Java

- Übersetzung und Ausführungavac
  $ javac ProgrammName.java
  - javac = Compiler
  - Übersetzt in Java-bytecode (nicht maschinencode): ProgrammName.class
  $ java ProgrammName
  - java = Java Virtual Machine, Interpreter
  - der plattformunabhängige Bytecode wird ausgeführt
- Operatoren
  - arithmetisch: +, -, *, /, %
  - logisch: &&, ||, !
  - Bitweise: &, |, ^, ~, <<, >>
  - Inkrement, Dekrement: ++, --
- Verzweigungen
  - if(Bedingung){…}else{…}
  - switch(Ausdruck){
    case wert:
    …;
    break;
    default: …
    }
    - Ausdruck ist int, String oder Enum
- **Schleifen**
  - while(Bedingung){…}
  - do{…}while(Bedingung);
  - for(Initialisierung; Bedingung; Aktualisierungsanweisung){…}
  - Schleife beenden: break;
  - Schleifendurchlauf beendden und bei Bedingungsprüfung fortsetzen: continue;
  - **Erweiterte for-Schleife**
    for(Type curr : meineListe){ // Type ist der Typ der Elemente der Liste
    // mach was mit curr // in curr nacheinander die Elemente der Liste
    }
    - bei allen Klassen möglich, die das Interface **Iterable** implementieren und bei **Arrays**
    - Löschen oder Einfügen innerhalb der erweiterten for-Schleife führt zu ConcurrentModificationException
  - **Durchlauf mit Iterator** (entspricht erweiterter for-Schleife)
    Iterator<String> it = meineListe.iterator();
    while(it.hasNext()){
    String curr = it.next(); // holen des aktuellen Elements und weitersetzen des Iterators
    // tu was mit curr
    }
    - Iteratoren sind “Zeiger” auf ein in einer ction gespeichertes Element. Sie Enthalten außerdem die Information welches Element als nächstes dran ist
    - Vorteile:
      - Löschen ist in der Schleife möglich, dafür remove()-Methode des Iterators nutzen
      - bei einem ListIterator ist auch add() möglich. (Bei Set und Map nicht, da nicht an bestimmter Position eingefügt werden kann)

- **Collections**
  - **Arrays** (keine collection)
    - Deklaration: Datentyp arrayname[];
    - Erzeugung: arrayname = new Datentyp[laenge];
    - Deklaraion mit initialisierung: Datentyp arrayname[] = {wert1, wert2, …};
    - Zugriff: arrayname[index] = wert;
  - **List**
    - lineare Datenstruktur, Objekte werden also hintereinander in fester Reihenfolge gespeichert und sind durchnummeriert
    - l.add(newEl); // fügt am Ende der liste ein
    - l.add(index, newEl); // fügt an angegebenem Index ein
    - l.remove(el); // löscht das erste Vorkommen des Elements
      - für Integer dann z.b. l.remove(new Integer(3));
    - l.remove(index); // löscht das Element mit dem angegebenen Index
    - l.size(); // Anz der Elemente
    - l.get(index); // Das El mit angegebenen Index oder IndexOutOfBoundsException
    - l.set(index, newEl); // ersetzt das Element des angegebenen Index mit dem neuen Wert
    - **ArrayList**
      - wie Array, nur ohne Begrenzung
      - zugriff per index besonders schnell
      - Zufügen/Löschen von Elementen langsamer als bei LinkedList
    - **LinkedList**
      - Elemente in verketteter Liste gespeichert

        - —> jedes Element kennt Vorgänger und Nachfolger, Durlaufen Element für Element
            - direkter Zugrif auf bestimmtes Element ist langsam
        - gut für lineares Durchlaufen der Liste
    - Einfügen und Löschen auch in Mitte schnell
    - LinkedList<String> myList = new LinkedList<>();
    - oder mit Polymorphie: List<String> myList = new LinkedList<>();
        - Allerdings sind Methoden wie addFirst(…), die nicht im List interface stehen nur über Cast zu LinkedList aufrufbar.
- **Set**
    - Menge, die Reihenfolge der Objekte also beliebig, jedes Element kann nur einmal vorkommen
    - **HashSet**
        - Speicehert Elemente indem intern ein sog. Hashwert berechnet wird, mit der von Object geerbten Methode hashCode().
        - Einfügen und Löschen sehr schnell, wenn Verteilung der Hashwerte nicht ungünstig
            - Notfalls hashCode() überschreiben
    - **TreeSet**
        - Elemente in Baumstruktur gespeichert, dadurch automatisch sotiert
        - Vorraussetzung: die gespeicherten Objekte sind vergleichbar (implementieren Comparable<E>)
- **Map**
    - Assoziative Datenstruktur
        - Zu jedem Objekt wird ein indentifizierender Schlüssel gespeichert.
        - Zugriff nicht über eine Nummer, sondern über den Schlüssel (z.B.: Name, Personalnummer,…)
    - Einfügen
      myMap.put(key, value); /* fügt wert unter angegebenem Schlüssel ein, überschreibt Wert des bisherigen Schlüssels */
    - Löschen
      myMap.remove(key); // löscht wert mit angebgenem Schlüssel
    - myMap.size(); // Anz. Elemente
    - myMap.get(key);
    - myMap.containsKey(key); // testet ob der Schlüssel in der Map ist
    - myMap.containsValue(wert); // testet ob der Wert in der Map ist
    - Erweiterte forschleife für Map (Map selbst implementiert Iterable nicht, man muss also entweder durch keySet, values oder entrySet iterieren)
        - Auf Schlüsseln:
          for(String k: meineMap.keySet()){ /* Mach was mit k*/ }
        - Auf Werten
          for(BigDecimal v : myMap.values()){ /* Mach was mit v*/}
        - Auf Einträgen
          for(Map.Entry<String, BigDecimal> e : myMap.entrySet()){
            /* Mach was mit e.getKey() und e.getValue() */
          }
    - Durchlaufen mit Iterator
        - Iterator der Menge der Schlüssel (keySet()), der Werte (values()) oder der Kombination aus beidem (entrySet()) wandern

    - **HashMap**
        - für Schlüssel wird intern Hashwert berechnet, der die Speicherreihenfolge angibt
        - Deklaration:
          HashMap<String, BigDecimal> myMap; // 1. Typ -> Schlüssel, 2. Typ —> Wert
          oder: Map<String, BigDecimal> myMap; // Polymorphie!
        - Erzeugen:
          myMap = new HashMap<String, BigDecimal>(); /* erzeugt leere Map, es gibt Konstruktoren mit weiteren Parametern */
    - **TreeMap**
        - Schlüssel werden sortiert und in dieser Reihenfolge in einer Baumstruktur gespeichert
    - **Properties**
        - als Schlüssel nur Strings erlaubt
        - gedacht für Speicherung von Systemeigenschaften
- Threadsichere Collections (siehe Nebenläufigkeit unter Bild)

- Generics

- **Generische Klassen**
  - haben Typ-Parameter (einen oder mehrere)
    - der Datentyp einer Eigenschaft ist nicht bei Entwicklung der Klasse festgelegt, sondern wird erst beim Erzeugen eines Objektes definiert.
  - class Klassenname<T>
  - **Deklaration**: Klassenname<String> meineVariable;
  - **Erzeugen**: meineVariable = new Klassenname<String>();
    - oder: meineVariable = new Klassenname<>(); // compiler weiß, dass String nötig
  - nur Klassen für die Typen erlaubt. Anstelle von primitiven Datentypen müssen also die Wrapper (z.b. Integer) verwendet werden
    - Bezeichner wie pD, nur großgeschrieben (Ausnahmen: int -> Integer, char -> Character)
  - **Polymorphie**
    - Unterklasse der generischen Oberklasse kann verwendet werden
      Oberklasse<Nummer> variable = new Unterklasse<Nummer>();
    - **Typparameter** muss der **selbe** ein. Also weder Oberklasse noch Unterklasse möglich, Folgendes geht also nicht:
      ~~Oberklasse<Number> variable = new Oberklasse<Object>();~~
      ~~Oberklasse<Number> variable = new Oberklasse<Integer>();~~
  - Generische Klasse schreiben

```
/** …
* @param <T> Bedeutung des Typs
*/
public class Klasse<T> {
  /* Klassenmitglieder, T kann an (fast) jeder Stelle verwendet werden, wo ein Datentyp
  gebraucht wird */
}
```

  - **Bounds**
    - Ermöglichen zusätzliche Anforderungen an den Typen T zu stellen
      <T **extends** Klasse1 & Interfce1 & Interface2>
      - —> T muss von Klasse1 erben und die beiden Interfaces implementieren
      - Auch wenn nur Interfaces angegben heißt es extends
  - Durchgestrichenes geht nicht:

```
class Klasse<T> ~~extends Exception~~ // keine generischen Exceptions
{
  public void methode() {
    ~~new T();~~ // keine Konstruktoraufrufe (Ausweg: Reflektion…)
    if (~~this instanceof Klasse<Integer>~~) {…} /* kein instanceof mit Typparametern (Ausweg:
Reflektion */
    T[] array = ~~new T[5]~~ // Keine Arrays von Typparametern
  }
  public ~~static~~ T methode2() {…} /* keine statischen Methoden, da T Parameter der Instanz,
nicht der Klasse ist */
}
```

- **Methoden mit Typparametern**
  Zugriffsmodifizierer <T> Rückgabetyp methode(T parameter, …){…}
  - meist T auch als Typ eines der Parameter, aber nicht notwendig
  - Generische Methoden müssen nicht in generischer klasse sein und schränken auch nicht ein, dass der verwenderte typ dem der Klasse entspricht
  - aufruf:
    - Compiler “rät” anhand der übergebenen Parameter, welcher Typ T sein soll
    - oder: variable.<String>methode(“bla”);
- **Wildcarts**
  - Klasse**<?>** ist quasi die Oberklasse aller parametrisierten Ausprägungen von Klasse
  - Klasse<?> x = new Klasse<String>(); // x ist zuweisungskompatibel mit jedem konkreten Typ
  - Zugriffsmodifizierer Rückgabetyp (Klasse<?> param){…} // man kann jede Ausprägung von Klasse als Parameter übergeben

- Probleme:
  List<?> x = new LinkedList<Integer>();
  int zahl = (Integer) x.get(0); // Bei Cast übernimmt Programmierer die Verantwortung
  ~~x.set(0, 7);~~ // geht nicht, weil 7 nicht zwingend zum “echten” Datentyp von x passt
  x.set(0, null); // null passt zu jedem Datentyp

- **JavaDoc**

```
/**
* Bescreibung des Programs, ggf.
* über mehrere Zeilen mit HtML-Tags
*
* @author Name
* @param arg Parameterbescreibung
* @return Beschreibung der Rückgabe
* @throws Exceptionyp wenn …
* @throws …
*/
public class Programm{…}
```

  - Beschreiben was passiert, nicht wie
  - 1. Satz: Kurzbeschreibung, danach Ausführliche beschreibung

- Exceptionhandling

  - try {
    // Fehlerverursachende Anweisung
    } catch (Exception e) {
    // Fehlerbehebung
    }
  - public void methode() **throws** NameDerException {…}
    - Abbruch der Methode im Fehlerfall, aufrufende Programmstelle muss sich darum kümmern (Fehler weitergeben).
    - in throws tatsächlich auftretende Exceptions nennen.
    - **RunetimeExceptions** = Unchecked Exceptions => **ohne Ankündigung**
  - Fehler kann im Rahmen der **Aufgabe der methode** behoben werden: **try-catch**
  - Fehler kann **nicht** im Rahmen der **Aufgabe Methode** behoben werden:
    throws in Signatur + Dokumentation

- **Vererbung**
  - extends Superclass
  - in Konstruktor kann mit **super(…)** in **erster Zeile** der Konstrukter der Superklasse aufgerufen werden
  - **@Override** zum kennzeichnen von überschriebenen Methoden
  - erst extends, dann implements
    public class Klasse extends Ober implements Interfaces1, Interface2 {…}
  - **Polymorphie**
    - **Oberklasse** variable = new **Unterklasse**(…);
      - Unterklassen-Objekte sind zuweisungskompatibel zu Oberklassenvariablen
    - **variable.ueberschriebeneMethode**();
      - hier wird die **Methode des Objektes** aufgerufen (hier Unterklasse), der Typ der Variablen ist egal

- **Interaces**
  - kein Konstruktorauffruf möglich
  - Interface implementieren

```
public class meineKlasse implements Interfacename, Interfacename2 {
  @Override
  public void interfacemethode1() {…}

  @Override
  public int interfacemethode2(){…}
}
```

  - Verwendung: Interfacename kann überall als Datentyp verwendet werden:

    - für Parameter
    - für Rückgabewerte
    - für die Deklaration von Variablen und Eigenschaften
  - Verwendungszwecke
    - Programmieren eines Algorithmus, der größtenteils implementiert werden kann, aber in kleinen Details von aktuellen Objekten abhängt
    - Neue Klassen implementieren nur die Details und können dann den fertigen Algorithmus nutzen
    - …
  - Methodennamen für fehlende klassen inkl Doku, was diese genau tun sollen hinteregen
  - Interface definition (in Java 8):

```
public interface InterfaceName {
  Datentyp KONSTANTE = wert; // static und final
  void methode();

  default void defaultMethode() {
    // tu was, z.b. auch durch Aufruf von this.methode();
  }

  static void statischeMethode() {
    // tu was
  }
}
```

    - alle Mitglieder sind automatisch public
    - default-Methoden
      - können, müssen aber nicht überschrieben werden in implementierender Klasee
      - Mit dem Interface-Methoden arbeitende Algorithmen können direkt in das Interface geschrieben werde
      - ziel: Ordnung im Programm, Arbeitsersparnis
- **Abstracte Klassen**
  - Abstrakte Klassen und Methoden sind mit abstract gekennzeichnet
  - Objekterzeugung nicht möglich, der Konstrukter kann ausschließlich als Oberklassenkonstruktor mit super(…) aufgerufen werden
- **instanceof**

```
if (objekt instanceof KlasseOderInterface) {…}
```

  - prüft **zuweisungskompatibilität** zur angegebenen Klasse/Interface
  - jedes Objekt ist zuweisungskompatibel zu
    - seiner **eigenen Klasse** und **allen Oberklassen**
    - zu allen **implementierten Interfaces**
  - (**null instanceof** Klasse) liefert immer **false**

- **final**
  - bei Elgenschaften: Konstante, ernete Zuweisung nicht möglich
  - bei Mehthoden: In Unterklassen nicht überschreibbar
  - bei Klassen: Die Klasse ist nicht vererbbar
- static
  - statische Mitglieder nur einmal Pro klasse, daher aufruf von Klasse
- **Zugriffsmodifizierer**
  - private: Zugriff nur innerhalb der Klasse
  - default (packageweit): Zugriff im gleichen package
  - protected: Zugriff im gleichen package und in erbenden Klassen
  - public: Uneingeschränkter zugriff
- **Enum**
  - Enums sind Sammlungen von **statischen Konstanten**.
  - Die Konstanten haben allerdings **keinen primitiven Datentyp**, sondern sind **Objekte**, die **Eigenschaften** und **Methoden haben können**.
  - public enum Aufzählung

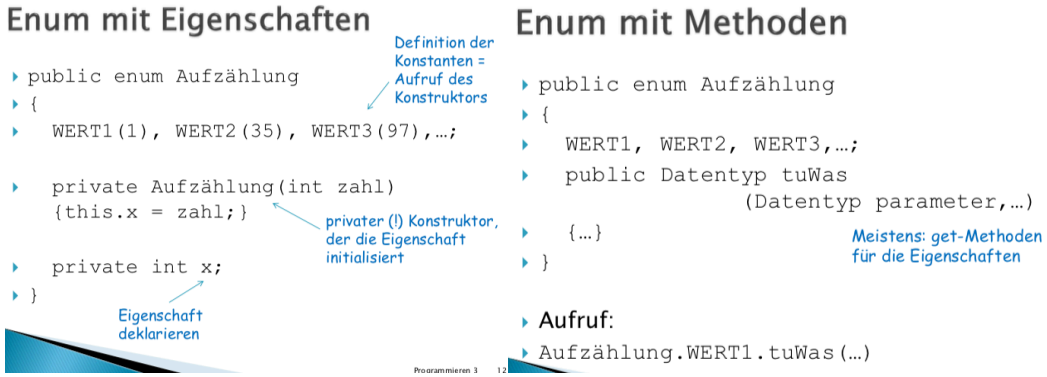
```
{
  WERT1, WERT2, WERT3, …;
}
```

    - eintspricht in etwa:

```
public class Aufzählung extends Enum {
  public static final Aufzählung WERT1 = new Aufzählung();
```

```
    public static final Aufzählung WERT2 = new Aufzählung();
    …
    private Aufzählung() {}
  }
```

- Zugriff:
  - Aufzählung variable = Aufzählung.WERT1;
  - Aufzählung variable = Aufzählung.valueOf(“WERT1”);
  - String text = variable.name(); // liefert “WERT1”
  - int nummer = variable.ordinal(); // jeder Konstante ist eine NUmmer zugeordnet
- Enum mit Eigenschaften oder Methoden



-
- Alle Konstanten einer Enum:
  for(int i=0; i < Aufzählung.values().length; i++){
    // Tu was mit Aufzählung.values()[i]
  }
- Vergleich
  if (variable == Aufzählung.WERT1) // equals() nicht notwendig

- **finalize-MEthode**

```
@Override
protected void finalize(){
  try{
  } catch(Throwable t){} // Exceptions ignorieren
  finally{
    super.finalize(); // Aufräumarbeiten der Oberklassen
  }
}
```

- wird von Garbage Collector aufgerufen, wenn er den Speicher eines Objekts freigibt

- Nebenläufigkeit

- der Hauptthread ist der Ausführungsstrang
- Neben dem **Ausführungsstrang** können weitere Threads gestartet werden
  - wenn nur **ein Prozessor** zur verfügung regelt die JVM die **abwechselnde** Bearbeitung der Aufgaben der Threads
- Ist der letzte Thread mit Abarbeitung seines Programms fertig ist das Programm beendet
  - Ausnahme: Hintergrundthreads (Damon-Threads)
  - Jeder Thread kann an jeder Stelle seines Codes unterbrochen werden, damit ein anderer arbeiten kann. #TODO widerspruch kritische Threads?
- **Threads erstellen**

```
public class MyRunnable implements Runnable {
  @Override
  public void run {…}
}
oder:
public class MyThread extends Thread { // Die Klasse Thread implementiert Runnable
  @Override
  public void run() {…}
}
```

  - run-Methode:

    - **weder Parameter noch Rückgabe**
        - daher alle “Eingaben” vorher z.b. in **Eigenschaften** des Runnable-Objektes speichern
        - Rückgabewerte können nach Ende des Threads aus seinenn Eigenschaften gelesen werden
- **Threads starten**

```
Runnable r = new MyRunnable();
// Eigenschaften von r setzen
Thread t = new Thread(r); /* ab hier wechselt JCM zw. den beiden Threads (Haupthread wird
 fortgesetzt, run()-Methdoe von r wird ausgeführt */
t.start();
// beliebiger code
(new Thread(()->{a();})).start(); // erzeugt Thread, der in der run methode a() aufruft und startet ihn
```

- **kritische Abschnitte**
    - Codezeilen, die nicht unterbrochen werden dürfen heißen kritische Abschnitte
- **Synchronisation**
    - löst probleme die durch gleichzeitiges Arbeiten auf den selben Daten entstehen, z.b. Lost Update (z.b. A holt ließt Eigenschaft von Objekt o, B ändert Eigenschaft von o, A verändert Eigenschaft von o basierend auf dem zuvor gelesenen)
    - Sperrobjekt oder Monitorcode, der ebenfalls das Sperrobjekt nennt darf nicht gleichzeitig mit dem von synchronized umgeben Abschnitt ausgeführt werden (anderer Code schon).

```
synchronized(sperrobjekt) {
  /* kritischer Abschnitt */
}
```

        - synchronisierte Blöcke sollten möglichst kurz sein und insb. nur eine nicht zu unterbrechende Aufgabe enthalten
        - Das Sperrobjekt sollte möglichst “klein” sein
            - ggf. kann in einer bel. Klasse ein öffentliches konstantes Objekt angelegt werden
    - Alternativ kann das **Stichwort synchronized** angegebenwerden für:
        - **Instanzmethoden**
          public synchronized void methode() {…} // **sperrt this**
        - **statische Methoden**
          public static synchronized void methode() {…} // sperrt das **Class-Objekt der Klasse**
    - Aktives Warten // Schlecht!

```
while(zustand nicht erreicht) {
  Thread.sleep(1000); // Kann auch der einzige Hauptthread sein
}
weiterarbeiten;
```

    - Thread.**yield()**
        - der aktuelle Thread gib Prozessor ab, ist sofort wierder arbeitsbereit
    - **threadVariable.join()**
        - wartet bis der angegebene Thread beendet ist
    - **Kommunikation**

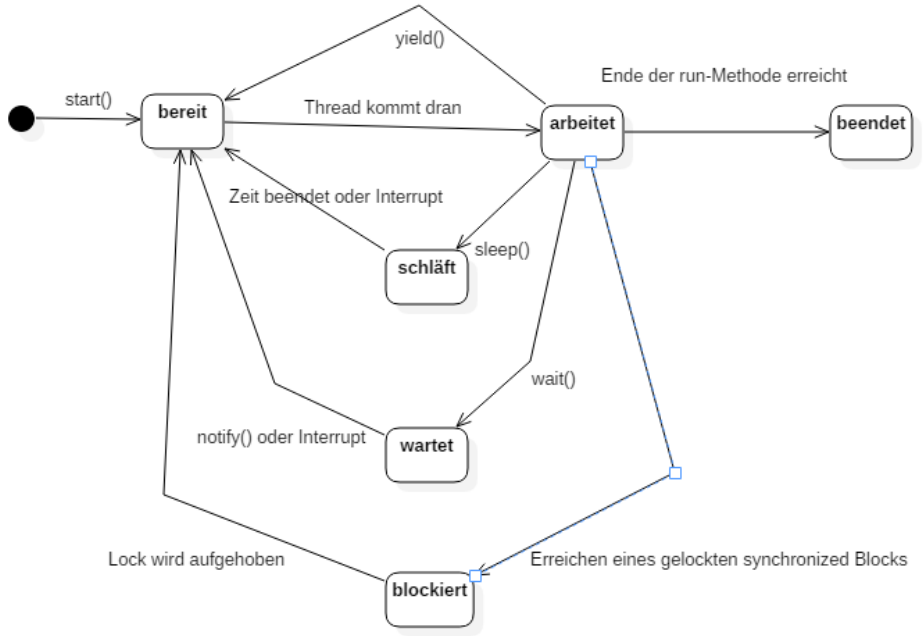
```
Thread 1:
  synchronized (object) {
    …
    objekt.wait(); // warten bis das objekt verändert wird
    ….
  }
Thread 2:
  synchronized (object) {
    …
    objekt.notify(); // Benachrichtigung über eine Veränderung in object
    ….
  }
```

        - das **object**, von dem **wait**(), **notify**() aufgerufen wird muss jeweils das **Sperrobjekt** der **synchonized** methode sein
        - wait und notify dürfen **nicht außerhalb** eines **synchronized**-Blocks aufgerufen werden
            - sonst IllegalMonitorStateException

- wait und notify sind in Klasse Object definiert
- Alternativen zu wait und notify:
  - void **wait(long anzahlMillisekunden)**
    - wartet höchstens die angegebene Zeitspanne
  - void **notifyAll()**
    - benachrichtigt **alle wartenden Threads** (notify() nur einen #TODO überprüfen)
- **wait**() hällt immer den **aktuell laufenden Thread** an. Man kann keinen Thread von außerhalb in den Wartezustand schicken.

- **Thread anhalten** (bzw. von außen Verhalten des Threads ändern)
  - sehr schlecht, da Thread keine Möglichkeit hat "Aufräumarbeiten" zu erledigen:
    - threadVariable.stop();
    - suspend() und resume()
  - Über Eingenschaft anhalten
    - Vorraussetzung: der Thread-Code muss "kooperieren", muss also immer wieder prüfen, ob die Ende-Bedingung vorliegt
    - runnableVariable.anhalen = true;
      … in run()-Methode:
      if(this.anhalten) {
        // aufräumen
        return;
      }
    - Problem: in langen Warte-/Schlafphasen bekommt der Thread die Aufforderung nciht mit
  - **Interrupt**
    threadVariable.interrupt();
    … in run()-Methode:
    try {
      if(Thread.interrupted()){
        // aufräumen
        return;
      }
      …
    } catch (InterruptedException e) {
      // aufräumen
      return;
    }
    - threadVariable.interrupt(); bewirkt:
      - Wenn der Thread in threadVaribale gerade **läuft** wird interene **Interrupt-Flag** gesetzt (die mit Thread.interrupted() abgefragt werden kann)
      - Wenn der Thread in threadVariable gerade **schläft**/**wartet** (Flag kann nicht gesetzt werden) wird **InterruptedException** ausgeläst
  - System.exit();

- Thread-Zustände:



  - **Threadsichere Collections**
    - von Klasse **Collections** über die folgenden **Statischen Methoden** eine **threadsichere Collection geben** lassen:
      - List<T> **synchronizedList** (List<T> liste)
      - Set<T> **synchronizedSet**(Set<T> menge)
      - Map<K, V> **synchronizedMap**(Map<K, V> map)
    - **ConcurrentHashMap<K, V>**
      - ebenfalls threadsicher und efffiienter

- Unschön/Problem: viele Klassen, viele Dateien, kein Zugrif auf Elemente der eigentlichen Ausführungsklasse
  - Lösung: Nicht öffentliche Klasse
    - Klasse innerhalb der gleichen Datei (möglich, wenn nicht public)
    - viele Klassen, weniger Dateien
    - Kein Zugriff auf Elemente der Oberklasse
  - Lösung: innere Klasse
    - (innerhalb der UmgebendenKlasse private classe InnerClass impements Runnable {…})
    - Erzeugung instanceOfUmgebendenKlasse.new InnerClass()
    - Voller Zugriff auf Elemente der Ausführungsklasse (auch private)
    - viele Klassen
    - Vollständiger Klassenname: UmgebendeKlasse.InnerClass
    - Statische Mitglieder der inneren Klasse verboten
    - Statische innere Klasse,, damit es nur eine innere Klasse für alle Objekte der UmbgebendenKlasse gibt. (Kein zugriff auf nicht-statische Eigenschaften)
    - this bezieht sich auf aktuelles Objekt der inneren Klasse, aktuelles Objekt der umgebenden Klasse: UmgebendeKlasse.this
  - Lösung: **anonyme Klasse**
    Runnable r = **new** Runnable()**{** /* hier kann ein **Interface** das **implementiert** (erbt dann von Object) oder eine **Klasse** von der **geerbt** wird stehen */
    public void run(){
    // Anweisungen
    }
    // ggf. weitere Eigenschaften und Methoden
    };
    - **Voller Zugriff** auf **Elemente** der **Ausführungsklasse** (sogar auf **finale** und **effektiv finale** Variablen der umgebenden Methode)
      - effektiv final: die variable wurde vor ihrer Verwendung noch nicht verändert
    - nicht viele Klassennamen

- this bezieht sich auf objekt der anonymen Klasse, aktuelles Objekt der umgebenden Klasse: UmgebendeKlasse.this

- **Lambda-Ausdrücke**
  - **Functionale Interfaces** (aus Standardbibliothek) (Funktionale Schnittstelle)
    - Ein Interface mit **genau einer abstrakten Methode** (nicht: abstrakte Klasse mit nur einer abstrakten Methode)
      - Daher: Wird das Interface implementiert ist klar welche Mehtode implementiert werden muss.
  - **Idee**: **Annonyme klasse**, die **ein Interface Funktionales Interface** implementiert. Da in Funktionalen Interfaces nur eine methode enthalten ist, kann der **Name** der **methode weggelassen** werden.
  - Lambda-Ausdrücle sind **nur erlaubt**, wenn der Compiler den **Interface-Typ erraten kann** (aus dem **Datentyp** der **Variabelen**, in dem der Ausdruck gespeichert wird; oder eines **Parameters**, für den er angegeben wird)
  - **Syntax**:
    ```
    public interface MeinFunktionalesInterface {
      public int meth(int x, String y);
    }

    MeinFunktionalesInterface var = (int x, String y) -> {…; return wert; };
    var.meth(3, "bla"); // um die Methode von dem erstellten Objekt aufzurufen
    ```
    - in () stehen die Parameter der methode mit ihrem Datentyp (entsprechend dem was sonst hinter methodenname steht)
    - es wird ein **Objekt** einer klasse, die das Interfaces **MeinFunktionalesInterface implementiert erstellt**.
    - **Funktionale Interfaces** in der Standardbibliothek
      - **Überprüfen** von **t**:
        ```
        public interface Predicate<T> {
          public boolean test(T t);
        }
        ```
      - **Aktion** für **t** ausführen:
        ```
        public interface Consumer<T> {
          void accept(T t);
        }
        ```
      - **Umwandeln** von **t** in einen **Wert** von **R**:
        ```
        public interface cion<T, R> {
          R apply(T t);
        }
        ```
      - **Zusammenfassen zweier Werte**:
        ```
        public interface BinaryOperator<T> {
          public T apply(T t1, T t2);
        }
        ```
  - Lambda-Ausdrücke **verkürzen**
    public interface MyFIface{ public int meth(int x);}
    - **Datentypen** der Parameter können **weggelassen** werden
      MyFIace var = (x) -> {…; return wert;};
    - Bei nur **einem einzigen Parameter** dürfen die **runden Klammern weggelassen** werden
      MyFIace var = x -> {…, return wert;};
    - Bei void-Methode und nur **einer einzigen Anweisung** dürfen die **geschweiften Klammern weggelassen** werden.
    - Besteht der Körper/Body **nur** aus {return **wert**;} kann es zu wert verkürzt werden (ohne return und ohne geschreibfte Klammern)
      MyFIface var = **x -> x+1;** // anstelle von x -> **{return x+1;}**
    - Der Name muss nicht mit der Interface-Definition übereinstimmen
      MyFIface var = bla -> bla + 1;
    - **Methodenreferenzen**

- (int x) -> {return Klasse.methode(x); };
  Klasse::methode
- (int x) -> {return etwas.methode(x); };
  etwas::methode
- (int x) -> {return new Klasse(x); };
  Klasse::new
- (Klasse o, int x) -> {return o.methode(x); }
  Klasse::methode
  - bsp: Function<Kunde, String> u = kunde-> kunde.getName();
    Function<Kunde, String> u = Kunde::getName; // mit Methodenreferenz

- **Zugriff auf Variablen**:
  - LA hat Zugriff auf **Eigenschaften** der **umgebenden Klasse** (auch **private**)
  - das **aktuelle Objekt** der **umgebenden Klasse** mit **this** (statt wie bei anonymer klasse mit UmgebendeKlasse.this)
  - LA hat lese-zugriff auf **finale** und **effektiv finale Variablen** und **Parameter** der **umgebenden Methode**
    - Der Wert wird nach der Initialisierung des LA nicht mehr geändert
    - der LA **kann Variablen/Parameter nicht verändern**

- Streams

- literatur
- Stream-Objekt **erzeugen**
  - Aus **Methode aus Collection**<E>
    - public Stream<E> **stream()**
      - z.b. myList.stream()
    - public Stream<E> **parralelStream()** // für paralele Weiterverarbeitung
  - Stream Objekt mit statischer Methode **Stream.of()** erzeugen
  - Stream-Objekt mit **Stream.Builder** erzeugen
- Stream-Operationen: Die an Stream-Methoden übergebenen Operationen müssen
  - **non-interfering** (nicht-eingreifent) sein, d.h. sie dürfen die zugurndeliegenede Datenquelle (die Collection) nicht ändern
  - **stateless** (zustandlos) sein, d.h. nicht abhängig von Variablen/Eigenschaften von außerhalb der Operation sein, die sich ggf. währen dder Operation verändern. (Übergebener Lambda-Ausdruck darf keine Variable der Umgebung verändern)
- Die Operationen werden vertikal ausgeführt, d.h. es werden immer erst alle Operationen auf ein Element des Streams angewendet (soweit notwendig), bevor das nächste Element bearbeitet wird.
- **intermediate Operations** in Stream<T> (geben Stream zurück, daraus können also Operationsketten aufgebaut werden)
  - **Objekte herausfiltern**
    - public Stream<T> filter(Predicate<T> p); // für Prädikat oft Lambda-Ausdruck
    - public Stream<T> distinct();
  - **Operation** auf **alle Objekte anwenden**
    - public Stream<R> map(Function<T, R> f); // für Funktion oft Lambda-Ausdruck
  - **sortieren**
    - public Stream<T> sorted();
    - public Stream<T> sorted(Comparator<T> c); // für Comparator oft Lambda-Ausdruck
  - **zusammenfügen**
    - public static Stream<T> concat(Stream<T> a, Stream<T> b);
- **terminal Operatios** in Stream<T> (geben keinen Stream zurück, bilden also das Ende einer Operationskette)
  - erst **durch terminale Operation** am Ende einer Kette werden **alle intermediate Operationen angewandt**.
    - lazy: nur die für das Ergebnis notwendigen Operationen
    - der zugrundeliegende Stream wird dadurch geschlossen und kann nicht weiter verwendet werden (sonst IllegalStateException)

- **Operation** auf alle **Objekte anwenden**
  - public void **forEach**(Consumer<T> c); // für Consumer oft Lambda-Ausdruck
- **Objekte** zu **einem Wert zusammenfassen**:
  - public T **reduce**(T beginn, BinaryOperator<T> b); // für BinaryOperator oft LA
    - bsp:
      String alle = list.stream().reduce(“”, (a,b) -> a + b.toString())); //BinaryOperator<String>
      - entspricht:
        String alle = “”;
        for(Object b: list) {
          alle = alle + b.toString();
        }
  - **collect**
    - eine Art zusammenfassung, wobei je nach Collector meist eine Collection entsteht
    - der verwendete Collector definiert den Rückgabetyp/was für eine Collectiom, mit welchem Start und welche Bedinung ein Element erfüllen muss um zugefügt zu werden.
    - bsp: List l = stream.collect(Collectors.toList());
    - **Collectoren**
      - **Collectors.toList**() : liefert einen Collector, der alle Elemente in eine List packt
      - **Collectors.toSet**() : … Set …
      - **Collectors.counting**() : liefert einen Collector, der alle Elemente zählt
      - …
- Test, auf ein gültiges Objekt bzw. nur gültige Objekte:
  - public boolean **anyMatch**(Predicate<T> p); // **Allquantor** // oft LA
  - public boolean **allMatch**(Predicate<T> p); // **Existenzquantor** // oft LA

- **Beispiele**
  - liste // ein List-Objekt
    .stream() //daraus Stream machen, der alle Elemente der liste enthällt
    .map(x -> x.toString()) /* Function-Methode map wird auf jedes Element des Streams angewendet. Ergebnis ist Ein Stream<String> */
    .forEach(System.out::println); /* Die Consumer-methode forEach wird auf jedes Element im Stream angewendet. Terminal Operation */

- Input/Output

- **I/O-Streams**
  - Ein Stream ist eine serielle Schnittstelle eines Programms nach außen
  - Daten werden in der gleichen Reihenfolge gelesen, wie sie geschrieben werden (Stream wie eine Pipe vorzustellen, in die von einer Seite durch Schreiben Daten reingeschoben werden und auf der anderen seite durch Lesen rauskommen)
  - **Daten** bleiben **im Stream bis** sie **gelesen** werden
  - Jede Leseaktion entfernt die gelesenen Daten aus dem Stream
  - **byteweise**: es werden einzelne Bytes gelesen/geschrieben (bsp. exe-Dateien)
    - InputStream, OutputStream
  - **zeichenweise**: mehrere Bytes werden zu Zeichen zusammengefasst (Unicode) (bsp. .txt-Dateien)
    - Reader (von InputStream mittels InputStreamReader), Writer (von OutputStream mittels OutputStreamWriter)

- Vorgehen

- Zuerst ein **Stream** benötigt, der die **Quelle** angibt, z.b.
  - FileReader, FileInputStream
  - CharArrayReader
  - StringReader
  - PipedReader, PipedInputStream
    - Quelle ist hierbei ein Stream (Reader bzw. InputStream)
  - AudioInputStream
  - ByteArrayInputStream
  - StringBufferInputStream
  - Teilweise stellen Klassen auch eine get…Stream()-Methode zur verfügung

- über Stream einen **Stream** mit den gewünschten Methoden “drüberstülpen” (entspricht in etwa “**Decorator**”-Muster)
  - BufferedReader : Lesen mit Puffer (mehr als ein Zeichen auf einmal)
  - LineNumberReader : Lesen mit Angabe der Zeilennummer
  - ObjectInputStream : Lesen vollständiger Objecte
  - DataInputStream : Lesen von Zahlen
  - CheckedInputStream : Lesen mit Checksumme
  - InflaterInputStream : Lesen komprimierter Daten
- entsprechend mit output

- **Bsp**:
  FileReader fr = new FileReader(“dateiname”); // Quelle angeben (Relativ zu Ort der ausführung, sprich bei intelliJ der Ordner in dem sich das Projekt befindet. Pfad zu ressourcen also “src/main/resources”
  BufferedReader br = new BufferedReader(fr); // “darüberstülpen”
  String zeile = br.readLine(); // Fähigkeit nutzen (Methoden aufrufen)
  br.close(); // schließt auf fr
- **try-with-Ressourcen** (try mit Angabe von Ressource(n) in Kopf, die am **Ende** des try-Blocks (auch bei Exeption oder return) **geschlossen** werden)
  try (FileReader var = new FileReader(“src/main/resources/bsp.txt”), Datentyp2 var2 = new Datentyp2(…)) {
  // Arbeit mit var, var2
  } catch (Excetption e)
  { /* Exceptionhandler */ } // catch-Teil kann wegfallen
  - die Datentypen der angegebenen Ressourcen (hier Datentyp, Datentyp2) müssen AutoCloseable implementieren

- Serialisierung

- Serialisierung ist die **Abbildung strukturierter Daten** auf eine **sequentielle Darstellungsform**
  - Objekte werden in eine **Form** gebracht, in der sie z.b. in eine **Datei geschrieben** oder über ein **Netzwerk versendet werden können**
  - sie können **deserialisiert** werden, also wieder als Objekte eingelesen werden. Z.b. auch in einem **anderen Programm** oder **nach** einem **Neustart** des **Programms**
- **Gespeichert** werden alle **Instanzeigenschaften** eines Objekts, d.h.
  - die **eigenen Eigenschaften** (in **jeder Sichtbarkeit**)
  - **geerbte Eigenschaften** (in jeder Sichtbarkeit), wenn die **Oberklasse** selbst **serialisierbar** ist
  - **keine statischen Eigenschaften**
- **Vorraussetzungen** an alle **Instanz**-**Eigenschaften** der Klasse (damit die Klasse selbst Serializable implementieren kann, also “serialisierbar” ist):
  - **primärer Datentyp** oder
  - **Typ** einer Klasse, die das Interface **Serializable** implementiert **oder Collection** oder **Array** von **Klassen**, die **Serializable** implementieren
  - Ist eine der Viraussetzungen nicht erfüllt und trotzdem Serializable implementiert kein Compilerfehler aber beim Versuch ein Objekt der klasse zu serialisieren wird NotSerializableException ausgelöst
- Standardserialisierung:
  OutputStream ziel = …; // z.b. Datei, Netzwerkverbindung
  ObjectOutputStream oos = new ObjectOutputStream(ziel);
  // Objekte speichern
  oos.writeObject(speicherObjekt1);
  oos.wrriteObject(speichertObjekt2);
  oos.flush(); // puffer leeren (schreiben auch wirklich ausführen)
  oos.close();
- **Deserialisieren**
  InputStream quelle = …; // z.b. Datei, Netzwerkverbindung
  ObjectInputStream ois = new ObjectInputstream(quelle);
  // Objekte lesen
  Object o1 = ois.readObject();
  Object o2 = ois.readObject();

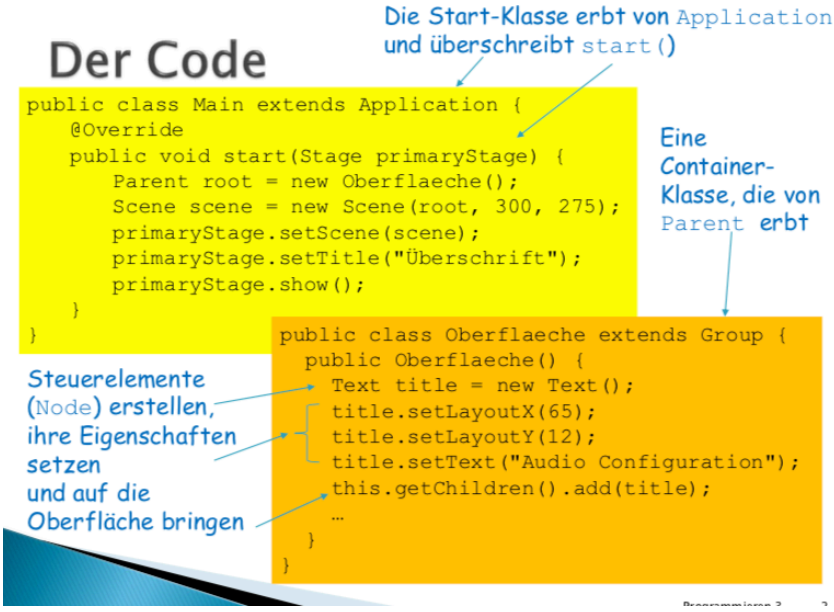
// Zur **weiterverarbeitung** ist üblicherweise ein **Cast** notwendig
ois.close();

- **Versionsnummer**
  - Wird eine Klasse **geändert** (z.b. neues Attribut), werden bereits **gespeicherte Objekte** dieser Klasse **ungültig** (Beim Einlesen tritt eine InvalidClassException auf
    - das **wegallen** von **Eigenschaften** ist **kein Problem** (wohl aber die Veränderung des Datentyps einer Eigenschaft)
  - das kann verhindert werden, indem man der Klasse eine **SUID** (**SerialUID**)
    private static final long serialVersionUID = …; // Zahl selbser ausdenken
    - diese ändert man bei Änderungen, die gespeicherte Objekte ungültig machen
- **transient** stichwort
  private transient Typ eigenschaft;
  - bewirkt, dass Eigenschaft **von** der **Serialisierung ausgenommen** wird (z.b. wegen nicht serialisierbarem Datentyp)
    - der Wert der eigenschaft wird nicht gespeichert, bei einlesen dann null oder 0
- **Vererbung**
  - Ist **Oberklasse serialisierbar**, so werden ihre Eigenschaften genauso gespeichert/ **eingelesen**
  - Ist Oberklasse **nicht serialisierbar**, werden ihre **Eigenschaften nicht gespeichert** und beim Einleseen durch **Aufruf** ihres **Standardkonstruktors initialisiert**
    - —> Keine Exception beim Speichern
    - —> eine **nicht-serialisierbare Oberklasse muss** einen **Standardkonstruktor** (nicht private) haben, sonst InvalidClassException beim einlesen
- **Serialisierung selbst beeinflussen**
  public vlass Speicherbar implements Serializable {
  private synchronized void **writeObject**(ObjectOutputStream s) throws IOException {…}
  private synchronized void **readObject** (ObjectInputStream s) throws IOEcception, ClassNotFoundException {…}
  }
  - sind die beiden methoden vorhanden, so ruft der Serialisierungsmechanismus sie auf
    - writeObject
      - sollte alle benötigten Eigenschaften von this (und ggf. Oberklassen) in den Parameterstrom s schreiben
    - readObject
      - sollte alle benötigten Eigenschaften aus s einlesen und allen anderen Eigenschaften sinnvolle Startwerte geben
    - … oder Exception werfen, wenn Serialisierung deieser Klasse doch nicht gewünscht ist

- **Tiefenkopipie** erstellen (Auch Inhalt von Listen klonen): (Mittels speicher inDatei und dann wieder auslesen)

```java
public class Bank implements Cloneable, Serializable{
// Alle Attribute erben ebensfalls von Serializable
    ...
    @Override
protected Bank clone() {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream out;
        try {
            out = new ObjectOutputStream(bout);
            out.writeObject(this);
            out.flush();
            out.close();
            byte[] thisAsByteArry = bout.toByteArray();
            ByteArrayInputStream in = new ByteArrayInputStream(thisAsByteArry);
            ObjectInputStream oin = new ObjectInputStream(in);
            Bank this_clone = (Bank) oin.readObject();
            in.close();
            return this_clone;
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            this.logger.log(e.toString());
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            this.logger.log(e.toString());
        }

        // clone not successfull
        return null;
}
```

- **Oberflächen** mit JavaFX
  - Die Start-Klasse erbt von Application und überschreibt start()
    - enthält entweder main oder wird anderweiteig aufgerufen
  - 

  - Dem Oberfläche-Konstruktor ggf. Model übergeben
  - Show bringt das ganze auf den bildschirm
  - ggf. main hinzufügen:
    public static void main(String[] args) {
      launch(args);
    }
  - Aufbau
    - **Stage**:
      - Das **Hauptfenster** der Anwendung
      - Einstellungen für Titelleiste, Größe,…, Reaktion auf Schließen der Anwendung
    - **Scene**:
      - **Anzeigefläche** des Fensters, hier sind alle Steuerelemente und sonstigen graphischen Elemente enthalten
      - Alle Elemente in Scene-Graph erben von Node
  - **Container**: BorderPane, GridPane, HBox, VBox, StackPane, Group, Pane, AnchorPane, TabPane, SplitPane, ScrollPane

- erben alle von **Parent**
- für **anordnung**.

- **Steuerelemente**
  - Erben von Node
  - haben Eigenschaft für Positionierung im umgebenden Container
  - haben alle Ereignishanler für Maus-, Drag-and-Drop-, Keyboard-, Zoom-, Rotations-, Touch-Ereignisse
- **Ereignisse**
  - Werden durch **Ereignishandler** behandelt
    - **Ereignishandler**: Eine Klasse, **implementiert** die in einem bestimmten **Interface** geforderte **Methode**. Meist **Lambda-Ausdruck**
  - bsp1: **schließenButton.setAction(e->controller.schliessen());**
  - bsp2: **stilChoiceBox.addEventHandler**(
    ActionEvent.ACTION, // Ereignis auf das man reagieren möchte
    e->controller.aendern(stilChoiceBox.getValue())); // Code der dann ausgefürt werden soll
  - allg: einElement.setOnAction
- **Binding** mit **Properties**
  - **Property**: **Wrapper** für **(primitive) Werte**, die man **beobachten** will. Sie **implementieren Property**, damit auch **ObservableValue** und damit auch **Observable**.
    - die in Property enthaltene methode **setValue(T) ruft für alle angemeldeten Listener** die **invalidated()-Methode** auf
    - Properties sind **bindungsfähig**
    - **Propertes anbieten:**
      private IntegerProperty meinWert = new SimpleIntegerProperty(); /* oder in Konstruktor initialisieren */
      public int getMeinWert(){ return meinWert**.get()**; }
      public void setMeinWert(int neu) { meinWert**.set(neu**) }
      public IntegerProperty **meinWertProperty**() { return meinWert; }
    - Es gibt **abstrakte Property-Klasse**n für primitive Datentypen, Collections, Strings und Objects. Davon Erben
      - **Simple…Property-Klassen**
      - **ReadOnly…Property**
        - Objekterzeugung durch ReadOnly…Wrapper
  - **Binding**
    - Wenn der beobachtbare Wert (ObervableValue) sich verändert, wird dadurch automatisch auch die zielProperty geändert
    - **zielProperty.bind(beobachtbarerWert);**
    - bsp:
      dbText.textProperty().bind(
      lsModel.lautstaerkeProperty().asString().concat(“ dB”)); / vom Model abgefragte eigenschaft verkettet zu einem Gesamtstring */
    - Bidirektionale Bindung (jede Änderung der einen Property bewirkt eine Änderung der anderen)
      eineProperty.bindBidirectional(andereProperty);
    -
- **FXML**

```
<?xml version=“1.0” encoding=“UTF-8”?> // XML-Deklaration
<?import javafx.scene.Group?>
<?import javafx.scene.text.Text?>
<?import uebungen.bankprojekt.verarbeitung.Girokonto?>
<?import uebungen.bankprojekt.verarbeitung.Kunde?>
<?import uebungen.Klasse?>
<?import uebungen.Klasse1?>

<Group xmlns="http://javafx.com/javafx/8.0.60" xmlns:fx=“http://javafx.com/fxml/1"
stylesheets="@style.css">

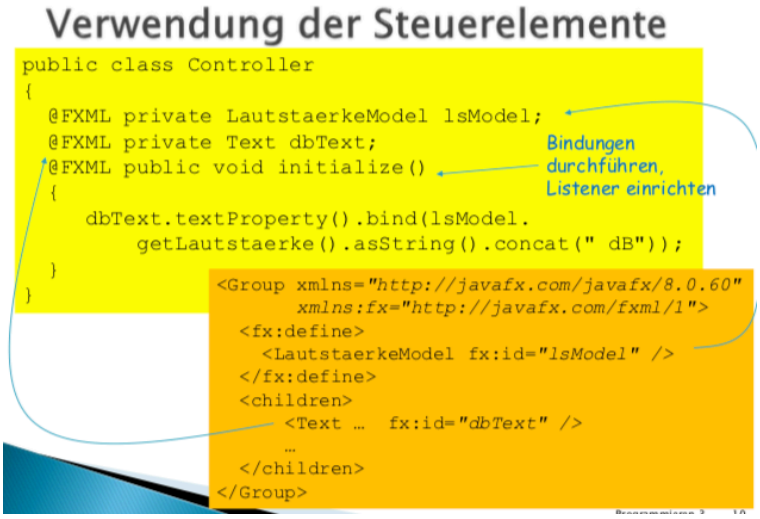
<fx:define>
```

```
  <Kunde fx:id="kundeModel" vorname="Janis" nachname="Schanbacher"
adresse="Limastr. 28”/> // Ruft Konstruktor auf und set methoden
  <Girokonto fx:id="kontoModel" inhaber="$kundeModel"/>
</fx:define>
<children>
  <Text layoutX="18.0" layoutY="90.0" text="Kontonummer:" styleClass="text-class"/>
  <Klasse attribut=“wert” />
  <Klasse>
    <attribut>wert</attribut>
  </Klasse>
  <Klasse>
    <rdonlyListAttribut>
      <Klasse1 />
      <Klasse1/>
    </rdonlyListAttribut>
  </Klasse>
</children>
</Group>
```

- Das Wurzelelement (hier Group) muss von Parent erben (siehe Container)
- Großgeschriebener Tag-Name entspricht Konstruktoraufruf
- Klassen müssen von Node erben ider in <fx:define> definiert sein
- Klassen müssen get-/set-Methoden gemäß Namenskonvention implementieren
- Attributwerrte/Inhalte sind primitiv oder String oder werden per valueOf(String) in den richtigen Typ umgewandelt
- Controller für FXML-Datei (sehr spezifisch für eine view, view nicht austauschbar)
  - Zugriff auf erzeugte Objekte (Steuerelemente) um Bindungen und Ereignishandler einzurichten
  - Initialisierung der Steuerelemente
  - Verbindung zum Model bzw. zu weiteren Objekten
  - **Controller** und **view verbinden**
    - entweder in FXML: <Wurzel xmlns=“…” xmlns:fx=“…” fx:controller=“ControllerName”>
    - oder beim laden (z.b. in Controller-Klassse, dann mit this)

```
public void start(Stage primaryStage) {
  FXMLLoader loader = new FXMLLoader(getClass().getResource(“datei.fxml”));
  loader.setController(controller) // bzw. loader.setController(this)
  Parent root = loader.load();

  Scene scene = new Scene(root, 300, 275);
  primaryStage.setScene(scene);
  primaryStage.show();
}
```

- 

- Objekte die in Controller mit **@FXML markiert** sind und in FXML datei die **gleichnamige fx:id="…"** tragen sind **verbuden**

- **Initialwert setzen und Binden**

FXML	im Java-Code
Wert einer Eigenschaft einmal als Initialwert verwenden:	

`<Klasse`
`attribut=`
`"$einName.eigenschaft" />`
	`Klasse var = new Klasse();`
	`var.setAttribut`
	`(einName.getEigenschaft());`

Eine Property binden:	

`<Klasse`
`attribut=`
`"${einName.eigenschaft}" />`
	`Klasse var = new Klasse();`
	`var.attributProperty.bind`
	`(einName.eigenschaftProperty());`

- **Ereignisbehandlung**
  <Klasse **onEreignis="#methode"** />
  - beim eintreten des Ereignisses wird **controller.methode()** aufgerufen
    - die Mehtode hat keinen Rückgabewert und keine Parameter oder als einzigen Parameter den zur Ereigniss passenden Unterklasse von Event
    - Alternativ in **initialize()-Methode** den **Listener** setzen, z.b.
      einzahlenButton.setOnAction(e ->{
      meldungText.setTect("Button gedrückt");
      });
      oder
      kontostandValueText.textProperty().addListener(e -> {
      if(kontoModel.getKontostandPositiv())
      kontostandValueText.setFill(Color.BLACK);
      else
      kontostandValueText.setFill(Color.RED);
      });
- **CSS**
  - **Selektoren**:
    - .styleKlasse
      - entweder vordefinierte oder selbstdefinierte mittels attribut styleClass="bla"
    - #id
    - .check-box:selected bzw. .klasse:pseudoklasse
      - elemente der Klasse im angegebenen Zustand
  - **Einbinden** der **CSS**-Datei
    - in FXML: <Wurzel stylesheeds="@datei.css" … </Wurzel>
      - @ : der folgende Text wird als Dateipfad relativ zur FXML-Datei interpretiert
    - im Code: Scne scene = …;
      scene.getStylesheets().add("datei.css"); // ggf mehrere Datein

Testen

- Prozess ein Program in der Absicht auszuführen Fehler zu finden
- Erfolgreicher Testfall, wenn unbekannter Fehler enddeckt
- Zeigt Anwesenheit von Fehlern auf, ist aber kein Nachweis für Fehlerfreiheit
- Unit-Tests
  - **Unit** = **Funktionales Einzelteil** eines Computerprogramms
    - in OOP meist eine Klasse
    - Es gibt klar definierte Schnittstelle (Bei Klasse die Methoden)
  - eine Unit wird (**möglichst**) **isoliert** getestet, d.h. ohne Zusammenarbeit mit anderen
    - weitere Units (z.b. Objekte anderer Klassen, die von denen der zu testenden verwendet werden) werden üblicherweise simuliert. Dafür z.b. **Mocking**
  - Unit-Tests sind Whitebox-Tests
    - der Tester kennt den zu testenden Code, üblicherweise testet der Programmierer die Unit selbst

- Oft auch im Rahmen der Testgetriebenen Entwicklung: erst Test schreiben, dann Code, der ihn zum Laufen bringt
- Trotzdem werden nur die nach außen sichtbaren Auswirkungen einer Methode überprüft
  - Rückgabewerte
  - Zustandsänderungen im Objekt (Eigenschaften)
  - Ausgaben
  - sonstige externe Auswirkungen
  - Gute Dokumentation ist Grundlage
- **JUnit**
  - Framework zum Durchführen von Unit-Tests
  - Testklasse sollte im gleichen Pagage liegen wie zu testende Klasse
    - Zugriff auf protected Mitglieder und mitglieder mit package-weiter Sichtbarkeit
    - Ggf. können private Mitglieder für Test package-weit sichtbar gemacht werden
  - Beispiel

```
import static org.junit.jupiter.api.Assertions.*; // assert-Methoden

@RunWith(JUnitPlatform.class)
public class KontoTest { // Nameskonvention: ZuTestendeKlasseTest
  @Test
  public void testEinzahlen100Euro() { // aussagekräftiger Name
    Konto k = new Girokonto(); // Testobjekt einrichten
    double vorher = k.getKontostand();
    k.einzahlen(100);
    assertEquals(vorher+100, k.getKontostand(), “Kontostand nach Einzahlung falsch!”);
  }
}
```

  - aufbau Testmethode:
    - (ggf. **SetUp**)
      - Initialisierung des Testobjekts (z.b. Konstruktoraufruf(e)
    - **Exercise**
      - Aufruf der zu testenden Methode(n) eines ganzen Ablaufs
      - Einfache get-Methoden müssen nicht getestet werden
    - **Verify**
      - Überprüfung der Ergebnisse der Methodenaufrufe (Kenntnis der erwarteten Ergebnisse vorrausgesetzt)
        - Was korrekt ist ist aus Dokumentation zu entnehmen
        - Rückgabewert korrekt?
        - Alle Eigenschaften des Objekts korrekt verändert bzw. unverändert gelassen?
        - Alle Parameter korrekt verändert bzw. unverändert gelassen?
        - allle externen Ressourcen benachrichtigt (Nebeneffekte)
        - Wird etwas getan, was nicht getan werden soll?
      - keine Ausgaben!
    - (ggf. **TearDown**)
      - Aufräumarbeiten (z.b. Löschen erstellter Dateien, Schließen von Datenbankverbindungen, …
  - **Testfälle** einer Methode
    - **normale Werte** im üblichen Bereich als Parameter
    - **Grenzfälle** (Parameterwerte, die an den Grenzen des Datentyps liegen (z.b. besonders groß)
    - **Sonderfälle** (null, 29.2.2000)
    - **falsche Werte** (z.b. negative Werte wenn logisch nur positive erlaubt)
    - **Methodenzusammenspiel**
      - “Happy Path” 0 normalfall, der übliche Ablauf
      - Spezielle Abläufe
      - fehlerhafte Abläufe
  - **Exceptions** testen
    - assertThrows in JUnit5
    - @Test
      public void exctest(){

```
    try {
      methMitException();
      fail("Meldung");
    } catch (ErwExceptiontyp e) {
    }
  }
```

- **Mocking**
  - Mockito als Ergänzung zu JUnit
  - Klassen unabhängig voneinander testen durch Einsatz von Mock-Objekten
  - Mock-Objekte sind **Platzhalter** für echte Objekte (mock=vortäuschen)
    - gleiche Schnittstellen
    - liefern vordefiniert Testwerte bei Methodenaufrufen
    - protokollieren was mit ihnen passert (welche Methodenaufrufe wie oft)
  - Mock-Objekte **erzeugen**
    - mittels methode public static <T> T mock(Class<T> classToMock)
      Klasse mockObjekt = Mockito.mock(Klasse.class);
    - erzeugt Objekt einer anonymen Klasse, die von Klasse erbt. daher gehen folgende Simulierungen mit Mockito **nicht**:
      - final Klassen (inkl. Enums)
      - anonyme Klassen
      - primitive Typen
  - Mock-Objekte **einrichten**
    - Mockito.when(mockObjekt.methode1(…)).thenReturn(wert);
    - Mockito.when(mockObjekt.methode(…)).thenReturn(wert1, wert2, wert3);
      - Für mehrfache Aufrufe einer Methode mit den selben params versch. werte zurüpckgeben
    - nicht eingerichtete Methodenaufrufe liefern 0, false, null oder leere Collection
  - Methodenaufrufe **verifizieren**
    - neben Methoden Testen auch Nebenefekte kontrollieren
      - Nur richtige Methoden mit richtigen Parametern in richtiger Anzahl aufgerufen durch die zu testende klasse?
    - public static <T> T verify(T mock, VerificationMode mode)
      - Mockito.verify(mockObjekt).methode1(…);
        - prfüft ob die angegebene Methode genau einmal aufgerufen wurde
      - Unter angabe eines VerificationMode ist folgendes möglich
        Mockito.verify(mockObjekt, VerificationMode).methode1(…);
        - Mockito.times(x) // x mal aufgerufen?
        - Mockito.atLeast(x)
        - Mockito.atMost(x)
        - Mockito.only() // nur diese eine Methode aufgerufen, keine andere
      - Mockito.verifyZeroInteractions(mockObjekt) // keine Methoden aufgerufen von mockObjekt
      - Mockito.verifyNoMoreInteractions(mockObjekt) // keine Methoden außer den vorher verifizierten mehr aufgerufen
  - **Exceptions testen**
    - Mockito.when(mockObjekt.methode2(…)).thenThrow(new MeineException());
    - oder: Mockito.doThrow(new MeineException()).when(mockObjekt).methode3(…);
  - **ArgumentMatchers**
    - Wenn das Argument zu dem Matcher passt, dann weiter
    - Mockito.when(mockObjekt.methode1(ArgumentMatchers.anyInt())).thenReturn(wert);
    - Mockito.verify(mockObjekt).methode1(ArgumentMatchers.anyInt());
    - Primitive Typen: anyBoolean(), anyByte(), anyChar(),…
    - Collections: anyCollection(), anyList(), anyMap(),…
    - Strings: anyString(), contains("teilstring"), endsWith("ende"), startsWith("start"), matches("Reg. Ausdruck")
    - Objekte: anyObject(), isNotNull(), isNull(), isA(EineKlasse.class)
    - Gleichheit: eq(wert)
    - Selbst definierte Tests: booleanThat(m), byteThat(m), charThat(m),… argThat(m)
      - m ist ein ArgumentMatcher:

        - Interface mit der Methode boolean matches(Object)
        - Testet das übergebene Objekt auf Gültigkeit
    - **Reihenfolge abtesten**
        - InOrder order = Mockito.inOrder(mockObjekt1, mockObjekt2,…); /* Mocks für die die Reihenfolge der Methodenaufrufe wichtig ist*/
          order.verify(mockObjekt1).methode1();
          order.verify(mockObjekt2).methode2();
          …
            - In dieser Reihenfolge müssen die Methoden aufgerufen werden
        - **Spys**
            - Ersetzen nur Teil des objekts
            - Klasse spyObjekt = Mockito.spy(realesObjekt);
            - **Mockito.doReturn(wert).when(spyObjekt).methode(…);**
        - Mockito.when(mockObjekt.methode1(…)).thenAnswer((InvocationOnMock invocation) -> {…; return einObjekt;});
            - führt beim aufruf den mehr oder weniger komplexn code aus
- **Integrationstests**
    - Annahme, dass jede Unit für sich fehlerfrei, hier wird deren Zusammenarbeit getestet
- #TODO Mocking

Tools

- StarUML
- **Maven**
    - **Abhängigkeiten** einrichten
    - ausführbare jar-Archive erzeugen
    - **Build-Management-Tool**. Untestüzt ein Softwareprojekt durch automatisierung des **Build-Ablaufs (Lebenszyklus** beim Anlegen, Kompilieren, Testen, Packen, Verteilen
    - Konvention vor Konfiguration (Convenience over configuration)
    - Fachbegriffe
        - Artefakt: Das Ziel-Produkt, meist ein jar- oder war-Archiv (Die Zieldatei eines Build-Vorgangs)
        - POM (Project Object Model): Konfigurationsdatei eines Maven-Prektes, basiert auf Super-POM
        - Maven-Plugin: Ein Programm, das einen Teil des Build-Prozesses durchführt oder um zusätzliche Aktionen erweitert
        - Goal: Kommando an ein Plugin.
            - Oft Name einer Phase des Lebenszyklus (Build-Vorgangs)
        - Dependency: Im Programm benötigte zusätzliche Ressourcen (Bibliotheken)
    - **Maven-Projekt anlegen**
        - File->New->Project , dann links auf Maven und weiter
        - **Group ID**: Hersteller-Bezeichnung. Überlicherweise umgedrehter Firmen-Domainname, z.b. com.janisschanbacher.projekt
        - **Artifact ID**: Bezeichnung für das Produkt: z.b. projekt
        - **Version**: Versionsnumer. -SNPSHOT sagt, dass diese Versionsnumer noch in der entwicklung ist. z.b. 1.0.0-SNAPSHOT
        - Group ID, Artifact ID und Version (**GAV**) müssen gemeinsam **eindeutig identifizieren**, da sonst im **öffentlichen** Repository nicht **findbar**
    - **Standardmäßige Projektstruktur**
        - **src/main/java: Quelltextdateien**
        - **src/main/resources: weitere benötigte Dateien**
        - **src/test/java: Testklassen für Unit-Tests**
        - **target**: alle **erzeugten Dateien**, insbesondere das **Ziel-jar-Archiv** (Ziel=Artifact)
        - **target/classes**: **übersetzte** .**class**-**Dateien**
        - **pom.xml: Konfigurationsdatei**
    - **Standard-Lebenszyklus** (Build-Vorgang) / pom.xml Konfiguration
        - erweiterbar und/oder konfigurierbar durch Maven-Plugins und Einstellungen in der pom.xml (die meisten innerhalb des <project> tags)

- Jede einzelne Phase des Build-Vorgangs wird durch ein Maven-Plugin ausgeführt
- **archetype**:
  - Es wird eine **Projektstruktur vorgegeben** (Ordnerstruktur, zusätzliche jar-Bibliotheken)
  - hier werden **abhängigkeiten** geladen und **Pfade** gesetzt. in konfiguration werden genauer Name und Versionsnummer benötigt:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.0.3</version>
  </dependency>

  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.0.3</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.23.4</version>
  </dependency>

</dependencies>
```

- **validate**:
  - **Projektstruktur** wird **überprüft**
- **compile**:
  - **Quelltext** wird **übersetzt**.
  - von maven-compiler-plugin durchgeführt und wie folgt anpassbar

```
<properties>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

- **test**:
  - **Testcode** wird **ausgeführt**
- **package**:
  - Der **übersetzte Code** und ggf. **Zusatzressourcen** werden **verpackt**, z.B. in ein **jar-Archiv**
- **integration-test**:
  - Das Paket wird in eine **Testumgebung geladen** und **ausgeführt**
- **verify**:
  - **Überprüfung** weiterer **Qualitätskriterien**
- **install**:
  - **Paket** ins **lokale Maven-Repository verschieben**
- **deploy**:
  - **Paket** ins **öffentliche Maven-Repository** verschieben und damit **allen zugänglich machen**

- in **intelliJ** rechts auf Maven-tab gehen, dann
  - Alle Dependencies neu importieren_
    - Reimport all Maven Projects (refresh symbol oben links)
  - Maven-Projekt builden
    - unter Lifecycle auf den Buildabschnitt klicken ab dem neu gebaut werden soll
      - z.b. compile, test, package, install…
- **Ausführbare Datei erzeugen**
  - in build tag das **maven-assembly-plugin einbinden**, als deskriptor jar-with-dependencies verwenden, als goal single eingeben (dadurch wird u.a. bei build die datei erzeugt), main klasse angeben:

```
...
<build>
```

```
<plugins>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.1.1</version>
    <configuration>
      <descriptorRefs>
        <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
      <archive>
        <manifest>
          <mainClass>playground.Test</mainClass>
        </manifest>
      </archive>
    </configuration>
    <executions>
      <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals>
          <goal>single</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
</project>
```

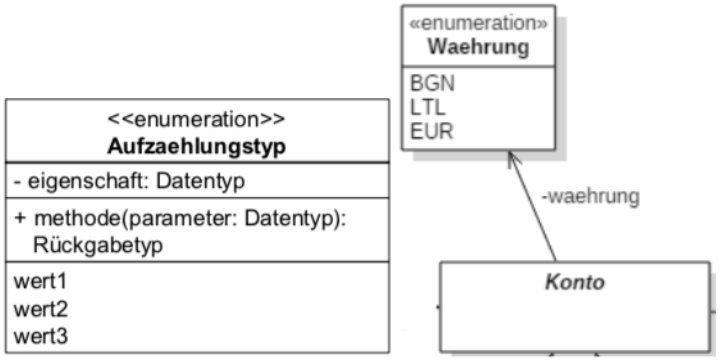
- ausführen: java -jar <jarfilename>.jar

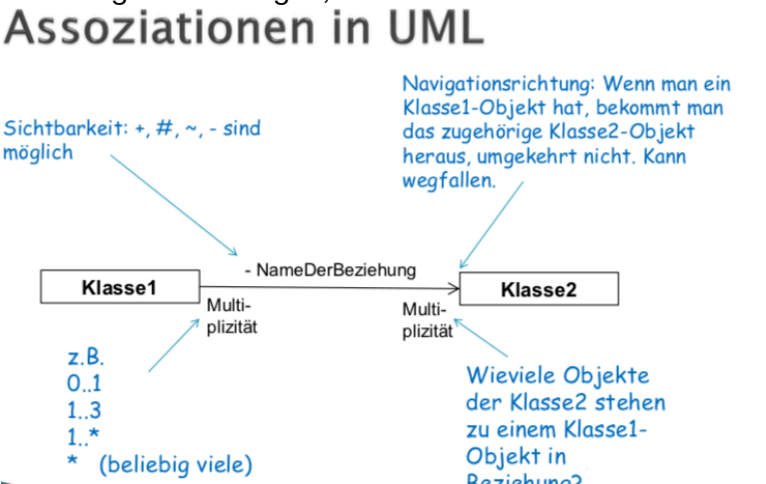
- **Git**
  - **anmelden**:
    git config --global user.name ‘Name’
    git config --global user.email ‘Email’
    - --global, da die angaben sonst nur für das aktuelle Repository gelten
  - Repository: Ordner, den git verwaltet
    - es wird in lokaler Kopie des remote Repositories gearbeitet
      - git clone “PfadZumServerrepository” zielordner
  - Commit: gespeicherter Stand der Dateien im Repository
    - Sinnvollerweise geschkissene lauffähige Version eines Projektes
  - Branch: Entwicklungszweiig
    - hHauptzweig ist der master
    - dier Oberste im Branch enthaltene Commit heißt HEAD
  - **Projekt auf Server hochladen**
    - git init // neues git-repository anlegen
    - git remote add origin https://github.com/yourusername/your-repo-name.git
    - git push -u origin master
      - änderungen hochladen, die natrülich zuerst geadded und committed sein müssen
  - **im Reepository arbeiten**
    - 0. aktuellen Stand vom Server holen
      git pull origin master
    - 1. Index updaten (add)
      git add dateiname // für jede neue/geänderte Datei
      git rm dateiname // für jede gelöschte datei
    - 2. Commit ausführen (Commit)
      git commit -m “Beschreibung des Commits.”
    - 2a. Commit zurücknehmen (Aktuellen Entwicklungsstand auf den eines früheren Commits zurücksetzen)
      git reset -hard commitID
      - mit git log erählt man Liste aller bisherigen Commits mit ID
    - 3. Zusammmenführung verschiedener Entwicklungsstände (merge)
      git fetch // holt de neuen Dateien vom Server in neuem Branch namens origin/master
      git merge origin/master // versucht origin/master mit dem lokalen master zu kombinieren und kennzeichnet ggf Konflikte

git status // zeigt Konfliktdateien an
Konfliktdateien bearbeiten, speichern, testen, zum Index hinzufügen und nochmal commiten
- 4. Daten zum Server pushen
git push origin

UML (Unified Modeling Language)
- Statische Strukturdiagramme
- Graphische Sprache, Programmiersprachenunabhängig
- Verschiedene Diagramme für unterschiednliche Akspekte eines Programms
  - Struktur-Diagramme: Klassendiagramm, Objektdiagramm, Komponentendiagramm, Paket-Diagramm, Kompositionsstrukturdiagram, Verteilungsdiagramm
  - Verhaltensdiagramme: Anwendungsfalldiagramm, Aktivitätsdiagramm, Zustandsdiagramm, Interaktionsübersichtsdiagramm, Sequenzdiagramm, Komunikationsdiagrmam, Zeitverlaufsdiagramm
- Im folgenden Klassendiagramme
- **Klassen**:
  - **1. Abschnitt:** (Fett und zentriert)
    - [Schlüsselwörter und Stereotypen] **Klassenname** [Eingeschaftsliste]
    - Schlüsselwörter und Stereotypen
      - <<auxiliary>> : Hilfsklasse
      - <<utility>> : hilfsklasse, ausschließlich statische Methoden/Eigenschaften
      - <<focus>> : setzt ein zu realisierendes Konzept im Wesentlichen um
    - Eigenschaften
      - abstrakte Klasse: kursiv oder {abstract}
  - **2. Abschnitt: Attribute/Eigenschafen**
    - [Sichtbarkeit] [/] Bezeichner: Datentyp [Multiplizität] [= Initialwert] [{Einschränkung1, Einschr2,…}]
    - bsp: -kontostand: double
    - bsp: -kontostaand: double = 0
      - die variable kontostand ist entweder bei definition oder im Konstruktor = 0 gesetzt.
    - Einschränkungen:
      - {readonly} : Attribut darf nicht vertändert werden (final)
      - {unique} : Jedes element muss einzigartig sein // z.b. Bei array
      - {ordered}
      - {eigene Enschränkung z.b. in natürlicher Sprache}
    - Sichtbarkeit Bezeichner: Datentyp Mulitiplizität = Anfangswert {Einschränkungen}
      - Standard: [1..1] = [1]
      - beliebig viele: [*] = [0..*]
      - optionalität: [0..1]
      - allgemein: [m..n]
      - -kontonummern: long[*] {readOnly, unique}
    - +/Name: String
      - ggf. taucht der Abgeleitete Wert garnicht im Programm auf, sondern nur über getName() {
        return this.vorname + " " + this.nachname; }
    - statisch unterstrichen
  - **3. Abschnitt: Methoden**
    - [Sichtbarkeit] Bezeichner(param1: Datentyp [Multiplizität][=Wert][{Eigenschaften}],p2: Datentyp,…) [:Rückgabetyp] [{Einschränkungen}]
      - rückgabetyp nur wenn nicht void
    - bsp: +getKKontostandFormatiert(): String
    - bsp2: +abheben(betrag: double) boolean
    - Einschränkungen/Eigenschaften
      - {abstract}
      - {query} : ändert nichts an this, nur abfrage (z.b. get)
      - {leaf} : final
      - {raised-Exception = IllegalArgumentException} :ggf geworfene Exceptions
    - statisch unterstrichen

- vor jeweiligen name die **Sichtbarkeit**:
  - + public
  - - private
  - # protected
  - ~ packageweit (ohne)
- **Weglassen** ist **erlaubt**, man zeigt nur die gerade relevanten Aspekte
  - Die **Bereiche** für **Attribute** und **Methoden** können weggelsssen werden
  - **Einzelne Attribute** und **Methoden** können weggelsassen werden
  - **Klassen**,m die man nicht darstellen möchte können einfach als **Datentyü benutz** werde (z.b. -geburtstag: LocalDate)
- **Statische** Mitglieder **Unterstrichen**
- Konstruktoren können in eigenem Bereich stehen

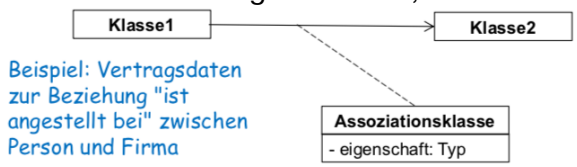
- **Kommentar**: Rechteck mit umgeknickter ecke, über gestrichelte Linie mit Klasse verbunden
- **Enums**
  - 

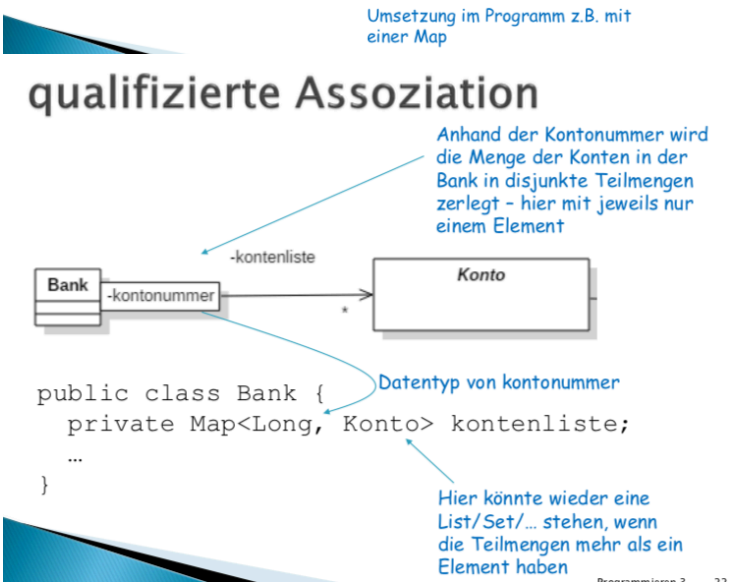
- **Beziehungen/Assoziationen**
  - Es werden **nur** statische (im sinne von **dauerhaft** über eine Methode hinaus **bestehenden**) **Beziehungen** zwischen klassen dargestellt
    - z.b. ein Konto gehört einem Kunden über alle mit ihm durchgeführten Aktionen hinweg
    - Flüchige Beziehungen, üblicherweise nicht im Klassendiagramm dargestellt
  - 

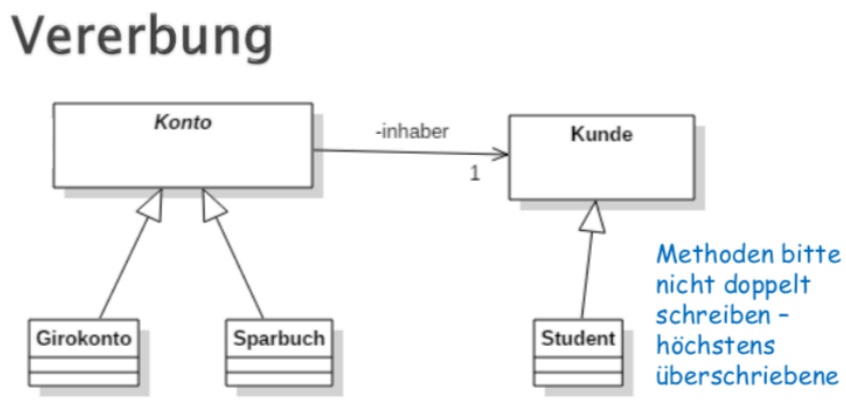
  - 

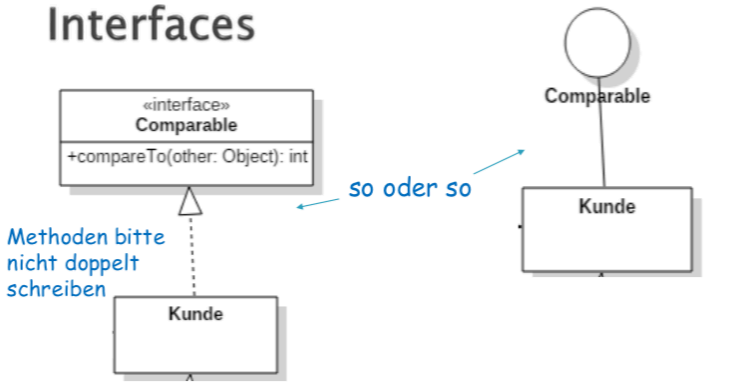
  - ```
    public class Klasse1
    {
    ◦ private Klasse2[] nameDerBeziehung;
    }
    ```
    Eine Collection wäre hier natürlich auch möglich...
    ```

- Assoziation mit Eingenschaften, die in Assoziationsklasse untergebracht werden können:



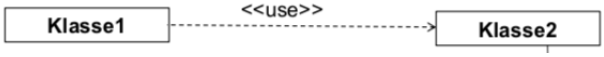
- 

- 

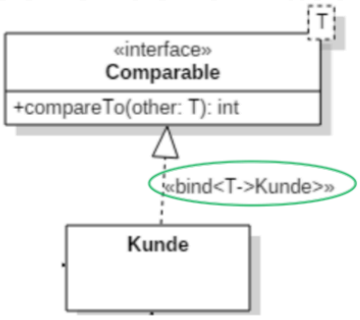
- 

- `public class Kunde implements Comparable`

allgemeine Abhängigkeit: kurzfristig, z.B. eine Methode von Klasse1 wirft eine Exception der Klasse2, ruft Methoden von Klasse2 auf,...

- 

Generische Klassen

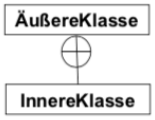



```
public class Kunde
          implements Comparable<Kunde> {
  public int compareTo(Kunde other)
  {…}
}
```

- bind kann an allen Beziehungen stehen (inkl. Vererbung nd Interfacerealisierung)
- T kann in den Eigenschaften und Methoden der generischen Klasse vorkommen
- Es können mehrere Typparameter und einschränkungen in dem feld wo T steht stehen

Was es sonst noch so gibt...

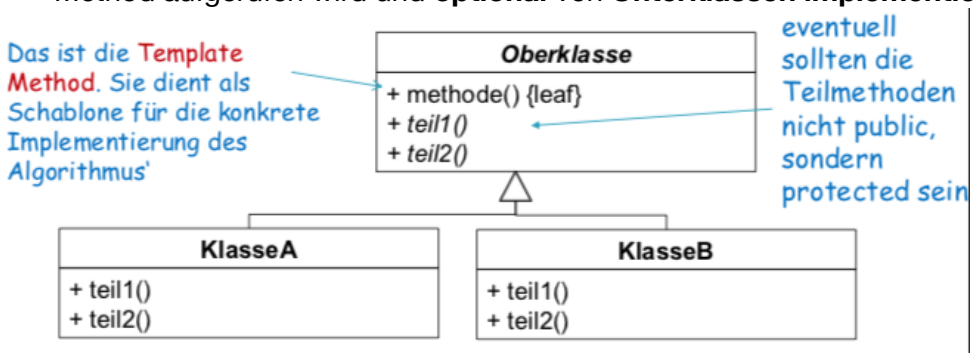
- mehrstellige Beziehungen
- Aggregationen und Kompositionen
- {xor}-Einschränkungen
- Rollen
- innere Klassen
- Symbole statt Stereotypen
- Eigenschaften für nebenläufige Methoden: {sequential}, {guarded}, {concurrend}
- Beziehungen zwischen Paketen (im Paketdiagramm)



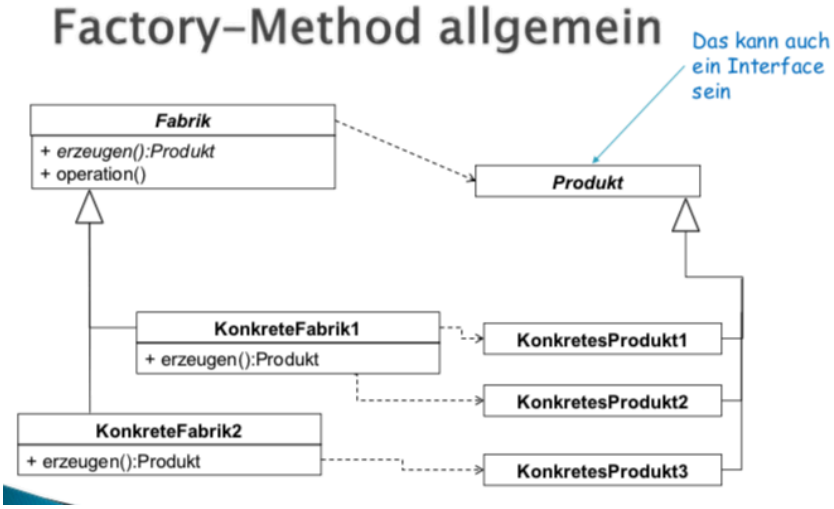
Entwurfsmuster

- Vorteile Entwurfsmuster
  - Es gibt immer wieder auftauchende Probleme, wofür es lohnt allgemeine Lösung zu haben
    - Weniger Entwicklungsarbeit im Einzelfall
    - Verständlich für andere Enwickler, weil jeder das Muster kennt
    - Wenn man zur Situation passendes Muster verwendet hat man sicher eine gute Lösung
  - Erweiterbarkeit
  - Vorhergesehene Änderungen können leicht gemacht werden
  - Einfaches Fehlerbeheben
- **Template Method**
  - **Verwendung**:
    - **Grundsätzlicher** Ablauf eines **Algorithmus** ist **klar** und wird sich (höchstwahrscheinlich/ vorhergesehenerweise) auch nie ändern. (Es gibt allgemein formulierbaren Ablaufplan)
    - In **Details** hängt die **konkrete Implementierung** aber vom jeweiligen objekt ab
      - Erwartung: In zukünft wird es vermutlich noch weitere neue sehr spezille Objekttypüen geben
    - Verhindert unschöne Situation, dass das gleiche Problem von verschiedenen Methoden (z.b. aus verschiedenen klassen) gelöst wird. Bsp: Tee zubereiten und Kaffekochen statt Getränk zubereiten
    - Bsp: Oberklasse AutomatenGetraenk hat die templateMethode final kochen(). Die Unterklassen Kaffee und Tee übeschreiben jeweils die methoden aufgießen() und zutatHinzufügen(), welche von der template methode kochen() verwendet werden (in oberklasse als abstrakte methoden). kochen() kann außerdem allgemeine Teile beinhalten, wie z.b. inTasseSchuetten()

- Der eigentliche **Algorithmus** ist **in** der **Oberklasse** implementiert (und wird nicht überschrieben, ist final), **ruft** aber für die **Details abstrakte Methoden** auf (zusätzlich zu eigenem code, eigenen hilfsmethoden und **ggf**. **Hook-Methoden**)
  - jede **konkrete Klasse** implementiert nur diese **Detail-Methoden**
  - Hook-Methoden
    - Methode mit leerer Implementierung (ggf. mit Standardrückgabe), die von der Template Method aufgerufen wird und **optional** von **Unterklassen implementiert** werden kann.

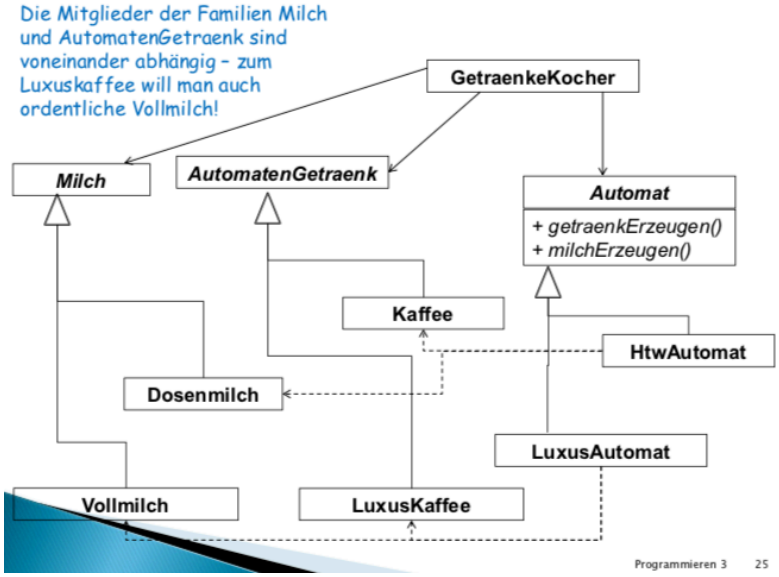


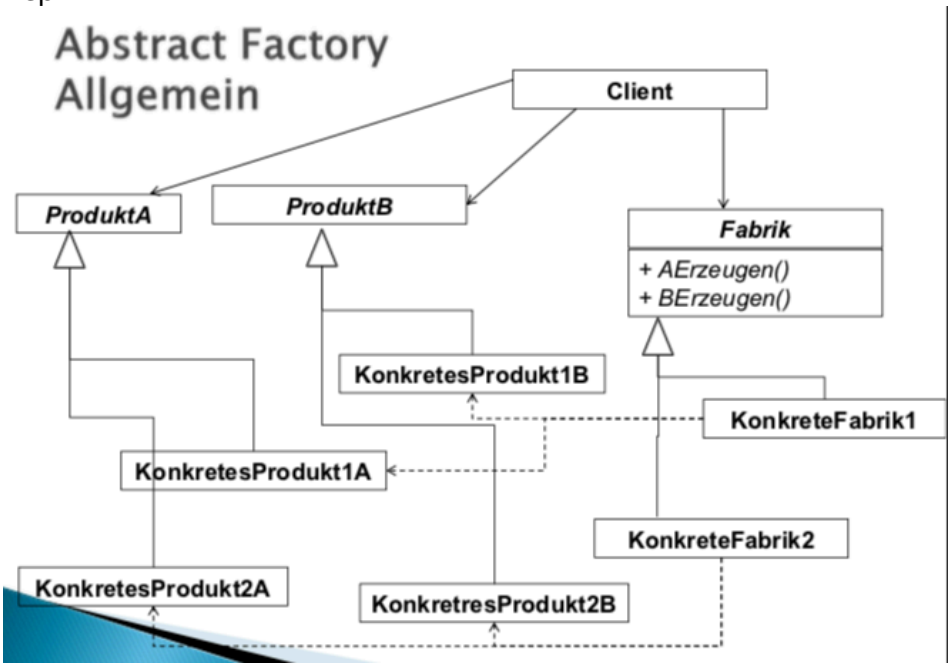
- **Factory Method**
  - **Verwendung**:
    - Man hat meherere verwandte Klassen, kann aber erst zur Laufzeit entscheiden welche genau instanziiert werden soll (Abhängig von einer oder mehreren Bedingungen).
    - Die Auswahl der verschiedenen Klassen soll erweiterbar sein
  - **Muster**:
    - Abstrakte Klasse Fabrik
      - enthält **abstrakte erzeugen-Methode**, die ein Objekt vom Typ Produkt erstellt
      - **Produkt** ist eine **abstrakte Klasse** oder ein **Interface**
        - wird von den **konkreten Produktklassen implementiert**
      - kann weitere Operationen, die nicht abstrakt sind enthalten
      - wird von **konkreten** Fabriken **implementiert**
        - **diese erzeugen** jew. ein oder mehrere **Produkte**
  - bsp: Abstrakter Automat hat abstrakte methode *erzeugen():Automatengetränk* und die methode getraenkKochen(). Automat wird implementiert von LuxusAutomat und HTWAutomat, die beide die methode erzeugen() implementieren. Automatengetränk wird implementiert/geerbt von Kaffe, Tee, Kakao und evt. weiteren.



  - **Abstract Factory** (Erweiterung des Factory-Method Musters) #TODOO implement
    - hat man evtl. **mehrere Familien verweander Klassen** (jew. eine Gruppe **zusammengehörender Klassen**, deren Mitglieder jew. für die **gleichen Aufgaben** zuständig sind), bei denen erst zur laufzeit entschieden wird welche (Familie) instanziiert wird

- Wirkliche trennung von Erzeugen der Objekte und die Arbeit mit ihnen
- Bsp:

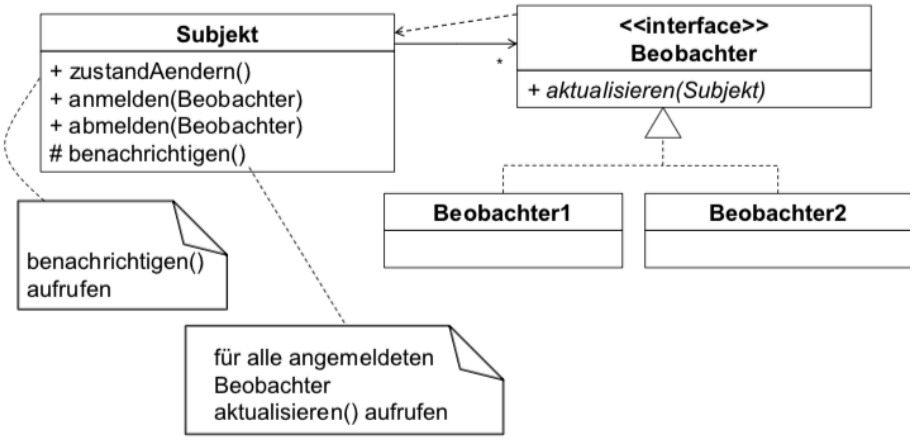




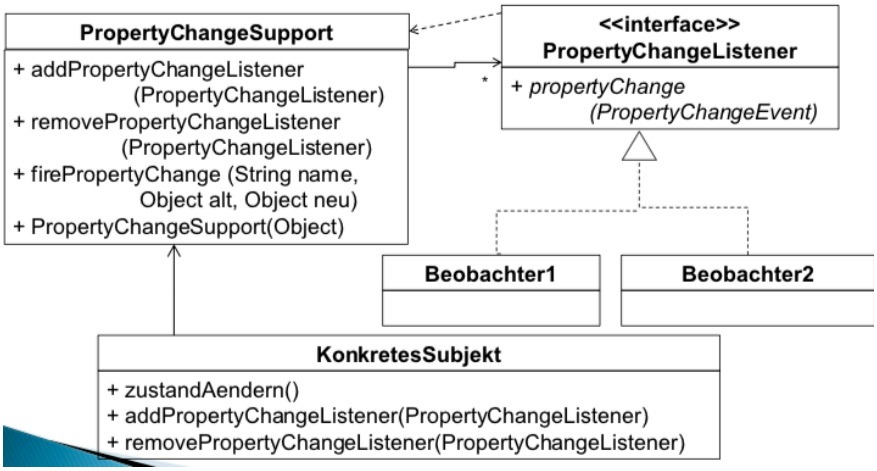
- **Observer**
  - Verwendung:
    - **Subjekt ändert** im Laufe der Zeit seinen **Zustand**. **Meherere** (bel. viele) **Beobachter** sollen über jede **Zustandsänderung informiert** und entsprechend aktualisiert werden
    - **Subjekt** (beobachtetes Objekt) soll **unabhängig** von den tatsöchlichen **beobachtern** geschrieben sein
      - Prinzip: Strebe nach Entwürfen mit lockerer Bindung
        - Subjekt wei0ß nicht welche Beobachter genau und wie viele es gibt. Einzige notwendige informatio: Sie implementieren alle die Schnittstelle Beobachter, haben also eine aktualisieren()-Methode
        - —> Man kann Beobachter ändern/austauschen/neu hinzufgen ohne den Code des Subjeks ändern zu müssen
  - Subjekt (z.b. Wetterdaten) hat methode, die **Zustand ändert** und **methoden** zum **an** und **abmelden** von **beobachtern**. **Beobachter** ist ein **Interface**, welches eine Methode **aktualisieren**(Wetterdaten) enthält und von konkreten beobahtern implementiert wird (z.b. Windfinder, StatistikAnzeige).
    - Das Beobachter-Interface kann mehrere Methoden haben um unterschiedlich auf verschiedene Arten von Zustandsveränderungen zu reagieren
    - die **methode** die **zustand verändert** (oder noch besser die von ihr aufgerufene methode benachrichtigen()) **benachrichtigt alle** angemeldeten **Beobachter**, z.b.
      protected void benachrichtigen() {

```
  beobachterliste.foreach( b-> b.aktualisieren(Wetterdaten.this));
}
public void setMesswerte(…){
  …
  this.benachrichtigen();
}
```

Observer-Lösung allgemein



Observer-Pattern in Java



Java-Observer im Wetter-Beispiel

```
public class Wetterdaten{                                    Subjekt
  private PropertyChangeSupport support
              = new PropertyChangeSupport(this);
  public void setMesswerte(…){ …
    support.firePropertyChange("temp", alt, neu);…
  }
  public addPropertyChangeListener                    +
                    (PropertyChangeListener pcl) {    removeProperty
    support.addPropertyChangeListener(pcl);           ChangeListener
  }
}
```

```
public class WetterOberflaeche implements
                                  PropertyChangeListener {
  public void propertyChange(PropertyChangeEvent evt){
    //evt.getNewValue();                                Beobachter
  }
  …
}
```

```
public void main(…) {
  Wetterdaten subjekt;
  WetterOberflaeche beobachter;
  …
  subjekt.addPropertyChangeListener(beobachter);
}
```

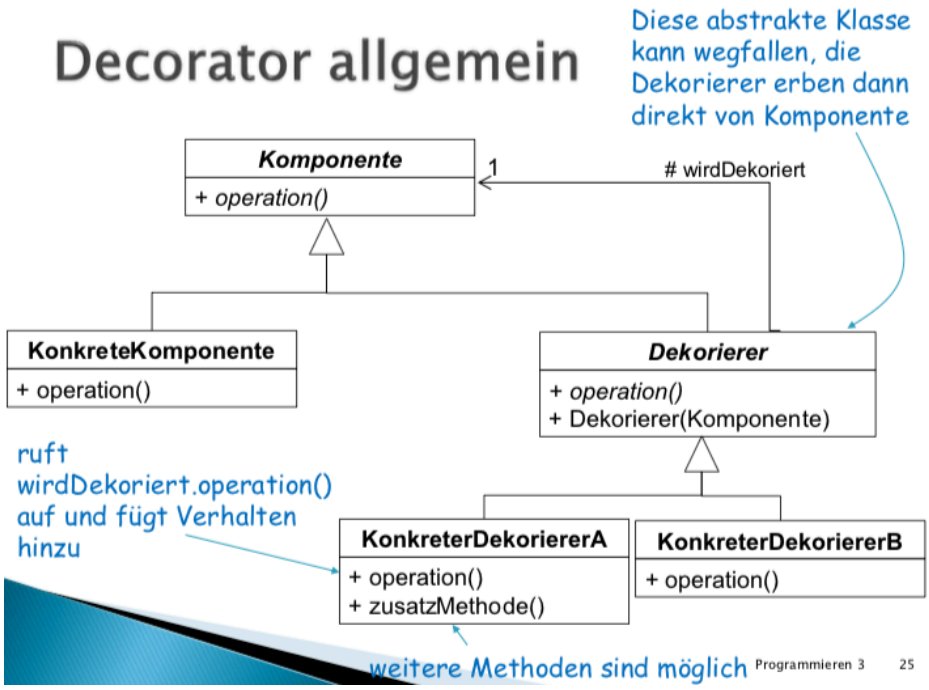
- Teilweise ist Observer-Pattern so dargestellt, dass die konkreten Beobachter dauerhafte Beziehung zum Subjekt haben (dieses also als Parameter haben, nicht andersrum). Diese muss anfangs (z.b. bei Konstruktoraufruf etabliert werden

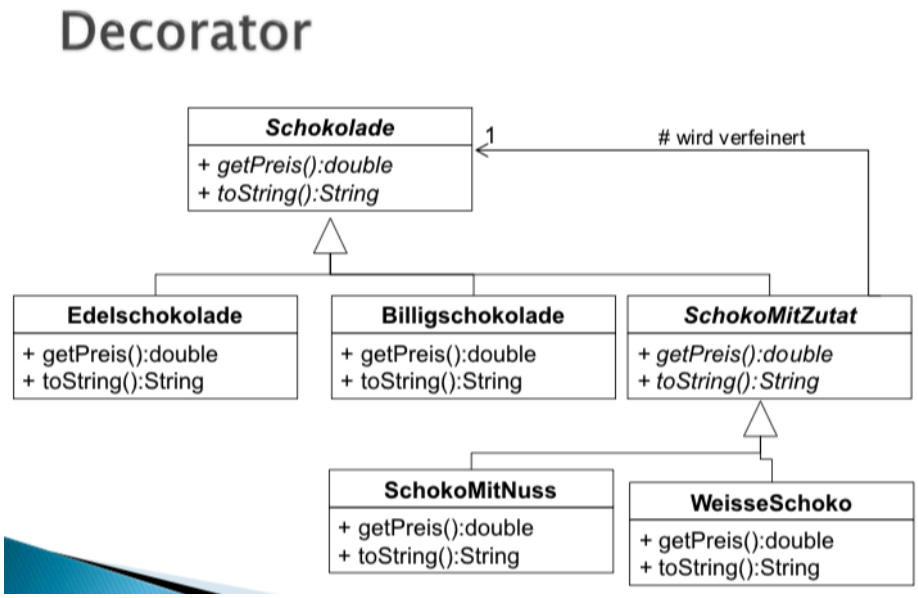
- **Singleton**
  - Verwendung: Von einer Klasse darf es im ges. Programm nur eine Instanz geben, die vpn überall zugreifbar sein soll
    - z.b.
  - privater Konstruktor, statische Variable in der die Instanz gespeichert wird. Getter. Alternativ erzeugung erst in getter, wenn instanz gebraucht

```
public class SingletonKlasse {
  private static SingletonKlasse einzigeInstance; // alternativ hier gleich initialisieren
  private SingletonKlasse() {…}
  public static SingletonKlasse getInstanz() {
    if(einzigeInstanz == null) {
      // ggf. Vorbereitung für Konstruktoraufruf
      einzigeInstanz  = new SingletonKlasse();
    }
    return einzigeInstanz;
  }
```

- **State**
  - Verwendung:
    - Für ein Objekt sollen **gleiche Methodenaufrufe** je nach **Zustand**, in dem sich das Objekt befinet, **unterschiedlich** ablaufen.
      - diese Methodenaufrufe bewirken meist selbst den Zustandswechsel
    - Umgehen riesiger abfragen nach aktuellen ZUstand in den Methoden und ermöglichn, dass Zustände leicht austauschbar sind (Kapsle was variieirt)
  - Ein **Interface Zustand**, was die **zustandsabhängigen Methoden** enthält
    - jeder Zustand wird durch eine Klasse, die dieses Interface implementiert repräsentiert und überschreibt entsprechend die darin enthaltenen Methoden
      - Als Parameter wird (u.A.) das Objekt der Hauptklasse übergeben
    - Variante
      - Statt Zustand-Interface **abstrakte Klasse** verwenden
        - **Standardverhalten** kann bereits implementiert sein und muss nur überschrieben werden, wenn der Zustand anderes Verhalten auslöst
        - Ggf. Eigenschaft in der zu steuerndes Objekt gespeichert wird, damit es nicht bei jedem Methodenaufruf übergeben werden muss.
  - In der **Hauptklasse** rufen die **zustandsabhängigen Methoden** die **entsprechende Methode** von dem Objekt des **aktuellen Zustands** auf.
    - Das objekt des **aktuellen Zustands** wird in der **Hauptklasse** gespeichert und **initial** auf den **Anfangszustand** gesetzt. Über eine Methode **setZustand**(Zustand neu) kann der aktuelle Zustand verändert werden (welche höchstwahrscheinlich von den zustandsabhängigen methoden der zustandsKlassen aufgerufen wird).
  - Vorteile
    - Leichtes hinzufügen von Zuständen ohne veränderung der Hauptklasse
  - Nachteile
    - Zugriff auf im Objekt der Hauptklasse gespeicherten Informationen
      - Lösung 1: Packageweite Sichtbarkeit
      - Lösung 2: Innere Klassen für die Zustäände nutzen

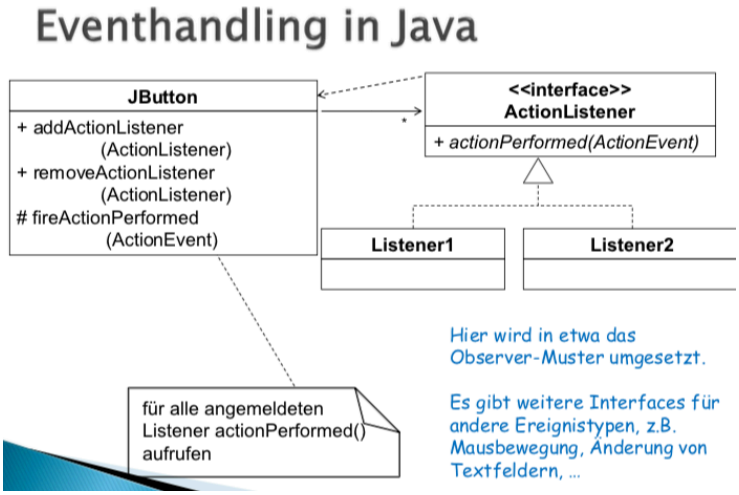
- **Decorator**
  - Verwendung
    - Objekte sollen **dynamisch** zur Laufzeit **verschiedene Erweiterungen** der **Funktionalität** hinzugefügt werden
      - Sowohl neue Methoden, als auch vorhandene im Verhalten erweitern
    - **Vorraussetzung**: **Dekorierer** müssen auf jede Basis- oder beriets **verfeinerte Komponente angewendet** werden **können**
      - **Verhaltenserweiterung** sollten **unabhängig** von **einander** sein
  - **Aufruf**: Schokolade s = new DecoratorA(new Decorator C(new DecoratorB(new KonkreteBasiskomponente())));

- 

  - **Dekorierer** haben die **bisherige** Komponente als **Eigenschaft gespeichert** (#**wirdDekoriert**), welche im **Konstruktor** gesettzt wird (KonkreterDekoriererA(Komponente wirdDekoriert))
  - **Dekorierer überschrieiben** (**ggf**. **Vorhandene) Methoden** unter Verwendung der bisherigen implementiereung, z.b.
    public double getPreis() {
      return 0.3 + wirdDekoriert.getPreis();
    }
  - Dekorierer fügen ggf. weitere Methoden hinzu
  - KonkreteKomponente ist kein Dekorierer, benötogt also auch in Konstruktoraufruf keine Komponente
- 

  - Edelschokolade, Billigschokolade sind Konkrete Komponenten

- **Architekturmuster Model-View-Controller (MVC)**
  - Architekturmuster: nur eine Idee, keine feste Festlegung wie in Entwurfsmustern
  - Vorteile:
    - klare Aufgabentrennung
      - kleine übersichtliche Klassen
      - Programmierer dürfen Spezialgebiet haben

    - Model in anderen Zusammenhängen wiederverwendbar
    - Austausch der View relativ leicht möglich
    - Komplexte Logik im Controller kann mehrfach verwendet werden
        - bsp: befehl ist über das normale Menü, per tastaturkürzel oder über das Kontextmenü aufrufbar
- Nachteil: Overkill bei kleinen Problemen
- **Model**
    - Datenmodell
    - entspricht üblicherweise einem “Ding aus der Wirklichkeit (z.b. Wetterdaten-Klasse)
    - Enthält Daten mit zugehörigen get-Methoden
    - Enthält meist auch die Geschäftslogik, also das was man mit dem “Ding” in der Wirklichkeit machen kann
    - keine Benutzerkommunikaiton
    - Verwaltet üblicherweise die views nach dem Observer-muster
- **View**
    - Darstellungsklasse
    - interagiert mit dem Benutzer (Ein und Ausgabe)
    - reagiert auf Änderungen im Model nach dem Observer-Muster
    - Eingaben werden nicht hier verarbeitet, sondern an den Controller weitergegeben
        - höchstens überprüfung der Eingabe auf Korrektheit (z.b. Zahlenformat)
    - Verwendet nur die get-Methoden des Models
- **Controller**
    - erzeugt das Modell (oder nimmt es von au0en entgegen)
    - erzeugt das/die View-Objekt(e)
    - reagiert auf die Elngaben des Benutzers und leitet sie an das Model weiter
        - aufruf der Methoden für Geschäftslogik (set-Methoden)
    - keine Benutzerkommunikation!
    - Variante: Controller registriert sich bei Model als Beobachter und steuert bei Änderungen die View(s)
- 

Konventionen / best Practices:

- Sprechende Namen verwenden
- Variablennamen-Benennung mit lowerCamelCase
- Klassenname UpperCamelCase
- Eigenschaften private
- Dokumentation
- Kommentierung
- korrekte Code-Formatierung
- Jede klasse/Methode hat genau eine Aufgabe, die sie vollständig erfüllt (Atomarität)
    - Aufteilung in sinnvolle, übersichtliche Einheiten (Dateien, Methoden, Klassen)
- newline mit System.lineSeparator(), statt \n
- Hollywood-Prinzip (“Don’t call us - we call you!”)

    - Highlevel-Komponente ruft die Methoden der Lowlevel-Komponentenn auft, wenn sie sie braucht
    - Lowlevel-Komponente greift nicht in den Ablauf des Algorithmus ein (ruft also nicht die Highlevel-Komponente auf) (Teilweise nötig)
- Don’t repeat yourself (DRY)
- Kapsle was variiert, behalte bei was immer gleich bleibt
- Zirkuläre Abhängigkeiten vermeiden (Aufrufe nur in einer Richtung, Hollywood-Prinzip)
- Klassen sollten offen für Erweiterungen aber geschlossen gegenüber Veränderungen sein
    - Leichtes Erweitern durch hinzufügen neuer klassen, nicht aber durhc ändern bestehenden Codes.
        - dafür hilfreich: Objekte, die einen Teil des Codes tragen als parameter übergeben
- Auf Abstraktionen, nicht auf konkrete Klssen stützen
    - Datentypen von Variablen sollten keine konkreten Klassen sein
    - Keine KLasse sollte von einer konkreten Klasse abgeleietet sein
    - keine Methdode sollte eine bereits implementierte Methode überschreiben
- Strebe nach Entwürfen mit lockerer Bindung
    - auch bei interagierenden Objekten sollten die Partner austauschbar sein
- Programmieren sie auf eine Schnittstelle (Interface), nicht auf eine konkrete Klasse
- Konvention vor Konfiguration (Convenience over configuration)
-

Literatur:
- Entwurfsmuster:
    - Entwurfsmuster von Kopf bis Fuß, Eric & Elisabeth Freeman, O‘Reilly-Verlag
    - Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, Eric Gamma,..., Addison Wesley- Verlag
- MVC: http://blog.bigbasti.com/tutorial-model-view-controller-mvc-struktur-in-java-projekten-nutzen/
- Programmierprinzipien
    - https://glossar.hs-augsburg.de/Programmierprinzipien
- API-Dokumentation:
    - https://docs.oracle.com/javase/10/docs/api/index.html?overview-summary.html
- Spezifikation von Java (für alle, die es hundertprozentig genau wissen möchten):
    - https://docs.oracle.com/javase/specs/jls/se10/html/index.html
- mehr siehe VL01
- Streams:
    - http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/
    - http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/
- Lambda-Ausdrücke: http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Software:
- StarUML: http://staruml.io/download
-