Inhalt

- Java Pakete
- Java Klassen
  - Deklaration
  - Erzeugen von Objekten
  - Dynamischer Speicher (Heap)
  - Verweissemantik vs Wertesemantik
  - Ausführung von Objektmethoden
- Dokumentation mit Javadoc
- Testen mit JUnit
- Entwurf mit MVC-Paradigma

Java Paktete

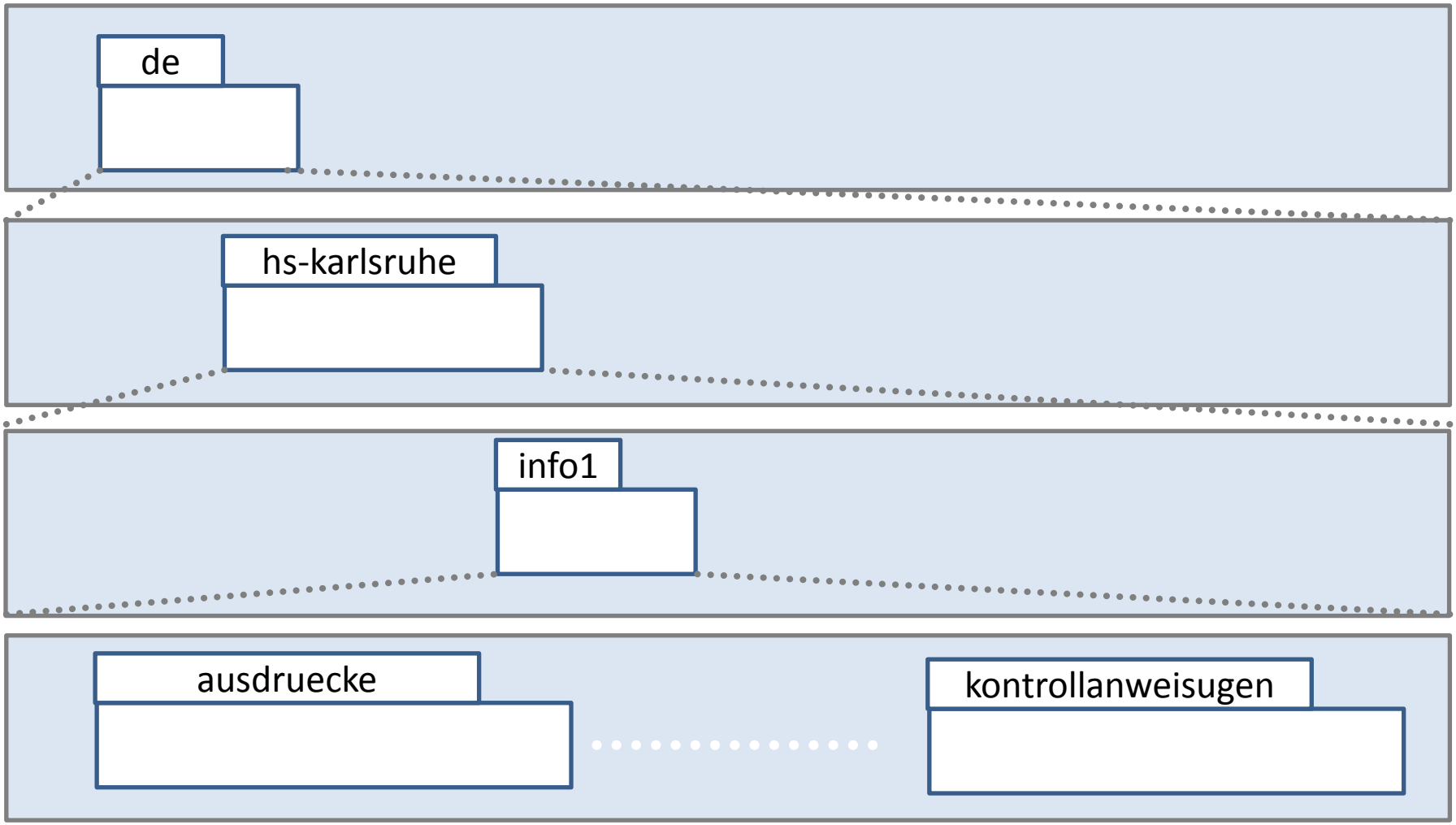
- Eine Klasse wird über eine Paketdeklaration vor der Klassendefinition *logisch* einem Paket zugeordnet
  - Klassen ohne Paketdeklaration sind dem default-package zugeordnet
  - Pakethierarchie wird mit . zwischen Namen der Pakete angegeben
  - Paketnamen folgen den Regeln von Bezeichner
  - Standard *physikalische* Hierarchie
    - Dateihierarchie mit Ordnern und Struktur der logischen Hierarchie
- Konventionen
  - Alles kleinschreiben, _ zum Trennen verwenden (wegen Dateisystemnamen)
  - Wegen der Länge sind Abkürzungen erlaubt
  - Um Paketnamen eindeutig zu machen, sollte die Hierarchie mit einem umgekehrten Domainnamen beginnen
  - java und javax als oberste Paketnamen sollen nicht verwendet werden

```
package personen;

public class Person {
  // ….
}
```

```
package de.hskarlsruhe.info1.personen;

public class Person {
  // ….
}
```

Java Pakete

- Informatik 1 Klassen Hierarchie



Java Paktete

- Klassen im default-Paket und java.lang sind überall sichtbar
- Klassen aus anderen Paketen müssen mit **import-Anweisung** sichtbar gemacht werden
- Zur Vermeidung von Namenskonflikten bei gleichen Klassennamen

  java.util.Date java.sql.Date

```
package de.hs-karlsruhe.info1.personen;
import java.util.Scanner;

public class Person {
    public static void main(String [] s) {
       Scanner scanner;   // Klasse Scanner sichtbar
    }
}
```

Java Pakete

- Alle Klassen aus einem Paket p können mit import p.* sichtbar gemacht werden
- Vermeiden (Namenkonflikte)

```
package de.hs-karlsruhe.info1.personen;
import java.util.*;

public class Person {
    public static void main(String [] s) {
        Date date;    // Date aus java.util
    }
}
```

```
package de.hs-karlsruh
import java.util.*;
import java.sql.*;

public class Person {
    public static void
        Date date;    // Namenskonflikt
    }
}
```

Java Pakete

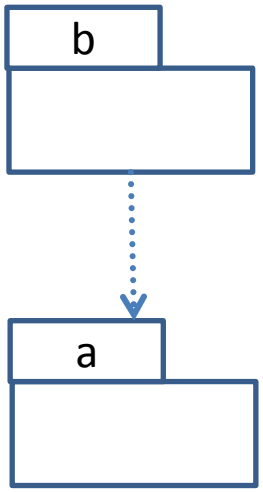
- Voll qualifizierter Klassenname:
  - Paketname gehört zum Klassenname

    java.util.Date de.hs-karlsruhe.info1.personen.Person
- Der voll qualifizierter Klassenname kann überall ohne import verwendet werden
  - Nötig, um Klassen gleichen Namens aus unterschiedlichen Paketen in einer Klasse zu verwenden

```
package de.hs-karlsruhe.info1.personen;
import java.util.*;

public class Person {
    public static void main(String [] s) {
        Date date1; // java.util.Date
        java.sql.Date date2;
    }
}
```

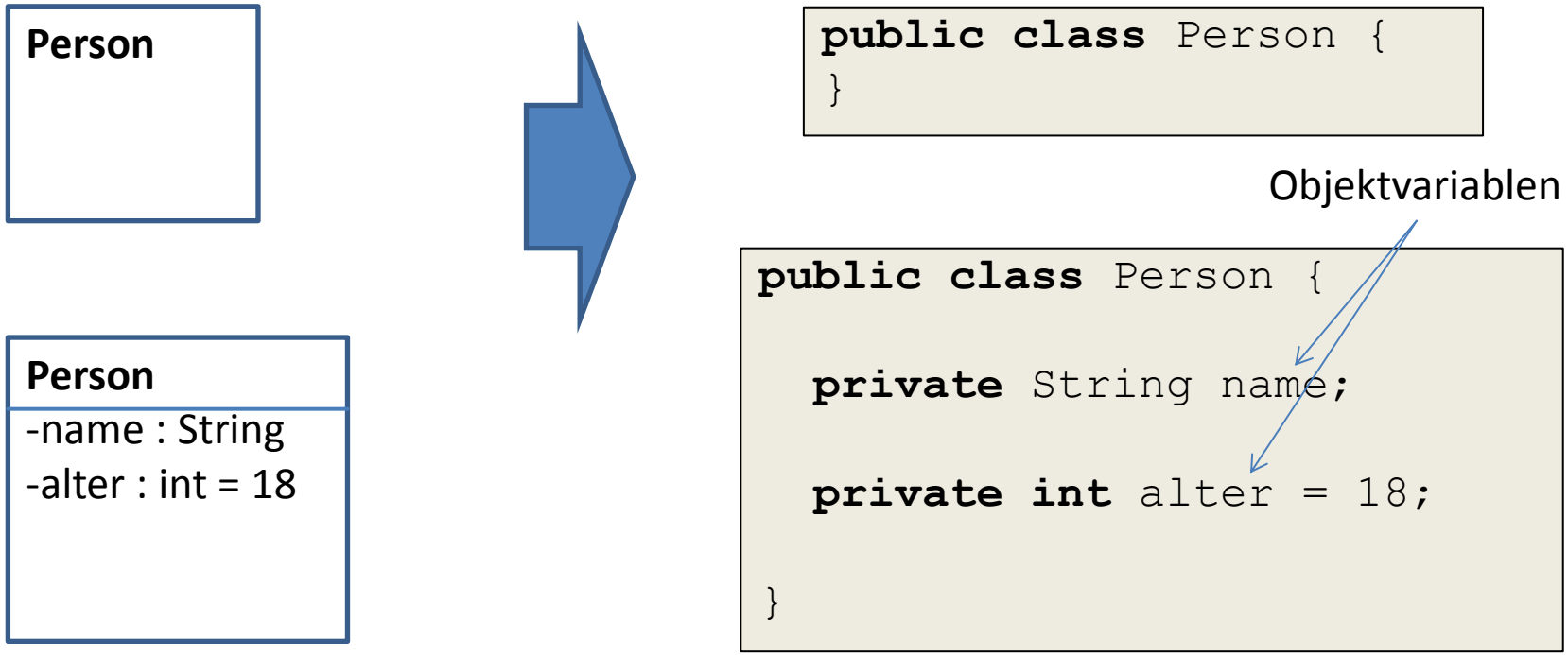
Java Pakete

- Zusammengehörige Klassen in einem Paket sammeln
  - *die einem gleichen Zweck dienen oder wiederverwendbar sind*
  - *Abhängigkeiten zwischen Klassen verschiedener Pakete reduzieren*
- Ein Paket sollten nicht mehr als 20-25 Klassen enthalten
  - Bei wachsender Anzahl in Teilpakete aufteilen
- Mit wenigen Paketen beginnen
  - Leichtes Hinzufügen und Ändern mit Entwicklungsumgebung



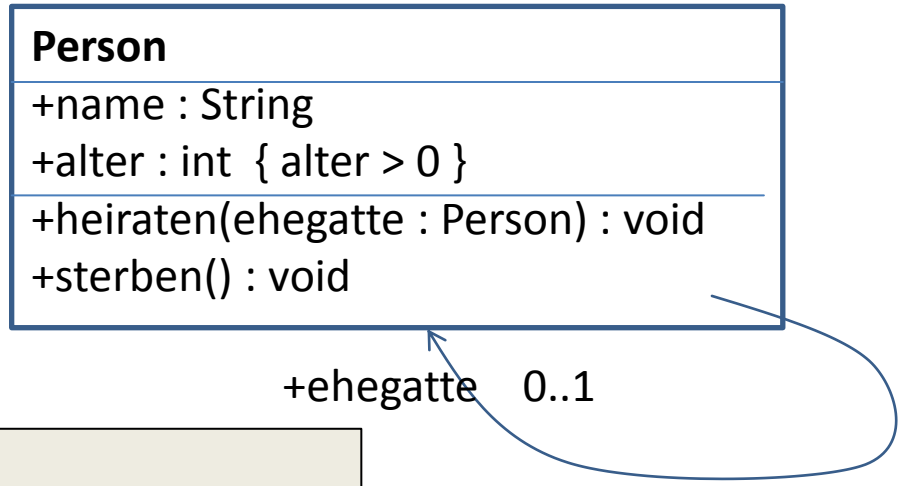
Java Klassen

- Objektvariablen mit Modifier, Datentyp, Bezeichner und optionaler Initialsierug deklarieren
  - ohne Schlüsselwort static
  - Reihenfolge spielt keine (große) Rolle



Java Klassen

- Objektvariablen für Beziehungen verwenden
- Häufigkeiten gibt es nicht (programmatisch lösen)
- Objektemethoden ohne static
  - Reihenfolge spielt keine Rolle




```
public class Person {
  public String name;
  public int alter = 18;
  public Person ehegatte; // optional

  public void heiraten(Person ehegatte) {
  }

  public void scheiden() {
  }
}
```

Java Klassen

- Objekte werden zur Laufzeit erzeugt
  - Daten eines Objekts verbrauchen Speicher für die Objektvariablen
  - Beliebig (aber endlich) viele Objekte möglich
  - Erzeugen mit Konstruktor-Aufruf (Ausdruck)
    - **Syntax**: Schlüsselwort new gefolgt von Konstruktor

```
new Person()
```

    - **Semantik**: Objekt wird im Heap angelegt, mit Werten initialisiert und ein Verweis auf das Objekt wird zurückgegeben
  - Der Verweis kann in einer Variablen mit Datentyp Person gespeichert werden
- Dynamischer Speicherbereich (Heap)
  - enthält in Java alle Objekte

Java Klassen

- Konstruktor
  - Hat immer den Namen der zugehörigen Klasse
  - Kann Parameter bei Aufruf übergeben bekommen
  - Es können eigene Konstruktoren implementiert werden
  - Sie werden wie bei gleichnamigen Methoden anhand der Datentypen der deklarierten Parameter unterschieden
- *Standard-Konstruktor*
  - Hat die Sichtbarkeit der Klasse
  - Existiert nur, wenn kein Konstruktor explizit programmiert wird
  - Hat keine Parameter
- Stellt sicher, dass von einer Klasse immer Objekte erzeugt werden können

Java Klassen

- Zugriff auf Objektvariablen über Variablennamen mit . (Punktoperator)

```
public class PersonMain {
  public static void main(String [] s) {
     Person meier;
     Person mueller;
     meier = new Person();
     mueller = new Person();
     meier.name = "Meier";
     meier.alter = 27;

     mueller.name = "Müller";
     mueller.alter = 75;
     System.out.println( meier.name );
  }
}
```

Java Namen

- Objektinitialisierung
  1. **Nach** Aufruf aber **vor** Ausführung des Konstruktors wird ein Objekt auf dem Heap erzeugt
  2. Falls nicht genügend Speicher vorhanden ist, bricht das Programm mit einem OutOfMemoryError ab
  3. Alle Objektvariablen werden mit Ihren Standardwerten initialisiert
  4. Dann erst wird der Rumpf des Konstruktor ausgeführt

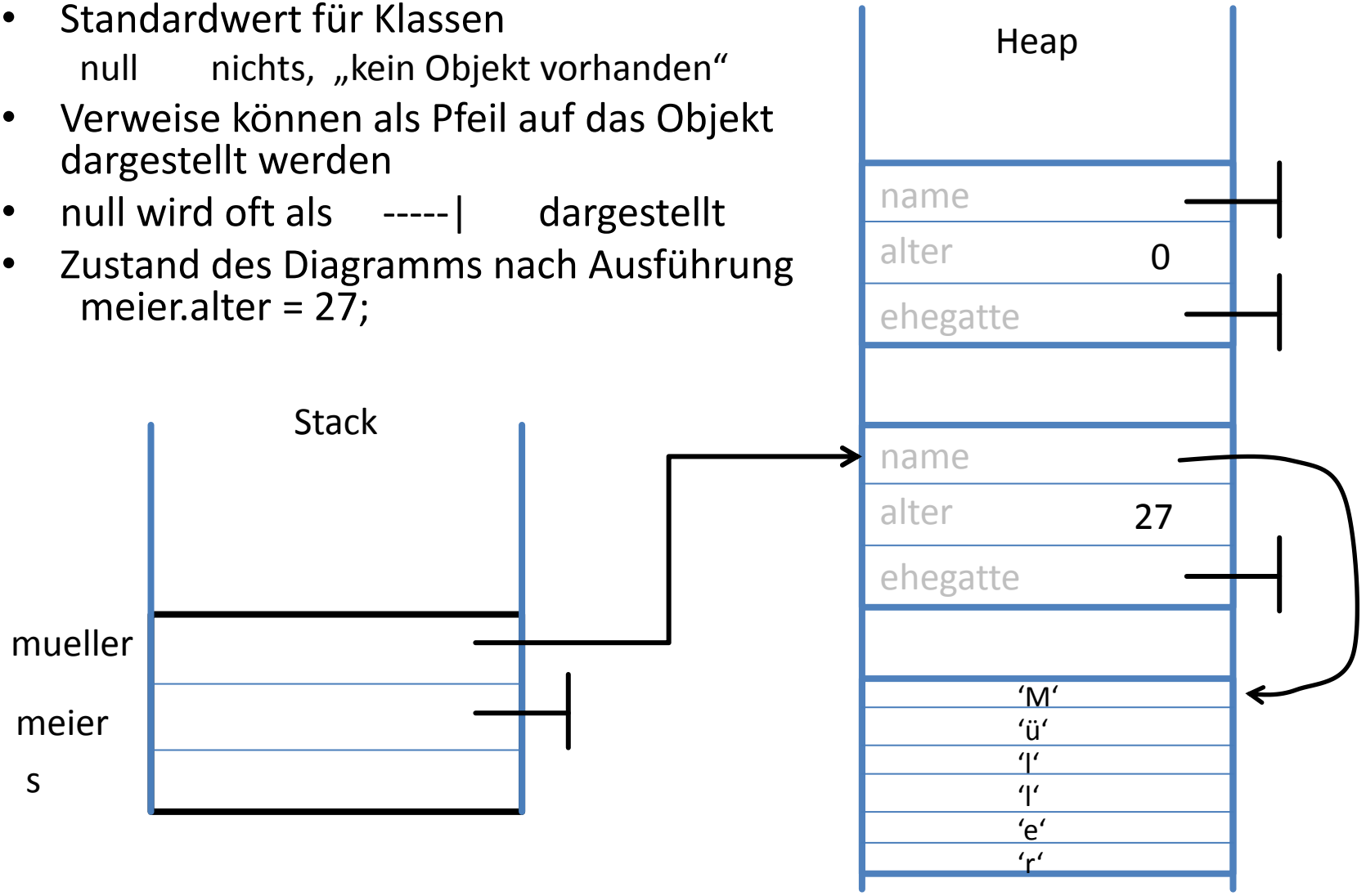
     Alle Initialisierungen der Objektvariablen von oben nach unten durchgeführt (später etwas genauer)

     Dann wird die erste Zeile des Rumpfs ausgeführt

     (beim Standard-Konstruktor enthält dieser keine Anweisungen)

Java Klassen

- Standardwert für Klassen
  null nichts, „kein Objekt vorhanden“
- Verweise können als Pfeil auf das Objekt dargestellt werden
- null wird oft als -----| dargestellt
- Zustand des Diagramms nach Ausführung
  meier.alter = 27;



Java Klassen

- Wertesemantik (*call by value*), bei primitiven Datentypen
  - Es werden **Kopien von Werten** als Parameter übergeben, bei Zuweisung einer Variablen zugewiesen und bei einem Funktionswert zurückgeben
- Verweissemantik (*call by reference*), bei Java Klassen
  - Es werden **Verweise auf einen Wert** übergeben, zugewiesen oder zurückgegeben
- In C++
  - Semantik kann bei Deklaration gewählt werden

```
int a;  // Variable a enthält einen int-Wert
int & b; // Varable b entählt Verweis auf int-Wert
b = a;  // Verweis von a wird nach b kopiert
b = 2;  // Wert von a ist 2
```

Java Klassen

- Beispiel
  - **Klassenmethode** zum Ändern des Alters einer Person
  - Parameter Person und neues Alter
  - Rückgabe der Person als Funktionswert

Person
+name : String +alter : int
+heiraten(ehegatte : Person) : void +sterben() : void +aendereAlter(person : Person, alter : alter) : Person

```
public static Person aendereAlter(Person person, int alter) {
  person.alter = alter;
  alter = 0;
  return person;
}
```

Java Klassen

- Verweissemantik bei Klassen
  - Keine praktische Einschränkung
  - Wertesemantik kann immer nachgeahmt werden, indem im Programm explizit Kopien von Objekten erstellt werden
  - Verweissemantik ist schnell bei komplexen Objekten, da Kopien entfallen
  - Verweisgröße in Java: 64 Bit
- Identitätsoperator ==, !=
  - Vergleicht nur die Verweise, nicht die Objekte, auf die verwiesen wird

  new Person() == new Person()

  ergibt false, da die Verweise verschieden sind, die Objekte gleichen sich aber

Objektmethoden

- Aufruf Objektmethode
  - Mit . nach Objektverweis
  - Objektverweis wird normalerweise durch eine Variable angegeben
  - „heiraten wird bei mueller aufgerufen“

```
mueller.heiraten(meier);
```

  - Verweis kann auch Ergebnis eines Ausdrucks sein

```
new Person().scheiden()
```

    - Verweis kann auch Rückgabewert einer Funktion sein
  - Sonderfall Stringliterale
    - “hallo“.equals(eingabe)
- Falls Objektverweis null ist, dann bricht das Programm mit einer NullPointerException ab

Objektmethoden

- Objektverweis vor dem Aufruf verhält sich wie ein impliziter Parameter mit Datentyp der Klasse
  - Zugriff auf Objektverweis in der Objektmethode mit Schlüsselwort **this**
  - Objektmethoden sind ähnlich zu Klassenmethoden, nur das ein zusätzlicher Parameter vor dem Aufruf steht
  - **this** kann nicht null sein
  - Objektvariablen sind auch ohne this sichtbar werden aber durch Parameter verdeckt

```
public void heiraten(Person ehegatte) {
    this.name = ehegatte.name;
}
```

Person meier = new Person();
Person mueller = new person();

meier.heiraten(mueller)

heiraten(meier, mueller)

```
public static void heiraten(Person this, Person ehegatte) {
    this.name = eheatte.name;
}
```

Objektmethoden

1. Beim heiraten soll beide Personen einen „Bindestrich“-Nachnamen bekommen
   - Nachname von this + „-“ + Nachname von ehegatte
   - Beide sollen ehegatten zueinander sein

Objektmethoden

1. Beim heiraten soll beide Personen einen „Bindestrich“-Nachnamen bekommen
   - Nachname von this + „-“ + Nachname von ehegatte
   - Beide sollen ehegatten zueinander sein

```
public class Person {

    public void heiraten(Person ehegatte) {
        this.name = this.name + "-" + ehegatte.name;
        ehegatte.name = this.name;
        this.ehegatte = ehegatte; // this wegen
                        // Sichtbarkeit nötwendig
        ehegatte.ehegatte = this;
    }

}
```

Objektmethoden

2. Eine Person (this) darf eine Person ehegatte nur heiraten, wenn

- Der Ehegatte existiert (nicht null ist) und nicht identisch zur Person ist
- Beide mindestens 18 Jahre alt sind und unverheiratet sind

Objektmethoden

2. Eine Person (this) darf eine Person ehegatte nur heiraten, wenn

- Der Ehegatte existiert (nicht null ist) und nicht identisch zur Person ist
- Beide mindestens 18 Jahre alt sind und unverheiratet sind

```
public class Person {
    public void heiraten(Person ehegatte) {
       if (ehegatte != null && this != ehegatte
            && this.ehegatte == null
            && ehegattte.ehegatte == null
            && this.alter >= 18
            && ehegatte >= 18) {
         this.name = this.name + "-" + ehegatte.name;
         ehegatte.name = this.name;
         this.ehegatte = ehegatte; // this wegen
                         // Sichtbarkeit notwendig
         ehegatte.ehegatte = this;
       }
    }
}
```

Objektmethoden

- Objektkonsistenz
  - Objekte sollen widerspruchsfrei sein
  - vor allem hinsichtlich der realen Objekte
- Vorangehendes Beispiel
  - Person darf sich nicht selbst heiraten oder zum Ehegatten haben
- Alle Objektmethoden sollen Konsistenz sicherstellen
  - Objektmethoden erst nach vollständiger Initialisierung eines Objekts aufrufen
  - Vor und nach dem Aufruf einer Objektmethode soll Objekt konsistent sein
- Problem
  - Objektvariablen im Beispiel sind public
  - Jederzeit Änderungen und Verstösse gegen Objektkonsistenz möglich
- Lösung Datenkapselung (*data encapsulation*)
  - Objektattribute immer private deklarieren
  - Nur in der Klasse direkt änderbar
  - Zugriff auf Objektattribute von aussen nur mit Objektmethoden
  - Objektkonsistenz muss in der Klasse sichergestellt werden

Objektmethoden

- Folgeproblem
  - Wie können dann Objekt mit Werten außerhalb der Klasse initialisiert werden?
- Lösung
  - Über Konstruktoren mit Parametern
  - Konstruktor wird analog einer Objektmethode deklariert
  - Kein Rückgabedatentyp
  - Konstruktor hat immer Namen der Klasse

```
public class Person {

   public Person(String name, int alter) {
     this.name = name;
     this.alter = alter;
   }

}
```

Objektmethoden

- Nächste Folgeproblem
  - Wie können in einer anderen Klasse auf Werte des Objekts lesend zugegriffen werden?
  - Z.B. für die Anzeige von Personenwerten in einer GUI
- Lösung
  - Zugriffsmethoden (accessor methods, getter, setter) implementieren zum Lesen (get) oder Setzen (set) von Objektattributen

```
public class Person {

    public int getAlter() {
      return this.alter;
    }

}
```

Objektmethoden

```
public class Person {
    // Objektkonsistenz in setter-methode sichertellen
    public void setAlter(int alter) {
      if (alter > 0) {
        this.alter = alter;
      }
    }
    public boolean isVerheiratet() {
      return this.ehegatte != null;
    }
}
```

- Namensschema auch verwenden, wenn einfache Werte berechnet werden
- Bei boolean is statt get verwenden

  public boolean isVerheiratet()

Objektmethoden

- In der Praxis
  - Bei Konsistenzverletzung eine Exception werfen
  - z.B. bei falschen Parameterwerte eine neue IllegalArgumentException mit **throw** IllegalArgumentException(“Alter negativ“);

```
public class Person {

    // Objektkonsistenz in setter-methode sichertellen
    public void setAlter(int alter) {
      if (alter <= 0) {
        throw new IllegalArguementException(“Alter
negativ oder Null“)“;
      }
      this.alter = alter;
    }

}
```

Objektmethoden

- Getter- und vor allem Setter nur implementieren, wenn nötig
- Teilweise Setter private deklarieren und nur in der Klasse verwenden, um Redundanz zu vermeiden
- Zugriffsmethoden sind Teil des Entwurfs nicht der Analyse

```
public Person(String name, int alter) {
  this.setName(name);
  this.setAlter(alter);
]

private int setAlter(int alter) {
  if (alter <= 0) {
   throw IllegalARguementException(“Alter negativ oder Null“)“;
  }
  this.alter = alter;
}
```

Objektmethoden

- Aufgabe: Scheiden implementieren
  - Wird bei einer Person aufgerufen
  - Beide verheirateten Personen sollen danach nicht miteinander verheiratet sein
  - Name bleibt bestehen
- Voraussetzung
  - Objekte sind konsistent
    - Eine Person ist entweder nicht verheiratet oder
    - Die beiden Personen sind miteinander verheiratet
  - Dreiecksverhältnisses oder ähnliches sind inkonsistent
    - Dürfen nicht auftreten
    - Brauchen nicht überprüft werden
    - Falls doch: assert verwenden (Info 2)

Konstruktoren

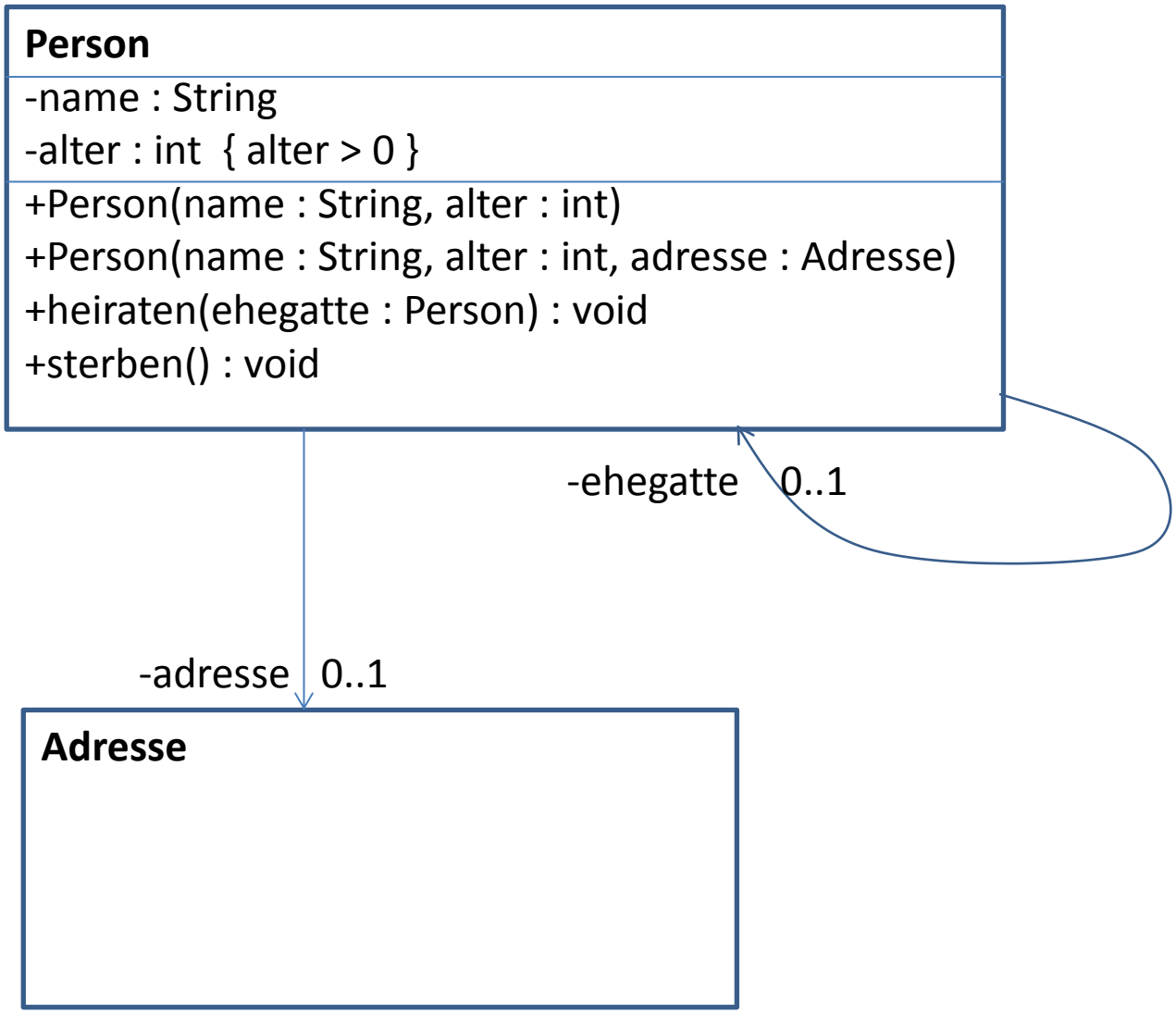
- Konstruktor dient der Objektinitialisierung
  - Keine aufwendigen Berechnungen im Konstruktor
- Mehrere Konstruktoren möglich
  - Konstruktoren müssen sich in Datentypen der deklarierten Parameter unterscheiden
  - Zur Vermeidung von Redundanz kann ein Konstruktor einen anderen aufrufen

    this(parameter);

    nur in der ersten Zeile eines Konstruktors möglich
- Konstruktor in UML
  - wie eine Methode angeben, aber ohne Rückgabedatentyp

Person
-name : String -alter : int { alter > 0 }
+Person(name : String, alter :int) +heiraten(ehegatte : Person) : void +sterben() : void

Konstruktoren



Konstruktor

```
public class Person {

  public Person(String name, int alter) {
     this(name, aller, null);
  }

  public Person(String name, int alter, Adresse adresse) {
     this.setName(name);
     this.setAlter(alter);
     this.setAdresse(adresse);
  }

}
```

Objektinitialisierung (vereinfacht)

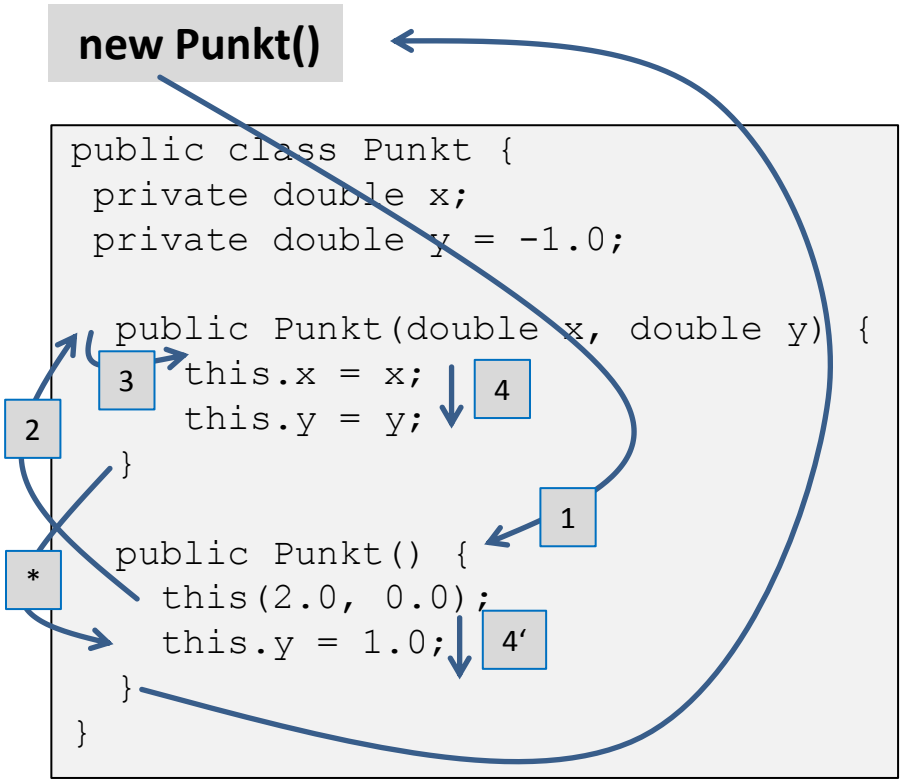
Initialisierung bei Aufruf eines Konstruktors

1: Objekt wird mit Standardwerten erzeugt.

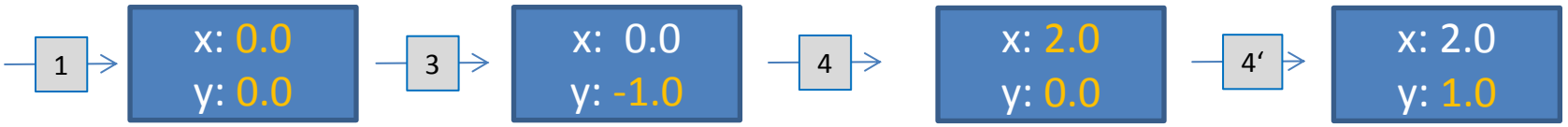
2: Ein optionaler Konstruktoraufruf mit this(…) in der ersten Zeile wird vollständig (*) ausgeführt.

3: Falls Konstruktor kein Aufruf mit this(…) enthält: Objektattribute werden von oben nach unten mit den Werten der optionalen Initialisierungen belegt.

4: Die Anweisungen des Konstruktors werden ausgeführt.



Zustandsänderungen des Objekts auf dem Heap:



Konstruktoren

- Sonderfälle, bei denen der Konstruktor private sein sollte
  - Klassen, von denen keine Objekte erzeugt werden sollen (vermeiden)
  - Klassen bei denen nur ein oder nur bestimmte Objekte erzeugt werden sollen
  - Objekt über öffentliche Klassenmethoden zurückgeben
- Z.B. bei einer Konfigurationsdatei
  - sie existiert nur einmal, deswegen sollte auch nur ein Objekt dazu gehören
- In Java
  - Klasse Toolkit kapselt Zugriff auf verschiedene Betriebssystem-Funktionalitäten

Konstruktor

- Ein-elementige Klasse (singleton)

```
public class Konfiguration {
  private static Konfiguration konfiguration
                              = new Konfiguration();
  private Konfiguration() {
     // Werte z.B. aus Datei einlesen
  }

  public static Konfiguration getKonfiguration() {
    return konfiguration;
  }
}
```

Javadoc

- Softwaredokumentation
  - Programme sind nicht selbsterklärend
  - Sie müssen für den menschlichen Leser ausreichend dokumentiert werden
  - Adressaten sind anderer Programmierer, die den Quelltext nicht geschrieben haben, aber verwenden wollen
  - Teilweise nur Bytecode ohne Quelltext vorhanden
- Javadoc
  - Softwaredokumentationswerkzeug, welches aus Java-Quelltexten HTML-Dateien erzeugt
  - Mindestens jede **öffentliche** Klassen, Attribute, Konstruktoren und Methoden muss immer dokumentiert sein

Javadoc

- Javadoc-Kommentar
  - Normaler Mehrzeilenkommentar mit zusätzlichem *****
    ```
    /**

     */
    ```
  - **Vor** jeder Klassendefinition, Deklaration, Konstruktor und Methoden
  - Text bis zum ersten . (Satzende) wird in Summary angezeigt
  - * in jeder Zeile am Anfang wird von Javadoc ignoriert
    ```
    /**
      * Nur dieser Test ohne * erscheint im Javadoc
      */
    ```
- Im Javadoc **kurz und genau** beschreiben
  - **Was** für Objekte die *Klasse* enthält
  - **Was** die *Methode* oder der *Konstruktor* genau macht
  - **Was** die Werte einer *Variable* bedeuten
- Im Javadoc nie beschreiben, *wie* etwas implementiert ist
  - Einzeilenkommentare // innerhalb des Rumpfs dazu verwenden

Javadoc (Klassenkommentar)

- Aus Sicht eines Objektes die **wichtigsten** Werte

```
/**
 Eine Person mit optionalem Ehegatten und optionaler
Adresse. Verheiratete Personen müssen volljährig sein.
*/
public class Person {

}
```

- oder **Verantwortung** der Klasse beschreiben

```
/**
  Person ist verantwortlich für die standesamtliche Heirat
und Scheidung zweier Personen.
 */
public class Person {

}
```

Javadoc (Methoden)

- Methoden vollständig dokumentieren
  - Mit Verb der Methode in Präsensform beginnen
  - Parameterwerte mit @param beschreiben

```
/**
 Heiratet die gegeben Person ehegatte, so dass danach beide
sich gegenseitig als Ehegatte haben. Beide Personen ändern
ihren Namen zu einem Namen, der aus dem Namen dieser Person
mit „-“ gefolgt dem Namen des ehegatte gebildet wird.
 Die Methode macht nichts, wenn diese Person oder der
ehegatte schon verheiratet, nicht volljährig sind oder der
übergebene ehegatte null ist.

@param ehegatte   der zu heiratende ehegatte, darf nicht
null sein
*/
public void heiraten(Person ehegatte) {
}
```

Javadoc (Methoden)

- Auch getter- und setter-Methoden dokumentieren

```
/**
 Gibt das positive Alter dieser Person zurück.
 */
public int getAlter() {
}

/**
  Setzt das Alter dieser Person auf den neuen Wert alter.
  Wenn das alter Null oder negativ ist, wird bricht die
Methode mit einer IllegalArgumentException ab.
  @param alter   muss größer 0 sein
 */
public void setAlter(int alter) {
}
```

Javadoc (Variablen)

- Für Klassen- und Objektvariablen
  - Beschreiben, was die Werte bedeuten
  - Wertebereich dokumentieren

```
/**
  Umfang der Erde um den Äquator herum in Metern.
 */
public static final double UMFANG_AEQUATOR = 40075.17;

/**
  Positive Bahngeschwindigkeit in Kilometern pro Stunden
   der Erde um die Sonne.
 */
private static double bahngeschwindigkeit = 29.78;
```

Javadoc (Methoden)

- Bei Funktionen am Javadoc-Ende das @return-Tag verwenden, wenn die Beschreibung umfangreich ist.
  - Beim @return-Tag kurz beschreiben, was der Rückgabewert bedeutet

  @return positive Alter dieser Person
- Bei getter- und setter-Methoden versuchen mehr Informationen anzugeben, als in der Methodendeklaration vorhanden ist
  - auch wenn dies nicht geht, immer einen Kommentar angeben, um Leser nicht im unklaren zu lassen
- Javadoc generell
  - Struktur der Javadoc führt teilweise zu redundanten Beschreibungen, z.B. Angabe von Attributewerten
  - mit @link auf Beschreibung des Wertebereichs verweisen, z.B. in getter- / setter-Methoden

```
/**
 Gibt das {@link Person#alter} dieser Person zurück.
 */
```

Javadoc (Methoden)

- Konstruktor wie Methode dokumentieren
  - Verb „erzeugt“ verwenden

```
/**
  Erzeugt eine neue Person mit dem gegebenen namen und
  alter. Die Person ist unverheiratet und hat keine
Adresse.
  @param name   ein beliebiger Name, darf nicht null sein
  @param alter  muss größer Null sein

  @throws IllegalArgumentException falls name null ist oder
alter nicht positiv
*/
public void Person(String name, int alter) {
}
```

Javadoc

- HTML mit Javadoc erzeugen
  - Warnungen weisen auf Inkonsistenzen hin, z.B. falscher Parametername
  - Nicht alle Warnungen sind Fehler

    z.B. @return Tag muss nicht notwendigerweise vorhanden sein, der voranstehende erklärende Satz aber schon
  - HTML ansehen, um Formatierungsfehler zu beseitigen

    z.B. falsch gesetzter Punkt

Testen

- **Tests dienen dazu Fehler zu finden**
  - Bisher manuell durch Eingabe von Werten und anschließende Überprüfung durch „Hinsehen“
  - Mit Testen kann nicht nachgewiesen werden, dass das Programm allgemeingültig funktioniert
- Voraussetzung für Testen ist eine genaue Beschreibung, was das Programm oder Programmteile machen sollen
  - Gegeben / Gesucht-Schema oder Beschreibung im Javadoc
- Modul (unit)
  - Funktional abgeschlossene, wiederverwendbare Programme
  - In objekt-orientierten Programmiersprache: eine Klasse
- Modultest (unit test)
  - Tests für ein Modul, um Fehler im Modul zu finden
- Regressionstests
  - Wiederholbare Tests, normalerweise automatisiert
  - Dienen dazu bei Programmänderungen schnell Fehler zu finden

Testen

- JUnit
  - Programmbibliothek, um automatisierte Regressionstests für Module zu programmieren
  - Nur Module ohne Benutzerinteraktion können damit getestet werden
- Für jede Klasse wird eine Testklasse erstellt
  - Für jede öffentliche Methode und Konstruktor wird mindestens eine Testmethode implementiert
- Ziel
  - Jede Anweisung der Klasse wird mit mindestens einem Test überprüft (100% Anweisungsüberdeckung)

Testen

- JUnit 3 nutzt Vererbung
  - Methodenname muss mit test beginnen

```
// import
public class PersonTest extends TestCase {
    public void testMethode1() {
        // Test von methode1
    }
}
```

- JUnit 4 nutzt Annotationen

```
// import
public class PersonTest {
    @Test
    public void testMethode1() {
      // Test von methode1
    }
}
```

Testen

- Aufbau eine Tests
  1. Testdaten erstellen
  2. Zu testende Methode aufrufen
  3. Ergebnisse mit assert-Methoden überprüfen

```
public void testHeiraten() {
1    Person meier = new Person("Meier", 27);
     Person mueller = new Person("Müller", 77);

2    meier.heiraten(mueller);

3    assertTrue(meier.isVerheiratet());
     assertTrue(mueller.isVerheiratet());
     assertEquals("Meier-Müller", meier.getName());
     assertEquals("Meier-Müller", mueller.getName());
     assertEquals(meier, mueller.getEhegatte());
     assertEquals(mueller, meier.getEhegatte());
}
```

Testen

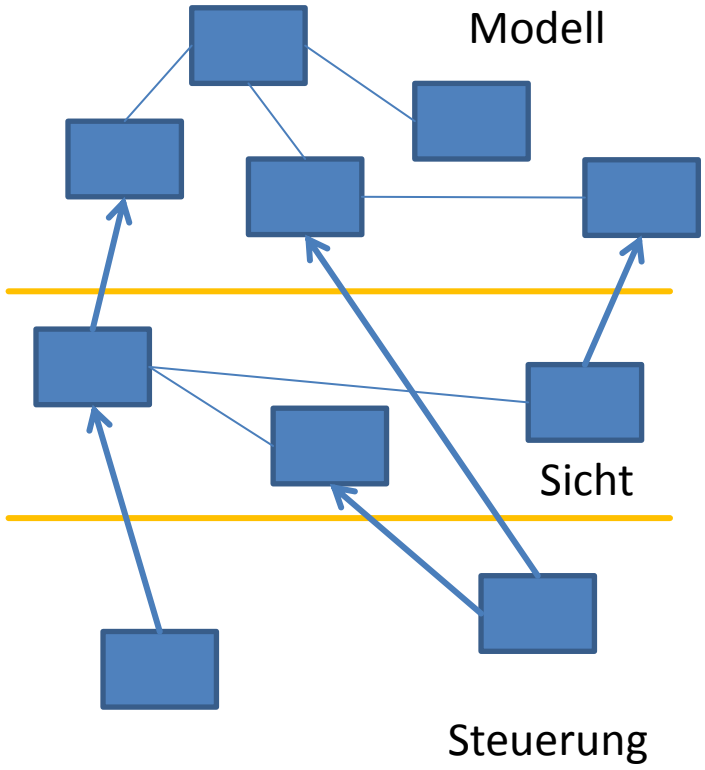
- Wizard in Eclipse (New->Junit ´Test case) verwenden
  - Zu testende Klasse anwählen
  - Richtige Version JUnit 3.0 / 4.0 im Wizard auswählen
- Tests ausführen
  - Run as -> Junit Test (Testklasse muss angewählt sein)
- assertTrue(BoolscherAusdruck);
  - BoolescherAusdruck sollte true sein, wenn die Methode korrekt ist
- assertFalse(BoolescherAusdruck);
  - BoolescherAusdruck sollte false sein, wenn die Methode korrekt ist
- assertEquals(erwarteterWert, berechneterWert);
  - Reihenfolge für Fehlertextausgabe wichtig
- assertEquals(2.564, x, 0.0000001);
  - 3. Parameter bei double / float gibt Schranke an, die zur Prüfung der Gleichheit verwendet wird

Testen

- Falls Tests fehlschlagen
  - Erst mögliche Fehler in der Testmethode suchen,
  - dann erst in der getesteten Methode
- Testmethoden sollten einfacher als die zu testende Methode sein
  - Wahrscheinlichkeit reduziert, dass Testmethode Fehler hat
- Viele kleine Testmethoden schreiben
  - Pro Testmethode nur einen einzelnen Test implementieren

Entwurf

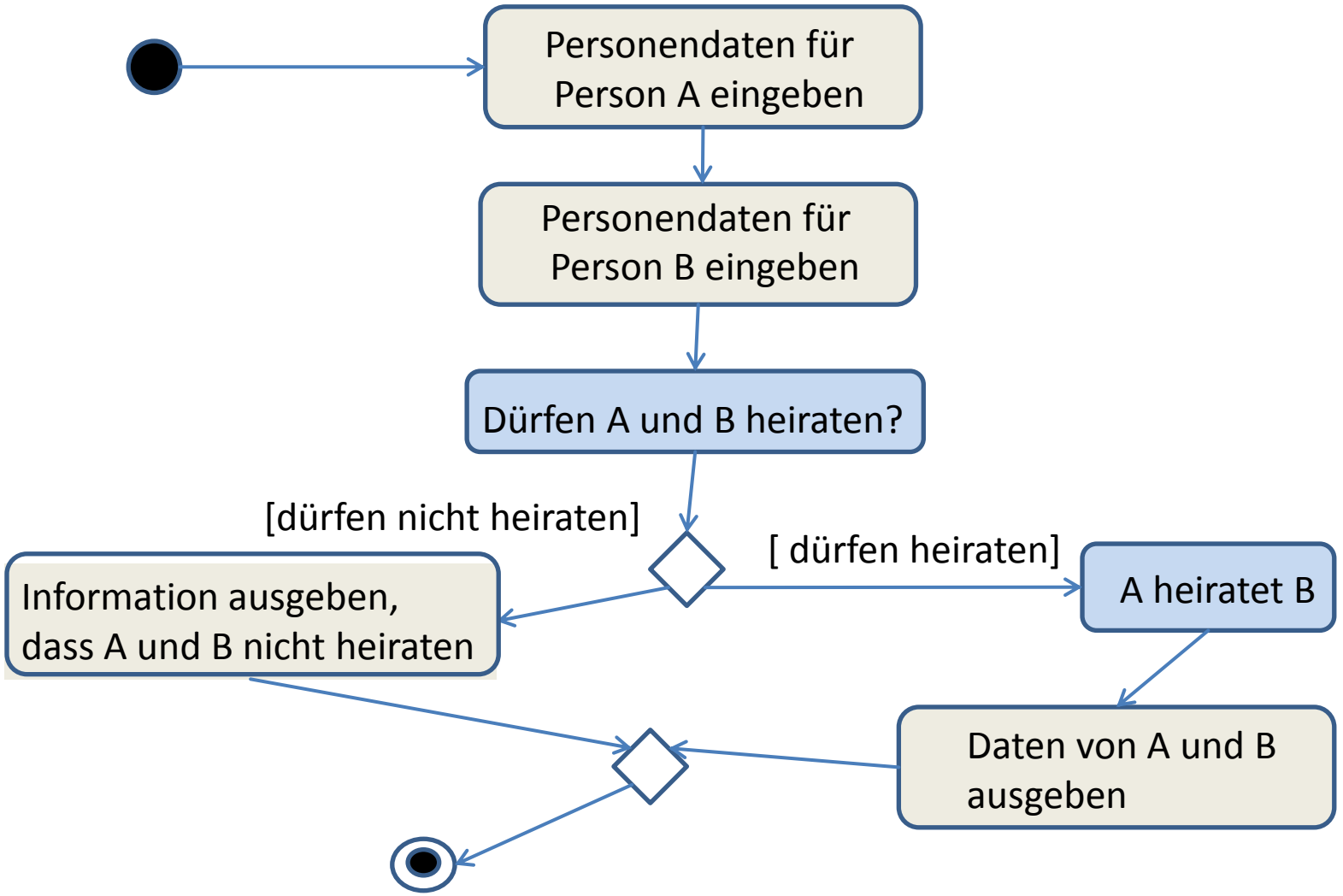
- Strukturierung von Software
  - Model-View-Control-Entwurfsprinzip (MVC)
- Jedes Programm hat Funktionalitäten für
  - Berechnungen, die der Simulation der realen Welt dienen (Modell)
  - Eingabe von Daten, z.B. vom Benutzer und Ausgabe von Daten, z.B. auf dem Bildschirm für den Benutzer (Sicht)
  - Abläufe, die alles zusammenfassen (Steuerung)
- Getrennte Module dazu entwickeln
  - Abhängigkeit von Control zu View zu Modell (aber nicht umgekehrt)
- Vorteil
  - Reduktion der Komplexität
  - Wiederverwendung von Module des Modells



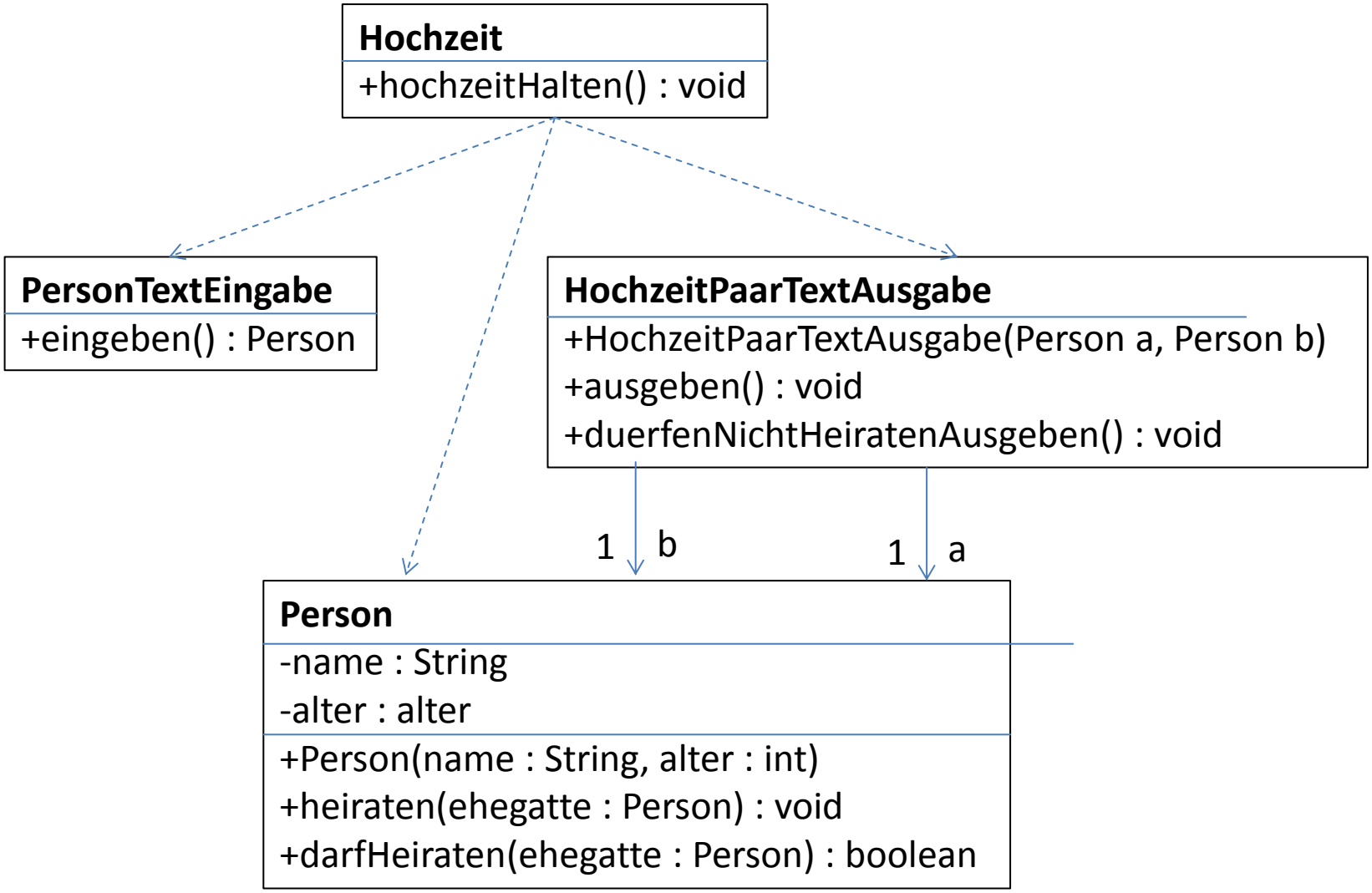
Entwurf

- Klasse Person gehört zum Modell
  - Enthält keine Ein- oder Ausgaben von Benutzerdaten
  - Kein Ablauf definiert
  - Lässt sich automatisiert testen
- Ablauf des Heiraten implementieren
  - Daten für zwei Personen eingeben (View)
  - Heiraten (Modell)
  - Information über die verheirateten Personen am Bildschirm ausgeben (View)
- Dieser Ablauf ist Teil der Steuerung

Entwurf


Personendaten für Person A eingeben
Personendaten für Person B eingeben
Dürfen A und B heiraten?
[dürfen nicht heiraten]
[dürfen heiraten]
Information ausgeben, dass A und B nicht heiraten
A heiratet B
Daten von A und B ausgeben

Entwurf


Hochzeit
+hochzeitHalten() : void
PersonTextEingabe
+eingeben() : Person
HochzeitPaarTextAusgabe
+HochzeitPaarTextAusgabe(Person a, Person b)
+ausgeben() : void
+duerfenNichtHeiratenAusgeben() : void
1
b
1
a
Person
-name : String
-alter : alter
+Person(name : String, alter : int)
+heiraten(ehegatte : Person) : void
+darfHeiraten(ehegatte : Person) : boolean

Entwurf

- Steuerung
  - darf selbst keine Ausgaben direkt oder indirekt enthalten
  - zur Verarbeitung Methoden des Modells aufrufen

```
public void hochzeitHalten() {
        PersonTextEingabe eingabe = new PersonTextEingabe();
        Person a = eingabe.eingeben();
        Person b = eingabe.eingeben();
        HochzeitPaarTextAusgabe paar
             = new HochzeitPaarTextAusgabe(a, b);
        if ( a.darfHeiraten(b)) {
            a.heiraten(b);
            paar.ausgeben();
        } else {
            paar.duerfenNichtHeiratenAusgeben();
        }
    }
```

Entwurf

- Sicht (Ausgabe)
  - Text und Ausgabeanweisungen sind alle hier enthalten
  - Leichte Änderung, z.B. auf Englisch möglich, nur Sicht muss geändert werden

```
public void ausgeben() {
       System.out.println(a.getName() + " und " + b.getName()
                        + " sind nur auf immer vereint!");
 }

 public void duerfenNichtHeiratenAusgeben() {
       System.out.println(a.getName() + " darf "
                  + b.getName() + " nicht heiraten");
       if (! a.isVolljaehrig() ) {
              System.out.println("Weil " + a.getName()
                   + " nicht volljährig ist!");
       }
 }
```

Entwurf

- Sicht (Eingabe)
  - Textausgaben könnten noch Ausgabeklasse verschoben werden
  - Bei Fehleingaben bricht Konstruktor ab, Fehler muss in der Steuerung behandelt werden (try-catch)

```
public Person eingeben() {
      Scanner scanner = new Scanner(System.in);
      String name = "";
      int alter = 0;

      System.out.print("Name: ");
      name = scanner.nextLine();
      System.out.print("Alter: ");
      alter = scanner.nextInt();

      return new Person(name, alter);
}
```

Entwurf

- MVC de-facto Standard im Programmentwurf
  - Teil des Entwurfs, nicht der Analyse
- Manchmal unklar, was zur Steuerung oder zur Sicht gehört (z.B. Kontrollanweisungen in der Sicht)
  - im Zweifel, Steuerung klein halten
  - Methoden der Sicht sollten nicht zu fein granular sein
- Eingabeüberprüfungen (z.B. alter > 0)
  - sollten als Methoden ins Modell, wenn es dem Verhalten realer Objekte nahe kommt, oder mit Exceptions in setter-Methoden Fehleingaben anzeigen
  - Manchmal werden Eingabeüberprüfungen auch in der Sicht vorgenommen
- Codierung und Interpretation von Werten gehört immer ins Modell
  - Text, die in der Sicht angezeigt werden gehören nicht ins Model
- Manche Anwendungen unterscheiden nur zwei Schichten
  - Präsentation (Steuerung und Sicht zusammen)
  - Modell