Hauptseminar Automatisierung:

Grundlagen der Java-Programmierung

Jörg Rüdenauer

- Proprietäre Sprache von sun, keine Standardisierung
- Entwicklung von Java seit 1991
- aktuelle Version ist Java 2 v 1.3.1
- die Syntax ist stark an C++ angelehnt
- Vorteile:
  - robust
  - portabel
  - sicher
  - wenig fehleranfällig
  - einfach
- Nachteile: langsam, nicht echtzeitfähig, kein direkter Hardwarezugriff

- Benutzung von Unicode für Kommentare, Identifier, Strings, Character
  `\u0042, ö, é, ß, ...`
- ansonsten ASCII
- zwei Arten Kommentare:
  - // bis Zeilenende: `// dies ist ein Kommentar`
  - /* bis */, nicht verschachtelt
    ```
    /* ein Kommentar, der mehr als eine Zeile Platz
    benötigt */
    ```
- Identifier (Namen):
  - (fast) nur Buchstaben, Ziffern, Unterstrich
  - Beginn nicht mit Ziffer
  - Groß- / Kleinschreibung wird unterschieden
    `alpha, t_3, _n3, i, MAX_VALUE`
    nicht: `3f, x!, t;`

Syntax: Literale

- Zeichenketten (String): in Anführungszeichen:
  `“Hallo, Welt!”, “du” + “ich”, “\””`
- Zeichen (char): in einfachen Anführungszeichen:
  `‘j’, ‘\u00A2’, ‘\n’, ‘\\’, ‘\’’, ‘ß’`
- Ganze Zahlen (int): `123, 0, 2001`
  - mit führender 0: oktal `0237, 072, nicht: 081`
  - mit führendem 0x: hexadezimal `0x2FF, 0x23`
  - mit folgendem L: long `124L, 0x37CL`
- Gleitkommazahlen (double): `0.123, 9.109E-31`
  - mit folgendem f: float `13.82f, 1.24E23f`
- Boolsche Werte (boolean): `true, false`
- Referenztypen: `null`

Syntax: Operatoren

- Auf Zahlen: ++, --, +, - (unär und binär), *, /, %
  `i++, -12, 13 % 5, 12.3 - d`
- Auf Zahlen, bitweise: ~, &, ^, <<, >>, >>>
- Auf boolean: !, &, |, &&, ||
- Auf allen: ==, !=, <, <=, >, >=
- Zuweisung: = sowie +=, -=, %=, <<= etc.
  `x += 7` entspricht `x = x + 7`
- Außerdem:
  - Cast: (*Typ*) `Kanzler k = (Kanzler)schroeder;`
  - Bedingungsoperator: *boolAusdruck* ? *Ausdruck1* : *Ausdruck2*
    `x = (x < 0) ? x : -x;`
  - instanceof

Syntax: Blöcke und Variablen

- Blöcke: Beginn mit {, Ende mit }
- statt Anweisung kann überall auch ein Block stehen
- Variablendeklaration kann überall im Block erfolgen
- Form: *Typname Identifier* [= *Ausdruck*][, *Identifier* [= *Ausdruck*]]*
  ```
  int i=0; double x, y; Embryo probe = new Mensch();
  ```
- Konstanten werden mit Schlüsselwort final versehen
  ```
  final int MEISTER = BAYERN_MUENCHEN;
  ```
- Gültigkeitsbereich: Von der Deklaration bis zum Ende des Blocks, in dem die Deklaration steht

Syntax: Verzweigungen

- Einfache Verzweigung: if (*Ausdruck*) *Anweisung* [else *Anweisung*]
  ```
  if (sterbehilfeErlaubt)
    toete(patient);
  else {
    beantrageGelder();
    pflege(patient);
  }
  ```
- Mehrfachverzweigung - nur auf Integer-Datentypen
  ```
  byte studiengang = ...; String s;
  switch(studiengang) {
    case SOFTWARETECHNIK: s = “Super!”; break;
    case INFORMATIK: s = “sehr gut!”; break;
    case BETRIEBSWIRTSCHAFTSLEHRE: s = “...”; break;
    ...
    default: s = “unbekannt”;
  }
  ```

Syntax: Schleifen (1)

- Schleife mit Eingangsprüfung: while (*Ausdruck*) *Anweisung*
```
while (antwort != 42) antwort = frage();
```
- Schleife mit Ausgangsprüfung: do *Anweisung* while (*Ausdruck*);
```
do {
  steuereFlugzeug();
} while (true);
```
- Zählschleife: for (*Init*; *Ausdruck*; *Änderung*) *Anweisung*
  - Init: Liste von Anweisungen, durch Kommata getrennt oder Variablendeklaration - gültig nur in Zählschleife
  - solange *Ausdruck* true ergibt, wird erst *Anweisung*, dann *Änderung* ausgeführt
```
for (int i=0; i<20; i++) {
  a[i] = i;
  b[i] = a[i] + 1;
}
```


Syntax: Schleifen (2)

- Mit break wird die aktuelle Schleife verlassen
- Mit continue wird zum Ende des Schleifenrumpfes gesprungen
- Schleifen können auch Namen (label) haben,
  break / continue *label* bezieht sich dann auf die genannte Schleife

```
int zielX, zielY;
outerLoop:
for (int i=0; i < a.length; i++) {
  for(int j=0; j < a[i].length; j++) {
    if (a[i][j] == suche) {
      zielX := i; zielY := j;
      break outerLoop;
    }
  }
}
```

Pakete (1)

- Paket (package) definiert einen eigenen Namensraum
- Paket beinhaltet Klassen, Interfaces, Unterpakete
  aber keine 2 Elemente mit dem gleichen Namen
- Typen aus Paketen werden eingebunden mit
  - import *paketname*.*; oder
  - import *paketname*.*Typname*; dann kann *Typname* selber noch verwendet werden (verdeckt den aus dem Paket, dieser kann immer noch über den vollen Namen angesprochen werden)
- Paket java.lang wird automatisch eingebunden
- Paketzugehörigkeit wird ganz am Anfang einer Datei deklariert mit package *Name;* z.B. `package berlin.xml.parser;`

Pakete (2)

- Fehlt die package - Deklaration, wird die Datei einem anonymen default-Paket zugeordnet
- Nur öffentliche (public) Typen sind ausserhalb des Paketes, also in anderen Paketen, sichtbar
- Jede Datei darf nur höchstens einen öffentlichen Typ beinhalten
- Die Datei muss dann *Typname*.java heissen
- Die hierarchische Struktur der Pakete muss sich im Dateisystem wiederspiegeln
- z.B. müssen die Dateien des Paketes de.uni-stuttgart.isw.berlin im Verzeichnis de/uni-stuttgart/isw/berlin liegen
- Paketnamen sollten (im Prinzip) global eindeutig sein

Objektorientierung: Konzept

- Objektorientierte Programmierung ist das einzige von Java unterstützte Programmierparadigma
- Alle modernen Konzepte der Objektorientierung werden von Java unterstützt:
  - Pakete
  - Klassen
  - Methoden
  - Attribute
  - Vererbung
  - Polymorphie
  - ...
- Jeglicher Code ist Bestandteil einer Klasse

Objektorientierung: Klassen

- Klassen sind die einzigen selbstdefinierten Typen in Java
- Deklaration:
  [modifier] class *Name* [extends *Name*] [implements *Name*[,Name]*] {...}
- mögliche Modifier (auch mehrere):
  - public: Klasse ausserhalb des eigenen Paketes sichtbar
  - abstract: Abstrakte Klasse, keine Instanzen möglich
    (<=> Klasse hat mindestens eine abstrakte Methode)
  - final: Keine Unterklassen von dieser Klasse erlaubt
    (nicht gleichzeitig mit abstract)
- extends, implements: später
- Im Rumpf zwischen { und }: Definition von Membern, Konstruktoren, Initialisierern

- Member sind Attribute, Methoden und innere Klassen / Interfaces
- Modifier von Membern:
  - static: Member gehört zur Klasse, nicht zu einzelnen Instanzen
  - Sichtbarkeit:
    - private: Nur für Klasse selbst (und innere Klassen) sichtbar
    - default (ohne Modifier): zusätzlich im eigenen Paket sichtbar
    - protected: zusätzlich für Unterklassen sichtbar
    - public: überall sichtbar

- In Java-Terminologie fields (Felder)
- Deklaration wie bei Variablen, aber mit Modifiern
- Gültigkeit gesamte Klasse (auch schon vor der Deklaration)
- Mögliche Modifier:
  - Modifier von Membern
  - final: Konstante, nur einmalige Initialisierung erlaubt
  - transient: Wird bei Serialisierung nicht beachtet
  - volatile: Synchronisierung bei Zugriff durch mehrere Threads
- Beispiele:

```
public static final int BLUE=0x0000FF;
protected Point start;
int x, y;
private transient double radius;
```

Objektorientierung: Methoden (1)

- Deklaration:
  [Modifier] *ResultType Name*([Parameter]*) [throws ...] { ... }
- Mögliche Modifier:
  - Modifier von Membern
  - abstract: abstrakte Methode ohne Implementierung, Body wird durch ‘;’ ersetzt
  - final: Methode kann in Unterklassen nicht überschrieben werden
  - native: Methode ist nicht in Java implementiert (kein Body)
  - synchronized: für Threads, s. später
- ResultType gibt an, welchen Typ das von der Methode zurückgegebene Resultat hat

Objektorientierung: Methoden (2)

- Gibt die Methode nichts zurück, ist der ResultType void
- Parameter: Deklaration wie lokale Variablen (aber ohne Initialisierung), durch Kommata getrennt
- Einziger Übergabemechanismus call-by-value (s. später)
- Gültigkeitsbereich gesamte Methode
- Signatur einer Methode besteht aus Name und Parametern
- Mehrere Methoden gleicher Signatur nicht erlaubt
  - insbesondere nicht 2 Methoden, die sich nur im ResultType unterscheiden
  - erlaubt hingegen mehrere Methoden gleichen Namens, aber unterschiedlicher Parameter (Overloading)

- Ist der ResultType nicht void, muss die Methode durch die Anweisung return *Ausdruck*; (oder eine Exception) verlassen werden, wobei *Ausdruck* den Typ ResultType hat
- Sonst kann die Methode durch return; verlassen werden, wobei der Ausdruck fehlen muss
- Beispiele für Methoden:

```
private abstract void printName();
static int getObjectCounter(final int type) { ... }
public synchronized void waehle(int stimme)
  throws UngueltigeWahlException { ... }
public synchronized void waehle
  (int erststimme, int zweistimme) throws ...
```


- In Java nur Einfachvererbung (*aber:* Interfaces)
- Oberklasse wird mit extends angegeben:
  `public class Softwaretechniker extends Informatiker`
- Fehlt extends, ist automatisch java.lang.Object die Oberklasse
- Vererbt werden alle sichtbaren Member
- Bei Redefinition von Attributen werden die Attribute der Oberklasse verdeckt
- Methoden dagegen werden überschrieben. Dabei darf die neue Methode nicht weniger Sichtbarkeit definieren, mehr Exceptions werfen oder den ResultType ändern
- Expliziter Zugriff auf Member der Oberklasse immer noch per super: `return super.x;`

Objektorientierung: Initialisierung (1)

- Beim Anlegen einer neuen Instanz wird ein Konstruktor aufgerufen
- Deklaration von Konstruktoren wie Methoden, aber:
  - Name muss der gleiche sein wie der der Klasse
  - kein ResultType
  - Modifier nur public, protected oder private
- Im Body kann die erste Anweisung einen Konstruktor der Oberklasse aufrufen per `super([Parameter]);` oder einen anderen der eigenen Klasse per `this([Parameter]);`
- Fehlt dieser Aufruf, wird implizit ein `super();` hinzugefügt

Objektorientierung: Initialisierung (2)

- Wird kein Konstruktor definiert, hat die Klasse automatisch einen default-Konstruktor ohne Parameter, der nur `super()` aufruft
- Es gibt auch Initialisierer, die keine Konstruktoren sind: reine Blöcke Code (`{...}`). Diese werden verwendet für:
  - anonyme Klassen (s. später)
  - statische Elemente: `static { ... }`
- Destruktor: Beim Löschen einer Instanz wird automatisch die Methode `public void finalize()` aufgerufen, die in Object definiert ist
- aber kein automatischer Aufruf von Oberklassendestruktor
- auch keine automatische Default-Implementierung von finalize

- Klassen können nicht nur auf oberster Ebene definiert werden
- Definition als Member: innere Klassen, haben Zugriff auf alle Attribute der umschließenden Klasse (auch private)
- Definition im Code: lokale Klassen
  - dürfen keine statischen Elemente haben
  - dürfen keine Sichtbarkeits-Modifier haben
- Anonyme Klassen: erzeugt durch new *Name*(...) { *class body* };
  - Unterklasse von *Name* (bzw. implementiert *Name*)
  - nie abstract, nie static, immer final, wie innere Klasse
  - kein expliziter Konstruktor möglich
  - Verwendung hauptsächlich bei der GUI - Programmierung

- Ermöglichen über die implements - Klausel eine Art Mehrfachvererbung
- Deklaration:
  [Modifier] interface *Name* [extends *Interface* [, *Interface*]*] { ... }
- automatisch abstrakt
- in fast jeder Hinsicht wie Klassen, aber:
  - können nur von Interfaces erben (aber von beliebig vielen)
  - alle Attribute sind automatisch public static final
  - alle Methoden sind automatisch public abstract
  - alle inneren Klassen sind automatisch public static
  - keine statischen Methoden erlaubt

Objektorientierung: Instanzen

- Instanzen von Klassen werden per new *Konstruktoraufruf* angelegt: `Politiker schroeder = new Kanzler();`
- Zugriff auf Member dann mit *Variable.Membername* (falls sichtbar): `schroeder.rede();`
- Bei Methoden- oder Konstruktoraufruf muss für jeden Parameter ein Ausdruck des entsprechenden Typs angegeben werden
- Expliziter Zugriff auf eigene Instanz mit this:
  `this.x = x; // x z.B. Parameter oder lokale Variable`
- Löschen von Instanzen geschieht automatisch (s. später)
- Zugriff auf statische Elemente auch mit *Klassenname.Elementname*: `Counter.getObjectCount(POINTS);`

Objektorientierung: Beispiel (1)


```
import java.util.*;
public class LinkedList { // verkettete Liste
  public interface Linkable { // Member-Interface
    Linkable getNext(); // implizit public abstract
    void setNext(Linkable node);
  }
  protected Linkable head; // Listenkopf
  public void append(Linkable node) { .... }
  public void remove(Linkable node) { .... }
  public Enumeration elements() {
    return new Enumerator(); // Instanz anlegen
  }
  // continued...
```

```
  protected class Enumerator implements Enumeration {
    private Linkable current;
    public Enumerator() { current = head; } // Konstr.
    public boolean hasMoreElements {
      return current != null;
    }
    public Object nextElement() {
       if (current != null) {
          Object value = current;
          current = current.getNext();
          return value;
       }
       else .... // Fehlerbehandlung
 }}} // Methode, Enumerator, LinkedList
```



```
import java.awt.print.*;

public class PrintableList extends LinkedList
  implements Printable {

  public int print(Graphics graphics,
                   PageFormat pageFormat,
                   int pageIndex) {
    ...
  }
}
```

Exceptions: Ausnahmen

- Fehlerbehandlung in Java durch Ausnahmen
- Voll und von Anfang an in die Sprache integriert
- Alle Ausnahmen sind Unterklassen von java.lang.Throwable
  => Objekte wie alle anderen auch
- Eigene Ausnahmen können definiert werden, in dem man die Klasse von Throwable oder (meistens) von Exception ableitet
- Ausnahmen kann man explizit auslösen (werfen) durch throw
  dabei meist Objekterzeugung: `throw new MyException();`
- Außerdem werden eine Reihe von Ausnahmen automatisch geworfen: OutOfMemoryError, NullPointerException, ClassCastException, ArrayIndexOutOfBoundsException, ArithmeticException, ...

Exceptions: Syntax (1)

- Eine Methode muss in der throws-Klausel die Ausnahmen deklarieren, die sie werfen (oder von unten hochreichen) kann:

```
public void pray(God god, String prayer)
  throws Blasphemy, NoSuchGodException { ... }
```

- Nicht deklariert werden müssen:
  - Unterklassen von Error (...Error): sollten zum sofortigen Programmabbruch führen
  - Unterklassen von RuntimeException wie z.B. ArrayIndexOutOfBoundsException
  - also im Prinzip diejenigen Ausnahmen, die durch Java automatisch geworfen werden

Exceptions: Syntax (2)

- Konstrukt zur Behandlung von Ausnahmen:
  try {
    // Block, in dem Ausnahmen auftreten können
  }
  catch (*Exception Bezeichner)* {
    // Ausnahmenbehandlung für *Exception* oder Unterklassen
  }
  finally {
    // Block, der auf jeden Fall ausgeführt wird
  }
- Auch mehrere catch - Blöcke möglich (erster passender wird genommen)
- Entweder catch oder finally können auch weggelassen werden

Exceptions: Beispiel


```
public void doSomeDB() throws MyException {
try {
  connection = DriverManager.getConnection(....);
  try {  doDBTransactions(); connection.commit(); }
  catch (SQLException e) {
    try {connection.rollback(); }
    catch (SQLException e2) {}
    finally { throw new MyException("..."); }
  }
}
catch (SQLException e) { throw new MyException("..."); }
finally {
  try { if (connection != null) connection.close(); }
  catch (SQLException e) {}
}}
```

Arrays (1)

- Arrays werden behandelt wie Objekte:
  - Anlage der Arrays auf dem Heap
  - können Variablen des Typs Object zugewiesen werden
  - haben Methoden, die in der Klasse Object definiert sind
- Typ eines Arrays ist der Typ der Elemente, gefolgt von []
- Arrays immer eindimensional, aber Arrays von Arrays:
  `double[][] matrix; // 2-dim. Array aus doubles`
- Bei mehreren Dimensionen kann die Länge der Unterarrays unterschiedlich sein
- Zugriff auf Elemente über []*:* `matrix[i][j]`
- Länge kann ermittelt werden durch *Variable.*length
- Index läuft immer von 0 bis length - 1

Arrays (2)

- Anlegen von Arrays:
  - mit new: `matrix = new double[3][4];` von hinten her kann Länge auch weggelassen werden (aber nicht in 1. Dim.)
  - durch Array - Initialisierer:
    ```
    String[][] ampeln = { {“rot”, “gruen”},
    {“schwarz”, “gelb”}, {“rot”, “gelb”, “gruen”}};
    ```
- Bei Arrays können autom. Exceptions ausgelöst werden: ArrayIndexOutOfBoundsException, NegativeArraySizeException, NullPointerException
- Arrays können mit clone() kopiert werden:
  `String[][] mehrAmpeln = ampeln.clone(); //tiefe Kopie`

- Threads werden von Java direkt unterstützt
- Threads sind abgeleitet von Klasse java.lang.Thread oder Interface java.lang.Runnable
- Start der Threads mittels der Methode run()
- Modifier synchronized bei Methoden: vor dem Ausführen der Methode wird ein Lock (Mutex, Semaphore) auf dem Objekt erlangt
  - => Schutz vor gleichzeitigem Zugriff mehrerer Threads
  - Locks sind rekursiv
- auch Code kann durch Lock geschützt werden, indem ein Block mit synchronized eingeleitet wird:
  `synchronized { ... }`

- Strenge Typbindung, statische und dynamischeTypprüfung
- Namensäquivalenz
- primitive Typen: boolean, byte, short, int, long, char, float, double
  - Repräsentation auf JVM festgeschrieben
  - bei Integer-Typen keine Over-/Underflow - Prüfung, bei Float-Typen gibt es ‘Infinity’ und ‘NaN’
  - Kapsel-Klassen: Character, Integer, Boolean, etc.
- sonst alles Referenztypen (=Zeiger), insbesondere auch Arrays und Strings
- Parameterübergabe zwar nur call-by-value, wegen obigem aber eigentlich call-by-reference für alle nicht primitiven Typen

Semantik: Typen (2)

- Variablen von Klassentypen können auf Instanzen ihrer Klasse und aller Unterklassen zeigen (Datenpolymorphismus)
- alle Klassen sind von Object abgeleitet
  - Methoden toString(), equals(), wait(), ...
  - Variable vom Typ Object kann auf alle Objekte zeigen
- Typ eines Objekts kann mit instanceof überprüft werden:
  `if (schroeder instanceof Kanzler) { .... }`
- Typkonversion:
  - von klein nach gross implizit (Kanzler->Politiker, float->double, int->long)
  - von gross nach klein nur mit Cast, dynamische Prüfung
  - von allen Objekten nach String mit '+'
  - nach boolean überhaupt nicht

Semantik: Variablen u. ä.

- Lokale Variablen müssen vor Benutzung initialisiert werden (statische Prüfung)
- Attribute und Array-Komponenten werden automatisch auf 0, null oder false initialisiert
- Java besitzt eine Garbage Collection
  - kein explizites Löschen von Objekten
  - Objekte werden automatisch gelöscht, wenn keine lokale Variable mehr auf sie zeigt
  - Garbage Collection läuft nichtdeterministisch an (besitzt eigenen Thread)
  - kann auch explizit mit System.gc() gestartet werden

Semantik: Ausdrücke

- Beliebige Klammerung möglich
- Auswertungsreihenfolge generell von links nach rechts
  - gilt auch bei Parametern
  - bei Exception in einem Ausdruck werden folgende Ausdrücke nicht ausgewertet
- Operanden werden vor der Operation ausgewertet
- bei Konstruktoren kommt OutOfMemoryError *vor* der Ausdrucksauswertung der Parameter, bei Arrays kommt er *nach* der Auswertung
- als Anweisung können benutzt werden: Zuweisung, ++, --, Methoden (auch falls sie nicht void als ResultType haben), Objektkreation mit new

Semantik: dispatching

- Bei Methodenaufruf wird dynamisch die Methode des Objektes ermittelt, auf das die Variable zeigt, und diese aufgerufen; der Typ der Variable spielt keine Rolle (Operationspolymorphismus)
- Methode muss also nicht explizit als virtuell deklariert werden
- kein dispatching wird dagegen durchgeführt auf
  - Attributen
  - privaten und statischen Methoden (klar)
  - final Methoden -> Compiler-Optimierungen (inline-Code)
- Beispiel:

```
Politiker schroeder = new Kanzler();
schroeder.rede(); // rede() aus Klasse Kanzler
float zahlung = schroeder.verdienst;
// verdienst u.U. aus Klasse Politiker
```

Ausführung (1)

- Quellcode wird durch Compilation in Java-Bytecode umgewandelt
- Dieser wird erst beim Start des Programms gelinkt
- Programme werden auf einer Java Virtual Machine (JVM) ausgeführt - also eigentlich interpretiert
- Dadurch große Plattformunabhängigkeit, aber Geschwindigkeitseinbußen
- JVMs verfügbar für fast alle bekannten Betriebssysteme (aber nicht unbedingt für aktuellste Java-Version)

Ausführung (2)

- Applikationen: eigenständige Java-Programme
  - gestartet durch expliziten Aufruf des Java-Interpreters mit `java <Applikationsname>`
  - dabei ist <Applikationsname> der Name der Hauptklasse
  - diese hat eine Methode public static void main(String[] args), die als erste aufgerufen wird
- Applets: ‘Applikatiönchen’, werden aus dem Internet geladen und im Browser gestartet
  - Hauptklasse ist von java.applet.Applet abgeleitet: Methoden, init, start, stop, destroy
  - ‘Sandbox’-Konzept für Sicherheit
  - Java-Plugin im Browser erforderlich
  - spezielles <applet>-Tag im HTML-Code

Ausführung (3)

- Servlets: sun's Antwort auf CGI - Skripte
  - werden auf einem Server im Internet ausgeführt und generieren HTML-Code
  - benötigen spezielle Servlet-Engine, die mit dem Webserver interagiert (z.B. Tomcat)
  - Klasse abgeleitet von javax.servlet.HttpServlet: Methoden doGet, doPost
- Java Server Pages (JSP): sun's Antwort auf ASP
  - Erweiterung der Servlet-Technologie mit besserer Trennung von Inhalt und Form
  - spezielle JSP-Tags im HTML-Code
  - beim ersten Aufruf wird daraus ein Servlet generiert

Standard-Pakete: Strings und Daten

- Strings:
  - verwaltet in java.lang.String
  - modifiziert in java.lang.StringBuffer: append, insert, delete, substring, ...
  - verglichen mit equals()
  - Umwandlung / Konkatenation mit '+'
- Daten:
  - verwaltet in java.util.Date
  - aktuelles Datum (Zeit) sowie Konversionen mit java.util.Calendar
  - formatierte Ausgabe über java.text.DateFormat
  - für Datenbanktransaktionen java.sql.Date

Standard-Pakete: Ein- / Ausgabe

- Alles Wichtige im package java.io
- Verwendung des Stream - Konzeptes
- U. a. Klassen File, FileReader, BufferedWriter, InputStream, ...
- Klasse System hat 3 statische Felder in, out und err für die Standardein- / bzw. -ausgabe
- Ausgabe z.B. mit

```
System.out.println(“Hallo, Seminar!”);
PrintWriter writer = new PrintWriter(
  new BufferedWriter(new FileWriter(“test.txt”)));
writer.println(“Hallo, Seminar!”);
```

- Eingabe z.B. mit

```
BufferedReader reader = new BufferedReader(
  new InputStreamReader(System.in);
String line = reader.readLine();
```


Standard-Pakete: Überblick

- java.lang: Basisklassen der Sprache: u.a. Object, Thread, Runtime, ClassLoader, Math, Integer, ...
- java.io: Ein- / Ausgabe
- java.net: Netzwerkoperationen, u.a. Socket, URL
- java.sql: Datenbankoperationen, u.a. Connection, ResultSet
- java.rmi: Remote Method Invocation auf Java-Art
- java.security: Verschlüsselung, Signatur etc.
- java.util: Hilfsklassen, u.a. Datenstrukturen (TreeSet, Hashtable, Vector, ...), StringTokenizer, Random, ...
- javax.naming: Zugriff auf Directories wie z.B. X500
- javax.swing sowie java.awt: GUI
- org.omg.* : CORBA

Weitere Java-Komponenten

- JDBC: Java Database Connectivity
- JNDI: Java Naming and Directory Interface
- Java XML:
  - Parsing and Processing: JAXP (javax.xml.parsers, org.w3c.dom)
  - XML Binding: JAXB
  - XML Remote Procedure Call: JAX RPC
  - XML Messaging: JAXM
  - noch im experimentellen Stadium
- Java Message Service: API zum Nachrichtenaustausch zwischen Komponenten
- JavaMail: API zum e-Mail Versand, Empfang, Verwaltung

Weitere Java-Konzepte (1)

- Reflection: Beschreibung von Klassen während der Laufzeit
  - package java.lang.reflection
  - u.a. Klassen Class, Method, Constructor, etc.
- JavaBeans: sich selbst beschreibende Java-Komponenten
  - package java.beans
  - Beans implementieren das Interface BeanInfo
  - Tools können dadurch High-Level - Informationen über die Bean bekommen
  - hauptsächlich für GUI-Building benutzt

Weitere Java-Konzepte (2)

- Enterprise Java Beans:
  - sun's Antwort auf COM+, .net etc.
  - Anwendung besteht aus mehreren Komponenten, die auf verschiedene Server verteilt sind
  - Interaktion der Beans über RMI
- jini:
  - Verteilte Architektur unabhängig vom Netzwerkprotokoll
  - Objekte im Netz beschreiben sich über Java-Klasse
  - Insbesondere auch für Kleingeräte, Embedded Systems, Haushaltsautomatisierung

Literatur (1)

- Alles über Java: http://java.sun.com
- Insbesondere Dokumentation zu Java 2 v 1.3:
  http://java.sun.com/j2se/1.3/docs/index.html
- Dokumentation der Standard-Klassen:
  http://java.sun.com/j2se/1.3/docs/api/index.html
- Spezifikation der Programmiersprache Java:
  http://java.sun.com/docs/books/jld/second_edition/html/jTOC.doc.html
- Spezifikation der Java Virtual Machine:
  http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
- Tutorials zu Java-Standard:
  http://java.sun.com/docs/books/tutorial/?frontpage-spotlight
- Tutorials zu Java-Enterprise (Servlets, JDBC, JNDI, ...):
  http://java.sun.com/j2ee/tutorial/index.html

Literatur (2)

- Ausführliche Einführung zu Java:
  Skript zum Kompaktkurs ‘Java’, erhältlich bei der Fachschaft der Fakultät Informatik der Universität Stuttgart
- Gutes Buch mit Codebeispielen:
  Flanagan, Java Examples In A Nutshell, O’Reilly, ISBN 0-596-00039-1
- Java XML:
  http://java.sun.com/xml/
- Jini:
  http://www.sun.com/jini/
- Viele Links zu öffentlichem Code und FAQ’s:
  http://java.sun.com/nav/developer/resources/index.html?frontpage-resources
- Insbesondere viele Antworten zu Java Enterprise:
  http://www.jguru.com/