Fachhochschule
Kaiserslautern
University of Applied Sciences

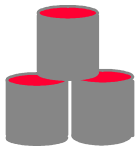
Informatik und
Mikrosystemtechnik
Zweibrücken

Grundlagen der Informatik
– Prozedurale Programmierung –

Prof. Dr. Bernhard Schiefer

(basierend auf Unterlagen von Prof. Dr. Duque-Antón)

bernhard.schiefer@fh-kl.de
http://www.fh-kl.de/~schiefer



Inhalt

- Unterprogramme
- Entwurfs-Konzepte
- Pakete
- Ein- und Ausgabe

Unterprogramme

- Probleme sind häufig zu groß und komplex, um sie am Stück zu lösen
- Lange komplexe Code-Stücke sind:
  - ⇨ sehr schwer wartbar,
  - ⇨ fehleranfällig und
  - ⇨ nicht wiederverwendbar
- Empfehlung aus den Richtlinien für „Clean Code Development“
  - ⇨ The first rule of functions is that they should be small.
    The second rule of functions is that they should be smaller than that.
  - ⇨ Functions should hardly ever be 20 lines long.
- Um ein Programm gut wartbar und wiederverwendbar zu entwickeln, muss es in Unterprogramme zerlegt werden.

Unterprogramme

- Mit Hilfe von Unterprogrammen können unterschiedliche Ziele verfolgt werden:
  - ⇨ **modulare Programmentwicklung**
    - Zur Lösung wird das Problem in mehrere Teilprobleme zerlegt, die einfacher zu programmieren sind. Die Gesamtheit aller Unterprogramme liefert die Gesamtlösung.
  - ⇨ **Abstraktion** vom konkreten Algorithmus
    - Ein Entwickler der Gesamtlösung muss nicht die Details der Lösung der Teilprobleme kennen. Es reicht aus, wenn er den Aufruf des entsprechenden Unterprogramms zur Lösung des Teilproblems kennt.
  - ⇨ **Wiederverwendbarkeit**
    - Spezielle Aufgaben, die mehrfach (an verschiedenen Stellen) gelöst werden sollen, werden als Unterprogramm definiert und können dann mehrfach genutzt werden.
- Ein Unterprogramm wird als Anweisungsblock realisiert, der einen Namen besitzt (zum Aufruf) und der parameterisierbar ist.

Ablauf eines Aufrufs eines Unterprogramms (UP)

- Falls im Programmverlauf ein UP aufgerufen wird, wird der Anweisungsblock über seinen Namen aufgerufen und die Anweisungen des UPs werden ausgeführt.
  - ⇨ bei diesem Aufruf können Parameter zur Verarbeitung übergeben werden
- Nach Abarbeitung der Folge der Anweisungen des UPs kann das Ergebnis als Rückgabewert weiterverarbeitet werden.
  - ⇨ In diesem Fall spricht man von einer **Funktion**.
  - ⇨ **int function (int n)** wird als (Teil-Ausdruck) verwendet, z.B. in: **y = x + function (x);**
- Falls keine Antwort von UP zurückgegen wird, spricht man von **Prozedur**.
  - ⇨ **void procedure (...)** wird als Anweisung verwendet, z.B. **procedure (...);**
  - ⇨ wobei procedure einen Effekt haben muss, wie beispielsweise println (...) oder das Verändern von Variablen.
- Das aufrufende Programm wird nach Beendigung des UPs bei der nächsten Anweisung fortgesetzt. Bei Funktionen wird der nächste Teilausdruck ausgewertet

Sichtbarkeit / Gültigkeit von Variablen

- Wichtige Begriffe:
  - ⇨ **Sichtbarkeit** einer Variablen bedeutet, dass man von einer Programmstelle aus die Variable sieht, also über ihren Namen auf sie zugreifen kann.
  - ⇨ **Gültigkeit** einer Variablen bedeutet, dass an einer Programmstelle der Name einer Variable dem Compiler durch Vereinbarung bekannt ist.
    - eventuell ist sie aber verdeckt – nicht sichtbar
  - ⇨ **Lebensdauer** einer Variablen ist die Zeitspanne, in der die VM der Variablen Speicherplatz zur Verfügung stellt.

Beispiel: ggT als Funktion

```
class Euklid {
  static int ggT (int a, int b) { // Funktion
    int r = a % b;
    while (r != 0) {
      a = b;
      b = r;
      r = a % b;
    } // while
    return b;
  } // ggT

  public static void main (String args []) {
    int a = 30, b = 6;

    int result = ggT (a,b);
    System.out.println (result);
  } // main
} // Euklid
```

formale Parameter: a und b

Definition von ggT

Aufruf von ggT

aktuelle Parameter: a und b

Beispiel: ggT als Prozedur

```
class Euklid {
   static void ggT (int a, int b) { // Prozedur
      int r = a % b;
      while (r != 0) {
         a = b;
         b = r;
         r = a % b;
      } // while
      System.out.println (b);
   } // ggT

   public static void main (String args []) {
      int a = 30, b = 6;
      ggT (a,b);
   } // main
} // Euklid
```

Methoden (in Java)

- Eine Methode in Java besteht aus zwei Teilen:
  - ⇨ Die Signatur (Schnittstelle) legt die Syntax fest . Sie enthält
    - den Namen der Methode
    - Angaben über Anzahl und Typen der formalen Parameter
    - ggf. den Typ des Rückgabewertes.
  - ⇨ Der Rumpf enthält die Anweisungen, die ausgeführt werden, und legt damit die Semantik des Unterprogramms fest.
- Die Signatur hat optional Modifizierer (z.B. static oder public) und kann Exceptions werfen (throws).
  - ⇨ werden später erläutert

Parameter

- Bei den formalen Parameter unterscheidet man:
  - ⇨ **Eingabeparameter**, die aktuelle Parameterwerte (in Methode) übernehmen.
  - ⇨ **Ausgabeparameter**, die Werte (aus Methode) zurückgeben können.
  - ⇨ **Ein/Ausgabeparameter**, die sowohl der Eingabe als auch der Ausgabe dienen.
- Der Aufruf des Unterprogramms erfolgt mit einer Liste **aktueller** Parameter, die an die **formalen** Parameter übergeben (gebunden) werden.
  - ⇨ Datentypen und Anzahl der formalen Parameter müssen mit denen der aktuellen Parameter übereinstimmen.
- Die Parameterliste darf auch leer sein.

Parameter-Übergabe allgemein

- Prinzipielle Möglichkeiten der Parameter-Übergabe :
  - ⇨ **call-by-value :** der aktuelle Parameter bzw. der entsprechende Ausdruck wird zum Zeitpunkt des Aufrufs ausgewertet und sein Wert dem formalen Parameter zugewiesen.
    - Geeignet für Eingabeparameter
  - ⇨ **call-by-reference** : der aktuelle Parameter ist eine Variable oder Ausdruck, über den eine Zuweisung stattfinden kann.

    Der formale Parameter wird zu einer Referenz auf den aktuellen.

    Innerhalb des Unterprogramms stellt damit der formale Parameter einen alternativen Namen dar, um auf die übergebene Variable zuzugreifen.
- Die formalen und aktuellen Parameter des Unterprogrammaufrufes müssen nicht den selben Namen tragen, brauchen aber einen kompatible Datentypen.

Parameter (in Java)

- In Java gibt es nur Eingabevariablen.
  - ⇨ Übergabe ist immer call-by-value
- Out-Parameter gibt es nicht explizit
  - ⇨ Jedoch können Referenzen auch bei Übergabe per call-by-value ähnlich genutzt werden
- Rückgabewerte können mittels return-Anweisung zurückgegeben werden
- Seit Java 5: Parameter-Listen mit variabler Länge
  - ⇨ Um einer Methode beliebig viele Parameter übergeben zu können, kann der letzte Parameter als: `dataTyp ... par` deklariert werden.
    - Der Compiler interpretiert dies als eindimensionales Array vom Typ **dataTyp [ ] par**
    - An dieser Stelle dürfen also beliebig viele oder auch keine Parameter des Typs **dataTyp** stehen

Beispiel: Hello-Methode

- Darstellung des „Hello World“-Beispiels mit einer Methode

```
class HelloWorld {

   // Methode, die Hello World ausgibt
   static void helloSagen (int inWert)  {
      System.out.println ("Hello World! " + inWert);
   } // helloSagen

   // Main-Methode, die Methode helloSagen aufruft
   public static void main (String[ ] args)  {

      // Aufruf von Methode helloSagen!
      helloSagen (42) ;

   } // main
} // class HelloWorld
```

Aufruf von Methoden

- Die im Aufruf aufgeführten Ausdrücke werden ausgewertet und als aktuelle Parameter mittels call-by-value Wert den formalen Parameter des Unterprogrammes übergeben.
- Eine Methode mit Rückgabewert muss eine return-Anweisung im Rumpf besitzen, dessen Ausdruckstyp zum Rückgabetyp passt.
- Eine Methode ohne Rückgabewert (`void`) braucht keine return - Anweisung im Rumpf haben.

Aufruf von Methoden

- Eine Methode mit Namen **methode** wird durch den folgenden Ausdruck aufgerufen:
  - ⇨ **methode (expression_1, ... , expression_n);**
- Beim Aufruf müssen die aktuellen Parameter in Anzahl und Typ mit den Elementen der formalen Parameter-Liste zusammenpassen.
- Methoden mit gleichem Namen aber unterschiedlichen Parametern heißen: ***überladen***
- Bei überladenen Methoden wird die speziellste aufgerufen, d.h. diejenige bei der die wenigsten Typumwandlungen benötigt werden.
  - ⇨ Ist dies nicht entscheidbar (z.B. weil es mehrerer solcher Wandlungen gibt), dann ist der Aufruf nicht gültig.

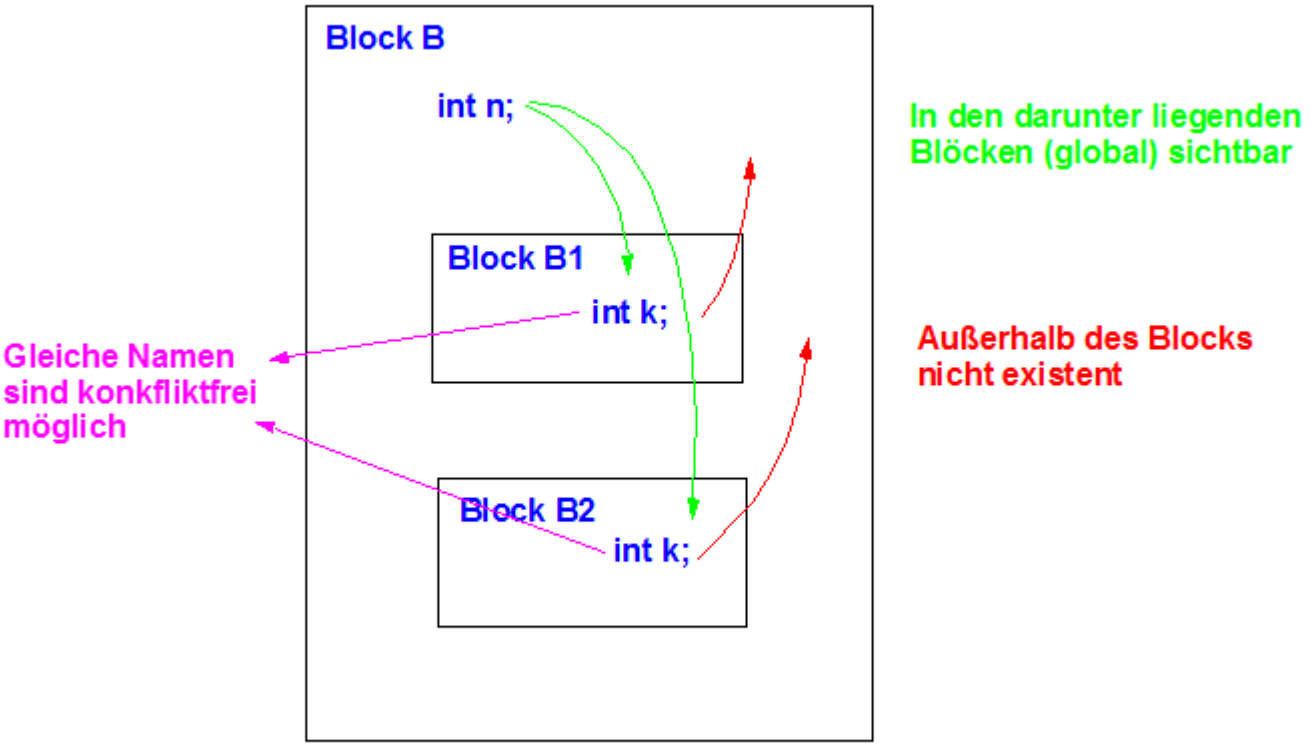
Überladen (Overloading) von Methoden

- Java kann verschiedene gleichbenannte Methoden an Hand der Signatur (Name, Anzahl und Argumenten-Typen) unterscheiden:
  - ⇨ Innerhalb einer Klasse dürfen nicht zwei Methoden mit derselben Signatur deklariert werden.
- Der Typ des Rückgabewertes zählt nicht zur Signatur. Zwei Methoden, die sich nur im Rückgabetyp unterschieden (ansonsten dieselbe Signatur aufweisen) sind nicht möglich und liefern einen Compiler-Fehler, z.B.:
  - ⇨ int berechne (int n) { . . . }
  - ⇨ float berechne (int n) { . . . }

Lokale Variablen

- Eine Variable, die lokal in ihrem(n) Programmsegment(en) gültig ist, wird als lokale Variable bezeichnet.
  - ⇨ Im Prinzip sind alle Variablen, die bisher vorgestellt wurden, lokale Variablen.
  - ⇨ Generell sind also lokale Variablen in Blöcken und Methoden möglich.
- Bei ineinander geschachtelten Blöcken sind alle Variablen des äußeren Blocks auch innen sichtbar.
- Variablen eines inneren Blocks sind außen nicht sichtbar
- Lokale Variablen müssen explizit initialisiert werden. Es gibt also keine Default- Werte für lokale Variablen.

Kommunikation via lokaler Variablen



Mehrere (gleiche) verschiedene lokale Variablen

```
class TestLocalVar1 {

   static void display ( ) {
      int j = 10;
      System.out.println ("j="+j);
   } // display

   public static void main (String[ ] args)  {
      int j = 20;
      display ( );
      System.out.println ("j="+j);

   } // main
} // class TestLocalVar1
```

Lokale Variable „außerhalb“ nicht sichtbar

```
//  Programm TestLocalVar2
//  Autor: Harry Hurtig
class TestLocalVar2 {
   public static void main (String[ ] args)  {

      int j = 20;
      System.out.println ("j="+j);

      {
         int i = 30;
         System.out.println ("i="+i);
      }

      System.out.println ("i="+i); // Fehler!

   } // main
} // class TestLocalVar2
```

Lokale Variable „innerhalb“ sichtbar

```
//  Was wird ausgegeben?
class TestLocalVar3 {
   public static void main (String[ ] args)  {

      int j = 20;
      System.out.println ("j="+j);

      { // Block1
         int i = 30;
         System.out.println ("i="+i);
         System.out.println ("j="+j);
      }  // Block1

   } // main
} // class TestLocalVar3
```

Globale Variablen

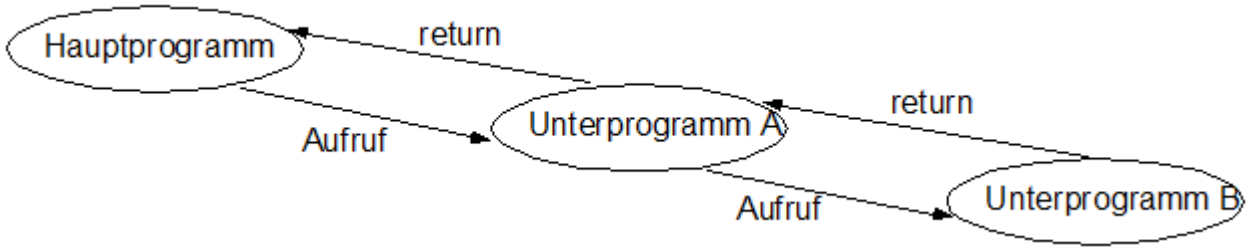
- Globale Variablen
  - ⇨ Variablen, die in allen Segmenten des Gesamtprogramms ohne Verwendung von Übergangsmechanismen gültig sind
  - ⇨ Globale Variable können insbesondere die Parameterübergabe erleichtern.
  - ⇨ Umgekehrt kann durch die Verwendung von globalen Variablen das Programm unübersichtlich werden.
- In Java treten globale Variablen nur in Form von Klassenvariablen auf
  - ⇨ Diese werden mit Hilfe des Attributs **static** außerhalb der Methoden-Definitionen aufgeführt
  - ⇨ Sie sind innerhalb aller Methoden sichtbar und gültig

Beispiel: Globale Variablen

```
class TestGlobalVar {
   static int j = 20; // Globale Variable in Form
                      // einer Klassenvariablen

   public static void main (String[ ] args)  {

      System.out.println ("j="+j);

      { // Block1
         int i = 30;
         System.out.println ("i="+i);
         System.out.println ("j="+j) ;
      }  // Block1

   } // main
} // class TestGlobalVar
```

Geschachtelte Methoden

- Methoden können geschachtelt aufgerufen werden.
  - ⇨ D.h.: Hauptprogramm (HP) ruft Unterprogramm (UP) A, das wiederrum UP B aufruft



- Ablauf ist immer „Stack“ (Stapel-) orientiert:
  - ⇨ Ein Programm unterbricht die Bearbeitung (bleibt an der aktuellen Position in der Folge der Anweisungen stehen) und gibt die Kontrolle an das UP ab.
  - ⇨ Nachdem ein UP seine Bearbeitung abgeschlossen hat, macht das aufrufende Programm an der (gemerkten) Position weiter.
- Die Variablen der Unterprogramme (der Java-Methoden) sind immer lokal

Rekursive Unterprogramme

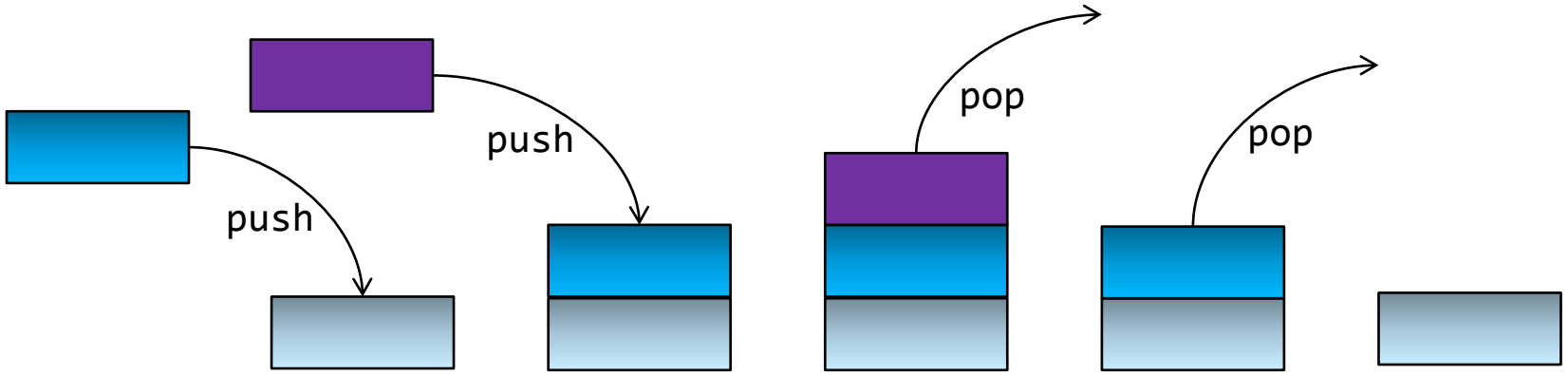
- In einem Unterprogramm A können andere Unterprogramme aufgerufen werden.
- Ruft ein Unterprogramm A sich selbst auf, so spricht man von einem rekursiven Aufruf.
  - ⇨ Da alle Variablen und Parameter eines Unterprogramms lokal sind, gibt es keine Konflikte mit Instanzen eines vorherigen Aufrufs desselben Unterprogramms.
- Eine Rekursion muss nicht immer offensichtlich sein:
  - ⇨ Die Rekursion kann auch indirekt erfolgen, wenn innerhalb eines Unterprogramms A ein Unterprogramm B aufgerufen wird und von dort wiederum A
  - ⇨ in diesen Fällen spricht man auch von einem *verschränkt rekursiven Aufruf*.

Beispiel: Rekursive Berechnung Fakultät

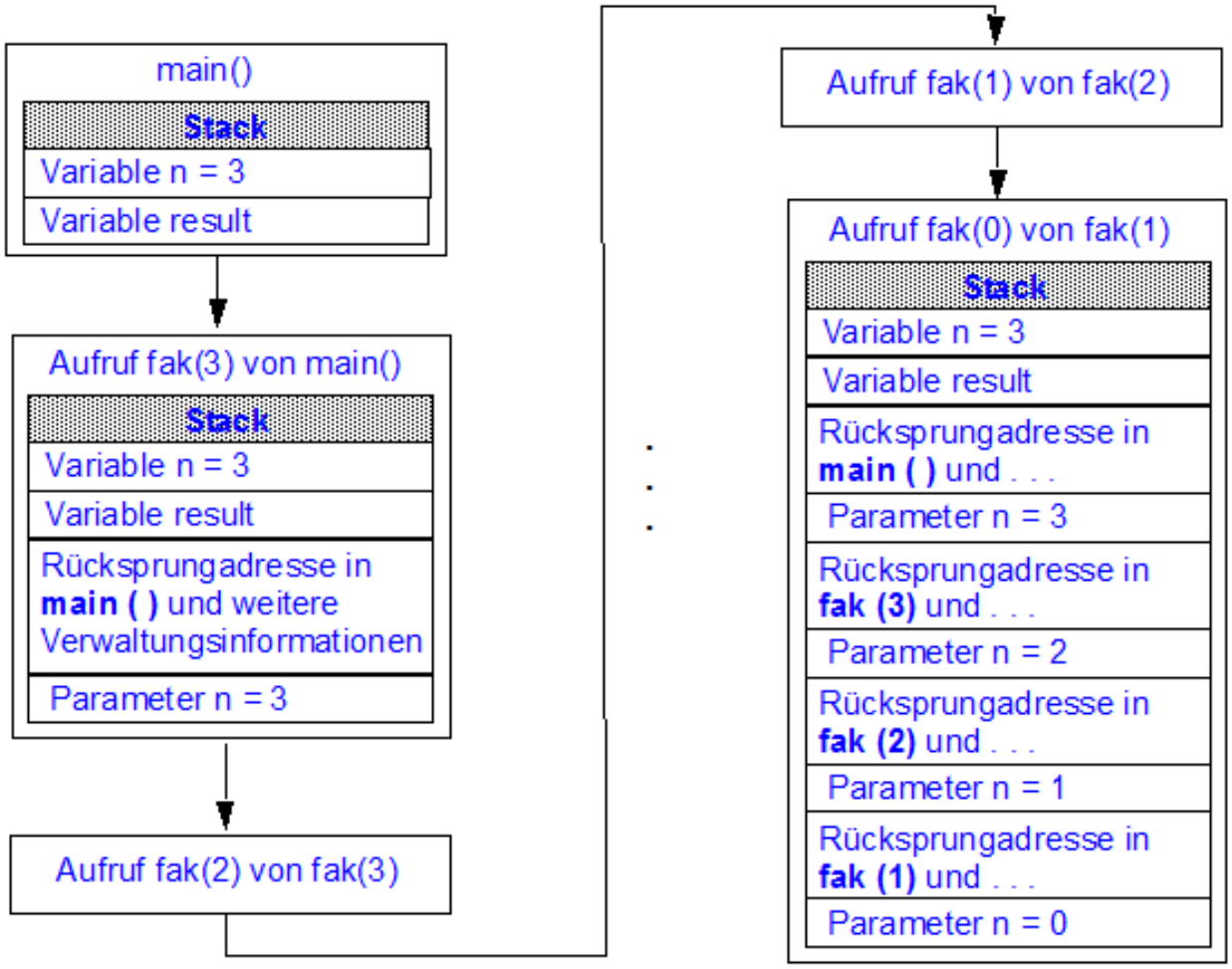
```
class RekursivFak {

   static int fakultaet (int n)  {
      if ( n <= 0)     { // 0! = 1
         return 1;
      } else { // n > 1
         return n * fakultaet (n-1);
      }
   } // fakultaet

   public static void main (String[] args)  {
      int n = 5;
      int result = fakultaet (n);
      System.out.println (n + "! = " + result);
   } // main
} // RekursivFak
```

Exkursion: Stack (Aufrufstapel)

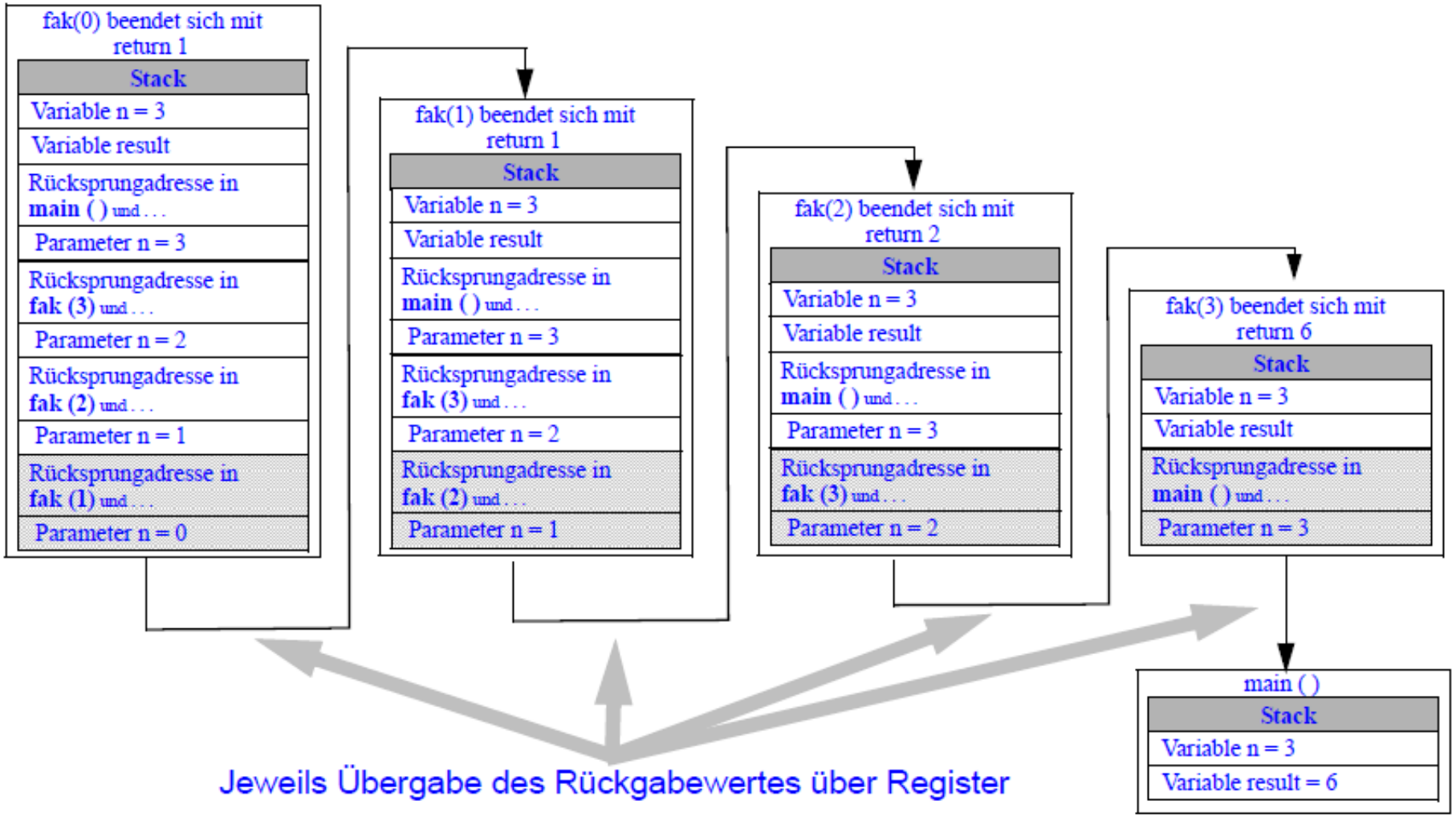
- Laufzeitumgebungen werden mit Hilfe eines Stapels (Stack) verwaltet.
  - ⇨ Jeder neue UP-Aufruf führt zu einem Ablegen (Push) der neuen Laufzeitumgebung auf dem **Stack.** Die darunterliegenden Schichten werden dadurch verdeckt
  - ⇨ Wird das UP beendet, erfolgt der Abbau (Pop) von dessen Umgebung
  - ⇨ Ein Stack ist ein Kellerspeicher, der nach dem LIFO-Prinzip funktioniert: LIFO - Last In First Out
  - ⇨ Der letzte Aufruf, der auf dem Stack abgelegt wird, wird als erster vom Stack entfernt.



Aufbau des Stacks: Fakultät



Abbau des Stacks: Fakultät



Top-Down-Entwurf

- Üblicherweise wird ein neu zu lösendes (Gesamt-) Problem mit Hilfe eines **Top-Down**-Entwurfs gelöst:
  - ⇨ Gesamtproblem wird von „oben“ im Hauptprogramm aufgegriffen und in sachgerechte Teilprobleme nach „unten“ weitergereicht.
  - ⇨ Tiefere Schichten werden später konkretisiert. Zu Beginn wird nur die formale Struktur der Unterprogramme, also die Deklaration der Methoden benötigt.
  - ⇨ Die Kunst bei diesem Vorgehen besteht zum einen darin, die richtigen Teilprobleme zu identifizieren und zum anderen, geeignete Schnittstellen (Signaturen) zu finden.
  - ⇨ Nachdem die Dekomposition in Teilprobleme erfolgt ist, können die einzelnen Teilprobleme unabhängig voneinander algorithmisch und programmtechnisch gelöst werden.
- In seltenen Fällen, wenn Lösungen zu Teilproblemen bereits vorliegen, kann auch eine umgedrehte Vorgehensweise sinnvoll sein. Man spricht in diesem Fall von **Bottom-Up**-Entwurf.

Top-Down-Entwurf: ggT-Beispiel

```
class Euklid {
   public static void main (String args []) {
      Scanner sc = new Scanner(System.in);
       int a = eingabe (sc); int b = eingabe (sc);
       ausgabe (ggT (a, b));
   } // main

   static int ggT (int a, int b) {
       int r = a % b;
       while (r != 0) {
         a = b; b = r; r = a % b; }
       return b;
   } // ggT

   static int eingabe (Scanner sc) {
        System.out.println ("Bitte Wert eingeben: ");
       return sc.nextInt();
   }  // eingabe

   static void ausgabe (int result) {
       System.out.println ("ggT liefert Ergebnis: " + result);
   }  // ausgabe
} // Euklid
```

!! Achtung:
Aus Platzmangel wurden in diesem Beispiel zur lesbareren Darstellung einige Anweisungen in eine Zeile geschrieben.

Top-Down-Entwurf: Matrix-Multiplikation-Beispiel

```
class MatrixMultiplikation {
   public static void main (String[] args) {
      // Datenstrukturen für eine Matrix m und einem Vektor v,
      // die zu multiplizieren sind und in a aufzunehmen
      int [] [] m;
      int [] v, a;
      m = einlesenMatrix (args[0]);
      v = einlesenVector (args[1]);
      if ( compatibility (m, v) )  {
         a = multiplikation (m, v);
         ausgabe (a, args[2]);
      } else { // Multiplikation nicht möglich
         System.out.println ("Daten sind nicht konsistent");
      }
   } // main

   static int [] [] einlesenMatrix (String datei ) { . . .}
   static int []  einlesenVector (String datei ) { . . .}
   static boolean compatibility (int [] [] m, int [] v) { . . .}
   static int []  multiplikation (int [] [] m, int [] v) { . . .}
   static void ausgabe (int [] a, String datei) { . . .}
} // MatrixMultiplikation
```

Wiederverwendbarer Code

- Um wiederverwendbaren Code zu schreiben, muss im Prinzip der Code, der von anderen Entwicklern potentiell genutzt werden soll:
  - ⇨ in eine separate Datei geschrieben werden und
  - ⇨ entsprechend zugänglich gemacht werden.
- Im Beispiel des ggT würde man dazu zwei Dateien schreiben, etwa
  - ⇨ TestEuklid, welches die main-Methode enthält und
  - ⇨ ggtEuklid, welches alle anderen Methoden enthält.
- Die main-Methode ist die einzige Methode, die ausführbar ist, d.h. die Java VM setzt hier an bei Programmstart.
  - ⇨ die anderen Methoden werden nur bei Bedarf dazu geholt und eingebunden.
  - ⇨ Zur exakten Unterscheidung der Methoden muss nun in der main-Methode die entsprechende Methode aus der ggT-Euklid-Klasse zusammen mit der Klasse, in der sie definiert ist, angesprochen werden.

Wiederverwendbarer Code: ggT-Beispiel

```
class GgtEuklid {
   static int ggT (int a, int b) {
      int r;
      r = a % b;
      while (r != 0) {
         a = b;
         b = r;
         r = a % b;
      } // while
      return b;
   } // ggT
   static int eingabe () {
      System.out.println ("Bitte Wert eingeben: ");
      return sc.nextInt();
   }  // eingabe
   static void ausgabe (int result) {
      System.out.println ("ggT liefert Ergebnis: " + result);
   }  // ausgabe
} // ggtEuklid
```

```
class TestEuklid {
  public static void main (String[] args){
    Scanner sc = new Scanner(System.in);
    int a = GgtEuklid.eingabe (sc);
    int b = GgtEuklid.eingabe (sc);
    int result = GgtEuklid.ggT (a, b);
    GgtEuklid.ausgabe (result);
    sc.close();
  } // main
} // TestEuklid
```

Programmier-Style Guide

- Eine Gestaltungsrichtlinie (Style Guide) für das Erstellen von Programmen umfasst Darstellungsregeln, welche die Lesbarkeit der Programme erleichtern sollen.
- Solche Regeln beruhen auf einer gegenseitigen Übereinkunft. Die Anwendung der Regeln ist nicht zwingend, da nicht die Korrektheit, sondern die Lesbarkeit der Programme davon abhängt.
- Zum Style Guide gehört zum Beispiel auch eine Konvention zum Einrücken in Blöcken oder dass vor der Definition einer Methode eine Leerzeile stehen soll, damit man leichter erkennt, dass jetzt eine neue Methode kommt.
- Ich habe bisher ebenfalls einen bestimmte Regel angewendet, welche?
- Im Kapitel „Kern imperativer Sprachen“ wurden bereits einige Anmerkungen zu den Namenskonventionen für Bezeichner gemacht.

Programmier-Style Guide

- Neben den Style Guides, wie ein Programm am besten zu schreiben ist, werden auch oft Regeln zur Programmierung selbst erstellt:
  - ⇨ Zu jeder Anwendung sollte eine Test-Anwendung (Umgebung) geschrieben werden.
  - ⇨ Bei der Programm-Entwicklung möglichst „kleine Schritte" gehen und diese austesten, bevor die nächsten Schritte avisiert werden.
- Am besten wäre es, wenn vor jeder Methode ein größere Kommentar-Block steht, der grob das Ziel der Methode beschreibt.

Pakete

- Pakete definieren Software-Einheiten, die unabhängig oder in Kombination mit anderen Paketen zur Bildung von Anwendungen verteilt werden können.
- Pakete sind Sammlungen von Klassen und eignen sich ideal zur Organisation von Code-Bibliotheken (eigene oder externe), aus verschiedenen Gründen:
  - ⇨ Pakete erzeugen eine Gruppierung verwandter Ressourcen (z.B. GUI).
  - ⇨ Pakete erzeugen Namensräume und vermeiden dadurch Namenskonflikte (da Paket als Präfix zur Unterscheidung herangezogen wird).
  - ⇨ Pakete ermöglichen ein angepasstes Zugriffskonzept (über die zugrundeliegende Verzeichnis-Struktur).

Pakete (package)

- In Java gibt es ein sehr umfangreiches System hierarchisch geordneter Pakete mit der Wurzel (oberster Knoten) Paket java.
- Um eine Klasse einen bestimmten Paket zuzuordnen (Anbieten), wird die folgende Anweisung verwendet:
  - ⇨ **package Paketname ;**
- Um eine Klasse aus einem (externen) Paket einzubinden, wird die folgende Anweisung verwendet:
  - ⇨ **import Paketname .Klasse ;**
- Daneben gibt es in Java eine spezielles Archivierungformat mit dem Namen **jar**.
  - ⇨ Diese stellt eine Bibliothek mit Inhalt aus einem oder mehreren Paketen dar.

Package

- Der Paketname wird auf das Dateiverzeichnis abgebildet.
  Es entspricht also jedem Namensteil ein Unterverzeichnis:
  - Falls z.B. in Umgebungsvariable **CLASSPATH** das Verzeichnis **D:\Vorlesungen\Java** enthalten ist, muss dazu dort ein Unterverzeichnis mit dem Namen **packagePath** erzeugt werden.
  - Durch den Beispiel-Code wird dann ein Paket **packagePath** generiert, welches die Klasse **ClassHello.java** enthält.
  - Alle weiteren Code-Dateien des Packages müssen in dieses Verzeichnis kopiert und dort compiliert werden: **D:\Vorlesungen\Java\packagePath**
  - Pakete sind allerdings allein nicht ausführbar, sondern nur in Verbindung mit darin befindlichen Klassen, von denen eine die main-Methode enthält.

```
package  packagePath;
public class ClassHello {
   public static void displayHello () {
      System.out.println ("Hallo aus Paket heraus");
   }
} // PackageHello
```

Import

```
import packageName.ClassHello;
class TestPackage {
    public static void main (String[] args) {
        ClassHello.displayHello();
    }
} // TestPackage
```

- Um nicht die gewünschte Klasse mit ihrem vollem Pfadnamen ansprechen zu müssen:
  - ⇨ packageName.ClassHello.displayHello();
- kann die import-Anweisung am Anfang einer Quelldatei verwendet werden (hinter möglichen package-Anweisungen).
- Klassen, in denen die package-Anweisung fehlt, gehören automatisch zu einem Default-Paket.
- Solche Klassen können ohne explizite import-Anweisung gesucht werden. Das Default-Paket entspricht dem aktuellen Arbeitsverzeichnis.

Ein- und Ausgabe (I/O-System)

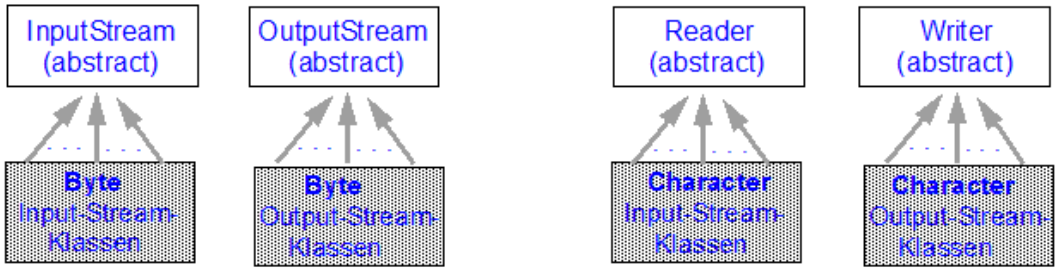
- Typische Programme laufen fast immer nach dem „EVA“-Prinzip (Eingabe -Verarbeitung-Ausgabe) ab.
- Daraus folgt die Bedeutung eines guten Ein-/Ausgabe-Systems für die Programm- Entwicklung bzw. für die Akzeptanz der entsprechenden Programmiersprache.
- Ein gutes Ein- und Ausgabesystem muss
  - ⇨ nicht nur viele Datenquellen und Datensenken berücksichtigen und möglichst einheitlich behandeln,
  - ⇨ sondern auch mit Hilfe dieser Quellen und Senken Informationen in verschiedenen Darstellungen wie binär (Daten) oder Zeichen (Character: ASCII bzw. Unicode) austauschen.

Ein- und Ausgabe (I/O-System)

- In Java wird dies mit Hilfe von Streams gelöst.
  - ⇨ Ein Stream ist eine geordnete Folge von Bytes (Bytestream), meist von unbekannter Länge (d.h. die Anzahl der Bytes, die transportiert werden sollen, ist im Voraus nicht ermittelbar).
  - ⇨ Das Stream-Konzept verbirgt die komplizierten Einzelheiten der Kommunikation. Die entsprechenden Stream-Klassen, die für die Ein- und Ausgabe verwendet wer- den, befinden sich in dem Paket java.io.

Klassifizierung von Streams

- Das Paket **java.io** ist eines der größten Pakete der Java-API und ist trotz eines durchdachten Entwurfs recht unübersichtlich.
- Zum exakten Verständnis des entsprechenden Pakets sind eigentlich Kenntnisse der Objekt-Theorie notwendig. Aus diesem Grund wird hier nur ein intuitiver Einstieg an Hand von typischen Anwendungsbeispielen vermittelt.
- Eine erste grobe Klassifizierung unterscheidet also:
  - ⇨ Sink-Stream-Klassen, die in Datensenken schreiben und
  - ⇨ Spring-Stream-Klassen, die aus Datenquellen lesen.
- Daneben kann man die Klassenhierarchie von java.io in
  - ⇨ Byte-Stream-Klassen (binäre Daten) und
  - ⇨ Character-Stream (Zeichen) unterscheiden.



Die Klasse System

- Im implizit (importierten) Paket java.lang ist insbesondere die Klasse **System** enthalten, welche wichtige Methoden zur Kommunikation mit dem zu Grunde liegenden Betriebssystem zur Verfügung stellt.
  - ⇨ Die Variante **System.out.println ("Hello World")** haben wir ja schon in einigen Beispielen kennen gelernt und gesehen, dass dort kein import java.lang.System nötig war
  - ⇨ **java.lang** ist das einzige Paket, das implizit importiert wird, weil es zentrale Elemente enthält.
- System enthält drei Klassenvariablen (Attribute):
  - ⇨ **in** – Standard-Eingabe
  - ⇨ **out** – Standard-Ausgabe
  - ⇨ **err** – Standard-Error-Ausgabe

Die Klasse System

- Wichtige Methoden betreffen die Speicherverwaltung, Status und die Ausgabe von System- Eigenschaften und Systemzeit.
  - ⇨ **System.gc ()** ruft den Garbage Collector explizit auf
  - ⇨ **System.exit (int status)** beendet das laufende Programm vorzeitig.
  - ⇨ **System.currentTimeMillis ()** liefert die Anzahl der Millisekunden, die seit dem 1.1.1970 um 00:00:00 Uhr GMT bis zum Aufruf der Methode vergangen sind.
  - ⇨ **System.getProperties ()** liefert die definierten Systemeigenschaften der Plattform als Objekt der Klasse Properties.

Standard I/O-Situationen

- In Java gibt es eine verwirrend hohe Anzahl an Möglichkeiten, Eingaben und Ausgaben vorzunehmen.
  Für die restliche Veranstaltung wird die folgende Unterteilung vorgenommen:
  - ⇨ Ausgabe
    - auf Display (Standard-Ausgabe)
    - in eine Datei (binär- oder Character-weise)
  - ⇨ Eingabe
    - **interaktiv**: von Tastatur (Standard-Eingabe)
    - **embedded**: von einer Datei (binär- oder Character-weise)
    - im **Batch**-Betrieb: von der Kommandozeile

Ausgabe auf Display

- Die Methode **printf ()** ermöglicht eine formatierte Ausgabe auf dem Bildschirm.
- Der allgemeine Aufbau der Formatelemente eines Formatstrings ist wie folgt (eckige Klammern sind optional):
  - %[Parameterindex$][Steuerzeichen][Feldbreite][.Genauigkeit]Umwandlungszeichen
  - Im einfachsten Fall ist einem auszugebenden Parameter genau ein Formatelement zugeordnet.
  - Die Reihenfolge der Formatelemente entspricht der Reihenfolge der Parameter in der Parameterliste. Mit Hilfe des Parameterindex kann die Ausgabereihenfolge explizit kontrolliert werden.
- Natürlich kann man auch mit print (), println () und weiteren Methoden weniger komfortabel arbeiten.

Ausgabe auf Display

- System.out.printf ("%d %d %n", 3, 4); → Ausgabe: 3 4
- System.out.printf ("%2$d %1$d %n", 3, 4); → Ausgabe: 4 3
- System.out.printf ("%2$d %2$d %n", 3, 4); → Ausgabe: 4 4
- System.out.printf ("%20d %d %n", 3, 4); → Ausgabe: 3 4
- System.out.printf ("%+-20d %d %n", 3, 4); →Ausgabe: +3 4
- System.out.printf("%.5e %n", 20.) → Ausgabe: 2.00000e+01

Character Ausgabe in eine Datei

```
import java.io.*;
class TestAusgabeDatei {
   public static void main (String[] args) throws IOException {
      // Character Output Stream pWriter wird deklariert
      PrintWriter pWriter;
      // pWriter wird mit der zu schreibenden Datei instanziert
      pWriter = new PrintWriter("D:/Vorlesungen/Java/test.txt");
      // Hilfsvariable, um Werte zu generieren,
      // die auf Datei geschrieben werden sollen
      int i = 0;

      // Daten in pWriter und damit in Datei schreiben
      for (int j=0; j < 3; j++) {
         pWriter.printf ("%-10d %n", i);
         i += 100;
      }

      // Daten aus Puffer in Datei schreiben und PrintWriter schliessen
      pWriter.flush ( );
      pWriter.close ( );
   } // main
} // TestAusgabeDatei
```

Character Ausgabe in eine Datei: Varianten

- ```
  . . .
  PrintWriter pWriter = new PrintWriter (new FileWriter("test.txt"));
  . . .
  ```

- ```
  . . .
  FileOutputStream fos = new FileOutputStream ("test.txt");
  PrintWriter pWriter = new PrintWriter (fos);
  . . .
  ```

- ```
  . . .
  FileOutputStream fos = new FileOutputStream ("test.txt");
  PrintWriter pWriter = new PrintWriter (new OutputStreamWriter (fos));
  . . .
  ```

- ```
  . . .
  File f = new File ("test.txt");
  FileOutputStream fos = new FileOutputStream (f);
  PrintWriter pWriter = new PrintWriter (new OutputStreamWriter (fos));
  . . .
  ```

Binäre Ausgabe in eine Datei

```
import java.io.*;
class TestAusBaseData {
   public static void main (String [] args) throws IOException {

      // Öffnen der Datei Daten.dat zum binären Schreiben
      FileOutputStream fos = new FileOutputStream ("Daten.dat");
      DataOutputStream dos = new DataOutputStream (fos);

      // Drei Beispielwerte werden in Daten.dat geschrieben
      dos.writeInt (1);
      dos.writeDouble(1.1);
      dos.writeInt (2);

      // Schliessen der Datei Daten.dat
      dos.close ();

   } // main
} // TestAusBaseData
```

Eingabe von der Tastatur I

```
import java.util.Scanner;
class TestEingabeTastatur {
   public static void main (String[] args) {
      // Scanner von der Tastatur anlegen
      Scanner sc = new Scanner(System.in);

      System.out.println("Geben Sie Daten folgender Typen ein:");
      System.out.print("int, double, String \n");
      // Daten vom Scanner lesen
      int i = sc.nextInt();
      System.out.println("int: " + i);

      double d = sc.nextDouble(); // mit Komma statt Punkt eingegeben!
      System.out.println("double: " + d);

      String str = sc.next();
      System.out.println("string: " + str);

      sc.close(); // Scanner abschliessen
   } // main
} // TestEingabeTastatur
```

Eingabe von der Tastatur II

```
import java.util.Scanner;
class TestBeliebigeEingabeTastatur {
   public static void main (String[] args) {

      Scanner sc = new Scanner(System.in); // Scanner von der Tastatur

      // Beispielvariablen unterschiedlichen Typs deklarieren
      int i;

      System.out.println("Geben Sie ganzzahlige Werte ein, "
                         + "gefolgt von einem Punkt: ");
      while (sc.hasNextInt()) {

         i = sc.nextInt(); // Daten vom Scanner lesen

         System.out.println("Der nächste Wert ist " + i);
      } // while

      sc.close(); // Scanner schließen
   } // main
} // TestBeliebigeEingabeTastatur
```

Character-Eingabe von einer Datei

```
import java.util.Scanner;
import java.io.*;
class TestEingabeDatei {
   public static void main (String[] args) throws IOException {

      // Scanner von der Datei "test.txt" anlegen
      Scanner sc = new Scanner(new File ("D:/demo/test.txt") );

      // Beispielvariablen - Typs beliebig
      int i;

      // Daten vom Scanner lesen
      for (int j = 0; j < 3; j++) {
         i = sc.nextInt(); System.out.printf ("%d %n", i);
      } // for

      sc.close(); // Scanner abschliessen

   } // main
} // TestEingabeDatei
```

Binäre Eingabe von einer Datei

```
import java.io.*;
class TestEinBaseData {
   public static void main (String [] args) throws IOException {

      // Öffnen der Datei Daten.dat zum binären Lesen mit Test-Ausgabe
      FileInputStream fis = new FileInputStream ("Daten.dat");
      DataInputStream dis = new DataInputStream (fis);

      // Die drei Beispielwerte werden aus Daten.dat gelesen
      //und auf dem Bildschirm ausgegeben
      System.out.println (dis.readInt ());
      System.out.println (dis.readDouble ());
      System.out.println (dis.readInt ());

      dis.close ();  // Schliessen der Datei Daten.dat

   } // main
} // TestEinBaseData
```

Eingabe von der Kommando-Zeile I

- Das Einlesen von der Kommando-Zeile kann bequem mit Hilfe der (den Datentypen) entsprechenden Hüll-Klassen (Wrapper-Klassen) realisiert werden.
- Zu jedem primitiven Datentyp gibt es eine sogenannte Hüllklasse, deren Objekte einer Variablen dieses Datentyps entsprechen:
  - ⇨ Die entsprechenden Hüll-Klassen sind im Package java.lang enthalten und damit implizit verfügbar, z.B. java.lang.Integer.
  - ⇨ Hüllklassen bieten eine Reihe von Methoden für den Zugriff auf die „umhüllten“ Variablen an. So kann z.B. mit Hilfe der Methode **parseInt** eine Folge von Zeichen (Zeichenkette) in einen (primitiven) int-Wert umgewandelt werden.
  - ⇨ Neben weiteren Methoden (wie equals() und toString()) werden in jeder numerischen Hüllklasse wichtige Konstanten angeboten wie z.B. MIN_VALUE und MAX_VALUE, die den kleinsten bzw. größten Wert des Wertebereichs des entsprechenden Datentyps darstellen.
- Zum Einlesen von der Kommando-Zeile kann wie folgt verfahren werden, falls z.B. das erste Argument eine int-Zahl ist.
  - ⇨ int i = Integer.parseInt (args[0]);

Eingabe von der Kommando-Zeile II

```
class TestEinAusArgumente {
   public static void main (String[] args)  {

      /* erstes Argument muss eine ganze Zahl sein */
      int i = Integer.parseInt(args[0]);

      /* zweites  Argument muss ein double sein */
      double d = Double.parseDouble(args[1]);

      /* drittes Argument muss ein String sein */
      String s = args[2];

      /* Test-Ausgabe der Werte */
      System.out.println("int = "+ i +", double = "+ d +", string = "+ s);

   } // main
} // TestEinAusArgumente
```

I/O-Zusammenfassung (Ausgabe)

- Ausgabe
  - ⇨ Display:
    - System.out.printf (...);
  - ⇨ characterweise Datei:
    - PrintWriter pWriter = new PrintWriter ("datei.txt");
      pWriter.printf (...);
  - ⇨ byteweise Datei:
    - DataOutputStream dos =
      new DataOutputStream (new FileOutputStream ("datei.dat"));
      dos.writeInt (n);

I/O-Zusammenfassung (Eingabe)

- Eingabe
  - ⇨ Tastatur:
    - Scanner sc = new Scanner (System.in);
      sc.nextInt ( );
  - ⇨ characterweise Datei:
    - Scanner sc = new Scanner (new File ("datei.txt"));
      sc.nextInt ( );
  - ⇨ byteweise Datei:
    - DataInputStream dis =
      new DataInputStream (new FileIntputStream ("datei.dat"));
      dis.readInt ( );
  - ⇨ Kommandozeile:
    - int i = Integer.parseInt (args [i]);