

# Softwaretechnik

Klaus Knopper

(C) 2009 Klaus.Knopper@fh-kl.de

## Organisatorisches

- ⇒ Vorlesung wöchentlich jeweils Mittwochs 14:00-15:30 Uhr
- ⇒ Übungen (Einteilung nach Gruppe) auch Mittwochs

 <http://knopper.net/bw/swt/>

Folie 1

## Zusammenfassung (1)

Nach den mathematischen und technischen Grundlagen der Informatik, und dem ersten Einblick in die Funktionsweise formaler Sprachen (Programmiersprachen) liegt der Schwerpunkt in „Softwaretechnik“ in der praktischen Programmierung. Hier werden die in vielen Programmiersprachen üblichen Konstrukte wiederholt und vertieft, und lösungsorientierte Ansätze implementiert.

Als „Lernsprache“ kommt im Teil 1 (Klaus Knopper) von Softwaretechnik, wie auch schon in „Grundlagen der Informatik“, größtenteils Java zum Einsatz, wobei jetzt vor allem die objektorientierten „Spezialitäten“ ausgeschöpft werden.

Folie 2

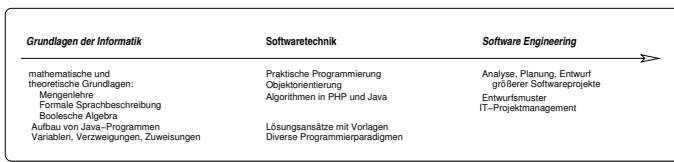
## Zusammenfassung (2)

Kursziel ist die Fähigkeit, für ein gegebenes Problem die „am besten“ passende programmertechnische Lösung mit den erlernten Hilfsmitteln selbst erstellen zu können.

Neben PHP/HTML im Teil 2 (Andreas Heß) für die praktische Web-Programmierung kommen auch weitere Programmiersprachen (z.B. Perl, Python, Bash) zum Einsatz, die die Grundlagen anderer Programmierparadigmen und -konzepte zeigen, die aber für diesen Kurs nicht im Detail beherrscht werden müssen.

Folie 3

## Einordnung „Softwaretechnik“



Folie 4

## Wiederholung: JAVA

Java ist eine **objektorientierte Programmiersprache**, die komplexe Zusammenhänge und Strukturen anhand von **Klassen** und **Objekten** modelliert, und dem Programmierer damit die Abstraktion einer Aufgabenstellung erleichtert und die Übersichtlichkeit auch großer Programme fördert. Durch Kapselung von Funktionen und Vererbung wird die Entwicklung von **Programmierschnittstellen** und **Wiederverwendbarkeit** besonders gut unterstützt, was für die Implementation von „großen“ Anwendungen (Middleware, GUIs auf dem Desktop) sehr hilfreich sein kann.

Folie 5

## Grundsätzliches zu Java

- ⇒ Der Java-Compiler (`javac`) erzeugt üblicherweise keinen selbstständig lauffähigen Maschinencode<sup>a</sup>, sondern einen sogenannten *Bytecode*, zu dessen Ausführung die Java Virtuelle Maschine (JVM bzw JRE) benötigt wird.
- ⇒ Java-Bytecode gibt es in einer „abgespeckten“ Form, die in einem Browser (mit Java-Support) lauffähig ist („Applet“-Klasse), sowie in der klassischen Form als Objektdatei zur Ausführung mit der JVM.

---

a

Folie 6

## Vorteile von Java-Programmen

- ⇒ Architektur (Rechner-) unabhängig, zur Ausführung wird lediglich eine auf der gewählten Rechnerplattform installierte und lauffähige Java Virtuelle Maschine benötigt.
- ⇒ Große Menge an vordefinierten Klassen und Methoden.
- ⇒ „Baukasten“-Prinzip.
- ⇒ Leistungsfähige Entwicklungsumgebungen (z.B. `eclipse`) verfügbar.

Folie 7

## Nachteile von Java

- ⇒ Langsam, da nur interpretiert und nicht direkt als Maschinencode ausgeführt.
- ⇒ Es ist eine Virtuelle Maschine notwendig, die zur Programmversion „passen“ muss.
- ⇒ Inkompatibilitäten durch mangelhafte Versionierung von Klassenbibliotheken und ggf. veraltete virtuelle Maschinen.
- ⇒ Wird schnell unübersichtlich ohne Entwicklungsumgebung.
- ⇒ Zwar sind viele Klassenmethoden „selbsterklärend“ benannt, jedoch ist der Quelltext gegenüber beispielsweise C oft unverhältnismäßig umfangreich bei gleicher Funktionalität.

Folie 8

## Beispiel: „Hello, World!“

```
public class hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Achtung: Die JVM führt standardmäßig die Klasse aus, die so heißt wie die übersetzte Datei. Der Quelltext muss hier also `hello.java` heißen, und der Bytecode wird nach der Übersetzung (`javac hello.java` erzeugt `hello.class`) entsprechend mit `java hello` ausgeführt!

Folie 9

## Ein Applet

„Java ist einfach.“ (?)

```
import java.applet.*;
import java.awt.*;
public class Text extends Applet {
    String hello = "Hello World";
    public void paint(Graphics g) {
        g.drawString(hello, 5, 25);
    }
}
```

Dieses Applet kann, in HTML-Seiten eingebettet per **SCRIPT**-Tag, vom Browser ausgeführt ein Fenster mit dem Textinhalt „Hello, World!“ öffnen, eine funktionierende JVM als Browser-Plugin vorausgesetzt.

Folie 10

## Java-Schlüsselwörter

Die folgenden Schlüsselwörter sind Bestandteil der Sprache Java, und dürfen nicht als Bezeichner für Variablen verwendet werden:

abstract	boolean	break	byte
case	char	class	const
continue	default	do	double
else	extends	finally	float
for	goto	if	implements
import	instanceof	int	long
native	new	package	private
protected	return	short	super
switch	synchronized	this	throws
transient	try	void	while

Folie 11

## Variablendeklarationen in Java

Im Gegensatz zu C können in Java Variablen auch zwischen Anweisungen bzw. Methodenaufrufen deklariert werden, z.B. erst unmittelbar vor ihrer ersten Benutzung.

```
public class VarDecl {
    public static void main(String[] args) {
        int a;
        a = 1;
        char b = 'x';
        System.out.println(a);
        double c = 3.1415;
        System.out.println(b); System.out.println(c);
        boolean d = false; System.out.println(d);
        for(int i = 0; i < 10; i++)
            System.out.println(i);
    }
}
```

Folie 12

## Klassen vs. Dateien

Es ist zwar möglich, Klassen innerhalb anderer Klassen zu definieren, üblicherweise werden Klassen aber in einzelnen Dateien gespeichert, wobei der Dateiname dem Klassennamen (plus Endung `.java`) entspricht, unter Beachtung von Groß- und Kleinschreibung.

Wird nun in einer Klasse auf Methoden oder Variablen einer anderen Klasse zugegriffen, so versucht die Java VM die jeweilige Klasse aus einer Datei `KlassenName.class` nachzuladen, d.h. jede in einem Java-Programm verwendete Klasse darf (und wird üblicherweise auch) in einer separaten Datei gespeichert und übersetzt werden.

Der „Zusammenbau“ des Programms und die Evaluation der Klassenmethoden und Variablen erfolgt zur Laufzeit, sofern nicht schon der Compiler die Informationen über eine Klasse benötigt, von der die korrekte Übersetzung des anderen Klasse abhängt. In diesem Fall muss der Quelltext für die jeweiligen Klassen in der richtigen Reihenfolge übersetzt werden.

Folie 13

## Klassenmethoden (1)

In jeder Klasse lassen sich neben Variablen auch *Methoden* (vergl. C: globale Funktionen) definieren, die nur im Zusammenhang mit einem Objekt dieser Klasse, oder einem expliziten Methodenaufruf aus der Klasse (solange dort nicht auf Objektelemente zugegriffen wird), aufrufbar sind.

```
public class Auto {
    public String name;
    public int    erstzulassung;
    public int    leistung;
    public int    alter(int jahr) {
        return jahr - erstzulassung;
    }
}
...
Auto a = new Auto();
a.erstzulassung = 2004;
System.out.println( a.alter(2008) );
```

Folie 14

## Klassenmethoden (2)

Als **static** deklarierte Methoden dürfen auch direkt innerhalb einer Klasse aufgerufen werden, ohne dass erst ein Objekt dieser Klasse erzeugt werden muss.

```
public class Klasse {
    public static void hallo() {
        System.out.println("Ich bin Klasse!");
    }
}
...
Klasse.hallo();
```

Folie 15

## Sichtbarkeit von Objektattributen in Java

Mit dem Schlüsselwort **private** vor einer Variablen innerhalb einer Klasse wird dafür gesorgt, dass diese Variable nur für die Klasse selbst und deren Klassenmethoden sichtbar ist. **public** sorgt hingegen dafür, dass auch Objekte von anderen Klassen Zugriff auf die Variablen bekommen.

Es gilt als guter Stil, klasseninterne Variablen stets als **private** zu deklarieren (Voreinstellung), und **public**-Methoden zu definieren, die einen kontrollierten Zugriff auf die privaten Variablen von außen erlauben.

```
public class Pizza {
    private int scharf;
    public boolean getPfefferMenge() { return scharf; }
    public void setPfefferMenge(int wieviel) {
        scharf = wieviel;
    }
}
```

Folie 16

## static-Variablen

Als **static** deklarierte Variablen existieren, ähnlich wie bei C, vom ersten Laden der Klasse bis zum Programmende, und können entsprechend von mehreren Instanzen/Objekten einer Klasse gemeinsam benutzt werden, werden also nicht für jede Instanz neu erzeugt. Manche von der Java VM und der Java Klassenbibliothek vorgegebene Variablen oder Methoden müssen als **static** definiert werden, z.B. das schon bekannte `public static void main()` als automatisch aufgerufene Methode.

Folie 17

## Deklaration von Arrays

Arrays in Java sind Objekte, das bedeutet auch, dass Arrays Methoden und Instanz-Variablen besitzen können. Für die Deklaration muss zuerst eine Array-Variablen deklariert werden und anschließend das erzeugte Array der Variablen zugewiesen werden.

```
int[] a; // Deklaration Array-Variablen a
// Zuweisung eines Arrays mit 5 int-Elementen
a = new int[5];
```

Die Deklaration und Initialisierung kann auch in einem Schritt durchgeführt werden.

```
int[] a = new int[5];
int[] a = { 1, 2, 3, 4, 5}; // Initialisierung
```

Folie 18

## Zugriff auf Arrays

Der Zugriff auf ein Array-Element erfolgt, wie in C, über seinen Index, beginnend mit 0.

```
public static void main(String[] args)
{
    int[] zahl = new int[2];
    zahl[0] = 2;
    zahl[1] = 3;
    System.out.println("zahl hat " + zahl.length +
        " Elemente.");
    System.out.println(zahl[0]);
    System.out.println(zahl[1]);
}
}
```

Folie 19

## Die **String**-Klasse (1)

In der Java-Klasse **String** sind im Gegensatz zum aus C bekannten **char \*** auch Methoden definiert, die es erlauben, mit aus der Arithmetik bekannten Operatorzeichen Zeichenketten zusammenzuhängen, oder Umwandlungen zwischen Text und Zahlen vorzunehmen.

```
public class StringVerkett
{
    public static void main(String[] args)
    {
        int a = 5;
        double x = 3.14;

        System.out.println("a = " + a);
        System.out.println("x = " + x);
    }
}
```

Folie 20

## Die String-Klasse (2)

```
public class StringVergleich
{
    public static void main(String[] args)
    {
        String a = new String("hallo");
        String b = new String("hallo");
        System.out.println("a == b liefert " + (a == b));
        System.out.println("a != b liefert " + (a != b));
    }
}
```

Folie 21

## Die String-Klasse (3)

```
public class StringVergleich2
{
    public static void main(String[] args)
    {
        String a = new String("hallo");
        String b = new String("hallo");
        System.out.println("a.equals(b) liefert " +
                           a.equals(b));
    }
}
```

Folie 22

## Integeroperationen - Beispiel

Die Standard-Klasse **Integer** in Java stellt einige praktische Methoden zur Verfügung, die mit dem Basistyp **int** nicht so einfach zu realisieren sind.

```
public class IntOut0 {
    public static void main (String args[]) {
        int k=987654321;
        System.out.println(k + " zur Basis 10 ist " +
                           Integer.toString(k));
        System.out.println(k + " zur Basis 2 ist " +
                           Integer.toBinaryString(k));
        System.out.println(k + " zur Basis 8 ist " +
                           Integer.toOctalString(k));
        System.out.println(k + " zur Basis 16 ist " +
                           Integer.toHexString(k));
        System.out.println(k + " zur Basis 3 ist " +
                           Integer.toString(k,3));
    }
}
```

Folie 23

## Objekterzeugung in Java

In Java sind Objektvariablen zunächst *Referenzen* (ähnlich wie Zeiger in C), denen entweder ein bereits existierendes Objekt (bzw. dessen Referenz) zugewiesen werden kann, oder ein *neues* Objekt, was mit dem Java-Schlüsselwort `new` und dem Aufruf des *Konstruktors* erzeugt wird.

```
public class Test {  
  
    public class NeuesObjekt {  
        public int zahl;  
    }  
  
    public static void main(String[] args) {  
        NeuesObjekt n = new NeuesObjekt();  
        n.zahl = 1;  
    }  
}
```

Folie 24

## Konstruktor

Jede Klasse enthält, unsichtbar, (mindestens) eine Methode, die so heißt wie die Klasse selbst, und die beim Erzeugen eines Objektes dieser Klasse automatisch aufgerufen wird. Man kann diese Methode selbst (über-)schreiben, und mit Parametern versehen:

```
public class Auto  
{  
    public String name;  
    public int    erstzulassung;  
    public int    leistung;  
    public Auto(String name) { this.name = name; }  
}
```

Später kann dann ein Objekt der Klasse `Auto` mit `Auto a = new Auto("Flitzer");` erzeugt werden, wodurch die Variable `name` mit dem String "Flitzer" vorbelegt wird. `this` ist stets eine Referenz auf das aktuelle Objekt, und kann normalerweise entfallen (allerdings nicht in diesem Beispiel, warum?).

Folie 25

## Destruktor

Jede Klasse enthält, unsichtbar, eine Methode, die nach Beendigung der Lebenszeit eines Objektes dieser Klasse automatisch von der sog. „Garbage Collection“ der Java VM aufgerufen wird. Es kann allerdings passieren, dass dies erst zum Ende des gesamten Programms passiert, und daher die Anweisungen im Destruktor gar nicht zur Ausführung gelangen.

```
public class Test {  
    public class KonstruktorTest {  
        KonstruktorTest() {  
            System.out.println("Konstruktor wurde aufgerufen!");  
        }  
        protected void finalize() {  
            System.out.println("Destruktor wurde aufgerufen!");  
        }  
    }  
    public static void main(String[] args) {  
        KonstruktorTest t1 = new KonstruktorTest();  
    }  
}
```

Folie 26

## Assoziationen zwischen Klassen

Ein Objekt **k1** der Klasse **Kunde** greift auf die Methode **abbuchen()** der Klasse **Konto** zu.

```
class Kunde
{
    Konto k1;
    void geld_abheben() { k1.abbuchen(); }
}

class Konto
{
    public void abbuchen() { ... };
}
```

Folie 27

## Aggregation

Ein Auto besteht aus 4 Rädern und einem Motor.

```
class Auto
{
    Rad vr, vl, hr, hl;
    Motor m;

    void starte()
    {
        m.ein();
    }
}
```

Die Klassen **Rad** und **Motor** müssen natürlich noch entsprechend definiert werden.

Folie 28

## Paradigmen

Ein Paradigma bezeichnet einen *Kernsatz*, der einen Sachverhalt beschreibt oder modelliert. Üblicherweise wird in einem Paradigma in einem Satz zusammengefasst, worum es geht.

In der Programmierung bezeichnet ein Paradigma die Herangehensweise an ein Problem. Für unterschiedliche Probleme sind unterschiedliche Lösungsansätze optimal, und dementsprechend gibt es für jede Aufgabe auch eine „am besten passende“ Programmiersprache.

Folie 29

## Imperatives Programmierparadigma

„Befehle werden ausgeführt.“

Beispiel in der Shell (bash):

```
read -p "Bitte x eingeben: " x
let xx=x*x
echo "x zum Quadrat ist" $xx
```

Folie 30

## Prozedurales Programmierparadigma

„Oft verwendete Befehlsfolgen werden zu Prozeduren und Funktionen zusammengefasst.“

Beispiel in C:

```
int quadrat(int x) { return x*x; }
...
int c2 = quadrat(a) + quadrat(b);
```

Folie 31

## Funktionales Programmierparadigma

„Die Lösung steckt in der Definition.“

Beispiel in Haskell:

```
fib :: Integer -> Integer
fib(0) = 0
fib(1) = 1
fib(n+2) = fib(n) + fib(n+1)
Main> fib(30)
```

Folie 32

## Objektorientiertes Programmierparadigma

„Objekte beschreiben konkrete Dinge durch Eigenschaften (Attribute) und Methoden.“

Beispiel in JAVA:

```
public class Auto extends Fahrzeug {
    Motor m;
    Räder[4] r;
    ...
    public boolean start_motor();
    public void tanken(Benzinart b);
    ...
}
```

Folie 33

## Vererbung von Klasseigenschaften

Ein Kaffeemaschine hat eine Funktion zur Befüllung von Tassen. Eine spezielle Kaffeemaschine Cafe2000 hat zusätzlich einen Timer.

```
class Kaffeemaschine
{
    int tasse;
    public void befüllung(int wieviel) {
        tasse = wieviel;
    }
}
class Cafe2000 extends Kaffeemaschine
{
    Time t;
    public void timer(Time time) {
        t = time;
    }
}
```

Folie 34

## Polymorphie

```
class PTier {
    public String farbe;
    String kennung() {
        return "Keine Ahnung was ich mal werde.";
    }
}
class PPinguin extends PTier {
    String kennung() { return "Ich bin ein Pinguin."; }
}
...
PPinguin pingu = new PPinguin();
System.out.println( pingu.kennung() );
```

Frage: Was würde ausgegeben werden für die Fälle

PTier pingu = new PTier(); oder

PTier pingu = new PPinguin();?

Folie 35

## abstract Klassen und Methoden

Mit dem Java-Attribut **abstract** werden solche Klassen oder Methoden gekennzeichnet, die nur an abgeleitete Klassen vererbt, aber nicht direkt in Objekten implementiert werden können.

```
public abstract class Pizza {
    ...
}

// Pizza p = new Pizza(); ist NICHT ERLAUBT!!!

public class Speziale extends Pizza { // OK
    ...
}
```

Folie 36

## Warum abstract?

- ⇒ Vermeidet „unpräzise“ Objekte, die direkt von einer nicht-konkreten (eben „abstrakten“) Klasse abgeleitet werden.
- ⇒ Zwingt bzw. erinnert den Programmierer, dass er die entsprechenden Klasseneigenschaften bzw. Methoden (erst) in den abgeleiteten Klassen/Klassenmethoden im Detail implementieren muss.
- ⇒ Hilft, „ähnliche“ Objekte zu definieren, die aber nicht zueinander kompatibel sind.

Folie 37

## interface-Klassen

„Eine abstrakte Klasse, die nur abstrakte Methoden/Funktionen enthält.“

```
public interface music_handler {
    Musikstück[] liste();
    Musikstück runterladen(URL u, String name);
    Musikstück runterladen(Gerät g, String name);
    void abspielen(Musikstück m);
    void übertragen(Musikstück m, Gerät g);
    void löschen(Musikstück m);
}

public class MusicPlayer
    implements music_handler {
    ...
}
```

Folie 38

## Warum **interface**-Klassen?

- ⇒ Ähnlich **abstract**, erfordern eine konkrete Implementierung in den abgeleiteten Klassen.
- ⇒ Bieten einen Ausweg aus der fehlenden „Mehrfachvererbung“, d.h. der Tatsache, dass in Java eine Unterklasse nur von einer Basisklasse erben kann. Mehrfache, durch Komma getrennte Interfaces hinter **implements** sind aber möglich.
- ⇒ Spezifizieren eher das **Verhalten** (Methoden + Aktionen) von Klassen als gemeinsame **Eigenschaften**.

Folie 39

## Fehler und Ausnahmebehandlungen

In Java lassen sich „ungewöhnliche“ Ereignisse wie die vorzeitige Beendigung von Tastatureingaben, Fehler beim Lesen und Schreiben von Dateien etc. mit Hilfe sogenannter „Exceptions“ abbilden, und unter Verwendung von **try ... catch** Fehlerbehandlungsmethoden definieren. Man kann jedoch auch durch gezieltes „Werfen“ von Fehlersignalen die Standardfehlerbehandlung der Java VM auslösen, ähnlich wie **exit (Fehlercode)**; bei C dem aufrufenden Programm einen Fehler nebst Programmabbruch mitteilt.

```
public class Ausnahmen
{
    public static void main(String[] args)
    {
        if (args.length < 2) {
            throw new IllegalArgumentException();
        }
        ...
    }
}
```

Folie 40

## Texteingaben mit Java

Das Einlesen von Variablen ist in Java, verglichen mit C, verhältnismäßig kompliziert, da für die verschiedenen Eingabemethoden zunächst Eingabeobjekte erzeugt werden müssen, und Fehler bei der Eingabe mit **try...catch**-Konstruktionen (sog. Ausnahmebehandlungen) abgefangen werden müssen.

Es bietet sich an, für häufig verwendete Einleseoperationen eigene Klassen und entsprechende Klassenmethoden anzulegen, die später von anderen Klassen einfach aufgerufen werden können.

Folie 41

## Textein- und ausgabe - Beispiel #1

Textein- und ausgabe über die Kommandozeile:

```
import java.io.*; // io-Klassen laden
public class TextEinAusgabe {
    public static void main(String[] args) {
        System.out.println("Text eingeben:");
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in) );
            String s = in.readLine();
            System.out.println("Der Text lautet: " + s);
        } catch( IOException ex ) {
            System.out.println( ex.getMessage() );
        }
    }
}
```

Folie 42

## Textein- und ausgabe - Beispiel #2a

Textein- und ausgabe als grafische Applikation:

```
import java.awt.*;
import java.awt.event.*;
public class TextEinAusgabe extends Frame
{
    TextField eingabe;
    Label    ausgabe;
    public static void main( String[] args ) {
        TextEinAusgabe meinFenster = new TextEinAusgabe("Eingabe");
        meinFenster.setSize(400, 200);
        meinFenster.show();
    }
    public TextEinAusgabe( String fensterTitel ) {
        super(fensterTitel);
        Label hinweis = new Label( "Text eingeben");
        eingabe = new TextField();
        ausgabe = new Label();
    }
}
```

Folie 43

## Textein- und ausgabe - Beispiel #2b

```
add( BorderLayout.NORTH,  eingabe );
add( BorderLayout.CENTER, hinweis );
add( BorderLayout.SOUTH,  ausgabe );
eingabe.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent ev ) {
            meineMethode(); } } );
addWindowListener(
    new WindowAdapter() {
        public void windowClosing( WindowEvent ev ) {
            dispose();
            System.exit( 0 ); } } );
}
void meineMethode() {
    ausgabe.setText( "Der eingelesene Text lautet: " +
                    eingabe.getText() );
}
}
```

Folie 44

## Textein- und ausgabe - Beispiel #3a

Text- und Ausgabe als Applet, eingebettet in eine HTML-Seite:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class TextEinAusgabe extends Applet
{
    TextField eingabe;
    Label      ausgabe;
    public void init() {
        Label hinweis = new Label( "Oben Text eingeben" );
        eingabe = new TextField( "      " );
        ausgabe = new Label();
        setLayout( new BorderLayout() );
        add( BorderLayout.NORTH, eingabe );
        add( BorderLayout.CENTER, hinweis );
        add( BorderLayout.SOUTH, ausgabe );
    }
}
```

Folie 45

## Textein- und ausgabe - Beispiel #3b

```
    eingabe.addActionListener(
        new ActionListener() {
            public void actionPerformed( ActionEvent ev ) {
                meineMethode(); } } );
}
void meineMethode() {
    ausgabe.setText( "Der Text lautet: " +
                    eingabe.getText() );
}
}
```

Folie 46

## Beispiel: Systemeigenschaften anzeigen

```
public class SystemProperties
{
    public static void main( String[] args )
    {
        System.out.println(
            System.getProperties().toString()
                .replace( ',', '\n' ).replace( '{', ' ' )
                .replace( '}', ' ' ) );
    }
}
```

NB: Jedes **replace()** liefert hier einen **String** zurück, in dem wieder die Methode **replace()** aufgerufen werden kann.

Folie 47

## Aus aktuellem Anlass...

ein kurzer Abschnitt über rechtliche Aspekte in der Softwaretechnik.

Folie 48

## Rechtliche Aspekte von Software / Lizenzen

- ⇒ Urheberrecht
- ⇒ Überlassungsmodelle (Lizenzen)
  - ⇒ Verkauf (selten)
  - ⇒ Nutzung / Miete (entgeltlich oder unentgeltlich)
  - ⇒ Open Source / Freie Software (weitgehende Übertragung der Verwertungsrechte auf den Lizenznehmer)
- ⇒ Patente (?)

Folie 49

## Proprietäre Software

- ⇒ Der Empfänger erwirbt mit dem Kauf eine eingeschränkte, i.d.R. nicht übertragbare *Nutzungslizenz*.
- ⇒ Der Empfänger darf die Software nicht analysieren („disassemble“-Ausschlussklausel).
- ⇒ Der Empfänger darf die Software nicht verändern.
- ⇒ Der Empfänger darf die Software nicht weitergeben oder weiterverkaufen.

Diese Restriktionen werden im Softwarebereich so breit akzeptiert, dass man fast schon von einem „traditionellen“ Modell sprechen kann.

Folie 50

## „Freie Software“

- ⇒ Freie Software stellt Software als Resource/Pool zur Verfügung.
- ⇒ Freie Software sichert dem Anwender (Benutzer und Programmierer) bestimmte Freiheiten.
- ⇒ Freie Software stellt eine Basis (Lizenz) für eine Zusammenarbeit von Gruppen (oder Firmen) zur Verfügung.

Folie 51

## Was ist Freie Software/Open-Source?

- ⇒ Open-Source (engl. = offene Quelle)
- ⇒ Freie Software (FSF, 1984) ist Teilmenge von Open-Source-Software.
- ⇒ Open-Source ist kein Produkt, sondern
- ⇒ eine *Methode*, um Software zu entwickeln.
- ⇒ Open-Source-Definition lt. **OSI**.
- ⇒ „Frei“ steht für **Freiheit** (ff.), nicht für „kostenfrei“!

Folie 52

## Die GNU General Public License

gibt den *Empfängern* der Software das Recht, ohne Nutzungsgebühren

- ⇒ die Software für alle Zwecke einzusetzen,
- ⇒ die Software (mit Hilfe der Quelltexte) zu analysieren,
- ⇒ die Software (mit Hilfe der Quelltexte) zu modifizieren,
- ⇒ die Software in beliebiger Anzahl zu kopieren,
- ⇒ die Software im Original oder in einer modifizierten Version weiterzugeben oder zu verkaufen, auch kommerziell, wobei die neuen Empfänger der Software diese ebenfalls unter den Konditionen der **GPL** erhalten.

<http://www.gnu.org/>

Folie 53

## Die GNU General Public License

- ⇒ zwingt NICHT zur Veröffentlichung/Herausgabe von Programm oder Quellcode,
- ⇒ zwingt NICHT zur Offenlegung ALLER Software oder Geschäftsgeheimnisse,
- ⇒ verbietet NICHT die kommerzielle Nutzung oder den Verkauf der Software,
- ⇒ verbietet NICHT die parallele Nutzung, oder lose Kopplung mit proprietärer Software.

Folie 54

## Die GNU General Public License

Aber: Alle EMPFÄNGER der Software erhalten mit der GPL die gleichen Rechte an der Software, die die Mitentwickler, Distributoren und Reseller ursprünglich hatten (und weiterhin behalten).



Folie 55

Wer legt die Lizenz fest?

**Der Urheber.**



Folie 56

## Für wen gilt eine Lizenz?

Eine Lizenz gilt für die in der Lizenz angegebenen Personenkreise (sofern nach landesspezifischen Gesetzen zulässig).

Beispiel: Die GNU GENERAL PUBLIC LICENSE gilt für

- ⇒ alle legalen EMPFÄNGER der Software, die
- ⇒ die Lizenz AKZEPTIERT haben.



Folie 57

## „Wer liest schon Lizenzen?“

- ⇒ Zumindest in Deutschland bedeutet das FEHLEN eines gültigen Lizenzvertrages, dass die Software NICHT ERWORBEN und NICHT EINGESETZT werden darf.
- ⇒ In Deutschland gibt es seit der letzten Änderung des Urheberrechtes keine generelle Lizenz-Befreiung mehr.
- ⇒ Wurde die Lizenz nicht gelesen, oder „nicht verstanden“ (weil z.B. nicht in der Landessprache des Empfängers vorhanden), so ist die rechtliche Bindung, und daraus resultierend, die Nutzungsmöglichkeit der Software, formal nicht gegeben.

Auch als „Freeware“ deklarierte Software ist hier keine Ausnahme. Wenn keine Lizenz beiliegt, die eine bestimmte Nutzungsart ausdrücklich ERLAUBT, gilt sie als VERBOTEN.

Folie 58

## „Kopierschutz“ (1)

- ⇒ Soll die nicht vom Rechteinhaber genehmigte Vervielfältigung unterbinden,
- ⇒ ist de facto technisch überhaupt nicht realisierbar,\*)
- ⇒ ein „wirksamer“ Kopierschutz (juristisch genügt die Angabe auf der Packung, technisch kann der Kopierschutz absolut wirkungslos sein) darf nach der Urheberrechtsnovelle von 2003 nicht mehr umgangen werden, auch nicht zum Anfertigen einer Kopie für den Privatgebrauch,

...

\*) Alles, was audiovisuell wahrgenommen werden kann, kann auch kopiert werden, notfalls über die „analoge Lücke“.

Folie 59

## „Kopierschutz“ (2)

- ⇒ dennoch bleibt das Umgehen eines Kopierschutzes zur ausschließlichen Eigennutzung nach §108b UrhG aber straffrei,
- ⇒ und laut §69a Abs. 5 UrhG ist das Umgehen einer Kopiersperre speziell bei Computerprogrammen auch nicht in jedem Falle ein Strafdelikt (VORSICHT!).

Folie 60

## Fragwürdige Lizenzklauseln

- ⇒ Genereller „Haftungsausschluss“,
- ⇒ Eigentumsvorbehalt,
- ⇒ Konkurrenzausschluss,
- ⇒ Abtreten von gesetzlich garantierten Grundrechten.



Folie 61

## Autor/Distributor haften...

- ⇒ für „Geschenke“ nur bei GROBER FAHRLÄSSIGKEIT,
- ⇒ für „Verkäufe“ bei allen vom Verkäufer/Hersteller verschuldeten Fehlern.



Folie 62

## GPL-Verträglichkeit

- ⇒ GPL erlaubt die Integration proprietärer Software auf dem gleichen Datenträger, solange die nicht-GPL-Komponenten wieder separierbar sind (Beispiel: KNOPPIX-CD, versch. Linux-Distributionen).
- ⇒ BSD-Lizenz erlaubt die Integration von Code in proprietäre Programme ohne Offenlegungspflicht. Es muss lediglich darauf hingewiesen werden, dass die Software BSD-Komponenten enthält (Beispiel: TCP/IP-Stack im Windows-Betriebssystem).
- ⇒ Die Programm-Urheber können für ihr Werk auch eine Auswahl verschiedener Lizenzen „zum Ausschuchen“ anbieten (Dual Licensing).

Folie 63

## Tabelle: Lizenzmodelle und Rechte

	Nutzung kostenlos	frei kopierbar	zeitlich unbegrenzt nutzbar	Quelltext wird mitgeliefert	Modifikation erlaubt	Einbau in prop. Produkte erlaubt	Derivate mit anderen Lizenzen mögl.
proprietäre Software							
Shareware	✓	✓					
Freeware	✓	✓	✓				
GPL	✓	✓	✓	✓	✓		
LGPL	✓	✓	✓	✓	✓	✓	
BSD	✓	✓	✓	✓	✓	✓	✓

Folie 64

## Creative Commons

- ⇒ Verschärfung des Urheberrechtes zugunsten der Content-Industrie führt zu Ablehnung durch viele Kreative.
- ⇒ Schaffung von rechtlichen Grundlagen zur Eigenvermarktung und Eigenverlag von Kunstwerken durch die Künstler ohne Exklusivvertrag mit einer Verwertungsgesellschaft.
- ⇒ „Lizenz-Baukasten“ für verschiedene Empfängerkreise und Verwertungszwecke.

Prominentes Beispiel: „[Elephants Dream](#)“.

Folie 65

## ...zurück zu Java.

Weitere, juristische und andere nicht-technische Aspekte der Softwaretechnik folgen in der Vorlesung „Software-Engineering“.

Folie 66

## Arrays vs. Listen

Nachteile von Arrays:

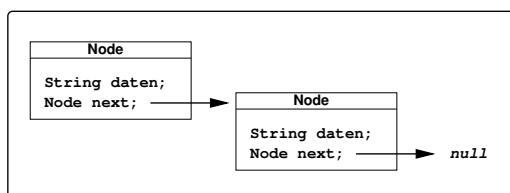
- ⇒ Konstante Anzahl Elemente nach Initialisierung,
- ⇒ „Einfügen“ von Elementen an bestimmter Stelle schwierig, da nachfolgende Elemente nicht ohne weiteres „verschoben“ werden können,
- ⇒ „Löschen“ von Elementen schwierig, da nachfolgende Elemente nicht „aufrücken“ können.

Folie 67

## Lösung: „Verketteten“ von Objekten

```
public class Node {  
    String daten;  
    Node next; // Zeiger auf NÄCHSTES Element  
}
```

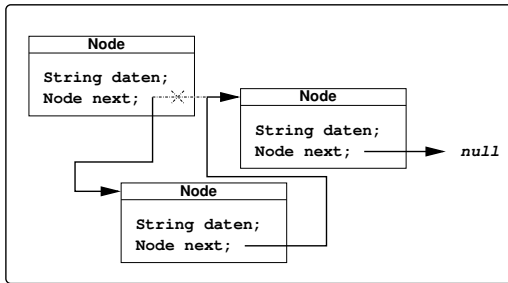
```
Node n = new Node();  
n.next = new Node();  
n.next.next = null;
```



Folie 68

## Hinzufügen eines Elements

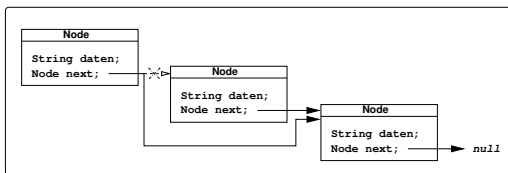
```
Node neu = new Node();  
neu.next = vorgaenger.next;  
vorgaenger.next = neu;
```



Folie 69

## Entfernen eines Elements

```
vorgaenger.next = vorgaenger.next.next;
```



Folie 70

## Suchen eines Elements

Da es nicht wie bei einem Array einen „Laufindex“ `array[i]` gibt, muss den `next`-Zeigern bis zum Ende der Liste gefolgt werden.

Beispiel: Stringsuche

```
Node n = listenanfang;  
while(n != null) {  
    if(n.data.equals("Suchbegriff")) break; // gefunden!  
    n = n.next;  
}
```

`n` ist nach der Schleife eine Referenz auf das gesuchte Element, oder `null`.

Folie 71

## Syntax (Grundlagen Wdh.)

Die **Syntax einer Programmiersprache** kann mit Hilfe von Regeln beschrieben werden. Diese Regeln können sowohl

- ⇒ umgangssprachlich oder
- ⇒ mit Hilfe einer formalen Methode

beschrieben werden. Bei mathematischen Definitionen und anderen exakten Festlegungen wird eher die formale Methode bevorzugt.

Folie 72

## Syntax

Formale Methoden, um eine Syntax zu beschreiben, sind:

- ⇒ **Syntaxdiagramme**,
- ⇒ **Formale Sprachen** (z.B. **Chomsky Typ 0-3 Grammatik**).

Wie ist beispielsweise die Syntax aller natürlichen Zahlen definiert?

Folie 73

## Grammatiken und Programmiersprachen

- ⇒ Jede **Programmiersprache** besitzt eine Grammatik.
- ⇒ Mit der Grammatik kann die Syntax von Programmen geprüft werden (Bestandteil des Compilers).
- ⇒ Mit Hilfe der Grammatik kann eine Programmiersprache „syntaktisch“ verstanden werden.

Folie 74

## Semantik

Die **Semantik** beschreibt schlicht und einfach die *Bedeutung* einer Darstellung.

Beispiele:

Formale Darstellung	Semantik
<b>A == B</b>	Es wird geprüft, ob <b>A</b> und <b>B</b> gleich sind.
<b>C = A - B</b>	Der Variablen <b>C</b> wird das Ergebnis der Subtraktion <b>A - B</b> zugewiesen.
<b>A a = new A()</b>	Der Referenz <b>a</b> wird ein neues Objekt vom Typ der Klasse <b>A</b> zugewiesen.

Folie 75

## Syntax und Semantik

Während *syntaktische* Fehler vom Compiler aufgrund algorithmischer Regeln präzise gefunden werden können, sind *semantische* Fehler erst aus dem Zusammenhang zwischen erwartetem und tatsächlichem Ergebnis zu erkennen, vor allem deswegen, weil sie oft auf Missverständnissen oder falscher Interpretation von vorgegebenen Algorithmen und Anwendungsbeschreibungen durch den Programmierer beruhen.

D.h.: Nicht jedes *syntaktisch* einwandfreie (also vom Compiler übersetzbare) Programm ist automatisch fehlerfrei!

Folie 76

## Grafik unter Java - die **Graphics()**-Klasse

Viele grundlegende Grafikfunktionen von Java sind in der **Graphics()**-Klasse untergebracht, die zum Paket **java.awt** gehört.

Die **Graphics()**-Klasse umfasst u.a. Methoden zum Zeichnen von Linien, Kreisen, Ellipsen, Rechtecken und Texten in verschiedenen Farben und Schriften.

Folie 77

## Interaktive Elemente in Java-Applets

... Beispiele in der Vorlesung und in Übung 9.

Folie 78

## Grundgerüst eines Applets

```
// Importieren der notwendigen awt-Klassen
public [Klassenname] extends java.applet.Applet {
// Variablen-Deklaration und Definition von Methoden
public void init() { ... } // beim Laden des Applets
public void start() { ... } // beim Start des Applets
public void stop() { ... } // wenn HTML-Seite mit Applet
// verlassen wird
public void destroy() { ... } // Löschen des Applets und
// Freigeben der Ressourcen
public void paint(Graphics g) { ... } // wird aufgerufen,
// wenn Applet gezeichnet wird
public void run() { ... } // bei Multithreading anstelle
// von paint()
...
}
```

Folie 79

## Die `drawline()`-Methode

Mit dieser Methode kann man eine einfache Linien zwischen zwei vorgegebenen Koordinatenpunkten zeichnen.

```
import java.awt.Graphics;
public class ZieheLinie extends java.applet.Applet {
public void paint(Graphics g) {
// Zeichnen einer Linie zwischen Anfangspunkt
// (42,42) und Endpunkt (100,100)
g.drawLine(42,42,100,100);
}
}
```

Folie 80

## Die `drawRect ()`-Methode

Für das Zeichnen von Rechtecken kann die `drawRect ()`-Methode genutzt werden. Als Parameter werden die Koordinaten des oberen linken Punktes, die Breite und die Höhe übergeben.

```
import java.awt.Graphics;
public class MaleRec extends java.applet.Applet {
    public void paint(Graphics g) {
        // Rechteck mit links oberer Koordinate (150,100),
        // einer Breite von 200 und Höhe von 120 Pixeln
        g.drawRect(150, 100, 200, 120);
    }
}
```

Folie 81

## Beispiele zur `Graphics ()`-Klasse #1

gefülltes Rechteck mit links oberer Koordinate (10,100), einer Breite von 200 Pixeln und einer Höhe von 42 Pixeln.

```
import java.awt.Graphics;
public class MaleRecFil extends java.applet.Applet {
    public void paint(Graphics g) {
        g.fillRect(10, 100, 200, 42);
    }
}
```

Folie 82

## Beispiele zur `Graphics ()`-Klasse #2

Zeichnet ein dreidimensionales Rechteck. Der 5. Parameter ist ein boolescher Wert, der angibt, ob das Rechteck als erhöht dargestellt werden soll.

```
import java.awt.Graphics;
public class MaleRec3D extends java.applet.Applet {
    public void paint(Graphics g) {
        g.draw3DRect(10, 100, 200, 42, true);
    }
}
```

Folie 83

## Beispiele zur **Graphics ()**-Klasse #3

Zeichnet ähnlich wie **drawRect ()** ein Rechteck, aber mit abgerundeten Ecken. Die beiden ersten Parameter geben den Winkel der Abrundung in der horizontalen und der vertikalen Ebene an.

```
import java.awt.Graphics;
public class MaleRecRound extends java.applet.Applet {
    public void paint(Graphics g) {
        g.drawRoundRect(10, 100, 50, 42, 5, 5);
        g.drawRoundRect(110, 200, 100, 42, 5, 50);
        g.drawRoundRect(150, 250, 25, 42, 50, 25);
    }
}
```

Folie 84

## Beispiel: Polygonzug

```
import java.awt.Graphics;
import java.awt.Polygon;
public class ZeichnePolygon2 extends java.applet.Applet {
    // Definiere ein Array mit X-Koordinaten
    int[] xCoords = { 10, 40, 60, 300, 10, 30, 88 };
    // Definiere ein Array mit Y-Koordinaten
    int[] yCoords = { 20, 0, 10, 60, 40, 121, 42 };
    // Bestimme Anzahl Ecken über Methode length
    // des Array-Objekts mit x-Koordinaten
    int anzahlEcken = xCoords.length;
    public void paint(Graphics g) {
        Polygon poly = new Polygon(xCoords, yCoords, anzahlEcken);
        // Zeichne ein 7-Eck
        g.drawPolygon(poly);
    }
}
```

Folie 85

## Beispiel: Zeichnen von Kreisen und Ellipsen

```
import java.awt.Graphics;
public class ZeichneOval extends java.applet.Applet {
    public void paint(Graphics g) {
        // Zeichne eine Ellipse
        g.drawOval(50, 100, 200, 120);
        // Zeichne einen Kreis
        g.drawOval(175, 175, 200, 200);
    }
}
```

Folie 86

## Farben setzen mit `setColor()`

```
import java.awt.*;
public class Farbe extends java.applet.Applet {
    public void paint(Graphics g) {
        // Erzeugen von Farbobjekten
        Color pinkColor = new Color(255, 192, 192);
        Color irgendeineFarbe1 = new Color(255, 100, 92);
        Color irgendeineFarbe2 = new Color(5, 12, 120);
        // Zeichnen von Ellipsen
        g.setColor(irgendeineFarbe2);
        g.drawOval(5, 5, 150, 250);
        // Zeichne einen Kreis
        g.setColor(pinkColor);
        g.fillOval(250, 150, 150, 150);
        // direkte Verwendung einer Farbkonstanten
        g.setColor(Color.green);
        g.drawRect(40,40,100,200);
    }
}
```

Folie 87

## Beispiel `setBackground()`

Mit `setBackground(Color c)` und `setForeground(Color c)` kann man die Hintergrund- und Vordergrundfarbe setzen.

```
public void paint(Graphics g) {
    /* Hier wird auf einem rosa Hintergrund */
    /* ein grünes Rechteck gezeichnet*/
    Color pinkColor = new Color(255, 192, 192);
    setBackground(pinkColor);
    g.setColor(Color.green);
    g.drawRect(40,40,100,20);
}
```

Folie 88

## Erstellen von Fontobjekten

```
import java.awt.Font;
import java.awt.Graphics;
public class FontTest extends java.applet.Applet {
    public void paint(Graphics g) {
        Font f = new Font("TimesRoman", Font.PLAIN, 18);
        Font fb = new Font("TimesRoman", Font.BOLD, 18);
        Font f2i = new Font("Arial", Font.ITALIC, 34);
        g.setFont(f);
        g.drawString("Normaler (plain) Font - TimesRoman", 10, 25);
        g.setFont(fb);
        g.drawString("Fetter (bold) Font - TimesRoman", 10, 50);
    }
}
```

Folie 89

## Bilder laden und anzeigen

```
import java.awt.Image;
import java.awt.Graphics;
public class DrawImage2 extends java.applet.Applet {
    Image samImage;
    Image bild;
    public void init() {
        // Bild laden - ein jpg-File
        samImage = getImage(getDocumentBase(), "bild.jpg");
    }
    public void paint(Graphics g) {
        g.drawImage(samImage, 0, 0, this);
    }
}
```

Folie 90

## Animationen unter Java

```
import java.awt.Image;
import java.awt.Graphics;
public class Animation1 extends java.applet.Applet {
    Image bild;
    public void init() {
        bild = getImage(getCodeBase(), "bild.gif");
        resize(600, 200);
    }
    public void paint(Graphics g) {
        for (int i=0; i < 1000; i++) {
            int bildbreite = bild.getWidth(this);
            int bildhoehe = bild.getHeight(this);
            int xpos = 10; // Startposition X
            int ypos = 10; // Startposition Y
            g.drawImage(bild, (int)(xpos + (i/2)),
                (int)(ypos + (i/10)), (int)(bildbreite * (1 +
                (i/1000))), (int)(bildhoehe * (1 + (1/1000))),
                this); } } }
```

Folie 91

## Beispielapplikation: Rotes Rechteck #1

```
import java.awt.*;
import java.awt.event.*;
class MaleRechtFill extends Frame {
    public MaleRechtFill() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                dispose();
                System.exit(0);
            }
        });
    }
}
```

Folie 92


## Beispielapplikation: Rotes Rechteck #2

```
public static void main(String args[]) {
    MaleRechtFill mainFrame = new MaleRechtFill();
    mainFrame.setSize(400, 400);
    mainFrame.setTitle("Test");
    mainFrame.setVisible(true);
}
public void paint(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(50, 70, 200, 100);
}
}
```

Folie 93

## GUI-Programmierung

Bei der Einführung von interaktiven Elementen muss einiges beachtet werden:

1. Der Programmablauf ist nicht mehr linear, weil man nicht weiß, wann der Anwender auf welches Element und in welcher Reihenfolge klickt.
2. Es soll auf jedes Bedienelement individuell reagiert werden  Callbacks.
3. Für die Reaktion auf Ereignisse („Events“) existieren in JAVA sog. (Event-),„Listener“.
4. Entweder einzelne Events individuell registrieren (**new ...**), oder alle auf einmal definieren und ggf. einige leer lassen (**implements**).

Folie 94

## GUI-Programmierung

```
import java.awt.*;
import java.awt.event.*;
public class HelloButton {
    public static void main(String[] args) {
        Frame frame = new Frame("Mein Frame"); // Ein Fenster
        frame.setSize(400,200); // Größe ändern (Breite, Höhe)
        Button button = new Button("Klick hier!");
        button.addMouseListener( new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                System.out.println("Knopf wurde gedrückt.");
                System.exit(0);
            }
        });
        frame.add(button);
        frame.setVisible(true);
    }
}
```

Folie 95

## GUI-Programmierung: Bedienelemente

Die Bedienelemente werden mit `add(element)` mit dem jeweiligen Fenster (**Frame**) verbunden und benachrichtigen den zugehörigen Event-Handler.

Klasse	Erklärung	Event-Handler
<b>Button</b>	Schaltfläche (Klick)	<b>ActionListener</b>
<b>TextField</b>	Einzeiliges Texteingabefeld	<b>ActionListener</b>
<b>TextArea</b>	Mehrzeiliges Texteingabefeld	<b>ActionListener</b>
<b>Choice</b>	Aufklapp-Menü	<b>ItemListener</b>
<b>Checkbox [Group]</b>	Auswahl[Radio]-Kästchen	<b>ItemListener</b>
<b>MenuBar</b>	Fenster-Menü	<b>ItemListener</b>
<b>Frame</b>	Fenster allgemein	<b>WindowListener,</b> <b>MouseListener,</b> <b>MouseMotionListener</b>

Folie 96

## Rekursion

*Rekursiv* heißt eine Funktion, die sich selbst aufruft.

Beispiel: Die Fakultäts-Funktion

Math.:

$$F(x) = x \cdot (x-1) \cdot (x-2) \cdot \dots \cdot 1$$
$$= \begin{cases} 1 & \text{falls } x \leq 1 \text{ (Rekursionsanfang)} \\ x \cdot F(x-1) & \text{sonst (Rekursionsschritt)} \end{cases}$$

JAVA:

```
public int f(int x) {
    if(x <= 1) { return 1; } // Abbruchbedingung
    else      { return x * f(x-1); }
}
```

Folie 97

## Rekursion

Übung: Aufaddieren der Zahlen von 1 ... x ohne `for()`-Schleife.

```
public int reihe(int x) {
    if(          ) return          ;
    return x + reihe(          );
}
```

Folie 98