



Leseprobe

Ulrich Stein

Objektorientierte Programmierung mit MATLAB

Klassen, Vererbung, Polymorphie

ISBN (Buch): 978-3-446-44536-9

Weitere Informationen oder Bestellungen unter

<http://www.hanser-fachbuch.de/978-3-446-44536-9>

sowie im Buchhandel.

# Vorwort

Zugegeben, es hat eine Weile gedauert, bis ich die Möglichkeiten der Objektorientierung in MATLAB® schätzen lernte. Im Jahr 2007 schrieb ich ein Lehrbuch, das sich mit dem „Programmieren mit MATLAB“ beschäftigt. Damals standen wir am Fachbereich Maschinenbau und Produktion der HAW Hamburg vor einem Problem: Unsere Studenten waren nicht recht zu motivieren, eine Programmiersprache zu erlernen. Wir „zwangen“ sie, sich längere Zeit mit Compiler, Linker und unverständlichen Fehlermeldungen herumzuquälen, um am Schluss als Ergebnis die Zahl 42 auf dem Bildschirm zu sehen. Ingenieure kommen inzwischen immer weniger mit Programmiersprachen wie C++ in Kontakt, ob in der Industrie oder im weiteren Verlauf des Studiums. Anders liegt die Situation beim Programm MATLAB. Dessen Funktionalität wird bei uns intensiv genutzt, beispielsweise zum Lösen von Differentialgleichungen oder zum Ansteuern von Robotern. Und besonders die grafischen Möglichkeiten von MATLAB fördern das Verständnis für viele technische Anwendungen. Deshalb habe ich damals, zusammen mit Kollegen, unser Bachelor-Modul „Angewandte Informatik“ auf die in MATLAB integrierte Programmiersprache umgestellt. Aus dem Skript zu dieser Vorlesung entstand das oben erwähnte Lehrbuch.

An die objektorientierte Programmierung (OOP) dachte ich dabei nur am Rande. Zwar hatte MATLAB auch damals bereits eingeschränkte OOP-Funktionalität, die ich in meinem Buch beschrieb. Die Vorgehensweise war jedoch noch sehr umständlich. Der große Schritt kam im Jahr 2008 mit der Einführung der *classdef*-Datei, die ähnlich wie in anderen OOP-Sprachen die Struktur einer Klasse festlegt. Durch die Definition von Handle-Klassen erhielt MATLAB außerdem eine Art von Referenzen.

Trotzdem, eine Zeit lang blieb ich noch skeptisch, ob man mit MATLAB wirklich vernünftig objektorientiert programmieren kann. Im Vergleich zu anderen OOP-Sprachen fehlten doch ein paar Dinge, beispielsweise die Möglichkeit, mehrere Methoden mit demselben Namen, aber unterschiedlicher Signatur zu deklarieren. Dies ist in MATLAB nicht vorgesehen. Aber hierfür – und auch für andere fehlende Bereiche – kann man sich recht einfach einen vernünftigen Workaround bauen.

Je länger ich mich mit der OOP-Funktionalität in MATLAB beschäftigte, desto mehr hat sie mich überzeugt. Und ich hoffe, dass ich auch Sie in diesem Buch dafür begeistern kann.

Nun zu Ihnen: Was erwarte ich von meinem Leser?

Idealerweise sollten Sie bereits ein wenig programmiert haben, nicht notwendigerweise in MATLAB. Zwar stelle ich Ihnen in diesem Buch, in den Kapiteln 1 und 2, die Oberfläche und die wichtigsten Sprachelemente von MATLAB vor – soweit es zum Verständnis der folgenden Abschnitte notwendig ist. Das ist jedoch nur eine knappe Zusammenfassung, eher gedacht für Umsteiger von anderen Programmiersprachen wie C++. Einem absoluten

Anfänger würde ich als Einstieg eher mein Buch „Programmieren mit MATLAB“ empfehlen – ein Lehrbuch, in dem Sie auch viele Beispiele und Übungen finden.

Doch zurück zum jetzigen Buch: Kapitel 3 beschreibt den objektorientierten Ansatz von MATLAB. Hier werden die zentralen Begriffe eingeführt, wie Klassen, Eigenschaften, Methoden, Datenkapselung, Vererbung, Polymorphie, Handle-Klassen etc.

In Kapitel 4 finden Sie längere Anwendungen aus der Physik und dem Maschinenbau. Dies soll den Umgang mit dem Erlernten vertiefen und weitere Tipps für ein strukturiertes Vorgehen geben.

Zum Abschluss liefert Kapitel 5 eine Befehlsreferenz und den Vergleich mit anderen OOP-Sprachen.

Dieses Buch ist aber kein Referenz-Handbuch für MATLAB. Die MATLAB-Funktionen werden oft nur so weit vorgestellt, wie es für die aktuelle Aufgabenstellung nötig ist. Für eine vollständige Definition der Funktionen sei auf die MATLAB-Hilfe verwiesen.

Die Idee, ein zweites Buch zu MATLAB zu schreiben, entstand während einer Unterredung mit Frau Franziska Jacob, M. A., meiner Ansprechpartnerin beim Fachbuchverlag Leipzig im Carl Hanser Verlag. Vielen Dank für ihr Engagement, mit dem sie das Projekt im Verlag durchsetzte. Dank auch an Frau Dipl.-Ing. (FH) Franziska Kaufmann, die mir beim Layout zur Seite stand. Dank an alle Kollegen, die mich zu diesem Projekt ermutigten und mir hilfreiche Tipps gaben, speziell Prof. Dr. rer. nat. Ivo Nowak, Prof. Dr. rer. nat. Thorsten Struckmann und Prof. Dr.-Ing. Jürgen Dankert. Und einen besonderen Dank an Elfriede Neubauer, die mir bei der stilistischen Überarbeitung eine große Hilfe war.

Die im Buch beschriebenen und abgebildeten Abläufe beziehen sich auf die Bedienoberfläche der Version MATLAB 2015a. Andere MATLAB-Versionen präsentieren sich dem Anwender zum Teil mit einer leicht abgewandelten Oberfläche. Lassen Sie sich deshalb nicht verwirren. Die vorgestellten Programme wurden mit verschiedenen Versionen getestet. Erweiterungen und die Lösungen der Aufgaben finden Sie auf meiner Homepage

[www.Stein-Ulrich.de/Matlab/](http://www.Stein-Ulrich.de/Matlab/)

Ich wünsche den Lesern, dass Ihnen das Programmieren auch Spaß macht und dass Ihnen möglichst viel vom hier präsentierten Stoff bei Problemlösungen nützt. Und nicht verdrängen oder vergessen: Informatik kann auch Schaden anrichten. Deshalb sollte jeder, der programmiert, sich überlegen, ob er sein Tun verantworten kann und will.

Hamburg, im August 2015

Ulrich Stein

# Inhalt

<b>1</b>	<b>Einführung</b> .....	<b>9</b>
1.1	Warum objektorientiert? .....	9
1.2	Erstes Objekt: Auto .....	11
1.3	MATLAB .....	15
1.4	Aufbau des Buches .....	18
<b>2</b>	<b>Programmieren mit MATLAB</b> .....	<b>20</b>
2.1	Variablen, Daten, Typen .....	20
2.2	Funktionen .....	26
2.3	Input/Output .....	29
2.4	Kontrollstrukturen .....	33
2.5	Grafik .....	40
2.6	Handles .....	42
2.7	Fragen .....	45
2.8	Aufgaben .....	46
<b>3</b>	<b>Objektorientierung</b> .....	<b>49</b>
3.1	Objekte und Klassen .....	49
3.2	Datenkapselung .....	52
3.3	Methoden .....	56
3.4	Vererbung .....	62
3.5	Polymorphie, abstrakte Klassen .....	69
3.6	Überladung von Operatoren .....	73
3.7	Handle-Klassen .....	76
3.8	Ereignisse .....	81
3.9	Destruktor .....	83
3.10	Attribute: <i>Constant, Static</i> .....	<b>86</b>
3.11	Aufzählungen ( <i>enumeration</i> ) .....	88
3.12	Pakete, Verzeichnisse, Namensbereiche .....	89
3.13	Fehlerbehandlung (Exceptions) .....	92
3.14	Fragen .....	99
3.15	Aufgaben .....	100

---

<b>4</b>	<b>Anwendungen</b>	<b>101</b>
4.1	Datenanalyse	101
4.1.1	<i>varargs</i> -Mechanismus	101
4.1.2	Datenübergabe und Datenausgabe	106
4.1.3	Methoden <i>mean</i> und <i>std</i>	109
4.1.4	Integration, Gauß-Glocke	111
4.1.5	Excel-Dateien lesen	113
4.1.6	Fragen	118
4.1.7	Aufgaben	119
4.2	Verkettete Listen	119
4.2.1	Listen-Knoten	120
4.2.2	Knoten-Destruktor	123
4.2.3	Listen aufbauen	126
4.2.4	Knoten löschen	131
4.2.5	Listen durchsuchen	133
4.2.6	Fragen	134
4.2.7	Aufgaben	134
4.3	Grafik-Liste	135
4.3.1	Grafik-Klasse <i>Shape</i>	136
4.3.2	Grafik-Text	137
4.3.3	Grafik-Linienelemente	139
4.3.4	Kopierkonstruktor	143
4.3.5	Grafik-Knoten	148
4.3.6	Grafik-Liste	150
4.3.7	Fragen	153
4.3.8	Aufgaben	153
4.4	Arduino-Board	154
4.4.1	Arduino und MATLAB	154
4.4.2	Serielle Schnittstelle (COM)	156
4.4.3	Klasse <i>MyArduino</i>	159
4.4.4	Fragen	163
4.4.5	Aufgaben	164
<b>5</b>	<b>Schlussbemerkungen</b>	<b>165</b>
5.1	Vergleich mit anderen Sprachen	165
5.2	OOP in MATLAB	167
	<b>Literatur</b>	<b>173</b>
	<b>Index</b>	<b>175</b>

# 2

## Programmieren mit MATLAB

In Kapitel 1 diente uns MATLAB hauptsächlich als Taschenrechner, der einzelne Befehle verarbeitet. Jetzt, in Kapitel 2, wollen wir zeigen, wie man mit MATLAB programmiert – jedoch ohne die erweiterten Möglichkeiten der OOP. Zu Klassen und Objekten kommen wir erst in Kapitel 3.

Denn vorher müssen wir noch ein paar Dinge klären, zum Beispiel:

- Was sind Variablen und Daten?
- Wie schreibt man in MATLAB Funktionen?
- Was sind Kontrollstrukturen?
- Wie kommuniziert ein Programm mit der Außenwelt?
- Welche Grafik-Möglichkeiten bietet MATLAB?
- Was sind Handles?

### ■ 2.1 Variablen, Daten, Typen

Variablen hatten wir bereits in Kapitel 1 kennengelernt, beispielsweise beim Aufruf:

```
>> x = int32( 5 );
```

Testen wir mit dem Befehl *whos*, welches Objekt dadurch erzeugt wurde:

```
>> whos x
  Name      Size      Bytes      Class
  x         1x1         4          int32 array
```

Es wurde eine Variable mit dem Namen *x* angelegt, vom Datentyp *int32* (ganze Zahl mit 32 Bits = 4 Bytes). Als Überschrift für den Datentyp verwendet MATLAB die Bezeichnung *Class*, was auf die Ähnlichkeit von MATLAB-Typen und benutzerdefinierten Klassen hinweist.

int32: (Typ)

x:

5
---

(Name) (Inhalt)

**Bild 2.1** Variable *x*

**Variablen** sind Objekte, denen man einen Namen gegeben hat. Der Name einer Variablen beginnt in MATLAB immer mit einem Buchstaben. Dann können Ziffern und das „\_“-Zeichen folgen. Zwischen Groß- und Kleinschreibung wird unterschieden. Bis zur Länge von 31 Zeichen sind die Namen eindeutig. Die Namen dürfen aber auch länger sein.

Dieser Name erlaubt den Zugriff auf die Objekte, um ihren Wert zu ändern oder um den Wert auszulesen – beispielsweise um ihn mit der Funktion *disp* auf dem Bildschirm auszugeben:

```
>> disp( x );
      5
```

Durch den Aufruf „`x = int32( 5 );`“ wurde die Variable *x* auch bereits mit Daten belegt. Als **Initialisierung** bekam *x* gleich zu Beginn den Wert 5 zugewiesen. Dafür benötigen die Objekte im Rechner einen Speicherplatz, je nach Datentyp unterschiedlich groß. Bei *int32* sind dies 4 Bytes = 32 Bits. 1 Byte ist eine Gruppierung von 8 Bits. Wie die Bits, also die abgelegten Daten im Speicher, vom Programm interpretiert werden, hängt vom Typ des Objekts ab.

0000 0000	0000 0000	0000 0000	0000 0101
-----------	-----------	-----------	-----------

**Bild 2.2** Bit-Folge (0 oder 1), 4 Bytes = 32 Bits

Die Zuordnung eines Namens und eines Datentyps zu einer Variablen bezeichnet man als **Deklaration** – als eine Vereinbarung, wie man den Inhalt der Variablen zu interpretieren hat. Wird bei einer Anweisung zusätzlich noch Speicherplatz belegt, nennt man dies **Definition**. Unsere Variable *x* wurde als Objekt vom Typ *int32* deklariert und durch die Zuweisung der Daten, der Zahl 5, auch gleichzeitig definiert.

In MATLAB verwendet man eine reine Deklaration für Variablen eher selten. Deklarationen sind normalerweise mit einer Definition verbunden. Eine Ausnahme bildet die *global*-Spezifikation, die darauf hinweist, dass diese Variable nicht eine lokale Variable der Funktion ist, sondern außerhalb definiert wurde und von dort beeinflusst werden kann. In der OOP wird die reine Deklaration bei den abstrakten Methoden eine wichtige Rolle spielen.

**Objekte** bezeichnet man auch als Exemplare oder **Instanzen** eines Typs (wobei der deutsche Begriff Instanz im täglichen Leben etwas anderes meint als das englische Wort *instance*).

In Abschnitt 1.2 hatten wir ein Objekt vom Typ *Auto* angelegt:

```
>> HH_BT_21 = Auto( 'VW_Golf', 2010 );
>> whos HH_BT_21
  Name      Size      Bytes  Class
HH_BT_21   1x1         102   Auto
```

Hierbei bekam die Variable den Namen *HH\_BT\_21*. Bei der Initialisierung wurden dem Auto-Objekt nicht nur ein, sondern sogar zwei Werte zugewiesen. Während ein Ganzzahl-

Objekt vom Typ *int32* nur für eine einzige Zahl gedacht ist, kann ein beliebiges Objekt, je nach seinem internen Aufbau, auch eine größere Zahl von Daten enthalten. Diese Daten werden als seine Eigenschaften (*properties*) in den Objektvariablen abgespeichert.

Doch zu selbst definierten Klassen und deren Objekten kommen wir erst in Kapitel 3. In diesem Kapitel beschränken wir uns auf die Typen, die standardmäßig in MATLAB eingebaut sind, wie *double*, *int32*, *char* etc.

Sie müssen in MATLAB den Datentyp bei einer Variablendefinition nicht explizit angeben. Wenn Sie den Typ nicht spezifizieren, werden Zahlen automatisch als *double* behandelt:

```
>> y = 5;
>> whos y
  Name      Size      Bytes  Class
  y         1x1         8    double array
```

Texte erscheinen automatisch als *char*, genauer: als ein Feld von Zeichen, als ein Charakter-Array mit einer Zeile und mehreren Spalten, hier 1x5:

```
>> t = 'Willy';
>> whos t
  Name      Size      Bytes  Class
  t         1x5         10    char array
```

Dieser Typ-Automatismus beschränkt sich in MATLAB nicht nur auf die Definition von Variablen. Sie bekommen auch keinerlei Warnung, wenn Sie beispielsweise Text einer Variablen zuweisen, die eigentlich für Zahlen gedacht ist. MATLAB ist nicht **typsicher**. Typverletzungen werden nicht registriert, obwohl explizite Datentypen und deren Überprüfung die Programme sicherer machen. Ein Teil der Fehler würde dadurch automatisch erkannt werden.

Manchmal ist es notwendig, dass man Datenwerte in ein anderes Format umwandelt. Beispielsweise kann man zu einem Zeichen (vom Typ *char*) die ganze Zahl (vom Typ *int32*) bestimmen, mit der das Zeichen intern dargestellt wird, seinen sogenannten ASCII-Wert:

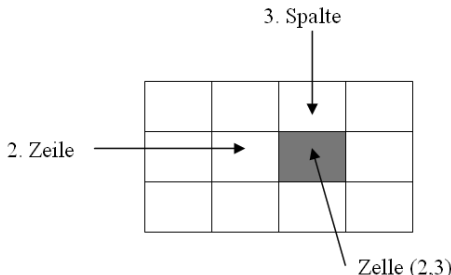
```
>> z = 'a';
>> whos z
  Name      Size      Bytes  Class
  z         1x1         2    char array
>> n = int32( z )
  n = 97
>> whos n
  Name      Size      Bytes  Class
  n         1x1         4    int32 array
```

Die *int32*-Anweisung erzeugt aus *z* die neue, ganzzahlige Variable *n*. Das Zeichen 'a', das in der Variablen *z* gespeichert ist, hat demnach den ASCII-Wert 97.

Typ-Umwandlungen bezeichnet man als Cast. MATLAB erzeugt einen Cast, wenn man den neuen Typ vor die Variable schreibt, beispielsweise „*int32(z)*“, analog der Definition von Variablen.



Wie bereits im ersten Kapitel erwähnt, sind die Standardtypen von MATLAB nicht einzelne Zahlen, sondern Matrizen, also zweidimensionale **Felder** (Arrays) mit einer gewissen Anzahl von Zeilen und Spalten. Eine Zelle des Arrays ist eindeutig durch die Angabe des Tupels „(Zeile,Spalte)“ definiert, im Beispiel die Zelle (2,3) in der 2. Zeile und 3. Spalte:



**Bild 2.3** Zelle (2,3) einer 3x4-Matrix

Arrays kann man entweder durch eine Wertliste in eckigen Klammern erzeugen – beispielsweise legt die folgende Anweisung das Feld *c* an, mit zwei Zeilen und zwei Spalten:

```
>> c = [ 1 2; 3 4 ]
c = 1 2
    3 4
```

Oder man definiert explizit die Zellen des Arrays durch ihren Inhalt, beispielsweise:

```
>> a(1,1) = 0;
>> a(1,2) = 3;
>> a(1,3) = -1;
>> a
a = 0 3 -1
```

Auf dieselbe Art kann man auf die Elemente eines Arrays zugreifen:

```
>> b = a(1,2)
b = 3
```

Um regelmäßig aufgebaute Arrays zu erzeugen, bietet MATLAB verschiedene Funktionen, etwa den „:“-Operator, mit dem man eindimensionale Felder definieren kann, beginnend mit einem *Startwert* im ersten Element und dann mit festem Abstand *Inkrement* bis zu einer Zahl, die kleiner oder gleich dem *Endwert* ist:

```
x = Startwert : Inkrement : Endwert;
```

Zum Beispiel für Zahlen zwischen 10 und 30 im Abstand von 5:

```
>> x = 10 : 5 : 30
x = 10 15 20 25 30
```

Eine ähnliche Funktionalität bietet die Funktion *linspace*, die ein Intervall in  $n$  Teile aufteilt, hier das Intervall von 5 bis 10 in  $n = 6$  Teile:

```
>> y = linspace( 5, 10, 6 )
y = 5 6 7 8 9 10
```

Zur Analyse der Eigenschaften von Feldern stellt MATLAB eine Reihe von Funktionen zur Verfügung. Die Funktion *size* informiert über die Zahl der Zeilen und Spalten eines Arrays. Der Rückgabewert von *size* ist ein Feld mit der Zahl der Zeilen im ersten Element und der Zahl der Spalten im zweiten:

```
>> a = [1 0; 2 1; 3 4];
>> sz = size( a )
sz = 3 2
>> zeilen = sz(1)
zeilen = 3
>> spalten = sz(2)
spalten = 2
```

Bei eindimensionalen Feldern, also Zeilen- oder Spaltenvektoren, kann man die Länge auch über die Funktion *length* abfragen:

```
>> b = [1 2 3];
>> len = length( b )
len = 3
>> sz = size( b )
sz = 1 3
```

Es gibt noch weitere Funktionen zur Analyse von Feldern, zum Beispiel die Funktion *numel*, die auch für mehrdimensionale Felder die Gesamtzahl der Elemente liefert, oder die Funktion *nnz*, die für ein Feld die Zahl der Elemente bestimmt, die ungleich null sind.

**Texte** (Strings) werden von MATLAB als Character-Arrays mit einer Zeile und  $n$  Spalten behandelt. Für Strings gibt es spezielle Funktionen, wie *strcmp* zum Vergleich zweier Texte oder die Funktion *upper* zum Wandeln von Klein- in Großbuchstaben.

Sie können auch Arrays von Strings anlegen. Nur müssen Sie hierbei darauf achten, dass alle Texte in so einem Array exakt dieselbe Länge haben, was für die Praxis wenig brauchbar ist.

Um Felder zu erzeugen, die Elemente mit unterschiedlichen Typen aufnehmen können, zum Beispiel unterschiedlich lange Texte, gibt es den Datentyp **Cell-Array**. Cell-Arrays können Sie wie normale Arrays mittels einer Wertliste erzeugen. Nur müssen Sie dazu geschweifte Klammern verwenden, beispielweise  $c = \{ 1, 'Willy', 2 \}$ . Auch zur Abfrage des Inhaltes einer Zelle brauchen Sie die geschweiften Klammern, zum Beispiel für das dritte Element  $c\{3\}$ . Bei der Verwendung von runden Klammern, wie  $c(3)$ , erhalten Sie den Wert als Inhalt einer Zelle:

```
>> c = {1, 'Willy', 2 }
c = [1 'Willy' 2]
>> c3 = c(3)
```

```

c3 = [2]
>> whos c3
  Name   Size  Bytes  Class  Attributes
  c3     1x1    68    cell
>> v3 = c{3}
v3 = 2
>> whos v3
  Name   Size  Bytes  Class  Attributes
  v3     1x1     8    double

```

Des Weiteren bietet MATLAB den Datentyp *struct*, der einzelne Daten zu einer Struktur zusammenfasst, beispielsweise für ein Adressenverzeichnis mit den Daten der einzelnen Personen:

```

>> p = struct( 'Name', 'Willy', 'Adresse', 'Berliner Tor 21' )
p =
  Name: 'Willy'
  Adresse: 'Berliner Tor 21'

```

Bei der Definition eines Struct gibt man paarweise die Komponenten des Struct und die zugehörigen Werte an. In unserem Beispiel besitzt der Struct *p* die beiden Komponenten *Name* und *Adresse*, die mit den Daten 'Willy' bzw. 'Berliner Tor 21' belegt sind. Auf den Inhalt der Komponenten können Sie mit Hilfe des Punktoperators zugreifen, analog dem Lesen der Eigenschaften eines Objekts, wie in Abschnitt 1.2 gezeigt:

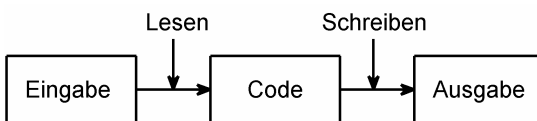
```

>> Adr = p.Adresse
Adr = Berliner Tor 21

```

Welche Operationen mit den Daten möglich sind, hängt vom Datentyp ab. Zahlen können Sie zum Beispiel addieren, Texte ausdrucken. Es macht aber wenig Sinn, zwei Textzeichen zu addieren. Was ergäbe wohl die Addition 'a' + 'b'?

Als **Datenverarbeitung** bezeichnet man jede Art von Operation mit Daten, also wie oben angeführt die Addition von Zahlen oder das Ausdrucken von Text. Datenverarbeitung erfolgt typischerweise in drei Schritten:



**Bild 2.4** Datenfluss

- Eingabe der Daten, zum Beispiel über die Tastatur,
- Verarbeitung der Daten, durch den Programm-Code von Funktionen,
- Ausgabe des Ergebnisses, zum Beispiel auf dem Bildschirm.

Für größere Aufgaben zerlegt man ein Programm in Teilprobleme, das heißt in einzelne Funktionen mit meist nicht mehr als einer Seite Programmcode. Größere Funktionen sind erfahrungsgemäß nicht zu überblicken und führen schnell zu Fehlern.

In den Funktionen werden die einzelnen Berechnungen durchgeführt. Hierzu dienen Ausdrücke und Zuweisungen, wie die Berechnung des Umfangs aus dem Radius durch die Multiplikation mit  $2\pi$ :

```
Umfang = 2 * pi * Radius;
```

Im Programm werden diese Anweisungen **sequentiell** ausgeführt, also zeilenweise nacheinander abgearbeitet. Innerhalb einer Zeile erfolgt die Auswertung zuerst von innen nach außen, wie Sie es bei Klammern in der Mathematik gewohnt sind, und danach in den Ausdrücken, wie beim Lesen, von links nach rechts. **Wertzuweisungen** erfolgen an Variablen, die links vor einem Gleichheitszeichen stehen.

Zum Beispiel im folgenden Aufruf die Addition des Wertes von  $r = 1.0$  mit der Zahl 3, dem Ziehen der Wurzel aus 4 ( $= 3 + 1$ ) mit der Funktion *sqrt* und der anschließenden Multiplikation mit 0.5, was für  $u$  letztendlich den Wert 1.0 ergibt:

```
r = 1.0;
u = 0.5 * sqrt( 3 + r );
```

Mit den Kontrollstrukturen, den Auswahlanweisungen und Schleifen, können Sie innerhalb einer Funktion den sequentiellen Ablauf verändern. Dies ist aber erst Thema in einem der späteren Abschnitte. Als Nächstes wollen wir uns anschauen, wie man in MATLAB Funktionen schreibt.

## ■ 2.2 Funktionen

Funktionen dienen dazu, Teilprobleme zu lösen. Dazu übergibt man ihnen eine Anzahl von Daten, aus denen die Funktion die nötigen Ergebnisse berechnet und diese dem Aufrufer zurückgibt.

Vom Blickpunkt des Aufrufers aus sind Funktionen eine Art Black Box, mit Eingangs- und Rückgabewerten – beispielsweise kann man die Zahl 9 als Eingabe an die Wurzel-Funktion *sqrt* geben, die dann als Rückgabe hoffentlich die Zahl 3 liefert.



**Bild 2.5** Black Box *sqrt*-Funktion

In MATLAB lautet dieser Aufruf folgendermaßen:

```
>> y = sqrt( 9 )
y = 3
```

Wie MATLAB oder eine andere Programmiersprache die Wurzel von 9 berechnet, das wird dem Anwender im Allgemeinen nicht mitgeteilt. Dieser verlässt sich darauf, dass das Ergebnis richtig ist. Es kann (bei etwas komplizierteren Funktionen) auch vorkommen, dass bei einem neuen Release von MATLAB der Berechnungsalgorithmus verändert wird, weil man zum Beispiel ein Verfahren entdeckt hat, das schneller ist.

# Index

## Symbole

\n 30  
:-Operator 23

## A

abgeleitete Klasse 62  
abgeleitetes Objekt 70  
Abstract 56, 72  
abstrakte Klassen 69, 72, 141  
abstrakte Methode 72  
Access 53, 86  
Access = { ?Class } 131, 166  
addlistener 82  
Aktualparameter 28  
annotation 154  
Anwendungen 101  
Arduino-Befehle 161  
Arduino-Board 154  
Array 16, 23, 119, 125, 133  
arrow 154  
assert 98  
Attribute 167  
Attribute: Constant, Static 86  
Aufbau des Buches 18  
Aufzählungen (enumeration) 88  
Ausgabefunktionen 30  
ausgelagerte Methoden 60  
Ausgleichsgerade 115  
Ausnahmen 93, 168

## B

back 126  
Balkendiagramm 109  
bar 109

base class 62  
Basisklasse 62  
Baudrate 157  
bedingte Auswahl 34  
beschleunigen 12, 53, 56, 79  
break 39  
BytesAvailable 158

## C

C++ 9  
Cast 22  
catch 95  
Cell-Array 24, 102  
char 16, 22  
Character-Array 17  
child class 62  
classdef 49, 167  
clc 18  
clear 75, 92, 138  
clear all 18, 54  
Color 138, 153  
color\_style\_marker 41  
COM 156  
Command Window 15  
Constant 86  
Copy Constructor 144  
createSerial 159

## D

data 120, 148  
Datenanalyse 101  
Datenkapselung 52  
Datentyp 16, 20, 59  
Datenübergabe und Datenausgabe 106

Datenverarbeitung 25  
Deep Copy 150, 166  
Definition 21  
Deklaration 21, 27, 72  
delete 75, 83, 123, 131, 148, 159  
derived class 62  
Destruktor 83, 123, 129, 143, 150  
digitalWrite 161  
disp 12, 30, 53, 54, 59, 62, 69, 107,  
166  
dispMsg 82  
dlnode 126  
doppelt verkettete Liste 120, 126, 135  
double 16, 22  
draw 65, 72, 138, 147, 148, 150  
Dynamic Properties 166

## E

Eigenschaften 11, 49  
einfache Alternative 35  
einfach verkettete Liste 120  
Eingabeaufforderung 16  
Eingabefunktion 31  
Eingangsparameter 27  
Ellipsen 148  
empty 93  
end 57  
enumeration 51, 88  
Ereignis 13, 81  
error 94, 102  
errordlg 103  
events 13, 51, 81  
Excel-Dateien lesen 113  
Exception 93

## F

fclose 159  
Fehlerbehandlung (Exceptions) 92  
Feld 16, 23, 119  
figure 43, 150  
FilledRect 68  
findobj 44, 133  
FontAngle 138  
FontSize 137

FontWeight 138  
fopen 32, 158  
Formalparameter 28  
Formatstring 30  
for-Schleife 36  
fplot 40  
fprintf 30, 33  
friend-Klassen 131, 166  
front 121, 126  
fscanf 162  
function 27, 57  
Function-Handle 42, 82  
Funktionen 26  
Funktions-Kopf 27  
Funktions-Rumpf 28  
fwrite 162

## G

Gauß-Glocke 111  
gca 44  
Generalisierung 62  
get 44, 166  
GetAccess 56  
getData 114  
getPre 132, 133  
gleichförmige Bewegung 115  
Grafik 40  
Grafik-Handles 44, 75  
Grafik-Klasse Shape 136  
Grafik-Knoten 148  
Grafik-Linienelemente 139  
Grafik-Liste 135, 150  
Grafik-Text 137  
grid 40

## H

Handle-Klassen 45, 57, 76, 136, 144, 165,  
168  
Handles 42, 76  
helper function 88  
Hierarchie 68  
Hilfsfunktion 88  
hold on 41

**I**

IDE 155  
if-Abfrage 34  
if-else-Abfrage 35  
Implementierung 52  
import 91  
Initialisierung 13, 21, 50  
input 31  
Input/Output 29  
insert 129  
instance 14, 21  
Instanz 21  
instrfind 157, 160  
int32 16  
integral 113  
Integration 111  
I/O 29  
isa 70, 93  
ischar 32, 104  
isempty 93, 104, 136  
ishandle 75, 136  
isMatlabHandle 136, 138  
isnan 136  
isnumeric 32  
isShapeObj 148  
isShLnObj 140  
isShTxtObj 138  
isvalid 158

**K**

Kapselung 54  
Kinderklasse 62  
Klasse 14, 49  
Klasse arduino 159  
Klasse Auto 11, 21, 49, 53, 83  
Klasse AutoData 77  
Klasse Data 101  
Klasse Fahrer 82, 84, 92  
Klasse GrList 150  
Klasse GrNode 148  
Klasse handle 77  
Klasse List 126, 134, 150  
Klasse LnStyle 140  
Klasse MException 95

Klasse MyArduino 159  
Klasse Node 120, 148  
Klasse PersData 77  
Klasse Shape 62, 136, 137, 144  
Klasse ShLine 141, 145  
Klasse ShLines 141  
Klasse ShRect 63, 147  
Klasse ShText 137, 152  
Klasse ShTriang 67  
Knoten 120  
Knoten-Destruktor 123  
Knoten einfügen 125  
Knoten löschen 123, 131  
Kommentar 27  
konstante Eigenschaften 86  
Konstruktor 13, 51, 53, 58, 101  
Konstruktor der Basisklasse 64  
Konstruktor, mehrere Basisklassen 69  
Kontrollstrukturen 33  
Kopien 77  
Kopierkonstruktor 143, 152, 166  
Kreise 148

**L**

leading parameter 71  
leeres Objekt 93  
length 24  
linspace 24  
Listen 120, 125  
Listen aufbauen 126  
Listen durchsuchen 133  
Listener 82  
Listen-Knoten 120  
Listenkopf 121

**M**

MAT-Files 33  
MathWorks 10  
MATLAB 15  
MATLAB-Handle 138  
mean 110  
mehrere Basisklassen 69  
Message-Identifizier 94  
messages 56

method dispatching 71, 165  
Methode der Basisklasse 70  
Methoden 12, 56, 167  
Methoden mean und std 109  
methods 12, 51, 56  
M-File 27  
Mikrocontroller-Board 154  
minus 73  
Mittelwert 110  
mixin 166  
move 62  
mtimes 73

## N

Nachrichten 56  
Name 21  
Namensbereich 91  
namespace 91  
nargin 102  
next 120  
noTicks 150  
notify 83  
num2str 32

## O

Objekt 11, 21, 49  
Objekte und Klassen 49  
Objektorientierung 49  
Objekt-Variable 50  
OOP 9  
openPort 159  
Operator \* 73  
Operator + 106  
OPO-Laser 116

## P

packages 89  
Paket 89  
Pakete, Verzeichnisse, Namensbereiche 89  
parent class 62  
Pass-by-Value 28, 165  
Pfad 105

Pfeile 154  
plot 40, 65, 75, 108  
plus 73, 107  
polyfit 115  
Polymorphie 60, 69, 70  
polyval 115  
popBack 132  
popFront 131  
private 52, 66, 86  
Programmieren mit MATLAB 20  
Prompt 15  
properties 11, 49, 51, 55  
protected 52, 66  
public 52, 86  
Punktoperator 12, 51, 57  
pushBack 127  
pushFront 129

## R

rectangle 148, 153  
Referenz 42, 57, 76, 121, 144, 165  
rethrow 97  
return 40  
Ringbuffer 121  
Rückgabeparameter 27, 57

## S

Schleifen 36  
Schnittstelle 52, 58, 72  
Semikolon 27, 63, 166  
separates Verzeichnis 60, 89  
sequentiell 26  
Sequenz 33  
serial 157  
serielle Schnittstelle 156  
set 45, 137, 140, 166  
SetAccess 56  
setData 145  
setFontSize 138  
setLnData 146  
setString 138  
Signatur 27, 69  
size 24  
Sketch 155



späte Bindung 59  
Speicherplatz 21  
Spezialisierung 62  
Standardabweichung 110  
Static 87  
statische Methoden 87, 140, 160  
std 110  
str2num 32  
String 137  
Strings 18, 24  
Stroustrup, Bjarne 9, 50, 62, 121  
struct 25  
subclass 62  
Suche 133  
superclass 62  
switch-Anweisung 36

## T

Templates 150, 166  
TestCase 166  
testGrList 152  
text 137  
Text 17, 24  
throw 95  
trapez 112  
Trapezregel 112  
try-catch-Block 95  
typsicher 22, 165

## U

Überladen 60, 71  
Überladung von Operatoren 73  
Überschreiben 60, 69, 70  
uchar 162  
uigetfile 104  
Unterfunktion 33  
USB-Schnittstelle 156

## V

Value-Klassen 76  
ValuesSent 162  
varargin 102  
varargs-Mechanismus 28, 71, 101,  
122, 141, 144, 166  
Variablen 21, 50  
Variablen, Daten, Typen 20  
Vaterklasse 62  
verbrauchen 79, 83  
Vererbung 62, 68, 168  
Vergleichsoperatoren 33  
verkettete Listen 119  
Verzweigungspunkt 33  
Vorgängersuche 132

## W

Wertzuweisung 26  
what 91  
which 92  
while-Schleife 39  
whos 16  
Wiederholschleife 39

## X

xlsread 33, 114  
xlswrite 33  
XTick 45

## Z

Zahlenformate 17  
Zählschleife 36  
Zeiger 76, 121, 165  
Zustand 11, 51