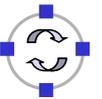


# Programmierung und Algorithmen

## Kapitel 9 Objektorientierung und Anwendungsprogrammierung



### Überblick

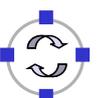
Einführung in die Objektorientierung

Objektorientierung in Java

Registermaschine in Java

Markov-Tafeln in Java

Java Code Conventions



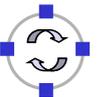
- ❑ Bisher:
  - ❑ Prozedurale Sichtweise
  - ❑ Prozeduren/Funktionen, die Daten manipulieren
  - ❑ Beispiel: Sortierroutine für `int`-Felder
- ❑ Nachteile:
  - ❑ Struktur der Daten muss bekannt sein
  - ❑ Keine logische Verbindung von Daten und darauf definierten Operationen
- ❑ Sortierung von beliebigen Feldern (Zeichenketten, Studenten usw.)?
  - ❑ Spezifische Sortierroutine
  - ❑ Fallunterscheidungen



- ❑ Im Mittelpunkt:
  - ❑ Nicht das **Wie?**: die Ausführung als Folge von Anweisungen
  - ❑ Sondern das **Was?**: die in einer Anwendung existierenden Objekte, deren Struktur und Verhalten
  - ❑ Objekte als besondere Daten- und Programmstruktur, die Eigenschaften und darauf definierte Operationen (Methoden) besitzen



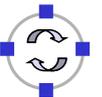
- Kapselung:
  - Verbergen von internen Eigenschaften
  - Zugriff nur durch bestimmte eigene Methoden
  - Geheimhaltung (z.B. Zugriff nur auf Diplomnote, nicht auf Teilnoten)
  - Vermeiden von inkonsistenten Änderungen (z.B. Änderung der Matrikelnummer bei Studierenden unerwünscht)
- Vererbung:
  - Erweiterung existierender Klassen
  - Hinzufügen neuer Eigenschaften und Methoden



- Beispiel: Zeichenprogramm mit geometrischen Elementen (Figuren)
- Objekte repräsentieren die zu zeichnenden Figuren:
  - Linien, Kreise, Rechtecke usw.
  - Jedes Objekt hat Eigenschaften und „weiß“, wie es zu zeichnen oder zu drucken ist



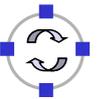
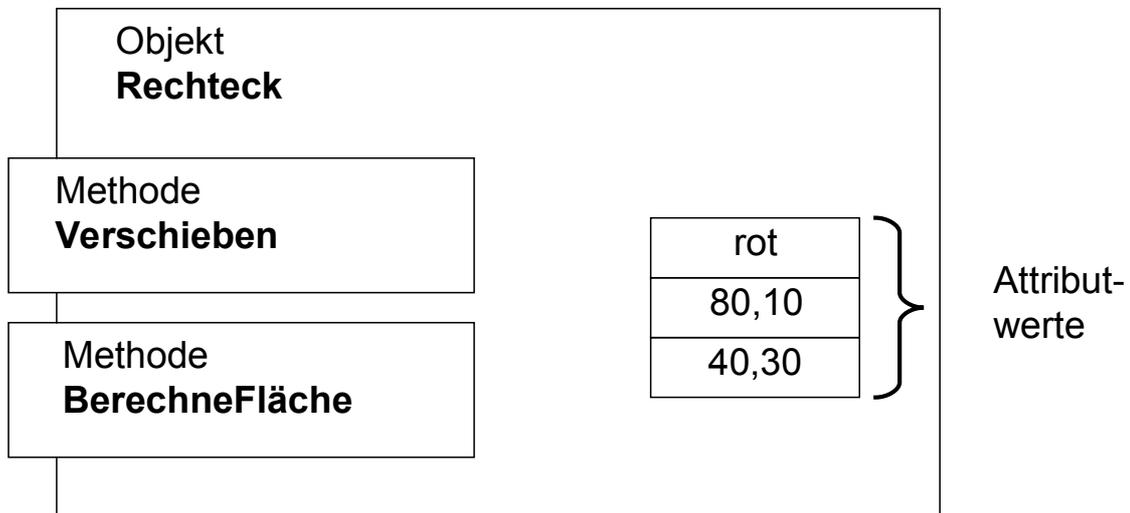
- ❑ Feld von Objekten (Zeichenketten, Studenten)
- ❑ Jedes Objekt „weiß“, wie es mit anderen Objekten verglichen werden kann:
  - ❑ Zeichenkette: alphabetisch
  - ❑ Studenten: Vergleich der Matrikelnummern



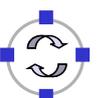
- ❑ Repräsentation eines „Dings“
- ❑ Identität: Eigenschaft, durch die sich das Objekt von anderen unterscheidet
- ❑ Statische Eigenschaften (Zustand): Attribute
- ❑ Dynamische Eigenschaften (Verhalten): Methoden
- ❑ Attribut: Variable
- ❑ Methode: Zusammenfassung von Anweisungen zu einer logischen Einheit (Funktion, Prozedur)



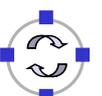
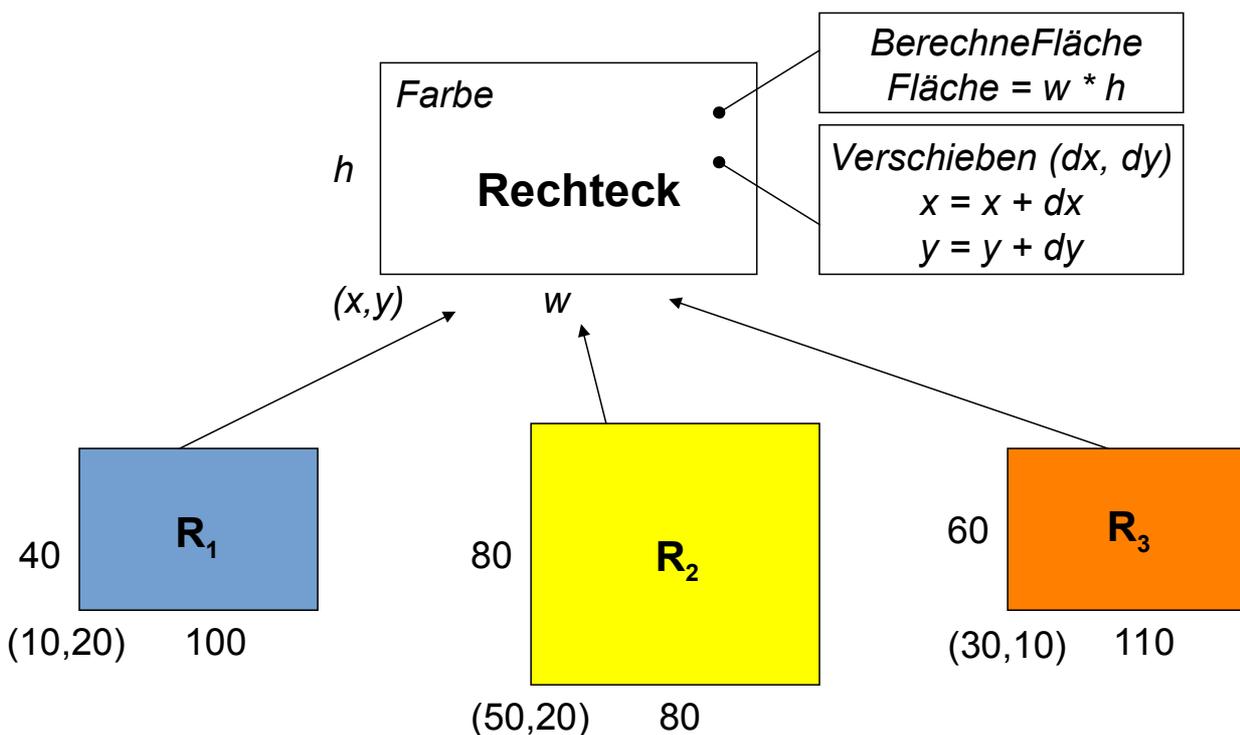
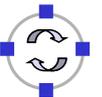
- Geometrische Figuren: Kreise, Linien, Rechtecke, . . .
- Eigenschaften: Farbe, Position, Abmessung
- Methoden: Verschieben, Drehen, Drucken, BerechneFläche
- Objekt



- Objekte interagieren durch Austausch von Nachrichten
- Beispiel:
  - Nachricht an Rechteck #1: „Verschieben um 10 mm nach rechts und 20 mm nach oben“
- Nachricht:
  - Aufruf der Methode des Objektes (hier: „Verschieben“)
  - Änderung des Zustandes (hier: Position)



- Programmierung von Objekten:
  - Vereinbarung von Variablen
  - Implementierung von Methoden
  - Nicht für einzelne Objekte sondern für Klassen
  
- Klasse:
  - Zusammenfassung von gleichartigen Objekten
  - Datentyp, der Eigenschaften von Objekten beschreibt
  - Objekte einer Klasse: Instanzen

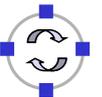


- Klasse aller Rechtecke, Klasse aller Kreise, . . .
- Klasse Rechteck:
  - Attribute: Farbe, x-Position, y-Position, Höhe, Breite
  - Methoden:
    - Verschieben um dx, dy:

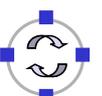
```
x-Position := x-Position + dx  
y-Position := y-Position + dy
```

- BerechneFläche:

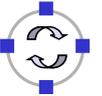
```
Fläche := Breite * Höhe
```



- Klassen als Hauptstrukturierungsmittel
- Aufbau von Klassenhierarchien durch Vererbung
- Wurzelklasse `java.lang.Object`
- Instantiierung von Objekten mittels **new**-Operator
- Objektinteraktion durch Aufruf von Methoden



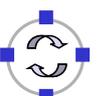
```
public class Rechteck {
    int x, y, b, h;
    public Rechteck() {
        x = y = 0; b = h = 10;
    }
    public void verschieben(int dx, int dy) {
        x = x + dx; y = y + dy;
    }
    public int berechneFlaeche() {
        return b * h;
    }
}
```



- Definieren Eigenschaften (Zustand) eines Objektes dieser Klasse

```
private int x, y, b, h;
```

- Variablen lokal zum Objekt
- Sichtbarkeit
  - **public**: von anderen Klassen/Objekten benutzbar
  - **private**: nur von Objekten der eigenen Klasse
  - **protected**: nur von Objekten der eigenen Klasse bzw. davon abgeleitete Klassen



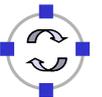
- ❑ Zusammenfassung von Anweisungen, die Zustand des Objektes ändern können (Seiteneffekt)
- ❑ Operationen auf Objekten

```
void verschieben (int dx, int dy)
```

- ❑ Oder Zugriff auf (private) Attribute

```
public void setzeGroesse(int d1, int d2) {  
    b = d1; h = d2;  
}  
  
public int getX() { return x; }
```

- ❑ Sichtbarkeit: **public**, **protected**, **private**



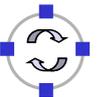
- ❑ Nicht-statische Eigenschaften und Methoden betreffen den Zustand (und das Verhalten) eines Objektes
- ❑ Statische Eigenschaften und Methoden (**static**) betreffen den Zustand (und das Verhalten) einer Klasse
  - ❑ Für alle Objekte einer Klasse gültig (global)
  - ❑ Kein Objekt für Aufruf notwendig
  - ❑ Verwendung:
    - Definition von Konstanten und Hilfsfunktionen
    - Definition der Funktion `main` (`main` wird aufgerufen, bevor das erste Objekt erzeugt werden kann)



- ❑ Spezielle Methode zur Initialisierung eines Objektes beim Erzeugen
- ❑ Belegung mit Standardwerten
- ❑ Bezeichner = Klassenname

```
Rechteck() {  
    x = 0; y = 0; b = 10; h = 10;  
}
```

- ❑ Mehrere Konstruktoren mit unterschiedlichen Parameterlisten möglich



- ❑ Klassen: Referenz-Datentypen
- ❑ Variable als typisierter Verweis auf Objekt

```
Klassenname Varname;
```

- ❑ Beispiel:

```
Rechteck r1; Kreis k1; String str;
```

- ❑ Erzeugung (Instantiierung) durch **new**-Operator

```
new Klassenname (Parameter)
```



- Liefert Referenz (Verweis) auf Objekt

```
Rechteck r1 = new Rechteck();
```

- Impliziter Aufruf des Konstruktors: Initialisierung der Eigenschaften (hier: `r1.x = 0` und `r1.y = 0`)
- Instanziierung mit Parametern

```
Rechteck r2 = new Rechteck(10, 15);
```

- Erfordert Konstruktor mit Parametern

```
public Rechteck(int xi, int yi) {  
    x = xi; y = yi;  
}
```



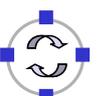
- Zugriff auf Objekteigenschaften (Attribute)
  - Wie Zugriff auf Variablen
  - Durch Voranstellen der Referenz-Variablen

```
r1.x = 34; int y = r1.y;
```

- Nur möglich, wenn öffentliche (`public`) Attribute
- Aufruf von Methoden
  - Durch Voranstellen der Referenz-Variablen

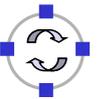
```
r1.verschieben(10, 20);
```

- Beeinflusst nur das Objekt `r1` !
- Innerhalb der Klasse: ohne Referenzvariable

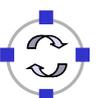


```
// Drei Rechtecke erzeugen
Rechteck r1 = new Rechteck();
Rechteck r2 = new Rechteck(10, 10);
Rechteck r3 = new Rechteck();

// und einzeln verschieben
r1.verschiebe(20, 30);
r2.verschiebe(30, 30);
r3.verschiebe(-40, 10);
r2.verschiebe(100, 10);
```

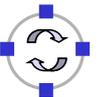


- ❑ In Java nicht notwendig, da automatische Speicherbereinigung (Garbage Collection)
- ❑ Objekt wird automatisch gelöscht, wenn es nicht mehr benötigt wird, d.h. wenn keine Variable und kein anderes Objekt mehr darauf verweist



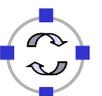
## □ Methode zum Vergleich bzgl. der Größe

```
public class Rechteck {  
    ...  
    public int vergleiche(Recteck r) {  
        int meineFl = berechneFlaeche();  
        int andereFl = r.berechneFlaeche();  
        if (meineFl < andereFl)  
            return -1;  
        else if (meineFl > andereFl)  
            return 1;  
        else  
            return 0;  
    }  
}
```

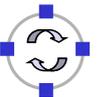


## □ Anwendung

```
Rechteck r1 = new Rechteck(20, 40);  
r1.setzeGroesse(80, 30);  
Rechteck r2 = new Rechteck(60, 90);  
r2.setzeGroesse(50, 40);  
int res = r1.vergleiche(r2);  
if (res == 0)  
    System.out.println("r1 == r2");  
else if (res < 0)  
    System.out.println("r1 < r2");  
else  
    System.out.println("r1 > r2");
```



```
type RatNumber
import Int, Bool
operators
mk_rat:    Int × Int → RatNumber
nom:       RatNumber → Int
denom:     RatNumber → Int
equal:     RatNumber × RatNumber → Bool
is_zero:   RatNumber → Bool
add:       RatNumber × RatNumber → RatNumber
normalize: RatNumber → RatNumber
```

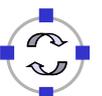


## □ Implementierung des ADT als Klasse

```
public class RatNumber {
    private int num = 0, denom = 1;
    ...
}
```

## □ Konstruktoren

```
public RatNumber() {}
public RatNumber(int n, int d) {
    num = d > 0 ? n : -n;
    denom = Math.abs(d);
    normalize();
}
```



## □ Selektoren

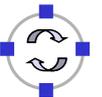
```
public int numerator()    { return num; }  
public int denominator() { return denom; }
```

## □ Implementierung der Funktion

```
add: RatNumber × RatNumber → RatNumber
```

## □ Varianten

```
static RatNumber add(RatNumber z1, RatNumber z2)  
void add(RatNumber z)  
RatNumber add(RatNumber z)
```



## □ Üblicherweise

```
public RatNumber add(RatNumber n) {  
    int n, d;  
    n = numerator() * r.denominator() +  
        r.numerator() * denominator();  
    d = denominator() * r.denominator();  
    return new RatNumber(n, d);  
}
```



## Erzeugen

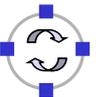
```
RatNumber r1 = new RatNumber(1, 3);  
RatNumber r2 = new RatNumber(3, 4);
```

## Rechnen

```
RatNumber res = r1.add(r2);
```

## Ausgeben

```
System.out.println("Ergebnis = " + res);
```



- Java-Klassenbibliothek:
- Vielzahl vordefinierter Klassen
- Beispiele:
  - `java.lang.String` für Zeichenketten
  - Datenstrukturen: Listen, Felder, ...
  - Ein-/Ausgabe: Dateien, Verzeichnisse, Netzwerkverbindungen, ...
  - Klassen für graphische Benutzerschnittstellen: Fenster, Schaltflächen (Buttons), Schieberegler, ...



## Erzeugung eines Strings

```
String s1 = new String("Hallo Java");  
String s2 = "Eine Zeichenkette";
```

## Länge eines Strings

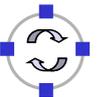
```
s1.length() // 10  
s2.length() // 17
```

## Zeichenweiser Zugriff

```
s1.charAt(2) // 'l'
```

## Vergleich

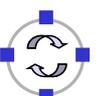
```
s1.equals(s2) // false
```



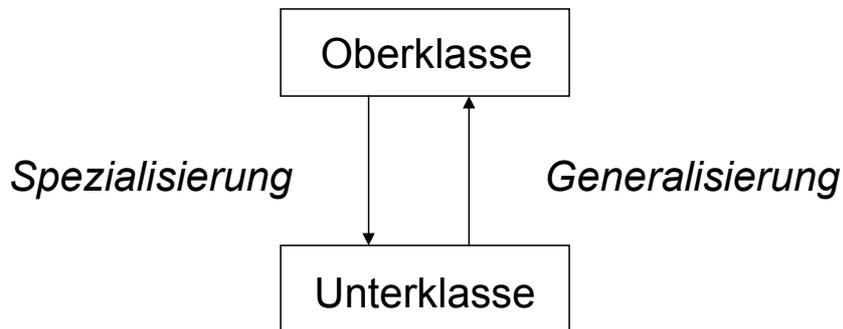
## Klassen zur Kapselung der primitiven Datentypen

- Beispiele: `int`, `double`, `byte`
- Nutzung von Werten primitiver Datentypen an Stellen, an denen Objekte erwartet werden (z. B. Kollektionen)
- Klassen `java.lang.Integer`, `java.lang.Double`, ...
- Beispiel

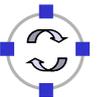
```
Integer iobj = new Integer(20);  
int ival = iobj.intValue();
```



- Definition von Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen Klassen



- **Spezialisierung:** Unterklasse erbt Methoden und Attribute der Oberklasse, ergänzt um neue
- **Generalisierung:** Oberklasse definiert gemeinsame Attribute und Methoden aller Unterklassen



- Objekte eines Zeichenprogramms
- Oberklasse `GeomObjekt`
  - Attribute: Position, Abmessung, Farbe
  - Methoden: `verschieben`, `zeichnen` (allgemein)
- Unterklassen `Rechteck`, `Oval`, `Linie`
  - Zusätzliche Attribute: Füllfarbe, Linienart, ...
  - Methoden: Spezialisierung von `zeichnen`

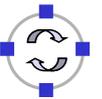


Syntax

```
class Unterklasse extends Oberklasse
```

 Beispiel

```
class Rechteck extends GeomObjekt {  
    int b, h; // zusätzliche Attribute  
  
    public Rechteck() {  
        // impliziter Aufruf von GeomObject()  
        b = h = 0;  
    }  
    ... }
```

 Geerbte Attribute / Methoden werden nicht erneut deklariert

- Oberklasse aller Java-Klassen
- Muss nicht explizit angegeben werden

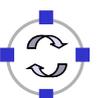
```
class GeomObject extends Object { ... }
```

äquivalent zu

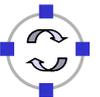
```
class GeomObject { ... }
```

 Wichtige Methoden:

- boolean** equals(Object o) : Vergleich zweier Objekte
- String toString() : String-Repräsentation liefern
- Object clone() : Objekt kopieren

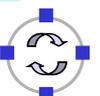


- ❑ Allgemein: „Vielgestaltigkeit“
- ❑ Als OO-Konzept: Methode kann in verschiedenen Formen implementiert werden
- ❑ Beispiele:
  - ❑ „Zeichnen“ für geometrische Objekte
  - ❑ „+“-Operator für `int` (Addition), `String` (Konkatenation)
- ❑ Formen:
  - ❑ Überschreiben (Overriding)
  - ❑ Überladen (Overloading)



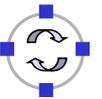
- ❑ Mehrfaches Verwenden eines Methodennamens innerhalb einer Klasse
- ❑ Unterscheidung durch verschiedene Anzahl bzw. Typen von Parametern; jedoch Ergebnistyp gleich

```
class Printer {  
    void print(int i) {  
        System.out.println("int = " + i);  
    }  
    void print(String s) {  
        System.out.println("String = " + s);  
    }  
}
```



- ❑ Auswahl der „richtigen“ Methode zur Laufzeit anhand des Parameters

```
Printer p = new Printer();  
p.print(12);           // int = 12  
p.print("Hallo");    // String = Hallo
```



- ❑ Methode einer Oberklasse wird in einer Unterklasse neu implementiert
- ❑ Signatur der Methode bleibt gleich

```
class KlasseA {  
    void print() {  
        System.out.println("KlasseA");  
    }  
}  
class KlasseB extends KlasseA {  
    void print() {  
        System.out.println("KlasseB");  
    }  
}
```



- ❑ Methodenauswahl abhängig vom Typ des aufgerufenen Objektes

```
KlasseA obj1 = new KlasseA();  
KlasseB obj2 = new KlasseB();  
  
obj1.print(); // Ausgabe: KlasseA  
obj2.print(); // Ausgabe: KlasseB
```

- ❑ Auswahl wird zur Laufzeit bestimmt, nicht durch Typ der Objektreferenz

```
obj1 = obj2; // Zuweisung muss jedoch erlaubt sein  
obj1.print(); // Ausgabe: KlasseB
```



- ❑ Implementierung gemeinsamer Methoden in Oberklasse nicht immer sinnvoll (z. B. „Zeichnen“ in `GeomObjekt`)
- ❑ **Abstrakte** Methode: ohne Implementierung
- ❑ **Schnittstelle** als Sammlung abstrakter Methoden
- ❑ Syntax

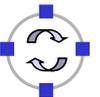
```
interface Name {  
    Methodensignaturen  
}
```

- ❑ Umgeht Probleme der fehlenden Mehrfachvererbung



- Methoden zum Speichern und Laden geometrischer Objekte

```
interface Speicherbar {  
    void speichern(OutputStream out);  
    void laden(InputStream in);  
}
```

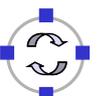


- Klassen können eine oder mehrere Schnittstellen **implementieren**, d. h. die abstrakten Methoden implementieren
- Notation

```
class Klasse implements Schnittstelle1,  
                        Schnittstelle2 ...{ ...}
```

- Beispiel

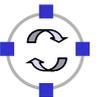
```
class Rechteck extends GeomObjekt  
    implements Speicherbar {  
    void zeichnen() { ...}  
    void speichern(OutputStream out) { ...}  
    void laden(InputStream in) { ...}  
}
```



- ❑ Umsetzung einer ausführbaren Registermaschine in Java
- ❑ Illustration eines abstrakten, einfachen Berechnungsmodells durch Verwendung einer Hochsprache
- ❑ **Ziel:** Anwendung Objektorientierung

(Code auf der Web-Seite zum Buch von Saake/Sattler)

- ❑ Siehe auch im Buch Saake/Sattler Abschnitt 6.5.2

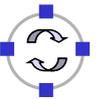


*”Eine Registermaschine hat ein **Programm**, welches aus **Instruktionen mit Parametern** besteht. Außerdem hat eine Registermaschine eine **Konfiguration**, die aus dem **Befehlszähler** und den **Registern** besteht.”*

- Abbildung dieses Sachverhalts auf Klassen möglicher Objekte und deren Attribute
- Freiheitsgrade bei Implementierung



- ❑ Befehlszähler als Variable `ic` (Instruction Counter)
- ❑ Feld `registers` für restliche Register, inklusive Akkumulator  $C_0$  als `registers[0]`
- ❑ Konstruktor: initialisiert Register mit 0
- ❑ Weitere Methoden:
  - ❑ Initialisierung der Register
  - ❑ Lesen und Setzen des Befehlszählers
  - ❑ Inkrementieren des Befehlszählers
  - ❑ Lesen und Setzen der anderen Register
  - ❑ **keine main-Methode!**

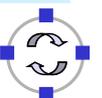


```
public class Configuration {
    public final static int NUM_REGISTERS = 10;
    int ic;
    int registers[] = new int[NUM_REGISTERS];

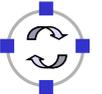
    public Configuration() {
        init();
    }

    public void init() {
        ic = 0;
        for (int i = 0; i < registers.length; i++)
            registers[i] = 0;
    }

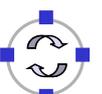
    ...
}
```



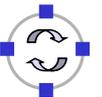
```
...  
  
public int getICounter() { return ic; }  
public void setICounter(int nic) { ic = nic; }  
public void incICounter() { ic++; }  
  
public void setRegister(int i, int val) {  
    registers[i] = val;  
}  
public int getRegister(int i) {  
    return registers[i];  
}  
}
```



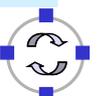
- ❑ Schnittstelle `Instruction` als einheitliche Festlegung für alle Befehle
- ❑ Programm besteht später aus Feld vom Typ `Instruction[]`
- ❑ Klassen für spezifische Befehle `Load`, `Store`, `Div` `IfGoto` etc. implementieren `Instruction`-Schnittstelle
- ❑ Konkrete Programmschritte sind Objekte der Befehlsklassen mit Parametern als Objektvariablen



```
public interface Instruction {  
  
    void eval(Configuration config);  
  
}
```



```
public class Load implements Instruction {  
    private int reg;  
  
    public Load(int i) {  
        reg = i;  
    }  
  
    public void eval(Configuration config) {  
        config.setRegister(0,  
            config.getRegister(reg));  
        config.incICounter();  
    }  
}
```



```
public class IfGoto implements Instruction {
    private int pos;

    public IfGoto(int p) {
        pos = p;
    }

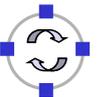
    public void eval(Configuration config) {
        if (config.getRegister(0) == 0)
            config.setICounter(pos - 1);
        else
            config.incICounter();
    }
}
```



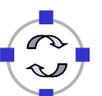
- ❑ Klasse mit Referenzen auf
  - ❑ Konfiguration `Configuration` der Maschine
  - ❑ Programm als Feld `Instruction[]`
- ❑ Konstruktor mit Erzeugung der initialen Konfiguration
- ❑ Weitere Methoden
  - ❑ Setzen des Aktuellen Programms
  - ❑ Lesen der aktuellen Konfiguration
  - ❑ Ausführen des Programms
  - ❑ **main-Methode!**



```
public class Machine {  
    private Configuration config = null;  
    private Instruction[] program = null;  
  
    public Machine() {  
        config = new Configuration();  
    }  
    public void setProgram(Instruction[] prog) {  
        program = prog;  
    }  
    ...  
}
```



```
...  
public void run() {  
    while (! program[config.getICounter()].  
            toString().equals("END"))  
        program[config.getICounter()].  
            eval(config);  
}  
...  
}
```



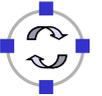
```
...  
public static void main(String[] args) {  
    Instruction[] prog = {  
        new Load(1), new Div(2), new Mult(2),  
        new Store(3), new Load(1), new Sub(3),  
        new Store(3), new End()  
    };  
    Machine machine = new Machine();  
    machine.setProgram(prog);  
    machine.getConfiguration().setRegister(1, 32);  
    machine.getConfiguration().setRegister(2, 5);  
    machine.run();  
}  
}
```



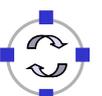
- ❑ Illustration der Funktionsweise von Markov-Tafeln durch Java-Applikation mit Grafischer Nutzungsoberfläche (GUI)
- ❑ Copyright: Ingolf Geist, Uni Magdeburg
  
- ❑ Siehe auch im Buch Saake/Sattler Abschnitt 6.5.1



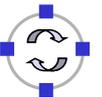
Nr	Phi	Psi	Nächste Regel	Nächste Regel (n. erfolgreich)
0		*#	1	-1
1 #0		0#	2	3
2 *		0*	1	-1
3 #1		1#	4	5
4 *		1*	1	-1
5 #			6	-1



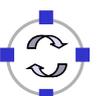
- ❑ Weiteres Strukturierungsmittel für Quelltexte
- ❑ "Kooperierende Klassen" in ein Package
- ❑ Zielsetzungen: Verbesserung der Lesbarkeit durch
  - ❑ Trennung von verschiedenen inhaltlichen Aspekten
  - ❑ Trennung von Applikationslogik und Nutzungsschnittstelle



1. `algdat.demo.program`:  
allgemeine Schnittstellen für Markov-Tafeln und  
-Algorithmen, aber auch nutzbar für Implementierung von  
Registermaschinen → angelehnt an Modell abstrakter  
Maschinen
2. `algdat.demo.markov`:  
Implementierung der Funktionsweise von Markov-Tafeln  
und -Algorithmen
3. `algdat.demo.gui`:  
Implementierung der Nutzerinteraktion

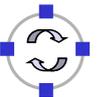


<b>Klasse</b>	<b>Dargestellter Aspekt</b>
<code>MarkovRule</code>	Basisklasse für Regel in Markov-Tafel oder -Algorithmus
<code>MarkovAlgorithmRule</code>	Klasse für Regel in Markov- Algorithmus
<code>MarkovTableRule</code>	Klasse für Regel in Markov-Tafel
<code>MarkovTableProgram</code>	Klasse für Programm einer Markov- Tafel (Menge von Regeln)
<code>MarkovTableMachine ...</code>	Klasse als Ausführungskomponente



- ❑ In Hauptprogramm `algdat.demo.AlgorithmDemo`
  - ❑ Initialisierung
  - ❑ Laden der Programme aus XML-Datei
  - ❑ Laden der System-Properties
  - ❑ Starten der GUI durch

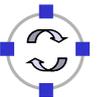
```
AlgorithmDemoMainFrame mainFrame =  
    new AlgorithmDemoMainFrame (programs)
```



```
...  
import javax.swing.*;  
...  
public class AlgorithmDemoMainFrame  
    extends JFrame {  
  
    private javax.swing.JPanel jContentPane = null;  
    private JToolBar algorithmsToolBar = null;  
    private JSplitPane jSplitPane = null;  
    private JScrollPane jScrollPane = null;  
    ...  
}
```



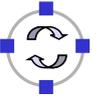
- ❑ Programmiersprache definiert Syntax und Semantik von Befehlen
- ❑ Also, beliebige Formatierung von
  - ❑ Quelltext?
  - ❑ Programmblöcken?
  - ❑ Klammern?
  - ❑ Variablen-, Methoden-, Klassennamen?
- ❑ Aber: allgemein anerkannte "gute Sitten" für Programmierstil
- ❑ Offizielle Festlegung von Sun:  
<http://java.sun.com/docs/codeconv/>



- ❑ 80% des Aufwands (Kosten!!!) für Software durch Pflege und Wartung von bereits existierendem Code
- ❑ In realen Projekten arbeiten meist viele Entwickler an einer (komplexen) Software
- ❑ Pflege und Wartung im Laufe langer Lebenszeit durch viele Autoren  
→ **Lesbarkeit** des Codes ist ein Hauptkriterium für Erfolg



```
public
class helloWrong {
public
static void main(String[] Argumente)
{
String s; s = "Hello World!"; helloWrong.
SCHREIBE(s);
}
public
static void SCHREIBE(String Der_Text)
{System.out.println(Der_Text);} }
```



```
/*
 * Name der Klasse
 * Infos zu Version, Datum, Copyrights etc.
 */

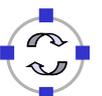
package algdat.myPackage; // zuerst Package-Name

import java.util.Vector; // dann Imports

class MyClass { // dann die Klasse

    ...

}
```

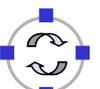


```
class MyClass {  
  
    public static int statischeVariable;  
  
    public ...; // erst statische Variablen  
    protected ...; // dann Objektvariablen  
    package ...; // in dieser Reihenfolge  
    private ...;  
    public MyClass() { // Konstruktoren zuerst  
    }  
  
    public methode() { // dann Methoden, main zuletzt  
    }  
}
```



- ❑ Einrückungen
  - ❑ Empfohlen 4 Leerzeichen oder 1 Tab (wenn genug Platz ist...)
  - ❑ Für jeden Scope, d. h. alles zwischen { und } bzw. nach **if**, **for**, etc.
- ❑ Zeilenumbruch generell nach {

```
public meineMethode(int a) {  
    if (a > 7) {  
        ...;  
    }  
}
```

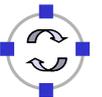


- Bei zu langen Zeilen nach Kommas

```
public methodeMitVielenParametern(int a,  
    String name, float b) { ...
```

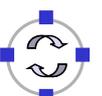
vor Operatoren

```
String meinString = "Zusammengesetzt aus"  
    + "einigen anderen"  
    + "Strings";
```



- Eine pro Zeile empfohlen (Kommentare möglich)
- Wenn möglich bei Definition initialisieren
- Immer am Blockanfang, danach eine Leerzeile

```
while (true) {  
    int meineVariable = 7;    // Glückszahl  
    int andereVariable = 13; // Pechzahl  
  
    if (meineVariable < andereVariable) ...
```



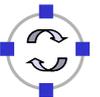
- Ein Statement pro Zeile empfohlen

```
a++; b--; c = a+b; // Unerwünscht
```

- Allerdings: Ergeben mehrere Befehle einen Sinnzusammenhang, so kann es manchmal angeraten sein, das durch Zusammenfassung in einer Zeile deutlich zu machen

- Klammern bei **if**, **for**, **try** etc. immer empfohlen

```
if (e <= 0) {  
    System.out.println("Fertig!");  
} else {  
    System.out.println("Weiter ...");  
}
```

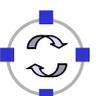


- Packages: alles klein, gegebenenfalls URL rückwärts

```
package algdat.mypackage;  
package com.microsoft.billspackage;
```

- Klassen und Interfaces: erster Buchstabe groß, bei internen Wortanfängen Großbuchstaben

```
class DasIstMeineKlasse { ...
```

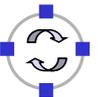


- ❑ Variablen und Methoden: erster Buchstabe klein, bei internen Wortanfängen Großbuchstaben

```
int c;  
String meinName = "Bond";  
  
public aendereNamen(String neu) { ...
```

- ❑ Konstanten: alles groß, interne Worttrennung durch Unterstrich

```
static final int MAX_SEMESTERZAHL = 14
```



- ❑ Das wichtigste zuerst:  
*„Don't document your code – if it was hard to write, it should be hard to read!“*
- ❑ Wenn sich Kommentare nicht vermeiden lassen:
  - ❑ Lüge in den Kommentaren – das muss kein aktives Lügen sein, oft reicht es, die Kommentare bei Änderungen nicht zu aktualisieren.
  - ❑ Peppe den Code mit Kommentaren wie `/* addiere 1 zu i */` auf. Aber: Kommentiere niemals die interessanten Dinge, wie z.B. der eigentliche Zweck eines Moduls oder einer Methode.
  - ❑ Kommentiere niemals eine Variable! Alle Informationen darüber, wofür und wie die Variable verwendet wird, ihre Wertebereiche, Maßeinheit, Ausgabeformat, wann ihrem Wert vertraut werden kann etc. sollten dem Code entnommen werden.



- Weitere Anregungen:
  - Stelle sicher, dass jede Methode etwas mehr (oder weniger) macht, als ihr Name vermuten lässt. Beispiel: die Methode `isValid(x)` könnte als Seiteneffekt die Variable `x` in Binärform konvertieren und in einer Datenbank speichern.
  - Verwende möglichst viele Abkürzungen um den Code kurz und bündig zu halten. Echte Programmierer definieren niemals Abkürzungen, sie verstehen sie aufgrund ihrer genetischen Veranlagung.
  - Cd wrttn wtht vwls s mch trsr (Code written without vowels is much terser).
  - Vermeide im Interesse der Effizienz grundsätzlich die Kapselung. Aufrufer einer Methode sollen alle verfügbaren Details der Methode benötigen, um Bewusstsein dafür zu schaffen, wie die Methode intern funktioniert.
  - Zur Steigerung der Effizienz: Nutze `cut/paste/clone/modify` ausgiebig, da es viel schneller geht, als viele kleine wiederverwendbare Module zu verwenden.
  - Erzeuge möglichst verwirrende Einrückungen für den Code



- Weitere Anregungen:
  - Versuche, soviel wie möglich in eine Zeile zu quetschen:
    - Das spart den Mehraufwand durch temporäre Variablen und macht die Quelltexte kleiner, da unnötige Zeilenumbrüche und Leerzeichen vermieden werden.
    - Gute Programmierer schaffen es oft an die Grenze der maximalen Zeilenlänge von 255 Zeichen mancher Editoren zu stoßen. Der klare Vorteil ist, dass jemand, der 6-Punkt-Schrift nicht lesen kann, gezwungen ist zu scrollen, um den Code zu lesen.
  - Sei kreativ bei Variablennamen:
    - Benenne unterschiedliche Variablen möglichst mit ähnlich lautenden Bezeichnern, Abkürzungen etc.
    - Beispiel: `Colours`, `Colors`, `clrs`, `kulerz` (“dude-speak”)
    - Verwende lange Variablennamen, die sich nur durch ein Zeichen oder Groß-Kleinschreibung unterscheiden: “`swimmer`” und “`swimner`”.
    - Nutze dabei ähnlich aussehende Zeichen aus: „l“, „I“ 1“ (`1`, `I`, `1`)



- ❑ Objektorientierung als Programmierparadigma
  - ❑ Strukturierung des Codes nach zusammenhängenden Klassen von Realweltobjekten
  - ❑ Algorithmen zu Daten
  
- ❑ Beispiele: Registermaschine und Markovtafeln
  - ❑ Anwendung der Objektorientierung am Beispiel
  - ❑ Trennung von Darstellung und Anwendungslogik
  
- ❑ Java Code Conventions zur Gewährleistung der Lesbarkeit von Programmquelltexten
  
- ❑ Literatur: Saake/Sattler: *Algorithmen und Datenstrukturen*, Kapitel 12

