
Oldenbourg

Perl

Anwendungen und fortgeschrittene Techniken

von
Helmut Seidel
unter Mitwirkung von
Prof. Dr. Jürgen Schröter

Fachhochschule Landshut

Oldenbourg Verlag München Wien

Titelabbildung: Peter Kornherr

Ebenfalls bei Oldenbourg erschien von den Autoren Schröter und Seidel der einführende Band „Perl – Grundlagen und effektive Strategien", ISBN 3-486-25889-3, der die wichtigsten Sprachelemente von Perl (Skalare, Arrays, Hashes, Referenzen, Formate, Reguläre Ausdrücke) erläutert und sehr intensiv auf die Objektorientierung und komplexe Datenstrukturen in Perl eingeht.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-verlag.de

Lektorat: Christian Kornherr
Herstellung: Rainer Hartl
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: R. Oldenbourg Graphische Betriebe Druckerei GmbH

ISBN 3-486-25902-4

Inhaltsverzeichnis

1 Vorwort

Als Professor Schröter die Idee zu diesem Buch-Projekt hatte, war ich sofort Feuer und Flamme. Viele Gedanken über das, was ich schreiben würde, gingen mir durch den Kopf. Langsam machte sich Ernüchterung breit. Jedes der Themen, die in diesem Buch behandelt werden sollten, würde spielend ein eigenes Buch dieses Umfanges füllen. Also, wie den Stoff begrenzen? Wie ein Buch schreiben, das nicht schon andere Autoren vor mir geschrieben haben? Diese Fragen haben mich eine ganze Weile verfolgt. Verschiedene Ansätze führten in die Irre und wurden wieder verworfen.

Schließlich besann ich mich auf das, was ich bin: Ein Perl-Programmierer, der täglich viele Stunden damit verbringt, Anwendungen der verschiedensten Art in Perl zu schreiben. Warum sollte ich also nicht *mein* Buch schreiben? Ein Buch, in dem ich die Techniken und Vorgehensweisen zusammenfasse, die ich bei meiner Arbeit anwende. Sicherlich, dachte ich, kann man über jedes der hier angesprochenen Themen weit mehr sagen; aber, so stellte ich mir die Frage, was von all dem Stoff benötige *ich* für meine tägliche Arbeit tatsächlich?

Die Idee begann Form zu gewinnen. Ich wollte ein Buch schreiben, das ich mir neben meinen Computer legen und in dem ich die Punkte nachlesen kann, die ich immer wieder vergesse. Es sollte daher auch Themen einschließen, die nicht unmittelbar zu Perl gehören, ohne die es allerdings nicht möglich ist, umfangreichere Programme zu erstellen. Zu diesen Themen gehört beispielsweise eine Einführung in das Design von Datenbanken und in die Grundlagen der Abfragesprache SQL genauso wie eine kurze Einführung in die Seitenbeschreibungssprache HTML oder die Einrichtung des Apache Webservers.

Den breitesten Raum in diesem Buch nimmt natürlich die Programmiersprache Perl selbst ein. Wir zeigen Ihnen, wie Sie Ihre Anwendungen mit Packages modularisieren und mit Modulen eigene Funktionsbibliotheken anlegen können. Auch Techniken, von denen so mancher C-Programmierer nur träumen mag, wie das Ausführen von Code zur Kompilierzeit oder das Nachladen von Code zur Laufzeit, stellen wir Ihnen vor. Falls Sie in einem Projekt an die Grenzen der Möglichkeiten von Perl stoßen, was unserer Meinung nach fast gar nicht möglich ist, zeigen wir Ihnen, wie einfach es ist, Module in C zu erstellen und diese mit Perl-Code zu verbinden.

Der Begriff der *Objektorientierten Programmierung* ist in aller Munde; wir stellen ausführlich dar, was sich hinter diesem Begriff verbirgt, und wie Sie diese Techniken in Ihren Projekten einsetzen können.

Anwendungen, die nicht in der einen oder anderen Weise mit Datenbanken kommunizieren, sind heute fast nicht mehr denkbar. Daher verfügt auch Perl über eine hervorragende Datenbankschnittstelle, die wir Ihnen umfassend vorstellen.

Perl ist mit dem Internet groß geworden, daher können wir den Bereich der CGI-Programmierung nicht beiseite lassen. Wir erläutern, wie Sie Ihre Testumgebung einrichten, stellen die Grundlagen der Seitenbeschreibungssprache HTML dar und tauchen dann tief in die CGI-Programmierung ein. Wir erklären Ihnen, wie Sie Datenbanken aus Webseiten heraus ansprechen, wie Sie Ihren Code mit Hilfe von Templates modularisieren und vieles mehr. Vor allem möchten wir Ihnen aber auch zeigen, dass der Einsatz von Perl und CGI auch dann lohnt, wenn Sie keine Web-Anwendungen schreiben möchten.

Das Perl/Tk-Toolkit zum Erstellen von grafischen Benutzerschnittstellen ist eine der mächtigsten Erweiterungen zu Perl. In unserem Buch stellen wir Ihnen dieses Toolkit in einem ausführlichen Querschnitt vor und zeigen auch einige Steuerelemente in Ihrer Anwendung, die in der Literatur mitunter vergessen werden, die aber in komplexen Programmen von größtem Nutzen sind.

Mit den beiden Bänden dieses Buches erhalten Sie einen umfassenden Einblick in die Softwareentwicklung mit Perl, von den Grundlagen bis zur fertigen Anwendung.

Dabei setzen wir im vorliegenden zweiten Band voraus, dass Sie die Grundlagen der Perl-Programmierung beherrschen, und auf Ihrem Rechner eine Perl-Installation vorhanden ist.

Ich würde mir wünschen, dass der Funke der Begeisterung, den mir meine tägliche Arbeit mit Perl beschert, auch auf Sie überspringt und Sie feststellen, dass Perl tatsächlich eine der mächtigsten Programmiersprachen ist, die derzeit auf dem Markt zu finden sind und, dass sie längst dem Stadium einer Skriptsprache entwachsen ist. Perl kann sich heute durchaus mit System-Sprachen wie C, C++ und Java messen und lässt diese in einigen Bereichen recht "alt aussehen".

Viel Spaß bei der Lektüre dieses Buches!

1.1 Ein Wort zu freier Software

Als ich vor nunmehr sechs Jahren damit begann, mich mit dem damals noch recht jungen Betriebssystem LINUX zu beschäftigen, warfen mir die meisten Menschen vor, ich sei verrückt. Doch mich faszinierte die Idee der freien Software, deren Quellcodes nicht in Panzerschränken versteckt sind, sondern bei der jedermann zum Programm gleich die Quellcodes mitbekommt. Und das zumeist umsonst[1]! 1999 beschloss ich, die Hauptaktivität meiner Firma in Richtung LINUX zu verlegen. Bis heute habe ich diesen Entschluss nicht eine Sekunde bereut! Die Linuxer sind ein hilfreiches Völkchen, und im Laufe der Jahre bin ich mit vielen Menschen rund um den Globus in Kontakt gekommen, die mir (und ich hoffentlich auch Ihnen) bei so manchem Problem mit Tipps und teilweise seitenlangen "Howto"-Mails zur Seite standen.

Dieses Buch ist ausschließlich mit freier Software entstanden! Hier ein kurzer unvollständiger Überblick über die Software, die ich bei der Erstellung dieses Buches eingesetzt habe:

[1] Es ist ein weit verbreiteter Irrtum, dass freie Software immer kostenlos sei! Der Begriff *frei* bezieht sich allerdings auf die Tatsache, dass jeder das Recht hat, den Quellcode dieser Software einzusehen und nach Belieben zu verändern.

Betriebssysteme. Auf meinen Computern setze ich derzeit zwei verschiedene LINUX-Distributionen ein: Im Firmenbereich, wo es vor allem auf Sicherheit ankommt, sowie auf den Servern setze ich Debian-LINUX ein, das kostenlos unter der Web-Adresse: `http://www.debian.org` bezogen werden kann. Dort finden Sie auch Adressen von Firmen, die CD-Sätze dieser Distribution vertreiben. Debian ist keine Anfänger-Distributior und die Installation ist nicht ganz einfach.

Auf meinen Desktop-Systemen verwende ich derzeit RedHat-LINUX in der aktuellen Version 9.0. Auch diese Distribution kann über die Web-Seite des Herstellers kostenlos (in einer drei-CD-Version) heruntergeladen werden (`http://www.redhat.de`). Daneben ist RedHat-LINUX im Handel in verschiedenen Paketierungen zu erwerben.

Textverarbeitungsprogramme und Editoren. Der größte Teil dieses Buches wurde mit den Editoren (in der Reihenfolge der Beliebtheit) Nedit, Emacs21 und Vim erstellt. Alle Editoren verfügen über Syntaxhighlighting (also das farbige Hervorheben bestimmter Schlüsselworte) für Perl und LaTeX und sind Bestandteil der genannten LINUX-Distributionen.

Bei der Korrektur leistete das umfangreiche Office-Paket OpenOffice wertvolle Dienste. OpenOffice kann in der deutschen Version kostenlos unter der folgenden Internet-Adresse bezogen werden: `http://de.openoffice.org`.

Der Text wurde mit LaTeX gesetzt, einem leistungsfähigen, aber nicht ganz einfach zu bedienenden Textsatz-System. LaTeX ist Bestandteil aller LINUX-Distributionen.

Webserver und Webbrowser. Als Webserver verwende ich Apache, der weiter unten beschrieben wird. Als Browser Mozilla, der ehemalige Netscape-Browser. Auch diese beiden Pakete finden Sie auf allen gängigen LINUX-Distributionen.

Datenbank-Systeme. Als Datenbank-Systeme nutze ich PostgreSQL und MySQL, die ebenfalls in jeder LINUX-Distribution vorhanden sind.

Grafik. Alle Grafiken und Bildschirmhardcopies haben ich mit dem Grafikprogramm The Gimp und dem Zeichenprogramm Xfig erstellt. Raten Sie mal, wo Sie diese Anwendungen finden.

Wenn Sie nichts mit LINUX zu tun haben möchten, brauchen Sie nicht traurig zu sein! Fast alle genannten Anwendungen sind mittlerweile auch für Windows-Systeme erhältlich und auch dort kostenlos. Lediglich Xfig ist meines Wissens bisher nicht portiert worden. Die Installation von PostgreSQL unter Windows ist nicht ganz einfach, so dass Sie vielleicht besser auf MySQL ausweichen.

2 Dank

An einem solchen Buchprojekt haben viele Menschen mitgewirkt. Diese Mitwirkung kann man in zwei Kategorien einteilen: *freiwillig* und *unfreiwillig*. Zunächst möchte ich mich bei den freiwilligen Helfern bedanken:

An erster Stelle gilt mein Dank dem Mitautor und Initiator dieses Buches, Herrn Professor Dr. Jürgen Schröter, der mir nicht nur stets mit Rat und Hilfe zur Seite stand, sondern der mir auch die Möglichkeit gab, mein Wissen an der Fachhochschule Landshut an den Mann und die Frau zu bringen.

Ebenfalls in die Kategorie der freiwilligen Helfer gehören "unsere" Lektoren Frau Irmela Wedler und Herr Christian Kornherr vom Oldenbourg Verlag, die unser Projekt stets engagiert betreuten. Auch Ihnen gilt mein Dank.

Bedanken möchte ich mich auch bei Herrn Peter Kornherr für den gelungenen Umschlagentwurf und dafür, dass er unsere Hündin Lilly (die allerdings in die andere Kategorie gehört) auf den Umschlag dieses Buches gezaubert hat.

Folgende Personen haben Ihre Hilfe zu diesem Buch eher unfreiwillig beigesteuert:

Mein größter Dank gehört meiner Frau Susanne, die am Entstehungsprozess dieses Buches in vielfältiger Weise beteiligt war: Sie hat mich und meine Launen ertragen und hat das Buch Seite für Seite gelesen, korrigiert und so manchen komplizierten Schachtelsatz in verständliches Deutsch übersetzt. Und das, obwohl sie Computer eigentlich gar nicht mag! Danke!

Mein Dank gebührt auch meinem Partner und Mitgeschäftsführer der idomeo GmbH Herrn Franz Held, der einige wichtige Projekte zu Gunsten des Buches verschob.

Und schließlich möchte ich meiner Tochter Emily danken, die auf viel Zeit mit mir verzichten musste.

Landshut 2003

Helmut Seidel

3 Perl-Pakete

Jeder, der schon einmal ein Perl-Skript von mehr als 1000 Zeilen erstellt hat, kennt die Probleme, die damit verbunden sind, ein solches Programm zu pflegen. Schon nach kurzer Zeit ist das Rad Ihrer Wheel-Maus heiß gelaufen, und Ihnen stehen die Tränen in den Augen, vom Hin- und Herscrollen in Ihrer Datei.

Auch ein anderes Szenario kennt fast jeder Programmierer. Sie haben für eine bestimmte Aufgabenstellung einen komplizierten Algorithmus entwickelt und in Ihre Anwendung integriert. Nach geraumer Zeit erstellen Sie eine weitere Anwendung, die diesen Algorithmus erneut verwenden soll. Hierfür bietet sich mit Ihrem bisherigen Wissensstand folgende Lösung an: Zunächst suchen Sie die Anwendung, in der Sie den Algorithmus formuliert haben. Glücklich können sich diejenigen schätzen, die ein *Programmiertagebuch* führen, indem alle Tätigkeiten und Lösungen verzeichnet sind. Sie können nun die Stelle suchen, an der Ihr Algorithmus ausformuliert ist und ihn mit Hilfe der Copy & Paste Funktionalität Ihres Editors in Ihr aktuelles Projekt einfügen. Nun müssen Sie die Namen aller globalen Variablen des Code-Blockes an das aktuelle Projekt anpassen. Ich hoffe, die Vorstellung dieses Vorganges treibt nicht nur mir den Angstschweiß auf die Stirn. Es grenzt schon an ein Wunder, wenn der so entstandene Code ohne Probleme funktionierte. Aber stellen Sie sich mal vor, nach einer weiteren längeren Zeitspanne, sagen wir mal von einem Jahr, ändern sich die Anforderungen an Ihren Algorithmus, weil sich beispielsweise die gesetzlichen Vorschriften geändert haben (etwa in einem Warenwirtschaftssystem). Sie müssen also den Code beider Anwendungen entsprechend anpassen. Und wieder schaffen Sie sich eine unüberschaubare Anzahl von möglichen Fehlerquellen, wieder endloses Testen von zwei Anwendungen ... Und was, wenn der Algorithmus so allgemein ist, dass Sie ihn nicht nur in zwei, sondern vielleicht in fünf oder gar zehn Skripten untergebracht haben? Wir brechen dies Horror-Szenarium an dieser Stelle ab ...

Was, so mögen Sie sich vielleicht fragen, haben die beiden vorgestellten Szenarien gemeinsam? Die Antwort ist: in beiden Fällen können Sie sich das Leben erheblich vereinfachen, wenn Sie Ihre Anwendung *modularisieren.* Modularisieren bedeutet, dass eine Anwendung in überschaubare und inhaltlich zusammengehörige Einheiten zerlegt und in einzelnen Dateien hinterlegt wird. In den meisten Programmiersprachen werden diese Einheiten *Module* oder *Bibliotheken* genannt.

Das Modularisieren von Anwendungen bietet Ihnen zwei nicht zu unterschätzende Vorteile:

1. Da Ihre Anwendung nun aus kleinen überschaubaren Einheiten besteht, ist sie leicht und bequem zu warten. Natürlich sollten Sie darauf achten, keine "Super-Module" mit vielen tausend Code-Zeilen zu schreiben. Vielmehr sollten Ihre Mo-

dule eine überschaubare Größe besitzen. Als Faustregel verwendet der Autor eine Referenzgröße von ungefähr 100 Zeilen als Maximum für ein Modul[1].

2. Code, welchen Sie in Modulen abgelegt haben, können Sie auf besonders einfache Art und Weise wiederverwenden, wenn Sie einige grundlegende Regeln beachten. Wir werden in diesem und dem folgenden Kapitel genau darstellen, wie Sie Ihre eigenen Funktions-Bibliotheken anlegen, bzw. wie Sie die Bibliotheken (*Module*) anderer Entwickler in Ihre Anwendungen einbinden.

In diesem Kapitel werden wir uns mit den *Packages* in Perl beschäftigen. Das Verständnis dieses Themas ist grundlegend für die Perl-Konzepte der *Module* sowie der *Objektorientierten Programmierung*. Sie sollten daher das vorliegende Kapitel in jedem Fall durcharbeiten. Im Einzelnen werden wir folgende Themen behandeln:

- Zunächst möchten wir Ihnen die Frage beantworten, was Pakete sind und den wichtigen Begriff des *Namensraumes* einführen.
- Anschließend beantworten wir die Frage, wie Pakete deklariert werden und vor allem, was Sie dabei beachten sollten.
- Wie Sie auf Variablen, Subroutinen etc. aus anderen Paketen zugreifen, wird das Thema des folgenden Abschnitts sein.
- Wir werden dann erläutern, wie Sie Pakete in Ihre Anwendungen einbinden.
- Besonders bei größeren Projekten kann es notwendig sein, Pakete bzw. Ihre Anwendung zu initialisieren. Hierunter verstehen wir, Code auszuführen, wenn die Anwendung oder das Paket kompiliert wird. Ebenso kann es notwendig sein, Code auszuführen, nachdem die Arbeit mit der Anwendung abgeschlossen wurde. Beides wird von Perl in hervorragender Weise unterstützt. Wie dies geschieht, wird das Thema des anschließenden Absatzes sein.
- Für einen schnellen und bequemen Zugriff auf Variablen und Routinen eines anderen Paketes kann es nützlich sein, die Symbole des betreffenden Paketes in den aktuellen Namensraum zu importieren. Auch dieses möchten wir weiter unten erläutern.
- Den Abschluss dieses Kapitels bildet eine Erklärung des ausgesprochen mächtigen Mechanismus des Autoloadings, mit dem es beispielsweise möglich ist, Code zur Laufzeit der Anwendung nachzuladen.

[1] Wenn Sie in einem Modul eine grafische Benutzerschnittstelle implementieren, werden Sie den Wert von 100 Zeilen sicher um ein Vielfaches überschreiten. Sie sollten sich daher angewöhnen, den Code für solche Benutzerschnittstellen vom restlichen Programmcode zu trennen und stets in einem eigenen Modul zu definieren.

3.1 Was sind Perl-Pakete?

Bisher haben Sie gelernt, dass Variablen, die ohne die Modifier `my` oder `local` deklariert wurden, *globale* Variablen und damit in der gesamten Anwendung sichtbar sind. Genau genommen war das nur die halbe Wahrheit. Tatsächlich gibt es in Perl solche globalen Variablen überhaupt nicht. Vielmehr beschränkt sich in Perl die Sichtbarkeit einer Variablen auf das umschließende *Package*. Ein Package ist also nichts anderes als ein eigener Namensraum, der die Sichtbarkeit von Variablen, Arrays, Hashes, Subroutinen, Datei- und Verzeichnishandles[2] auf das umschließende Package einschränkt. Wird Programmcode nicht explizit einem Paket zugeordnet, übernimmt dies Perl für uns und ordnet den entsprechenden Code dem Package `main` zu. Folglich existiert in Perl-Anwendungen kein Code, der nicht einem Paket zugeordnet ist. Dies bedeutet, dass eigentlich auch keine wirklich globalen Variablen sondern lediglich *modulglobale* Variablen existieren. Alle Bezeichner eines Namensraumes werden in einer eigenen *Symboltabelle* abgelegt. Eine Symboltabelle wird dabei von einer Hash-Struktur gebildet, auf die, wie wir weiter unten sehen werden, mit den üblichen Hash-Methoden zugegriffen werden kann.

Der Gültigkeitsbereich eines Paketes erstreckt sich dabei stets bis zur nächsten Paket-Deklaration oder, falls keine folgt, auf den umschließenden Block, in dem das aktuelle Paket deklariert wurde. Falls Sie gar keine Pakete explizit deklariert haben oder die Paket-Deklaration unmittelbar an den Anfang einer Quellcode-Datei gestellt haben, ist dies die gesamte Datei.

Was ist aber an diesem Umstand so bemerkenswert? Die Antwort auf diese Frage ist eigentlich ganz einfach: Nur so ist es überhaupt möglich, Anwendungen zu modularisieren und Code zu Bibliotheken zusammenzufassen. Wäre es anders, müssten Sie bei jeder neuen Variablen, die Sie verwenden, prüfen, ob Sie nicht schon irgendwo in Ihrem Anwendungs-Dschungel eine Variable dieses Namens verwendet haben. Stellen Sie sich nur einmal vor, Sie haben eine Variable `$x` als Zählvariable einer `for`-Schleife verwendet, nun rufen Sie innerhalb des Schleifenkörpers eine Funktion aus einem Modul auf, die ebenfalls eine Variable dieses Namens verwendet und dieser einen festen Wert zuweist. Ihre `for`-Schleife würde mit Sicherheit nicht so reagieren, wie Sie es erwarteten. Doch zum Glück war dies nur der Ausschnitt aus einem Albtraum[3]. Da in Perl allerdings der Gültigkeitsbereich von Bezeichnern stets eingeschränkt ist, können Sie problemlos Module und Bibliotheken erstellen, ohne Angst haben zu müssen, mehrmals den gleichen Variablennamen verwendet zu haben. Erst durch dieses Konzept wird auch die Programmierung in Teams möglich.

[2] Wir werden im weiteren Verlauf zusammenfassend von *Bezeichnern* sprechen.

[3] In meinen Seminaren ist es übrigens eine beliebte Fehlerquelle, in einer Anwendung mehrmals dieselben Variablennamen (einmal im Hauptprogramm, einmal in einer Unterroutine etc.) zu verwenden. "Er nimmt mir da was nicht ..." ist in diesem Falle der Anfang einer gängigen Frage! Prinzipiell sollten Sie, um Ihrer Gesundheit willen, stets das Pragma `strict` verwenden und auf (modul-) globale Variablen ganz verzichten!

3.2 Deklaration von Paketen

Um ein Paket zu erstellen, verwenden Sie die **package**-Anweisung von Perl, welche die folgende allgemeine Syntax besitzt.

```
package Paketname;
```

Also beispielsweise:

```
package Kontoverwaltung;
```

In der Regel sollten Sie in einer Datei jeweils nur eine einzige **package**-Anweisung als erste Anweisung der Datei[4] aufnehmen. Damit weisen Sie den gesamten Code einer Datei diesem Paket zu. Eine solche Datei sollte zudem stets die Dateinamenserweiterung `.pm` besitzen. Die Endung `.pm` steht übrigens für **P**erl **M**odule. Nur Dateien, die diese Endung besitzen, können, wie wir im nächsten Abschnitt sehen werden, mit Hilfe der Anweisung `use` in eine Quellcode-Datei eingebunden werden.

Gemäß der Philosophie von Perl: "Der Programmierer wird schon wissen, was er tut" können innerhalb einer Datei auch mehrere Pakete deklariert werden. Zudem kann beliebig zwischen diesen Paketen gewechselt werden. Beachten Sie hierzu das folgende Beispiel.

```
package GiroKonto;                          # Package GiroKonto deklarieren
$inhaber = "Micky Maus";                    # Einige Package-Variablen anlegen
$kontostand = 5000;

package SparKonto;                          # Ein weiteres Package deklarieren
$inhaber = "Daisy Duck";                    # auch hier ein paar Variablen
$kontostand = 3500;

package GiroKonto;                          # Zurueck zum Girokonto
# Kontodaten GiroKonto ausgeben
print "$inhaber hat $kontostand Euro auf seinem Konto.\n";

package SparKonto;                          # Zurueck zum Sparkonto
\# Kontodaten SparKonto ausgeben
print "$inhaber hat $kontostand Euro auf seinem Konto.\n";
```

Es ist also tatsächlich möglich, in einer Quellcode-Datei mehrfach Variablen gleichen Namens zu deklarieren, sofern Sie in unterschiedlichen Packages angelegt wurden. Dies funktioniert, da die Variablen jeweils in der Symboltabelle des aktuellen Packages abgelegt wurden. Allerdings sei an dieser Stelle nochmals darauf hingewiesen, dass es sich bei dem oben dargestellten Beispiel um einen ausgesprochen **schlechten Stil** handelt.

[4] Sofern das von Ihnen angelegte Paket allerdings der Startpunkt einer größeren Anwendung ist, also das Paket `main` ersetzt, sollte die **package**-Anweisung erst nach der **SheBang**-Zeile stehen.

Genauso wie es möglich ist, mehrere Pakete in einer Datei zu deklarieren, ist es auch möglich, ein Paket über mehrere Dateien zu verteilen, indem mehrfach die gleiche package-Deklaration verwendet wird. Aber auch dies ist ein schlechter Stil und sollte in jedem Falle vermieden werden.

3.3 Einbinden von Paketen mit use und require

Nachdem Sie nun ein Paket mit wichtigen Routinen für Ihre Anwendung geschrieben haben, möchten Sie dieses natürlich auch nutzen. Um dies tun zu können, müssen Sie das betreffende Paket zunächst in Ihre Anwendung *einbinden*. *Einbinden* bedeutet, dass der Code eines Paketes[5] in die aktuelle Quellcode-Datei eingelesen wird[6]. Dieser Einlesevorgang kann entweder über die Anweisung use oder mit Hilfe der Anweisung require geschehen.

3.3.1 Die Anweisung use

Der sicherste Weg, ein Paket in Ihre Anwendung einzubinden, ist das Verwenden der Anweisung use. Die allgemeine Syntax der use-Anweisung ist

```
use Paketname;
```

Um das Paket Kontoverwaltung.pm in Ihre Anwendung einzubinden, verwenden Sie die Anweisung

```
use Kontoverwaltung;
```

Mit Hilfe von use können allerdings nur Pakete eingebunden werden, die in Dateien gespeichert sind, deren Dateinamenserweiterung .pm ist. Damit die Datei korrekt geladen werden kann, ist es zudem notwendig, dass beim Laden ein boolscher Erfolgswert zurückgegeben wird. Die letzte ausführbare Anweisung im globalen Gültigkeitsbereich des Paketes sollte daher wie folgt lauten.

```
return 1;
```

Weniger gut lesbar, aber auch möglich ist folgende Variante.

```
1;
```

Damit kann der allgemeine Aufbau einer Paket-Datei wie folgt dargestellt werden.

```
# Deklaration des Paketes
package Paketname;

# evtl. Einbinden von weiteren Modulen/Paketen
use Paketname;
```

[5] Wir sollten eigentlich von einem *Modul* sprechen. Da wir aber diesen Begriff erst im nächsten Kapitel erläutern, sprechen wir hier noch etwas unhandlich von einem *Paket*.

[6] C/C++-Programmierern wird dies bekannt vorkommen. Der Vorgang erinnert an das Einbinden von (header)-Dateien mit Hilfe der Präprozessor-Anweisung #include.

```
...
# Deklaration Modulglobaler Variablen
$var = wert;
...
# Einen wahren Rueckgabewert erzeugen
return 1;

# Deklaration der Unterroutinen (Methoden) des Paketes

sub unterroutine1 {
    ...
}

sub unterroutine2 {
    ...
}
...
```

Der use-Befehl wird bereits zum Zeitpunkt des *Kompilierens*[7] der Quellcode-Datei ausgeführt. Kann Perl also das gewünschte Paket nicht korrekt einbinden, wird die Anwendung gar nicht erst gestartet. So können Sie sicher sein, dass Sie nicht zur Laufzeit Ihrer Anwendung eine unangenehme Überraschung erleben, weil ein einzubindendes Paket nicht gefunden wurde.

3.3.2 Die Anweisung `require`

Wie oben beschrieben, können mit Hilfe des use-Befehls nur Dateien eingebunden werden, welche die Dateinamenserweiterung `.pm` besitzen; zudem ist es nicht möglich, Pakete dynamisch zur Laufzeit zu laden. Beides wird mit Hilfe der Anweisung `require` möglich. Um beispielsweise eine Datei mit Namen `Kontoverwaltung.pl` in Ihre Anwendung einzubinden, verwenden Sie die folgende Anweisung.

```
require Kontoverwaltung;
```

Sofern Sie keine Dateinamenserweiterung angeben, nimmt `require` als Endung `.pm` an. Auch für die Anwendung von `require` ist es notwendig, dass die zu ladende Datei einen wahren boolschen Wert zurückgibt.

Die Anweisung `require` bietet zudem eine zweite Syntax-Variante an.

```
require "Kontoverwaltung.pm";
```

Diese Variante unterscheidet sich von der ersten in einigen Punkten. So ist es nun möglich, Dateien einzubinden, deren Datei-Endung nicht `.pm` lautet, also beispielsweise `Kontoverwaltung.pl`. Ein weiterer wichtiger Unterschied ist, dass diese Variante das einzubindende Paket nur im aktuellen Verzeichnis findet. Liegt das Paket an einem

[7] Genau genommen ist der Zeitpunkt des Einbindens bereits die *Parsing-Phase*, die noch vor dem eigentlichen Kompiliervorgang durchlaufen wird.

anderen Ort, so muss der Pfad explizit, wie im folgenden Beispiel gezeigt, angegeben werden.

```
require "/home/micky/kontoPackages/Kontoverwaltung.pm";
```

Die Anweisung `require` wird erst zur Laufzeit der Anwendung ausgeführt. Es ist also möglich, dynamisch Code in Ihre Applikation zu laden. Dies ist vor allem dann sinnvoll, wenn bestimmte Programmfunktionen nur sehr selten aufgerufen werden, wie beispielsweise eine Setup-Routine oder Ähnliches.

Beim Aufruf prüft `require` zunächst, ob das gewünschte Paket bereits geladen wurde. Ist dies der Fall, wird der Aufruf ignoriert. Ist dies nicht der Fall, versucht die Anwendung das angegebene Paket zu laden. Der Entwickler selbst ist dafür verantwortlich, sicherzustellen, dass das angeforderte Paket auch existiert. Ist dies nicht so, kommt es zu einem *Programmabsturz*, das heißt, Perl beendet mit einem Fehler die Ausführung Ihrer Anwendung.

Wenn es also nicht notwendig ist, Pakete dynamisch zu laden oder Pakete einzubinden, deren Dateinamenserweiterung nicht auf `.pm` endet[8], sollten Sie in jedem Fall die `use`-Anweisung verwenden.

3.3.3 Wo Perl nach Paketen sucht

Nicht immer möchten Sie die einzubindenden Dateien im aktuellen Programmverzeichnis ablegen. Schon nach kurzer Zeit hätten Sie bei häufig verwendeten Paketen ähnliche Probleme wie bei der (nun obsoleten) Verwendung der Copy & Paste-Methode. Es stellt sich also die Frage, wo Perl Pakete sucht, die Sie mit `use` oder `require` einbinden möchten.

Falls der Perl-Interpreter die angegebenen Dateien nicht im aktuellen Anwendungsverzeichnis findet, sucht Perl in allen im Array `@INC` eingetragenen Verzeichnissen nach dem geforderten Paket. Sofern Sie ein eigenes Bibliotheksverzeichnis anlegen möchten, müssen Sie dieses Array um Ihre Pfadangaben erweitern. Hierzu gibt es im Wesentlichen drei Möglichkeiten:

1. Der einfachste Weg ist der, den Kommandozeilenschalter `-I`[9] zu verwenden.

   ```
   perl -I/home/micky/perlPackages -I/home/micky/kontoPackages
   meinProgramm.pl
   ```

   Wie gezeigt, geben Sie also beim Aufruf Ihrer Anwendung vor dem Namen des auszuführenden Programmes die Pfade zu Ihren Paket-Bibliotheken an. Rufen Sie die Anwendung des Öfteren auf, ist es sinnvoll, ein kurzes Shellskript oder unter Windows eine Batch-Datei zu schreiben, die den Aufruf für Sie erledigt.

[8] Und von Ihnen auch nicht umbenannt werden kann.

[9] Für include = engl. *einschließen*

2. Wenn Sie sehr viele Bibliotheken verwenden, wird es mit der Zeit müßig, bei jedem Aufruf erneut den Rattenschwanz von Include-Pfaden anzugeben. Sofern Sie stets die gleichen Bibliotheks-Pfade verwenden, können Sie diese auch dauerhaft in der Umgebungsvariablen `PERL5LIB` hinterlegen. Diese Variable enthält eine durch Doppelpunkte getrennte Pfadliste. Dabei ist zu beachten, dass Unterverzeichnisse zu den angegebenen Pfaden automatisch mit eingeschlossen werden. Um diese Variable auf einem UNIX/LINUX-System zu verändern, tragen Sie die Ergänzungen in eine der Dateien `.profile` oder `.bashrc` des jeweiligen Users oder in die systemglobale Datei `profile` im Verzeichnis `/etc` in der folgenden Art ein.

```
PERL5LIB="/home/micky/kontoPackages:
/home/micky/perlPackages:$PERL5LIB"
export PERL5LIB
```

Nutzer von Windows-Systemen müssen die entsprechenden Eintragungen in der Datei `autoexec.bat` vornehmen, die im Rootverzeichnis des Laufwerkes `C:\`zu finden sind.

Bevor Sie die Umgebungsvariable `PERL5LIB` dauerhaft verändern, sollten Sie aber wirklich sicher sein, was Sie tun! Im Zweifelsfalle werfen Sie einen Blick in das Handbuch Ihres Betriebssystems.

3. Schließlich können Sie auch das Array `@INC` ergänzen, bevor Sie `require` oder `use` aufrufen.

```
# @INC anpassen
unshift(@INC, "/home/micky/perlPackages");
unshift(@INC, "/home/micky/kontoPackages");

# Pakete einbinden
require "Kontoverwaltung.pm";
require "InputTools.pm";
```

Um `use` zu verwenden, müssen Sie das Array, wie im Abschnitt "Initialisieren und Zerstören von Paketen", verändern, bevor die Anwendung kompiliert wird.

3.4 Zugriff auf Bezeichner anderer Pakete

Sie sind nun in der Lage, Packages zu erstellen und in Ihre Anwendung einzubinden. Was allerdings noch fehlt, um mit Paketen sinnvoll arbeiten zu können, ist eine Möglichkeit, auf die Bezeichner anderer Pakete Zugriff zu erlangen.

Das Geheimnis, um dies zu erreichen, ist der sogenannte *voll qualifizierte Name*, der aus Typenzeichen und dem Namen des Paketes, gefolgt von einem zweifachen Doppelpunkt sowie dem Namen des Bezeichners besteht. Also in allgemeiner Form:

`[Typenzeichen]Paketname::bezeichner`

Also beispielsweise zum Zugriff auf die Variable `$kontostand` aus dem Paket `Konto`.

```
$Konto::kontostand;
```

Analog geschieht auch der Zugriff auf Array- oder Hash-Variablen.

```
@Konto::buchungsdaten[0];
%Konto::kontoInhaberNamen{"Maus"};
```

Beachten Sie hierbei, dass das Typenzeichen stets vor dem Paketnamen steht und nicht vor dem Variablen-Bezeichner!

Um eine Subroutine in einem anderen Paket aufzurufen, verwenden Sie die folgende Syntax.

`Paketname::subroutine(argument, argument[,...])`

Also beispielsweise.

```
Konto::einzahlen($aBetrag);
```

Lassen Sie uns dieses Vorgehen anhand eines Beispieles illustrieren. Wir gehen davon aus, dass Sie ein Paket `Konto` erstellt haben, in dem die Variablen `kontoinhaber` und `kontostand` sowie die beiden Subroutinen `einzahlen` und `auszahlen` deklariert sind. Beide Subroutinen erwarten als Argument einen Betrag. Zunächst erstellen wir das Paket `Konto` wie folgt.

```
package Konto;

# Datei: Konto.pm

#  Globale Variablen deklarieren
$kontoinhaber = "";
$kontostand = 0.0;

# ein 'wahrer' Wert muss zurueckgegeben werden
return 1;

sub einzahlen{
    # Einzahlungsbetrag uebernehmen
    my $betrag = $_[0];

    # Wenn Einzahlung groesser als 0, buchen
    if ($betrag > 0.0) {
        $kontostand += $betrag;
    } else {
```

```
        print "Sie muessen einen Buchungsbetrag angeben!\n";
    }
}

sub auszahlen{
    # Auszahlungsbetrag uebernehmen
    my $betrag = $_[0];

    # Wenn Auszahlung moeglich, Betrag abbuchen
    if ($betrag < $kontostand) {
        $kontostand -= $betrag;
    } else {
        print "Sie haben nicht genug Geld auf Ihrem Konto\n";
        print "Auszahlung nicht moeglich!\n";
    }
}
```

Nun schreiben wir eine kurze Anwendung, mit der wir einige Variablenwerte setzen und die Subroutinen aufrufen.

```
#!/usr/bin/perl -w

# Paket einbinden
use Konto;

# Kontoinhaber setzen
print "Geben Sie den Namen des Kontoinhabers ein: ";
my $name = <STDIN>;
chomp($name);
$Konto::kontoinhaber = $name;

# Eine Einzahlung auf das Konto vornehmen
print "Geben Sie einen Einzahlungsbetrag ein: ";
my $eBetrag = <STDIN>;
chomp($eBetrag);
Konto::einzahlen($eBetrag);

# Eine Auszahlung vornehmen
print "Geben Sie den Auszahlungsbetrag ein: ";
my $aBetrag = <STDIN>;
chomp($aBetrag);
Konto::auszahlen($aBetrag);

# Daten ausgeben
print "\n\n$Konto::kontoinhaber hat $Konto::kontostand " .
"Euro auf seinem Konto\n";
```

Wenn Sie die Skripte abtippen und starten, wird die in Abbildung 3.1 dargestellte Ausgabe erzeugt. Sie sollten dabei aber unbedingt beachten, dass beide Skripten im selben Verzeichnis stehen, sofern Sie nicht die Umgebungsvariable `@INC` verändern möchten.

***Abb. 3.1:** Ausgabe des Konto-Test-Skriptes*

3.5 Pakete initialisieren und terminieren

Als wir uns mit der Frage beschäftigt haben, wo Perl einzubindende Pakete findet, wurde angedeutet, dass es auch möglich sei, die Array-Variable `@INC` so zu verändern, dass Pakete mit der `use`-Anweisung eingebunden werden können. Wenn Sie die folgenden Anweisungen in Ihre Anwendung aufnehmen, produzieren Sie allerdings eine dicke Fehlermeldung.

```
# Das folgende Code-Fragment produziert eine huebsche Fehlermeldung!!
unshift(@INC, "/home/micky/perlPackages");
unshift(@INC, "/home/micky/kontoPackages");

# Hier wird's falsch!!
use "Kontoverwaltung.pm";
use "InputTools.pm";
```

Der Grund für dieses Verhalten ist recht einfach und wurde bereits erläutert. Der `use`-Befehl versucht nämlich, das gewünschte Paket bereits zum Zeitpunkt des Kompilierens

zu laden. Für unser Skript bedeutet dies, dass nicht die `unshift`-Anweisungen zuerst ausgeführt werden, sondern bereits vorher versucht wurde, die `use`-Anweisungen auszuführen. Dies führt folglich zu einem Fehler und zum Abbruch der Programmausführung.

Aber Perl wäre nicht Perl und damit die Lieblingssprache des Autors, wenn es nicht auch hier einen Ausweg gäbe. Trifft Perl beim Kompilieren einer Anwendung auf einen Block `BEGIN` oder eine Subroutine dieses Namens, wird der darin enthaltene Code sofort, also noch während der Kompilierungsphase ausgeführt. Probieren Sie einmal das folgende Beispiel aus[10].

```
# der folgende Code wird zur Kompilierzeit ausgefuehrt
sub BEGIN { # oder einfach BEGIN
    unshift(@INC, "/home/micky/perlPackages");
    unshift(@INC, "/home/micky/kontoPackages");
}

# und siehe da, es funktioniert
use "Kontoverwaltung.pm";
use "InputTools.pm";
```

Falls Ihnen dieses Beispiel zu kompliziert war, lassen Sie uns noch ein einfacheres versuchen.

```
# der folgende Code wird zur Kompilierzeit ausgefuehrt
BEGIN {
    print "\nEinen Moment bitte, ich kompiliere die Anwendung...\n";
}

# und eine unkorrekte Anweisung um die Kompilierung abzubrechen
diese Anweisung ist der totale Bloedsinn;
```

Nachdem Sie das Skript eingegeben und gestartet haben, erwartet Sie die in Abbildung 3.2 gezeigte Ausgabe.

Wie Sie sehen, wurde die Anweisung innerhalb des `BEGIN`-Blockes anstandslos ausgeführt, obwohl das Programm einen kapitalen Syntax-Fehler enthält und noch während der Kompilierungsphase abgebrochen wird.

Wir haben Ihnen mit unserem ersten Beispiel bereits einen möglichen Anwendungsfall für einen `BEGIN`-Block gezeigt. Besonders bei größeren Applikationen kann es immer wieder notwendig oder einfach nur bequem sein, Initialisierungen wie die Veränderung der Spezialvariablen `@INC` um spezifische Pfadangaben des Benutzer-Systems zu setzen.

[10] Beachten Sie aber, dass die angegebenen Pfade und Pakete an der entsprechenden Stelle vorhanden sein müssen.

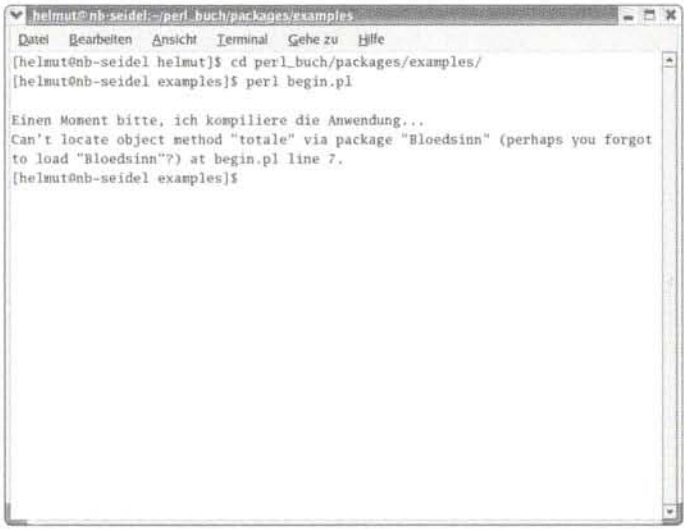

***Abb. 3.2:** Ausgabe des Beispielskriptes begin.pl*

Betrachten Sie hierzu auch das folgende Beispiel, das der Initialisierung einer Anwendung entspricht, wie sie der Autor für Kunden erstellt, bei denen er zur Programmierzeit nicht die Pfadangaben auf dem Zielsystem kennt.

```
# Initialisierungs-Routine einer groesseren Anwendung
sub BEGIN {
    # eine .ini-Datei im aktuellen Anwendungsverzeichnis oeffnen
    open (IN, "begin2.ini";
    $pfad = <IN>;
    chomp($pfad);
    close IN;

    # Variable @INC ergaenzen
    unshift(@INC, $pfad);

    # Kontrollausgabe - in der fertigen Anwendung loeschen
    print "\n\$pfad in der BEGIN-Routine: $pfad\n";
}

# Kontrollausgabe - in der fertigen Anwendung loeschen
print "\pfad im Hauptprogramm: $pfad\n;
```

Wie Sie sehen, können in einer `BEGIN`-Routine nicht nur Perls Spezialvariablen verändert werden, sondern auch "ganz normale" Variablen. Interessant ist dabei, dass Perl sich

die in der Initialisierungs-Routine gesetzten Variablenwerte auch im Hauptprogramm merkt! Gerade bei großen Anwendungen, die möglicherweise noch an mehrere Kunden mit unterschiedlichen Betriebssystemen ausgeliefert werden, ist dieses Verhalten Gold wert.

Bei komplexen Anwendungen kann es auch notwendig sein, nach Beenden der Applikation Code auszuführen, um beispielsweise Programmzustände auf der Festplatte zu sichern. Beim "planmäßigen" Ende der Applikation ist dies auch kein Problem: Sie schreiben einfach eine Unterroutine, die aufgerufen wird, bevor die Anwendung ihre Arbeit beendet. Doch leider leben wir nicht in dieser heilen Welt, wo Anwendungen stets zur rechten Zeit unter idealen Umständen enden. Vielmehr kommt es immer wieder zum abrupten Abbruch von Programmen, weil bestimmte Voraussetzungen für die Applikation plötzlich nicht mehr erfüllt sind.[11] In diesem Falle kann die Routine zum Beenden der Anwendung natürlich nicht mehr aufgerufen werden. Für derartige Fälle bietet Perl den `END`-Block an. Die Anweisungen innerhalb dieses Blockes werden immer ausgeführt, bevor die Anwendung die Kontrolle an das Betriebssystem zurück reicht, auch dann, wenn die Applikation mit einem Fehler beendet wurde. Betrachten Sie hierzu das folgende Beispiel.

```
# Ein unsinniger Befehl, um die Anwendungsausfuehrung zu beenden
dies ist ein vollkommen unsinniger Befehl;

sub END {
    # Die folgenden Anweisungen werden noch ausgefuehrt,
    # obwohl die Anwendung mit einem Fehler beendet wird
    print "\nDie Anwendung hat einen Fehler verursacht!\n";
    print "Die Anwendung wird beendet!\n";
}
```

Wie Sie sehen, werden die Anweisungen innerhalb des `END`-Blockes noch ausgeführt, obwohl die Anwendung einen Fehler erzeugt hat. Bedenken Sie aber, bevor Sie Ihre Anwendungen nun mit `END`-Blöcken spicken, dass dies stets nur der letzte Ausweg sein sollte und Sie nicht davon befreit, sinnvolle *Fehlerabfang-Routinen* zu schreiben!

3.6 Symbole aus anderen Symboltabellen importieren

Mitunter kann es recht lästig werden, immer dann, wenn man ein neues Konto anlegen möchte, den folgenden Rattenschwanz eintippen zu müssen.

```
Kontoverwaltung::neu('Maus');
```

Viel angenehmer wäre es, könnte man einfach

[11] Also weniger poetisch ausgedrückt: weil ein Fehler aufgetreten ist!

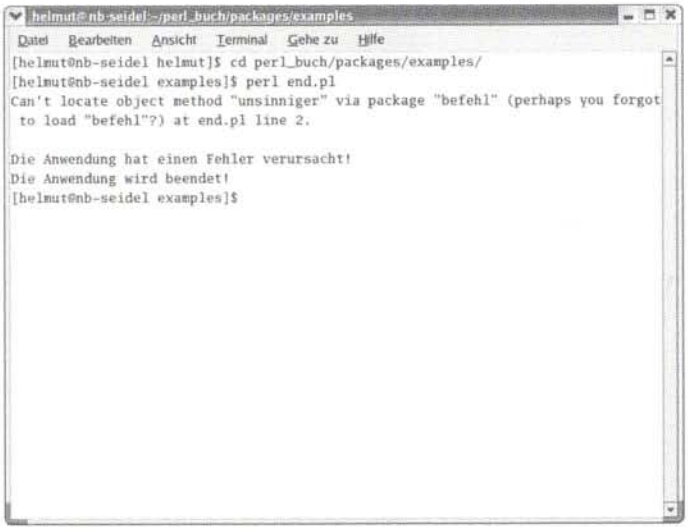

Abb. 3.3: *Die Ausgabe des Beispielskriptes* end.pl

```
neu('Maus');
```

verwenden. Und genau dies kann erreicht werden, indem man zum use-Befehl in einer optionalen Liste angibt, welche Symbole in den aktuellen Namensraum importiert werden sollen.

```
use Kontoverwaltung('neu', 'suchen', 'loeschen');
neu('Maus); # statt Kontoverwaltung::neu('Maus');
```

Damit man allerdings bei use eine Liste der zu importierenden Namen angeben kann, muss das Paket, das diese Liste liefert, in die Lage versetzt werden, die Namen zu *exportieren*. Zusätzlich muss das Paket wissen,welche Symbole zu exportieren sind, wenn beim use-Befehl keine Liste angegeben wurde. Um diese Aufgabe zu erledigen, stellt Perl das Standard-Modul Exporter zur Verfügung. Mit Hilfe dieses Moduls kann das Paket Kontoverwaltung auf folgende Weise realisiert werden.

```
package Kontoverwaltung;

use Exporter; # Modul Exporter einbinden
@ISA = ('Exporter'); # Von Exporter erben
# Siehe hierzu Abschnitt zur Objektorientierten Programmierung

# Liste der Symbole, die exportiert werden koennen
```

```
@Export_Ok = ('neu', 'suchen', 'loeschen');

# Implementierung der Funktionen
sub neu {
    ...
}

sub suchen {
    ...
}

sub loeschen {
...
}
```

Das Array `@ISA` dient dazu, dass das Paket `Kontoverwaltung` seine Methoden und Eigenschaften[12] vom Modul `Exporter` erbt. Die vielen neuen Begriffe werden wir weiter unten, wenn wir uns mit der Objektorientierten-Programmierung beschäftigen, ausführlich erklären. Wir bitten um ein wenig Geduld. Wichtig für unsere Problemstellung ist vor allem die Deklaration des Arrays `@Export_Ok`, das angibt, welche Symbole exportiert werden dürfen. Der Aufrufer kann eines, mehrere oder alle genannten Funktionsnamen in seinen Namensraum importieren.

```
use Kontoverwaltung('neu'); # nur ein Funktionsname wird importiert
```

Neben dem Array `@Export_OK` kann auch ein zweites Array `@Export` definiert werden, das angibt, welche Symbole exportiert werden, wenn der Aufrufer den `use`-Befehl ohne eine Liste mit Symbol-Namen verwendet.

```
...
# Symbole die standardmaessig exportiert werden
@Export = ('neu', 'suchen');
# Symbole die nur exportiert werden, wenn sie explizit angefordert werden
@Export_Ok('loeschen');
```

Binden Sie das Modul einfach mit der Anweisung

```
use Kontoverwaltung;
```

ein, dann können Sie mit der folgenden Anweisung ein neues Konto anlegen.

[12] Beide Begriffe werden weiter unten erläutert, als Eselsbrücke sei nur erwähnt: *Methoden* sind die Funktionen einer Klasse, *Eigenschaften* die Variablen.