

**Inhalt**

5	Numerische Methoden der praktischen Mathematik (I) .....	5-2
5.1	<i>Einführung</i> .....	5-2
5.2	<i>Funktionen</i> .....	5-4
5.2.1	Funktionsklassen .....	5-4
5.2.2	Beispiel „Lineare Funktionen - Geraden“ .....	5-7
5.3	<i>Polynomberechnung</i> .....	5-13
5.3.1	Hornerschema .....	5-13
5.3.2	Erweitertes Horner Schema .....	5-14
5.3.3	Klasse Polynom .....	5-15
5.3.4	Klasse RationaleFunktion .....	5-18
5.4	<i>Reihenentwicklungen</i> .....	5-21
5.4.1	Exponentialfunktion .....	5-21
5.4.2	Klasse Exp .....	5-22
5.4.3	Winkelfunktionen .....	5-26
5.4.4	Klasse Sinus .....	5-28
5.4.5	Klasse Cosinus .....	5-30

## 5 Numerische Methoden der praktischen Mathematik (I)

### 5.1 Einführung

Zwischen Mathematik und Praxis (Technik, Physik, Ökonomie, ...) ist über die Jahrhunderte hinweg eine befruchtende Wechselwirkung zu beobachten:

Mit *Rechenhilfsmitteln* wurden zunächst nur *formelmäßig* lösbare Probleme mathematisch bearbeitet. Später, zur Zeit der *mechanischen Rechenmaschinen*, entwickelten die Mathematiker Verfahren, welche mittels sogenannten *Rechenschemata* gelöst wurden. Heute gestatten die modernen *elektronischen Rechenanlagen* neue numerische Lösungsverfahren, die als *Programme* auf diesen Anlagen ausgeführt werden.

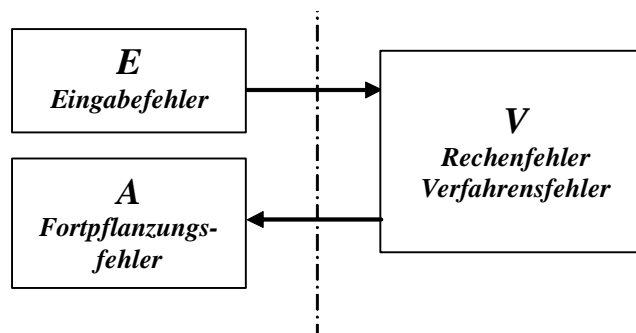
Rechentechnik	Algorithmen	Entwickler	Ausführer
<b>Rechenhilfsmittel (Urzeit)</b>	Formeln	Naturwissenschaftler	Naturwissenschaftler
<b>Mechanischer Rechner (ab Ende 17. Jh.)</b>	Rechenschemata	Naturwissenschaftler	Technische Rechner
<b>Elektronische Datenverarbeitung (ab Mitte 20. Jh.)</b>	Programme	Naturwissenschaftler	Programmierer

### Fehlerfortpflanzung

- *Ausgangsgrößen* bei der Lösung praktischer Probleme mittels der Mathematik und der Rechentechnik sind *häufig* durch Messungen ermittelte Werte.  
⇒ **Fehlerbehaftete Eingabewerte**, deren Fehlertoleranzen von der verwendeten Messtechnik abhängen (Landvermessung, Elektronenmikroskop).
- *Numerische Verfahren* sind oft Näherungsverfahren.  
⇒ **Verfahrensfehler**, deren Größen von dem verwendeten Algorithmus abhängen.
- *Mathematische Hilfsmittel*, früher mechanische Rechenmaschinen, heute elektronische Rechenanlagen, arbeiten *ungenau*.  
⇒ **Rechenfehler**, diese werden zwar mit den *immer genauer* arbeitenden Computer geringer, sind aber durch die Endlichkeit des Speichers, die Endlichkeit der Zahlendarstellung und die Endlichkeit der Rechenzeit *nicht* zu beseitigen.

Die Lösung eines Problems ist folglich mit *Eingabefehlern*, *Verfahrensfehlern* und *Rechenfehlern* behaftet. Die Korrektheit einer Lösung wird durch **Fehlerfortpflanzung** beeinflusst:

**Fehlerbehaftete Eingabewerten werden mittels fehlerbehafteten Verfahren und ungenauer Rechnung verarbeitet.**



Die **Numerik**, eine Teildisziplin der Mathematik, beschäftigt sich mit *Verfahren zur Lösung mathematischer Probleme*, deren *Fehlerfortpflanzung* und der *Ergebnisauswertung*. Einem numerischen Verfahren liegt stets ein Algorithmus zugrunde, es enthält Kontrollen zur Fehlerfortpflanzung und entscheidet, in welchem Bereich eine Lösungsmethode verwendbare Ergebnisse liefert.

Bei der Auswahl eines Verfahrens sind Rechenzeit und Speicherökonomie zu beachten. Mathematisch elegante Lösungen sind manchmal rechentechnisch ungünstig. Man beurteilt ein numerisches Verfahren *streng nach ökonomischen Gesichtspunkten*:

**Ein verwendetes numerisches Verfahren muss die notwendige Genauigkeit gewähren, aber unnötigen Rechenaufwand vermeiden.**

## 5.2 Funktionen

Viele Algorithmen arbeiten mit reellen Funktionen  $f(x)$  einer Veränderlichen. Da Werteberechnungen von Funktionen sehr oft notwendig sind, sollen verwendete Verfahren möglichst Zeit optimierend gestaltet werden. Nicht alle Funktionen lassen sich immer rechenstechnisch exakt bestimmen. Wir werden hier *mathematisch exakte* Verfahren, welche *optimierende* Gesichtspunkte berücksichtigen, und auch *Näherungsverfahren* zur Funktionsberechnung kennenlernen.

### 5.2.1 Funktionsklassen

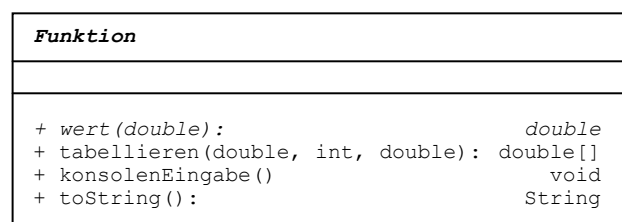
Um den Funktionsbegriff systematisch zu modellieren, werden verschiedene, voneinander abhängige Funktionsklassen entwickelt. Zunächst definieren wir eine Klasse `Funktion` allgemein, d.h. **abstrakt, ohne auf eine spezielle Funktion einzugehen**.

Die wichtigste Fähigkeit einer Funktion ist deren Wertberechnung bei einem gegebenen Argument:

**Eine Funktion ordnet jedem Element  $x$  aus einem Definitionsbereich  $D$  eindeutig ein Element  $y$  aus einem Wertevorrat  $W$  zu.**

Diese Abbildung soll in einer Methode `wert` realisiert werden. Da wir noch keine besonderen Funktionen betrachten, bleibt diese Methode ebenfalls **abstrakt, d.h. ohne auf eine spezielle Funktionswertberechnung einzugehen**. Unter der Voraussetzung, dass es solch eine Methode gibt, kann man eine weitere Methode `tabellieren` zum Erstellen einer Wertetabelle angeben. Schließlich wird eine Methode `toString` die Funktion als Zeichenkette in der mathematisch üblichen Syntax darstellen. Eine Methode `konsolenEingabe` ermöglicht eine Tastatureingabe evtl. vorhandener Attribute.

Das folgende UML-Klassendiagramm beschreibt die Klasse `Funktion`, wobei *abstrakte* Bestandteile (Klassen und Methoden) *kursiv* dargestellt werden.



#### ***Funktion.java***

```
// Funktion.java
```

MM 2010

```
/**
 * Funktion, abstract.
 */
public abstract class Funktion
{
```

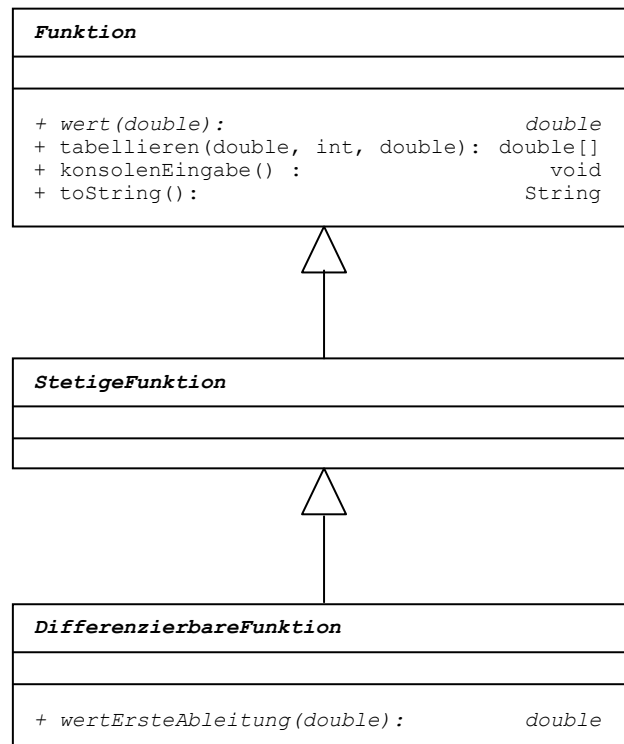
```
/* ----- */
// service-Methoden
/**
 * Berechnen eines Funktionswertes, abstract.
 * @param arg Argument
 * @return Wert der Funktion an der Stelle arg
 */
public abstract double wert( double arg);

/**
 * Tabellieren von Funktionswerten.
 * @param x0 Startwert
 * @param n Anzahl der Werte
 * @param h Schrittweite
 * @return Wertetabelle
 */
public double[] tabellieren
( double x0, int n, double h)
{
    double arg = x0;
    if( n < 1) n = 1;
    if( n == 1) h = 0;
    double[] tabelle = new double[ n];
    for( int i = 0; i < n; i++)
    {
        tabelle[ i] = wert( arg);
        arg += h;
    }
    return tabelle;
}

/**
 * Eingabe von Parametern einer Funktion ueber Konsole,
 * falls welche existieren.
 */
public void konsolenEingabe() {}

/**
 * Darstellen einer Funktion.
 * @return Funktion in linearer Schreibweise
 */
public String toString()
{
    return "";
}
}
```

Spezielle Funktionen sind stetige und stetig differenzierbare Funktionen. Von den letzteren kann die erste Ableitung berechnet werden. Da sie die Funktionalität der Funktionen erweitern, werden sie von der allgemeinen Klasse Funktion abgeleitet.

**StetigeFunktion.java**

// StetigeFunktion.java

MM 2008

```

/**
 * Stetige Funktion, abstract,
 * besitzt stets eine Funktionswert.
 */
public abstract class StetigeFunktion extends Funktion
{
}
  
```

**DifferenzierbareFunktion.java**

// DifferenzierbareFunktion.java

MM 2008

```

/**
 * Differenzierbare Funktion, abstract.
 */
public abstract class DifferenzierbareFunktion
  extends StetigeFunktion
{
  /** ----- */
  // service-Methode

  /**
   * Berechnen der ersten Ableitung, abstract.
   * @param arg Argument
   * @return Wert der ersten Ableitung an der Stelle arg
   */
  public abstract double wertErsteAbleitung
    ( double arg);
}
  
```

## 5.2.2 Beispiel „Lineare Funktionen - Geraden“

Um eine Klasse für eine spezielle Funktion einzuführen, muss man zunächst wissen, ob es sich um eine unstetige, stetige oder stetig differenzierbare Funktion handelt. Oft benötigt man zusätzlich noch Attribute, die die Funktion eindeutig bestimmen. Die Methoden `wert`, `toString` und `konsolenEingabe` sind zu spezifizieren. Die Methode `wertErsteAbleitung` ist im Fall einer differenzierbaren Funktion festzulegen.

Wir führen eine Klasse `Gerade` ein. Diese ist eine stetig differenzierbare Funktion und wird deshalb von der Klasse `DifferenzierbareFunktion` abgeleitet. Eine `Gerade` ist eindeutig durch zwei Werte definiert, dem linearen Glied, hier das Attribut `m`, und dem absoluten Glied, hier das Attribut `n`. Eine Methode `setGerade` setzt deren Werte. Für die Geradenberechnungen sind die Methoden `wert` und `wertErsteAbleitung` festzulegen. Eine Methode `toString` gibt die Gerade als `String` aus, eine Methode `konsolenEingabe` ermöglicht die Tastatureingabe der Attribute.

Gerade	
- m:	double
- n:	double
+ setGerade(double, double):	void
+ konsolenEingabe():	void
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String

### Gerade.java

```
// Gerade.java
import Tools.IO.*;
// MM 2010
// Eingaben

/**
 * Geradenfunktion  $f(x) = mx + n$ ,  $D = W = \mathbb{R}$ .
 */
public class Gerade
    extends DifferenzierbareFunktion
{
    /** ----- */
    // Attribute

    /**
     * Anstieg m.
     */
    private double m = 0;

    /**
     * Absolutglied n.
     */
    private double n = 0;
```

```
/* ----- */
// set-Methoden
/**
 * Setzt Koeffizienten m und n.
 * @param anstieg Anstieg
 * @param absolut Absolutglied
 */
public void setGerade( double anstieg, double absolut)
{
    m = anstieg;
    n = absolut;
}

/**
 * Geradeneingabe ueber Konsole.
 */
public void konsolenEingabe()
{
// Eingabe der Geradenkoeffizienten
    System.out.println( "Geradeneingabe");

    double anstieg
    = IOTools.readDouble( " Anstieg          m = ");

    double absolut
    = IOTools.readDouble( " Absolutglied n = ");

// Setzen der Geradenkoeffizienten
    setGerade( anstieg, absolut);
}

/* ----- */
// service-Methoden
/**
 * Berechnen eines Funktionswertes.
 * @param arg Argument
 * @return f( arg)
 */
public double wert( double arg)
{
    return m * arg + n;
}

/**
 * Berechnen einer ersten Ableitung.
 * @param arg Argument
 * @return Wert der ersten Ableitung an der Stelle arg
 */
public double wertErsteAbleitung( double arg)
{
    return m;
}
```

```

/* ----- */
// toString-Methode
/**
 * Darstellen der Funktion.
 * @return Funktion in linearer Schreibweise
 */
public String toString()
{
    return m + "x + " + n;
}
}

```

Wir betrachten nun zwei Anwendungen für die Klasse Gerade. Im Ersten Programm werden Funktionsberechnungen für Geraden durchgeführt und im zweiten Programm werden Geraden tabelliert.

### **GeradenBerechner.java**

```

// GeradenBerechner.java
import Tools.IO.*;
// Eingaben
MM 2010

/**
 * Berechnet Geraden.
 */
public class GeradenBerechner
{
/**
 * Geradenberechner:
 * Geradeneingabe, Wertberechnung, Ergebnisausgabe.
 */
    public static void main( String[] args)
    {
// Dialog
        char weiter = 'j';
        do
        {
// Neue Gerade
            System.out.println();
            Gerade g = new Gerade();
            g.konsolenEingabe();

// Geradenausgabe
            System.out.println();
            System.out.println( "g( x) = " + g);

            do
            {
// Argumenteingabe
                System.out.println();
                System.out.println( "Wertberechnung");
                double x0 = IOTools.readDouble( " x0 = ");

```

```

// Wertberechnung
    double y0 = g.wert( x0);
    double y1 = g.wertErsteAbleitung( x0);

// Wertausgabe
    System.out.print( " g( " + x0 + ") = " + y0);
    System.out.println( "\t\tg'( " + x0 + ") = " + y1);
    System.out.println();

// Weiter
    weiter = IOTools.readChar( "Neues Argument (j/n)? ");
    } while( weiter == 'j');

    weiter = IOTools.readChar( "Neue Gerade (j/n)? ");
    } while( weiter == 'j');
    }
}

/* ----- */
/*                                     // Testbeispiel
/*
* g( x) = 3.0x + 2.0
*
* Wertberechnung x0 = 3
* g( 3) = 11.0
* g'( 3) = 3.0
*/

```

**GeradenTabulator.java**

```

// GeradenTabulator.java
import Tools.IO.*;
MM 2010
// Eingaben

/**
 * Tabelliert Geraden.
 */
public class GeradenTabulator
{
/**
 * Geradentabulator:
 * Geradeneingabe, Eingabe der Tabellenparameter,
 * Berechnen und Ausgabe der Tabelle.
 */
    public static void main( String[] args)
    {
// Dialog
        char weiter = 'j';
        do
        {
// Neue Gerade
            System.out.println();

```

```
Gerade g = new Gerade();
g.konsolenEingabe();

// Geradenausgabe
System.out.println();
System.out.println( "g( x) = " + g);

do
{
// Eingabe der Tabellenparameter
System.out.println();
System.out.println( "Wertbereich");
double x0
= IOTools.readDouble( " Startargument x0 = ");
int n
= IOTools.readInteger( " Anzahl n (n > 0) = ");
double h = 0;
if( n > 1)
h = IOTools.readDouble( " Schrittweite h = ");

// Wertetabelle
double[] tabelle = g.tabellieren( x0, n, h);

// Tabellenausgabe
double arg = x0;
for( int i = 0; i < tabelle.length; i++)
{
System.out.println
( " g( " + arg + ")\t= " + tabelle[ i]);
arg += h;
}
System.out.println();

// Weiter
weiter = IOTools.readChar( "Neue Tabelle (j/n)? ");
} while( weiter == 'j');

weiter = IOTools.readChar( "Neue Gerade (j/n)? ");
} while( weiter == 'j');
}

/* ----- */
/* // Testbeispiel
* g( x) = 3.0x + 2.0
*
* Wertbereich
* Startargument x0 = 0
* Anzahl n (n > 0) = 5
* Schrittweite h = 0.25
*
*/
```

```
* g( 0.0) = 2.0  
* g( 0.25) = 2.75  
* g( 0.5) = 3.5  
* g( 0.75) = 4.25  
* g( 1.0) = 5.0  
*/
```

### 5.3 Polynomberechnung

Polynome spielen in der Numerik eine besondere Rolle. Oft verwendet man sie als *Interpolationspolynome* zur angenäherten Darstellung einer Funktion. Die Funktion selbst muss nicht bekannt sein. Man benötigt diskret gegebenen Funktionswerte, welche zum Beispiel durch Messwerte ermittelt wurden, legt durch diese Polynome und approximiert damit eine Funktion. Eine Funktionsberechnung kann so näherungsweise mittels Polynomberechnungen ausgeführt werden.

$$P_r(x) = a_r x^r + a_{r-1} x^{r-1} + \dots + a_1 x + a_0 = \sum_{i=0}^r a_i x^i \quad \text{mit } a_r \neq 0$$

Wir behandeln ein Rechenzeit optimierendes Verfahren zur Polynomberechnung.

#### 5.3.1 Hornerschema

##### Grundgedanke

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

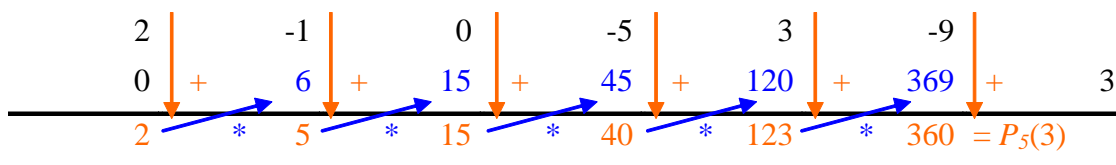
##### Beispiel

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 durch Ausklammern so um, dass keine Potenzen mehr vorhanden sind:

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

Dieses umgestellte Polynom gestattet eine sehr schnelle Berechnung, auch mit dem Taschenrechner.

Das hierfür verwendete *Rechenschema* trägt den Namen **Horner Schema**, nach **William Georg Horner (1786 - 1837)**, und sieht folgendermaßen aus:



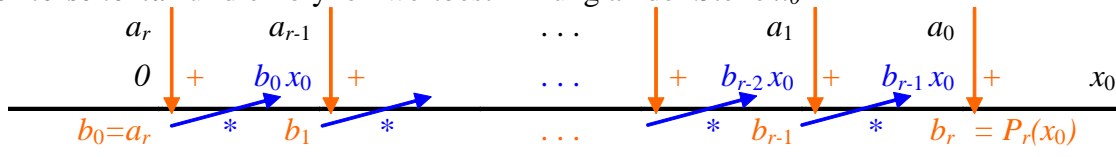
##### Probe

$$P_5(3) = 486 - 81 - 45 + 9 - 9 = 360.$$

##### Allgemein

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

**Horner Schema** für die Polynomwertbestimmung an der Stelle  $x_0$



Durch dieses Verfahren wird der Rechenaufwand wesentlich verringert:

**Polynomberechnung ohne Horner Schema:**

$$\begin{aligned} \# \text{ Mult} &= \sum_{i=1}^r i = \frac{r(r+1)}{2} \\ \# \text{ Add} &= r \\ \# \text{ Mult} + \# \text{ Add} &= \frac{r^2 + 3r}{2} \end{aligned}$$

**Polynomberechnung mit Horner Schema:**

$$\begin{aligned} \# \text{ Mult} &= r \\ \# \text{ Add} &= r && +1 \text{ (Anfangsaddition mit 0)} \\ \# \text{ Mult} + \# \text{ Add} &= 2r && +1 \text{ (Anfangsaddition mit 0)} \end{aligned}$$

**Rechentechische Umsetzung des Horner Schemas**

Ein Polynom ist durch seine Koeffizienten eindeutig bestimmt. Deshalb werden diese als Attribute abgespeichert. Die Wertberechnung erfolgt durch das Horner Schema.

```
private double[] koeffizienten;

public double wert( double arg)
{
    double erg = 0;
    for( int i = koeffizienten.length - 1; i >= 0; i--)
        erg = erg * arg + koeffizienten[ i];
    return erg;
}
```

### 5.3.2 Erweitertes Horner Schema

Aus dem Horner Schema folgt, dass  $b_0 = a_r$ ,  $b_{i+1} = a_{r-i-1} + b_i x_0$  für  $i = 1, \dots, r$  und  $b_r = P_r(x_0)$ . Umgeformt ergibt sich  $a_r = b_0$  und  $a_{r-i-1} = b_{i+1} - b_i x_0$ .

Setze  $j = r - i - 1$ , so ist  $a_j = b_{r-j} - b_{r-j-1} x_0$  und wegen  $a_r = b_0$  und  $b_r = P_r(x_0)$  folgt:

$$\begin{aligned} P_r(x) &= a_r x^r + a_{r-1} x^{r-1} + a_{r-2} x^{r-2} + \dots + a_1 x + a_0 \\ &= \underline{b_0 x^r} + \underline{(b_1 - b_0 x_0) x^{r-1}} + \underline{(b_2 - b_1 x_0) x^{r-2}} + \dots + \underline{(b_{r-1} - b_{r-2} x_0) x} + \underline{(b_r - b_{r-1} x_0)} \\ &= \underline{(x - x_0) b_0 x^{r-1}} + \underline{(x - x_0) b_1 x^{r-2}} + \dots + \underline{(x - x_0) b_{r-2} x} + \underline{(x - x_0) b_{r-1}} + \underline{b_r} \\ &= \underline{(x - x_0) (b_0 x^{r-1} + b_1 x^{r-2} + \dots + b_{r-2} x + b_{r-1})} + \underline{b_r} \end{aligned}$$

$$\Rightarrow P_r(x) = \underline{(x - x_0) (b_0 x^{r-1} + b_1 x^{r-2} + \dots + b_{r-2} x + b_{r-1})} + \underline{P_r(x_0)}$$

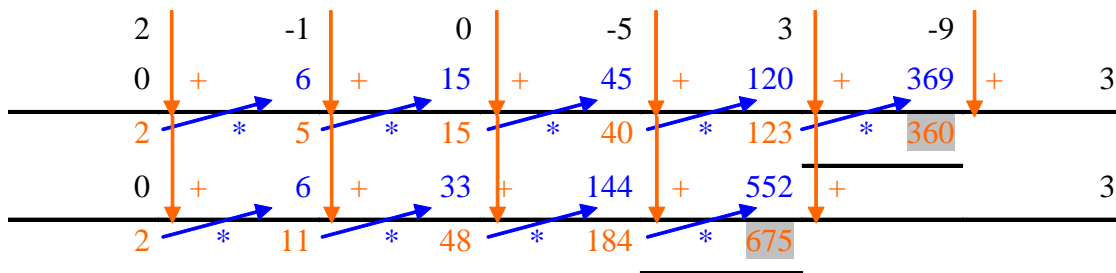
$$\Rightarrow \frac{P_r(x) - P_r(x_0)}{x - x_0} = p_r(x) \text{ mit } p_r(x) = \underline{b_0 x^{r-1} + b_1 x^{r-2} + \dots + b_{r-2} x + b_{r-1}}$$

$\Rightarrow p_r(x)$  kann als Differenzenquotient zur Berechnung der 1. Ableitung an der Stelle  $x = x_0$  genutzt werden:

$$P'(x_0) = \lim_{x \rightarrow x_0} \frac{P(x) - P(x_0)}{x - x_0} \quad \text{mit} \quad P'(x) = b_0 x^{r-1} + b_1 x^{r-2} + \dots + b_{r-2} x + b_{r-1}$$

**Beispiel**

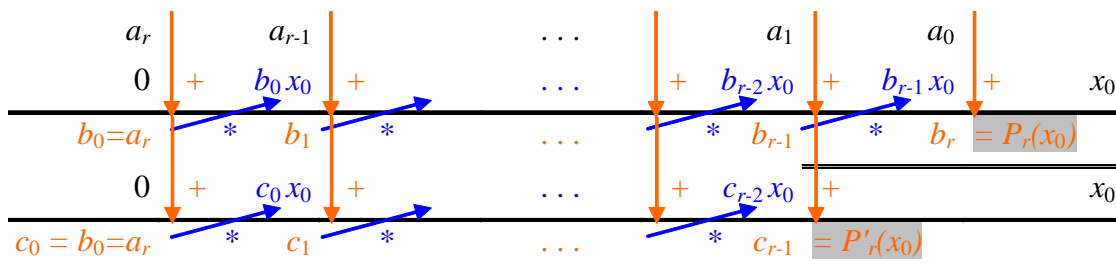
$P_5(x) = 2x^5 - x^4 - 5x^2 + 3x - 9$ , gesucht  $P'_5(3)$ .



**Probe**

$P'_5(x) = 10x^4 - 4x^3 - 10x + 3$ ,  $P'_5(3) = 810 - 108 - 30 + 3 = 675$ .

**Erweitertes Hornerschema** für die Polynomwertbestimmung und für die Berechnung der 1. Ableitung an der Stelle  $x_0$ :



**Rechentechische Umsetzung des erweiterten Hornerschemas**

```
public double wertErsteAbleitung( double arg)
{
    double erg = 0, erg1 = 0;
    for( int i = koeffizienten.length - 1; i > 0; i--)
    {
        erg = erg * arg + koeffizienten[ i];
        erg1 = erg1 * arg + erg;
    }
    return erg1;
}
```

**5.3.3 Klasse Polynom**

Zum Abschluss fassen wir die vollständige Klasse Polynom zusammen. Ein Polynom ist eine differenzierbare Funktion und wird von der Klasse DifferenzierbareFunktion abgeleitet. Es wird eindeutig durch seine Koeffizienten definiert. Diese werden als Feld

abgelegt. Eine Methode `setPolynom` setzt deren Werte. Für die Polynomberechnungen sind die Methoden `wert` und `wertErsteAbleitung` festzulegen. Eine Methode `toString` gibt das Polynom als String aus, eine Methode `konsolenEingabe` ermöglicht die Tastatureingabe der Koeffizienten.

Polynom	
- koeffizienten:	double[]
+ setPolynom(double[]):	void
+ konsolenEingabe():	void
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String

### **Polynom.java**

```
// Polynom.java
import Tools.IO.*;

/**
 * Polynom, Berechnung mit Hornerschema.
 */
public class Polynom
    extends DifferenzierbareFunktion
{
    /* ----- */
    // Attribute

    /**
     * Polynomkoeffizienten.
     */
    private double[] koeffizienten = { 0 };

    /* ----- */
    // set-Methoden

    /**
     * Setzt Polynomkoeffizienten fest.
     * @param koef Koeffizienten des Polynoms
     */
    public void setPolynom( double[] koef)
    {
        koeffizienten = koef;
    }

    /**
     * Polynomeingabe ueber Konsole.
     */
    public void konsolenEingabe()
    {
        // Eingabe der Polynomkoeffizienten
        System.out.println( "Polynomeingabe");
        int grad;
    }
}
MM 2008
// Eingaben
```

```
do
{
    grad = IOTools.readInteger
        ( " Grad des Polynoms (>=0): ");
} while( grad < 0);

double[] koeff = new double[ grad + 1];
for( int i = 0; i < grad + 1; i++)
    koeff[ i] = IOTools.readDouble
        ( " " + i + ". Koeffizient: ");

// Setzen der Polynomkoeffizienten
setPolynom( koeff);
}

/* -----*/
// service-Methoden
/**
 * Berechnen eines Funktionswertes mittels HORNER.
 * @param arg Argument
 * @return Wert der Funktion an der Stelle arg
 */
public double wert( double arg)
{
    double erg = 0;
    for( int i = koeffizienten.length - 1; i >= 0; i--)
        erg = erg * arg + koeffizienten[ i];
    return erg;
}

/**
 * Berechnen einer Ableitung mittels
 * erweiterem HORNER.
 * @param arg Argument
 * @return Wert der ersten Ableitung an der Stelle arg
 */
public double wertErsteAbleitung( double arg)
{
    double erg = 0, erg1 = 0;
    for( int i = koeffizienten.length - 1; i > 0; i--)
    {
        erg = erg * arg + koeffizienten[ i];
        erg1 = erg1 * arg + erg;
    }
    return erg1;
}

/* ----- */
// toString-Methode
/**
 * Darstellung eines Polynoms.
 * @return Polynom in linearer Schreibweise
 */
```

```

*/
public String toString()
{
    String str = "" + koeffizienten[ 0];
    for( int i = 1; i < koeffizienten.length; i++)
        str += " + " + koeffizienten[ i] + " x^" + i;
    return str;
}
}

```

### 5.3.4 Klasse RationaleFunktion

$$f(x) = \frac{P_r(x)}{P_s(x)} = \frac{\sum_{i=0}^r a_i x^i}{\sum_{i=0}^s b_i x^i}$$

Rationale Funktionen unterteilt man in *ganzzonale* ( $s = 0, r > 0$ : *Polynome*) und *gebrochen rationale* ( $s > 0; r < s$  *echte*,  $r \geq s$  *unechte*) Funktionen. Im zweiten Fall sind diese nicht notwendig stetig. Folglich werden sie direkt von der Klasse Funktion abgeleitet. Zähler und Nenner sind Polynome. Diese werden als Attribute aufgenommen. Die set-Methode übergibt diese. Für die Wertberechnung wert, die Methode konsolenEingabe und die Funktionsdarstellung toString werden die entsprechenden Methoden für das Zähler- und Nennerpolynom der Klasse Polynom herangezogen.

RationaleFunktion	
- zaehler:	Polynom
- nenner:	Polynom
+ setRationaleFunktion(Polynom, Polynom):	void
+ konsolenEingabe():	void
+ wert(double):	double
+ toString():	String

#### RationaleFunktion.java

```

// RationaleFunktion.java
/**
 * Rationale Funktion, Polynom / Polynom.
 */
public class RationaleFunktion
    extends Funktion
{
    /* ----- */
    // Attribute
    /**
     * Zaehler.
     */
}

```

MM 2010

```
private Polynom zaehler;

/**
 * Nenner.
 */
private Polynom nenner;

/* ----- */
// set-Methoden

/**
 * Rationale Funktion, setzt Zaehler- und Nennerpolynom.
 * @param z Zaehlerpolynom
 * @param n Nennerpolynom
 */
public void setRationaleFunktion
( Polynom z, Polynom n)
{
    zaehler = z;
    nenner = n;
}

/**
 * Eingabe einer rationalen Funktion ueber Konsule.
 */
public void konsolenEingabe()
{
// Eingabe der Zaehler- und Nennerpolynome
    System.out.println( "Zaehler");
    Polynom zaehler = new Polynom();
    zaehler.konsolenEingabe();

    System.out.println( "Nenner");
    Polynom nenner = new Polynom();
    nenner.konsolenEingabe();

// Setzen des Zaehler- und Nennerpolynoms.
    setRationaleFunktion( zaehler, nenner);
}

/* ----- */
// service-Methoden

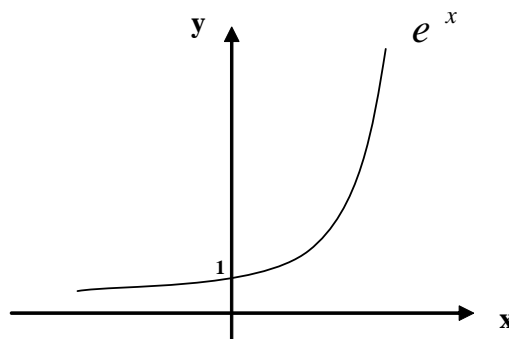
/**
 * Berechnen eines Funktionswertes,
 * muss nicht existieren.
 * @param arg Argument
 * @return Wert der Funktion an der Stelle arg
 */
public double wert( double arg)
{
    return zaehler.wert( arg) / nenner.wert( arg);
}
```

```
/* ----- */
// toString-Methode
/**
 * Darstellung einer rationalen Funktion.
 * @return Funktion in linearer Schreibweise
 */
public String toString()
{
    return "( " + zaehler + " )" + " / " +
           "( " + nenner + " )";
}
}
```

## 5.4 Reihenentwicklungen

Einige in einem Bereich der reellen Zahlen stetig differenzierbare Funktionen lassen sich als Grenzwerte konvergenter Potenzreihen entwickeln, so zum Beispiel Exponentialfunktion und Winkelfunktionen. Da diese Reihen unendlich sind und nur endliche Algorithmen verarbeitet werden. Damit sind *Verfahrensfehler* zu erwarten. Es werden endliche Datendarstellungen verarbeitet. Damit erhalten wir mit *Rechenfehler* belastete Näherungswerte für die Funktionsberechnungen und die Berechnungen müssen geeignet abgebrochen werden. Die dabei entstehenden *Fortpflanzungsfehler* dürfen sich nur unwesentlich auf das Ergebnis auswirken, ansonsten wird eine Fehlerbehandlung notwendig oder das verwendete Verfahren ist unbrauchbar.

### 5.4.1 Exponentialfunktion



Reihenentwicklung: 
$$f(x) = e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Konvergenzbereich:  $|x| < \infty$

#### Algorithmus für eine Reihenentwicklung allgemein

Um die Berechnung zu optimieren, bestimme man den neuen Summanden aus dem alten und addiere ihn zur bis dahin berechneten Summe. Man wiederhole das Aufsummieren solange, bis der neue Summand so klein ist, dass er keinen Beitrag zur Summe bildet, d.h. die Summe sich nicht mehr verändert (*Rechnergenauigkeit*).

#### *Rechentechische Umsetzung der Reihenentwicklung*

```
private double myExp( double x)
{
    double alteSumme, neueSumme = 1, summand = 1;
    int i = 0;

    do // Reihenentwicklung
    {
        i++;
        summand *= x / i; // neuer Summand

        alteSumme = neueSumme; // neue Summe
        neueSumme += summand;

    } while( neueSumme != alteSumme);
    return alteSumme;
}
```

}

In einem Testprogramm wurden einige Funktionswerte mit der Reihenentwicklung berechnet und die Ergebnisse mit der Klassenmethode `exp` der Klasse `Math` verglichen.

Ausschnitt aus der Wertetabelle für  $x \in [-25,0]$ :

The image shows two screenshots of a Windows command prompt window titled "Cmd und Java auf C". The first screenshot shows results for x = 0.0, -0.5, -1.0, and -1.5. The second screenshot shows results for x = -23.5, -24.0, -24.5, and -25.0. In each case, the value of 'myExp' is compared to 'Math.exp'. For negative x, 'Math.exp' returns a value in scientific notation, while 'myExp' returns a decimal value.

x	myExp	Math.exp
0.0	1.0	1.0
-0.5	0.6065306597126333	0.6065306597126334
-1.0	0.36787944117144245	0.36787944117144233
-1.5	0.22313016014842976	0.22313016014842982
-23.5	3.453750578655889E-9	6.224144622907783E-11
-24.0	3.4430535428809986E-7	3.775134544279098E-11
-24.5	-8.280647456316813E-7	2.289734845645553E-11
-25.0	8.181278981020606E-7	1.3887943864964021E-11

### Fehlerbehandlung

Für sehr kleine  $x$  wird der Wert der Funktion negativ. Diese Fehler entstehen als *Rechenfehler und deren Fortpflanzung*, bedingt durch die Endlichkeit der Zahlendarstellung. Es ist eine Ergebniskorrektur notwendig:

$$e^x = \begin{cases} e^x, & x \geq 0 \\ \frac{1}{e^{|x|}}, & x < 0 \end{cases}$$

```
private double myExpBesser( double x)
{
    if( x < 0) return 1 / myExp( -x);
    return myExp( x);
}
```

### 5.4.2 Klasse Exp

Der Aufbau der Klasse entspricht dem der Klasse `Polynom`, wobei Attribute und damit eine `set`-Methode für die Exponentialfunktion überflüssig sind. Die Reihenentwicklung und deren Verbesserung werden in zwei `private`-Methoden vorbereitet.

Die Werteberechnung erfolgt unter Verwendung der Potenzreihe:

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

Exp	
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String
- myExp(double):	double
- myExpBesser(double):	double

**Exp.java**

// Exp.java

MM 2008

```

/**
 * Exponentialfunktion e ^ x.
 */
public class Exp
    extends DifferenzierbareFunktion
{
    /* ----- */
    // service-Methoden

    /**
     * Berechnen eines Funktionswertes f( x) = e ^ x.
     * @param arg Argument
     * @return e ^ arg
     */
    public double wert( double arg)
    {
        return myExpBesser( arg);
    }

    /**
     * Berechnen der ersten Ableitung f'( x) = e ^ x.
     * @param arg Argument
     * @return Wert der ersten Ableitung an der Stelle arg
     */
    public double wertErsteAbleitung( double arg)
    {
        return wert( arg);
    }

    /* ----- */
    // Reihenentwicklung

    /**
     * Exponentialfunktion (Reihenentwicklung):
     * e ^ x = 1 + x + ( x ^ 2) / 2! + ( x ^ 3) / 3! ...
     * @param x Argument
     * @return Wert an der Stelle x nach Reihenentwicklung
     */
    private double myExp( double x)
    {
        double alteSumme, neueSumme = 1, summand = 1;
        int i = 0;
    }
}

```

```

        do                                // Reihenentwicklung
        {
            i++;
            summand *= x / i;              // neuer Summand

            alteSumme = neueSumme;        // neue Summe
            neueSumme += summand;

        } while(( neueSumme != alteSumme)/*&& (i < 10)*/);

        return alteSumme;
    }

/**
 * Verbesserte Exponentialfunktion:
 * Fuer x < 0 berechne e ^ x = 1 / (e ^ -x).
 * @param x Argument
 * @return Wert an der Stelle x nach Reihenentwicklung
 */
    private double myExpBesser( double x)
    {
        if( x < 0) return 1 / myExp( -x);
        return myExp( x);
    }

/* ----- */
// toString-Methode

/**
 * Darstellen der Funktion e.
 * @return Funktion in linearer Schreibweise
 */
    public String toString()
    {
        return "e ^ x";
    }
}

```

Das Testprogramm berechnet die Funktion  $e^x$  für gegebene Argumente  $x$ , vergleicht die Ergebnisse der Reihenentwicklung und der verbesserten Reihenentwicklung mit denen die `Math.exp( x)` liefert.

### ***ExpTest.java***

```

// ExpTest.java                                MM 2010

/**
 * Test der Klasse Exp.
 */
public class ExpTest
{
/**
 * Berechnet fuer gegebene x die Funktion e ^ x,

```

```

* vergleicht Ergebnis der Reihenentwicklung und
* Ergebnis der verbesserten Reihenentwicklung
* mit Ergebnis der Math.exp( x).
*/
public static void main( String[] args)
{
// Neue Funktion
    Exp fkt = new Exp();

// Funktionsausgabe
    System.out.println();
    System.out.println( "f( x) = " + fkt);

// Wertevergleich
    System.out.println();
    System.out.println
    ( "Vergleich von 0 bis -25, Schrittweite -0.5");

    for( double x = 0; x > -25.5; x -= 0.5)
    {
        System.out.println();
        System.out.println( "e ^ " + x);

        System.out.println
        ( "\tmyExp          \t= " + fkt.myExp( x));

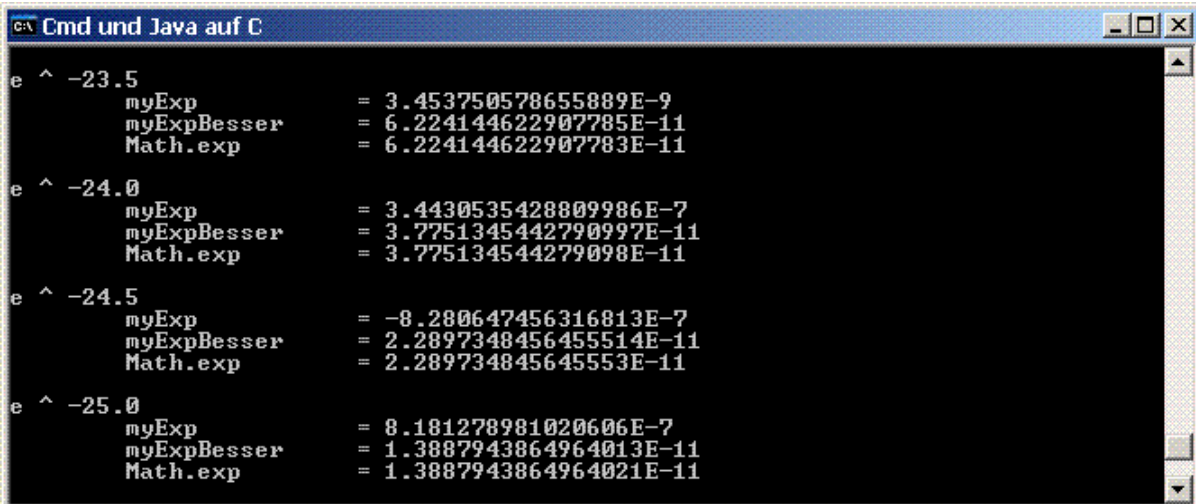
        System.out.println
        ( "\tmyExpBesser\t= " + fkt.myExpBesser( x));

        System.out.println
        ( "\tMath.exp      \t= " + Math.exp( x));
    }

    System.out.println();
}
}

```

Ausschnitt aus der Wertetabelle für  $x \in [-25,0]$ :



```

c:\ Cmd und Java auf C
e ^ -23.5
    myExp          = 3.453750578655889E-9
    myExpBesser    = 6.224144622907785E-11
    Math.exp       = 6.224144622907783E-11
e ^ -24.0
    myExp          = 3.4430535428809986E-7
    myExpBesser    = 3.7751345442790997E-11
    Math.exp       = 3.775134544279098E-11
e ^ -24.5
    myExp          = -8.280647456316813E-7
    myExpBesser    = 2.2897348456455514E-11
    Math.exp       = 2.289734845645553E-11
e ^ -25.0
    myExp          = 8.181278981020606E-7
    myExpBesser    = 1.3887943864964013E-11
    Math.exp       = 1.3887943864964021E-11

```

### 5.4.3 Winkelfunktionen

Reihenentwicklung:  $f(x) = \sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$

Konvergenzbereich:  $|x| < \infty$

#### Rechentechische Umsetzung der Reihenentwicklung

```
private double mySin( double x)
{
    double alteSumme, neueSumme = x, summand = x;
    int j = 1;

    do // Reihenentwicklung
    {
        j++; summand *= -x * x / j; // neuer Summand
        j++; summand /= j;

        alteSumme = neueSumme; // neue Summe
        neueSumme += summand;

    } while( neueSumme != alteSumme);

    return alteSumme;
}
```

Ausschnitt aus der Wertetabelle für  $x = k \cdot \pi$ ,  $k \in [0,20] \Rightarrow \sin(x) = 0$

```

cmd und Java auf C
sin 53.4070751110265
  mySin      = 751926.7425829438
  Math.sin   = -1.5681668835452004E-14

sin 56.5486677646163
  mySin      = -1048387.2294607962
  Math.sin   = 1.911191783433777E-14

sin 59.690260418206094
  mySin      = 3.7338351599725133E8
  Math.sin   = -2.2542166833223536E-14

sin 62.83185307179589
  mySin      = 1.2114966957620123E10
  Math.sin   = 2.59724158321093E-14

sin 65.97344572538569
  mySin      = 2.450148449630589E10
  Math.sin   = -2.9402664830995064E-14
```

#### Fehlerbehandlung

Die Ergebnisse weichen sehr von den Funktionswerten ab:  $\sin(x) \in [-1,1]$ . Auch hier entstehen Fehler in der Berechnung, *Rechenfehler und deren Fortpflanzung*, bedingt durch die Endlichkeit der Zahlendarstellung. Diese fehlerbehaftete Funktionswerte entstehen für sehr große Argumente  $x$  durch deren geringere Darstellungsdichte als Maschinenzahl und lassen sich durch eine Transformation des Wertes  $x$  auf das dichtere Intervall  $x \in \left[0, \frac{\pi}{2}\right]$  beheben.

```

private double mySinBesser( double x)
{
    int s = 1; // Vorzeichen
// x < 0
    if( x < 0)
    {
        x = -x;
        s = -s;
    }
// x > PI
    while( x > Math.PI)
    {
        x -= Math.PI;
        s = -s; // Vorzeichenwechsel
    }
// x > PI/2
    if( x > Math.PI/2) x = Math.PI - x;

    return s * mySin( x);
}

```

Ausschnitt aus der Wertetabelle für  $x = k \cdot \pi$ ,  $k \in [0,20] \Rightarrow \sin(x) = 0$

```

c:\ Cmd und Java auf C
sin 56.5486677646163
  mySin          = -1048387.2294607962
  mySinBesser    = -3.552713678800501E-15
  Math.sin       = 1.911191783433777E-14

sin 59.690260418206094
  mySin          = 3.7338351599725133E8
  mySinBesser    = 3.552713678800501E-15
  Math.sin       = -2.2542166833223536E-14

sin 62.83185307179589
  mySin          = 1.2114966957620123E10
  mySinBesser    = -3.552713678800501E-15
  Math.sin       = 2.59724158321093E-14

sin 65.97344572538569
  mySin          = 2.450148449630589E10
  mySinBesser    = 3.552713678800501E-15
  Math.sin       = -2.9402664830995064E-14

```

*Weitere trigonometrische Funktionen* ließen sich analog durch ihre Reihenentwicklungen berechnen. Besser ist die Anwendung von ausgewählten Grundbeziehungen zwischen den Winkelfunktionen, wie zum Beispiel

$$\cos(x) = \sin\left(\frac{\pi}{2} + x\right) \text{ für } |x| < \infty,$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \text{ für } |x| < \frac{\pi}{2} \text{ und}$$

$$\cot(x) = \frac{\cos(x)}{\sin(x)} \text{ für } 0 < x < \pi.$$

### 5.4.4 Klasse Sinus

Die Werteberechnung erfolgt unter Verwendung der Potenzreihe:  $\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$

Sinus	
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String
- mySin(double):	double
- mySinBesser(double):	double

#### Sinus.java

// Sinus.java

MM 2010

```

/**
 * Sinusfunktion.
 */
public class Sinus
    extends DifferenzierbareFunktion
{
    /* ----- */
    /* service-Methode
    /**
     * Berechnen eines Funktionswertes f( x) = sin( x).
     * @param arg Argument
     * @return sin( arg)
     */
    public double wert( double arg)
    {
        return mySinBesser( arg);
    }

    /**
     * Berechnen der ersten Ableitung
     * f'(x) = cos( x) = sin( PI/2 + x).
     * @param arg Argument
     * @return Wert der ersten Ableitung an der Stelle arg
     */
    public double wertErsteAbleitung( double arg)
    {
        return wert( Math.PI / 2 + arg);
    }

    /* ----- */
    /* Reihenentwicklung
    /**
     * Sinus-Funktion (Reihenentwicklung):
     * sin x = x - ( x ^ 3) / 3! + ( x ^ 5) / 5! ...
     * @param x Argument
     * @return Wert an der Stelle x nach Reihenentwicklung
     */

```

```
private double mySin( double x)
{
    double alteSumme, neueSumme = x, summand = x;
    int j = 1;

    do                                // Reihenentwicklung
    {
        j++; summand *= -x * x / j;    // neuer Summand
        j++; summand /= j;

        alteSumme = neueSumme;        // neue Summe
        neueSumme += summand;
    } while(( neueSumme != alteSumme)/*&& ( j < 10)*/);

    return alteSumme;
}

/**
 * Verbesserte Sinusfunktion:
 * Transformation von x auf den Wertebereich [0,PI/2].
 * @param x Argument
 * @return Wert an der Stelle x nach Reihenentwicklung
 */
private double mySinBesser( double x)
{
    int s = 1;                        // Vorzeichen

    // x < 0
    if( x < 0)
    {
        x = -x;
        s = -s;
    }

    // x > PI
    while( x > Math.PI)               // 1. Variante
    {
        x -= Math.PI;
        s = -s;                        // Vorzeichenwechsel
    }

    /**
     * if( x > Math.PI)                 // 2. Variante
     * { int temp = ( int)( x / Math.PI);
     *   if( temp%2 !=0) s = -s;
     *   x = x - temp * Math.PI;
     * }
     */

    // x > PI/2
    if( x > Math.PI/2) x = Math.PI - x;

    return s * mySin( x);
}
```

```

/* ----- */
// toString-Methode
/**
 * Darstellen der Funktion sin.
 * @return Funktion in linearer Schreibweise
 */
public String toString()
{
    return "sin( x)";
}
}

```

### 5.4.5 Klasse Cosinus

Die Wertberechnung erfolgt unter Verwendung der Klasse Sinus und der Beziehung:

$$\cos(x) = \sin\left(\frac{\pi}{2} + x\right)$$

Cosinus	
+ wert(double):	double
+ wertErsteAbleitung(double):	double
+ toString():	String

#### *Cosinus.java*

```
// Cosinus.java
```

MM 2008

```

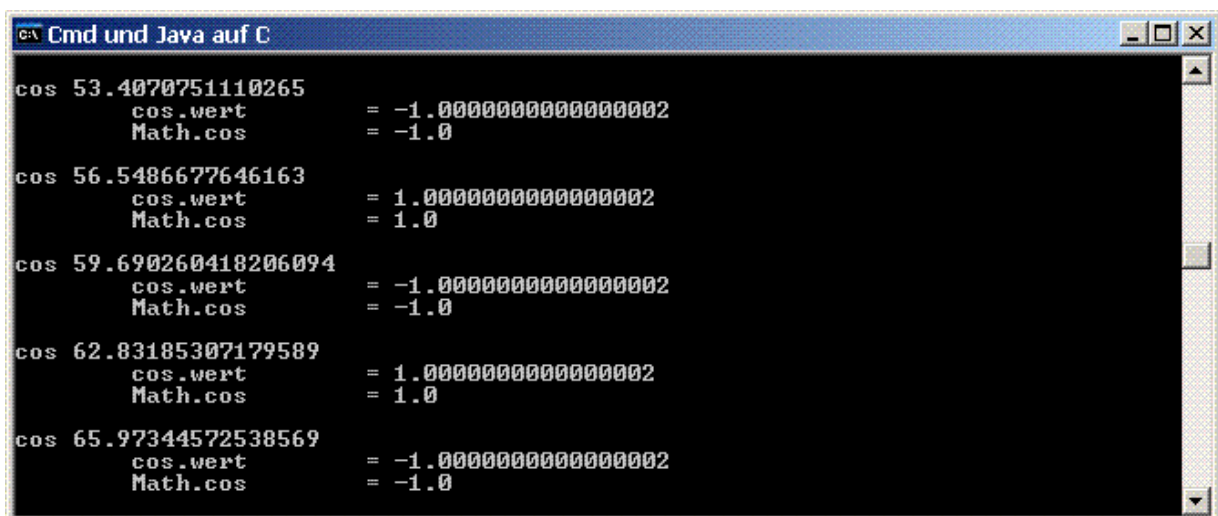
/**
 * Cosinusfunktion.
 */
public class Cosinus
    extends DifferenzierbareFunktion
{
    /* ----- */
    // service-Methode
    /**
     * Berechnen eines Funktionswertes
     * f( x) = cos( x) = sin( PI/2 + x).
     * @param arg Argument
     * @return cos( arg)
     */
    public double wert( double arg)
    {
        Sinus sin = new Sinus();
        return sin.wert( Math.PI/2 + arg);
    }
}
/**

```

```
* Berechnen der ersten Ableitung f'(x) = -sin(x).
* @param arg Argument
* @return Wert der ersten Ableitung an der Stelle arg
*/
public double wertErsteAbleitung( double arg)
{
    Sinus sin = new Sinus();
    return -sin.wert( arg);
}

/* ----- */
// toString-Methode
/**
 * Darstellen der Funktion cos.
 * @return Funktion in linearer Schreibweise
 */
public String toString()
{
    return "cos( x)";
}
}
```

Ausschnitt aus der Wertetabelle für  $x = k \cdot \pi$ ,  $k \in [0,20] \Rightarrow \cos(x) = \pm 1$



```
Cmd und Java auf C
cos 53.4070751110265
    cos.wert      = -1.0000000000000002
    Math.cos     = -1.0

cos 56.5486677646163
    cos.wert      = 1.0000000000000002
    Math.cos     = 1.0

cos 59.690260418206094
    cos.wert      = -1.0000000000000002
    Math.cos     = -1.0

cos 62.83185307179589
    cos.wert      = 1.0000000000000002
    Math.cos     = 1.0

cos 65.97344572538569
    cos.wert      = -1.0000000000000002
    Math.cos     = -1.0
```