



Informatik I

Prof. Dr. Christian Pape

Kapitel 6

Objekt-orientierte Programmierung mit Java



Inhalt

- Attribute
- Beziehungen
- Objekt-Methoden
- Konstruktor
- Geheimnisprinzip
- static
- Java Packages



Attribute

- **Objekt-orientierung**
 - Klassen beschreiben Mengen von Objekten
 - Objekte haben Eigenschaften und Verhalten

Person
vorname : String nachname : String alter : int geburtsDatum : Date
heiraten(ehegatte : Person) : void



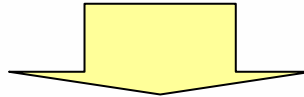
Attribute und Methoden

Person
vorname : String nachname : String
heiraten(ehegatte : Person) : void



Attribute und Methoden

Person
vorname : String nachname : String
heiraten(ehegatte : Person) : void



```
public class Person {
```

```
    public String vorname;  
    public String nachname;
```

Attribute (kein static)

```
    public void heiraten(Person ehgatte) {  
        // heiraten programmieren  
    }
```

Objektmethode (kein static)

```
}
```



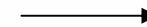
Attribute

- Objekt meier erzeugen mit Aufruf *Konstruktor*: `new Person()`
 - Erzeugt Objekt vom Typ `Person` im Hauptspeicher
- Speichern in lokaler Variablen
 - Lokale Variable ist **Verweis** auf Objekt im Speicher

```
public class PersonTest {

    public static void main(String argv[])
        Person meier = new Person();
    }
}
```

meier



RAM (Heap)

Platz für Attribut-
werte
einer Person

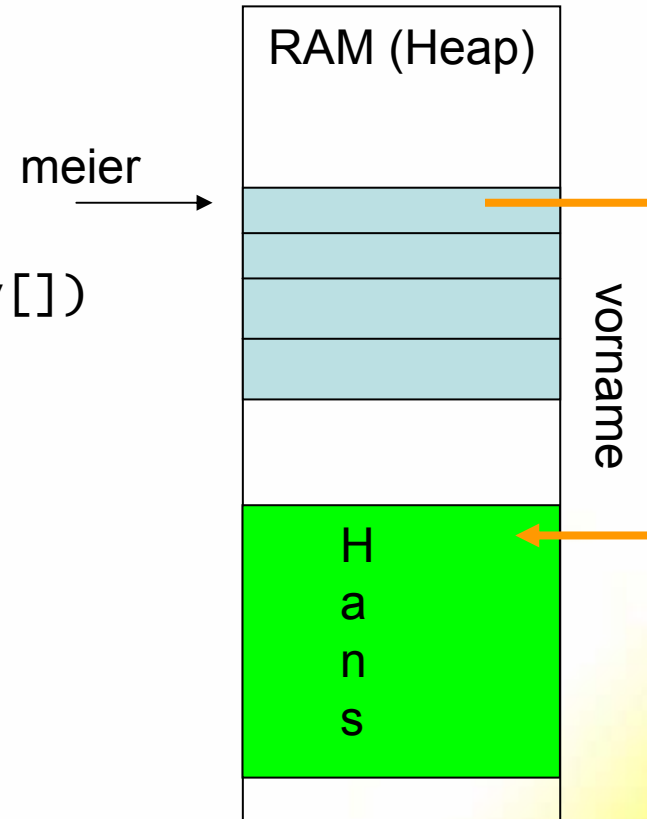
meier : Person

vorname = „Hans“
nachname = „Meier“

Attribute

- Attribute einer Objektvariablen mit „.“ referenzieren
- Attribute vorname ist Verweis auf Objekt vom Typ String

```
public class PersonTest {
    public static void main(String argv[])
        Person meier = new Person();
        meier.vorname = "Hans";
        meier.nachname = "Meier";
    }
}
```

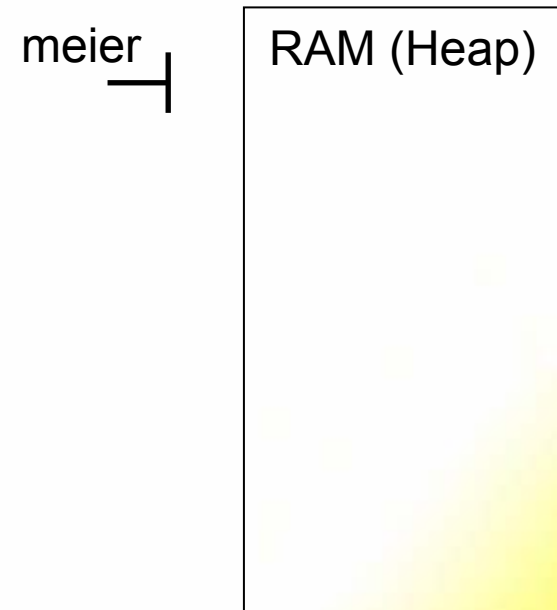


Attribute

- Variablen (Attribute) mit Objekttyp haben initial Wert „null“, falls bei Deklaration kein Wert zugewiesen wird
- Zugriff auf Attribute (Methoden) bei nicht-initialisierten Attribute ergeben NullPointerException Laufzeit
- **Defensiv Programmieren: Mögliche Fehler abfangen**
 - Darauf achten, das Variablen immer mit Objekt initialisiert sind
 - Im Zweifelsfall Inhalt Variablen vor Zugriff auf null überprüfen

```
public class PersonTest {

    public static void main(String argv[])
        Person meier;
        // NullPointerException zur Laufzeit:
        meier.nachname = „Meier“;
        // irgendwelche Anweisungen
        if (meier != null) {
            meier.nachname = „Meier“;
        }
    }
}
```





Attribute

- Attribute:
 - Bei Deklaration initialen Wert angeben
 - Falls kein Wert: Attribut wird mit null initialisiert
 - Rechte Seite von „=“ kann komplexer Ausdruck sein (Ergebnistyp muss mit Typ Attribut kompatibel sein)
- Bei Aufruf Konstruktor
 - Objekt wird im Speicher erzeugt
 - Attribute werden sequentiell von „oben“ nach „unten“ mit Werten initialisiert
 - Zuvor initialisierte Attribute können zu Initialisierung nachfolgender Attribute verwendet werden (auch Aufruf Methoden möglich)

```
public class Person {  
    public String vorname = "Hans";  
    public String nachname = "Meier";  
    public int alter = 28;  
    public String vollerName = vorname + " " + nachname;  
}
```



Attribute

<u>: Person</u>

<u>: Person</u>
vorname = „Hans“

<u>: Person</u>
vorname = „Hans“ nachname = „Meier“ alter = 28 vollerName = „Hans Meier“

- Vor Initialisierung Attribute
- Nach Initialisierung „vorname“
- Nach Initialisierung aller Attribute

```
public class Person {
    public String vorname = "Hans";
    public String nachname = "Meier";
    public int alter = 28;
    public String vollerName = vorname + " " + nachname;
}
```



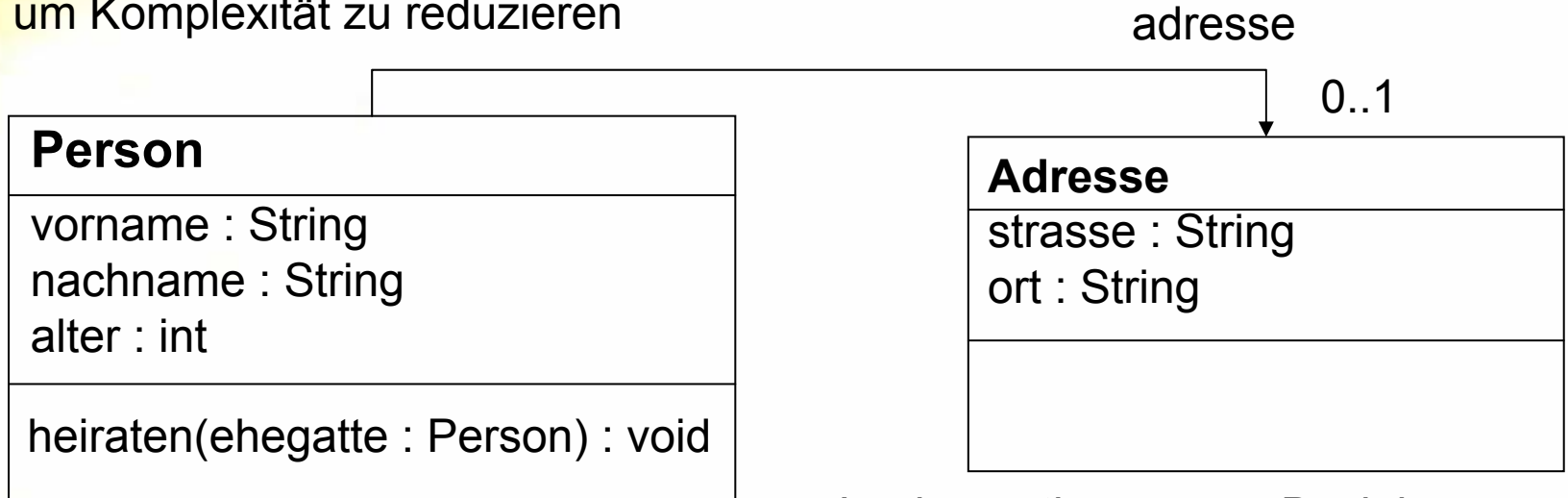
Inhalt

- Attribute
- **Beziehungen**
- Objekt-Methoden
- Konstruktor
- Geheimnisprinzip
- static
- Java Packages



Beziehungen

Nur eine Richtung der Beziehung im Entwurf gewählt,
um Komplexität zu reduzieren

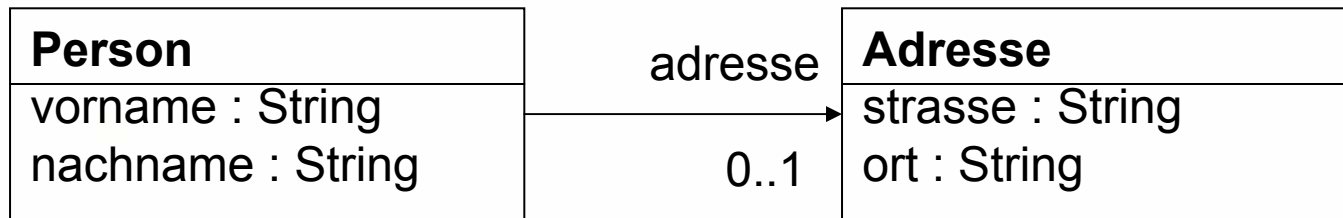


- Implementierung von Beziehungen als Java-Attribut
- Klassen sind auch Typen
- 1-n Beziehungen via Felder oder Datenstrukturen (später)
- Gerichtete Beziehung (Pfeil), Navigierbar von einer Person zu seiner Adresse (aber nicht umgekehrt)



Beziehungen

- Gerichtete Beziehung (Pfeil), *Navigierbar* von einer Person zu seiner Adresse (aber nicht umgekehrt)



```

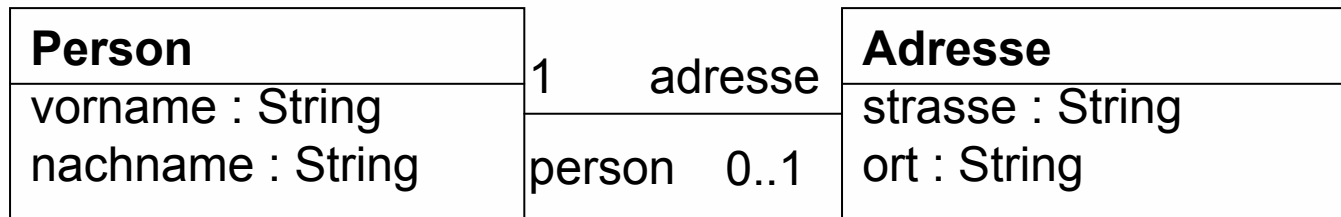
public class PersonTest {

    public static void main(String argv[])
    {
        Person person = new Person();
        // zur Adresse „navigieren“
        person.adresse = new Adresse();
        person.adresse.strasse = "Moltkestrasse";
    }
}
  
```



Beziehungen

- Ungerichtete Beziehung, *Navigierbar* von einer Person zu seiner Adresse und von Adresse zu seiner Person

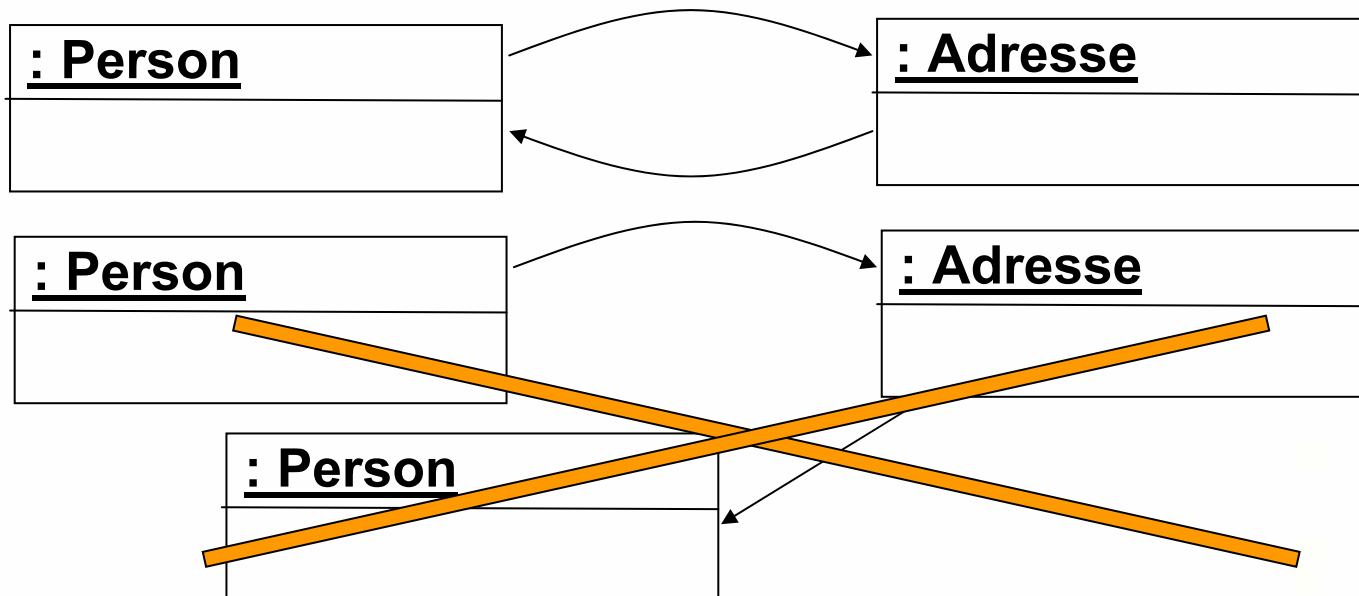
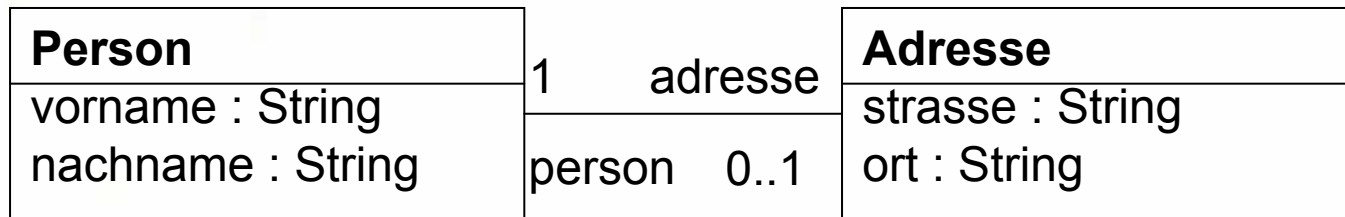


```
public class Person {
    public String vorname;
    public String nachname;
    public Adresse adresse;
}
```

```
public class Adresse {
    public String strasse;
    public String ort;
    public Person person;
}
```

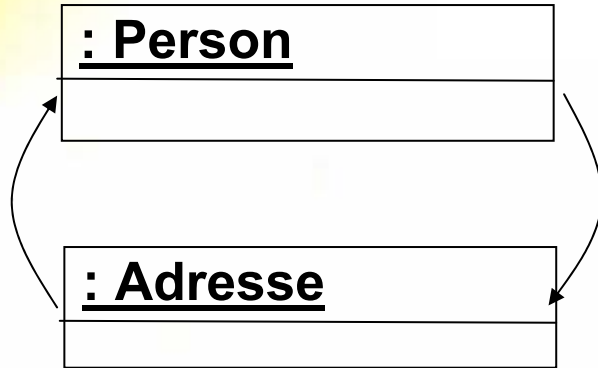
Beziehungen

- Adresse, die zur Person gehört muss über diese Beziehung ihrerseits zur Ursprungsperson gehören



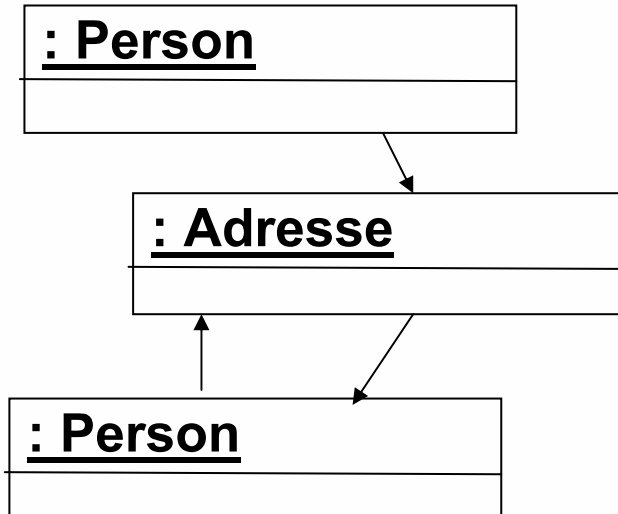


Beziehungen



```

Person person = new Person();
Adresse adresse = new Adresse();
person.adresse = adresse;
adresse.person = person;
// Konsistenter Zustand
// zu Klassenmodell
  
```



```

Person person1 = new Person();
Person person2 = new Person();
Adresse adresse = new Adresse();
person1.adresse = adresse;
adresse.person = person2;
person2.adresse = adresse;
// Inkonsistenter Zustand
  
```



Inhalt

- Attribute
- Beziehungen
- **Objekt-Methoden**
- Konstruktor
- Geheimnisprinzip
- static
- Java Packages



Objekt-Methoden

- Syntax und Deklaration wie schon besprochen (aber ohne static)
- Objekt Methoden
 - Werden „auf“ Objekt aufgerufen
 - Aufruf via Variablen/Attribut mit „.“
 - Programmkontext bei Ausführung Methode immer mit einem Objekt verbunden

```
public class PersonTest {  
  
    public static void main(String argv[])  
        Person meier = new Person();  
        Person mueller = new Person();  
  
        meier.heiraten(mueller);  
    }  
}
```




Objekt-Methoden

- Objekt Methoden

- Können via `Attributname` auf `Attributwerte` des Objektes zugreifen, auf dem sie aufgerufen worden sind

```
public class Person {  
    public String nachname;  
  
    public void heiraten(Person ehEGatte) {  
        nachname = ehEGatte.nachname + "-" + nachname;  
    }  
}
```



Verweis auf `nachname` des Objekts (`this`),
zu dem diese Methode aufgerufen wurde

Objekt-Methoden

```

public class Person {
    public String nachname;

    public void heiraten(Person ehegatte) {
        nachname = ehegatte.nachname + "-" + nachname;
    }
}

public class PersonTest {

    public static void main(String argv[])
        Person meier = new Person();
        Person mueller = new Person();
        meier.nachname = "Meier";
        mueller.nachname = "Müller";
        meier.heiraten(mueller);
    }
}

```

Methodenaufruf

Verweis auf

meier : Person

nachname = „Meier“

mueller : Person

nachname = „Müller“

Objekt-Methoden

```

public class Person {
    public String nachname;

    public void heiraten(Person ehegatte) {
        nachname = ehegatte.nachname + "-" + nachname;
    }
}

public class PersonTest {

    public static void main(String argv[]) {
        Person meier = new Person();
        Person mueller = new Person();
        meier.nachname = "Meier";
        mueller.nachname = "Müller";
        meier.heiraten(mueller);
    }
}

```

Methodenaufruf

Rückkehr

<u>meier : Person</u>
nachname = „Müller-Meier“

Zustand nach Ende
Programm

<u>mueller : Person</u>
nachname = „Müller“

Objekt-Methoden

```

public class Person {
    public String nachname;

    public void heiraten(Person ehegatte) {
        nachname = ehegatte.nachname + "-" + nachname;
    }
}

public class PersonTest {

    public static void main(String argv[])
    Person meier = new Person();
    Person mueller = new Person();
    meier.nachname = "Meier";
    mueller.nachname = "Müller";
    mueller.heiraten(meier);
}
}

```

Methodenaufruf

Verweis auf

meier : Person

nachname = „Meier“

mueller : Person

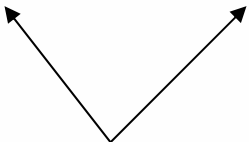
nachname = „Müller“



Objekt-Methoden

- Parameter (oder lokale Variable) *verdeckt* Attribut, falls Bezeichner identisch

```
public class Person {  
    public String nachname;  
  
    public void nachnameAendern(String nachname) {  
        nachname = nachname;  
    }  
}
```



Beides male ist dies der Parameter „nachname“

Der Parameter „nachname“ *verdeckt* das Attribut „nachname“



Objekt-Methoden

- Bei verdeckten Attributnamen: Zugriff auf Objekt via Schlüsselwort „this“
- „this“ bezeichnet im Kontext des Aufrufes befindliche Objekt
- Möglichst keine künstlichen Parameternamen einführen, um verdeckte Attribute zu vermeiden (nicht: String neuerNachname), sondern „this“ verwenden
- Manchmal findet man die Konvention bei Parametern oder lokalen Variablen den (un)bestimmte Artikel zu verwenden: aNachname, einNachname, theNachname, derNachname (bitte vermeiden: besser this)

```
public class Person {  
    public String nachname;  
  
    public void nachnameAendern(String nachname) {  
        this.nachname = nachname;  
    }  
}
```

Objektattribut „nachname“

Parameter „nachname“



Objekt-Methoden

- Aufruf von Objektmethoden
 - während der Initialisierung und
 - aus anderer Objektmethode heraus möglich

```
public class Person {  
    public String nachname = nachnameAendern("");  
  
    public void heiraten(Person ehEGatte) {  
        nachnameAendern(ehEGatte.nachname + "-" + nachname);  
    }  
  
    public void nachnameAendern(String nachname) {  
        this.nachname = nachname;  
    }  
}
```



Objekt-Methoden

```
public class Person {  
    public String nachname;  
  
    public void heiraten(Person ehedatte) {  
        nachnameAendern(ehedatte.nachname + "-" + nachname);  
    }  
    public void nachnameAendern(String nachname) {  
        this.nachname = nachname;  
    }  
    public static void nachnameAendern(String nachname) {  
        this.nachname = nachname;  
    }  
}
```

- Compilerfehler

- doppelte Methoden (static reicht zur Unterscheidung nicht aus)
- this im Kontext einer statischen Methode existiert nicht (macht gar keine Sinn)



Objekt-Methoden

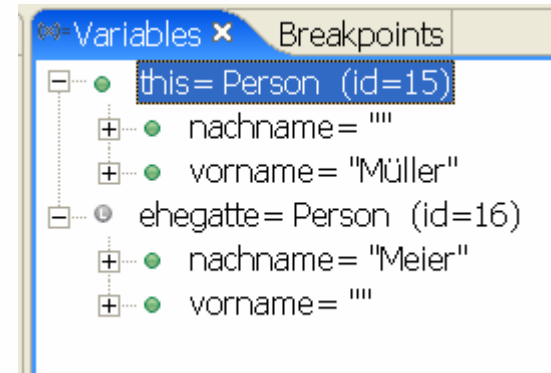
- Beispielausführung in Eclipse

```
public class Person {  
  
    public String vorname = "";  
    public String nachname = "";  
  
    public void heiraten(Person ehegatte) {  
        this.nachname = ehegatte.nachname + "-,,  
            + this.nachname;  
    }  
}
```

Objekt-Methoden

- Zustand des Programms bei Aufruf `heiraten`, *vor* Ausführung von Zeile 24

```
public static void main(String argv[]) {  
    Person meier = new Person();  
    Person mueller = new Person();  
  
    meier.nachname = "Meier";  
    mueller.nachname = "Müller";  
    mueller.heiraten(meier);  
}
```

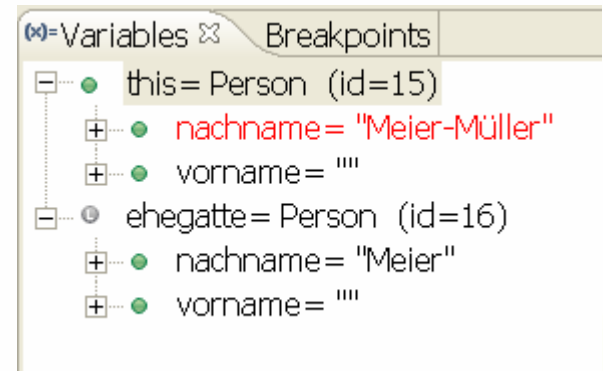


```
23 public void heiraten(Person ehegatte) {  
24     this.nachname = ehegatte.nachname + "-" + this.nachname;  
25 }
```

Objekt-Methoden

- Zustand des Programms bei Aufruf `heiraten`, *nach* Ausführung von Zeile 24

```
public static void main(String argv[]) {  
    Person meier = new Person();  
    Person mueller = new Person();  
  
    meier.nachname = "Meier";  
    mueller.nachname = "Müller";  
    mueller.heiraten(meier);  
}
```



The screenshot shows a debugger's Variables window with two tabs: "Variables" and "Breakpoints". The "Variables" tab is active, displaying a tree view of variables. The root variable is "this = Person (id=15)", which is expanded to show "nachname = 'Meier-Müller'" and "vorname = ''". Below it is another variable "ehegatte = Person (id=16)", which is also expanded to show "nachname = 'Meier'" and "vorname = ''".

```
23 public void heiraten(Person ehegatte) {  
24     this.nachname = ehegatte.nachname + "-" + this.nachname;  
25 }
```



Inhalt

- Attribute
- Beziehungen
- Objekt-Methoden
- **Konstruktor**
- Geheimnisprinzip
- static
- Java Packages



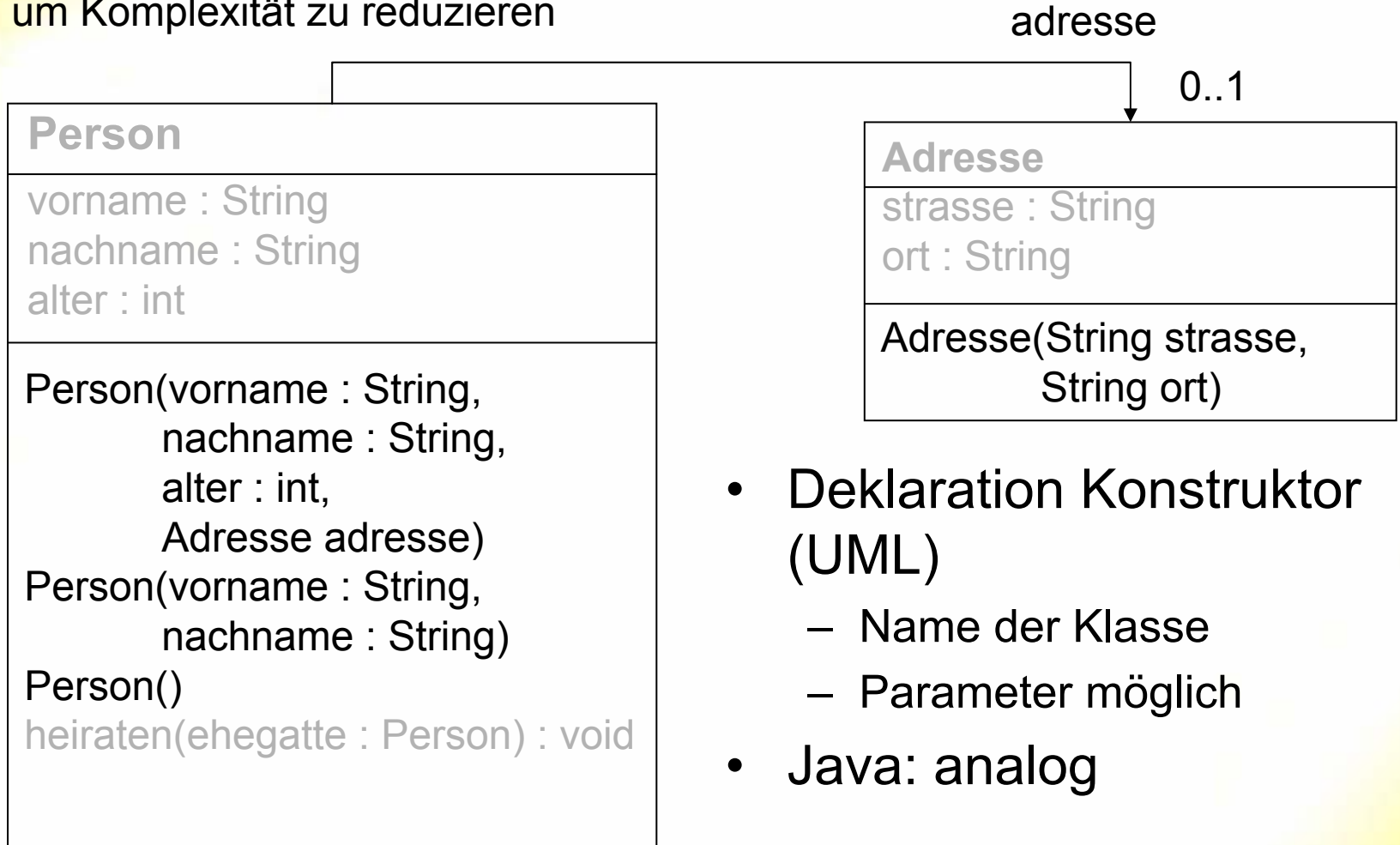
Konstruktor

- Default Konstruktor
 - Name der Klasse
 - Keine Parameter, z.B. Person()
 - Aufruf Konstruktor mit „new“, z.B. new Person()
 - Existiert immer, *es sei denn* ein Konstruktor wird explizit deklariert
- Konstruktor implementieren mit für die Initialisierung des Objekts *notwendigen* Werten
 - Alle wichtigen Attributwerte
 - Für Konsistenz der Beziehungen notwendigen Objekte (Voraussetzung ist ein Entwurf der Klassen z.B. mit UML Klassendiagramm)



Konstruktor

Nur eine Richtung der Beziehung im Entwurf gewählt,
um Komplexität zu reduzieren



- Deklaration Konstruktor (UML)
 - Name der Klasse
 - Parameter möglich
- Java: analog

Konstruktor

- Konstruktor, so dass Objekt von Typ Person mit Vor- und Nachnamen erzeugt wird

```
public class Person {
```

```
    public String vorname;
    public String nachname;
```

Name Klasse, kein Rückgabotyp

```
    public Person(String vorname, String nachname) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
}
```

```
    Person meier = new Person();
```

Compilerfehler: Default Konstruktor existiert nicht

meier : Person

vorname = „Hans“
nachname = „Meier“

Erzeugt

```
    Person meier = new Person("Hans", "Meier");
```



Konstruktor

```
public class Person {
```

```
    public String vorname = ""; // leerer String, statt null
    public String nachname = "Kein Name"; // unsinniger Wert
```

Sinnvolle Werte wählen

```
    public Person(String vorname, String nachname) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
```

```
    public Person() {
    }
```

```
}
```

```
Person meier = new Person();
```

```
new Person("Frau", "Musterfrau");
```

meier : Person

vorname = „“
nachname = „Kein Name“

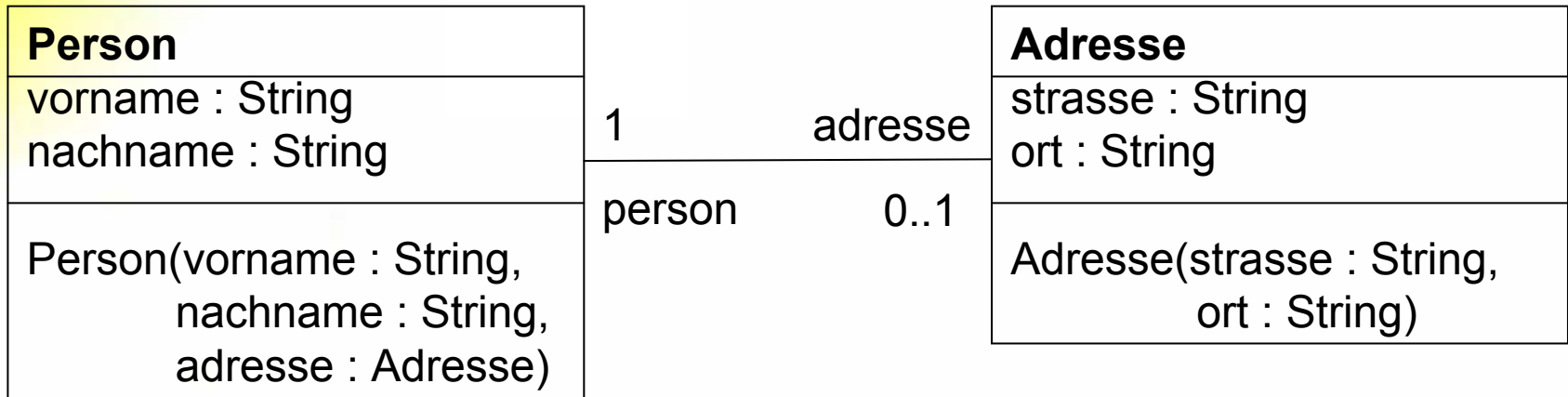
Erzeugt

: Person

vorname = „Frau“
nachname = „Musterfrau“



Konstruktor



```

public Person(String vorname, String nachname
                Adresse adresse) {
    this.vorname = vorname;
    this.nachname = nachname;
    // Stelle Konsistenz mit Beziehung
    // in UML Diagramm sicher
    // (aber: besser in der Datenbank abhandeln)
    this.adresse = adresse;
    if (adresse != null) {
        adresse.person = this;
    }
}

```



Konstruktor

- Konstruktor mit vorhandenen Konstruktor implementieren mit „this(...)“
- Möglichst einen umfassenden Konstruktor implementieren
 - alle anderen via „this“ schrittweise auf diesen zurückführen

```
public class Person {
    public String vorname;
    public String nachname;
```

```
    public Person(String vorname, String nachname) {
        this.vorname = vorname;
        this.nachname = nachname;
```

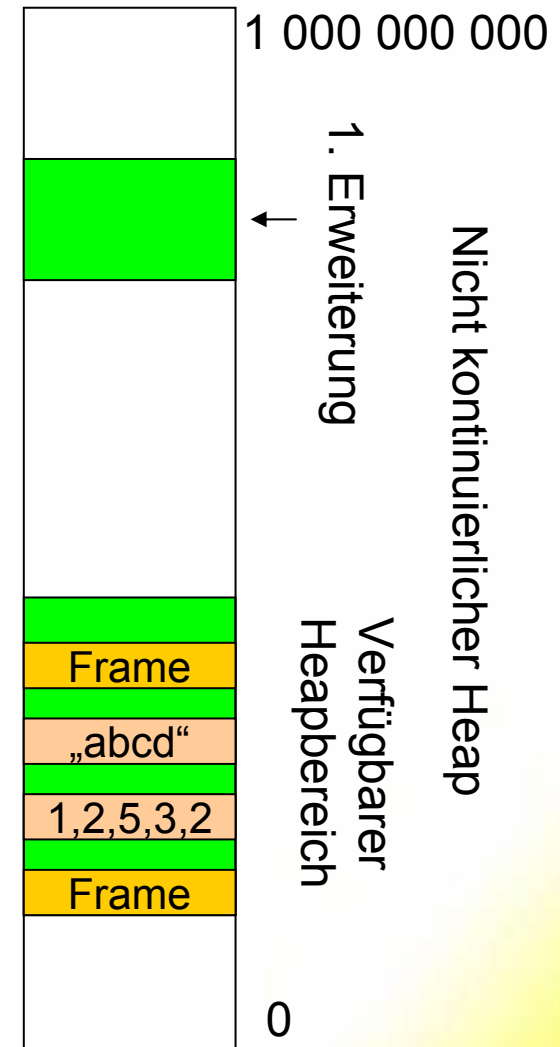
```
    }
    public Person() {
        this(““, ““);
    }
}
```

← this(...) muss – falls vorhanden –
erste Anweisung im Konstruktor sein

Konstruktor

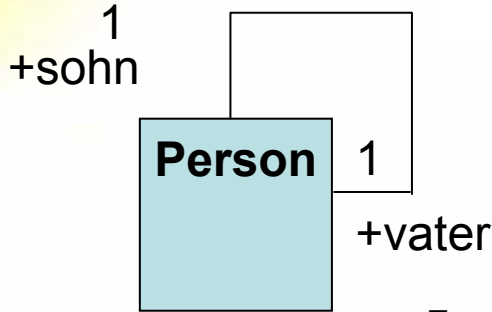
- Objekte (`new ...`) werden auf **Heap** (*Haufen*) angelegt
 - „The heap is the runtime data area from which memory for all class instances and arrays is allocated“ [Abschnitt 3.5.3 der JVM Spezifikation]
- Obergrenze Heapspeicher bei Java
 - „`java -Xmx128m`“ setzt Obergrenze auf 128MB
 - „`java -Xms2MB`“ setzt initiale Größe auf 2MB (wird dynamisch bis Obergrenze erhöht)
- Falls Obergrenze erreicht: `OutOfMemoryError`
- Heapbereich braucht nicht kontinuierlich zu sein
- Speicher auf Heap wird vom **Garbage Collector** (GC) periodisch überprüft
 - Speicher für Objekte, auf die keine Referenz von anderen Objekten, lokalen Variablen, Parameter mehr existiert, werden wieder freigeben
 - Implementierte Verfahren funktionieren nicht perfekt und sind nicht durch JVM Spezifikation definiert
 - Nicht verwendete Objekte können den Heap füllen (**Speicherleck**)

Hauptspeicher





Garbage Collector

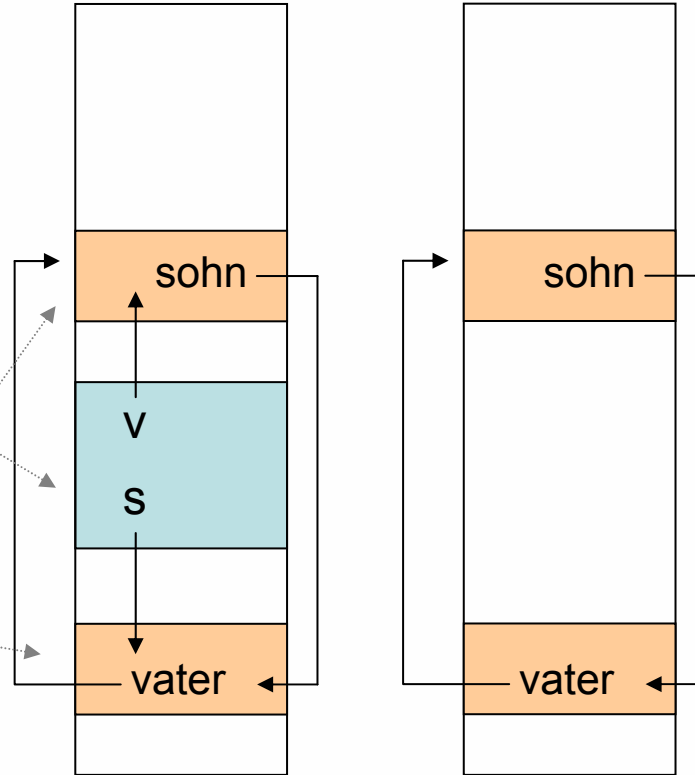


```

public void erzeugeVaterSohn() {
    Person v = new Person();
    Person s = new Person();

    v.sohn = s;
    s.vater = v;
}
    
```

Frame für
Methodenaufruf



Zustand kurz bevor
Methode verlassen wird

Zustand nachdem
Methode beendet wurde

Objekte verbleiben ggf. im Speicher
(obwohl gar nicht mehr genutzt)

- Beispiel gilt (zum Glück) nur bei den ersten Java Versionen
 - Das Problem bleibt aber für jede Java Version bestehen (Beispiele sind dann komplexer)
- Details z.B. [<http://java.sun.com/developer/Books/javaprogramming/bitterjava/bitterjavach06.pdf>]

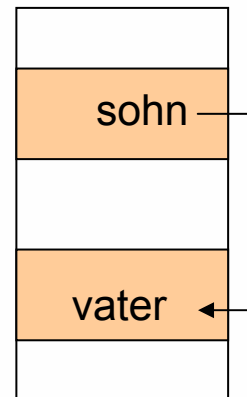
Garbage Collector

- StackOverflowError vermeiden
 - Rekursion nur verwenden, wenn Rekursionstiefe stark begrenzt ist
 - Obergrenze Stack bei Start JVM ggf. erhöhen
- OutOfMemoryError vermeiden
 - Programm mit realen Datenmengen testen
 - Obergrenze Heap bei Start JVM ggf. erhöhen
- Speicherlecks vermeiden
 - Programm mit realen Daten und über längere Zeit testen
 - Bei zirkulären Referenzen einen der Verweise auf null setzen, wenn Objekte nicht mehr verwendet werden

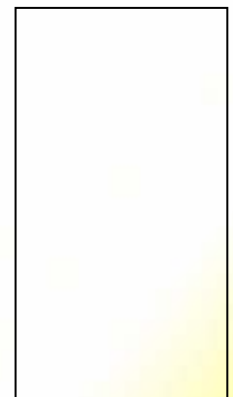
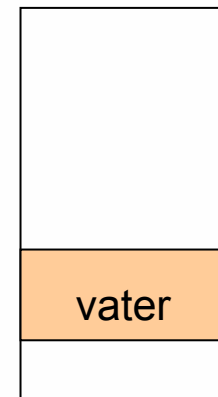
```
public void erzeugeVaterSohn() {
    Person v = new Person();
    Person s = new Person();

    v.sohn = s;
    s.vater = v;
    // etwas sinnvolles machen
    s.vater = null;
}
```

Keine Referenz auf
Vater



Vater freigeben
Keine Referenz
auf Sohn





Garbage Collector

- In anderen Programmiersprachen wie C, C++, PASCAL, ...
 - Programm muss dynamisch erzeugte Objekte / Daten explizit freigeben
- Probleme bei expliziter Speicherverwaltung durch Programmierer
 1. Speicher wird vergessen freizugeben (Speicherleck)
 2. Speicher wird mehrfach freigeben oder Speicher, der nicht mit new reserviert wurde, wird freigeben (Laufzeitfehler, free error)
 3. Es wird auf freigegebenen Speicher zugegriffen (Programmabsturz)
 4. Verweise auf gültige Speicherbereiche werden überschrieben (Programmabsturz)
- In Java (und .NET): Probleme 2 bis 4 gibt es dort nicht.

```
public void erzeugeVaterSohn() {  
    Person v = new Person();  
    Person s = new Person();  
  
    v.sohn = s;  
    s.vater = v;  
    // etwas sinnvolles machen  
    delete(v);  
    delete(s);  
}
```



Inhalt

- Attribute
- Beziehungen
- Objekt-Methoden
- Konstruktor
- **Geheimnisprinzip**
- static
- Java Packages



Geheimnisprinzip

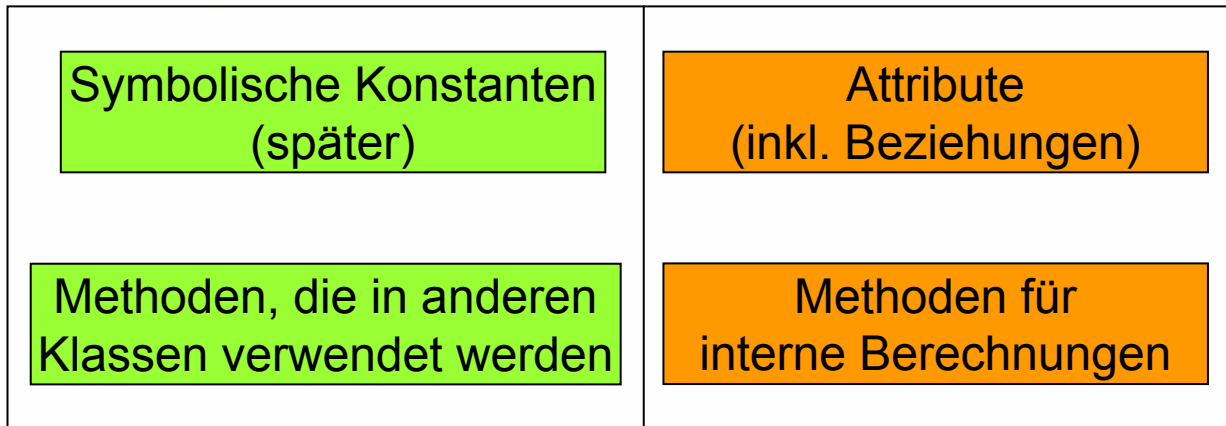
- engl. *information hiding*
- **Verbergen von Implementierungsdetails**
- **Unterscheidung: Innensicht und Außensicht einer Klasse**
- Ziele
 1. Programme weniger anfällig für Fehler machen
 2. Programme leichter änderbar machen
 3. Direkten Zugriff auf Objektattribute wie `hans.alter = -10` verbieten, da Alter einer Person immer muss positiv sein
- Implementierungsdetails sind (z.B.)
 - Alle Attributen/Beziehungen (Daten, Struktur)
 - Nur „innerhalb“ der Klasse verwendete Methoden
- Geheimnisprinzip in der Objekt-Orientierung
 - Verberge alle Attribute und „interne“ Methoden (**Datenkapselung**)
 - Greife ausschließlich über öffentliche Methoden auf Attribute zu
 - Methoden können Werte auf Korrektheit überprüfen



Geheimnisprinzip

Außensicht

Innensicht



- In anderen Klassen verwendete Methoden
 - Sollen sich nicht oft ändern, da sonst in sie vorkommende Klassen geändert werden
- Attribute ändern sich, Zugriff über Methoden soll sich nicht ändern
 - Alter einer Person von int auf double, um 15.5 Jahre abzubilden
 - Alter einer Person von int auf java.util.Date, um Alter aus aktuellem Datum zu berechnen

Geheimnisprinzip

- Geheimnisprinzip in der Objekt-Orientierung
 - Verberge Attribute mit Schlüsselwort „private“ statt „public“
 - Greife ausschließlich über öffentliche Methoden auf Attribute zu
 - Implementiere Konsistenzbedingungen in Zugriffsmethoden / Konstruktor
- Namenskonventionen für *Zugriffsmethoden (getter/setter Methoden)*
 - Attributname X Typ Y
 - Methode, um Wert auszulesen „public Y getX()“
 - Methode, um Wert zu ändern „public void setX(Y y)“
 - Spezialfall Typ boolean: boolean isX(), setX(boolean y)

```
public class Person {  
    private int alter = 0;  
    public int getAlter() {  
        return alter;  
    }  
    public void setAlter(int alter) {  
        this.alter = alter;  
    }  
}
```

Sichtbarkeit public - private

```
public class Person {  
    private int alter = 0;  
    public int getAlter() {  
        return alter;  
    }  
    public void setAlter(int alter) {  
        this.alter = alter;  
    }  
}
```

```
public class PersonTest {  
    public static void main(String argv[]) {  
        Person person = new Person();  
        person.alter = 17; // Compilerfehler  
        person.setAlter(17); // OK  
    }  
}
```

Sichtbarkeit public - private

```
public class Person {
```

```
private int x;
```

```
private void aendereY() {
    // Zugriff auf x und y erlaubt
    x = 17;
}
```

```
public int y;
```

```
public void aendereX() {
    x = y; // Zugriff auf x und y erlaubt
    aendereY();
}
```

```
public void erzeugePerson() {
    Person person = new Person();
    person.x = 7; // erlaubt
}
```

```
}
```

```
public class AndereKlasse {
```

```
public void machewas () {
    Person person = new Person();
```

```
// Kein Zugriff auf x
// oder aendereY()
// Compilerfehler:
```

```
person.aendereY();
person.x = 17;
```

```
person.aendereX();
person.y = 5;
erzeugePerson();
```

```
}
}
```

Widerspricht noch
dem Geheimnisprinzip



Geheimnisprinzip

- Konsistenzbedingung (Geschäftslogik)
 - Durch Problemstellung (Anforderungen) definierten Regeln
 - Fehlerbehandlung bei Konsistenzverletzung durch Ausnahmen (Exceptions, in Informatik II)
- Beispiele
 - Geburtsdatum darf nicht in der Zukunft liegen
 - Kliniksoftware: Geburtsdatum darf nicht in der (weiten) Vergangenheit liegen
 - Einwohnermeldeamt: Person muss immer eine Adresse besitzen
 - Rechnungssumme muss Total der einzelnen Rechnungspositionen sein



Geheimnisprinzip

- Änderung an Person
 - Nicht mehr „alter“, sondern „geburtsDatum“ speichern, *ohne dass aufrufende Programmteile geändert werden müssen*

```
public class Person {
```

```
    private Date geburtsDatum;
```

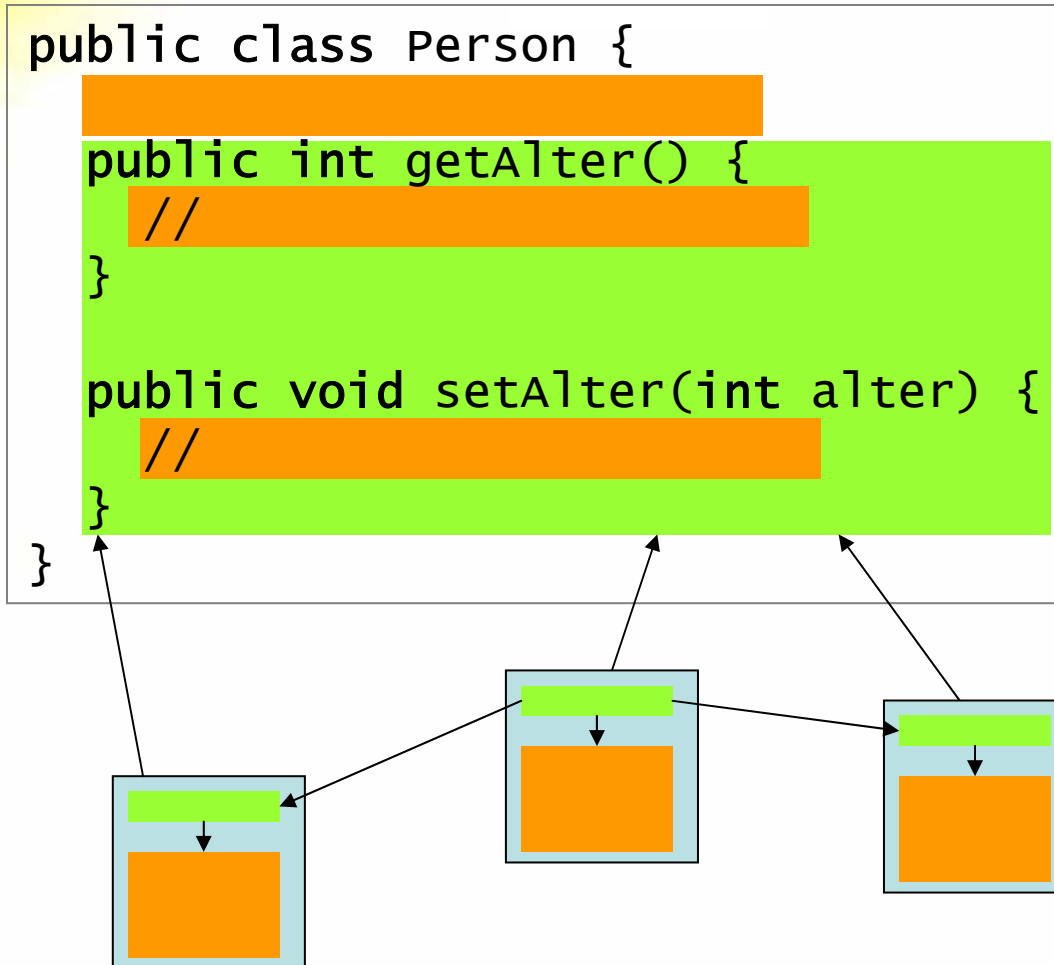
```
    // zugriffsmethoden set/getGeburtsDatum() fehlen
```

```
    public int getAlter() {  
        return new Date().getYear()  
            - getGeburtsDatum().getYear();  
    }
```

```
    public void setAlter(int alter) {  
        // keine sinnvolle Implementierung möglich  
        // Aufrufe im Code ändern  
        // + Laufzeitfehler erzeugen  
    }
```

```
}
```

Geheimnisprinzip

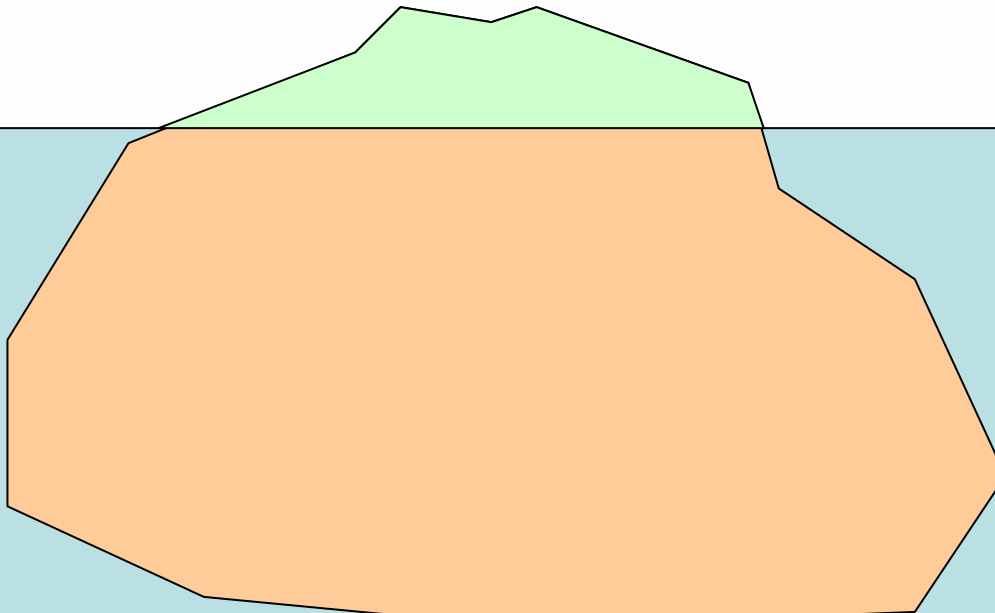


- Abhängigkeiten zwischen Klassen nur noch im öffentlichen Teil
- Abhängigkeiten zum nicht öffentlichen Teil nur innerhalb der Klasse selbst
- Programme werden wartbarer:
 - Änderungen haben oft nur lokale Auswirkung
 - Änderungen oft nur innerhalb einer Klasse



Geheimnisprinzip

- Entwurf / Implementierung
 - Möglichst viel geheim halten
 - Öffentlicher Teil einer Klasse sollte nur Spitze des Eisberges darstellen (10%)





Geheimnisprinzip

- Ab heute:
 - Alle Attribute (bis auf folgende Ausnahmen) immer „private“ deklarieren
 - Nur über Methoden auf Attribute zugreifen
 - Zugriffsmethoden mit Präfix set / get
 - Nur getter- und setter-Methoden implementieren, die auch verwendet werden
- Bei Methoden / Konstruktor
 - Nur „public“ deklarieren, wenn sie von anderen Klassen aus verwendet werden (sollen); ansonsten „private“ deklarieren



Geheimnisprinzip

- Namenskonventionen get/set gelten auch für „abgeleitete“ Attribute:
 - Wert ist nicht bei Objekt gespeichert, sondern wird aus anderen Attributen und / oder Objektmethoden berechnet

```
public class Person {  
  
    private int alter;  
  
    public int getAlter() {  
        return alter;  
    }  
  
    // aus „alter“ abgeleitete Eigenschaft  
    // setVolljaehrig() dazu gibt es nicht  
    public boolean isVolljaehrig() {  
        return getAlter() >= 18;  
    }  
}
```



Geheimnisprinzip

- Modifier (private, public) in UML
 - public durch vorangestelltes „+“
 - private durch vorangestelltes „-“
- Gelten auch für Klassen

Person
-vorname : String -nachname : String
+Person(vorname : String, nachname : String) +heiraten(ehegatte : Person) void +getVorname() : String +setVorname(vorname : String) : void +getNachname() : String +setNachname(nachname : String) : void



Geheimnisprinzip

- Geheimnisprinzip
 - Keine Implementierungsdetails in der Javadoc verraten
 - „Eine Person mit seinem Alter als ~~int Wert~~“
 - Öffentliche Getter- und Settermethoden verstoßen oft gegen das Geheimnisprinzip
 - Getter: Eine Methode, die bestimmte Information zurückgibt, um damit etwas zu tun
 - Öffentliche Methoden, sollen nicht einfach interne Information zurückgeben, sondern komplexes Verhalten implementieren
 - Dazu wird die Information anderer Objekte oft benötigt: nur für diese Informationen die Getter- und Settermethoden implementieren
 - Nur die wirklich benötigten Getter- und Settermethoden implementieren

Geheimnisprinzip

- Vorsicht bei Rückgabe oder Setzen von Objektreferenzen
 - Ggf. Kopien in getter- und setter Methoden erstellen
 - Trade-off: performance/memory und Wartbarkeit
 - Manche Klassen implementieren dazu die clone() Methode

```
Person person = ...
Person.getGeburtsDatum()
    .setYear(-1000);
// person nun inkosistent
```



```
public class Person {

    public Date getGeburtsDatum() {
        return geburtsDatum;
    }

    public void setGeburtsDatum(
        Date geburtsDatum) {
        if ( /* Datum prüfen */ ) {
            this.geburtsDatum = geburtsDatum;
        }
    }
}
```

```
public class Person {

    private Date geburtsDatum;

    public Date getGeburtsDatum() {
        return geburtsDatum.clone();
    }

    public void setGeburtsDatum(Date geburtsDatum) {
        if ( /* Datum pruefen */ ) {
            this.geburtsDatum = geburtsDatum.clone();
        }
    }
}
```

```
Person.getGeburtsDatum().setYear(-1000);
// person weiterhin konsistent (nur Kopie geändert)
```



Inhalt

- Attribute
- Beziehungen
- Objekt-Methoden
- Konstruktor
- Geheimnisprinzip
- **static**
- Unterschied elementare Typen - Klassen
- Java Packages



static

- Vermeide static, da sonst die Vorteile der Objekt-Orientierung verloren gehen
 - Keine Vererbung möglich
 - Keine Datenkapslung
 - Erweiterungen aufwendig
 - Viele Methoden müssen geändert werden
 - Es kommen oft unnötig Methoden hinzu



Prozedurale Programmierung

- Vertreter
 - ALGOL, ADA, C, Modula II, PASCAL, ...
- Prozedurale Programmierung
 - Daten und Programm getrennt
 - Datentypen werden definiert
 - Prozeduren („statische Methoden“)
manipulieren Daten von definierten Typ
 - Datum als Argument übergeben



OO- Entwurf + Prozedurale Programmierung / Beispiel „C“

Datei „geschwindigkeit.h“
mit Datentypen („Objekt“)

```
typedef struct Geschwindigkeit {
    double kiloMeterProStunde;
}
```

Geschwindigkeit

-kiloMeterProStunde : double

+getMilesPerHour() : double
+getMeterProSekunde() : double



Datei „geschwindigkeit.c“
mit Programm („Objekt-Methoden“)

```
#include "geschwindigkeit.h"

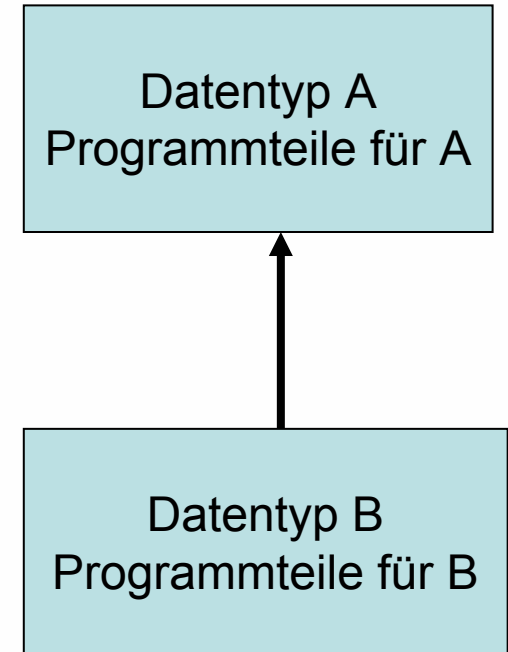
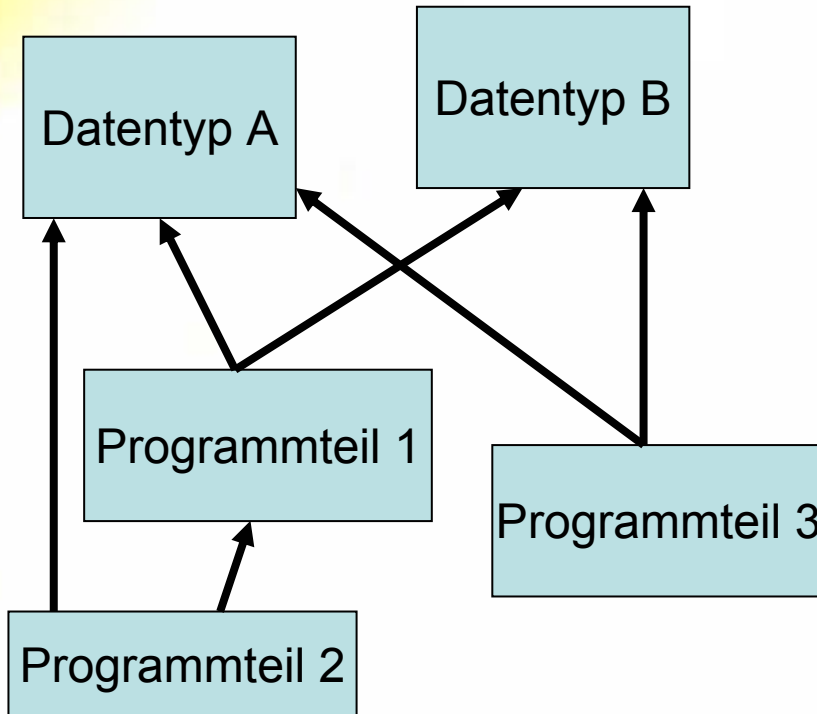
double getMilesPerHour(Geschwindigkeit this) {
    return this.kiloMeterProStunde * 0.621;
}

double getMeterProSekunde(Geschwindigkeit this) {
    return this.kiloMeterProStunde / 3.6;
}
```



„this“ ist *kein* Schlüsselwort in C

Prozedural – Objekt-orientiert



- Prozedurale Trennung von Datentyp und Programm führt meist zu vielen Abhängigkeiten (Ausnahme: Modulkonzept von Modula II)
- Objekt-Orientierung kann Abhängigkeiten vermeiden / vermindern



static / final

- Verwende `static` möglichst nur bei unveränderlichen oder konstanten Eigenschaften
 - Zahl Pi
 - Umrechnungsfaktoren 0.621
- Zusätzliches Schlüsselwort „`final`“
 - Eigenschaft wird bei Deklaration mit Wert initialisiert
 - Wert kann nicht mehr geändert werden (unveränderlich, konstant)
 - Auch bei Deklaration von lokalen Variablen und Parametern möglich



static / final

```
public class Math {  
    public static final double PI = 3.14157;  
    ...  
}  
  
public class Short {  
    public static final short MAX_VALUE = 32767;  
    public static final short MIN_VALUE = -32768;  
    ...  
}
```

Namenskonvention

Schreibe Namen für Konstanten immer mit Grossbuchstaben.

Verwende „_“ bei Teilwörtern

Konvention

Verwende symbolische Konstanten für alle Werte, die etwas bestimmtes bedeuten (Codes)



static / final

```
System.out.println( Math.PI );
```

```
// Compilerfehler, Wert darf nicht geändert werden:  
Math.PI = 3.0;
```

```
Short aShort = new Short(435);
```

```
// Konstanten nicht über Objekt referenzieren  
zahl = aShort.MAX_VALUE;  
// sondern immer via Klasse referenzieren  
zahl = Short.MAX_VALUE;
```



final

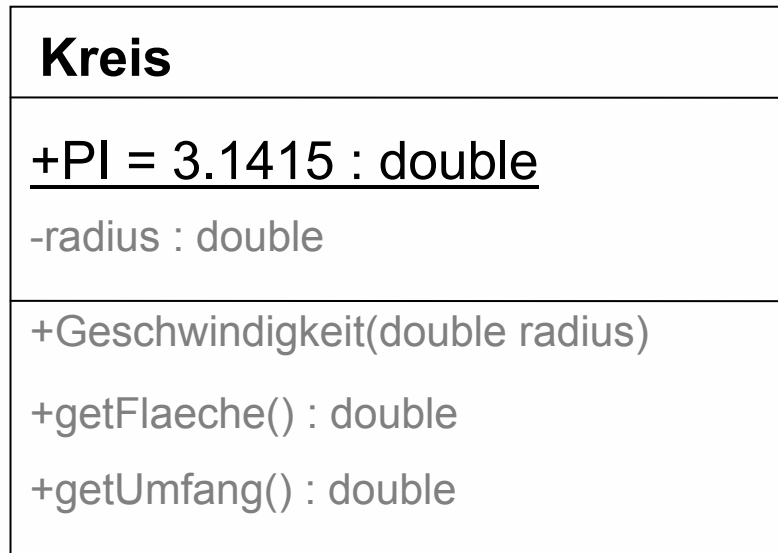
```
public class Person {  
  
    private String nachname;  
  
    // Möglich, aber in der Praxis eher unüblich  
    public void heiraten(final Person ehEGatte) {  
        final String nachname = this.namenname + "-"  
                               + ehEGatte.nachname;  
  
        this.nachname = nachname;  
        ehEGatte.nachname = nachname;  
  
        nachname = " "; // Compilerfehler  
        ehEGatte = null; // Compilerfehler  
    }  
  
}
```

.....



static

- Statische Methoden und Eigenschaften werden in der UML *unterstrichen*





Inhalt

- Attribute
- Beziehungen
- Objekt-Methoden
- Konstruktor
- Geheimnisprinzip
- static
- **Java Packages**



Java Packages

- Jeder Typ (Klasse) ist einem Paket (engl. Package) zugeordnet
- Ein Paket ist eine Sammlungen zugehöriger Klassen
 - Datenstrukturen (java.lang, java.util)
 - User Interface Komponenten (javax.swing, java.awt)
- Java (J2SE) hat Dutzende von Paketen und Hunderte von Klassen
- Typen (Klassen) müssen aus einem Paket importiert werden, damit sie verwendet werden können
- Pakete sind hierarchisch (ähnlich Dateisystem)
 - Folgen von Bezeichnern mit „.“ getrennt.
 - java.util, java.lang (sind in Java vorhanden)
 - pape.christian.vorlesung (ist gültig, muss selbst zur Verfügung gestellt werden)



Java Packages

1. Import aller Klassen eines Pakets

Schlüsselwort

```
import java.util.*;  
public class Person {  
    Date geburtsDatum;  
}
```

2. Import einer Klasse eines Pakets

```
import java.util.Date;  
public class Person {  
    Date geburtsDatum;  
}
```

3. Verwendung einer Klasse ohne Import

```
public class Person {  
    java.util.Date  
        geburtsDatum;  
}
```



Java packages

- Eigene Klassen sollten ebenfalls in einem Paket deklariert werden
- Eindeutigkeit wichtig
- Es sind nur Buchstaben/Zahlen erlaubt
- Beispiele
 - `ch.swisscom.it.pdl, de.sap`
 - `de.hskarlsruhe, de.hskarlsruhe.fbi`
 - `de.hska.fbi.info1.beispiele`

Konventionen Paketnamen

java, javax sind nicht erlaubt

Schreibe alles klein

Verwende (jeweils in umgekehrter Reihenfolge)
-zuerst **Internet Domain**
-dann **Organisationseinheiten**
-dann Projekt/Produktname
und/oder **logische Namen**

Wähle kurze Namen



Java packages

```
package de.fhkarlsruhe.fbi.info1;
```

```
import java.util.*;  
import java.lang.String;
```

Schlüsselwort für Name des Pakets
(nur eines erlaubt)

```
public class Person {
```

Schlüsselwort für Importe
(mehrere erlaubt)

```
    private String vorname;
```

```
    private String nachname;
```

```
    private Number alter;
```

import java.lang ist immer implizit
vorhanden und braucht nie explizit
angegeben zu werden

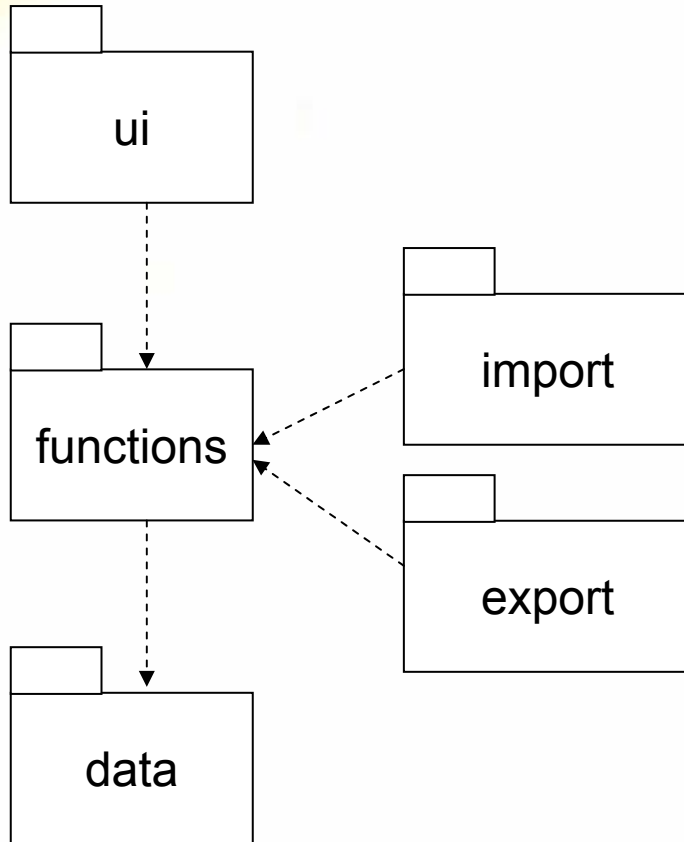
```
    private Date geburtsDatum;
```

Number, Date in java.util

```
}
```



Packages in UML



- Packages „enthalten“ Klassen oder andere Packages
- Abhängigkeiten zwischen Paketen:
 - Paket A abhängig von Paket B, wenn A Klasse enthält, die von einer Klasse in B abhängt
- Entwurfsziel
 - Pakete mit wenig Abhängigkeiten entwerfen



Java Packages + Sichtbarkeit

- Zwei weitere Modifier
 - *protected* (UML „#“)
Methoden, Attribute einer Klasse X sind sichtbar von allen Klassen, die Subklassen von X sind oder sich im gleichen Paket wie X befinden
 - „nichts angegeben“
package visibility, Methoden, Attribute einer Klasse X sind sichtbar von allen Klassen, die sich im gleichen Paket wie X befinden
- Package visibility vermeiden
- Protected höchstens bei Vererbung