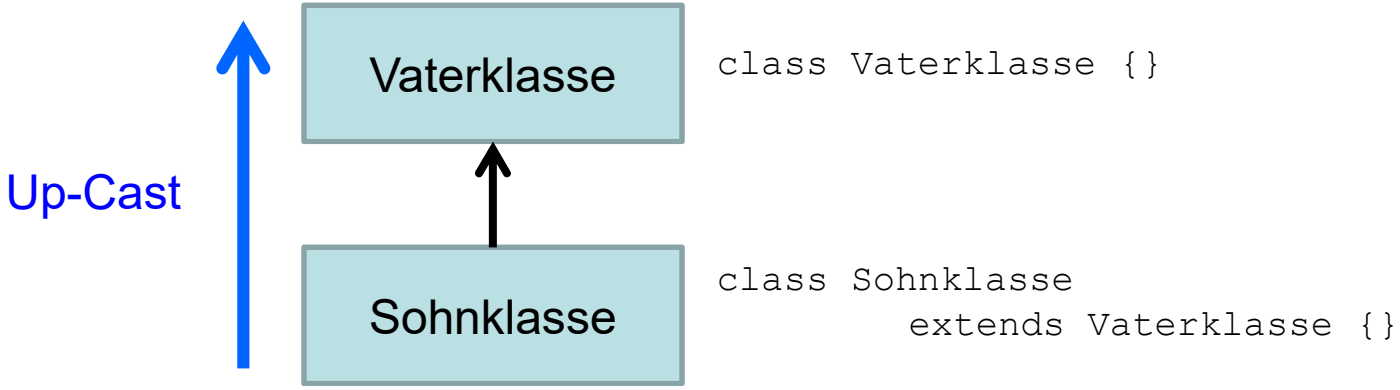


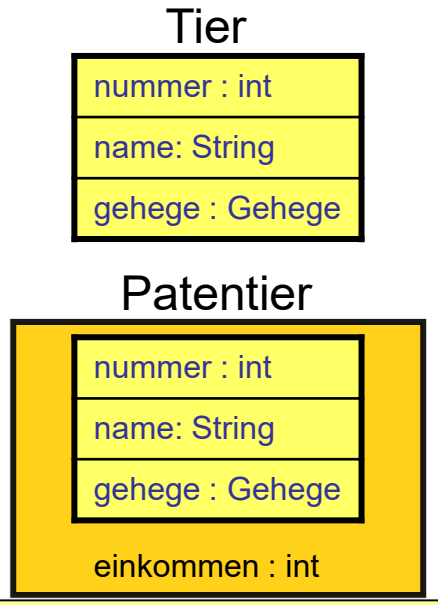
Up-Cast

- Von einem spezielleren (abgeleiteten) Typ auf den allgemeinen Typ
- Unproblematisch, weil in dem Fall `super` Teil von `this` ist
- Kann implizit oder explizit angewandt werden

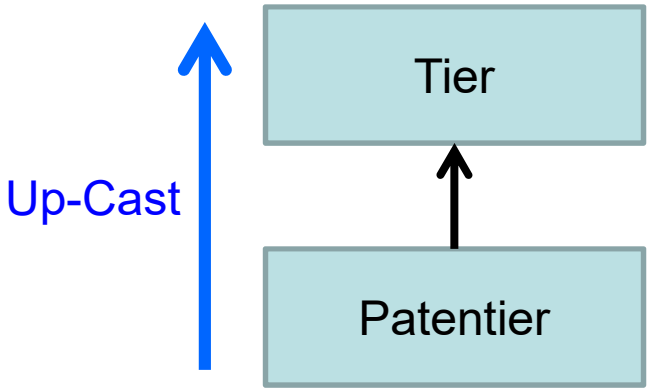
- Beispiel:

```
Patentier pt = new Patentier();
Tier t1 = (Tier) pt;    // hier explizit
Tier t2 = pt;           // hier implizit
```

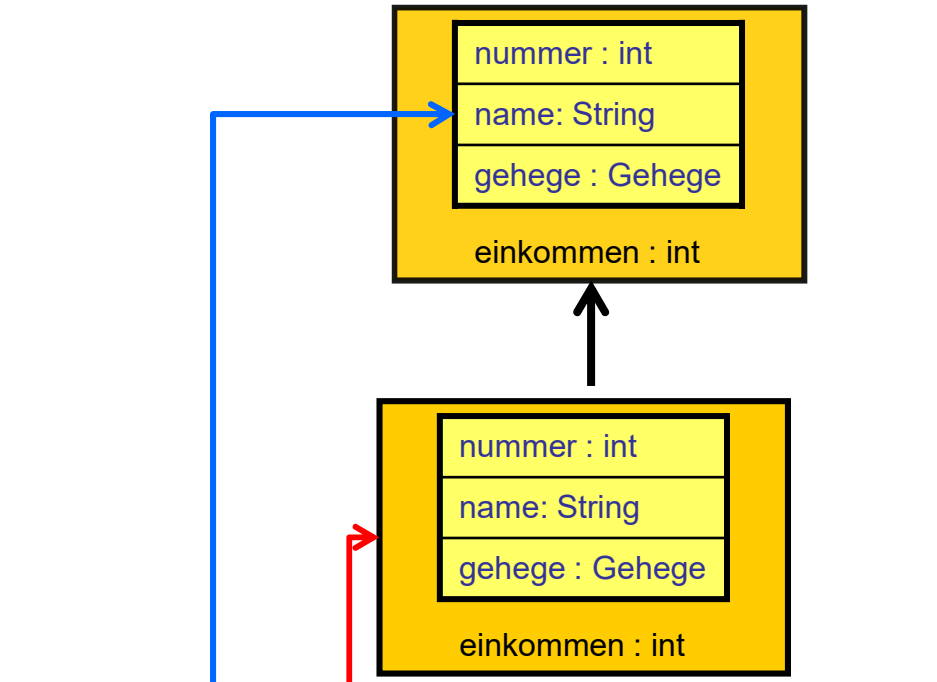




Was passiert bei einem Up-Cast?




```
class Tier {}
class Patentier extends Tier {}
```



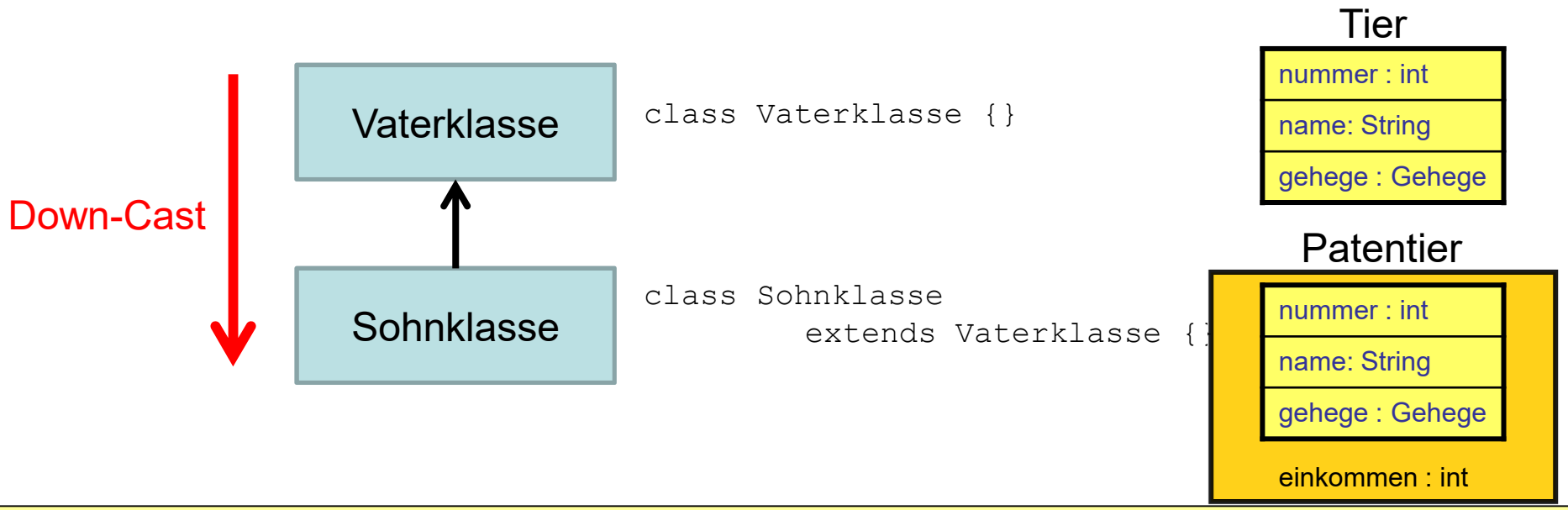

```
Patentier pt = new Patentier();
Tier t = (Tier) pt;
```

- Durch den Up-Cast sieht man nur noch den Tier-Teil des Patentiers.
- Damit sind nur noch Attribute und Methoden eines Tieres sichtbar.
- **Achtung**: **es ist und bleibt ein Patentier-Objekt**, nur die Sicht ist eingeschränkt auf den Tierteil. (Testen Sie dies z.B. mit dem instanceof-Operator später!)

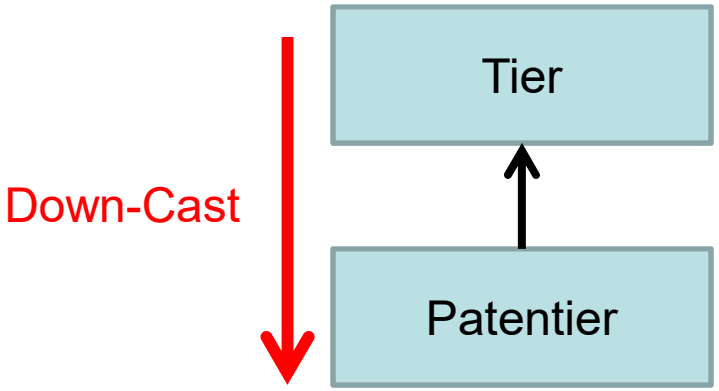
Down-Cast

- Von einem allgemeinen Typ auf den spezielleren Typ
- Evtl. problematisch
- Kann nur explizit angewandt werden
- Beispiel:

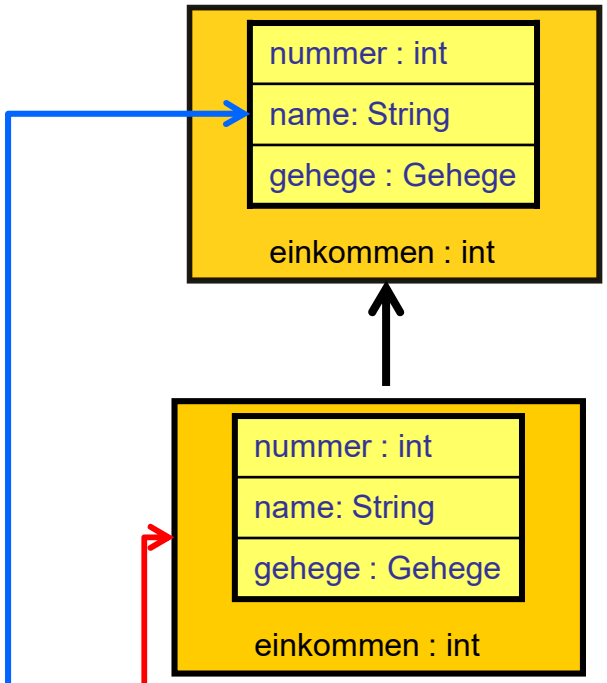
```
Tier t = (Tier) (new Patentier());
Patentier pt = (Patentier)t;  // hier unproblematisch
```



Was passiert bei einem Down-Cast?



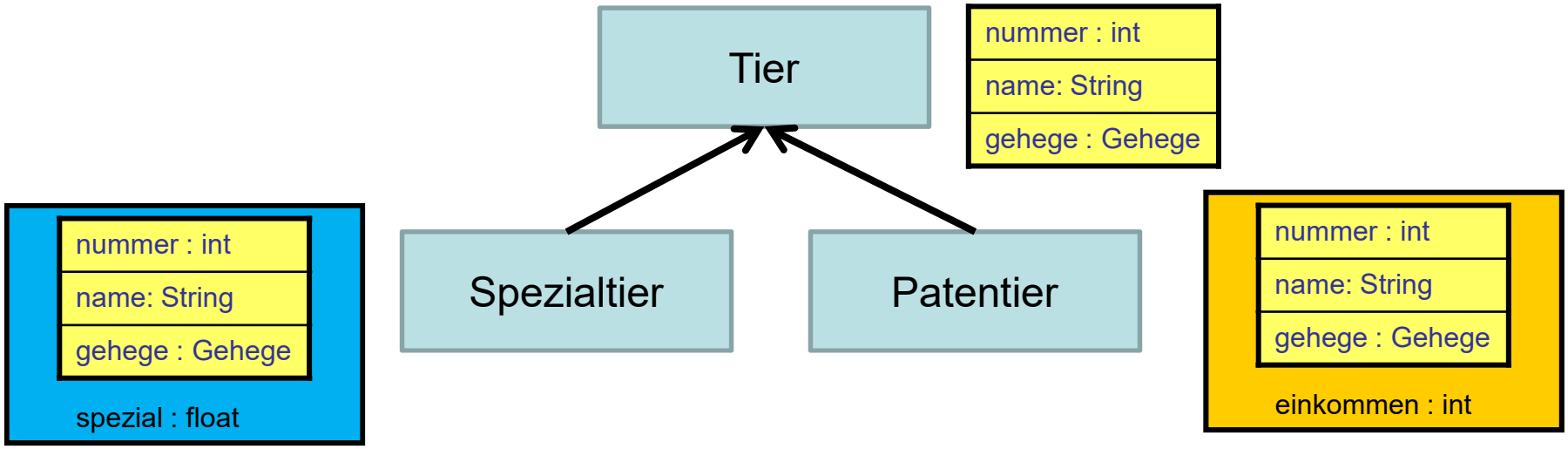

```
class Tier {}
class Patentier extends Tier {}
```



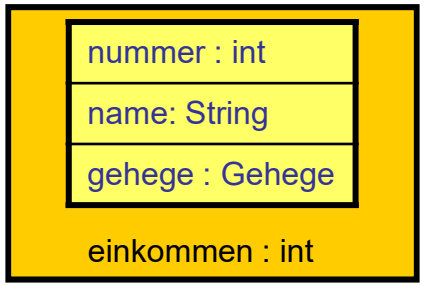

```
Tier t = (Tier) (new Patentier());
Patentier pt = (Patentier)t;
```

- Durch den Down-Cast sieht man das Gesamt-Patentier wieder
- Das Gesamtobjekt **muss** aber auch von diesem Zieltyp sein! Ansonsten gibt es einen Laufzeitfehler `java.lang.ClassCastException`

Wieso kann ein Down-Cast problematisch sein?

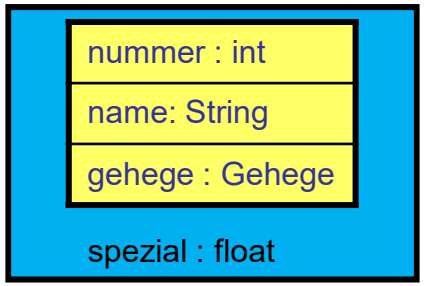



```
Tier t = (Tier) (new Patentier());
```



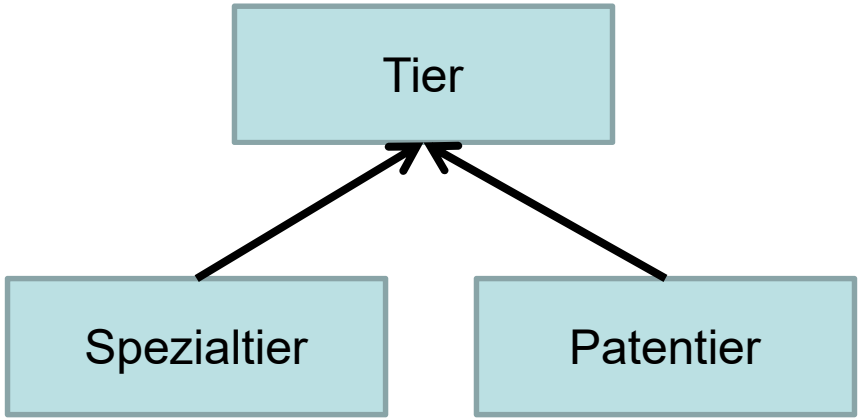
??? (macht keinen Sinn)

```
Spezialtier st = (Spezialtier)t;
// führt zu einem Laufzeitfehler
```



Alle Möglichkeiten zusammengefasst

```
class Tier { ...}
class Patentier extends Tier { ... }
class Spezialtier extends Tier { ...}
```




```
Patentier pt = new Patentier();
```

```
Tier t = pt;                // impliziter Up-Cast
     t = (Tier)pt;          // expliziter Up-Cast
    pt = (Patentier)t;      // expliziter Down-Cast moeglich
    pt = t;                 // Compilerfehler: impliziter Down-Cast nicht moeglich
```

```
Spezialtier st = pt;        // Compilerfehler: impliziter Cast nicht moeglich
    st = (Spezialtier)pt;   // Compilerfehler: expliziter Cast nicht moeglich
    st = (Spezialtier)t;    // Laufzeitfehler:
           // java.lang.ClassCastException: Patentier cannot be Cast to Spezialtier
```

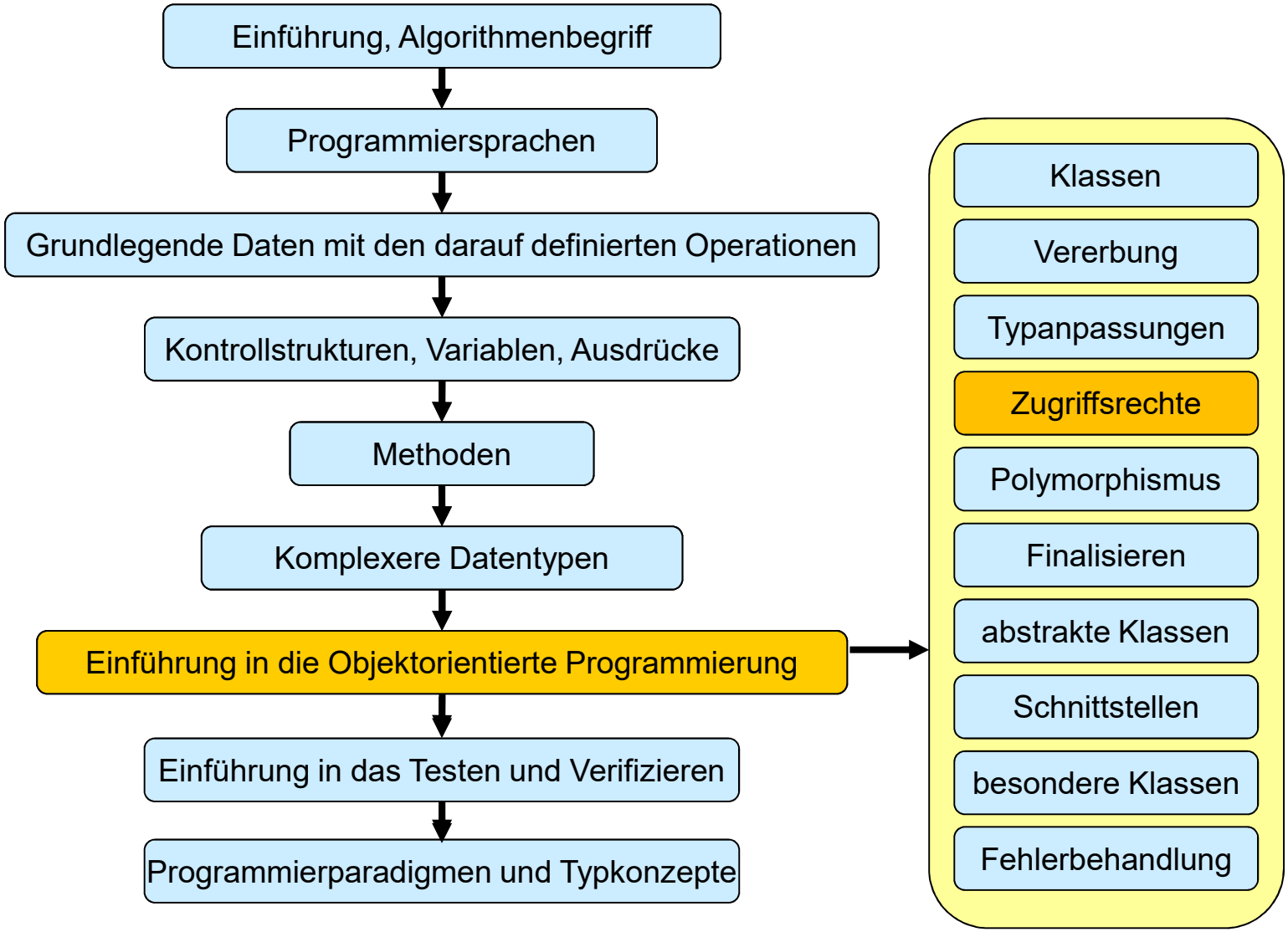
Zwischenstand

- Cast-Operationen machen bei Referenztypen nur innerhalb einer Klassenhierarchie Sinn
- Mit einem Cast wird die Sicht auf ein Objekt beeinflusst
- Up-Cast sind immer möglich und können implizit oder explizit stattfinden.
- Mit einem Up-Cast wird die Sicht auf den Objektteil der Basisklasse eingeschränkt.
- Down-Casts sind nur explizit mit einem Cast-Operator möglich.
- Bei einem Down-Cast kann es zu Laufzeitfehlern kommen.

Reflektion

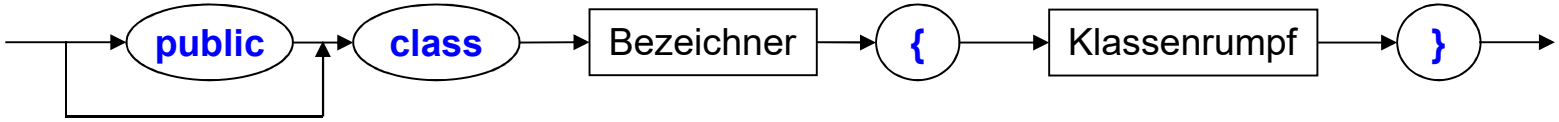
- Schauen Sie sich die bisherigen Beispiele zu Typanpassungen bei Referenztypen an. Überlegen Sie sich allgemeine Szenarien, wo Up- bzw. Down-Casts sinnvoll eingesetzt werden können.

Inhalt dieser Veranstaltung



Rechte für Klassen

Klassendefinition



- Klassen, die mit `public` markiert sind, können von beliebigen anderen Klassen genutzt werden
- Klassen, die ohne `public` angegeben sind, können nur von Klassen innerhalb des eigenen Pakets genutzt werden

- Jede Klasse `ABC`, die als `public` definiert ist, muss in einer eigenen Datei mit Namen `ABC.java` abgelegt sein.

- Für Klassen ohne `public` gilt diese Einschränkung nicht. Solche Klassen können
  - in einer Datei beliebigen Namens stehen
  - zusammen mit genau einer `public`-Klasse in einer Datei stehen
  - mehrere solcher Klassen ohne `public` können in einer Datei stehen

Beispiel

paket1/Klasse1.java

```
package paket1;

// Datei paket1/Klasse1.java
public class Klasse1 {
}

// kann in gleicher Datei stehen
class Klasse2 {
}
```

paket1/Klasse3.java

```
package paket1;

// Datei paket1/Klasse3.java
public class Klasse3 {
    Klasse1 k1 = new Klasse1();
    Klasse2 k2 = new Klasse2();
}

// kann in gleicher Datei stehen
class Klasse4 {
    Klasse1 k1 = new Klasse1();
    Klasse2 k2 = new Klasse2();
}
```

paket2/Klasse1.java

```
package paket2;

// Datei paket2/Klasse1.java
public class Klasse1 {
   paket1.Klasse1 k1 = new paket1.Klasse1();
   //falsch: paket1.Klasse2 k2 = new paket1.Klasse2();
   paket1.Klasse3 k3 = new paket1.Klasse3();
   //falsch: paket1.Klasse4 k4 = new paket1.Klasse4();
}
```

Dateistruktur:

paket1/Klasse1.java
paket1/Klasse3.java
paket2/Klasse4.java

Übersetzen:

javac paket1/Klasse1.java
javac paket1/Klasse3.java
javac paket2/Klasse4.java

Steuerung der Zugriffsrechte innerhalb einer Klasse

- (Selbststudium: Pakete; siehe Kapitel im Skript)
- Innerhalb einer Klasse sind alle Attribute und Methoden gültig und sichtbar
- Zu Methoden und Attributen lassen sich über vorangestellte Schlüsselwörter "Zugriffsrechte" vergeben (beeinflussen Gültigkeit des entsprechenden Namens)
- Beispiel: `private int nummer;`
- Der Name einer Instanz/Klassen-variable/-methode ist gültig innerhalb:

	eigene Klasse	**Klasse in gleichem Paket**	**zusätzlich Unterklasse in anderem Paket**	**überall**
`public`	ja	ja	ja	ja
`protected`	ja	ja	ja	nein
keine Angabe	ja	ja	nein	nein
`private`	ja	nein	nein	nein

Hierarchie

- Aufgrund der möglichen Hierarchie in einer Klasse gibt es dazu weitere Regeln, die diese Hierarchie beachten

- In einer tieferen Hierarchiestufe wird die Sichtbarkeit weiter eingeschränkt
- Beispiel:

```
public class Test1 {
    private int x;      // nicht gueltig in anderen Klassen
    public int y;       // gueltig in anderen Klassen
}
```

- In einer höheren Hierarchiestufe wird die Sichtbarkeit komplett eingeschränkt
- Beispiel:

```
class Test2 {
    private int x;      // nicht gueltig in anderen Klassen
    public int y;       // nicht gueltig in anderen Klassen anderer Pakete
}
```

Beispiel mit allen Szenarien

```
package paket1;
// Basisklasse
public class Zugriffsrechte1 {
  private int a;     // innerhalb der eigenen Klasse
  int b;             // in allen Klassen des gleichen Pakets
  protected int c;   // abgeleitete Klassen anderer Pakete
  public int d;      // ueberall gueltig

  void test() {
    a = 4711;   // moeglich
    b = 4712;   // moeglich
    c = 4713;   // moeglich
    d = 4714;   // moeglich
  }
}

// andere Klasse gleiches Paket
class AndereKlasseGleichesPaket1 {
  void test(Zugriffsrechte1 obj) {
    //obj.a = 4711;   // nicht moeglich wegen private
    obj.b = 4712;   // moeglich, weil gleiches Paket
    obj.c = 4713;   // moeglich, weil gleiches Paket
    obj.d = 4714;   // moeglich, weil gleiches Paket
  }
}

// andere Klasse gleiches Paket, abgeleitet
class AndereKlasseGleichesPaket2 extends Zugriffsrechte1 {
  void test() {
    // a = 4711;   // nicht moeglich wegen private
    b = 4712;   // moeglich, weil gleiches Paket
    c = 4713;   // moeglich, weil gleiches Paket
    d = 4714;   // moeglich, weil gleiches Paket
  }
}
```

```
// andere Klassen in anderem Paket
package paket2;

// andere Klasse anderes Paket, abgeleitet
class AndereKlasseAnderesPaket2
                extends paket1.Zugriffsrechte1 {
  void test() {
    //a = 4711;   // nicht moeglich wg. private
    //b = 4712;   // anderes Paket
    c = 4713;   // abgeleitete Klasse anderes Paket
    d = 4714;   // public
  }
}

// andere Klasse anderes Paket
class AndereKlasseAnderesPaket1{
  void test(paket1.Zugriffsrechte1 obj) {
    //obj.a = 4711;   // nicht moeglich wg. private
    //obj.b = 4712;   // anderes Paket
    //obj.c = 4713;   // nicht abgeleitet
    obj.d = 4714;   // moeglich
  }
}
```

analog für Methoden

Datenkapselung

- In einer Klasse ist die Zustandbeschreibung (Instanzvariablen) und die Funktionalität (Instanzmethoden) zu gleichartigen Objekten an einer Stelle zusammengefasst

- Über eine Referenz hat man direkten Zugriff auf die Instanzvariablen
- Oft möchte man aber nicht, dass der Zustand von außerhalb der Klasse direkt über eine Referenz geändert werden kann

- Deshalb: Datenkapselung
  - alle Informationen zu gleichartigen Objekten sind in der Klasse konzentriert
  - Zugriffsrechte auf Instanzvariablen und Methoden (und sogar Klassen) können gezielt gesteuert werden
  - wohldefinierte (Methoden-)Schnittstelle nach außen, über die alle gewünschte Funktionalität bereit gestellt wird

Getter-/Setter-Methoden

- Meist ist es ratsam, den direkten Zugriff auf Instanzenvariablen von außerhalb der eigenen Klasse einzuschränken (Verwendung von `private`)
- Möchte man aber trotzdem eine Abfrage des Inhalts oder Veränderung des Zustands erlauben, bieten sich Getter- und/oder Setter-Methoden an
- Vorteil: nur kontrollierte Änderungen der Instanzenvariablen sind möglich

- Zu einer Instanzenvariablen mit einer Deklaration (`typ` kann beliebiger Typ sein)

```
private typ xyz;
```

werden definiert:

  - Setter-Methode

```
public void setXyz(typ wert) { xyz = wert; }
```

  - Getter-Methode

```
public typ getXyz() { return xyz; }
```

- Statt `public` auch `protected` möglich
- Achtung: Namensgebung ist exakt einzuhalten (Groß-/Kleinschreibung)!

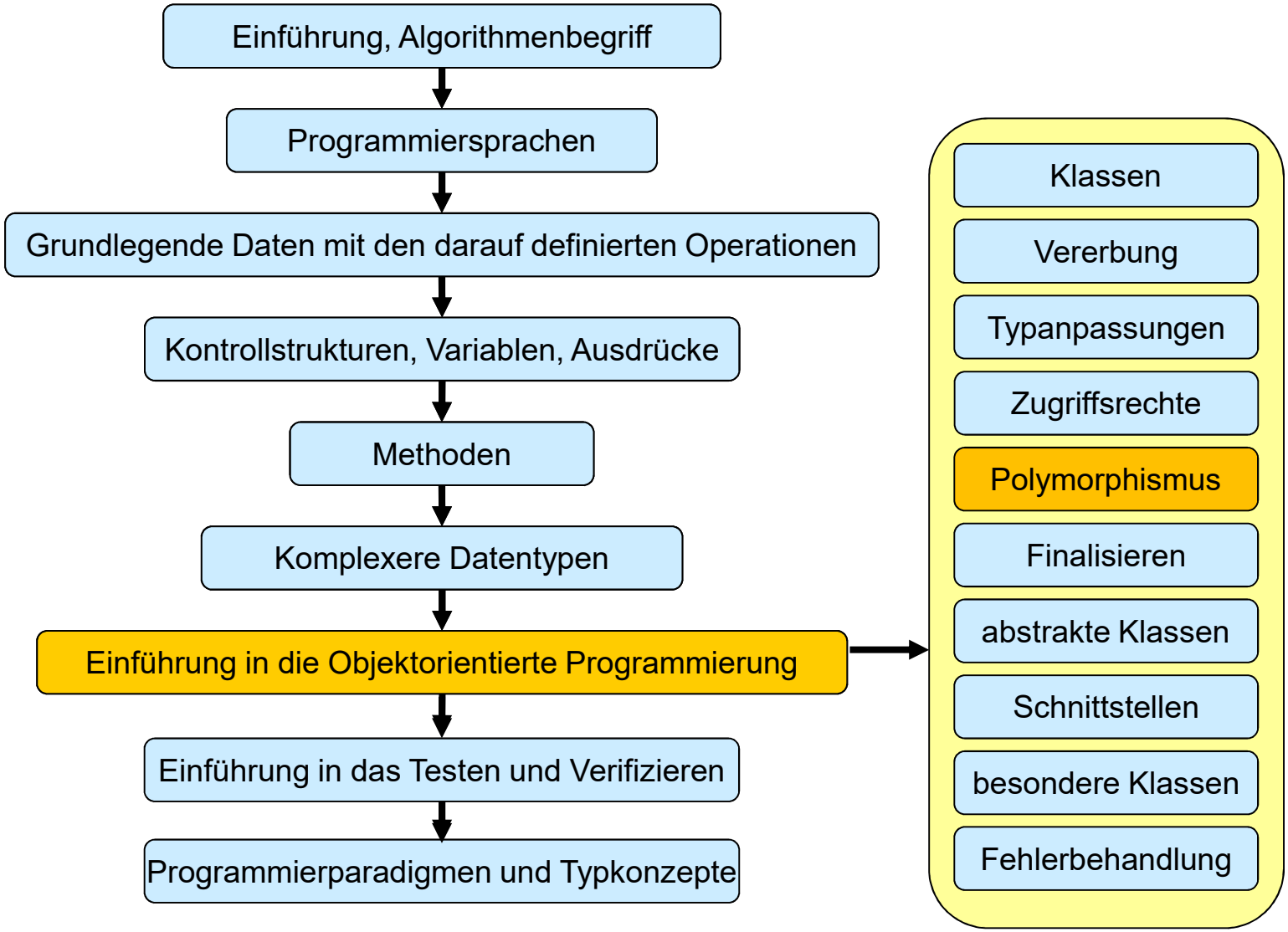
Zwischenstand

- Über Schlüsselwörter lässt sich die Sichtbarkeit von Klassen sowie Klassen-/Instanzvariablen-/methoden beeinflussen
- Getter/Setter-Methoden verhindern den direkten Zugriff auf Attribute

Reflektion

- Geben Sie ein Szenario an, wie man den Wert einer als `private` markierten Instanzvariablen eines Referenztyps auch außerhalb der eigenen Klasse verändern kann, ohne selbst auf die Instanzvariable direkt zuzugreifen.
- Wieso macht man nicht alles grundsätzlich `public`?

Inhalt dieser Veranstaltung



Überladen / Überschreiben

- Zur Erinnerung: einen Methodennamen nennt man überladen, wenn es in der Klasse mehrere Methoden mit diesem Namen gibt. Die Signaturen müssen sich allerdings unterscheiden!

- Beispiel:

```
public class Arithmetik {
    public static int addieren(int a, int b) {... }
    public static int addieren(int a, int b, int c) {...}
    public static int addieren(int a, int b, int c, int d) {...}
}
```

- Gibt es in einer Oberklasse K_1 eine Methode mit Signatur S (diese enthält den Namen), so kann man in einer abgeleiteten Klasse K_2 eine Methode mit **gleicher** Signatur definieren und damit diesen Namen / diese Signatur in der abgeleiteten Klasse K_2 überschreiben

Beispiel

```
public class Tier {
  private int nummer;              // eindeutige Nummer des Tieres
  private String name;             // Name des Tieres (evtl. nicht vorhanden)
  private Gehege gehege;           // Gehege, in dem es wohnt
  ...
  public String toString() {
    return "Tier " + name
        + ", Gehege=" + ((gehege == null)?"ohne Gehege" : gehege.getNummer());
  }
}
```

gleiche Signatur: toString wird in Patentier überschrieben

```
public class Patentier extends Tier {
  private int einkommen; // jaehrliches Einkommen
  ...
  public String toString() {
    return super.toString() + ", Einkommen=" + einkommen;
  }
}
```

Polymorphismus

- Polymorphismus (Viel-/Verschiedengestaltigkeit):
  Das sichtbare Verhalten ist gleich, auch wenn die interne Arbeitsweise verschieden sein kann (zum Beispiel in Abhängigkeit vom konkreten Objekt)

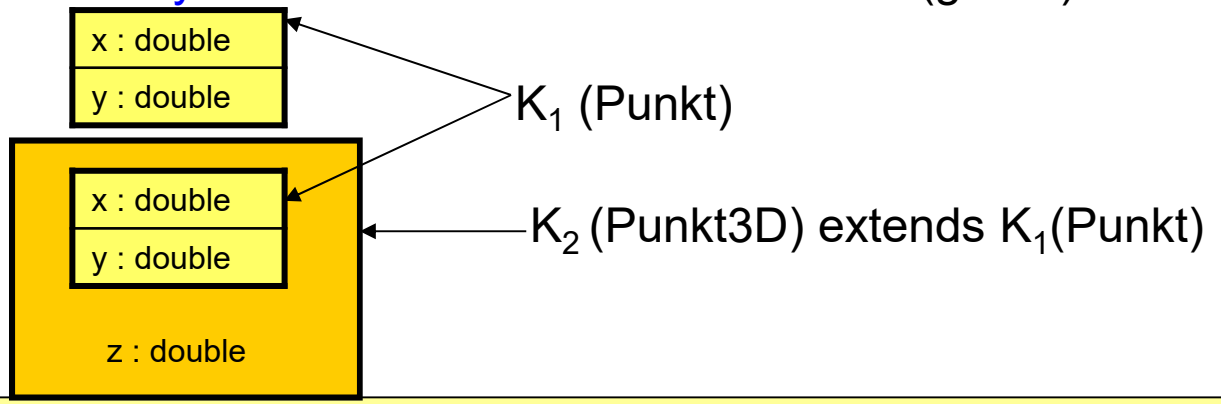
- Beispiele:
  - `public String toString()`
    Diese Methode soll in jeder Klasse das gleiche Verhalten zeigen: Erzeuge einen String mit einem sinnvollen Inhalt zu diesem Objekt (beispielsweise den Zustand des Objekts).
  - Verschieben eines Punkts (sowohl bei 2D als auch 3D)
    Die Aktion ist sinngemäß die gleiche (verschieben des Punktes), aber die konkrete Arbeitsweise ist für 2D und 3D verschieden

- Wir kennen bereits zwei Formen zur Realisierung von Polymorphismus:
  - Überladen eines Methodennamens in gleicher Klasse (Beispiel `addieren`)
  - Überschreiben einer Methodensignatur / Methodennamens in abgeleiteter Klasse (Beispiel `toString`)

Polymorphismus auf Klassenebene

ganz wichtig!!!!!!!

- Objekte können sich polymorph verhalten
- **Substitutionsprinzip:**
  Überall dort, wo in Java ein Objekt einer Klasse K_1 genutzt werden kann, kann auch ein Objekt einer Klasse K_2 genommen werden, wobei K_2 direkt oder indirekt von K_1 abgeleitet ist
- Beispiel:
  - Überall, wo ein Tier genutzt wird, kann auch ein Patentier genutzt werden
  - Überall, wo ein Punkt genutzt wird, kann auch ein Punkt3D genutzt werden
- Siehe dazu auch Typumwandlung: Up-Casts
- In diesem Zusammenhang wird dann eine weitere Form der Polymorphismus-Realisierung interessant: dynamisches Binden von Methoden (gleich)



Beispiel 1

```
public class Polymorphismus {

    // zeige Zustand eines Tieres auf dem Bildschirm
    public static void zeigeTier(String ueberschrift, Tier t) {
        System.out.println(ueberschrift + " : " + t.toString());
    }

    public static void main(String[] args) {
        // normales Tier erzeugen
        Tier t = new Tier();
        // Patentier erzeugen
        Patentier pt = new Patentier(100);

        // Methode kann mit beiden Tieren aufgerufen werden
        zeigeTier("normales Tier", t);
        zeigeTier("Patentier", pt);
    }
}
```

impliziter Upcast

Ausgabe:

```
normales Tier : Tier unbekannt, Gehege=ohne Gehege
Patentier : Tier unbekannt, Gehege=ohne Gehege, Einkommen=100
```

Beispiel 2

```
public class Polymorphismus2 {
    // zeige Zustand eines Tieres auf dem Bildschirm
    public static void zeigeTier(String ueberschrift, Tier t) {
        System.out.println(ueberschrift + " : " + t.toString());
    }

    public static void main(String[] args) {
        // Feld von Tieren
        Tier[] tiere = new Tier[2];

        // normales Tier erzeugen
        tiere[0] = new Tier();
        // Patentier erzeugen (impliziter Upcast in Zuweisung)
        tiere[1] = new Patentier(100);

        // Methode kann mit einem beliebigen Tier aufgerufen werden
        for(int i=0; i<tiere.length; i++) {
            zeigeTier("was ist es?", tiere[i]);
        }
    }
}
```

Ausgabe:

```
was ist es? : Tier unbekannt, Gehege=ohne Gehege
was ist es? : Tier unbekannt, Gehege=ohne Gehege, Einkommen=100
```

Dynamische Bindung von Methoden

- Im letzten Beispiel:

```
// zeige Zustand eines Tieres auf dem Bildschirm
public static void zeigeTier(String ueberschrift, Tier t) {
    System.out.println(ueberschrift + " : " + t.toString());
}
```

- Woher weiß Java, welches `toString()` zu nehmen ist? Insbesondere, wenn es sich um ein Patentier handelt!

- Möglichkeit 1: Durch einen Up-Cast hat man nur noch die Sicht auf ein Tier. Nehme die Methode, die sich durch diese eingeschränkte Sicht ergibt (also `toString()` aus Tier)

- Möglichkeit 2: Es ist und bleibt ein Patentier! Nehme das Original-`toString()` aus Patentier.

- Was wollen wir haben? Antwort: beides

Dynamische / statische Bindung von Methoden

- Im letzten Beispiel:

```
// zeige Zustand eines Tieres auf dem Bildschirm
public static void zeigeTier(String ueberschrift, Tier t) {
    System.out.println(ueberschrift + " : " + t.toString());
}
```

- Woher weiß Java, welches `toString()` zu nehmen ist?

- Antwort (nur für Instanzmethoden relevant, gilt nicht für Klassenmethoden):
  - Stelle fest, ob es eine Ausnahmeregelung an der Stelle gibt
  - Ansonsten nehme die Default-Regelung

Dynamische / statische Bindung von Methoden

- **Die Ausnahme (statische Bindung):**
  Falls von der Aufrufstelle aus gesehen eine Methode mit der gesuchten Signatur sichtbar ist, die mit `private`, `static` oder `final` (kommt später) deklariert ist, so nehme diese.
  (Grund dafür: solche Methoden können nicht überschrieben werden!)

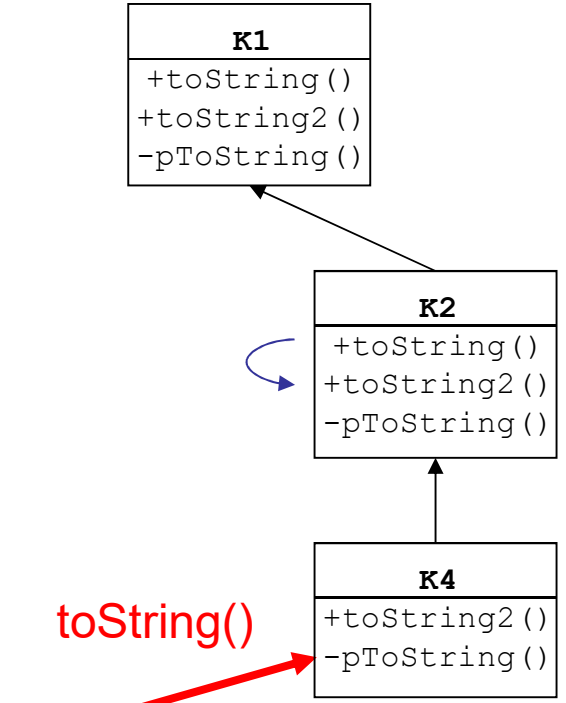
- **Die Regel: dynamische Bindung (der Default)**
  Ansonsten gehe **immer** ausgehend vom realen Gesamtobjekt (also Patentier im Beispiel) die Klassenhierarchie hoch, bis die Signatur zum Aufruf gefunden ist. Führe diese Methode dann aus.
  (Es ist und bleibt ein Patentier!)

```
// zeige Zustand eines Tieres auf dem Bildschirm
public static void zeigeTier(String ueberschrift, Tier t) {
    System.out.println(ueberschrift + " : " + t.toString());
}
```

Beispiel

```
public class AufrufTest {
    public static void main(String[] args) {
        K4 k4 = new K4();
        System.out.println(k4.toString());
    }
}

class K1 {
    private String pToString() { return " priv in K1"; }
    public String toString2() { return " toString2 in K1"; }
    public String toString() { return "in K1" + pToString(); }
}
class K2 extends K1 {
    private String pToString() { return " priv in K2"; }
    public String toString2() { return " toString2 in K2"; }
    public String toString() { return "in K2" + pToString()
                                     + toString2(); }
}
class K4 extends K2 {
    public String toString2() { return " toString2 in K4"; }
    private String pToString() { return " priv in K4"; }
}
```



Ausgabe: `in K2` `priv in K2` `toString2 in K4`

Operator instanceof

- Beim Überschreiben kann es in der abgeleiteten Klasse sinnvoll sein, dass man den Typ der Basisklasse z.B. als Typ eines Parameters wählt, um so eine Methode überschreiben zu können

- Beispiel:

```
class Punkt   { void move(Punkt p) {…} }
class Punkt3D { void move(Punkt p) {…} }   // überschreiben (das wollen wir)
class Punkt3D { void move(Punkt3D p) {…} } // falsch, kein überschreiben
                                           // sondern überladen der Methode
```

- Dann muss man aber ggfs. in der abgeleiteten Klasse die Möglichkeit haben, dynamisch (!) auf den Typ des tatsächlichen Objekts reagieren zu können

- Dazu neuer boolscher Operator:

Was bin ich?

```
public class Punkt {
  double x,y;
  void move(Punkt p) { x +=p.x;  y += p.y; }
}

public class Punkt3D extends Punkt {
  double z;
  void move(Punkt p) {  // hier muss Punkt stehen, sonst kein Überschreiben
    super.move(p);
    if(p instanceof Punkt3D) { z+= ((Punkt3D)p).z; }
  }
}

public class Gerade {
  Punkt p1, p2;          // Basisklasse Punkt (auch für Punkt3D Objekte!)
  void move(Punkt p) { // wichtig: hier Basisklasse fuer Punkte!
    p1.move(p);          // verschiebe beide Endpunkte (egal ob 2D oder 3D)
    p2.move(p);
  }
}

…
Gerade g1 = new Gerade(new Punkt(0,0), new Punkt (1,1));          // 2D Version
g1.move(new Punkt(1,1));
Gerade g2 = new Gerade(new Punkt3D(0,0,0), new Punkt3D(1,1,1)); // 3D Version
g2.move(new Punkt3D(1,1,1));
```

Nur in der Klasse Punkt3D findet die Differenzierung 2D/3D statt, **nicht** in der Klasse Gerade!

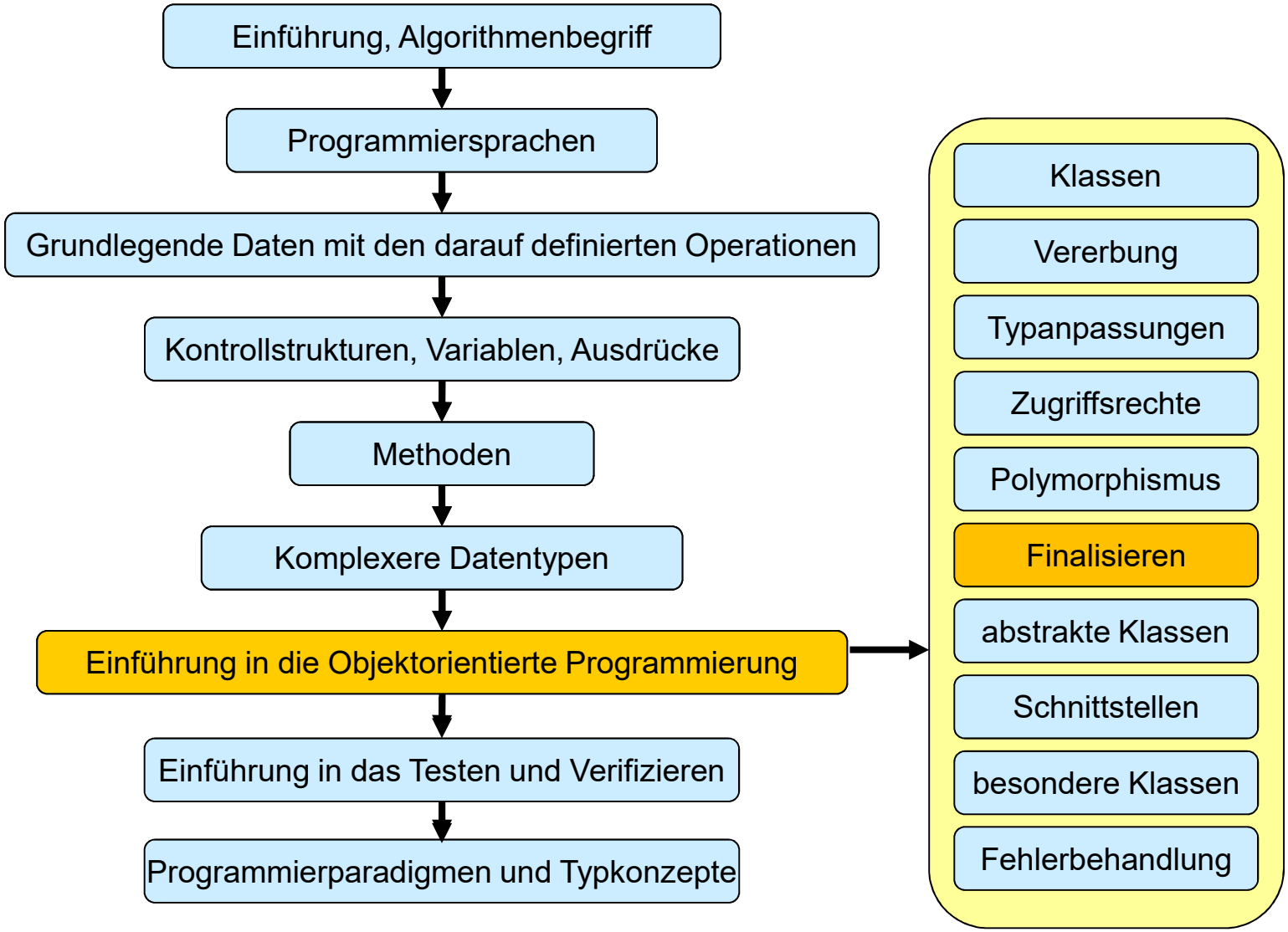
Zwischenstand

- Überladen einer Methode bedeutet, mehrere Alternativen der Nutzung anzubieten.
- Überschreiben einer Methode bedeutet, dass in einer abgeleiteten Klasse eine andere / angepasste Version (mit gleicher Signatur) angegeben wird.
- Auch Objekte einer abgeleiteten Klasse können überall dort genutzt werden, wo Objekte der Oberklasse verlangt werden (Substitutionsprinzip)
- Bei Methoden mit dynamischer Bindung wird immer vom realen Objekt ausgehend die aufzurufende Methodeninstanz gesucht

Reflektion

- Diskutieren Sie, welche Vor- und Nachteile mit statischer / dynamischer Methodenbindung existieren. In welchen Programmszenarios würde man sich statische Bindung wünschen, in welchen dynamische?

Inhalt dieser Veranstaltung



Finalisieren von Variablen (Wiederholung)

- Durch die Markierung einer Variablen (blocklokal, Instanz-/Klassenvariable) mit `final` innerhalb der Variablendeklaration wird die Variable als (benannte) Konstante deklariert.

- Zweck dieser Verwendung:
  - Einer Zahl einen selbstredenden Namen geben.
    Beispiel: `final double PI = 3.1415;`
  - Den Compiler überprüfen lassen, dass eine Variable nicht mehr innerhalb ihres Gültigkeitsbereichs geändert wird (Code-Sicherheit)
  - Dem Compiler Zusatzinformationen für Optimierungen geben

- Vereinbarung Java Code Conventions:
  - Finale Klassenvariablen werden ausschließlich mit Großbuchstaben geschrieben mit _ als Trenner zwischen Teilworten
  - Beispiel: `static final MOTOROEL_KLASSE = 1;`

Finalisierung von Methoden

- Bei der Definition einer Methode lässt sich auch `final` verwenden
- Wirkung: Diese Methode lässt sich nicht mehr in einer abgeleiteten Klasse überschreiben
- Beispiel V1:

```
class Authorization {
    boolean check(String password) { … }
}

class Phisher extends Authorization { // Substitutionsprinzip!
    boolean check(String password) {
        sendToDarthVader(password);     // schlimme Sache
        return super.check(passwort);   // faellt so nicht auf
    }
}
```

- Beispiel V2:

```
class Authorization {
    final boolean check(String password) { … }
}

class Phisher extends Authorization {
    // Compiler Fehler
    boolean check(String password) {…}
}
```

Finalisierung von Klassen

- Auch bei Klassen lässt sich `final` angeben
- Wirkung: Von dieser Klasse kann nicht mehr abgeleitet werden
- Einsatz z.B. aus Sicherheitsgründen

- Beispiel:

```
final class Authorization {
    boolean check(String password) { … }
}

// Compiler Fehler
class Phisher extends Authorization {
}
```

- Einige fundamentale Klassen der Java Klassenbibliothek sind als final markiert. U.a. `String`, `Math`, `System`

Zwischenstand

- Variablen, Methoden und Klassen lassen sich finalisieren
- Aus Variablen macht man damit benannte Konstanten
- Bei Methoden verhindert man ein Überschreiben der Methoden
- Bei Klassen verhindert man eine Ableitung
- Die richtige Nutzung des Finalisierens sorgt für größere Code-Sicherheit

Reflektion

- Sollte man grundsätzlich nur Instanz- / nur Klassenmethoden finalisieren? Oder macht beides Sinn?
- Macht das Finalisieren einer Methode Sinn, wenn es nur eine statische Methodenbindung zu der Methode gibt? Nur dynamische?