

ComputerMathematik - Einführung in Maple / Teil V

Winfried Auzinger (Mai 2009)

Siehe auch MIPG, Kap. 7,8,10

Schwerpunkte: I/O, Debugging, Bibliotheken, Modules, Maplets

- Abschnitt 1, 2 sind praxisrelevant für jeden Maple-Benutzer
- Restliche Abschnitte: Hinweise für die Entwicklung professioneller Maple-Software

Teil V ist ein 'Schnellkurs' bzw. 'Kurzreferenz' - genaue Details in den einschlägigen Hilfeseiten.

1. Ein/Ausgabe, Dateihandling

1.1 Ausgabe mit `print`, `lprint`

* `print` für simple unformatierte Ausgabe von Objekten:

```
> print(a,b,1/3,Int(x^2,x=a..b));
```

$$a, b, \frac{1}{3}, \int_a^b x^2 dx$$

```
> t:=table([Mo=Montag,Di=Dienstag ]):
```

```
> t; # last name evaluation!
```

t

```
> print(t);
```

table([Mo = Montag, Di = Dienstag])

```
> p:=proc(x) x^2 end proc:
```

```
> p; # last name evaluation!
```

p

```
> print(p);
```

proc(x) x^2 end proc

* `lprint` für 'line print' (unformatiert, linear):

```
> lprint(a,b,1/3,Int(x^2,x=a..b));
```

```
a, b, 1/3, Int(x^2, x = a .. b)
```

1.2 Formatierte Ausgabe mit `printf` und Verwandten

Syntax eng verwandt zu C bzw. MATLAB, aber es gibt auch 'algebraische' Formatspezifikation (z.B. `%a`)

```
> ? printf
```

Einige Beispiele:

```
> a:=1.2345: printf("Wert von a: %a = %10.2g \n",'a',a);
```

```
Wert von a: a = 1.2
```

```
> printf("%10.2e \n",a);
```

```

1.23e+00
> printf("%a",1/3);
1/3
... etc.

* sprintf schreibt auf String:
> AusgabeString:=sprintf("Hallo, ich bin %a String",1);
AusgabeString := "Hallo, ich bin 1 String"

* fprintf schreibt auf Datei (muss geöffnet sein, siehe unten):
> fd := fopen("my_file",WRITE);
q:=fprintf(fd,"x = %d, y = %g",1,2); # gibt # der geschriebenen
Zeichen zurück
fclose(fd);

fd := 1
q := 12

> fd := fopen("my_file",READ);
text:=readline(fd); # siehe unten
fd := 1
text := "x = 1, y = 2"

> fclose(fd);

```

▼ 1.3 Interaktiver Input (z.B. für Prozeduren)

* Dafür gibt es **readline(terminal)**:
readline gibt einen **String** zurück, den wir mit **sscanf** in gewünschtem Format 'einscannen' können. Achtung: **sscanf** gibt **Liste** zurück.

```

> p:=proc()
> local x;
> printf("Bitte x eingeben \n");
> x:=readline(terminal);
x:=op(1,sscanf(x,"%a")); # Konvertierung
> printf("Der Wert von x^2 lautet %a",x^2);
> end proc:
> p(); # gebe Wert ein (4)
Bitte x eingeben
Der Wert von x^2 lautet 16

```

▼ 1.4 Arbeiten mit externen Dateien

```

> currentdir(kernelopts(homedir)); # entspricht pwd
"C:\Dokumente und Einstellungen\winfried\Eigene Dateien\act\L V A\CompMath VL 106.080\Maple"

> currentdir("C:/"); # gibt immer das ZUVOR gültige zurück!
"C:\Dokumente und Einstellungen\winfried"

> currentdir();
"C:\\"

```

Die Maple-spezifischen Dateitypen sind:

* **Worksheet**-Dateien (.mws bzw. .mw)

* **'Language'**-Dateien = Textdateien mit gültigen Maple-Befehlen (oft Prozeduren), erstellt etwa mit einem Text-Editor oder mit **save** :

```
> a:=1;
```

```
a := 1
```

```
> p:=proc() print("Hello World") end proc;
```

```
p := proc( ) print("Hello World") end proc
```

```
> save a,p,"my_file.txt";
```

```
> read "my_file.txt";
```

```
a := 1
```

```
p := proc( ) print("Hello World") end proc
```

```
> restart;
```

* **'Daten'-Dateien** für [un](formatierte) numerische Daten oder Text, typischerweise mittels **readline**, **fscanf** bzw. **writeline**, **fprintf**

```
> currentdir();
```

```
"C:"
```

```
> fd:=fopen("my_file.txt",WRITE);
```

```
fd := 0
```

```
> writeline(fd,"Ausgabestring1","Ausgabestring2");
```

```
30
```

```
> fclose(fd);
```

```
> for i from 1 to 2 do String[i]:=readline("my_file.txt") end do;
```

```
String1 := "Ausgabestring1"
```

```
String2 := "Ausgabestring2"
```

AUCH: **readdata**, **writedata** für komplexere numerische Daten (Arrays etc.)

▼ 2. Debugging und kontrollierte Fehlerbehandlung

```
> restart;
```

▼ 2.1 Prozedurverfolgung und Debugger

Die Systemvariable **printlevel** (Voreinstellung: 1) steuert, wieviel Information über den internen Ablauf einer Prozedur ausgegeben wird.

Für eine Verfolgung des Ablaufes kann es sinnvoll sein, schrittweise den Wert von **printlevel** etwa um 5 zu erhöhen.

```
> ? printlevel
```

```
> printlevel;
```

```
1
```

-- Einfaches Beispiel:

```
> geom:=proc(x,n)
```

```
description "Symbolische Berechnung einer geometrischen Reihe",  
"Auswertung für gegebenes x";
```

```
> local series,i,N,X;
```

```
> series:=sum('X^i','i'=0..N);
```

```

> return (subs(X=x,N=n,series));
end proc:
> geom(2,n);

$$2^{n+1} - 1$$

> printlevel:=5; geom(2,n);
printlevel:= 5
{--> enter geom, args = 2, n
series :=  $\frac{X^{N+1}}{X-1} - \frac{1}{X-1}$ 
<-- exit geom (now at top level) = `+`(`^`(2, `+`(n, 1)), `-`(1))}

$$2^{n+1} - 1$$

> printlevel:=10: geom(2,n);
{--> enter geom, args = 2, n
{--> enter sum, args = `^`(X, i), i = 0 .. N
subsIndexed := { }
<-- exit sum (now in geom) = `+`(`/`(`*`(`^`(X, `+`(N, 1))), `*`(`+`(X, `-`(1)))),
`-`(`/`1, `*`(`+`(X, `-`(1))))})}
series :=  $\frac{X^{N+1}}{X-1} - \frac{1}{X-1}$ 
<-- exit geom (now at top level) = `+`(`^`(2, `+`(n, 1)), `-`(1))}

$$2^{n+1} - 1$$

> printlevel:=20: geom(2,n);
{--> enter geom, args = 2, n
{--> enter sum, args = `^`(X, i), i = 0 .. N
{--> enter Preprocess, args = `^`(X, i), i = 0 .. N
summand :=  $X^i$ 
ind := i = 0..N
ind, rng := i, 0..N
a, b := 0, N
ans := [ $X^i$ , i, 0, N]
has_diffs := false
[[ $X^i$ , i, 0, N], false]
<-- exit Preprocess (now in sum) = [[`^`(X, i), i, 0, N], false]
input := [[ $X^i$ , i, 0, N], false]
subsIndexed := { }
{--> enter SumTools:-DefiniteSum:-ClosedForm, args = [[`^`(X, i), i, 0, N], false]
input, has_diffs := [ $X^i$ , i, 0, N], false
s, x :=  $X^i$ , i
rof := false
cf :=  $\frac{X^{N+1}}{X-1} - \frac{1}{X-1}$ 

$$\frac{X^{N+1}}{X-1} - \frac{1}{X-1}$$


```

```
<-- exit SumTools:-DefiniteSum:-ClosedForm (now in sum) = `+`(`/`(`*`(`^`(X, `+`(N, 1))), `*`(`+`(X, `-(1))))), `-(`/`(1, `*`(`+`(X, `-(1))))))})
```

$$\text{output} := \frac{X^{N+1}}{X-1} - \frac{1}{X-1}$$

$$\frac{X^{N+1}}{X-1} - \frac{1}{X-1}$$

```
<-- exit sum (now in geom) = `+`(`/`(`*`(`^`(X, `+`(N, 1))), `*`(`+`(X, `-(1))))), `-(`/`(1, `*`(`+`(X, `-(1))))))})
```

$$\text{series} := \frac{X^{N+1}}{X-1} - \frac{1}{X-1}$$

```
<-- exit geom (now at top level) = `+`(`^`(2, `+`(n, 1)), `-(1))}
2^{n+1} - 1
```

```
> printlevel:=1;
```

```
printlevel:=1
```

Laufzeit-Fehlermeldungen bei der Ausführung von Prozeduren liefern im Allgemeinen keinen Hinweis darauf, wo der Fehler genau aufgetreten ist.

```
> geom(1,n);
```

```
Error, (in geom) numeric exception: division by zero
```

```
> printlevel:=5: geom(1,n);
```

```
{--> enter geom, args = 1, n
```

$$\text{series} := \frac{X^{N+1}}{X-1} - \frac{1}{X-1}$$

```
<-- ERROR in geom (now at top level) = numeric exception: division by zero}
geom called with arguments: 1, n
```

```
 #(geom,2): return subs(X = x,N = n,series)
```

```
Error, (in geom) numeric exception: division by zero
```

```
locals defined as: series = `+`(`/`(`*`(`^`(X, `+`(N, 1))), `*`(`+`(X, `-(1))))), `-(`/`(1, `*`(`+`(X, `-(1))))))}, i = i, N = N, X = X
```

```
> # printlevel:=100: geom(1,n); # auch interessant, aber lieber nicht...
```

```
> printlevel:=1;
```

```
printlevel:=1
```

* Offenbar ein Fehler in Zeile 2, aber was ist genau passiert?

Aktivierung von **trace(procname)** wirkt ähnlich wie printlevel:

```
> trace(geom);
```

```
geom
```

```
> geom(1,n);
```

```
{--> enter geom, args = 1, n
```

$$\text{series} := \frac{X^{N+1}}{X-1} - \frac{1}{X-1}$$

```
<-- ERROR in geom (now at top level) = numeric exception: division by zero}
```

```
Error, (in geom) numeric exception: division by zero
```

```
> untrace(geom);
```

```
geom
```

*** JETZT: SYSTEMATISCHES, GEZIELTES DEBUGGING**

Dazu gibt es den GRAFISCHEN DEBUGGER (wird in eigenem

Fenster geöffnet, siehe VO). Hier protokollieren wir die Debug-Befehle mit, wie sie im klassischen 'command line debugger' (classic workshett, DBG>) aussehen würden.

Für das Debugging nummerieren wir zunächst die Zeilen der Prozedur:

```
> showstat(geom);
```

```
geom := proc(x, n)
local series, i, N, X;
  1  series := sum('X^i', ('i') = 0 .. N);
  2  return subs(X = x, N = n, series)
end proc
```

Wir setzen einen **Breakpoint** am Beginn der Prozedur...

```
> stopat(geom);
```

```
[geom]
```

... und starten die Prozedur nochmals. Damit wird die Ausführung der Prozedur am Beginn gestoppt, und der Debugger wird aktiviert (**EIGENES FENSTER**, bzw. Prompt **DBG>**):

```
> geom(1,n):
```

```
DBG> showstat # aktuellen Status abfragen (*=breakpoint, !=
aktuelle Position)
```

```
DBG> next # 1 Schritt ausführen, nächste Zeile anzeigen
```

```
X^(N+1)/(X-1)-1/(X-1)
geom:
  2  return subs(X = x, N = n, series)
```

```
DBG> series,x,n # Variablenwerte abfragen
```

```
X^(N+1)/(X-1)-1/(X-1),
1,
n
geom:
  2  return subs(X = x, N = n, series)
```

```
DBG> next
```

Der Fehler ist offenbar in Zeile 2 aufgetreten. Aus dem zuvor angezeigten Ergebnis sehen wir, dass durch $x-1=0$ dividiert wurde (sum lieferte Formel für die endliche geometrische Reihe!)

```
> unstopat(geom); # entfernt alle aktiven breakpoints in geom
[ ]
```

```
> geom(1,n);
```

```
Error, (in geom) numeric exception: division by zero
```

Mittels **stoperror** kann man einen 'Watchpoint' setzen und damit bewirken, dass im Fehlerfall angehalten und *automatisch* in den Debug-Modus übergegangen wird (**sehr praktisch !**):

```
> stoperror(`all`);
```

```
[all]
```

```
> geom(1,d);
```

```
DBG> quit
```

```
> ? stoperror
```

Oder auch gezielt auf individuelle Fehlermeldungen hin

```
> unstoperror(): # neue Initialisierung des stoperror-Mechanismus  
stoperror("fehler was weiss ich");  
    ["fehler was weiss ich"]
```

```
> geom(1,n);
```

```
Error, (in geom) numeric exception: division by zero
```

```
> unstoperror():  
stoperror("numeric exception: division by zero");  
    ["numeric exception: division by zero"]
```

```
> geom(1,n);
```

```
DBG> quit
```

* Weitere wichtige Debugging-Befehle:

next ... 1 Zeile ausführen

cont ... setzt Prozedur fort bis zum Ende oder bis zum nächsten Breakpoint

step ... wie next, aber führt ganze Blöcke auf einmal aus (z.B. if ... end if)

stopat(procname,N) ... setze (unbedingten) Breakpoint an Zeile N

unstopat(procname[,N]) ... lösche alle oder spezifizierten Breakpoint

showstat ... aktuellen Status anzeigen

ZUWEISUNG von Werten an Variablen unter dem **DBG>** prompt

quit ... DBG beenden

```
> restart;
```

Weitere Techniken:

* Setzen eines **Watchpoints für eine Variable** mittels **stopwhen**:

Unterbricht Ausführung, sobald sich der **Wert der spezifizierten Variablen ändert**
(deaktivieren mit **unstopwhen**)

Artifizielles Beispiel: Angenommen, gerade Werte der lokalen Variablen n dürfen nicht auftreten. Wir definieren eine Boole'sche Hilfsvariable, die in diesem Fall den Wert true annimmt (initialisiert mit false)

```
> p:=proc()  
local i,n,watch;  
watch:=false;  
for i from 1 to 100 do  
    n:=rand(); # Zufalls-Integer  
    watch:=is(n,even);  
end do;  
printf("Fertig");  
end proc:  
  
> stopwhen(p,watch);  
    [[p, watch]]  
  
> showstat(p);
```

```
p := proc()
```

```
local i, n, watch;
```

```
1 watch := false;
```

```
2 for i to 100 do
```

```
3   n := rand();
```

```
4   watch := is(n,even)
```

```
end do;
```

```
5 printf("Fertig")
```

```
end proc
```

```
> p();
```

```
DBG> cont
```

```
watch := true
```

```
p:
```

```
3      n := rand();
```

```
DBG> n;
```

```
395718860534
```

```
p:
```

```
3      n := rand();
```

```
DBG> unstopwhen
```

```
[]
```

```
p:
```

```
3      n := rand();
```

```
DBG> cont
```

```
Fertig
```

```
> p();
```

```
Fertig
```

* **WEITERE TECHNIK:** Debug-Befehle a-priori in Source-Code einbauen:

* Explizites, festes Setzen von Breakpoints im Source-Code mittels

DEBUG() ... fixer Breakpoint

DEBUG(condition) ... breakpoint nur unter spezifizierter Bedingung aktiv

DEBUG(expression) ... wie **DEBUG()**, expression wird ausgegeben

Beispiel:

```
> p:=proc()
```

```
local i;
```

```
for i from 1 to 5 do
```

```
  i;
```

```
  DEBUG(is(i,prime));
```

```
end do;
```

```
DEBUG("Jetzt ist die Schleife zu Ende. Aktueller Wert von i:",i)
```

```
;
```

```
printf("Fertig");
```

```
end proc:
```

```
> p();
```

```
DBG> cont
```

```
3
```

```
p:
```

```
2      i;
```

```
DBG> cont
```

```
5
```

```
p:
```

```
4      DEBUG("Jetzt ist die Schleife zu Ende. Aktueller Wert von i:",i);
```



```

DBG> cont
"Jetzt ist die Schleife zu Ende. Aktueller Wert von i:",
6
p:
5   printf("Fertig")
DBG> cont
Fertig

```

2.2. Kontrollierte Fehlerbehandlung

Bereits bekannt: **error(...)** provoziert eine 'Exception' und bricht die Prozedur ab.

Allgemeinere Syntax gemäß folgendem Beispiel (irgendwie ähnlich wie bei printf):

```

> restart; with(LinearAlgebra):
> p:=proc(x)
  if (abs(x)<10^(-100)) then
    error "Division durch so eine kleine Zahl ist aber
verdächtig: %1",x;
  else
    return 1/x;
  end if;
end proc:
> p(1E-101);
Error, (in p) Division durch so eine kleine Zahl ist aber verdächtig: 0.1e-100

```

-- '**Kontrollierte Fehlerbehandlung**' bedeutet, dass man mit geeigneten Programmierkonstrukten auf Exceptions ('Ausnahmesituationen') in systematischer Weise reagiert.

Dies benötigt man insbesondere in Situationen, wenn *nicht (genau) vorhersehbar* ist, ob bzw. an welcher Stelle eine Ausnahmesituation (z.B. Division durch 0) eintritt, etwa weil dies in (komplizierterer) Weise von den Eingabedaten abhängt.

Wie geht man damit um: Man deklariert einen Block von Anweisungen als '**verdächtig**' (suspicious) und lässt den Algorithmus 'probeweise' durchlaufen. Fehler werden in der **Mausefalle** gefangen ('trapping errors').

- Falls O.K. (keine Exception): na gut...
- Falls nicht O.K. (Exception): Spezifiziere, was dann zu geschehen hat...

... sogenannter **try ... catch** Mechanismus

-- Beispiel:

Gauß-Elimination ohne Pivot-Suche: Hier ist im Allgemeinen nicht a priori feststellbar, ob eine Division durch 0 droht.

Wir programmieren das mittels Zeilenoperationen aus LinearAlgebra (siehe ? RowOperation) und try .. catch . Die Umformungen werden nur an einer gegebenen Matrix durchgeführt (Umwandlung in obere Dreiecksmatrix)

Bemerkung: Hier ist die 'kritische Stelle' klar (potentielle Division durch 0)

und könnte auch 'manuell' abgefangen werden. try ... catch ist dann erst wirklich notwendig und sehr praktisch, wenn eine komplexere Situation mit verschiedenen potentiellen Ausnahmesituationen vorliegt, die a priori nicht klar lokalisierbar sind.

Damit kann man anstatt einzelner Codezeilen ganze Codeblöcke einer Beobachtung unterziehen.

```
> gauselim:=proc(A::Matrix)
  uses LinearAlgebra;
  local i,j,n,pivot;
  n:=RowDimension(A);
  try # try-Block ('erwünschter' Algorithmus)
    for i from 1 to n-1 do
      pivot:=A[i,i];
      for j from i+1 to n do
        RowOperation(A,[j,i],[-A[j,i]/pivot,inplace=true);
      end do;
    end do;
    printf("try-Block erfolgreich ausgeführt.\n");
    return simplify(A);
  catch: # Mausefalle - hier nur : erlaubt
    printf("try-Block gescheitert.\n");
    return A; # gebe aktuellen Status der Matrix
              # bei Auftreten des Fehlers zurück
              # hier kann man ggf. alternativen Algorithmus angeben!!
              #
  finally # Dieser (optional angebbare) Block wird immer
  ausgeführt
    printf("finally - Block ausgeführt.\n");
  end try;
end proc;
```

```
> A:=Matrix(3,3,symbol=a);
```

$$A := \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

```
> gauselim(A);
```

try-Block erfolgreich ausgeführt.

finally - Block ausgeführt.

$$\left[\left[a_{1,1}, a_{1,2}, a_{1,3} \right], \right.$$

$$\left[0, \frac{a_{2,2} a_{1,1} - a_{2,1} a_{1,2}}{a_{1,1}}, \frac{a_{2,3} a_{1,1} - a_{2,1} a_{1,3}}{a_{1,1}} \right],$$

$$\left[0, 0, \right.$$

$$\left. \frac{a_{3,3} a_{1,1} a_{2,2} - a_{3,3} a_{2,1} a_{1,2} - a_{3,1} a_{1,3} a_{2,2} - a_{3,2} a_{1,1} a_{2,3} + a_{3,2} a_{2,1} a_{1,3} + a_{3,1} a_{1,2} a_{2,3}}{a_{2,2} a_{1,1} - a_{2,1} a_{1,2}} \right]$$

```
> A:=Matrix([[1,2,3],[1,4,5],[1,7,9]]); Determinant(A);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 1 & 4 & 5 \\ 1 & 7 & 9 \end{bmatrix}$$

2

```
> gauselim(A);
```

```
try-Block erfolgreich ausgeführt.
```

```
finally - Block ausgeführt.
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> A:=Matrix([[0,2,3],[2,4,5],[1,7,9]]); Determinant(A);
```

$$A := \begin{bmatrix} 0 & 2 & 3 \\ 2 & 4 & 5 \\ 1 & 7 & 9 \end{bmatrix}$$

4

```
> gauselim(A);
```

```
try-Block gescheitert.
```

```
finally - Block ausgeführt.
```

$$\begin{bmatrix} 0 & 2 & 3 \\ 2 & 4 & 5 \\ 1 & 7 & 9 \end{bmatrix}$$

-- Varianten von **catch**: Individuellere Fehlerbehandlung (vgl. MIPG, Kap.8):

Z.B.: catch reagiert nur auf **individuelle** Exception (ähnlich wie oben bei **stoperror**)

```
> p:=proc()
```

```
  local fd;
```

```
  try
```

```
    fd:=fopen("filename",READ)
```

```
  catch "file or directory does not exist":
```

```
    printf("Datei existiert nicht");
```

```
  end try;
```

```
end proc:
```

```
> p();
```

```
Datei existiert nicht
```

... **Sieht in konkreten Anwendungen typischerweise so aus, dass im try-Block eine Prozedur aufgerufen wird, die ggf. mit error eine Exception zurückgibt, und catch reagiert darauf!**

Demo-Beispiel: Falls die try-Prozedur (p1 aufgerufen von p2) versagt, wird von p2 infinity zurückgegeben (Division durch 0)

```
> p1:=proc(n)
```

```
  if is(n,odd) then
```

```

        error "n muss gerade sein";
    else
        return 1/(1-(n mod 2));
    end if;
end proc:
> p2:=proc(n) # ruft p1 auf
    local x;
    try
        x:=p1(n);
    catch "n muss gerade sein":
        x:=infinity;
    finally
        return x;
    end try;
end proc:
> p2(0);
1
> p2(1);
∞

```

3. Erstellen eines eigenen Packages, Abspeichern in Bibliothek (Beispiel)

-- Ein **package** ist im Wesentlichen eine **Tabelle von Prozeduren**.

Wir schreiben ein einfaches Package, das zwei Matrixoperationen implementiert:

```

> Commutator:=proc(A::Matrix,B::Matrix)
    uses LinearAlgebra;
    return simplify(A.B-B.A);
end proc;
    Commutator := proc(A:Matrix, B:Matrix) return simplify(`.`(A, B) - `.`(B, A)) end proc
> IsCommuting:=proc(A::Matrix,B::Matrix)
    uses LinearAlgebra;
    local n;
    n:=RowDimension(A);
    return Equal(Commutator(A,B),ZeroMatrix(n));
end proc;
IsCommuting := proc(A:Matrix, B:Matrix)
    local n;
    n := LinearAlgebra:-RowDimension(A);
    return LinearAlgebra:-Equal(Commutator(A, B), LinearAlgebra:-ZeroMatrix(n))
end proc
> A:=Matrix([[1,2],[3,4]]); B:=Matrix([[5,6],[7,8]]);

```

$$A := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B := \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

```
> Commutator(A,B);
```

$$\begin{bmatrix} -4 & -12 \\ 12 & 4 \end{bmatrix}$$

```
> IsCommuting(A,B);
```

false

```
> MyPackage[`Commutator`] := eval(Commutator); # wichtig: eval!
                                                # (wegen last name
evaluation
```

für Prozeduren)

```
MyPackageCommutator := proc(A::Matrix, B::Matrix) return simplify(`.(A, B) - `.(B, A)) end proc
```

```
> MyPackage[`IsCommuting`] := eval(IsCommuting);
```

```
MyPackageIsCommuting := proc(A::Matrix, B::Matrix)
```

```
  local n;
```

```
  n := LinearAlgebra:-RowDimension(A);
```

```
  return LinearAlgebra:-Equal(Commutator(A, B), LinearAlgebra:-ZeroMatrix(n))
```

```
end proc
```

```
> print(MyPackage);
```

```
table([Commutator = proc(A::Matrix, B::Matrix) return simplify(`.(A, B) - `.(B, A)) end proc,
```

```
IsCommuting = proc(A::Matrix, B::Matrix)
```

```
  local n;
```

```
  n := LinearAlgebra:-RowDimension(A);
```

```
  return LinearAlgebra:-Equal(Commutator(A, B), LinearAlgebra:-ZeroMatrix(n))
```

```
end proc])
```

```
> MyPackage[IsCommuting](A,A);
```

true

-- Ein **repository** ist eine Sammlung von Maple-Objekten (packages, Prozeduren etc.),
Verwaltung am besten mittels des packages **LibraryTools** (in MIPG nicht erwähnt)
(oder mittels **march**)

```
> with(LibraryTools);
```

```
[ActivationModule, AddFromDirectory, Author, Browse, BuildFromDirectory, ConvertVersion, Create,
Delete, FindLibrary, PrefixMatch, Priority, Save, ShowContents, Timestamp, UpdateFromDirectory,
WriteMode]
```

```
> currentdir();
```

"C:\"

Wir legen eine Library an (der Einfachheit in unserem Home-Verzeichnis):

```
> mylib := "MyNewLibrary.lib";
```

```
mylib := "MyNewLibrary.lib"
```

```
> Create(mylib); # erzeugt 2 Dateien (.lib und .ind)
```

```
Error, (in march) there is already an archive in "MyNewLibrary.lib"
```

```
> ssystem("dir"); # bzw. ssystem("ls"); # (Windows/Unix system call!  
)
```

```
[0, " Volume in Laufwerk C: hat keine Bezeichnung.
```

```
Volumeseriennummer: A8F9-012B
```

```
Verzeichnis von C:\
```

```
13.03.2009 10:50 <DIR>      83f57c701523fabcd23d2920  
04.06.2008 16:57          0 AUTOEXEC.BAT  
04.06.2008 16:57          0 CONFIG.SYS  
06.02.2009 18:43 <DIR>      cygwin  
06.02.2009 18:12 <DIR>      cygwin-temp  
09.06.2008 09:55 <DIR>      Dokumente und Einstellungen  
05.06.2008 17:33      7.118 HPDIU.log  
05.06.2008 10:43 <DIR>      Intel14.03.2009 14:27 <DIR>      MyMapleDirectory  
24.05.2009 12:46      40.960 MyNewLibrary.ind  
24.05.2009 12:46      1.768 MyNewLibrary.lib  
24.05.2009 12:51      32 my_file.txt  
10.07.2008 21:10 <DIR>      ozeanzeugs  
30.04.2009 09:39      30 pfile.m  
16.08.2008 15:56 <DIR>      Program Files  
08.04.2009 09:17 <DIR>      Programme  
05.06.2008 13:48 <DIR>      spoolerlogs  
02.05.2009 11:13 <DIR>      UDC Output Files  
06.08.2008 09:56 <DIR>      watcom-1.309.05.2009 10:14 <DIR>      WINDOWS  
7 Datei(en)      49.908 Bytes  
13 Verzeichnis(se), 182.222.663.680 Bytes frei"]
```

```
> print(MyPackage);
```

```
table([Commutator = proc(A::Matrix, B::Matrix) return simplify(`.(A, B) - `.(B, A)) end proc,
```

```
IsCommuting = proc(A::Matrix, B::Matrix)
```

```
local n;
```

```
n := LinearAlgebra:-RowDimension(A);
```

```
return LinearAlgebra:-Equal(Commutator(A, B), LinearAlgebra:-ZeroMatrix(n))
```

```
end proc])
```

```
> Save(MyPackage,mylib); # Abspeichern des Packages in mylib!
```

```
> ShowContents(mylib);
```

```
["MyPackage.m", [2009, 5, 24, 12, 52, 48], 1024, 248]]
```

```
> restart; # Neustart!
```

```
> with(LinearAlgebra): with(LibraryTools):
```

```
> currentdir();
```

```
"C:\"
```

```

> ssystem("dir My*");
      [0, " Volume in Laufwerk C: hat keine Bezeichnung.
      Volumeseriennummer: A8F9-012B

      Verzeichnis von C:\

      14.03.2009 14:27 <DIR>      MyMapleDirectory
      24.05.2009 12:52          40.960 MyNewLibrary.ind
      24.05.2009 12:52          1.272 MyNewLibrary.lib
      24.05.2009 12:51           32 my_file.txt
      3 Datei(en)          42.264 Bytes
      1 Verzeichnis(se), 182.222.675.968 Bytes frei"]

> mylib:="MyLibrary.lib";
      mylib := "MyLibrary.lib"

> ShowContents(mylib);
Error, (in march) there is no existing archive in "MyLibrary.lib"

> libname; # vordefinierter Pfad
march('open',currentdir()): # erweitert library-Pfad um mein
Verzeichnis
      "C:\Programme\Maple 12/lib"
      libname := "C:\Programme\Maple 12/lib", "C:\

> with(MyPackage); # wurde gefunden!
      [Commutator, IsCommuting]

> packages();
      [LinearAlgebra, LibraryTools, MyPackage]

> print(IsCommuting);
proc(A::Matrix, B::Matrix)
  local n;
  n := LinearAlgebra:-RowDimension(A);
  return LinearAlgebra:-Equal(Commutator(A, B), LinearAlgebra:-ZeroMatrix(n))
end proc

> Commutator(IdentityMatrix(3), ZeroMatrix(3));
      
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$


> IsCommuting(IdentityMatrix(3), ZeroMatrix(3));
      true

```

Das war nur ein einfaches Beispiel; so etwas reicht für 'private' Zwecke (Verwaltung eigener Packages).

Professionelles Verwalten von Bibliotheken geht über den Rahmen dieser Einführung hinaus und ist eher nur für den Entwickler echter Anwendungssoftware in Maple relevant

Als Mathematiker kann man aber durchaus in die Lage kommen, nichttriviale Packages zu erstellen und zu verwalten.

4. Modules

Datenkapselung und **Objektorientierung** wird in den neueren Maple-Versionen mit Hilfe von **Modules** organisiert; dies ersetzt und erweitert auch die traditionelle Verwaltung von packages als Tabellen von Prozeduren.

Hier nur ein einfaches Beispiel:

```
> zähler:=module()  
  export init,maxi,nächste;  
  local inkrementiere,zahl; # innerhalb des Modules eigentlich  
global  
  zahl:=0;  
#  
  maxi:=infinity; # Default  
#  
  inkrementiere:=proc()  
    zahl:=zahl+1;  
  end proc;  
#  
  init:=proc()  
    zahl:=0;  
  end proc;  
#  
  nächste:=proc()  
    zahl:=min(inkrementiere(zahl),maxi);  
  end proc;  
#  
end module;  
zähler := module( ) local inkrementiere, zahl; export init, maxi, nächste; end module
```

-- Bedeutung und Funktion dieses Modules:

(Nur) die **'exportierten'** Objekte **nächste()** und **maxi** sind bei der Verwendung des Moduls nach außen sichtbar (maxi ist hier auch von außen veränderbar). Damit hat man eine Prozedur und eine globale Variable erzeugt, die durch Zugriff auf den Modul verwendet werden können. (Im Prinzip kann man beliebige Objekte in einen Module packen und nur das exportieren, was nach außen sichtbar sein soll.)

```
> zähler:-maxi:=5; # Achtung auf Syntax  
maxi := 5  
  
> for i from 1 to 6 do zähler:-nächste() end do;  
1  
2  
3  
4  
5  
5
```



```
> print(zähler:-init);
```

```
proc( ) zahl := 0 end proc
```

Oder auch so:

```
> use zähler in
```

```
    maxi:=3;
```

```
    init();
```

```
    nächste();
```

```
    nächste();
```

```
    nächste();
```

```
    nächste();
```

```
end use;
```

```
maxi := 3
```

```
0
```

```
1
```

```
2
```

```
3
```

```
3
```

Eine etwas sauberere Lösung im Sinne der objektorientierten Programmierung würde darin bestehen, gar keine Variablenwerte zu exportieren, sondern nur Funktionsaufrufe:

```
> zähler:=module()
```

```
    description "Module Zähler";
```

```
    export init,get_maxi,set_maxi,nächste;
```

```
    local inkrementiere,maxi,zahl;
```

```
    zahl:=0;
```

```
#
```

```
    inkrementiere:=proc()
```

```
        zahl:=zahl+1;
```

```
    end proc;
```

```
#
```

```
    init:=proc()
```

```
        zahl:=0;
```

```
        maxi:=infinity; # default
```

```
    end proc;
```

```
#
```

```
    nächste:=proc()
```

```
        zahl:=min(inkrementiere(zahl),maxi);
```

```
    end proc;
```

```
#
```

```
    set_maxi:=proc(n)
```

```
        maxi:=n;
```

```
    end proc;
```

```
#
```

```
    get_maxi:=proc()
```

```
        return maxi;
```

```
    end proc;
```

```
#
```

```
end module:
```

```
> Describe(zähler);
```

```
# Module Zähler
```

```
module zähler:
```

```
    init( )
```

```
    get_maxi( )
```

```
    set_maxi( n )
```

```
    nächste( )
```

```
> use zähler in
```

```
    init();
```

```
    set_maxi(2);
```

```
    get_maxi();
```

```
    nächste();
```

```
    nächste();
```

```
    nächste();
```

```
end use;
```

```
∞
```

```
2
```

```
2
```

```
1
```

```
2
```

```
2
```

▼ Anmerkung:

Über Maple gäbe es noch sehr viel zu sagen.. In dieser VO wurde versucht, einen Stoffumfang abzudecken, der etwa dem täglichen Umgang des Mathematikers mit Maple als Werkzeug und Experimentierlabor entspricht (dabei fließt die persönliche Erfahrung des Vortragenden natürlich mit ein).

Der Umgang mit einem Computeralgebra-System ist aber *kein Ersatz* für persönliches mathematisches Know-How. Je besser man versteht, welche Mathematik `dahinter' steckt, desto kompetenter kann man mit einem derartigen System umgehen.

Syntaxwissen und `Befehle auswendig wissen' ist zweitrangig; es ergibt und erweitert sich in der Praxis von selbst (unterstützt durch die Maple-Hilfe bzw. online-Recherche).

Professionelle Softwareentwicklung in Maple -- im Sinne einer echten *Erweiterung* des Funktionsumfangs -- geht über den Rahmen dieser Einführung hinaus und wäre ein Thema für eine vertiefende LVA. Siehe auch Spezialliteratur (diverse, veraltet leider relativ schnell) und die Originaldokumente bei Maplesoft, insbesondere den **Maple Advanced Programming Guide**'.

▼ 5. Maplets

Anmerkung: **Maplets** sind **GUIs** (graphical user interfaces) für Maple-Anwendungen, die individuell erstellt werden können (so etwas Ähnliches gibt es auch in MATLAB). Keine Details.