Inhalt

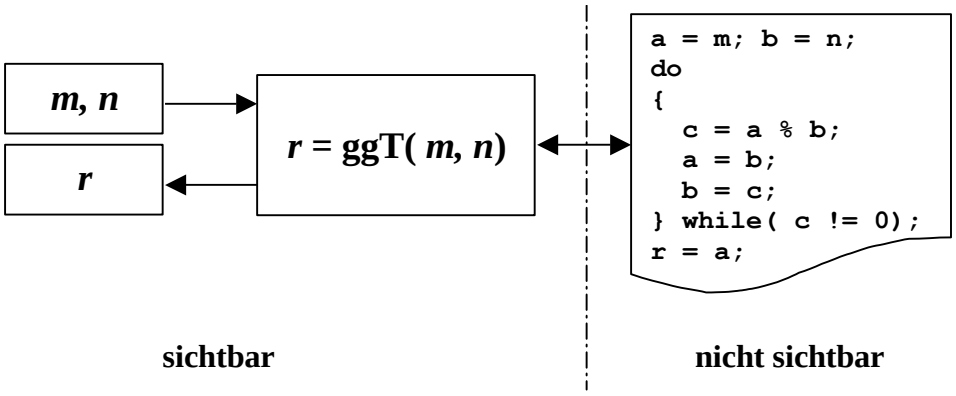
7 Methoden

7.1 Prozedurorientierte Programmierung

Das letzte Beispiel zeigt, dass die Programme durch ihren Umfang immer *unübersichtlicher* wurden. Es wird Zeit, dass wir Mittel zum **Strukturieren** eines Programms in der Softwareentwicklung kennenlernen.
Ein weiterer Aspekt für eine Strukturierung ist die *Wiederverwendbarkeit* von bereits entwickelten Programmteilen. Logisch zusammengehörige Befehle werden dabei zu einer **Prozedur** (Unterprogramm) zusammengefasst. Den **Prozedurnamen**, die **Argumente** (Eingabewerte) und **Resultate** (Ausgabewerte) einer Prozedur bezeichnet man als **Schnittstelle** bzw. als **Signatur** dieser. Durch klare Schnittstellenbestimmung wird deren Wiederverwendbarkeit ermöglicht, *ohne* dass der Nutzer den in der Prozedur realisierten Algorithmus selbst kennen muss.

Schnittstelle einer Prozedur (Signatur):
Prozedurname, Argumente (Eingabewerte) und Resultate (Ausgabewerte)

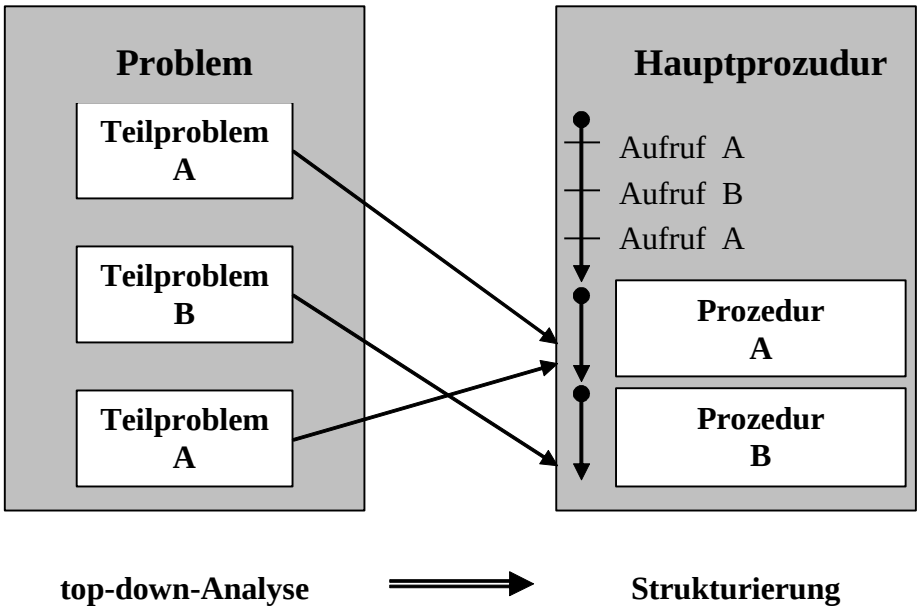
Dem euklidischen Algorithmus werden die beiden Startwerte *m* und *n* übergeben, nach seiner Ausführung liefert er das Ergebnis *r* zurück. *m*, *n* und *r* bilden die Schnittstelle der Funktion *ggT*, deren Datentypen sind bekannt. Dabei interessiert nicht, wie der euklidische Algorithmus umgesetzt wurde, ob mit einer der oben angeführten Befehlsfolge oder mit einer ganz anderen.



Zielgerichteter Entwurf durch schrittweise Verfeinerung (Niklaus Wirth 1971 - Pascal)
Mittels **top-down-Analyse** zerlegt man ein *komplexes Problem* solange in *kleinere Teilprobleme* bis diese überschaubar werden. Jedes Teilproblem wird zunächst als **Prozedur** *realisiert* und auch *ausgetestet*, danach werden alle zum komplexen Problem entwickelten Prozeduren zu einem Ganzen zusammenfasst.

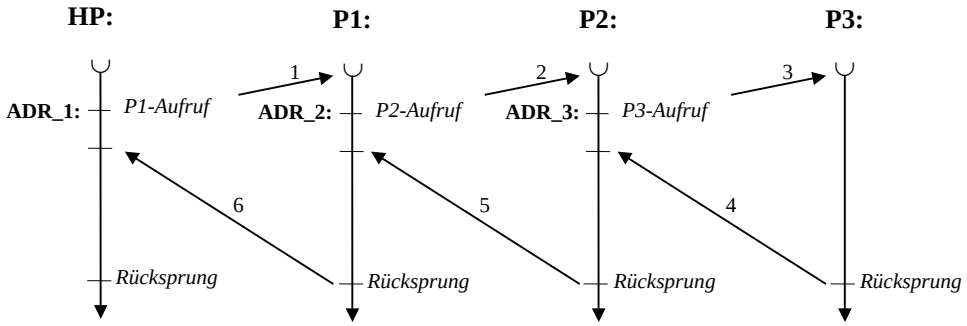
⇒ **Prozeduren dienen der Strukturierung eines Programms,**

1. **der Zerlegung komplexer Probleme in kleinere und**
2. **der Ausgliederung wiederkehrender Berechnungen.**

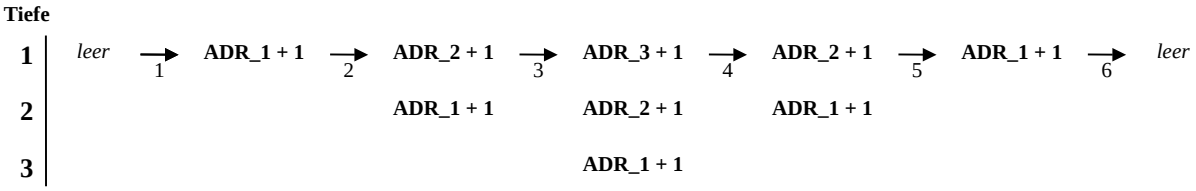


Die Gesamtheit *aller* Prozeduren, zusammen mit der Vorschrift ihres Zusammenwirkens in einer Hauptprozedur, realisiert eine Lösung zu einem Problem.

Verarbeitung von Prozeduren im Compiler
HP ... Hauptprozedur, P*i* ... Prozedur *i*, ADR_*i* ... Adresse der Prozedur *i*



Keller für Rücksprungadressen



Zur Realisierung von Prozeduren in Programmiersprachen sind drei Sprachelemente notwendig:

<u>Prozeduren:</u> **Prozeduraufruf**
Prozedurdefinition
Prozedurrücksprung

Alle modernen Programmiersprachen lassen Prozeduren zu. Man spricht in diesem Zusammenhang von der **prozedurorientierten Programmierung**. In den Sprachen **C** und **Java** werden *Funktionen* im Sinne von *Prozeduren* verwendet.

In der objektorientierten Programmierung bezeichnet man Prozeduren als **Methoden**. Wie schon oben kurz erwähnt, gehören *Methoden* neben den *Attributen* zum festen Bestandteil von *Klassen*. Da bisher noch nicht objektorientiert programmiert wurde, werden hier zunächst sogenannte **Klassenmethoden** betrachtet. Klassenmethoden existieren unabhängig von Objekten.

7.2 Klassenmethoden

Bereits im Kapitel 1 haben wir verschiedene Programme zur Berechnung des größten gemeinsamen Teilers und des kleinsten gemeinsamen Vielfachen zweier natürlicher Zahlen für unseren Modellrechner diskutiert.
In einer Klasse **`Euklid`** im Paket `Tools` werden die Funktionen `ggT( m, n)` und `kgV( m, n)` als **Klassenmethoden** definiert. Klassenmethoden kann man aufrufen, *ohne* ein Objekt der entsprechenden Klasse gebildet zu haben. Die Methoden selbst sind imperativ programmiert, vgl. entsprechende Programme unseres Modellrechners. Werden die Methoden mit nicht zugelassenen Parametern aufgerufen, so weisen sie auf einen entsprechenden Fehler hin und brechen die Bearbeitung ab.

Zur Umsetzung wurden die im ersten Kapitel angegebenen Programme für den Modellrechner leicht verändert.

Programm *r* = ***ggT(*** *m, n****)*** ***// ohne Rekursion***

```
a = m;
b = n;
c = a % b;
while( c != 0)
{
  a = b;
  b = c;
  c = a % b;
}
r = b;
```

Programm *s* = ***kgV(*** *m, n****)*** ***// 3. Variante***

```
s = m * n / ggT( m, n)
```

Euklid.java

```
// Euklid.java                                          MM 2014
// Diese Klasse gehoert zum package Tools.
package Tools.Euklid;

/**
 * Bestimmen des groessten gemeinsamen Teilers ggT
 * und des kleinsten gemeinsamen Vielfachen kgV
 * zweier natuerlicher Zahlen m und n
 * mittels des Euklidschen Algorithmus.
 */
public class Euklid
{
/**
 * ggT( m, n),
 * erzeugt ein Objekt der Klasse NPlusException,
 * falls Parameter m, n <= 0.
 * @param m natuerliche Zahlen > 0
 * @param n natuerliche Zahlen > 0
 * @return ggT( m, n)
 */
  public static int ggT( int m, int n)
```

```
    throws NPlusException
  {
    if( m <= 0 || n <= 0) throw new NPlusException();
    int r;
    do
    {
      r = m % n;
      m = n;
      n = r;
    } while( n != 0);
    return m;
  }

/**
 * kgV( m, n),
 * erzeugt ein Objekt der Klasse NPlusException,
 * falls Parameter m, n <= 0.
 * @param m natuerliche Zahlen > 0
 * @param n natuerliche Zahlen > 0
 * @return ggT( m, n)
 */
  public static int kgV( int m, int n)
    throws NPlusException
  {
    if( m <= 0 || n <= 0) throw new NPlusException();
    return m * n / ggT( m, n);
  }
}
```

Bei Fehleingaben erzeugt diese Klasse eine Fehlerausschrift, indem sie ein Objekt der Klasse `NPlusException` erzeugt und weiterleitet:

NPlusException.java

```
// NPlusException.java                                MM 2014
package Tools.Euklid;

/**
 * Definition der Ausnahme NPlusException.
 */

public class NPlusException extends Exception
{
  public String toString()
  {
    return
    "NPlusException: Keine positive natuerliche Zahl!";
  }
}
```

In einer Klasse **GgTKgVApplication** wird mit der Klassenmethode **main** ein Programm gestartet, welches Methoden der Klasse **Euklid** und der Klasse **IOTools** des Paketes **Tools** nutzt:

- **Euklid.ggT()** und **Euklid.kgV()** rufen die entsprechend definierten Klassenmethoden der Klasse **Euklid** zur Berechnung des größten gemeinsamen Teilers bzw. des kleinsten gemeinsamen Vielfachen zweier natürlicher Zahlen auf.
- Die Klassenmethode **IOTools.readInteger()** der Klasse **IOTools** liest ganze Zahlen von der Tastatur.

GgTKgVApplication.java

```
// GgTKgVApplication.java                                    MM 2014

import Tools.IO.*;                                        // IOTools
import Tools.Euklid.*;                                     // Euklid

/**
 * Berechnung ggT und kgV,
 * Anwendung des Pakets Tools.
 */
public class GgTKgVApplication
{
  public static void main( String[] args)
  {
    int m = IOTools.readInteger( "m(m>0) = ");
    int n = IOTools.readInteger( "n(n>0) = ");

    try
    {
      System.out.print( "ggT( " + m + ", " + n + ") = ");
      System.out.println( Euklid.ggT( m, n));

      System.out.print( "kgV( " + m + ", " + n + ") = ");
      System.out.println( Euklid.kgV( m, n));
    }
    catch( Exception e)
    {
      System.err.println( "FEHLER! " + e);
    }
  }
}
```

Bei den weiteren Methodenaufrufen handelt es sich um keine Klassenmethoden:

- Der Aufruf **System.out.println()** veranlasst eine zeilenweise Konsolenausgabe der Ergebnisse der Berechnung als Text.
- **System.err.println()** veranlasst eine zeilenweise Konsolenausgabe bei einem aufgetretenen Fehler.

7.2.1 Methodenvereinbarung

Eine Methode besteht aus einem **Methodenkopf** und einem **Methodenrumpf**. Im *Methodenkopf* werden die *Schnittstellen* (sichtbarer Bestandteil, *Signatur*) einer Methode festgelegt. Der *Methodenrumpf* (unsichtbarer Bestandteil) beinhaltet den bei Aufruf der Methode auszuführenden Code.

Eine **Java-Methode** besteht aus einer Folge hintereinandergeschriebener *Deklarationen* und *Anweisungen*.

public static *Ergebnistyp Methodenname* **(** *Parameterliste* **)** // Methodenkopf
{
Anweisungen // Methodenrumpf
}

- **public static** legt fest, dass die Methode eine *öffentliche*, für jeden zugreifbare *Klassenmethode* ist.
- Eine Methode hat *höchstens* ein Ergebnis. Der *Ergebnistyp muss* angegeben werden. Wird *kein* Wert als Ergebnis zurückgegeben, so ist der Ergebnistyp `void`.
- Die *Parameterliste* ist eine durch Kommas getrennte *Liste von Variablendeklarationen*. Man nennt diese Variablen **formale Parameter**. Alle in einer Parameterliste definierten formalen Parameter sind nur innerhalb des Methodenrumpfes **sichtbar**. Außerhalb der Methode kann man *nicht* auf die Parameter zugreifen.
  Auch wenn die Parameterliste *leer* ist, müssen die Klammern `()` gesetzt werden.
- **Lokale Parameter** sind Variablen, die im Methodenrumpf definiert werden. Alle lokal verwendeten Variablen sind ebenfalls nur innerhalb des Methodenrumpfs *sichtbar*. Außerhalb der Methode kann man *nicht* auf diese zugreifen.

Bereits bekannt ist der Methodenkopf der **Hauptmethode** `main`, welche ein Programm startet:

```
public static void main( String[] args){ ... }
```

Diese Methode hat den Ergebnistyp `void` und liefert damit keinen Wert zurück. Der Methodenname ist `main` und die Parameterliste besteht aus einem Feld vom Typ `String`, welches den Namen `args` trägt. `String` ist eine Java-Klasse zum Darstellen und Verarbeiten von Text.

Für die Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen wird im **Methodenkopf** als *Methodenname* `ggT` gewählt. Es werden zwei natürliche Zahlen benötigt. Damit sind zwei *formale Parameter* von einem ganzzahligen Typ einzuführen (`int m, int n`). Das Ergebnis ist ebenfalls eine natürliche Zahl vom Typ `int`. Ein *Fehler* `NPlusException`, der innerhalb der Methode auftreten kann, wird dem übergeordneten Programm weitergeleitet.

```
public static int ggT( int n, int m)       // Methodenkopf
  throws NPlusException
```

Im **Methodenrumpf** wird zunächst überprüft, ob die Methode mit *nicht* zugelassenen Parametern aufgerufen wurde. In diesem Fall wird ein entsprechender *Fehler* erzeugt und die Bearbeitung abgebrochen.

Im anderen Fall wird mittels Euklidischem Algorithmus die Berechnung unter Verwendung der *aktuellen Parameter* `int m` und `int n` ausgeführt. Dazu wird noch eine *lokale Variable* `int r` benötigt.
Das Ergebnis vom Typ `int` wird durch die *Rücksprunganweisung* `return` an die aufrufende Methode zurückgegeben

```
{                                                    // Methodenrumpf
  if( m <= 0 || n <= 0) throw new NPlusException();

  int r;                                          // lokale Variable
  do                                                       // Euklid
  {
    r = m % n;
    m = n;
    n = r;
  } while( n != 0);

  return m;                                          // Ruecksprung
}
```

7.2.2 Methodenaufruf

Zum **Methodenaufruf** übergibt man für Klassenmethoden die *Klasse*, welche die Methode definiert, und mit einem *Punkt* verknüpft, den *Namen der Methode*. Anschließend folgt eine Liste von **Argumenten**, die sogenannten **aktuellen Parameter**. Als Argumente sind auch *typenverträgliche* Ausdrücke erlaubt.

Klassenname.Methodenname **(** *Argumentenliste* **)**

```
int m = IOTools.readInteger( "m(m>0) = ");
int n = IOTools.readInteger( "n(n>0) = ");
```

Wird eine Klassenmethode in der definierenden Klasse selbst verwendet, so kann der Klassenname beim Aufruf wegfallen. Ein Methodenaufruf selbst stellt einen *elementaren Ausdruck* dar und kann deshalb auch in komplexen Ausdrücken auftreten.

```
return m * n / ggT( m, n);
```

Die Werte der *Argumente* werden als *Kopie* übergeben (**call by value**): Bei jedem Methodenaufruf wird Speicher für die *formalen Parameter* angefordert und mit den Werten der *Argumente* initialisiert. Dann wird der Methodenrumpf ausgeführt. Nach dem Abarbeiten des Methodenrumpfs wird der berechnete Wert an die Aufrufstelle zurückgegeben, der Speicher für aktuelle Parameter und lokale Variablen freigegeben und das Programm fortgesetzt.

Die Methode `kgV` wertet nach einer Fehlerüberprüfung den Ausdruck `m * n / ggT( m, n)` aus. Dieser wird berechnet und als Ergebnis der Methode schließlich an die aufrufende Methode zurückgegeben.

```
public static int kgV( int m, int n)
```

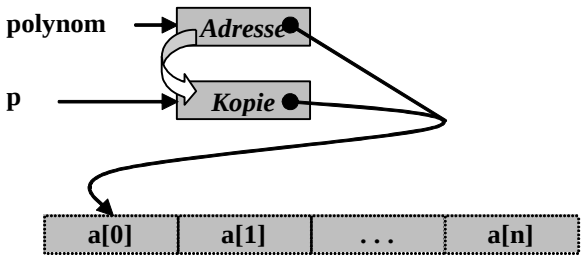
```
    throws NPlusException
{
    if( m <= 0 || n <= 0) throw new NPlusException();

    return m * n / ggT( m, n);              // Methodenaufruf
}
```

7.2.3 Referenzdatentypen in der Schnittstelle einer Methode

Referenzdatentypen können sowohl als *Parameter* übergeben werden, als auch *Ergebnistyp* einer Methode sein. Da der Wert einer Referenz eine *Adresse* ist, werden bei der Parameter- bzw. Ergebnisübergabe Adressen kopiert.

**Methoden mit Referenzen in der Schnittstelle greifen direkt auf den Speicher zu.
Damit besteht die Möglichkeit,
durch eine Methode mehrere Werte gleichzeitig zu manipulieren.**



Manchmal ist es aus Datenschutzgründen besser, statt der Referenz auf die Originaldaten, eine Kopie der Daten zu übergeben. Damit wird eine Manipulation der Originaldaten verhindert.

7.2.4 Beispiel „Polynomwertberechnung mittels Hornerschema“

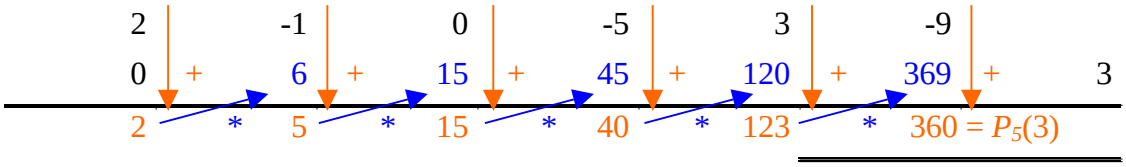
Grundgedanke
Durch geeignetes Ausklammern verhindert man das zeitaufwendige Berechnen der Potenzen in einem Polynom und spart damit Rechenzeit.

Ausgangspunkt
Polynom *r*-ten Grades $P_r(x) = a_r x^r + a_{r-1} x^{r-1} + \cdots + a_1 x + a_0$

Gegeben sei das Polynom 5. Grades $P_5(x) = 2x^5 - x^4 - 5x^2 + 3x - 9$, gesucht sei der Wert des Polynoms an der Stelle $x_0 = 3$, also $P_5(3)$. Nun formt man das Polynom so um, dass keine Potenzen mehr vorhanden sind.

$$P_5(x) = 2x^5 - x^4 - 5x^2 + 3x - 9 = ((((2x-1)x+0)x-5)x+3)x-9$$

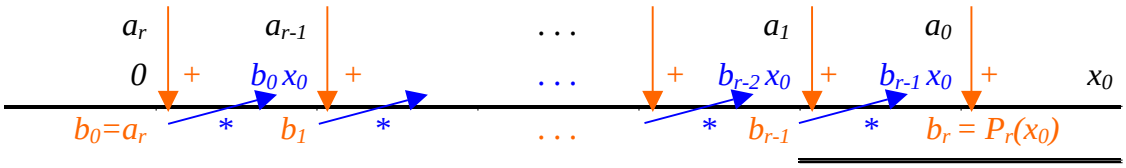
Diese Darstellungsweise gestattet eine sehr schnelle Berechnung, auch mit dem Taschenrechner. Das dafür verwendete Rechenschema trägt den Namen **Hornerschema**, nach **William Georg Horner (1786 - 1837)**, und sieht folgendermaßen aus:



Die Probe zeigt, dass das Ergebnis korrekt ist.

Allgemein $$P_r(x) = a_r x^r + a_{r-1} x^{r-1} + \cdots + a_1 x + a_0 = (\cdots(a_r x + a_{r-1})x + \cdots + a_1)x + a_0$$

Hornerschema für die Polynomwertberechnung an der Stelle x_0



mit $b_0 = a_r$, $b_{i+1} = a_{r-i-1} + b_i x_0$ für $i = 1, ..., r$ und $b_r = P_r(x_0)$.

Durch dieses Verfahren wird der Rechenaufwand wesentlich verringert:

Polynomwertberechnung ohne Hornerschema:

Mult $= \sum_{i=1}^{r} i = \frac{r(r+1)}{2}$

Add $= r$

Mult + # Add $= \frac{r^2 + 3r}{2}$

Der Rechenaufwand steigt quadratisch mit dem Grad des Polynoms.

Polynomwertberechnung mit Hornerschema:

Mult = r

Add = $r + 1$ (*einschließlich der Addition mit 0 am Anfang*)

Mult + # Add = $2r + 1$

Der Rechenaufwand steigt linear zum Grad des Polynoms.

Klasse `Horner`:

Methode Tastatureingabe des Grades und der Koeffizienten eines Polynoms.
Schnittstelle **`double[] polynomEingabe()`**

Methode Konsolenausgabe eines Polynoms.
Schnittstelle **`void polynomAusgabe( double[] p)`**

Methode Berechnung eines Polynomwertes mittels *Horner*.
Schnittstelle **`double horner( double[] p, double x)`**

Methode Konsolenausgabe einer Polynomwertberechnung.

Schnittstelle **`void ergebnisHorner( double[] p, double x, double y)`**

Ein Testprogramm demonstriert die Verwendung der Methoden.

Horner.java

```
// Horner.java                                          MM 2014
import Tools.IO.*;                                   // Eingaben

/**
 * Polynomwertbestimmung mittels HORNER-Schema,
 * P(x) = a[0] + a[1]*x + a[2]*x^2 + ...+ a[r]*x^r
 */
public class Horner
{
/**
 * Tastatureingabe des Grades und
 * der Koeffizienten eines Polynoms.
 * @return double-Feld, Koeffizienten des Polynoms
 */
  public static double[] polynomEingabe()
  {
    int r;
    do
    {
      r = IOTools.readInteger( "Grad des Polynoms(r>0): ");
    } while( r < 1);
    double[] p = new double[ r + 1];

    System.out.println( "Koeffizienteneingabe");
    for( int i = 0; i < p.length; i++)
      p[ i] = IOTools.readDouble( "a[" + i + "]= ");

    return p;
  }

/**
 * Konsolenausgabe eines Polynoms.
 * @param p double-Feld, Koeffizienten des Polynoms
 */
  public static void polynomAusgabe( double[] p)
  {
    System.out.println( "Polynomausgabe");
    System.out.print( "P( x) = ");
    for( int i = 0; i < p.length; i++)
    {
      if( i != 0) System.out.print( " + ");
      System.out.print( " " + p[ i] + "*x^" + i);
    }
    System.out.println();
  }
```

```
/**
 * Berechnung eines Polynomwertes mittels Horner.
 * @param p double-Feld, Koeffizienten des Polynoms
 * @param x Argument der Polynomberechnung
 * @return Polynomwert
 */
  public static double horner( double[] p, double x)
  {
    double y = 0;
    for( int i = p.length - 1; i >= 0; i--)
      y = p[ i] + y * x;
    return y;
  }

/**
 * Konsolenausgabe einer Polynomwertberechnung.
 * @param p double-Feld, Koeffizienten des Polynoms
 * @param x Argument der Polynomberechnung
 * @param y Polynomwert
 */
  public static void ergebnisHorner
  ( double[] p, double x, double y)
  {
    polynomAusgabe( p);
    System.out.println( "P( " + x + ") = " + y);
  }

/**
 * Testprogramm.
 */
  public static void main( String[] args)
  {
// Eingabe Polynom
    double[] polynom = polynomEingabe();

// Eingabe Argument
    double x = IOTools.readDouble( "Argument x = ");

// Berechnung und Ausgabe des Polynomwertes
    ergebnisHorner( polynom, x, horner( polynom, x));
  }
}

/*
  Testpolynom
  P(x) = 1 + 2*x + 3*x^2 + 4*x^3
  P(0) = 1, P(1) = 10, P(2) = 49,
  P(3) = 142, P(4) = 313, P(5) = 586
 */
```

7.2.5 Überladen von Methoden

Es ist möglich, mehrere Methoden mit dem gleichen Namen zu versehen. Man spricht vom **Überladen** der Methoden. Welche der Methoden aufgerufen wird, entscheidet die Parameterliste, d.h. *Anzahl, Typ und Position der Parameter*. **Es wird *nicht* nach dem Typ des *Rückgabewertes* unterschieden!**

Max.java

```
public static int max( int x, int y)                    // 1
{
  return ( x > y) ? x : y;
}

public static double max( double x, double y)           // 2
{
  return ( x > y) ? x : y;
}

public static double max( double x, int y)              // 3
{
  return ( x > y) ? x : y;
}

public static int max( int x, int y, int z)             // 4
{
  return max( max( x, y), z);
}


/*
  max( 5, 3)            // 1 Anzahl der Argumente + Datentyp
  max( 5, 3, 7)         // 4 Anzahl der Argumente
  max( 1.2, 3.4)        // 2 Datentyp
  max( 1.2, 3)          // 3 Position
  max( 5, 3.4)          // 2 Position
*/
```

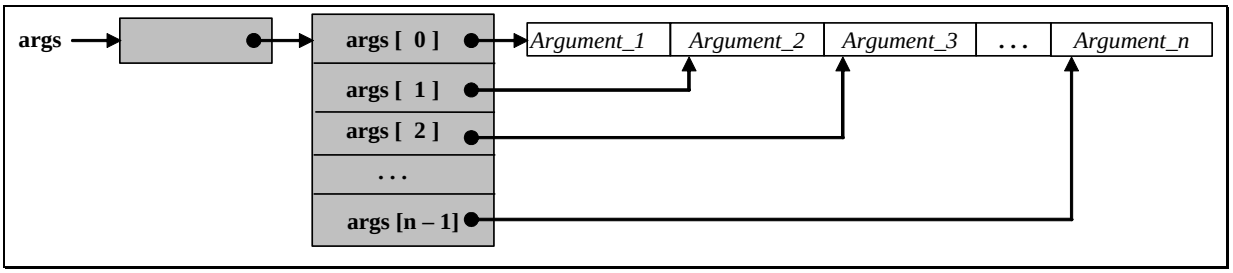
7.3 Die Kommandozeile

Der Standard sieht vor, dass sich ein Programm über das Betriebssystem eine Reihe von Zeichenketten besorgen kann, die beim Aufruf des Programms festgelegt werden.

Die **Kommandozeile** *Kommando Argument_1 Argument_2 . . . Argument_n* wird dabei anhand der Leerzeichen in verschiedene Zeichenketten unterteilt. Zur Beschaffung dieser Zeichenketten dient der Parameter `args` im Methodenkopf der Hauptmethode `main`:

```
public static void main( String[] args) { ... }
```

Die Parameterliste besteht aus einem Feld von Strings, welches den Namen `args` trägt und auf den Anfang der *Argumentliste* zeigt. Jede Stringkomponente wiederum zeigt auf den Anfang eines Arguments der Kommandozeile, so ergibt sich folgende Datenstruktur:



Kmd.java

```
// Kmd.java                                          MM 2003

/**
 * Auswertung einer Kommandozeile, Demoprogramm.
 */
public class Kmd
{
  public static void main( String[] args)
  {
    System.out.println
    ( "Anzahl der Argumente: " + args.length);
    for( int i = 0; i < args.length; i++)
      System.out.println
      ( "" + i + ". Argument: '" + args[ i] + "'");
  }
}
```

$ java Kmd 1 2 3 => **Anzahl der Argumente: 3**
0. Argument: ´1´
1. Argument: ´2´
2. Argument: ´3´

7.4 Rekursionen

Rekursionen sind Programmabschnitte, die sich selbst aufrufen. Dabei muss ein *Abbruch* der Aufrufe garantiert werden.

Programm *r = ggT(m, n)* mit Rekursion

```
a = m;
b = n;
if( b != 0)
{
  r = ggT( b, a % b);
}
else
{
  r = a;
}
```

Rekursion, problemorientiert

Direkte Rekursionen:	Methoden, die sich selbst aufrufen.
Indirekte Rekursionen:	Methoden, die sich wechselseitig aufrufen.

In Java sind *direkte* und *indirekte Rekursionen* erlaubt. Eine Ausnahme bildet hierbei die Methode `main`, die grundsätzlich *nicht* aufgerufen wird.

7.4.1 Beispiel „Eine unendliche Geschichte“

```
>>
Es war einmal ein Mann, der hatte sieben Söhne.
Die sieben Söhne sprachen: „Vater erzähle uns eine Geschichte!“
Da fing der Vater an:

Es war einmal ein Mann, der hatte sieben Söhne.
Die sieben Söhne sprachen: „Vater ...

Und wenn sie nicht gestorben sind, so erzählen sie noch heute.
<<
```

Erzaehlung.java

```
// Erzaehlung.java                                    MM 2014
import Tools.IO.*;                                 // Eingaben

/**
 * Eine unendliche Geschichte.
 */
public class Erzaehlung
{
/**
 * Methode mit Rekursion.
 */
  public static void geschichte()
  {
    if( mannLebtNoch())
```

```
      {
        erzaehlung();
        geschichte();                          // Rekursion
      }
    }

  /**
   * Erzaehlen der Geschichte.
   */
    public static void erzaehlung()
    {
      System.out.print( "\nEs war einmal ein Mann, ");
      System.out.println( "der hatte sieben Soehne.");
      System.out.print( "Die sieben Soehne sagten: ");
      System.out.print( "\"Vater, erzaehl uns");
      System.out.println( " eine Geschichte\"");
      System.out.println( "Da fing der Vater an:\n");
    }

  /**
   * Test, Abbruch der Rekursion.
   * @return true, falls Mann noch lebt, sonst false
   */
    public static boolean mannLebtNoch()
    {
      char weiter = 'j';
      weiter = IOTools.readChar( "Lebt Mann noch (j/n)? ");
      if( weiter != 'j') return false;
      return true;
    }

  /**
   * Hauptprogramm, ruft die rekursive Methode auf.
   */
    public static void main( String[] args)
    {
      geschichte();
    }
  }
```

7.4.2 Beispiel „Türme von Hanoi“

Türme von Hanoi des Monsieur Claus,
Anagramm des Mathematikers Lucas (vor 100 Jahren)

Legende: Zu Benares in Indien gibt es einen Tempel, in dem seit vielen Jahrhunderten Priester bemüht sind, einen Turm so umzulegen, wie Brahma es ihnen selbst geboten hat. Der Turm besteht aus 64 der Größe nach geordneten und auf einer Stange aufgesteckten dünnen goldenen Plättchen. Die Aufgabe lautet, den wenige Zentimeter hohen Turm von der ersten Stange auf die letzte Stange der drei zur Verfügung stehenden Stangen zu

schaffen. Dabei darf jeweils nur eine Scheibe transportiert werden und niemals eine größere Scheibe auf eine kleinere gelegt werden. Sobald die Mönche dieses Ritual hinter sich gebracht haben, wird der Tempel zu Staub zerfallen und die Welt mit einem Donnerschlag untergehen! Wann wird wohl mit der indischen Götterdämmerung zu rechnen sein?

Trotz der indischen Geschichte ist das Puzzle als „Türme von Hanoi" bekannt geworden.

public static void hanoi(int k, int a, int b)

```
/**
 * Rekursive Funktion, Transport der Scheiben.
 *   1.  (k-1) Scheiben von Stange a nach Stange c
 *   2.   k. Scheibe von Stange a nach Stange b
 *   3.  (k-1) Scheiben von Stange c nach Stange b
 * @param k Anzahl der Scheiben
 * @param a Startstange
 * @param b Zielstange
 */

public static void hanoi( int k, int a, int b)
{
  if( k > 0)
  {
    hanoi( k - 1, a, 6 - a - b);                    // 1.
                                                    // 2.
    System.out.println("" + k + ": " + a + " => " + b);

    hanoi( k - 1, 6 - a - b, b);                    // 3.
  }
}
```

Vermutung: **Bei** n **Scheiben sind** $f(n) = 2^n - 1$ **Bewegungen erforderlich.**

Beweis: Sei *M* die Menge aller der natürlichen Zahlen, für die die Behauptung gilt.

IA ($1 \in M$): Bei einer Scheibe ist nur eine Bewegung notwendig.

$$f(1) = 2^1 - 1 = 1$$

IS ($n \in M \rightarrow n' \in M$):

$$f(n+1) = 2 * f(n) + 1, \text{ wegen 1., 2. und 3. s.o.}$$
$$= 2 * (2^n - 1) + 1, \text{ nach IV}$$
$$= 2^{n+1} - 2 + 1 = 2^{n+1} - 1$$

Nach dem Induktionsaxiom gilt *M* = *N* und damit die Vermutung.

Man benötigt bei n Scheiben mindestens $2^n - 1$ ***Züge, um das Problem zu lösen. Das sind bei 64 Scheiben 18'446'744'073'709'551'615 Aktionen.***

Die Priester werden wohl noch Milliarden von Jahren zu tun haben.

7.4.3 Iteration oder Rekursion

Jede *Rekursion* ist als *Iteration* auflösbar. Iterationen sind durch Nichtvorhandensein der rekursiven Methodenaufrufe in der Programmausführung *effizienter*.

Betrachten wir nochmals das Bestimmen des größten gemeinsamen Teiler *r* zweier natürlichen Zahlen *m* und *n* mittels euklidischem Algorithmus ***r = ggT(m, n)***, rekursiv und iterativ. Wir haben bereits beide Varianten kennengelernt.

iteratives Programm ***r = ggT(m, n)***

```
a = m; b = n; c = a % b;
while( c != 0)
{
  a = b; b = c; c = a % b;
}
r = b;
```

Java:

```
public static int ggT( int a, int b) throws NPlusException
{
  if( m <= 0 || n <= 0) throw new NPlusException();
  int c;
  do
  {
    c = a % b;
    a = b;
    b = c;
  } while( c != 0);
  return a;
}
```

rekursives Programm ***r = ggT(m, n)***:

```
a = m; b = n;
if( b != 0)
{
  r = ggT( b, a % b);
}
else
{
  r = a;
}
```

Rekursion

Java:

```
public static int ggT(int a, int b) throws NPlusException
{
  if( m <= 0 || n <= 0) throw new NPlusException();
  if( b != 0) return ggT( b, a % b);
  return a;
}
```

In der Klasse **`Euklid`** des Pakets **`Tools`** wurde eine *iterative* Variante verwendet. Diese Methode ließe sich, bei Beibehaltung der Schnittstellen ohne weiteres durch eine *rekursive* Methode ersetzen. Alle Programme, die auf die alte Klasse `Euklid` zurückgreifen, verhalten sich auch nach einer Korrektur unverändert. Natürlich wäre diese Variante *uneffektiv*.

7.5 Konstanten und Methoden der Klasse `java.lang.Math`

Die vordefinierte (`final`) Klasse **`Math`** gehört zum Paket **`java.lang`** des Standardumfangs von Java. Dieses Paket wird beim Übersetzen automatisch eingebunden und muss deshalb *nicht* importiert werden. **`Math`** stellt mehrere mathematische **Klassenkonstanten** und **Klassenmethoden** zur Verfügung.

Klassenkonstanten

```
public static final double E;                  // Leonard Euler
public static final double PI;              // Ludolf van Ceulen
```

Klassenmethoden (Auswahl)

Name	Parameter-anzahl	Parameter-typ	Ergebnis-Typ	Beschreibung
abs	1	double	double	Betrag eines Wertes
abs	1	float	float	Betrag eines Wertes
abs	1	long	long	Betrag eines Wertes
abs	1	int	int	Betrag eines Wertes
acos	1	double	double	Arcus Cosinus
asin	1	double	double	Arcus Sinus
atan	1	double	double	Arcus Tangens
ceil	1	double	double	ganzzahliges Aufrunden
cos	1	double	double	Cosinus
exp	1	double	double	E-Funktion
floor	1	double	double	ganzzahliges Abrunden
log	1	double	double	natürlicher Logarithmus
max	2	double	double	Maximum
max	2	float	float	Maximum
max	2	long	long	Maximum
max	2	int	int	Maximum
min	2	double	double	Minimum
min	2	float	float	Minimum
min	2	long	long	Minimum
min	2	int	int	Minimum
pow	2	double	double	Potenzfunktion
random	0		double	Zufallszahl zwischen 0 und 1
round	1	double	double	ganzzahliges Runden
sin	1	double	double	Sinus
sqrt	1	double	double	Quadratwurzel
tan	1	double	double	Tangens

Der Aufruf einer *Klassenmethode* erfolgt über die Klasse mit dem Punktoperator:

$$y = \frac{(a-b)^3 - 1}{1 + \sin^2 x}$$

```
double a, b, x, y;
y = ( Math.pow( a - b, 3.0) - 1)
     /( 1 + Math.sin( x) * Math.sin( x));
```

```
double x, y;
  y = 1 + Math.sqrt( x) / 2
        + Math.sqrt( Math.sqrt( x)) / 24;
```

$$y = 1 + \sqrt{x}/2 + \sqrt{\sqrt{x}}/24$$