Jürgen Bayer

Grundlagen der objektorientierten Programmierung

Von einfachen Klassen zum Polymorphismus

Inhaltsverzeichnis

1 Einführung in die OOP

Dieser Artikel beschreibt die grundlegenden Konzepte der objektorientierten Programmierung (OOP). Dabei gehe ich einen eher praxisorientierten als theoretischen Weg. In der OOP-Theorie wird vieles diskutiert, was in der Praxis wenig oder oft auch gar keine Bedeutung hat. Diese teilweise sehr theoretischen Themen lasse ich einfach weg. Trotzdem behandelt dieser Artikel die objektorientierte Programmierung recht ausführlich.

Ich setze dabei voraus, dass Sie die Grundlagen der »normalen« Programmierung (Variablen, Schleifen, Verzweigungen, Funktionen, Prozeduren etc.) einigermaßen beherrschen und nun die OOP kennen lernen wollen. Um objektorientiert programmieren zu lernen müssen Sie nicht unbedingt vorher schon sehr viel prozedural programmiert haben. Sie sollten aber die *Grundlagen* kennen. Die objektorientierte Programmierung basiert (natürlich) auf diesen Grundlagen.

Dieses Kapitel klärt die in der OOP verwendeten Begriffe. Hier erfahren Sie zunächst, was ein Objekt überhaupt ist. Nach dieser Definition erläutere ich, was die objektorientierte von der strukturierten Programmierung unterscheidet. Außerdem führe ich einige Beispiele auf, die die Vorteile der OOP deutlich machen. Dieses Kapitel liefert Ihnen damit einige Argumente für die Verwendung der OOP an Stelle der strukturierten Programmierung.

In **Kapitel 2** erfahren Sie dann, wie Sie in verschiedenen gängigen Programmiersprachen eine Klasse programmieren und aus dieser Klasse Objekte erzeugen. Daneben werden wichtige Basiskonzepte der OOP (wie Kapselung, Konstruktoren und Destruktoren) näher beleuchtet.

Ab **Kapitel 3** geht es dann ans »Eingemachte«. Dieses Kapitel beschreibt zwei wichtige Konzepte der OOP, die so genannte *Vererbung* und den *Polymorphismus*.

In **Kapitel 4** werden schließlich einige spezielle Features der OOP, wie z. B. *virtuelle Methoden*, *statische Klassenelemente* und *Schnittstellen* beschrieben, die Sie kennen sollten, wenn Sie die Möglichkeiten der OOP wirklich ausnutzen wollen.

In diesem Artikel fehlen dann noch einige spezielle Themen, wie z. B. *Klassenhierarchien*, *Komponenten*, *Objekt-Auflistungen* und das Speichern von Objekten in Dateien und Datenbanken, für die kein Platz blieb.

> Dieser Artikel beschreibt die OOP-Konzepte nicht für die Scriptsprachen VBScript und JavaScript. In VBScript ist es zwar möglich, einfache Klassen zu erzeugen, die Basiskonzepte der OOP (wie Konstruktoren, Destruktoren, Vererbung etc.) werden jedoch, so weit mir bekannt ist, nicht unterstützt. JavaScript unterstützt zwar die Basiskonzepte der OOP, jedoch auf eine sehr undurchsichtige Art und Weise. Diese beiden Sprachen sind außerdem ja immer noch Scriptsprachen. Dass die Verwendung der objektorientierten Programmierung in (meist sehr einfach gestalteten) Scripting-Programmen Sinn macht, stelle ich sehr in Zweifel.

1.1 Eine Definition des Begriffs »Objekt«

Abstrakt betrachtet ist ein Objekt eine geschlossene, aus Daten und Operationen bestehende Einheit. Die Daten eines Objekts können von außen meist gelesen und oft auch geschrieben werden. Operationen werden von außen aufgerufen, um mit dem Objekt oder seinen Daten zu arbeiten. Diese abstrakte Sicht auf Objekte ist uns Menschen sehr nahe, auch wenn es auf den ersten Blick nicht so aussieht.

Alles, was uns in der realen Welt umgibt, kann normalerweise als Objekt betrachtet werden. Viele einfache Objekte, wie z. B. eine Kaffeetasse oder ein Buch, sind zwar real vorhanden und können von uns in die Hand genommen – quasi »erfühlt« – werden, bestehen aber – abstrakt betrachtet – eigentlich nur aus Daten. Die wichtigsten Daten einer einfachen Kaffeetasse sind

zum Beispiel die *Höhe*, der *Durchmesser*, die *Farbe* und das *Muster*. Da Kaffeetassen in der Realität jedoch sehr viele unterschiedliche Formen besitzen können, sind wesentlich mehr Daten notwendig, um eine Kaffeetasse zu beschreiben. Im Prinzip kann man aber alle Objekte über ihre Daten beschreiben. In der OOP werden diese Daten im Allgemeinen als *Eigenschaften* bezeichnet.

Solch einfache Objekte wie eine Kaffeetasse bieten uns (zurzeit noch) keine Operationen zur Arbeit mit sich selbst oder ihren Daten (Eigenschaften) an. Das wird sich für einige Objekte in Zukunft wahrscheinlich einmal ändern, wenn Kaffeetassen zum Beispiel auf Befehl endlich selbst an den Mund fliegen, den Kaffee in unseren Mund ausleeren und sich anschließend dafür bedanken, uns bedient haben zu dürfen. Eine heutige Kaffeetasse besitzt aber nur Eigenschaften. Schauen Sie sich einmal in Ihrer Umgebung um und Sie werden noch viel mehr *Objekte* mit ganz unterschiedlichen *Eigenschaften* sehen.

Die meisten Objekte unserer Umgebung kommen allein mit Eigenschaften aus. Meist handelt es sich dabei um einfach zu verwendende Objekte, wie z. B. Kaffeetassen oder Türen (die übrigens meist von anderen, komplexeren Objekten – in diesem Beispiel sind das Menschen – verwendet werden). Bei einigen Objekten erwarten wir aber nicht nur das Vorhandensein von Eigenschaften, sondern auch, dass das Objekt *eigene Operationen* ausführen kann. Ein Auto besitzt z. B. sehr viele Eigenschaften, wie die *Leistung*, die *Farbe*, die *Reifengröße* etc. Würde ein Auto allerdings nur aus Eigenschaften bestehen, könnten wir nicht allzu viel damit anfangen. Von einem Auto erwarten wir, dass dieses Operationen besitzt, mit denen es gestartet, beschleunigt, gelenkt und gebremst werden kann. Diese Operationen auf einem Objekt werden in der OOP im Allgemeinen als *Methoden* bezeichnet.

Und genau das beschreibt den zentralen Punkt der OOP: Ein Objekt besitzt eigene Eigenschaften (Daten) und eigene Methoden (Operationen). Methoden können von außen (im Fall des Autos vom Menschen) aufgerufen werden, Eigenschaften können (wenigstens im Computer) vom Benutzer des Objekts gesetzt und abgefragt werden. Für ein im Computer abgebildetes Auto könnten Sie z. B. die Eigenschaft *Leistung* erhöhen, wenn das Auto zu langsam ist. Im realen Leben ist dies nicht so ohne weiteres möglich. Es gibt aber auch Eigenschaften, die auch im realen Leben geändert werden können. Bei einem Auto können Sie z. B. den Wert der Eigenschaft *Tankfüllgrad* durch das Tanken von Benzin oder Diesel erhöhen.

Ein Objekt besteht also grundsätzlich aus Eigenschaften und Methoden. Die folgenden Seiten beschreiben Eigenschaften und Methoden genauer und zeigen, wie diese für die Objekte, die wir in der Programmierung verwenden, eingesetzt werden.

Eigenschaften

Eigenschaften beschreiben das Aussehen und/oder die Funktionsweise eines Objekts. Dieser einfache Satz sagt alles über die Grundbedeutung von Eigenschaften aus. Beim Auto beschreibt z. B. die `Farbe` das Aussehen und die `Leistung` des Motors die Funktionsweise.

Dass eine Eigenschaft wie z. B. die *Farbe* das Aussehen beschreibt, brauche ich, wie ich denke, wohl nicht weiter zu erklären. Das Verständnis, dass es auch Eigenschaften gibt, die die Funktionsweise eines Objekts beeinflussen, ist dagegen schon etwas schwieriger, aber eigentlich auch schnell an Beispielen erklärt: Die Eigenschaft *Leistung* eines Autos sagt z. B. aus, wie stark das Auto beschleunigt und welche Endgeschwindigkeit es erreichen kann. Diese Eigenschaft beschreibt also das Verhalten eines Autos (möglicherweise auch das Aussehen, nämlich dann, wenn der Fahrer die Leistung zu sehr ausschöpft und sein Auto gegen einen Baum fährt).

Sie finden diese Grundaussage über Eigenschaften eigentlich in allen Objekten aus Ihrer Umgebung wieder. Bei einem Schreibstift z. B. beschreiben die Eigenschaften `Länge` und `Durchmesser` das Aussehen und die Eigenschaft `Tintenfarbe` das Verhalten (in welcher Farbe der Stift schreibt). Wenn Sie sich einfach einmal irgendwelche Objekte in Ihrer Umgebung anschauen, werden Sie feststellen, dass diese meist mehr Eigenschaften besitzen, als Sie zuvor vielleicht dachten, und dass die Aussage am Anfang dieses Abschnitts eigentlich immer passt.

Eigenschaften eines Objekts werden in fast allen Programmiersprachen mit derselben Syntax beschrieben und ausgelesen. Dabei wird zuerst der Name des Objekts geschrieben, gefolgt von einem Punkt (dem *Verweisoperator*) und dem Namen der Eigenschaft. Die Farbe eines Autos könnten Sie also z. B. so setzen:

```
MeinAuto.Farbe = ROT;
```

(wobei *ROT* hier als symbolische Konstante verstanden werden soll), und so auslesen:

```
AutoFarbe = MeinAuto.Farbe;
```

(wobei *AutoFarbe* hier eine einfache Variable ist).

> Bis auf die Tatsache, dass bei Eigenschaften der Objektname und der Punkt vor dem Eigenschaftennamen stehen, unterscheiden sich Eigenschaften beim Zugriff syntaktisch nicht von einfachen Variablen

Bei Objekten der Realwelt können Eigenschaften nur sehr selten von außen beschrieben werden. Die Farbe eines Autos kann z. B. (leider) nur gelesen, und nicht geschrieben werden. Bei Objekten im Computer sind solche *schreibgeschützten* Eigenschaften auch möglich, aber wesentlich seltener anzutreffen. Bei dem Abbild eines Autos im Computer macht es z. B. viel Sinn, dass der Programmierer oder der Anwender auch die Farbe des Autos einstellen kann.

Methoden

Objekte, die keine Methoden besitzen, sind entweder lediglich mehr oder weniger schön anzuschauen oder einfach zu benutzende Objekte, wie z. B. Kaffeetassen oder Hämmer. Objekte, die benutzt werden und dabei irgendwelche Arbeiten selbst erledigen sollen (wie z. B. ein Auto), müssen hingegen Methoden besitzen. Über Methoden können wir mit dem Objekt arbeiten. Anders gesagt: Wir lassen das Objekt irgendwelche Aktionen ausführen, *für dessen Abarbeitung das Objekt erzeugt (bzw. konstruiert oder gebaut) wurde*.

Die kursive Hervorhebung im letzten Satz ist für die OOP schon recht wichtig. Wir können mit dem Objekt über seine Methoden nur Operationen ausführen, die das Objekt uns anbietet. Ein Auto kann man *starten*, *lenken*, *bremsen* etc., aber nicht *fliegen* (außer von der Brücke herunter). Das macht Sinn, denn ein Auto ist eben ein Auto.

Beim Auto, dessen grundsätzliche Bedeutung ja ist, gefahren zu werden, können wir mit der `Starten`-Methode den Motor anlassen, mit der `Kuppeln`-Methode die Kupplung betätigen, mit der `Gangschaltung`-Methode einen Gang einlegen, mit der `Beschleunigen`-Methode den Motor beschleunigen, mit der `Lenken`-Methode lenken und mit der `Bremsen`-Methode abbremsen. Stellen Sie sich ein Auto ohne Methoden vor. Was können Sie mit diesem Auto noch anfangen?

Bei der Programmierung werden Methoden in allen mir bekannten Programmiersprachen über dieselbe Syntax aufgerufen. Wie bei Eigenschaften schreiben Sie zuerst den Namen des Objekts gefolgt vom Verweisoperator (dem Punkt) und dem Namen der Methode. Ein Auto könnte in einer Programmiersprache also z. B. so gestartet werden:

```
MeinAuto.Starten();
```

> Bis auf die Tatsache, dass der Objektname und der Punkt vor dem Methodennamen stehen, unterscheidet sich die Syntax des Methodenaufrufs nicht von der Syntax eines Funktions- oder Prozeduraufrufs in der jeweiligen Programmiersprache.

Viele Methoden besitzen, wie normale Funktionen und Prozeduren auch, Argumente, über die deren Ausführung gesteuert werden kann. Bei der Ausführung der *Starten*-Methode eines Autos wird z. B. nichts weiter berücksichtigt, der *Lenken*-Methode müssen wir aber übergeben, in welche Richtung und wie stark gelenkt werden soll. Sie übergeben der Methode dazu Argumente, genau wie Sie es bereits bei normalen Prozeduren und Funktionen machen.

Angenommen, die Lenken-Methode besitzt ein Argument für die Richtung (mit den Werten –30 bis +30, wobei –30 ganz links, 0 geradeaus und +30 ganz rechts bedeutet), können Sie folgendermaßen etwas nach links lenken:

```
MeinAuto.Lenken(-10);
```

Denken Sie daran, dass sich Methoden im Prinzip genauso verhalten wie Prozeduren oder Funktionen[1]. Der Unterschied zu diesen ist lediglich, dass Prozeduren und Funktionen für Ihr Programm einen allgemeinen Charakter besitzen und Methoden immer im Kontext eines Objekts aufgerufen werden und sich eigentlich immer (sofern das Objekt konsequent den Regeln der OOP folgt) auf das Objekt beziehen.

Was bei dem hier verwendeten Auto-Beispiel schlecht dargestellt werden kann, ist die Tatsache, dass Methoden auch dazu benutzt werden können, um auf die Daten des Objekts zuzugreifen. Diese Technik wird häufig dazu verwendet, dass ein Objekt den Zugriff auf seine Daten kontrolliert und somit z. B. das Schreiben unzulässiger Werte nicht erlaubt. Dieses Konzept, die so genannte *Kapselung*, beschreibe ich in Kapitel 2 ausführlicher.

Die Klasse als Bauform für Objekte

Wenn Sie in Ihrem Programm Objekte verwenden wollen, müssen Sie diese in der Regel zunächst erzeugen (einige Programmiersprachen bieten Ihnen aber auch einige wenige vordefinierte Objekte an, die Sie nicht erst erzeugen müssen). Beim Erzeugen eines Objekts müssen Sie angeben, aus welcher Klasse Sie dieses erzeugen wollen. Die Klasse ist quasi die Bauform für neue Objekte. In der Klasse ist beschrieben, welche Eigenschaften, Methoden und evtl. Ereignisse[2] ein Objekt besitzt. Aus einer Klasse können beliebig viele Objekte erzeugt werden. Alle diese Objekte gehören dann derselben Klasse an, besitzen also dieselben Eigenschaften, Methoden und Ereignisse. Beachten Sie dabei, dass die verschiedenen Objekte einer Klasse wohl dieselben Eigenschaften besitzen, diese aber ganz unterschiedliche Ausprägungen haben können. Ein Auto kann ja z. B. eine rote, eine gelbe oder irgendeine andere Farbe haben.

Wenn Sie z. B. ein Autorennen-Spiel programmieren, in dem Sie mehrere Autos desselben Typs benötigen, werden Sie *eine* Klasse *Auto* mit den notwendigen Eigenschaften und Methoden erzeugen und aus dieser Klasse dann im Programm *mehrere* Autos erzeugen. Diese Autos besitzen zwar alle dieselben Methoden und Eigenschaften, Sie können die Eigenschaften aber ganz individuell einstellen. Die Eigenschaft *Farbe* des einen Autos wird z. B. auf hellrosa, die des anderen auf dunkelschwarz[3] eingestellt. Genauso können Sie z. B. auch mit der Eigenschaft *Leistung* verfahren, die Sie bei dem einen Auto auf 250 KW und bei dem anderen auf 400 KW einstellen.

Sie müssen Klassen aber nicht immer selbst programmieren, viele Programmiersprachen liefern vordefinierte Klassen mit, die entweder im Quellcode oder in Klassenbibliotheken[4] vorliegen. Sie müssen die Quellcode-Klasse bzw. die Bibliothek in Ihr Projekt einbinden und können dann daraus Objekte erzeugen. Diese Klassen sind meist an eine bestimmte Programmiersprache gebunden, das heißt, Sie können z. B. nicht eine Klasse in C++ entwickeln und dann in Visual Basic verwenden.

---

[1] Methoden können wie eine Funktion auch Werte zurückgeben

[2] Ereignisse gehören zur ereignisorientierten Windows-Programmierung und werden aufgrund dieser Spezialisierung in diesem Buch nicht beschreiben.

[3] Wenn Sie sich jetzt über diesen Begriffen wundern: Ich habe eine ganze Zeit lang meine Motorräder selbst, vorwiegend schwarz, lackiert. Es gibt eigentlich kein Schwarz, das wirklich schwarz ist. Mal besitzt das Schwarz einen leichten Braunstich, mal einen Blaustich usw.

[4] Eine Klassenbibliothek ist, wie der Name schon sagt, eine Bibliothek, die Klassen enthält. Klassenbibliotheken sind in der Regel vorkompiliert und können in Programme eingebunden werden, damit der Programmierer aus den enthaltenen Klassen Objekte erzeugen und mit diesen Objekten arbeiten kann.

Eine modernere Art der Klassenbibliothek liefert das Microsoft COM-Konzept, bei dem die Klassen in kompilierten und auf dem System in der Registry[5] eingetragenen DLL- oder EXE-Dateien gespeichert sind. Das COM-Konzept erlaubt aber leider keine echte Vererbung, weswegen dieses für echte OOP nur eingeschränkt verwendbar ist. Die modernste Art der Klassenbibliothek liefert Microsoft mit dem neuen, ab Visual Studio.NET verfügbaren .NET-Konzept, bei dem die Registrierung der Klassen in der Registry wieder entfällt und echte Vererbung möglich ist. COM und .NET besitzen den Vorteil, dass die unterschiedlichsten Programmiersprachen die mit COM bzw. .NET erzeugten Klassenbibliotheken verwenden können, sofern diese nur das entsprechende Konzept unterstützen.

Eine technische Sicht auf Objekte

Technisch betrachtet ist ein Objekt eine Datenstruktur im Arbeitsspeicher. Eigenschaften werden wie Variablen im Speicher angelegt und wie diese beim Lesen und Schreiben entsprechend dem Datentyp interpretiert. Alle Eigenschaften eines Objekts liegen in einem zusammengehörigen Speicherbereich, der bei einer bestimmten Adresse, der Startadresse des Objekts beginnt. Stellen Sie sich alle Eigenschaften eines Objekts einfach direkt hintereinander angelegt im Arbeitspeicher vor. Insofern unterscheiden sich Objekte überhaupt nicht von normalen Strukturen (benutzerdefinierten Typen, Records). Methoden werden übrigens nicht für jedes Objekt einer Klasse separat gespeichert, sondern nur einmal für jede Klasse. Das Kapitel 4 geht noch näher darauf ein.

1.2 Warum OOP?

Jetzt, wo Sie wissen, was ein Objekt ist, stellt sich Ihnen wahrscheinlich die Frage, warum die OOP überhaupt eingesetzt wird (wenn diese Frage für Sie schon geklärt ist, müssen Sie diesen Abschnitt hier nicht lesen). Programme können ja auch prozedural, das heißt mit Funktionen, Prozeduren und Modulen, programmiert werden. Die Gründe für den Einsatz der OOP will ich in diesem Abschnitt klären. Aber zuerst zeige ich die Probleme der strukturierten Programmierung auf, damit Sie die Vorteile der objektorientierten Programmierung besser verstehen.

1.2.1 Die Probleme der strukturierten Programmierung

Prozeduren, Funktionen und Module erleichtern die Programmierung und sorgen – korrekt angewendet – für eine gute Wiederverwendbarkeit, Wartbarkeit und Erweiterbarkeit von Programmen. Trotzdem leidet die strukturierte Programmierung unter einigen Problemen, die besonders bei großen Programmen auftreten. Wie Booch in seinem Buch »Objektorientierte Analyse und Design« zitiert, »scheint die strukturierte Programmierung zu versagen, wenn die Applikationen 100 000 Codezeilen oder mehr umfassen«. Aber die Probleme werden auch schon in kleineren Programmen deutlich (womit ich sagen will: »OOP macht eigentlich immer Sinn«).

Funktionen ohne Kontext

Der Einsatz von Modulen für einzelne Programmteile macht ein Programm übersichtlicher und die einzelnen Programmteile (in anderen Projekten) wiederverwendbar. Eine Aufteilung der Aufgaben eines Moduls in mehrere Funktionen verbessert die Übersichtlichkeit und die Wiederverwendbarkeit und macht ein Programm besser wartbar. Manche Module enthalten nur

[5] die Registry (die auch als Registrierdatenbank bezeichnet wird) ist eine Datei, in der Windows alle möglichen Einstellungen für das System und die installierte Software verwaltet

eine einzige öffentliche Funktion, die von außen aufgerufen werden kann. Diese Funktion verwendet häufig mehrere private Funktionen, die nur in dem Modul gelten.

Eine Anwendung zur Abwicklung von Bestellungen kann z. B. ein Modul zum Ausdrucken von Lieferscheinen mit einer globalen Funktion *Print_DeliveryNote*[6] enthalten. Diese Funktion muss für jede neue Seite einen Seitenfuß und einen Seitenkopf drucken. Der Seitenfuß und der Seitenkopf werden von jeweils einer privaten Funktion gedruckt, die in demselben Modul gespeichert ist. Dieses Aufteilen in einzelne Funktionen macht den Programmteil »Lieferschein drucken« leichter wartbar und übersichtlicher.

Solche *privaten* Funktionen führen meist nicht zu Problemen, weil der Programmierer diese nur im Kontext des Programmteils aufrufen kann, an dem er gerade arbeitet. In Programmen besteht aber auch sehr häufig Bedarf an *globalen* Funktionen, die von verschiedenen Teilen des Programms aus aufgerufen werden können. Wenn Sie vor dem Lesen dieses Artikels bereits selbst prozedural programmiert haben, werden Sie diesen Bedarf kennen. Falls nicht, hilft Ihnen zum Verständnis ein Beispiel: Eine Bestell-Anwendung muss an verschiedenen Stellen Lieferscheine, Rechnungen, Artikellisten und Personallisten drucken. In einem gut modularisierten Programm würde man diese Aufgaben in einzelnen Modulen unterbringen. Ein Modul ist für den Lieferscheindruck zuständig, das andere für den Rechnungsdruck usw. Die einzelnen Funktionen dieser Module müssen einzelne Textzeilen, Linien, evtl. Kreise und vielleicht auch Bilder ausdrucken. Die Funktionen, die Windows zum Drucken bereitstellt, sind aber sehr kompliziert. Falls die Programmiersprache, mit der der Programmierer arbeitet, keine einfach zu handhabenden und ausreichend flexiblen Druckfunktionen anbietet, müssen diese eben selbst programmiert werden (es sei denn, der Programmierer besitzt bereits eine fertige Bibliothek mit Druckfunktionen oder will diese kaufen). Zu diesen Funktionen gehören dann solche zur Ausgabe von Text, zum Zeichnen von Linien, zur Ausgabe von Bildern usw. Die einzelnen Funktionen müssen global deklariert sein, damit sie in den verschiedenen Modulen zum Drucken von Lieferscheinen, Rechnungen usw. verwendet werden können.

Größere Programme enthalten häufig sehr viele globale Funktionen. Das Problem für den Programmierer ist nun, herauszufinden, welche Funktionen er in dem Kontext des Programmteils, an dem er gerade arbeitet, sinnvoll und sicher verwenden kann. Die modulare Programmierung hilft dem Programmierer zwar schon dadurch, dass Funktionen, die in einem gemeinsamen Kontext stehen, in einem Modul zusammengefasst sind. Aber das Herausfinden, welche Funktionen in einem Modul gespeichert sind, ist häufig mit einiger Arbeit verbunden. Viele Programmierer behelfen sich damit, dass der Name zusammengehöriger Funktionen mit einem bestimmten Präfix beginnt. So könnte der Name von Druckfunktionen z. B. mit »prn_« beginnen. Die meisten Entwicklungsumgebungen bieten Tools zur Anzeige der verfügbaren Funktionen an. Mit Hilfe einer solchen Namenskonvention kann der Programmierer Funktionen, die in Zusammenhang stehen, über diese Tools recht schnell finden.

Abgesehen davon, dass es manchmal schwierig ist, die richtige Funktion in einem Modul zu finden, ist der Programmierer aber nicht auf die in Zusammenhang stehenden Funktionen eingeschränkt. Beim Entwickeln des Moduls zum Drucken von Lieferscheinen kann er neben den eigentlichen Druckfunktionen (*Print_Text*, *Print_Circle* etc.) z. B. auch, aus Versehen, eine Funktion *Write_Text* aufrufen, die in einem ganz anderen Modul gespeichert ist und die leider nicht druckt, sondern den übergebenen Text an die serielle Schnittstelle sendet. Dieses Beispiel mag etwas konstruiert sein, stellt aber das Problem recht gut dar. Diese Möglichkeit, an *jeder* Stelle des Programms *alle* globalen Funktionen aufrufen zu können, kann zu schwer wiegenden Fehlern führen: Der Programmierer denkt, er habe die richtige Funktion aufgerufen, verwendet aber eine, die in einem falschen Kontext steht. Die Zuordnung der für den aktuellen Kontext verwendbaren Funktionen ist für den Programmierer also schwierig und fehlerträchtig. Und dieses Problem wird sehr deutlich, wenn mehrere Programmierer gleichzeitig an einem großen Software-Projekt arbeiten.

---

[6] »DeliveryNote« ist der englische Begriff für Lieferschein.

Ungeschützter Zugriff auf globale Daten

Dass bei der strukturierten Programmierung Funktionen nicht in einem Kontext zueinander stehen, ist bereits ein Problem. Ein viel größeres Problem ist aber, dass der Zugriff auf globale Daten nicht geschützt ist. Obwohl bei der strukturierten Programmierung immer versucht wird, möglichst keine globalen Daten zu verwenden, lassen sich diese manchmal nicht vermeiden. Die im vorigen Abschnitt genannten Druckfunktionen sind ein gutes Beispiel dafür. Funktionen zum Drucken sollten so flexibel sein, dass neben den auszudruckenden Daten mindestens auch der Drucker angegeben werden kann, auf dem ausgedruckt werden soll. In der Praxis würden diese Funktionen wahrscheinlich aber auch noch so programmiert werden, dass beim Drucken Ränder berücksichtigt werden. Jede der einzelnen Druckfunktionen muss nun wissen, auf welchem Drucker gedruckt wird und wie die Einstellungen für die Druck-Ränder sind. Die Funktionen könnten dazu einfach mit Argumenten ausgestattet werden. Dummerweise müssen diese Argumente dann aber bei jedem Aufruf der Funktion mit übergeben werden. In der Praxis kann das recht schnell nervig werden. Um sich die Arbeit zu erleichtern, verwalten viele Programmierer solche Einstellungen einfach in globalen Variablen. So müssen die Einstellungen nur einmal vor dem Ausdrucken vorgenommen werden.

Das Problem ist nun, dass diese Variablen nicht nur für die Druckfunktionen global sind, sondern für das gesamte Programm. Angenommen der Ausdruck eines Lieferscheins ist sehr kompliziert und besteht aus sehr vielen einzelnen Funktionsaufrufen. Dann kann es sein, dass mittendrin irgendeine, vielleicht untergeordnete Funktion den Wert einer dieser globalen Variablen ändert. Das Programm läuft in diesem Fall nicht korrekt ab. Der zugrunde liegende Fehler ist in solchen Fällen nur sehr schwer zu finden.

Dieser Nachteil mag Ihnen vielleicht nicht einleuchten, wenn Sie selbst noch nicht viel programmiert haben. In der Praxis ist es aber häufig so, dass ab einer bestimmten Programmgröße niemand mehr sagen kann, welche Funktion welche globale Daten verwendet. Eigentlich ist in strukturierten Programmen alles irgendwie miteinander verknüpft. Jede Funktion kann auf *alle* globalen Daten zugreifen und diese verändern. Die dadurch möglichen Fehler führen oft zu nächtelanger Fehlersuche und lassen Programmierer sehr vorsichtig werden, wenn es darum geht, Programme nachträglich zu verändern. In großen Programmen weiß häufig niemand mehr, was passiert, wenn ein neuer Programmteil geschrieben wird, dessen Funktionen irgendwelche globale Daten verändern. Möglicherweise läuft dann zwar der neu geschriebene Programmteil wunderbar, irgendein anderer aber nicht mehr. Glauben Sie mir, das ist in der Praxis so, ich habe damit schon sehr viel Erfahrung gesammelt ...

1.2.2 Was unterscheidet die OOP von der strukturierten Programmierung?

Bei der strukturierten Programmierung versucht der Programmierer Algorithmen, die an mehr als einer Stelle im Programm benötigt werden, in Prozeduren oder Funktionen zu implementieren. Wird der Algorithmus irgendwo benötigt, reicht ein Aufruf dieser Prozedur oder Funktion aus. Für einfache Probleme ist diese Art der Programmierung auch heute noch sehr gut geeignet. Wenn Sie z. B. an mehreren Stellen in einer Anwendung oder in mehreren verschiedenen Anwendungen aus einem gegebenen Einkommen die Einkommensteuer berechnen müssen (was ein recht komplexer Algorithmus ist), werden Sie natürlich eine Funktion zur Berechnung der Einkommensteuer schreiben. Eine einfache Funktion (und keine Klasse) reicht für solche relativ allein stehenden Probleme vollkommen aus. Auch für viele allgemeine Algorithmen, wie z. B. das Runden einer Dezimalzahl auf eine bestimmte Anzahl Nachkommastellen oder das Ersetzen von Teilen einer Zeichenkette durch eine andere Zeichenkette, die nicht in einem speziellen Kontext stehen, sind Prozeduren oder Funktionen viel besser geeignet als eine Klasse.

In viele Fällen ist ein strukturiertes Vorgehen aber nicht gerade vorteilhaft: Beim strukturierten Programmieren müssen Sie den Job machen, beim objektorientierten Programmieren überlassen Sie die Arbeit den Objekten.

Anlehnend an einen Internet-Artikel (sorry lieber Autor dieses Artikels, ich habe die Internetadresse vergessen) verdeutlicht die folgende Analogie den Vorteil des objektorientierten Vorgehens: Stellen Sie sich vor, Sie sitzen mit mehreren Freunden beim Essen an einem großen Tisch. Sie wollen Ihr Essen salzen, der Salzstreuer steht aber bei Ihrer Freundin Trillian und Sie kommen nicht heran. Wenn Sie dieses Problem prozedural angehen, würde Ihr »Programm« zur Lösung des Problems etwa folgendermaßen aussehen:

```
ArmHeben('Trillian', '10cm')
ArmNachVorne('Trillian', '20cm')
ArmNachLinks('Trillian', '5cm')
HandÖffnen('Trillian')
Solange Handhöhe('Trillian') > 2
   ArmSenken('Trillian')
Wiederholen
HandSchließen('Trillian')
ArmHeben('Trillian', '10cm')
ArmNachVorne('Trillian', '40cm')
HandÖffnen('Trillian')
```

Abgesehen davon, dass Ihrer Freundin Sie entweder für absolut kreativ (das wäre gut) oder für absolut verrückt (das wäre nicht so gut) hält, ist dieses Vorgehen für Sie – den Programmierer – nicht gerade ideal. Eigentlich würden Sie ja auch lieber einfach sagen: »Trillian, gib mir bitte den Salzstreuer«. Im Pseudocode ausgedrückt würde das also etwa so aussehen:

```
Trillian.GibMirBitte("Salzstreuer")
```

Das ist ein wichtiger (wenn nicht sogar *der* wichtigste) Unterschied zwischen der strukturierten und der objektorientierten Programmierung: Sie überlassen die Arbeit dem Objekt, das Objekt entscheidet, was zu tun ist, um das Problem zu lösen. Für Sie als Anwender des Objekts ist die Arbeit wesentlich vereinfacht (und führt in der realen Welt außerdem dazu, dass diese netten, weiß gekleideten Leute, die Ihnen diese eigenartige Jacke anziehen wollen, erst gar nicht erscheinen).

1.2.3 Die Vorteile der objektorientierten Programmierung

Sie haben nun die wichtigsten Nachteile der strukturierten Programmierung kennen gelernt. Ich hätte die Nachteile nicht genannt, wenn die OOP damit nicht aufräumt. Und das ist dann auch der Fall. Ich will Ihnen auf den folgenden Seiten an einigen Beispielen die Vorteile der OOP erläutern.

Die OOP erleichtert die Programmierung

Ein wichtiger Aspekt der OOP ist, dass diese die Programmierung erleichtert. Objekte sind wesentlich einfacher anzuwenden als relativ zusammenhanglose Funktionen und Daten, weil der Programmierer beim Anwenden eines Objekts immer genau weiß, welche Daten (Eigenschaften) und welche Funktionen (Methoden) er verwenden kann. Er kann nicht aus Versehen die falschen Daten überschreiben oder die falschen Funktionen aufrufen. Das Programmieren einer Klasse, aus der später die Objekte erzeugt werden, macht natürlich in etwa genauso viel Arbeit wie das Programmieren eines äquivalenten strukturierten Moduls. Der wichtige Aspekt bei der OOP ist aber nicht das Programmieren, sondern das *Anwenden* des Objekts. Beim Anwenden von Objekten wird noch ein weiterer Vorteil sichtbar: Aus einer Klasse können ohne Probleme auch *mehrere* Objekte erzeugt werden, die jedes für sich ihre eigenen Daten speichern. Ich will Ihnen diesen Vorteil mit einem Beispiel erläutern: In unserem Bestell-Programm soll der Programmteil zum Drucken von Lieferscheinen programmiert werden. Lieferscheine sollen immer so gedruckt werden, dass neben dem eigentlichen Lieferschein auch ein Deckblatt mit einem Anschreiben an den Kunden gedruckt wird. Deckblatt und Lieferschein sollen auf verschiedenen Druckern ausgegeben werden, weil diese

auf unterschiedlichen Papiervorlagen gedruckt werden müssen. Außerdem soll der Ausdruck mit unterschiedlichen Randeinstellungen erfolgen.

In der strukturierten Programmierung würde der Programmierer dazu vielleicht Druckfunktionen erzeugen, die neben den eigentlichen Daten noch Argumente übergeben bekommen, die Aussagen über den zu verwendenden Drucker und die Randeinstellungen machen.

Bei der Anwendung dieser Funktionen, also beim Programmieren des Lieferschein-Drucks, muss der Programmierer für jede auszudruckende Zeile nun immer den Namen des Druckers und die Randeinstellungen übergeben:

```
procedure Print_Delivery_Note(Customer: TCustomer; Orders: TOrders);
 { Die Datentypen TCustomer und TOrders sind in einer
   anderen Unit deklariert. Hier werden die Kundendaten
   und die Bestelldaten übergeben. Für das Beispiel ist
   die Deklaration dieser Typen nicht wichtig }
begin
 { Anschreiben drucken }
 Print_Text(Kunde.Name, 'HP Deskjet', 30, 20, 50, 20);
 Print_Text(Kunde.Strasse, 'HP Deskjet', 30, 20, 50, 20);
 Print_Text(Kunde.Ort, 'HP Deskjet', 30, 20, 50, 20);
 Print_Text(' ', 'HP Deskjet', 30, 20, 50, 20);
 Print_Text('Sehr geehrte Damen und Herren, 'HP Deskjet', 30, 20, 50, 20);
 ...
 { Lieferschein drucken }
 Print_Text(Kunde.Name, 'OKI OL-810', 25, 20, 40, 20);
 Print_Text(Kunde.Strasse, 'OKI OL-810', 25, 20, 40, 20);
 Print_Text(Kunde.Ort, 'OKI OL-810', 25, 20, 40, 20);
 Print_Text(' ', 'OKI OL-810', 25, 20, 40, 20);
 Print_Text('Lieferschein', 'HP Deskjet',25, 20, 50, 20);
 ...
end;
```

Sie sehen an diesem Beispiel, dass es ziemlich nervig ist, immer wieder die Argumente übergeben zu müssen, die doch eigentlich für den jeweiligen Kontext (»Anschreiben drucken« oder »Lieferschein drucken«) immer dieselben sind. Daneben ist dieses Vorgehen sehr fehlerträchtig. Die letzte *Print_Text*-Anweisung verwendet z. B. den falschen Drucker. Dieser Fehler ist entstanden, weil diese Anweisung einfach über die Zwischenablage kopiert wurde, wobei aber vergessen wurde, den Drucker anzupassen.

Bei der OOP würde der Programmierer für diese Druckfunktionen eine Klasse entwerfen. Er würde sich vor dem Programmieren Gedanken machen, welche Eigenschaften und welche Methoden das Objekt braucht. Dabei würde für unser Beispiel herauskommen, dass die Angaben zum Drucker zu den Rändern idealerweise in Eigenschaften verwaltet werden. Die Methoden zum Drucken müssen dann nur noch mit den eigentlich wichtigen Daten versorgt werden:

```
procedure Print_Delivery_Note(Customer: TCustomer; Orders: TOrders);
 { Die Datentypen TCustomer und TOrders sind in der Unit
   Globals deklariert. Hier werden die Kundendaten und die
   Bestelldaten übergeben. Für das Beispiel ist die
   Deklaration dieser Typen nicht wichtig }
 Var prnLetter, prnDeliveryNote: TPrinter;
begin
 { Erzeugen und Initialisieren der Objekte }
 prnLetter := TPrinter.Create('HP Deskjet', 30, 20, 50, 20);
 prnDeliveryNote := TPrinter.Create('OKI OL-810', 25, 20, 40, 20);

 { Drucken des Anschreibens }
 prnLetter.Print_Text(Customer.Name);
 prnLetter.Print_Text(Customer.Street);
 prnLetter.Print_Text(Customer.City);
 prnLetter.Print_Text(' ');
 prnLetter.Print_Text('Sehr geehrte Damen und Herren');

 { ... }
```

```
 { Neue Seite erzeugen }
 prnLetter.NewPage;

 { Lieferschein drucken }
 prnDeliveryNote.Print_Text(Customer.Name);
 prnDeliveryNote.Print_Text(Customer.Street);
 prnDeliveryNote.Print_Text(Customer.City);
 prnDeliveryNote.Print_Text(' ');
 prnDeliveryNote.Print_Text('Lieferschein');
 { ... }
end;
```

Sieht diese Funktion nun nicht wesentlich einfacher und leichter zu programmieren aus? Das objektorientierte Programmieren macht zwar mindestens so viel Arbeit wie das strukturierte Programmieren. Wichtig bei der OOP ist aber das *Anwenden* der Objekte. Und das ist wesentlich einfacher.

Die OOP ist weniger fehleranfällig

Die OOP arbeitet nicht mehr mit zusammenhanglosen Daten und Funktionen, sondern fasst alle Daten und Funktionen, die in einem gemeinsamen Kontext stehen, zu einer Klasse zusammen. Der anwendende Programmierer kann nur noch die Daten verwenden und die Funktionen aufrufen, die für den Kontext des Objekts definiert wurden. Er kann nicht, aus Versehen, irgendwelche Funktionen aufrufen oder Daten verändern, die in einem anderen Kontext stehen. Damit ist eine große Fehlerquelle der strukturierten Programmierung beseitigt. Um hier nicht wieder ein komplexes Beispiel anzubringen, verwende ich für die Erläuterung dieser Aussage ein einfaches Beispiel. Angenommen, ein Modul enthält eine Funktion zum Hochzählen einer modulglobalen Variablen, eine Funktion zum Herunterzählen dieser Variablen und eine Funktion zur Ausgabe des aktuellen Werts. Ein Programm verwendet dieses Modul und implementiert eine Möglichkeit, zwei verschiedene Anzahlen von Personen (z. B. Mitarbeiter und Besucher) zu zählen. Über einen Schalter wird die Anzahl der Mitarbeiter, die sich im Gebäude befinden, hochgezählt. Über einen anderen Schalter wird deren Anzahl heruntergezählt. Zwei weitere Schalter dienen dem Zählen der Besucher. Die Anwendung ist quasi automatisch mit einem Fehler versehen: Da die Funktionen zum Zählen immer dieselbe globale Variable verwenden, ist es gar nicht möglich, zwei verschiedene Mengen von Personen zu verwalten.

Die objektorientierte Variante verwendet einfach zwei Objekte einer Zählklasse (die ähnlich aufgebaut ist, wie das Zählmodul), die ihre eigenen Daten verwalten. Damit tritt dieses Problem erst gar nicht auf.

Die OOP verbessert die Wiederverwendung von Programmcode

Die Wiederverwendung einmal programmierter Lösungen ist ein wichtiges Thema bei der Programmierung. Man will das Rad ja nicht immer wieder neu erfinden. Schon die strukturierte Programmierung ermöglicht die Wiederverwendung von Programmcode in Form von Funktionen oder Prozeduren, die idealerweise in thematisch organisierten Modulen geschrieben sind. Das Modul mit den Duckfunktionen oben ist ein Beispiel dafür. Solange es sich dabei um relativ einfache, voneinander unabhängige Funktionen handelt, wie z. B. um Funktionen zur Berechnung der Einkommensteuer oder zur Berechnung der Fakultät einer Zahl, ist gegen die Wiederverwendung von Programmcode in Form von Funktionen oder Prozeduren auch eigentlich gar nichts zu sagen. Werden die Funktionen aber komplexer und stehen in einem Zusammenhang oder verwenden gemeinsame, globale Daten, ist die Wiederverwendung für den verwendenden Programmierer nicht mehr ganz so einfach. Problematisch dabei ist schon, dass der anwendende Programmierer nicht genau weiß, welche globalen Funktionen und Daten eigentlich zusammengehören. Zur Wiederverwendung gehört aber auch, dass der verwendende Programmierer die Möglichkeit hat, den »geerbten« Programmcode zu erweitern oder zu verändern, ohne den ursprünglichen Programmcode zu verändern (da dieser ja noch in anderen Programmen verwendet wird). Die *Print_Text*-Funktion des Druckfunktionen-Moduls könnte zum Beispiel so umgeschrieben werden, dass diese zusätzliche Informationen über die Schriftart

des Ausdrucks verwendet, die in neuen globalen Variablen gesetzt werden. Für den Programmierer stellt sich nun die Frage, wo die neue Funktion geschrieben werden soll. Schreibt er die Funktion in dem alten Druckfunktionen-Modul, wird dieses irgendwann einmal sehr unübersichtlich, nämlich dann, wenn immer mehr neue Funktionen hinzukommen. Schreibt er die Funktion in einem separaten, neuen Modul, so ist dieses zwar übersichtlich, erfordert aber das Einbinden des alten Druckfunktionen-Moduls, da dieses ja noch die anderen Funktionen und globale Variablen enthält, die ebenfalls benötigt werden. Hinzu kommt, dass die Namen globaler Funktionen dummerweise in einem Programm immer eindeutig sein müssen, die neue Funktion muss also z. B. *Print_Text2* heißen, die nächste Version heißt dann *Print_Text3* etc. Das macht ein erweitertes Programm auch nicht gerade übersichtlich. Natürlich könnte man einfach alle Funktionen des alten Moduls in ein neues Modul kopieren, die *Print_Text*-Funktion anpassen und für neue Programme immer nur dieses neue Modul verwenden. Aber das ist nicht wirklich Wiederverwendung.

Die objektorientierte Programmierung erleichtert die Wiederverwendung einmal geschriebener Programme über die so genannte *Vererbung*. Vererbung heißt, das eine neue Klasse erzeugt wird, die von einer existierenden Klasse abgeleitet wird und damit alle Eigenschaften und Methoden der Basisklasse erbt. Die neue Klasse muss natürlich anders heißen als die Basisklasse, die geerbten Eigenschaften und Methoden dieser Klasse haben aber automatisch denselben Namen wie die der Basisklasse. Die Vererbung ist in der OOP prinzipiell relativ einfach. In Object Pascal wird eine neue Klasse, die von einer existierenden Klasse abgeleitet wird, z. B. so erzeugt:

```
type TExtPrinter = class(TPrinter)

end;
```

Diese neue Klasse *TExtPrinter* ist von der Klasse *TPrinter* abgeleitet und erbt damit alle Methoden und Eigenschaften von *TPrinter*. Voraussetzung für die Vererbung ist, dass die Basisklasse im Programm bekannt ist. Zumindest muss das Modul (in Object Pascal die *Unit*), in dem die Basisklasse definiert ist, in das Modul der neuen Klasse eingebunden sein.

Erzeugt der Programmierer nun ein Objekt der Klasse *TExtPrinter*, so kann dieser das Objekt auch jetzt schon genau wie ein Objekt der Klasse *TPrinter* verwenden. Der Sinn der Vererbung ist aber, dass die Methoden der Basisklasse in der neuen Klasse mit einer neuen Programmierung *überschrieben* und sogar ohne Probleme neue Methoden und neue Eigenschaften in die neue Klasse eingefügt werden können. In unserem Beispiel kann z. B. die Methode *Print_Text* in der neuen Klasse mit einer Programmierung überschrieben werden, die neue Eigenschaften zur Definition der Schriftart verwendet.

Ein Programmierer, der die neue Druckfunktionalität verwenden will, muss jetzt in sein Programm nur noch die neue Klasse einbinden und wird bei der Verwendung der Klasse nicht dadurch irritiert, dass noch irgendwelche alten Programmierungen enthalten sind. Er sieht nur die für ihn wichtigen Eigenschaften und Methoden. Vielleicht werden Sie jetzt einwenden, dass der Programmierer die geerbten Eigenschaften und Methoden ja eigentlich gar nicht sieht und damit gar nicht weiß, was die Klasse sonst noch so alles kann. Prinzipiell haben Sie mit diesem Einwand Recht, ohne eine Hilfestellung führt die Vererbung zu relativ undurchsichtigen Programmen. In der Vergangenheit war das auch ein großer Kritikpunkt. Heutzutage stellen Ihnen alle modernen Entwicklungsumgebungen aber bei der Verwendung von Objekten eine Hilfe zur Verfügung, meistens in der Form, dass Ihnen alle Eigenschaften und Methoden des Objekts aufgelistet werden, wenn Sie bei der Verwendung des Objekts den Punkt schreiben:

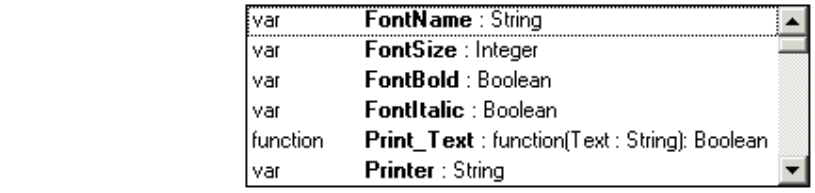

Abbildung 1.1: Die Objektelementliste von Delphi, über die der Programmierer sehr schnell Informationen über die Eigenschaften und Methoden des gerade verwendeten Objekts erhält. Nur die ersten Eigenschaften beziehungsweise Methoden des Objekts sind in der Abbildungen sichtbar. Alle weiteren Elemente kann der Programmierer erreichen, indem er den Scrollbalken rechts von der Liste betätigt. Andere Programmiersprachen, wie z. B. Visual Basic, bieten dem Programmierer ähnliche Hilfen an.

Sie sehen: Über die Vererbung wird die Wiederverwendung von Programmcode erheblich erleichtert. Die Vererbung bietet jedoch noch mehr Möglichkeiten. Ab Kapitel 3 erkläre ich die wichtigen Aspekte der Vererbung.

Objekte können die Realwelt besser abbilden

Objekte im Computer können Objekte der Realwelt nahezu 1:1 abbilden und damit das natürliche, objektorientierte Denken des Menschen beim Programmieren unterstützen. Unsere Gedanken und Gespräche drehen sich immer wieder um Objekte. Wenn wir z. B. über einen Drucker reden, meinen wir entweder einen konkreten Drucker (»der Epson aus der Buchhaltung macht mal wieder Ärger«) oder eine Klasse von gleichartig aufgebauten Druckern (»der Brother HL-720 hat im Test ganz gut abgeschnitten«). Wenn wir selbst eigentlich immer in Objekten und Klassen denken, warum sollen wir diese bei der Programmierung nicht einfach möglichst genau abbilden?

Die meisten Programme sind Abbildungen der Realwelt. Ein Grafikprogramm muss z. B. die Möglichkeiten, die ein Mensch mit realen Zeichenobjekten besitzt, möglichst genau abbilden, um dem Benutzer die Arbeit möglichst einfach zu machen. Ein Programm zu Abwicklung von Bestellungen muss den Ablauf von Bestellungen und alle Objekte, die damit in Beziehung stehen (das sind z. B. der Artikel und der Kunde), möglichst genau abbilden, damit der Benutzer die real vorgenommene Bestellung und den real existierenden Kunden im Programm nachvollziehen kann. Die besten Programme sind die, die die Realwelt möglichst genau abbilden.

Die strukturierte Programmierung hat aber so ihre Probleme mit der Abbildung der Realwelt. Ein Kunde, ein Artikel, eine Bestellung ist bei dieser nur eine zusammenhanglose Sammlung von Daten und Funktionen. Die OOP kann dagegen die Objekte der Realwelt auf ideale Weise nachbilden. Ein Kunde wird über ein Kundeobjekt repräsentiert, ein Artikel über ein Artikelobjekt, eine Bestellung über ein Bestellungsobjekt. Das Programm speichert natürlich nur die relevanten Daten der realen Objekte, eine echte 1:1-Abbildung ist in der Regel nicht möglich, wird aber meist auch nicht benötigt.

Daneben können auch die Beziehungen zwischen Objekten der realen Welt über die OOP wunderbar abgebildet werden. Als Beispiel verwende ich gerne einen Stammbaum. Ein Stammbaum stellt einzelne Personen mit wichtigen Informationen über diese Person und die Beziehungen zwischen diesen Personen dar. Jede Person besitzt einen Namen und ein Geburtsdatum und Beziehungen zum Vater, zur Mutter und zum Partner (und zu ihren Kindern, aber das wird aufgrund der Komplexität in diesem Beispiel nicht dargestellt). Über diese Beziehungen (die über Eigenschaften der Objekte implementiert werden) können alle Informationen über Objekte eines auch sehr komplexen Beziehungsgeflechts ohne große Probleme ausgelesen werden.

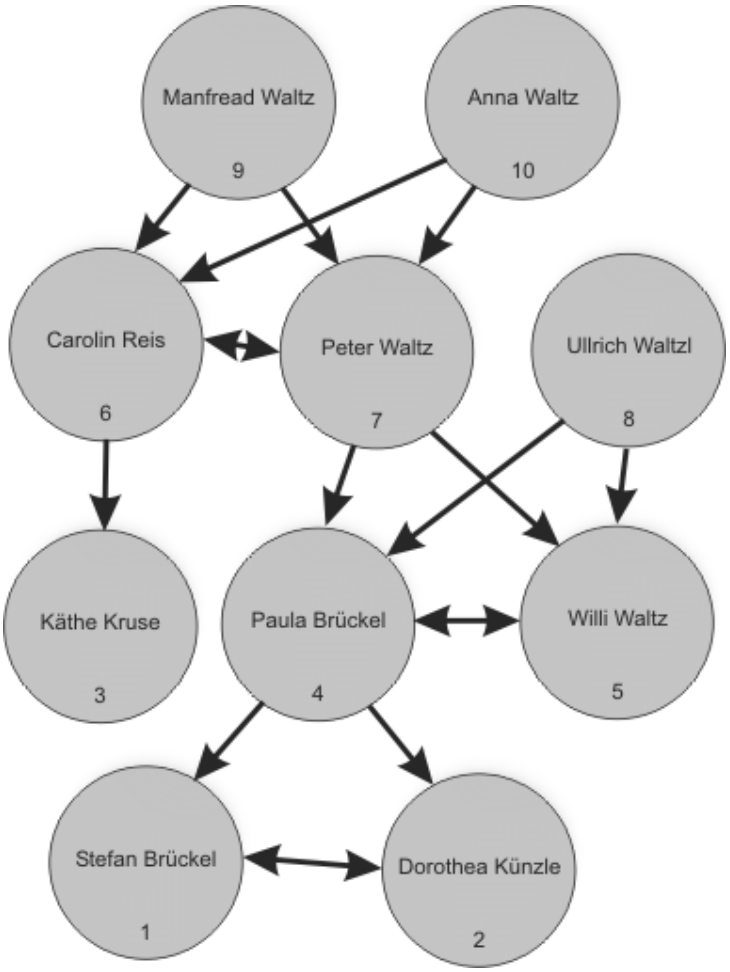

Abbildung 1.2: Ein Beispiel-Stammbaum

In einem objektorientiert im Computer angelegten Stammbaum könnten Sie z. B. den Namen der Großmutter einer Person folgendermaßen ermitteln:

```
writeln 'Anna Waltz hat die folgenden Grossmütter: ' +
   AnnaWaltz.Mutter.Mutter.Name + ', ' +
   AnnaWaltz.Vater.Mutter.Name
```

Im Prinzip entspricht dies genau der menschlichen Vorgehensweise. Testen Sie dies an sich selbst, indem Sie in einem Stammbaum die Großmutter einer Person ermitteln.

Der »Trick« dabei ist, dass Eigenschaften von Objekten selbst auch wieder Objekte sein können. Einen Personen-Objekt eines Stammbaums besitzt, neben den einfachen Eigenschaften *Vorname*, *Nachname* und *Geburtsdatum* auch solche Eigenschaften wie *Vater*, *Mutter* und *Partner*, die im Prinzip nichts anderes als Referenzen auf andere Personen-Objekte sind.

Soll ein Stammbaum im Computer abgebildet werden, werden genau solche Referenzen auf andere Personenobjekte in einem Person-Objekt in Form von speziellen Eigenschaften eingebaut. Eine Eigenschaft Vater referenziert den Vater der Person, eine Eigenschaft Mutter die Mutter und eine Eigenschaft Partner den Partner der Person. Die Auflösung solcher Referenzen bei der Anwendung des Objekts ist dann sehr einfach, wie das Beispiel oben zeigt.

Wenn Sie sich jetzt vielleicht fragen, wo denn da der Vorteil gegenüber der strukturierten Programmierung liegt, versuchen Sie einmal einen Stammbaum nur mit Hilfe von Variablen, Strukturen und Arrays aufzubauen. Die Programmierung eines Stammbaums mit strukturierten Mitteln wird sehr komplex, unübersichtlich und ist nur sehr schwer erweiterbar. Für das Beispiel am schlimmsten ist aber, dass die Beziehungen zwischen einzelnen Personen nicht

mehr, wie beim OOP-Beispiel, automatisch aufgelöst werden können, sondern von „Hand“ (programmatisch) aufgelöst werden müssen.

> Wenn Ihnen die damit entstehenden Probleme (schwierige Speicherung, kompliziertes Auflösen der Beziehungen) unklar sind, lesen Sie den Artikel »Vergleich der strukturierten Programmierung mit der OOP« (»Vergleich-SP-OOP«) lesen. Dieser Artikel greift das Stammbaum-Beispiel auf und zeigt eine strukturierte und eine objektorientierte Lösung.

Die OOP erlaubt die Erzeugung von neuen Datentypen

Die Programmierung erfordert häufig die Arbeit mit neuen, eigenen Datentypen. Stellen Sie sich vor, Sie müssen häufig mit XY-Koordinaten oder mit komplexen Zahlen[7] arbeiten. Bei der strukturierten Programmierung haben Sie lediglich die Möglichkeit, für diese neuen Datentypen eine Struktur zu erzeugen. Die Struktur setzt sich aus einzelnen Basisdatentypen zusammen und ermöglicht so das gemeinsame Speichern von mehreren unterschiedlichen Datentypen. Ein großes Problem dabei sind die Operationen, die Sie auf diesen neuen Datentypen ausführen wollen bzw. müssen. Das Problem beginnt schon dann, wenn Sie einfach nur die Werte einer Struktur-Variablen in eine andere kopieren wollen oder wenn Sie einer Struktur-Variablen Werte aufaddieren wollen. Die meisten Programmiersprachen lassen solche Operationen mit Struktur nicht zu. Werden die Operationen dann komplizierter, wie das z. B. bei komplexen Zahlen der Fall ist, werden auch die Probleme größer.

Sie müssen also separate Funktionen oder Prozeduren schreiben, die diese Arbeit übernehmen. Das Problem dabei ist wieder, dass diese Funktionen nicht im Kontext des neuen Datentyp stehen und damit schwierig anzuwenden sind. Wenn Sie die neuen Datentypen stattdessen in Klassen definieren, können Sie die Operationen auf diesen Datentypen gleich mit in die Klasse integrieren. In allen Programmiersprachen können Sie dazu zumindest Methoden einsetzen (z. B. eine Methode zum Addieren von einer XY-Koordinate auf eine andere). Einige Programmiersprachen, wie z. B. C++, erlauben aber auch das so genannte Überschreiben der Standard-Operatoren der Sprache, so dass es z. B. möglich ist, ein Objekt mit einem anderen Objekt zu addieren.

> Der Artikel »Vergleich der strukturierten Programmierung mit der OOP am Beispiel von neuen Datentypen« demonstriert das Problem der strukturierten Programmierung mit neuen Datentypen und die Lösung durch die OOP.

[7] Eine komplexe Zahl besteht aus zwei Zahlteilen, einem so genannten Realteil und einem Imaginärteil. Komplexe Zahlen wurden entwickelt, weil sich einige Probleme in der Mathematik nicht mit denen uns bekannten reellen Zahlen erklären lassen. Ein Beispielproblem dazu ist die Frage, welche Zahl ins Quadrat genommen -1 ergibt. Keine der uns bekannten Zahlen erfüllt diese Bedingung (wahrscheinlich weil die Anzahl der Dezimalstellen viel zu groß ist). Mit Hilfe der komplexen Zahlen können Sie, auf sehr komplizierte Art und Weise solche Probleme lösen. Eine gute Beschreibung der komplexen Zahlen finden Sie im Internet bei WWW.KOOPIWORLD.DE/PUB/KOMPLEX.HTM.

2 Die Basis der OOP: Die Klasse

Die Klasse ist die Basis der OOP. Eine Klasse beschreibt, welche Methoden und Eigenschaften ein Objekt besitzt, das mit Hilfe dieser Klasse erzeugt wird. Eine Klasse ist damit quasi ein Bauplan, mit dem der Programmierer später in seinem Programm Objekte erzeugt. Es gibt keine Objekte, die keiner Klasse angehören. Jedes Objekt gehört genau einer Klasse an[8]. Aus einer Klasse können aber beliebig viele Objekte erzeugt werden. Alle diese Objekte besitzen dann exakt dieselben Eigenschaften und dieselben Methoden. Betrachten Sie uns Menschen: Wir gehören alle derselben Klasse *Mensch* an und haben alle dieselben Eigenschaften (wie z. B. die *Größe*, die *Haarlänge*, die *Haarfarbe*, die *Augenfarbe* etc.) und dieselben Methoden (wie z. B. *Gehen*, *Sprechen*, *Denken* etc.). Lediglich die Ausprägung der Eigenschaften ist bei den verschiedenen Objekten einer Klasse oft unterschiedlich. Unterschiedliche Menschen besitzen unterschiedliche Augenfarben, Haarfarben usw.

Die folgenden Seiten zeigen Ihnen, wie Sie in den verschiedenen im Artikel behandelten Sprachen eine einfache Klasse mit Eigenschaften und Methoden erzeugen.

2.1 Basiswissen

Eigenschaften und Methoden

In allen im Artikel behandelten Sprachen werden Eigenschaften einer Klasse grundsätzlich genauso deklariert wie Variablen, die Deklaration von Methoden entspricht der Deklaration von Prozeduren oder Funktionen. Innerhalb einer Methode können Sie auf alle Eigenschaften und auf alle privaten Daten der Klasse (wozu private Daten verwendet werden, erkläre ich in Abschnitt 2.4 bei der Kapselung) zugreifen. Eine einfache Klasse zur Speicherung von Personendaten könnte in Object Pascal so aussehen:

```
{ Deklaration der Klasse }
type
  TPerson = class
    { Deklaration der Eigenschaften }
    Vorname: String;
    Nachname: String;

    { Deklaration der Methoden }
    function GetFullName: String;
  end;

{ Implementierung der Methode }
function TPerson.GetFullName: String;
begin
   result := Vorname + ' ' + Nachname;
end;
```

Wie Klassen für die einzelnen Sprachen erzeugt werden, erfahren Sie in Abschnitt 2.2.

Das Arbeiten mit Objekten

Wenn Sie in einer Programmiersprache mit Objekten arbeiten wollen, müssen Sie diese zunächst erzeugen. Dieses Erzeugen wird im Allgemeinen als »instanzieren« bezeichnet: Ein Objekt ist eine Instanz einer Klasse.

[8] Diese Aussage ist nicht ganz korrekt: Eine Klasse kann über die Vererbung auch von anderen Klassen abgeleitet werden. Diese neuen Klassen erben damit alle Eigenschaften und Methoden der Basisklasse. Ein Objekt, das aus einer abgeleiteten Klasse erzeugt wird, gehört damit zwar immer noch genau einer Klasse an, besitzt aber die Elemente mehrerer Klassen.

Aus einer Klasse können Sie beliebig viele Objekte erzeugen. Sie benötigen lediglich eine *Referenz* auf die einzelnen Objekte. Eine Referenz ist im Prinzip ein Zeiger auf den Speicherbereich, in dem das Objekt beim Instanzieren angelegt wird. Wenn Sie ein Objekt erzeugen, legt das Programm dieses Objekt irgendwo im Arbeitsspeicher an und weist die Adresse dieses Speicherbereichs der Referenz zu. Über diese Referenz können Sie das Objekt dann ansprechen. Jede Programmiersprache besitzt ihre eigene Syntax zur Erzeugung von Objekten. Ich beschreibe diese in Abschnitt 2.2. An dieser Stelle soll erst einmal ein Object Pascal-Beispiel ausreichen. Sie müssen normalerweise zuerst eine Variable deklarieren, deren Typ der Klassenbezeichner ist. Angenommen, das Programm enthält bereits eine Klasse *TPerson* zur Speicherung von Personendaten. Dann sieht die Deklaration in Object Pascal etwa so aus:

```
{ Deklaration der Referenzvariable }
var p: TPerson;
```

Im Programm müssen Sie in den meisten Programmiersprachen das Objekt dynamisch[9] erzeugen. In Object Pascal rufen Sie dazu den Konstruktor `Create` auf (was Konstruktoren sind, erfahren Sie in Abschnitt 2.5):

```
{ Instanzierung des Objekts }
p := TPerson.Create;
```

Der Konstruktor erzeugt das Objekt im Arbeitsspeicher und gibt die Adresse dieses Objekts zurück. Über die Referenzvariable können Sie nun mit dem Objekt arbeiten:

```
p.Vorname = 'Ford';
p.Nachname = 'Prefect';
```

Abbildung 2.1 demonstriert, wie ein Objekt im Arbeitsspeicher gespeichert wird.

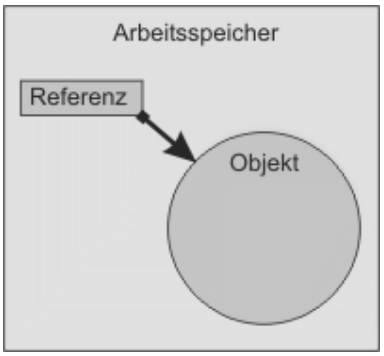

Abbildung 2.1: Ein Objekt ist im Arbeitsspeicher gespeichert. Eine Referenz zeigt auf dieses Objekt.

> Sie müssen Referenzen von normalen Variablen unterscheiden. Eine normale Variable speichert ihren Wert direkt. Eine Referenz zeigt nur auf den Speicherbereich, in dem der Wert bzw. das Objekt gespeichert ist.

Die Frage, warum die OOP mit Referenzen arbeitet, lässt sich recht einfach beantworten: Jedes Objekt ist individuell, speichert seine eigenen Daten und besitzt seine eigene Identität. Wenn nun der Bedarf besteht, dass *mehrere* Programmteile (Funktionen, Prozeduren, andere Objekte etc.) mit *demselben* Objekt arbeiten müssen, lässt sich dieses Problem nur mit Referenzen lösen. Wenn Sie eine Variable an eine Prozedur oder Funktion übergeben, wird (normalerweise, wenn Sie die Variable nicht schon »By Reference« übergeben) der *Wert* der Variable in das Argument

[9] Dynamisches Erzeugen bedeutet, dass das Objekt erst in der Laufzeit des Programms dann angelegt wird, wenn das Programm an der entsprechenden Anweisungen angelangt ist und nicht bereits, wenn das Programm gestartet wird.

kopiert. Bei Objekten macht das aber keinen Sinn. Verschiedene Programmteile sollen schließlich mit *demselben* Objekt arbeiten und nicht mit Kopien eines Objekts.

Auf Objekte können also gleich mehrere Referenzen zeigen. Enthält ein Programm z. B. eine Funktion, der ein Objekt übergeben wird, so zeigen nach dem Aufruf der Funktion schon zwei Referenzen auf das Objekt:

```
{ Eine Funktion, der ein TPerson-Objekt übergeben wird }
function PrintPersonData(var person: TPerson): boolean;
begin
  { Hier zeigen zwei Referenzen auf das Objekt,
    die in dieser Funktion und die des Programms }
  ...
end;

{ Eine Referenz auf ein TPerson-Objekt }
var p: TPerson;

{ Das eigentliche Programm }
begin
  { Das Objekt wird erzeugt }
  p := TPerson.Create;

  { Die Funktion wird aufgerufen, wobei die Referenz
    auf das Objekt an die Funktion übergeben wird }
  PrintPersonData(p);
end;
```

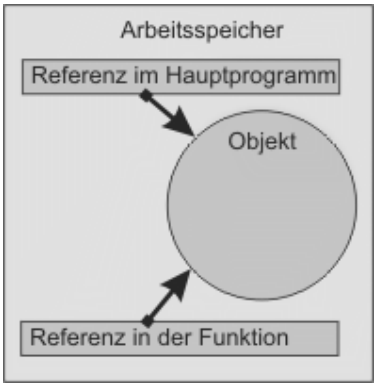

Abbildung 2.2: Bei dem Aufruf einer Funktion, der eine Referenz auf ein Objekt übergeben wird, zeigen schon zwei Referenzen auf dasselbe Objekt.

> Aus der Tastsache, dass Objekte immer über Referenzen bearbeitet werden, ergibt sich, dass Argumente von Funktionen, Prozeduren oder Methoden, denen Objektreferenzen übergeben werden sollen, prinzipiell »By Reference« deklariert werden müssen. Einige Compiler, wie z. B. der von Delphi und Kylix, erkennen aber auch selbstständig, anhand des Datentyps des Arguments, dass es sich um ein Objekt handelt, und übergeben das Argument automatisch »By Reference«.

Objektreferenzen als Eigenschaften von anderen Objekten

Objektreferenzen können auch selbst wieder Eigenschaften von anderen Objekten sein. Die Eigenschaft wird dazu einfach mit dem Klassentyp des anderen Objekts deklariert. Das Objekt, das über die Eigenschaft referenziert wird, wird meist bei der Erstellung des »Vaterobjekts« automatisch erzeugt (was der Programmierer dann im so genannten *Konstruktor* erledigt) oder von außen, nach der Instanzierung des Vaterobjekts im Programm gesetzt. Objekteigenschaften machen immer dann Sinn, wenn ein Unterobjekt in mehreren verschiedenen Klassen verwendet werden kann. Ein gutes Beispiel für ein solches Vorgehen ist der Mensch: Ein Mensch besitzt u. a. zwei *Augenobjekte* als Objekteigenschaft.

Manchmal ist es aber auch einfach notwendig, eine Objekteigenschaft zu implementieren. Das Stammbaum-Beispiel in Kapitel 1 demonstriert dies anhand von Personenobjekten, die in den Eigenschaften *Vater*, *Mutter* und *Partner* auf andere Personenobjekte zeigen. Daran erkennen Sie auch, dass eine Objekteigenschaft von demselben Klassentyp sein kann wie die Klasse, in die diese eingefügt wird.

Wie wird ein Objekt zerstört?

Irgendwann einmal muss ein Objekt wieder aus dem Speicher entfernt werden. Wann und wie das geschieht, hängt von der Programmiersprache ab.

In **C++** können Sie Objekte statisch erzeugen lassen oder im Programm dynamisch erzeugen (wie ich dies im nächsten Abschnitt erkläre). Statische Objekte werden automatisch zerstört, wenn die Referenz, die auf das Objekt zeigt, ungültig wird. Dynamisch erzeugte Objekte müssen Sie über die `delete`-Anweisung selbst zerstören. Wenn Sie z. B. ein Objekt in einer Funktion dynamisch erzeugen und vergessen, das Objekt zu zerstören, bleibt dieses im Arbeitsspeicher, solange Ihr Programm läuft, und Sie haben keinen Zugriff mehr auf das Objekt. Damit erzeugen Sie dann diese schönen Speicherlöcher, die bei fehlerhaft programmierten Anwendungen den freien Speicher immer kleiner werden lassen, je länger die Anwendung läuft.

In **C#, Visual Basic .NET** und **Java** werden Objekte immer dynamisch angelegt. Um das Zerstören des Objekts kümmert sich in diesen Sprachen ein so genannter Garbage Collector, der in regelmäßigen Abständen immer dann, wenn das Programm gerade nicht beschäftigt ist, durch das System geht und Objekte, auf die keine Referenz mehr zeigt, aus dem Speicher entfernt. In diesen Sprachen müssen Sie sich nicht um das Zerstören des Objekts kümmern.

In **Object Pascal (Delphi/Kylix)** werden Objekte ebenfalls ausschließlich dynamisch erzeugt. Sie müssen ein erzeugtes Objekt auf jeden Fall über die `Free`-Methode des Objekts zerstören, wenn Sie das Objekt nicht mehr benötigen. Ansonsten drohen, wie bei den dynamischen C++-Objekten, Speicherlöcher.

In **Visual Basic 6** erzeugen Sie Objekte ebenfalls dynamisch. Visual Basic 6-Objekte entsprechen dem COM-Modell von Microsoft. Diese Objekte zählen die Referenzen, die auf sie zeigen, und zerstören sich – auch wenn dies vielleicht etwas komisch klingt – selbst, wenn keine Referenz mehr auf das Objekt zeigt. Sie müssen sich in Visual Basic 6 also ebenfalls nicht um das Zerstören eines Objekts kümmern.

2.2 Klassen in den verschiedenen Sprachen

C++

In C++ können Sie eine oder mehrere Klassen in einer *cpp*-Datei unterbringen. Sie müssen dabei unterscheiden, ob die Klasse in der Datei deklariert ist, in der sie auch verwendet wird, oder in einer anderen Datei. Der erste Fall ist relativ einfach. Eine Klasse, die in derselben Datei verwendet werden soll, in der sie geschrieben wurde, kann die Methoden-Implementierung direkt in der Klassendeklaration erhalten:

```
#include <iostream.h>

class Circle
{
 public:
   /* Eigenschaften */
   int Radius;

   /* Methoden */
   double getCircumference()
   {
      return 3.1415927 * 2 * Radius;
   }

   double getSurface()
   {
      return 3.1415927 * Radius * Radius;
   };
};

...
```

Wenn Sie ein Objekt dieser Klasse verwenden wollen, müssen Sie dieses Objekt nicht, wie in anderen Sprachen, explizit erzeugen. Eine einfache Deklaration reicht in C++ aus. C++ erzeugt das Objekt dann automatisch:

```
...

int main()
{
   /* Variable der Klasse deklarieren, womit gleich
      auch ein Objekt erzeugt wird */
   Circle c;

   /* Radius setzen */
   c.Radius = 20;

   /* Fläche berechnen und ausgeben */
   cout << "Die Flaeche eines Kreises mit "
        << c.Radius << " mm ist " << c.getSurface()
        << " mm^2";
}
```

Das Problem bei so erzeugten »statischen« Objekten ist, dass Sie schon bei der Entwicklung wissen müssen, wie viele Objekte gespeichert werden sollen, weil Sie ja die entsprechende Anzahl an Variablen deklarieren müssen. Für manche Probleme reicht das aber nicht aus. Wollen Sie z. B. ein Programm erzeugen, das einen Stammbaum darstellen soll, können Sie bei der Entwicklung noch gar nicht wissen, wie viele Personen-Objekte der Anwender erzeugen und speichern will.

Deshalb ist es in C++ auch möglich, Objekte dynamisch zu erzeugen. Dazu deklarieren Sie die Variable als Zeiger:

```
Circle *c;
```

und erzeugen das Objekt dann mit dem `new`-Operator:

```
c = new Circle;
```

So kann Ihr Programm beliebig viele Objekte erzeugen. Das Problem dabei ist lediglich, wie Sie die Objekte speichern, weil Sie zur Entwicklungszeit noch nicht wissen können, wie viele Referenzen Sie benötigen. Dafür existieren aber einige einfach handhabbare Lösungen, wie z. B. dynamische Arrays oder Collections (die im Rahmen dieses Artikels aber nicht beschrieben werden können).

> Objekte, die in C++ dynamisch erzeugt wurden, müssen über den Zeiger-Operator angesprochen werden: `c->Radius = 20;`

> Objekte, die Sie in C++ dynamisch erzeugen, müssen Sie über die `delete`-Anweisung wieder aus dem Speicher entfernen. Vergessen Sie dies, bleibt das Objekt im Speicher so lange Ihre Anwendung läuft, und verbraucht unnötig Ressourcen.

In der Praxis werden Klassen oft in separaten Dateien deklariert, um diese einfach in anderen Programmen wiederverwenden zu können. Um eine oder mehrere Klassen aus separaten Dateien in ein C++-Programm einbinden zu können, können Sie in C++ zwei Wege gehen. Der einfachere ist der, die komplette Klasse einfach in einer *cpp*-Datei zu deklarieren und diese *cpp*-Datei über die #include-Anweisung einzubinden:

```
#include <iostream.h>
#include "Circle.cpp"

int main()
{
   Circle c;
   c.Radius = 20;
   cout << "Die Flaeche eines Kreises mit "
        << c.Radius << " mm ist " << c.getSurface()
        << " mm^2";
}
```

Beim Kompilieren muss diese separate Datei dann mit angegeben werden:

```
bcc32 Circle_Demo_2.cpp Circle.cpp
```

Wenn Sie eine Entwicklungsumgebung verwenden, übernimmt diese das Kompilieren der eingebundenen Klasse natürlich automatisch. Voraussetzung für das Einbinden ist, dass die Klasse entweder in demselben Ordner gespeichert ist wie das eigentliche Programm oder dass die Klasse im Library-Ordner gefunden wird (den Sie dem Compiler über die Option –L mitteilen).

Eine etwas kompliziertere Möglichkeit, Klassen in separaten Dateien zu verwalten, ist das Trennen der Deklaration von der Implementierung. Dazu erzeugen Sie eine Datei mit der Endung .H. Diese *Header-Datei* erhält denselben Namen wie die eigentliche *cpp*-Datei und wird nur mit der Deklaration der Klasse versehen:

```
/* Datei Circle.h */

class Circle
{
  public:
    int Radius;
    double getCircumference();
    double getSurface();
};
```

Dazu erzeugen Sie eine gleichnamige Datei mit der Endung *.cpp*. Diese Datei enthält dann die Implementierung der Methoden. Um die Methoden der u. U. verschiedenen Klassen zu unterscheiden, wird jedem Methodenname der Klassenname, gefolgt von zwei Doppelpunkten,

vorangestellt. Die Header-Datei muss in die *cpp*-Datei eingebunden werden, damit der Compiler die zugehörige Klasse erkennen kann:

```
/* Datei Circle.cpp */

#include "Circle.h"

/* Implementierung der Methoden */

double Circle::getCircumference()
{
   return 3.1415927 * 2 * Radius;
};

double Circle::getSurface()
{
   return 3.1415927 * Radius * Radius;
};
```

Im eigentlichen Programm binden Sie dann nur die Header-Datei ein:

```
#include <iostream.h>
#include "Circle.h"

int main()
{
   Circle c;
   c.Radius = 20;
   cout << "Die Flaeche eines Kreises mit "
        << c.Radius << " mm ist " << c.getSurface()
        << " mm^2";
}
```

Viel Sinn macht dieses Trennen der Deklaration von der Implementierung in meinen Augen nicht. Einer eventuell besseren Übersichtlichkeit Ihres Programms steht der deutlich größere Arbeits- und Pflegeaufwand gegenüber.

C#

C# ermöglicht Ihnen, eine oder mehrere Klassen innerhalb einer CS-Datei zu speichern, die auch öffentlich sein können. Idealerweise trennen Sie, wie in den anderen Sprachen auch, Ihre eigentliche Programmdatei von den Klassendateien. Um die Klassen einer Datei in ein Programm einbinden zu können, müssen Sie die Klassen in ein so genanntes *Namespace* einbinden. Ein Namespace hat die Bedeutung, mehrere Klassen (die auch in unterschiedlichen Dateien gespeichert sein können) zu einer logischen Einheit zusammenzufassen. Diesen Namespace binden Sie dann in Ihr Programm ein. Eine einfache Klasse zur Kreisberechnung sieht in C# so aus:

```
/* Deklaration des Namespace */
namespace Geometric_Classes
{
   using System;

   /* Deklaration der Klasse */
   public class Circle
   {
      /* Eigenschaften */
      public int Radius;

      /* Methoden */
      public double getCircumference()
      {
         return 3.1415927 * 2 * Radius;
      }

      public double getSurface()
      {
         return 3.1415927 * Radius * Radius;
      }
   }
}
```

Wenn Sie die Klasse dann im Programm verwenden wollen, sollten Sie das Namespace (hier: *Geometric_Classes*) in das Programm einbinden, was über die `using`-Anweisung geschieht. Wenn Sie das Namespace einbinden, müssen Sie in die darin enthaltenen Klassen nicht immer über den vollen Namen (inklusive Namespace) referenzieren. Das Objekt erzeugen Sie dann mit dem `new`-Operator:

```
/* Einbinden des System-Namespace */
using System;

/* Einbinden des Namspace, in dem die
   Klasse Circle enthalten ist */
using Geometric_Classes;

/* Startklasse der Anwendung */
public class Start
{
   /* Startmethode der Anwendung */
   public static int Main(string[] args)
   {
      /* Variable für das Objekt deklarieren */
      Circle c;

      /* Objekt erzeugen */
      c = new Circle();

      /* und verwenden */
      c.Radius = 20;
      Console.WriteLine("Die Flaeche eines Kreises mit " +
         c.Radius + " mm ist " + c.getSurface() + " mm^2");

     return 0;
   }
}
```

Java

In Java erzeugen Sie Klasse in Textdateien mit der Endung *.java*. In einer solchen Datei können Sie nur eine öffentliche Klasse deklarieren. Diese Klasse muss genau denselben Namen tragen wie die Datei (inklusive Groß- und Kleinschreibung!). In dieser Datei können Sie allerdings auch weitere, private Klassen (die dann nur innerhalb der Datei gelten) deklarieren. Eine einfache Klasse für Kreisberechnungen sieht dann z. B. so aus:

```
public class Circle
{
   int Radius;

   double getCircumference()
   {
      return 3.1415927 * 2 * Radius;
   }

   double getSurface()
   {
      return 3.1415927 * Radius * Radius;
   }
}
```

Wie Sie sehen, können Sie die Methoden in Java direkt in der Klassendeklaration implementieren.

Wenn Sie die Klasse in einer anderen Java-Datei verwenden wollen, müssen Sie diese zuvor kompilieren. Die erzeugte *.class*-Datei muss dann im Ordner des Programms oder in einem Ordner gespeichert sein, der im Java-Classpath integriert ist. Wie bei anderen Sprachen auch, benötigen Sie eine Variable für das Objekt dieser Klasse. In Java erzeugen Sie Objekte dann mit dem `new`-Operator:

```
public class Circle_Demo
{
   public static void main(String[] args)
   {
      /* Variable für ein Objekt der Circle-Klasse */
      Circle c;

      /* Erzeugen eines Objekts der Circle-Klasse */
      c = new Circle();

      /* Objekt verwenden */
      c. Radius = 20;
      System.out.println("Die Flaeche eines Kreises mit " +
         c.Radius + " mm ist " + c.getSurface() + " mm^2");
  }
}
```

Object Pascal (Delphi/Kylix)

In Object Pascal legen Sie Klassen in einer Unit an. Die Klasse muss im Interface-Teil der Unit deklariert werden, wenn die Klasse öffentlich zugänglich sein soll. Im Implementation-Abschnitt werden dann die Methoden implementiert, wobei dem Methodennamen der Name der Klasse vorangestellt werden muss. Die einfache Version einer Klasse sieht dann, am Beispiel einer *Circle*-Klasse, so aus:

```
unit Circle_Class;

interface

type
  TCircle = class
    { Deklaration der Eigenschaften }
    Radius: integer;
    { Deklaration der Methoden }
    function getCircumference: double;
    function getSurface: double;
  end;

implementation

{ Implementierung der Methoden }

function TCircle.getCircumference: double;
begin
   getCircumference := 3.1415927 * 2 * Radius;
end;

function TCircle.getSurface: double;
begin
   getSurface := 3.1415927 * Radius * Radius;
end;

end.
```

Die erzwungene Implementierung der Methoden im Implementation-Abschnitt der Unit ist bei Object Pascal etwas mühevoll, ist aber damit begründet, dass diese Sprache keine so genannten Modifizierer für Klassen und Methoden kennt, wie es z. B. mit den `Private`/`Public`-Modifizierern bei Java der Fall ist.

Klassen werden immer in einem `type`-Block deklariert, der mit dem `end;` der letzten enthaltenen Klassendeklaration abgeschlossen wird. In einer Unit können mehrere Klassen deklariert werden. Diese können Sie in einem `type`-Block erzeugen:

```
type
   TClass1 = class
     { ... }
   end;

   TClass2 = class
     { ... }
   end;
```

Alternativ können Sie auch pro Klasse einen separaten `Type`-Block erzeugen:

```
type
   TClass1 = class
     { ... }
   end;

type
   TClass2 = class
     { ... }
   end;
```

Sie sollten der Version mit einem Type-Block den Vorzug geben. Mit separaten Blöcken funktionieren einige spezielle Techniken wie das so genannte Vorwärts-Deklarieren von Klassen nicht.

Wenn Sie Objekte dieser Klasse erzeugen wollen, müssen Sie den Konstruktor `Create` der Klasse aufrufen um eine Instanz zu erzeugen. Dieser Konstruktor ist eine spezielle Klassenmethode, die auch ohne eine Instanz der Klasse aufgerufen werden kann. Was ein Konstruktor genau ist, erfahren Sie in Abschnitt 2.5. Hier zeige ich Ihnen zuerst einmal, wie Sie ein Objekt erzeugen:

```
program Circle;

{ die uses-Anweisung bindet die Unit mit der Circle-
  Klasse ein }
uses SysUtils, Circle_Class;

{$APPTYPE CONSOLE}

{ Eine Variable des Klassentyps deklarieren }
var c: TCircle;

begin
 writeln('Einfache Klasse zur Kreisberechnung');

 { Erzeugen des Objekts der Klasse TCircle }
 c := TCircle.Create;

 { Radius setzen }
 c.Radius := 20;

 { Fläche berechnen und ausgeben }
 writeln('Die Flaeche eines Kreises mit ' +
  IntToStr(c.Radius) + ' mm ist ' +
  FloatToStr(c.GetSurface) + ' mm^2.');

 readln;

 { Objekt freigeben }
 c.Free;
end.
```

In Object Pascal werden Objekte immer dynamisch erzeugt. Da Object Pascal keinen Garbage Collector kennt (den Java, C# und Visual Basic .NET besitzen), der nicht mehr benötigte Objekte selbstständig freigibt, müssen Sie in Object Pascal die erzeugten Objekte immer über die `Free`-Methode des Objekts freigeben, um keine Speicherlöcher zu erzeugen.

Visual Basic 6

In Visual Basic 6 erzeugen Sie eine Klasse in einem separaten Klassenmodul, das Sie über das Menü PROJEKT | KLASSENMODUL HINZUFÜGEN in Ihr Projekt einfügen. Der Name der Klasse ist in VB 6 der Name des Klassenmoduls. Ein Klassenmodul beschreibt immer genau eine Klasse. Die Klasse zur Kreisberechnung sieht in VB 6 dann so aus:

```
Option Explicit

' Eigenschaften
Public Radius As Long

' Methoden
Public Function GetCircumference() As Double
  GetCircumference = 3.1415927 * 2 * Radius
End Function

Public Function GetSurface() As Double
  GetSurface = 3.1415927 * Radius * Radius
End Function
```

> In Visual Basic 6 ist `Circle` ein reserviertes Schlüsselwort, weswegen ich die Klasse `CCircle` genannt habe.

Eine Klasse wird in Visual Basic wie ein normales Modul in das Projekt eingebunden. Deswegen erfordert VB 6 auch keine weitere Anweisung zur Einbindung der Klasse, wenn Sie diese verwenden wollen. Wie in anderen Sprachen auch, benötigen Sie in Visual Basic eine Variable vom Typ der Klasse. In VB 6 erzeugen Sie das Objekt dann mit dem `New`-Operator. Das folgende Beispiel verwendet ein CCircle-Objekt in der Ereignisprozedur eines Schalters:

```
Private Sub cmdCalcCircle_Click()
 ' Variable vom Typ der Klasse
 Dim c As CCircle

 ' Ein Objekt der Klasse erzeugen
 Set c = New CCircle

 ' Radius setzen
 c.Radius = 20

 ' Fläche berechnen und ausgeben
 MsgBox "Die Fläche eines Kreises mit " & c.Radius & " mm ist " & _
    c.GetSurface & " mm^2"
End Sub
```

Visual Basic .NET

In Visual Basic .NET können Sie, wie in C#, mehrere Klassen, die auch öffentlich sein können, in einer Datei unterbringen. VB.NET erfordert nicht, wie C#, die Deklaration eines Namespace für die Klassen. Die Deklaration einer Klasse beginnt mit `Public Class Name_der_Klasse` und endet mit `End Class`. Deklarieren Sie die Klasse einfach wie im folgenden Beispiel:

```
Imports System

' Deklaration der Klasse
Public Class Circle
   ' Eigenschaften
   public Radius AS Integer

   ' Methoden
   public Function getCircumference() AS Double
      return 3.1415927 * 2 * Radius
   End Function

   public Function getSurface() As Double
      return 3.1415927 * Radius * Radius
   End Function
End Class
```

Zur Erzeugung eines Objekts der Klasse müssen Sie eine Variable vom Typ der Klasse deklarieren und diese mit dem `New`-Operator erzeugen. Wenn die Datei, die die Klasse enthält, im selben Ordner wie die Programmdatei gespeichert ist, können Sie die Klassen einfach verwenden, ohne die Datei einbinden zu müssen (was dann, wie bei C#, über einen Namespace erfolgen würde):

```
imports System

Public Module Circle_Demo

Sub Main
 ' Variable für das Objekt deklarieren
 Dim c As Circle

 ' Das Objekt erzeugen
 c = New Circle()

 ' und verwenden
 c.Radius = 20
 Console.WriteLine("Die Flaeche eines Kreises mit " & _
   c.Radius & " mm ist " & c.getSurface() & " mm^2")
End Sub

End Module
```

2.3 Private und öffentliche Elemente einer Klasse

Eine Klasse sollte natürlich Eigenschaften und Methoden besitzen, die von außen über eine Referenz auf ein Objekt dieser Klasse verwendet werden können. Diese Elemente werden als öffentliche (Public-)Elemente bezeichnet. Innerhalb einer Klasse gibt es aber auch immer wieder den Bedarf, Daten zu speichern und Methoden zu programmieren, die privat sind und nur innerhalb der Klasse verwendet werden können. Private Eigenschaften werden häufig für die Kapselung verwendet, die ich weiter unten, in Abschnitt 2.4 beschreibe. Private Methoden werden häufig verwendet, wenn ein bestimmter Algorithmus innerhalb mehrerer (öffentlicher) Methoden einer Klasse benötigt wird. Im Prinzip ist das vergleichbar mit privaten Variablen und privaten Funktionen/Prozeduren in den Modulen der strukturierten Programmierung. In dem Zusammenhang spricht man übrigens häufig von der *Sichtbarkeit* von Klassen-Elementen. Öffentliche Elemente sind innerhalb und außerhalb der Klasse sichtbar, private Elemente nur innerhalb der Methoden der Klasse. Jede Programmiersprache stellt Ihnen Sprachelemente zur Verfügung, über die Sie die Sichtbarkeit der Klassenelemente festlegen können.

Die folgenden Beispiele demonstrieren die Anwendung von privaten Klassenelementen anhand einer neuen Methode *PI* für die Circle-Klasse. Diese Methode soll die Zahl PI möglichst genau berechnen und zurückgeben und wird von den Methoden zur Berechnung des Umfangs und der Fläche verwendet. Da diese Methode nur innerhalb der Klasse benötigt wird und nicht nach außen sichtbar sein soll, wird sie privat deklariert.

> Um die Beispiele überschaubar zu halten, habe ich die sehr komplizierte Berechnung von PI nicht in die Berechnungsmethode implementiert, sondern einfach eine möglichst genaue Zahl für PI zurückgegeben. Komplexere Algorithmen wie der von *Euler* berechnen PI wesentlich genauer. Im Internet finden Sie mehr Informationen dazu, wenn Sie bei `www.google.de` nach »calculate PI« suchen.

C++

In einer C++-Klasse können Sie mit den Schlüsselworten `Private` und `Public` Blöcke für private beziehungsweise öffentliche Deklarationen einrichten. Alle Deklarationen unterhalb von `Private` oder `Public` bis zum nächsten Sichtbarkeitsschlüsselwort bzw. bis zum Ende der Klassendeklaration besitzen dann diese Sichtbarkeit:

```
class Circle
{
 private:
   /* Private Elemente der Klasse */
   double PI()
   {
      return 3.1415927;
   }

 public:
   /* Öffentliche Elemente der Klasse */
   int Radius;

   /* Methoden */
   double getCircumference()
   {
      return PI() * 2 * Radius;
   }
   double getSurface()
   {
      return PI() * Radius * Radius;
   };
};
```

C#

In C# geben Sie vor der Eigenschaft bzw. Methode das Schlüsselwort `private` oder `public` an um die Sichtbarkeit einzustellen:

```
/* Deklaration des Namespace */
namespace Geometric_Classes
{
   using System;

   /* Deklaration der Klasse */
   public class Circle
   {
      /* Private Methode */
      private double PI()
      {
         return 3.1415927;
      }

      /* Eigenschaften */
      public int Radius;

      /* Methoden */
      public double getCircumference()
      {
         return PI() * 2 * Radius;
      }

      public double getSurface()
      {
         return PI() * Radius * Radius;
      }
   }
}
```

Object Pascal (Delphi/Kylix)

In einer Object Pascal-Klasse können Sie wie bei C++ mit den Schlüsselworten `Private` und `Public` Blöcke für private beziehungsweise öffentliche Deklaration einrichten. Alle Deklarationen unterhalb von `Private` oder `Public` bis zum nächsten Sichtbarkeitsschlüsselwort bzw. bis zum Ende der Klassendeklaration besitzen dann diese Sichtbarkeit:

```
type
  TCircle = class
    public
       Radius: integer;

    private
       function PI: double;

    public
       function getCircumference: double;
       function getSurface: double;
end;
```

Wenn Sie in einer Object Pascal-Klassedeklaration übrigens kein Sichtbarkeits-Schlüsselwort unterbringen, sind alle Elemente dieser Klasse automatisch öffentlich.

Alternativ können Sie jeder Deklaration das entsprechende Schlüsselwort voranstellen, wie das in anderen Sprachen auch üblich ist:

```
type
  TCircle = class
    public Radius: integer;
    private function PI: double;
    public function getCircumference: double;
    public function getSurface: double;
end;
```

Welche dieser Methoden Sie anwenden, ist Geschmackssache. Ich verwende lieber die erste Variante, weil diese weniger Schreibarbeit verursacht.

Java

Java verwendet ebenfalls die Schlüsselworte `private` und `public` zur Einstellung der Sichtbarkeit, die hier allerdings kleingeschrieben werden müssen. In Java müssen Sie jeder Methode und jeder Eigenschaft das Schlüsselwort einzeln voranstellen. Lassen Sie das Schlüsselwort weg, ist das Element automatisch öffentlich:

```
public class Circle
{
   public int Radius;

   private double PI()
   {
      return 3.1415927;
   }

   public double getSurface()
   {
      return PI() * Radius * Radius;
   }

   public double getCircumference()
   {
     return PI() * 2 * Radius;
   }
}
```

Visual Basic 6

In Visual Basic 6 schreiben Sie zur Einstellung der Sichtbarkeit einfach die Schlüsselworte `Private` oder `Public` an den Anfang der Eigenschaft- oder Methoden-Deklaration:

```
Option Explicit

' Eigenschaften
Public Radius As Long

' Private Methode
Private Function PI() As Double
  PI = 3.1415927
End Function

' Öffentliche Methoden
Public Function GetCircumference() As Double
  GetCircumference = PI * 2 * Radius
End Function

Public Function GetSurface() As Double
  GetSurface = PI * Radius * Radius
End Function
```

Visual Basic .NET

In Visual Basic .NET geben Sie, wie in C#, für die Einstellung der Sichtbarkeit vor der Eigenschaft bzw. Methode das Schlüsselwort `private` oder `public` an:

```
Imports System

' Deklaration der Klasse
Public Class Circle
   ' Eigenschaften
   public Radius AS Integer

   ' Private Methode
   Private Function PI() As Double
      PI = 3.14927
   End Function

   ' Öffentliche Methoden
   public Function getCircumference() AS Double
      return PI() * 2 * Radius
   End Function

   public Function getSurface() As Double
      return PI() * Radius * Radius
   End Function
End Class
```

2.4 Daten schützen: Die Kapselung

Die Kapselung ist ein wichtiges Grundkonzept der OOP. Kapselung bedeutet, dass Objekte den Zugriff auf ihre Daten selbst kontrollieren. Besitzt ein Objekt nur einfache Eigenschaften, so kann es den Zugriff auf seine Daten nicht kontrollieren. Ein Programmierer, der dieses Objekt benutzt, kann in die Eigenschaften hineinschreiben, was immer auch zu dem Datentyp der Eigenschaften passt. Das kann dazu führen, dass das Objekt ungültige Daten speichert und damit u. U. beim Aufruf von Methoden fehlerhaft reagiert. Stellen Sie sich ein Objekt zum flexiblen Drucken von Texten und Grafiken vor. Dieses Objekt besitzt Eigenschaften zum Einstellen der Ränder, die beim Drucken berücksichtigt werden. Stellt der Programmierer diese Eigenschaften z. B. auf negative Werte ein, wird der Ausdruck wahrscheinlich etwas konfus aussehen. Kontrolliert das Objekt jedoch den Zugriff auf diese Eigenschaften, so ist ein fehlerhaftes Einstellen (wie z. B. auf negative Werte) erst gar nicht möglich. Das Objekt kann dann unter allen Umständen sicher arbeiten.

Kapselung kann auf zwei Arten realisiert werden. Die ältere Methode ist, die Daten des Objekts einfach immer in privaten Variablen innerhalb der Klasse zu speichern und für den Zugriff auf diese Daten jeweils eine Methode zum Schreiben des Werts und eine zum Lesen des Werts zur Verfügung zu stellen. Diese Methoden können dann den Zugriff auf die Daten sehr gut kontrollieren. Eine andere, modernere Variante stellen einige Programmiersprachen zur Verfügung: Über so genannte *Property-Prozeduren* können Sie Eigenschaften implementieren, die nach außen wie eine normale Eigenschaft beschrieben und gelesen werden können, aber intern in Wirklichkeit aus einer Schreib- und einer Lesemethode bestehen. Der Vorteil einer solchen speziellen Eigenschaft ist, dass die Verwendung sehr einfach ist, weil einfach nur Werte hineingeschrieben und gelesen werden. Und trotzdem ist die Eigenschaft in der Lage, die geschriebenen oder gelesenen Daten zu kontrollieren.

2.4.1 Die klassische Kapselung

Die klassische Kapselung arbeitet so, dass die Daten des Objekts grundsätzlich in privaten Variablen gespeichert werden und zum Zugriff auf diese Daten eine Lese- und eine Schreibmethode erzeugt werden. Die Schreibmethode kann entscheiden, welche Daten in das Objekt geschrieben werden, die Lesemethode entscheidet, wie die Daten zurückgegeben werden. Das folgende Beispiel demonstriert dieses Vorgehen für eine Personenklasse, die den Zugriff auf das Geburtsdatum der Person kontrolliert:

```
unit Personen_Klasse;

interface

uses SysUtils;

type TPerson = class
 private
   { Private Variable zum Speichern des Geburtsdatums }
   FBirthdate: TDateTime;

 public
   { Einfache, ungeschützte öffentliche Eigenschaften }
   FirstName: string;
   LastName: string;

   { Schreibmethode für das Geburtsdatum }
   procedure setBirthdate(NewValue: TDateTime);

   { Lesemethode für das Geburtsdatum }
   function getBirthdate: TDateTime;

   { Lesemethode für das Alter, die ebenfalls auf das
     Geburtsdatum zurückgreift }
   function getAge: word;
end;

implementation

procedure TPerson.setBirthdate(NewValue: TDateTime);
begin
  { Die Schreibmethode überprüft, ob das übergebene Datum
    nicht in der Zukunft liegt }
  if NewValue > Date() then
     FBirthDate := 0
  else
     FBirthDate := NewValue;
end;

function TPerson.getBirthdate: TDateTime;
begin
  { Die Lesemethode gibt das Geburtsdatum
    oder weitere Überprüfungen einfach zurück }
  result := FBirthdate;
end;

function TPerson.getAge: word;
  var Year1, Month1, Day1: Word;
  var Year2, Month2, Day2: Word;
  var age: word;
begin
  { Berechnen des Alters in Jahren }
  DecodeDate(FBirthdate, Year1, Month1, Day1);
  DecodeDate(Date, Year2, Month2, Day2);
  age := Year2 - Year1;
  if (Month2 < Month1) or ((Month2 = Month1) and (Day2 < Day1)) then
     age := age - 1;
  result := age;
end;
```

```
end.
```

Die Kapselung ist in diesem Beispiel klar sichtbar an der Eigenschaft *FBirthdate*, deren Schreibmethode überprüft, ob das übergebene Datum in der Zukunft liegt. Die Lesemethode *getBirthdate* macht zwar nichts weiter, als das Geburtsdatum zurückzugeben, die zusätzliche Lesemethode *getAge* demonstriert aber, dass es auch Sinn macht, den lesenden Zugriff auf die Daten zu steuern. Ich habe dem Namen der Eigenschaft übrigens ein »F« vorangestellt, weil das in Object Pascal für private Variablen in Klassen so üblich ist, um private Eigenschaften besser von öffentlichen Eigenschaften unterscheiden zu können. Wofür das »F« steht, habe ich allerdings nie herausgefunden.

2.4.2 Bessere Kapselung mit Exceptions

Etwas ungeschickt an dem vorherigen Beispiel ist, dass der aufrufende Programmierer nicht unbedingt mitbekommt, dass die Schreibmethode fehlgeschlagen ist, wenn er ein ungültiges Datum übergibt. Er erkennt dies höchstens daran, dass das Geburtsdatum etwas eigenartig aussieht, wenn die Methode ein ungültiges Datum zurückweist (das ist dann der 31.12.1899, weil dies das Basisdatum für Object Pascal ist. Die Zahl 0 entspricht dem Basisdatum) und die Person ziemlich alt zu sein scheint (so alt war Smilla in dem Film auf jeden Fall nicht). Eine Methode, die den Zugriff auf die Daten kontrolliert, muss dem Programmierer das Fehlschlagen der Methode also so mitteilen, dass dieser gezwungen ist das Fehlschlagen zu quittieren. Dafür stellen die verschiedenen Sprachen so genannte *Exceptions* (Ausnahmefälle) zur Verfügung, die der Entwickler der Klasse im Fehlerfall aufrufen kann und die dann im Programm zu einem Laufzeitfehler führen, wenn der Fehlerfall eintritt.

> Mir ist bewusst, dass die etwas schwierig zu verstehenden Exceptions eigentlich nicht in einen Grundlagenartikel gehören, aber ich möchte Ihnen diese wichtige Technik nicht vorenthalten. Wenn Sie bereits Exceptions beziehungsweise Laufzeitfehler in Ihren Programmen abgefangen haben, wissen Sie ja eigentlich schon, was das ist. Jetzt sind Sie einmal in der glücklichen Lage, eigene Exceptions auslösen zu können.

Das folgende Beispiel demonstriert, wie in Object Pascal eine Exception ausgelöst wird:

```
procedure TPerson.setBirthdate(NewValue: TDateTime);
begin
  { Die Schreibmethode überprüft, ob das übergebene Datum
    nicht in der Zukunft liegt }
  if NewValue > Date() then
     begin
     { Beim Schreiben eines ungültigen Wertes erzeugt
       die Methode eine Exception }
     raise Exception.Create('Ein Geburtsdatum, ' +
        ' das in der Zukunft liegt, ist nicht möglich.');
     end
  else
     FBirthDate := NewValue;
end;
```

Das Abfangen der Exception sieht im Object Pascal folgendermaßen aus:

```
{ Aufruf der Schreibmethode mit einem unzulässigen Wert }
try
  { Im try-Block werden in Object Pascal die Anweisungen
    geschrieben, die eine Exception auslösen können
    und deren Exception abgefangen werden soll }
  p.setBirthdate(StrToDate('31.12.2099'));

  { Ausgabe der Daten der Person }
  writeln(p.FirstName + ' ' + p.LastName + ' ist am ' +
     DateToStr(p.getBirthDate) + ' geboren und ' +
```

```
     IntToStr(p.getAge) + ' Jahre alt.');
except
  { Die Anweisungen im except-Block werden nur dann
    ausgeführt, wenn im try-Block eine Exception aufgetreten ist }
  on e: Exception do
     writeln('Beim Versuch, das Geburtsdatum zu ' +
             'schreiben, ist ein Fehler ' +
             'aufgetreten: ' + e.Message);
end;

...
```

Aufgrund der Komplexität dieses Themas finden Sie hier keine weiteren Beispiele für die unterschiedlichen Sprachen. Verschiedene Bücher, wie z. B. die der Nitty-Gritty-Reihe von Addison Wesley, beschreiben, wie Sie Exceptions behandeln und erzeugen.

2.4.3 Kapselung mit Property-Prozeduren

Die Kapselung der Daten eines Objektes über Zugriffsmethoden ist in der Praxis bei der Verwendung des Objekts oft etwas mühselig. Viel schöner wäre doch, wenn Sie einfach einen Wert in die Eigenschaft hineinschreiben beziehungsweise die Eigenschaft einfach auslesen könnten und die Eigenschaft trotzdem den Zugriff kontrolliert. Genau dafür stellen die meisten modernen Programmiersprachen so genannte Property-Prozeduren zur Verfügung. Die Syntax zur Erzeugung solcher speziellen Eigenschaften unterscheidet sich in den einzelnen Sprachen sehr stark. Ich beschreibe die Grundlagen deshalb hier nur für Object Pascal. In Object Pascal benötigen Sie zur Erzeugung von Property-Prozeduren zunächst einmal, genau wie bei der einfachen Kapselung, eine private Variable, die das Datum speichert, und je eine Schreib- und Lesemethode. Diese Zugriffsmethoden werden allerdings nur im Private-Block der Klasse deklariert. Der Public-Block wird um eine spezielle Deklaration der Property-Prozedur erweitert:

```
unit Personen_Klasse;

interface

uses SysUtils;

type TPerson = class
 private
   { Private Variable zum Speichern des Geburtsdatums }
   FBirthdate: TDateTime;

   { Schreib-Methode für die Property-Prozedur }
   procedure setBirthdate(NewValue: TDateTime);

   { Lesemethode für die Property-Prozedur }
   function getBirthdate: TDateTime;

 public
   { Einfache, ungeschützte öffentliche Eigenschaften }
   FirstName: string;
   LastName: string;

   { Property-Prozedur für das Geburtsdatum }
   property Birthdate: TDateTime read getBirthDate write setBirthDate;

   { Methode zur Berechnung des Alters }
   function getAge: word;
end;
```

Die Implementierung der Zugriffsmethoden hat sich im Vergleich zum Beispiel bei der einfachen Kapselung gar nicht geändert, weswegen ich diese hier nicht mehr aufführe.

Beachten Sie, dass die Schreib- und die Lesemethode auf die private Eigenschaft *FBirthdate* zurückgreifen und nicht auf die nun öffentliche Eigenschaft *Birthdate*. Falls diese versehentlich doch auf *Birthdate* zurückgreifen würden, würde dies eine Endlos-Rekursion verursachen.

2.5 Konstruktoren und Destruktoren

In der OOP besitzt eigentlich[10] jedes Objekt einen Konstruktor und einen Destruktor. Der Konstruktor hat die Aufgabe, das Objekt zu erzeugen, der Destruktor zerstört das Objekt, wenn dieses nicht mehr benötigt wird. Auch wenn Sie selbst keinen Konstruktor und keinen Destruktor für eine Klasse erzeugen, besitzen alle Objekte dieser Klasse einen voreingestellten, einfachen Konstruktor und Destruktor.

Im Prinzip können Sie sich den Konstruktor vorstellen als eine Methode der Klasse, die direkt, ohne ein Objekt dieser Klasse aufgerufen wird, wenn Sie ein Objekt erzeugen, und die die Aufgabe hat, das Objekt mit Hilfe des »Bauplans« der Klasse im Arbeitsspeicher zu erzeugen. Abbildung 2.3 stellt den Vorgang bei der Erzeugung eines Objekts dar.

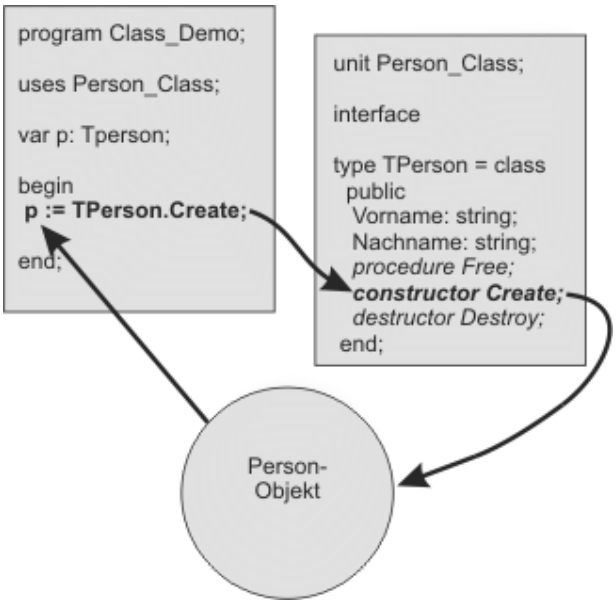

Abbildung 2.3: Der Konstruktor (in Object Pascal heißt dieser »Create«) wird aufgerufen und erzeugt ein Objekt, wobei er seine Klasse als Bauplan verwendet. Er gibt eine Referenz auf das Objekt zurück, die in eine Variable gespeichert wird. Die kursiv dargestellten Methoden sind von der Basisklasse TObject geerbt.

Den Destruktor können Sie sich ebenfalls vorstellen wie eine ganz normale Methode des Objekts. Er wird aufgerufen, wenn das Objekt zerstört, also aus dem Arbeitsspeicher wieder entfernt wird. Wie ich bereits erläutert habe, müssen Sie in einigen Programmiersprachen Objekte selbst aus dem Speicher entfernen, wenn Sie diese nicht mehr benötigen, andere Programmiersprachen, wie z. B. Java, entfernen Objekte selbstständig. Auf jeden Fall wird dann, wenn das Objekt zerstört wird, der Destruktor aufgerufen.

[10] Java, C# und VB.NET entsorgen ihre Objekte über einen so genannten Garbage Collector. Diese Sprachen kennen keinen Destruktor, aber so genannte Finalisierungsmethoden. Im Prinzip ist das aber dasselbe.

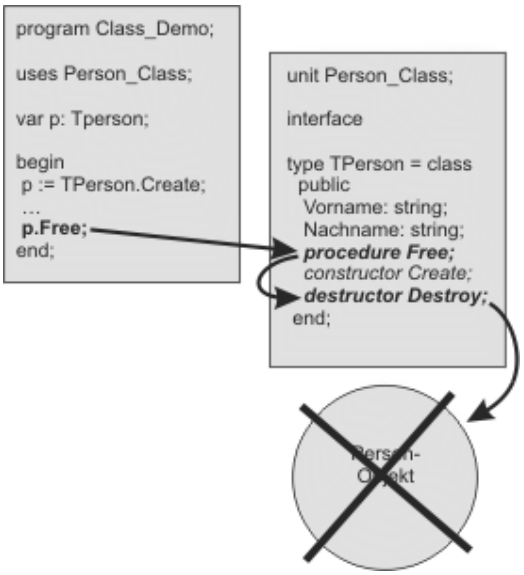

Abbildung 2.4:Die Free-Methode ruft den Destruktor auf. Dieser zerstört dann das Objekt. Die kursiv dargestellten Methoden sind von der Object Pascal-Basisklasse TObject geerbt. Die Abbildung gilt für Object Pascal, ist aber für andere Sprachen ähnlich zu verstehen.

Sie werden sich jetzt vielleicht fragen, warum dieses Wissen für Sie wichtig ist. Die Antwort ist ziemlich einfach: Sie können in den meisten Programmiersprachen eigene Konstruktoren und eigene Destruktoren in die Klasse integrieren. Sie verwenden eigene Konstruktoren, um ein Objekt direkt bei dessen Erzeugungen zu initialisieren. Bei der Personenklasse wäre es z. B. sinnvoll, einem erzeugten Personenobjekt bei der Erzeugungen gleich den Vornamen und den Nachnamen der Person zu übergeben. In der Realität geben wir einem Kind ja auch meist direkt schon dann einen Namen, wenn dieses geboren wird. Konstruktoren können aber auch verwendet werden, wenn das Objekt bei seiner Erzeugung selbstständig irgendwelche Aktionen ausführen soll. Das Beispiel in Abschnitt 2.5.2 demonstriert diese Technik für eine Klasse, deren Objekte dazu verwendet werden, Protokolldaten in eine Datei zu schreiben. In dieser Klasse öffnet der Konstruktor die Protokolldatei, damit die Schreibmethode der Klasse später in diese Datei schreiben kann.

Destruktoren werden analog für Aktionen verwendet, die ein Objekt vor seiner Zerstörung auf jeden Fall noch ausführen muss. Das Protokolldaten-Beispiel in Abschnitt 2.5.2 verwendet einen Destruktor, um die Datei, die im Konstruktor geöffnet wurde, wieder zu schließen.

Die Erzeugung des Objekts im Konstruktor und seine Zerstörung im Destruktor müssen Sie übrigens nicht selbst programmieren. Dazu rufen Sie einfach den Standardkonstruktor und -destruktor auf, den jede Klasse ja besitzt.

2.5.1 Der Konstruktor

Das folgende Beispiel demonstriert einen Konstruktor für die Personenklasse, der bei der Erzeugung einer Person den Vornamen und den Nachnamen der Person in die entsprechenden Eigenschaften schreibt:

```
unit Person_Class;

interface

type TPerson = class
  public
    { Die Eigenschaften der Klasse }
    Vorname: string;
    Nachname: string;
    Strasse: String;
    Ort: string;

    { Der Konstruktor }
    constructor Create(Vorname, Nachname: String);
 end;

implementation

{ Der Konstruktor }
constructor TPerson.Create(Vorname, Nachname: String);
begin
  { Aufruf des Standardkonstruktors, damit dieser
    ein Objekt der Klasse erzeugt }
  inherited Create;

  { Setzen der übergebenen Eigenschaften }
  self.Vorname := Vorname;
  self.Nachname := Nachname;
end;

end.
```

> Beachten Sie, dass Sie in den meisten Programmiersprachen den Standardkonstruktor direkt als erste Anweisung in Ihrem neuen Konstruktor aufrufen müssen. Falls Sie das vergessen, kann es sein – das kommt ganz auf die Programmiersprache an –, dass das Objekt nicht erzeugt wird und beim Zugriff auf das Objekt ein Laufzeitfehler oder sogar ein Programmabsturz erfolgt. Sie sollten es sich einfach zur Gewohnheit machen, als erste Anweisung in einem neuen Konstruktor den Standardkonstruktor aufzurufen.

Wenn Sie dann ein Objekt der Klasse erzeugen, müssen Sie den Vornamen und den Nachnamen übergeben:

```
program Konstruktor_Demo;

{$APPTYPE CONSOLE}

uses SysUtils, Person_Class;

var p: TPerson;

begin
  { Erzeugung eines Objekts der Personklasse }
  p := TPerson.Create('Ford', 'Prefect');

  writeln('Der Name der Person ist ' + p.Vorname + ' ' + p.Nachname);
  readln;

  { Freigeben des Objekts }
  p.Free;
end.
```

Der Aufruf des Standardkonstruktors und der self-Operator

Wie Sie in dem Beispiel sehen, rufen Sie den Standardkonstruktor in Ihrem neuen Konstruktor in Object Pascal mit Hilfe der `inherited`-Anweisung auf. Erst danach beginnen Sie mit den Initialisierungen. Die in dem Beispiel übergebenen Werte für den Vornamen und den Nachnamen der Person werden in die Eigenschaften geschrieben. In dem Beispiel tritt ein Problem auf, das in der Praxis sehr häufig anzutreffen ist: Die Bezeichner der Argumente des Konstruktors sind dieselben wie die der Eigenschaften. Deswegen müssen Sie dem Compiler mitteilen, was Sie mit den Bezeichnern denn nun meinen. Würden Sie nur `Vorname := Vorname;` schreiben, würde der Compiler annehmen, dass Sie mit beiden Bezeichnern das übergebene Argument meinen. Hier kommt (in Object Pascal) der Operator `self` ins Spiel. In einer Methode einer Klasse – und der Konstruktor ist eigentlich auch nichts weiter als eine Methode – steht dieser Operator für das Objekt selbst, ist also eine Referenz auf das Objekt, das später aus der Klasse erzeugt wird. Wenn Sie also in Object Pascal `self.`*Irgendetwas* schreiben, steht *Irgendetwas* immer für eine Eigenschaft bzw. eine Methode des Objekts. Dieser wichtige Operator wird übrigens in C++, C# und Java »`this`« genannt, Visual Basic und Visual Basic .NET nennen diesen Operator »`Me`«.

Geerbte Standardkonstruktoren

In den meisten echt objektorientierten Programmiersprachen wird eine Klasse, wenn Sie diese nicht explizit von einer anderen Klasse ableiten, immer automatisch von einer grundlegenden Basisklasse abgeleitet. Wenn Sie in Object Pascal z. B. schreiben

```
type TPerson = class
```

macht der Compiler daraus

```
type TPerson = class(TObject)
```

Sie sehen diese Veränderung nicht, der Compiler verwendet sie aber.

Ihre neue Klasse erbt damit alle Methoden und Eigenschaften dieser Basisklasse. Diese *Vererbung* erkläre ich eigentlich erst Kapitel 3, für das Verständnis der Standardkonstruktoren und -destruktoren muss ich hier aber ein wenig vorgreifen. Diese sind nämlich in den grundlegenden Basisklassen definiert und werden in die abgeleitete Klasse vererbt. Aus diesem Grund müssen Sie in Object Pascal z. B. auch *inherited* `Create;` schreiben, um den Standardkonstruktor aufzurufen. Die Übersetzungen des englischen Begriffs »inherited« ist »geerbt«. Sie rufen also den geerbten Konstruktor auf.

> Wenn Sie einen so geerbten Konstruktor aufrufen, müssen Sie natürlich dessen Argumente – sofern dieser Argumente besitzt – übergeben. Die Standardkonstruktoren von grundlegenden Basisklassen besitzen eigentlich nie Argumente. Wenn Sie aber Ihre Klasse von einer anderen, speziellen Klasse ableiten, kann es sein, dass der Konstruktor dieser Klasse doch Argumente besitzt. Dann müssen Sie in der Dokumentation dieser Klasse nachschauen, um herauszufinden, wie Sie den geerbten Konstruktor aufrufen müssen.

2.5.2 Der Destruktor

Wie ich es weiter oben, bei der Einführung zu Konstruktoren und Destruktoren, bereits erwähnt habe, werden Destruktoren verwendet, wenn ein Objekt bei seiner Zerstörung noch irgendwelche Aufräumarbeiten erledigen muss. In der Praxis kommt das relativ selten vor, nämlich eigentlich immer nur dann, wenn das Objekt in seinem Konstruktor irgendwelche Ressourcen (wie z. B. Dateien) geöffnet hat, die auf jeden Fall wieder geschlossen werden müssen. Das folgende Beispiel demonstriert die Anwendung eines Destruktors in einer Klasse, die für Protokollierungen verwendet wird. Mit Hilfe dieser Klasse soll es dem Programmierer auf einfache Weise möglich sein, während des Programmablaufs irgendwelche Protokolldaten in eine Log-Datei zu schreiben. Ein Objekt dieser Klasse soll in seinem Konstruktor eine Textdatei öffnen, damit die Log-Methode in diese Datei hineinschreiben kann. Der Destruktor soll die Datei dann schließen, wenn das Objekt zerstört wird. Diese Anwendung ist relativ typisch für die Verwendung eines Destruktors, denn die Datei darf erst dann geschlossen werden, wenn das Objekt nicht mehr benötigt wird (alternativ könnte man statt eines eigenen Destruktors auch eine Methode in die Klasse implementieren, die die Datei schließt. Diese müsste der Programmierer dann aber explizit aufrufen. Und das wird in der Praxis sehr häufig einfach vergessen).

```
unit Log_Class;

interface

type TLogFile = class
  private
    f: Textfile;

  public
    { Der Konstruktor }
    constructor Create(Filename: string);

    { Die Methode zum Protokollieren }
    procedure Log(Text: string);

    { Der Destruktor }
    destructor Destroy; override;
end;

implementation

{ Der Konstruktor }
constructor TLogFile.Create(Filename: string);
begin
  { Aufruf des geerbten Konstruktors }
  inherited Create;

  { Initialisieren des Objekts: Öffnen der Datei }
  AssignFile(f, FileName); { Datei der Variable zuweisen }
  Rewrite(f); { Datei neu erzeugen und öffnen }
end;

{ Die Methode zum Protokollieren }
procedure TLogFile.Log(Text: string);
begin
  writeln(f, Text);
end;

{ Der Destruktor }
destructor TLogFile.Destroy;
begin
  { Aufräumarbeiten: Schließen der Datei }
  CloseFile(f);

  { Aufruf das geerbten Destruktors }
  inherited Destroy;
```

```
end;
end.
```

Wie Sie dem Beispiel entnehmen können, heißt der Destruktor in Object Pascal »`Destroy`«, obwohl ja eigentlich die `Free`-Methode aufgerufen wird, wenn ein Objekt zerstört werden soll. Das hängt damit zusammen, dass die `Free`-Methode ein wenig mehr macht als der Destruktor. Diese überprüft nämlich vor dem Aufruf von `Destroy`, ob das Objekt überhaupt noch existiert. Würden Sie `Destroy` für ein gar nicht mehr existierendes Objekt direkt aufrufen, würde das zu einem Programmabsturz führen. Diese Eigenart der Unterscheidung einer Technik zum Zerstören des Objekts (in Object Pascal: `Free`) zum eigentlichen Destruktor kommt aber im Prinzip in allen Programmiersprachen vor. In Java z. B. wird ein Objekt normalerweise ganz automatisch von einem so genannten *Garbage-Collector* (»Müllsammler«) zerstört, wenn das Programm dieses nicht mehr benötigt. Der Destruktor wird in diesem Fall vom Garbage-Collector aufgerufen.

Wie Sie dem Beispiel ebenfalls entnehmen können, müssen Sie den Destruktor in Object Pascal mit dem Schlüsselwort `override` deklarieren. Damit erreichen Sie, dass Sie den geerbten Destruktor so *überschreiben*, dass der Compiler auch wirklich *Ihren* Destruktor aufruft, und nicht den der Basisklasse. Der Grund dafür hängt mit der Vererbung zusammen, und deswegen erkläre ich diesen auch erst im nächsten Kapitel. Würden Sie `override` weglassen, würde der Compiler Ihren Konstruktor einfach gar nicht aufrufen.

Die Anwendung der Klasse unterscheidet sich eigentlich überhaupt nicht von den bisherigen Beispielen, soll aber der Vollständigkeit halber nicht fehlen:

```
program Destruktor_Demo;

{$APPTYPE CONSOLE}

uses SysUtils, Log_Class;

var logfile: TLogFile;

begin
  { Erzeugung eines Objekts der LogFile-Klasse.
    Hier wird der Konstruktor aufgerufen }
  logfile := TLogFile.Create('c:\Testlog.txt');

  { Loggen des Programmstarts }
  logfile.Log('Programmstart: ' + DateToStr(Date));

  { ... }

  { Loggen des Programmendes }
  logfile.Log('Programmende: ' + DateToStr(Date));

  { Freigeben des Objekts.
    Hier wird der Destruktor aufgerufen }
  logfile.Free;
end.
```

2.5.3 Konstuktoren und Destruktoren in den verschiedenen Sprachen

C++

In C++ deklarieren Sie den Konstruktor und den Destruktor im Public-Bereich der Klasse. Der Konstruktor erhält denselben Namen wie die Klasse und eine optionale Argumentliste. Der Destruktor wird ebenfalls mit dem Namen der Klasse benannt, allerdings mit einer vorangestellten Tilde (~).

> Den Destruktor sollten Sie in C++ immer virtuell deklarieren. Damit stellen Sie sicher, dass der Compiler auch dann den Destruktor der Klasse aufruft, wenn die Klasse in Programmen eingesetzt wird, die mit Polymorphismus arbeiten. Polymorphismus beschreibt das Kapitel 3, dann gehe ich auch noch einmal auf Destruktoren ein. Merken Sie sich hier nur, dass Sie den Destruktor in C++ immer mit `virtual` deklarieren sollten.

```
#include <iostream.h>
#include <string.h>
#include <fstream.h>

class LogFile
{
  private:
    ofstream* out;

  public:
    /* Der Konstruktor */
    LogFile(char* Filename)
    {
        /* Öffnen der Datei */
        out = new ofstream(Filename);
    };

    /* Der Destruktor */
    virtual ~LogFile()
    {
        /* Freigeben des Objekts und damit
         * Schliessen der Datei */
        delete out;
    };

    /* Eine Methode zum Protokollieren */
    void Log(string Info)
    {
        *out << Info << "\n";
    };
};
```

Wenn Sie in einer von einer Basisklasse abgeleiteten Klasse (siehe bei »Vererbung« im Abschnitt 3.1) den geerbten Konstruktor aufrufen wollen, um dessen Implementierung zu nutzen, müssen Sie den Klassennamen der Basisklasse gefolgt von einer eventuellen Argumentliste, durch einen Doppelpunkt getrennt der Argumentliste des Konstruktors anhängen:

```
    /* Der Konstruktor */
    LogFile(char* Filename): BaseFile(Filename)
```

Wenn Sie keinen Konstruktor angeben, erzeugt der Compiler automatisch einen Aufruf des Standardkonstruktors der Basisklasse.

C#

In C# erhält der Konstruktor wie in C++ denselben Namen wie die Klasse und eine optionale Argumentliste. Der Destruktor erhält ebenfalls den Namen der Klasse, allerdings mit einer vorangestellten Tilde (~). Der Konstruktor sollte öffentlich deklariert werden, dem Destruktor stellen Sie keinen Modifizierer voran. Wenn Sie im Konstruktor einen geerbten Konstruktor aufrufen wollen, geben Sie das Schlüsselwort `base` gefolgt von einer Argumentliste, mit einem Doppelpunkt getrennt hinter der Argumentliste des Konstruktors an. Wenn Sie keinen besonderen geerbten Konstruktor aufrufen wollen, können Sie `base()` auch weglassen, der Compiler fügt diese Anweisung automatisch ein.

> Sie sollten beachten, dass der Destruktor erst dann aufgerufen wird, wenn der Garbage Collector Zeit hat das Objekt aus dem Speicher zu entfernen. Wenn das Objekt Ressourcen geöffnet hält, bleiben diese unnötig geöffnet. Microsoft empfiehlt daher, dass Sie zum Schließen der Ressourcen eine Methode *Close* (wenn die Ressource danach wieder geöffnet werden kann) oder *Dispose* (wenn die Ressource für das Objekt endgültig geschlossen wird) in das Objekt einfügen.

```
public class LogFile
{
   private System.IO.StreamWriter s;

   /* Der Konstruktor */
   public LogFile(string Filename): base()
   {
      /* Öffnen der Datei */
      s = new System.IO.StreamWriter(Filename, true);
   }

   /* Der Destruktor */
   ~LogFile()
   {
      /* Schliessen der Datei */
      s.Close();
   }

   /* Weil der Destruktor zu einem unbekannten
    * Zeitpunkt nach der Freigabe der Objekt-
    * Variablen aufgerufen wird, wird empfohlen
    * die Ressourcen immer auch über eine
    * Dispose-Methode freizugeben */
   public void Dispose()
   {
      s.Close();
   }

       /* Eine Methode zum Protokollieren */
   public void Log(string Info)
   {
      s.WriteLine(Info);
   }
}
```

Object Pascal (Delphi/Kylix)

Konstruktoren und Destruktoren für Object Pascal wurden bereits in Abschnitt 2.5 behandelt.

Java

In Java erhält der Konstruktor wie in C++ denselben Namen wie die Klasse und eine optionale Argumentliste. Der Destruktor wird allerdings wie eine normale Methode deklariert und erhält den Namen `finalize` und den Sichtbarkeitsberich `protected`. Wenn Sie im Konstruktor einen geerbten Konstruktor aufrufen wollen, verwenden Sie den `super`- Operator gefolgt von der Argumentliste des geerbten Konstruktors. Der Aufruf des geerbten Konstruktors muss die erste Anweisung im Konstruktor sein. Wenn Sie keinen besonderen geerbten Konstruktor aufrufen wollen, können Sie `super()` auch weglassen, der Compiler fügt diese Anweisung dann automatisch ein.

> Sie sollten beachten dass der Destruktor erst dann aufgerufen wird wenn der Garbage Collector Zeit hat das Objekt aus dem Speicher zu entfernen. Wenn das Objekt Ressourcen geöffnet hält bleiben diese unnötig geöffnet. Implementieren Sie daher, wie bei C# und Visual Basic .NET, eine Methode *Close* (wenn die Ressource danach wieder geöffnet werden kann) oder *Dispose* (wenn die Ressource für das Objekt endgültig geschlossen wird) in der die Ressource geschlossen wird.

```
import java.io.*;

/* Anm.: Die Ausnahmen, die beim Dateizugriff abgefangen
 * oder weitergegeben werden müssen, werden über die
 * throws-Anweisung an die benutzende Klasse weitergegeben. */

class LogFile
{
   private FileWriter f;

   /* Der Konstruktor */
   public LogFile(String Filename) throws java.io.IOException
   {
      /* Aufruf des geerbten Standardkonstruktors
       * (ist nur dann notwendig, wenn ein bestimmter
       * Konstruktor aufgerufen werden soll der
       * Argumente besitzt */
      super();
      /* Erzeugen eines neuen Output-Stream */
      f = new FileWriter(Filename);
   }

   /* Der Destruktor */
   protected void finalize() throws java.io.IOException
   {
      f.close();
   }

   /* Eine Methode zum Protokollieren */
   public void Log(String Info) throws java.io.IOException
   {
      f.write(Info + "\r\n");
   }

   /* Eine Methode zum expliziten Schliessen des Streams */
   public void Dispose() throws java.io.IOException
   {
      f.close();
   }
}
```

> Sie können in Java den Garbage Collector dazu auffordern, dass dieser die freien Objekte möglichst bald entfernt. Setzen Sie dazu die Objektvariablen auf `null` und rufen Sie `System.gc()` auf:

```
public class Log_Demo
{
   public static void main(String[] args)
   {
      LogFile log = null;
      try
      {
         log = new LogFile("C:\\Test.log");
         log.Log("Test");
         log.Dispose();
      }
      catch (Exception ex)
      {
         System.out.println(ex.getMessage());
      }

      /* Expliziter Aufruf des Garbage Collectors damit die
       * Destruktoren noch innerhalb der Programmausführung
       * aufgerufen werden. Dazu wird von Sun empfohlen, die
       * Objektvariablen auf null zu setzen und mit System.gc()
       * dem Garbage Collector mitzuteilen, dass es eine
       * gute Idee wäre jetzt die Objekte freizugeben (was aber nicht
       * garantiert ist  */
      log = null;
      System.gc();
   }
}
```

Visual Basic 6

Visual Basic 6 kennt keine leider echten Konstruktoren. Sie können in einer Klasse das `Initialize`-Ereignis verwenden um bei der Erstellung eines Objekts dieser Klasse Initialisierungen vorzunehmen. Leider handelt es sich dabei um ein Ereignis, weswegen Sie keine Möglichkeit besitzen einem Objekt direkt bei seiner Initialisierung Argumente zu übergeben. Eine Notlösung ist die Implementierung einer separaten Initialisierungsmethode, die nach dem Erstellen der Instanz aufgerufen wird. Visual Basic 6-Klassen kennen zwar auch keinen echten Destruktor, aber das `Terminate`-Ereignis besitzt für den Programmierer dieselbe Bedeutung. Da Visual Basic keine echte Vererbung und auch keine Konstruktoren kennt besteht kein Bedarf, einen geerbten Konstruktor aufzurufen.

```
Private f As Integer

' Der "Konstruktor"
Public Sub Init(Filename As String)
  ' Öffnen der Datei
  f = FreeFile()
  Open Filename For Append as #f
End Sub

' Der "Destruktor"
Private Sub Class_Terminate()
  ' Schliessen der Datei
  Close f
End Sub

' Eine Methode zum Protokollieren
Public Sub Log(Info As String)
  Print #f, Info & vbCrLf
End Sub
```

Visual Basic .NET

In Visual Basic .NET erhält der Konstruktor den Namen `New` und eine optionale Argumentliste. Der Konstruktor sollte die Sichtbarkeit »Public« erhalten. Der Destruktor besitzt den Namen `Finalize` und muss mit `Protected Overrides` deklariert werden. Wenn Sie im Konstruktor einen geerbten Konstruktor aufrufen wollen, verwenden Sie `MyBase.New` gefolgt von einer Argumentliste als erste Anweisung im Konstruktor. Wenn Sie keinen besonderen geerbten Konstruktor aufrufen wollen können Sie `MyBase.New()` auch weglassen, der Compiler fügt dies automatisch ein.

> Sie sollten beachten, dass der Destruktor erst dann aufgerufen wird, wenn der Garbage Collector Zeit hat das Objekt aus dem Speicher zu entfernen. Wenn das Objekt Ressourcen geöffnet hält, bleiben diese unnötig geöffnet. Microsoft empfiehlt daher, dass Sie zum Schließen der Ressourcen eine Methode *Close* (wenn die Ressource danach wieder geöffnet werden kann) oder *Dispose* (wenn die Ressource für das Objekt endgültig geschlossen wird) in das Objekt einfügen.

```
Imports System

Public Class LogFile
   Private s As System.IO.StreamWriter

   ' Der Konstruktor
   Public Sub New(Filename As String)
      ' Aufruf des geerbten Standardkonstruktors
      ' (ist nur dann notwendig, wenn ein bestimmter
      ' Konstruktor aufgerufen werden soll der
      ' Argumente besitzt
      MyBase.New()
      ' Öffnen der Datei
      s = new System.IO.StreamWriter(Filename, True)
   End Sub

   ' Der Destruktor
   Protected Overrides Sub Finalize()
      ' Schliessen der Datei
      s.Close()
   End Sub

   ' Weil der Destruktor zu einem unbekannten
   ' Zeitpunkt nach der Freigabe der Objekt-
   ' Variablen aufgerufen wird wird empfohlen
   ' die Ressourcen immer auch über eine
   ' Dispose-Methode freizugeben.
   Public Sub Dispose()
      s.Close()
   End Sub

   ' Eine Methode zum Protokollieren
   Public Sub Log(Info As String)
      s.WriteLine(Info)
   End Sub
End Class
```

3 Vererbung und Polymorphismus

3.1 Vererbung

Die Vererbung ist, neben der Kapselung, ein weiteres, sehr wichtiges Grundkonzept der OOP. Über die Vererbung können Sie neue Klassen erzeugen, die alle Eigenschaften und Methoden von einer Basisklasse, die bei der Erzeugung der neuen Klasse angegeben wird, erben. So können Sie sehr einfach die Funktionalität einer bereits vorhandenen Klasse erben, erweitern und/oder neu definieren.

Das *Erweitern* einer Klasse ist nicht besonders schwierig: Sie implementieren dazu einfach weitere Eigenschaften und Methoden in der neuen Klasse. Neue Methoden können dabei auch geerbte Methoden aufrufen, womit Sie die – in der Regel getestete und für gut befundene – Funktionalität der Basisklasse auf einfache Weise wiederverwenden können. In diesem Sinne wird die Vererbung häufig für Programme genutzt, bei denen eine Basisfunktionalität in mehreren, unterschiedlichen Klassen vorhanden sein soll.

Aber auch das so genannte *Überschreiben* geerbter Methoden mit einer neuen Programmierung ist möglich. So können Sie neue Klassen erzeugen, die im Prinzip genauso verwendet werden wie die Basisklasse, die aber ihren Job anders erledigen.

Diese beiden Grundkonzepte der Vererbung, das *Erweitern* und *Überschreiben*, werden häufig gemeinsam genutzt.

> Die Vererbung wird in der Praxis für *eigene* Klassen nur relativ selten eingesetzt. Ein Grund dafür ist, dass relativ selten wirklicher Bedarf für Vererbung besteht. Ein anderer Grund ist, dass Programme, die Vererbung sehr intensiv nutzen, meist sehr unübersichtlich und dann auch fehleranfällig werden. Wenn Sie aber echt objektorientierte Programmiersprachen, wie z. B. Java und C# verwenden, kommen Sie ohne die Vererbung nicht aus. Diese Sprachen stellen Ihnen viele Programm-Komponenten in Form von Klassen zur Verfügung, die aber erst erweitert werden müssen, bevor sie sinnvoll verwendet werden können. Wenn Sie z. B. in Java ein Windows-Fenster für Ihre Anwendung benötigen, leiten Sie eine neue Klasse von der Java-Klasse `Frame` ab. Die Klasse `Frame` liefert nur ein einfaches Fenster, das noch um die benötigten Fensterelemente (wie Schalter, Textfelder etc.) erweitert werden muss. Ohne Vererbung könnten Sie die Frame-Klasse nicht erweitern und müssten immer mit einfachen Fenstern ohne weitere Elemente arbeiten.

3.1.1 Warum Vererbung?

Die wichtigen Punkte bei der Vererbung sind:

- die Wiederverwendung von Programmcode,
- die Wartbarkeit von Programmen
- und der über die Vererbung erreichte Polymorphismus.

Den Polymorphismus beschreibe ich erst später, im Abschnitt 3.2. Die Wiederverwendung und die Wartbarkeit sind miteinander verwoben. Ich will Ihnen diese Features anhand eines Beispiels erläutern:

Bei der Programmierung ist es manchmal notwendig, dass eine gewisse *Basisfunktionalität*, die in einem Programm (oder Programmteil) verwendet wird, in einem anderen Programm in einer *erweiterten Form* zur Verfügung stehen muss. Die Basisfunktionalität darf dabei allerdings nicht verändert werden, damit beide Programme zuverlässig laufen.

Das folgende Beispiel verdeutlicht diesen wichtigen Punkt: Eine Klasse soll eine serielle Schnittstelle des Computers repräsentieren und die Kommunikation über diese mit irgendwelchen angeschlossenen Geräten vereinfachen. Diese Klasse sollte einige Eigenschaften zur Einstellung der Parameter der seriellen Schnittstelle besitzen. Das wären zum Beispiel Eigenschaften wie *ComPort*, *Baudrate*, *Parity* und *Handshake* (die Bedeutung dieser Parameter ist an dieser Stelle nicht wichtig). Eine Methode *Open* öffnet die serielle Schnittstelle mit den angegebenen Parametern, eine Methode *Send* erledigt das Senden von Zeichenketten. Eine Methode *Read* liest empfangene Zeichen aus dem Puffer der serielle Schnittstelle und gibt diese zurück. Ich nenne diese Klasse *TComm*. Diese Klasse berücksichtigt beim Senden und Empfangen von Daten kein besonderes Protokoll[11], Daten werden einfach Zeichen für Zeichen gesendet und empfangen. Irgendwann ist die Klasse fertig gestellt, getestet und funktioniert (hoffentlich) einwandfrei.

```
type TComm = class
 public
   ComPort: integer;
   Baudrate: integer;
   Handshake: integer;
   Parity: char;
   procedure Open;
   function Send(Data: string): boolean;
   function Read: String;
   procedure Close;
end;

{ Die Programmierung der Methoden ist nicht Bestandteil
  dieses Beispiels }
```

In einem Programm, das irgendwann später entwickelt wird, soll eine Kommunikation über die serielle Schnittstelle mit einem externen Gerät programmiert werden, das Daten nach einem DIN-Protokoll sendet und empfängt. Bei einem solchen Protokoll werden die Daten u. a. mit einer Checksumme versehen, die beim Empfang dazu verwendet wird, zu überprüfen, ob die Daten korrekt empfangen wurden. Der Programmierer würde sich jetzt sehr viel Arbeit machen, wenn er das, was in der *TComm*-Klasse bereits fertig programmiert ist, nicht wiederverwenden würde. Er schreibt also einfach eine neue Klasse, die von der *TComm*-Klasse abgeleitet wird und die damit alle Eigenschaften und Methoden erbt. Um die neue Funktionalität zu programmieren, muss er lediglich eine neue Eigenschaft *Protocol* hinzufügen und die Methoden *Send* und *Read* neu schreiben. Dabei kann er auf die bereits vorhandene Funktionalität der geerbten Methoden zurückgreifen, indem er diese einfach aufruft.

```
interface

uses Comm;

type TCommDIN = class(TComm)
 public
   { Neue Eigenschaft das zu verwendende Protokoll }
   Protocol: string;
   { Die Send- und die Read-Methode wird neu definiert,
     alles andere wird geerbt }
   function Send(Data: string): boolean;
   function Read: String;
end;

implementation

function TCommDIN.Send(Data: string): boolean;
```

[11] Ein Protokoll beschreibt (im Computer), wie Daten gesendet und empfangen werden. Einfache Protokolle versehen die gesendeten Daten mit einer so genannten Checksumme, die aus den Daten errechnet und an diese hinten angehängt wird. Mit Hilfe dieser Checksumme kann der Empfänger überprüfen, ob die Daten korrekt gesendet wurden. Komplexere Protokolle, wie TCP/IP, trennen die gesendeten Daten zusätzlich noch in kleine Pakete auf, die jedes für sich mit einer Checksumme und einer Paketnummer versehen werden.

```
begin
  { Anhängen einer Checksumme, je nach Protokoll.
    Für das Beispiel nur simuliert. }
  If Protocol = "66348" then
     Data := AddCheckSum(Data);

  { Aufruf der geerbten Methode }
  result := inherited Send(Data);
end

...
```

Das ist der Kernpunkt der Vererbung, die einfache Wiederverwendung von Programmcode. Die Wiederverwendung hängt direkt mit dem weiteren wichtigen Feature der Vererbung zusammen, der sehr einfachen Wartbarkeit von objektorientierten Programmen: Wenn der Entwickler dieser Klassen irgendwann einmal feststellt, dass eine Basismethode (z. B. die Send-Methode) nicht richtig funktioniert, kann er diese einfach in der Basisklasse »reparieren«. Da alle abgeleiteten Klassen die Basismethode erben, gilt die Reparatur dann für alle abgeleiteten Klassen, auch wenn dies noch so viele sind. Dabei kann z. B. vorkommen, dass der Programmierer vielleicht auch vergessen hat, wichtige Eigenschaften oder Methoden zu implementieren. In dem obigen Beispiel fehlen z. B. die wichtigen Parameter *DataBits* und *StopBits*. Kein Problem: Diese Eigenschaften werden einfach der Basisklasse hinzugefügt, in der Open-Methode berücksichtigt und damit automatisch in alle abgeleiteten Klassen vererbt. OK, bei der Anwendung von Objekten der abgeleiteten Klassen müssen Sie nun daran denken, diese Eigenschaften jetzt auch zu setzen, damit die Objekte korrekt funktionieren. Das kann Ihnen die Vererbung nicht abnehmen. Aber Sie müssen diese Eigenschaften nicht in jede der abgeleiteten Klassen implementieren und vor allen Dingen nicht in der Open-Methode immer wieder neu berücksichtigen. Das ist einfache Wartbarkeit.

3.1.2 Das Beispiel

Ich beschreibe die Vererbung an einem einfachen Beispiel, einer Kontoverwaltung für eine Bank. Dazu benötigen Sie Girokonten und Sparkonten, die Sie – natürlich ☺ – als Klasse implementieren. Diese beiden Kontenarten besitzen einige gemeinsame Elemente, wie z. B. eine Nummer, einen (Konto-)Stand und Methoden zum Einzahlen und Abheben. Das Girokonto besitzt zusätzlich einen Dispobetrag (das ist der Betrag, bis zu dem der Kontostand überzogen werden kann), das Sparkonto soll individuell verzinst werden und besitzt deswegen einen Zinssatz (Girokonten werden bei unserer Bank nicht verzinst). Um die Berechnung von Zinsen für das Sparkonto möglichst flexibel handhaben zu können, soll dieses eine Methode *Verzinsen* besitzen. Diese Methode soll die Zinsen für den aktuellen Kontostand für den Zeitraum seit der letzten Verzinsung berechnen und auf den Kontostand addieren (ich weiß, dass dieses Vorgehen nicht der Praxis entspricht, aber es vereinfacht die Beispiele).

Für die gemeinsamen Elemente der beiden Kontenarten wird nun eine Basisklasse *Konto* erzeugt, von der die Klassen für das Girokonto und das Sparkonto abgeleitet werden. Abbildung 3.1 stellt diese Vererbung dar.

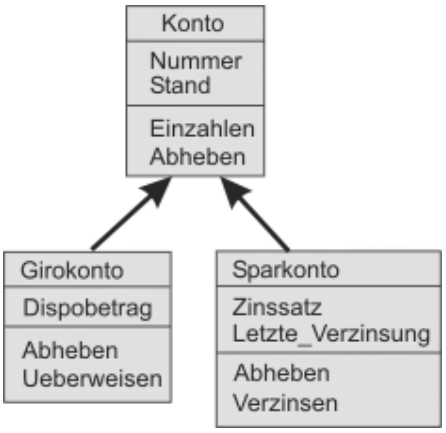

Abbildung 3.1: Vererbung bei einer einfachen Kontoverwaltung. Das vielleicht etwas irritierende (aber übliche) Zeigen der Pfeile nach oben deutet an, dass die unteren Klassen von der oberen Klasse vererbt werden.

Die Klasse *Konto* definiert die Basisfunktionalität eines Kontos. Über die Methode *Einzahlen* kann ein Betrag auf das Konto eingezahlt werden. Die Methode *Abheben* reduziert den Kontostand um den abgehobenen Betrag.

Im *Girokonto* wird eine Eigenschaft *Dispobetrag* und eine Methode zum Überweisen eines Betrags auf ein anderes Konto hinzugefügt. Die Abheben-Methode soll in dieser Klasse ein Abheben bis zum eingestellten Dispobetrag ermöglichen, also eine andere Funktionalität bieten als die entsprechende Methode in der Basisklasse.

Dem *Sparkonto* wird eine Methode *Verzinsen* hinzugefügt, die auf die neuen Eigenschaften *Zinssatz* und *Letzte_Verzinsung* zurückgreift. Die Abheben-Methode lässt hier ein Abheben nur dann zu, wenn der Kontostand dadurch nicht negativ wird.

3.1.3 Die Grundlagen der Vererbung

Fast alle modernen Programmiersprachen bieten Ihnen die Möglichkeit, neue Klassen zu erzeugen, die von vorhandenen Klassen erben. Bei den im Artikel behandelten Sprachen gehören Visual Basic 6 und VBScript leider nicht dazu. Diese beiden Sprachen lassen die echte Vererbung, die so genannte *Implementierungsvererbung*, leider nicht zu. Bei der Implementierungsvererbung erbt die abgeleitete Klasse alle Elemente der Basisklasse inklusive deren Implementierung (also quasi des Programmcodes der Methoden). Das ist der eigentliche Sinn der Vererbung: Eine abgeleitete Klasse erbt alle Elemente der Basisklasse und braucht diese nicht neu zu programmieren. Visual Basic 6 kennt nur die so genannte *Schnittstellenvererbung*. Bei dieser Art der Vererbung erbt die abgeleitete Klasse zwar alle Eigenschaften und alle Methoden der Basisklasse, aber leider nicht die Implementierung der Methoden. Die Schnittstellenvererbung hat eigentlich gar nichts mit der in diesem Abschnitt beschriebenen Grundbedeutung der Vererbung zu tun und wird in Visual Basic 6 dazu genutzt, Polymorphismus zu realisieren. Polymorphismus beschreibe ich im Abschnitt 3.2, Schnittstellen in Abschnitt 4.4. VBScript kennt übrigens überhaupt keine Vererbung.

Von einer Klasse zu erben ist zunächst einmal ziemlich einfach. Sie geben die Basisklasse einfach bei der Deklaration Ihrer neuen Klasse mit an. Das folgende Beispiel zeigt, wie Sie das in Object Pascal machen:

```
type TKonto = class
  public
    Nummer: integer;
    Stand: double;
    procedure Abheben(Betrag: double);
    procedure Einzahlen(Betrag: double);
end;

type TGirokonto = class(TKonto);
  { ... }
end;
```

Sie geben also den Namen der Basisklasse einfach in Klammern hinter dem `class`-Schlüsselwort an. Damit erbt die neue Klasse automatisch alle Elemente der Basisklasse.

> Vererbung wird oft auch als *Ableitung* bezeichnet: Eine Klasse wird von einer Basisklasse abgeleitet. Eine solche Basisklasse wird oft auch als *Superklasse* bezeichnet. Die abgeleitete Klasse nennt man auch *Subklasse*.

Auch wenn in der abgeleiteten Klasse gar keinen neuen Elemente deklariert sind, können Sie daraus bereits Objekte erzeugen und verwenden:

```
procedure Teste_Konto;
  var k1: TGiroKonto;
begin
  k1 := TGiroKonto.Create;
  k1.Einzahlen(1000);
  ShowMessage(FloatToStr(k1.Stand));
end;
```

Das macht natürlich nicht besonders viel Sinn. Die neue Klasse sollte schon mit einer neuen Funktionalität versehen werden. Dazu können Sie einfach neue Eigenschaften und Methoden hinzufügen:

```
type TGirokonto = class(TKonto)
  public
    Dispobetrag: double;
    procedure Ueberweisen(Zielkonto: TGirokonto);
end;
```

Das Girokonto verfügt nun über die (teilweise geerbten) Eigenschaften *Nummer*, *Stand* und *Dispobetrag* und über die (teilweise ebenfalls geerbten) Methoden *Abheben*, *Einzahlen* und *Ueberweisen*. Zur besseren Übersicht folgt der komplette Quellcode für die beiden Konto-Klassen:

```
unit Konten_Klassen;

interface

type TKonto = class
  public
    Nummer: integer;
    Stand: double;
    procedure Abheben(Betrag: double);
    procedure Einzahlen(Betrag: double);
end;

type TGirokonto = class(TKonto)
  public
    Dispobetrag: double;
    procedure Ueberweisen(Zielkonto: TGirokonto; Betrag: double);
end;

implementation
```

```
procedure TGiroKonto.Ueberweisen(Zielkonto: TGirokonto; Betrag: double);
begin
  Zielkonto.Einzahlen(Betrag);
  self.Abheben(Betrag);
end;

procedure TKonto.Abheben(Betrag: double);
begin
  Stand := Stand - Betrag;
end;



procedure TKonto.Einzahlen(Betrag: double);
begin
  Stand := Stand + Betrag;
end;

end.
```

Wenn Sie ein Objekt der Klasse *TGirokonto* erzeugen, können Sie nun alle Elemente verwenden:

```
{ Erzeugen von Konto-Objekten }
k1 := TGiroKonto.Create();
k2 := TGiroKonto.Create();

{ Setzen der Eigenschaften }
k1.Nummer := 1001;
k2.Nummer := 1002;

{ Einzahlen }
k1.Einzahlen(1000);

{ Abheben }
k2.Abheben(200);

{ Von Konto 1 nach Konto 2 überweisen }
k1.Ueberweisen(k2, 300);

{ Informationen über die Konten ausgeben }
writeln('Der Stand von Konto Nummer ' + IntToStr(k1.Nummer) +
  ' ist ' + FloatToStr(k1.Stand) + ' Euro');

writeln('Der Stand von Konto Nummer ' + IntToStr(k2.Nummer) +
  ' ist ' + FloatToStr(k2.Stand) + ' Euro');

readln;

{ Objekte freigeben }
k1.Free;
k2.Free;
```

3.1.4 Vererbung in den verschiedenen Sprachen

C++

In C++ geben Sie die Basisklasse (Superklasse) mit einem Doppelpunkt getrennt hinter dem Klassennamen der abgeleiteten Klasse (Subklasse) an. C++ unterstützt mehrere Vererbungsarten, die ich in diesem Artikel allerdings nicht erläutern kann. Die `public`-Ableitung ist die gebräuchlichste und entspricht der Vererbung in anderen Sprachen. Sie müssen also das Schlüsselwort `public` noch mit angeben, damit die Klasse öffentlich wird:

```
Public Class Subklasse: public Superklasse
```

C#

In C# geben Sie, ähnlich C++, die Basisklasse mit einem Doppelpunkt getrennt hinter dem Klassennamen an:

```
Public Class Subklasse: Superklasse
```

Object Pascal (Delphi/Kylix)

Die Vererbung in Object Pascal wurde bereits auf den vorhergehenden Seiten beschrieben. Hier noch einmal die Syntax:

```
Type Subklasse = class(Superklasse);
```

Java

In Java verwenden Sie das Schlüsselwort `extends`:

```
public class Subklasse extends Superklasse
```

Visual Basic 6

Visual Basic 6 kennt keine echte Vererbung (nur die Schnittstellenvererbung, Schnittstellen werden in Abschnitt 4.4 beschrieben).

Visual Basic .NET

Visual Basic .NET arbeitet mit dem Schlüsselwort `inherits`:

```
Public Class Subklasse
  Inherits Superklasse
```

3.1.5 Verwenden von geerbten Methoden

In einer abgeleiteten Klasse können Sie alle geerbten Eigenschaften und Methoden der Basisklasse verwenden. Wenn die abgeleitete Klasse allerdings eine Methode definiert, die genauso heißt wie eine Methode der Basisklasse (das ist das so genannte Überschreiben von Methoden, das ich im Abschnitt 3.1.8 beschreibe), müssen Sie dem Compiler mitteilen, welche Version der Methode Sie aufrufen wollen. Schreiben Sie nur den Namen der Methode, verwenden Sie die Methode der abgeleiteten Klasse. Zum Aufruf einer geerbten Methode stellt Ihnen jede Programmiersprache (außer Visual Basic 6) ein spezielles Schlüsselwort oder einen Operator zur Verfügung.

C++

In C++ rufen Sie Methoden der Basisklasse auf, indem Sie den Namen der Basisklasse gefolgt von einem Doppelpunkt und dem Methodennamen schreiben:

```
Superklasse::Methodenname(Argumente);
```

Diese Vorgehensweise macht Sinn, da C++ die Mehrfachvererbung erlaubt und eine Subklasse somit von mehreren Superklassen abgeleitet werden kann.

C#

In C# verwenden Sie den `base`-Operator:

```
base.Methodenname(Argumente);
```

Java

In Java verwenden Sie den `super`-Operator:

```
super.MethodenName(Argumente);
```

Object Pascal (Delphi/Kylix)

In Object Pascal rufen Sie Methoden der Basisklasse mit der `inherited`-Anweisung auf:

```
inherited Methodenname(Argumente);
```

Visual Basic .NET

In Visual Basic .NET verwenden Sie den `MyBase`-Operator:

```
MyBase.Methodenname(Argumente);
```

Visual Basic 6

Visual Basic 6 erlaubt keine echte Vererbung, kennt also keine Möglichkeit, geerbte Methoden aufzurufen.

3.1.6 Die Sichtbarkeit bei der Vererbung

In Kapitel 2 wurde bereits die Sichtbarkeit von Klassenelementen für die Bereiche *Private* (privat) und *Public* (öffentlich) besprochen. Private Klassenelemente sind nur innerhalb der Methoden der Klasse sichtbar, öffentliche Klassenelemente können auch von außen aufgerufen bzw. verwendet werden.

Öffentliche Klassenelemente werden grundsätzlich immer in eine abgeleitete Klasse vererbt. In vielen Abhandlungen über die OOP liest man nun, dass private Klassenelemente nicht an abgeleitete Klassen vererbt werden. Das ist eigentlich falsch. Private Klassenelemente werden wohl vererbt, sind in den abgeleiteten Klassen aber nicht sichtbar. Die Methoden, die in der Basisklasse mit den privaten Klassenelementen arbeiten, müssen ja auch in der abgeleiteten Klasse mit den privaten Elementen arbeiten können. Lediglich Methoden, die der abgeleiteten Klasse hinzugefügt oder in dieser überschrieben werden, haben keinen Zugriff auf die geerbten privaten Elemente. In manchen Fällen kann das zum Problem werden, nämlich dann, wenn die neuen Methoden eben doch Zugriff auf die privaten Elemente benötigen. Für die Lösung dieses Problems stellen Ihnen die meisten Programmiersprachen einen weiteren Sichtbarkeitsbereich, den Protected[12]-Bereich zur Verfügung. Eigenschaften und Methoden, die mit dieser Sichtbarkeit ausgestattet werden, verhalten sich nach außen (für Objekte dieser Klasse) wie Private-Elemente, werden aber so vererbt, dass alle Methoden der abgeleiteten Klasse darauf zugreifen können. Das folgende Beispiel verdeutlicht den Umgang mit Protected-Elementen: In der Kontenverwaltung soll der Zugriff auf die Kontonummer und den Kontostand nur über Methoden des Objekts möglich sein, damit eine möglichst große Datensicherheit gewährleistet ist. In einem ersten Versuch werden diese Eigenschaften deshalb im Private-Bereich angelegt:

```
{ Die Basisklasse Konto }
class TKonto = class
  private
    Nummer: integer;
    Stand: double;

  public
    { Der Konstruktor }
    constructor Create(Nummer: integer: Stand: double);

    { Methode zum Einzahlen }
    procedure Einzahlen(Betrag: double);

    { Methode zum Abheben }
    procedure Abheben(Betrag: double);

    { Methode zum Lesen des Kontostands }
    function getStand: double;
end;
```

In der Konto-Klasse nimmt die Abheben-Methode keine Überprüfung vor. In der abgeleiteten Girokonto-Klasse soll die Abheben-Methode mit einer neuen Funktionalität überschrieben werden (das Überschreiben von Methoden beschreibe ich eigentlich erst im nächsten Abschnitt, das Beispiel kommt aber ohne diese Technik nicht aus). Hier soll diese Methode überprüfen, ob durch das Abheben der Dispobetrag unterschritten wird. Die Abheben-Methode muss auf jeden Fall auf die Eigenschaft *Stand* zurückgreifen.

```
{ Die abgeleitete Klasse Girokonto }
class TGirokonto = class
  public
    { Die Methode zum Abheben wird überschrieben }
    procedure Abheben(Betrag: double);
end;
```

Für die Implementierung stelle ich zur besseren Übersicht nur die Abheben-Methode dar:

[12] Protected: englischer Begriff für „geschützt“

```
{ Methode zum Abheben bei einem einfachen Konto }
procedure TKonto.Abheben(Betrag: double);
begin
  Stand := Stand - Betrag
end;

...

{ Methode zum Abheben bei einem Girokonto }
procedure TGirokonto.Abheben(Betrag: double);
begin
  { Überprüfen, ob durch das Abheben der Kontostand
    den Dispobetrag nicht unterschreitet. Die
    Methode greift auf die geerbte Eigenschaft
    Stand zu. Dieser Zugriff führt allerdings zu einem
    Fehler beim Kompilieren, weil die Eigenschaft
    Stand in dieser Klasse nicht sichtbar ist }
  if Stand - Betrag >= Dispobetrag then
     Stand := Stand - Betrag
  else
     { Der Kontostand würde den Dispobetrag unterschreiten,
       was durch eine Exception verhindert wird }
     raise Exception.Create('Abheben nicht möglich. ' +
        'Das Limit würde unterschritten werden.');
end;
```

Wenn Sie versuchen würden, diesen Quellcode zu kompilieren, würde der Compiler den Fehler melden, dass der Bezeichner *Stand* unbekannt ist.

Die Lösung für dieses Problem ist, der Eigenschaft *Stand* in der Basisklasse den Protected-Sichtbarkeitsbereich zu geben:

```
{ Die korrigierte Basisklasse Konto }
class TKonto = class
  private
    Nummer: integer;

  protected
    Stand: double;

  public
     ...
```

Damit kann ein Programm, das ein Objekt dieser oder einer abgeleiteten Klasse benutzt, wie gewünscht nicht auf die Eigenschaft zugreifen, weil diese sich nach außen wie eine private Eigenschaft verhält. Die Eigenschaft wird aber nun so in die abgeleitete Klasse *TGirokonto* vererbt, dass deren Methoden darauf zugreifen können.

Die Protected-Sichtbarkeit gilt nicht nur für Eigenschaften, Sie können damit auch eigentlich private Methoden für die Benutzung in abgeleiteten Klassen freigegeben. Ich will dafür nicht noch ein Beispiel anbringen. Ich denke, Sie erkennen bei der Programmierung selbst, wann Sie Eigenschaften oder Methoden, die eigentlich privat sein sollten, für die Sichtbarkeit Ihrer abgeleiteten Klassen freischalten müssen. Dazu ein Tipp: Deklarieren Sie zunächst alle privaten Elemente einer Basisklasse auch mit dem Private-Sichtbarkeitsbereich. Wenn Sie in einer abgeleiteten Klasse vom Compiler den Fehler gemeldet bekommen, dass er einen Bezeichner, der eigentlich eine Eigenschaft oder Methode der Basisklasse kennzeichnet, nicht erkennt, deklarieren Sie das betreffende Element mit der Protected-Sichtbarkeit.

3.1.7 Veröffentlichen von geschützten Elementen in abgeleiteten Klassen

Geschützte (Protected-)Elemente können Sie in abgeleiteten Klassen auch veröffentlichen, das heißt in den Public-Bereich aufnehmen. Das macht manchmal – aber eigentlich nur sehr selten – Sinn, wenn in einer abgeleiteten Klasse eigentlich private Elemente doch von außerhalb benutzt werden sollen. Ein sinnvolles Beispiel dazu fällt mir jetzt nicht ein (aber dieses Feature wird eben auch sehr selten genutzt ...☺). Also nehmen wir an, eine abgeleitete Klasse *TSpielkonto* soll den Zugriff auf den Kontostand direkt über die Eigenschaft *Stand* nach außen freigegeben:

```
{ Die Basisklasse TKonto }
class TKonto = class
  private
    Nummer: integer;

  protected
    Stand: double;

  public
    ...

{ Die abgeleitete Klasse TSpielkonto }
type TSpielkonto = class(TKonto)
  public
   { Veröffentlichen der Protected-Eigenschaft Stand }
   Stand: double;
end;
```

Ein Objekt der Klasse *TSpielkonto* kann nun direkt auf die Eigenschaft *Stand* zugreifen:

```
{ Ein Spielkonto erzeugen }
k3 := TSpielkonto.Create(1003, 0);

{ und direkt auf die im Spielkonto veröffentlichte
  Eigenschaft Stand zugreifen }
k3.Stand := 1000;

...
```

> Dieses Verhalten müssen Sie natürlich berücksichtigen, wenn Sie Klassen entwickeln, die von anderen Programmierern verwendet werden. Enthalten diese Klassen Protected-Elemente, kann der andere Programmierer diese Elemente sehr einfach veröffentlichen. Die Kapselung, die in der Basisklasse für diese Elemente noch enthalten war, geht damit verloren.

3.1.8 Überschreiben von geerbten Methoden

Wie ich es bereits in den vorherigen Abschnitten angedeutet habe, können Sie eine abgeleitete Klasse nicht nur mit *neuen* Eigenschaften und Methoden *erweitern*, sondern auch *geerbte* Methoden mit einer neuen Funktionalität *überschreiben*. Mit dem Überschreiben von Methoden erreichen Sie, dass ein Objekt einer abgeleiteten Klasse sich beim Aufruf dieser Methoden anders *verhält* als ein Objekt einer Basisklasse. Wenn Sie *mehrere* Klassen von einer Basisklasse ableiten, können Sie damit erreichen, dass die Objekte dieser Klassen zwar prinzipiell gleich verwendet werden, sich aber eben auf eine unterschiedliche Weise verhalten. Ein Beispiel: In einer Autorennen-Simulation soll es möglich sein, verschiedene Autotypen (Ford Puma, Porsche 911, Ferrari Testarossa) zu verwenden. Alle Autos haben die Methoden *Starten*, *Beschleunigen*, *Bremsen* und *Lenken* gemeinsam. Diese Methoden werden deshalb in einer Basisklasse *Auto* implementiert. Von dieser Basisklasse werden drei Klassen für die verschiedenen Autotypen abgeleitet. Nun beschleunigt ein Ferrari Testarossa wesentlich besser

als ein Porsche und dieser (ein wenig[13]) besser als ein Ford Puma. Die Methode *Beschleunigen* muss also in den einzelnen Klassen mit einer neuen Funktionalität versehen werden. Dazu wird diese Methode in den abgeleiteten Klassen einfach überschrieben.

In unserer Kontenverwaltung kann das Überschreiben ebenfalls benutzt werden: Die Methode *Abheben* soll ja in einem Sparkonto ein Abheben nur dann zulassen, wenn der Kontostand dadurch nicht negativ wird, in einem Girokonto soll ein Abheben allerdings bis zum eingestellten Dispobetrag möglich sein. Beide Konten müssen die Methode dazu überschreiben, um die neue Funktionalität zu ermöglichen.

Das folgende Beispiel habe ich, zur besseren Übersicht, in einzelne Units unterteilt. Jede Unit definiert eine Klasse. Die Unit *Konto_Class* enthält die Basisklasse *TKonto*:

```
unit Konto_Class;

interface

{ Die Basisklasse Konto }
type TKonto = class
 private
   Nummer: integer;

 protected
   Stand: double;

 public
   { Der Konstruktor }
   constructor Create(Nummer: integer; Stand: double);

   { Methode zum Einzahlen }
   procedure Einzahlen(Betrag: double);

   { Methode zum Abheben }
   procedure Abheben(Betrag: double);

   { Methode zum Lesen der Kontonummer }
   function getNummer: integer;

   { Methode zum Lesen des Kontostands }
   function getStand: double;
end;

implementation

{ Der Konstruktor }
constructor TKonto.Create(Nummer: integer; Stand: double);
begin
  inherited Create;
  self.Nummer := Nummer;
  self.Stand := Stand;
end;

{ Methode zum Einzahlen }
procedure TKonto.Einzahlen(Betrag: double);
begin
  Stand := Stand + Betrag;
end;

{ Methode zum Abheben }
procedure TKonto.Abheben(Betrag: double);
begin
  Stand := Stand - Betrag
end;
```

[13] Ich konnte mir bisher keinen Porsche leisten. Deswegen die leicht optimistische Einschätzung der Beschleunigungsleistung eines Ford Puma. Aber dieses Buch wird diesen Missstand ja beseitigen. In einer der nächsten Auflagen wird der Porsche dann wohl wesentlich besser beschleunigen ...

```
{ Methode zum Lesen der Kontonummer }
function TKonto.getNummer: integer;
begin
  result := Nummer;
end;

{ Methode zum Lesen des Kontostands }
function TKonto.getStand: double;
begin
  result := Stand;
end;

end.
```

Die Unit *Girokonto_Class* definiert ein Girokonto. Die Klasse *TGirokonto* überschreibt die Abheben-Methode:

```
unit Girokonto_Class;

interface

{ Die Unit mit der Basisklasse wird eingebunden,
  damit diese hier bekannt ist }
uses SysUtils, Konto_Class

{ Die Klasse TGirokonto wird von der Klasse TKonto abgeleitet }
type TGirokonto = class(TKonto)
  public
    { Die neue Eigenschaft für den Dispobetrag }
    Dispobetrag: double;

    { Die Methode Abheben wird mit einer neuen
      Funktionalität überschrieben }
    procedure Abheben(Betrag: double);
end;

implementation

procedure TGirokonto.Abheben(Betrag: double);
begin
  { Überprüfen, ob durch das Abheben der Kontostand
    den Dispobetrag nicht unterschreitet }
  if Stand - Betrag >= Dispobetrag then
     Stand := Stand - Betrag
  else
     { Der Kontostand würde den Dispobetrag unterschreiten,
       was durch eine Exception verhindert wird }
     raise Exception.Create('Abheben nicht möglich. ' +
        'Das Limit würde unterschritten werden.');
end;

end.
```

Eine andere Möglichkeit, die Abheben-Methode in der abgeleiteten Klasse zu implementieren, wäre, die geerbte Methode aufzurufen um den Kontostand zu reduzieren (womit die Funktionalität der Basisklasse genutzt wird). Wie bereits beschrieben habe, können Sie in abgeleiteten Klassen ja auch die geerbten Methoden aufrufen:

```
procedure TGirokonto.Abheben(Betrag: double);
begin
  { Überprüfen, ob durch das Abheben der Kontostand
    den Dispobetrag nicht unterschreitet }
  if Stand - Betrag >= Dispobetrag then
     inherited Abheben(Betrag);
  else

  ...
```

In unserem Beispiel funktioniert dies, da die Basisklasse *TKonto* beim Abheben nichts weiter überprüft. Sie müssen beim Aufrufen von geerbten Methoden jedoch vorsichtig sein:

Wenn Sie in überschriebenen Methoden geerbte Methoden aufrufen, müssen Sie absolut sicher sein, dass die geerbte Methode unter den veränderten Umständen der überschriebenen Methode auch funktioniert. Würde die Abheben-Methode der *TKonto*-Klasse unseres Beispiels überprüfen, ob der Kontostand beim Abheben negativ wird, und in diesem Fall eine Exception auslösen, würde der Aufruf der geerbten Abheben-Methode in der abgeleiteten *TGirokonto*-Klasse nicht funktionieren.

3.1.9 Vererbung und Überschreiben von Konstruktoren und Destruktoren

Bei den im Artikel behandelten Sprachen gibt es eigentlich nur eine Sprache, die Konstruktoren und Destruktoren wirklich an abgeleitete Klassen vererbt: Object Pascal. Der Standardkonstruktor und -destruktor ist in Object Pascal in der Basisklasse `TObject` definiert und wird damit automatisch an alle Klassen vererbt, da in Object Pascal jede Klasse mindestens von `TObject` abgeleitet ist. Jede Klasse enthält damit also automatisch einen Konstruktor und einen Destruktor. Enthält eine Klasse mehrere (überladene) Konstruktoren, müssen Sie aber etwas aufpassen, damit eine abgeleitete Klasse auch alle diese Konstruktoren erbt. Wenn Sie in der abgeleiteten Klasse keinen neuen Konstruktor erzeugen, müssen Sie nichts weiter beachten. Soll die abgeleitete Klasse aber neben den geerbten Konstruktoren noch einen oder mehrere eigene besitzen, müssen Sie diese Konstruktoren mit dem Schlüsselwort `overload` kennzeichnen. Vergessen Sie dies, versteckt Object Pascal die geerbten Konstruktoren so, dass diese nicht mehr von außen benutzt werden können:

```
type A = class
  public
    { Zwei überladene Konstruktoren in der Basisklasse }
    constructor Create(); overload;
    constructor Create(s: string); overload;
end;

type B = class(A)
  public
    { Ein neuer Konstruktor in der abgeleiteten Klasse.
      Dieser muss mit overload gekennzeichnet werden,
      damit der geerbte zweite Konstruktor in Objekten
      der Klasse verfügbar ist }
    constructor Create; overload;
end;
```

In Object Pascal sollten Sie einen geerbten Konstruktor in einem neuen Konstruktor einer abgeleiteten Klasse immer an erster Stelle aufrufen. Vergessen Sie dies, wird das Objekt zwar erstellt, der Konstruktor der Basisklasse jedoch nicht aufgerufen. Das kann zu üblen Fehlern führen, nämlich dann, wenn der Konstruktor der Basisklasse irgendwelche wichtigen Arbeiten erledigt, die dann natürlich nicht ausgeführt werden. Wenn Sie den geerbten Konstruktor immer an erster Stelle in einem neuen Konstruktor aufrufen, führt das dazu, dass auch in großen Vererbungshierarchien die Konstruktoren der Basisklassen auf jeden Fall in der richtigen Reihenfolge aufgerufen werden.

Einen geerbten Destruktor sollten Sie in Object Pascal immer an letzter Stelle eines Destruktors einer abgeleiteten Klasse aufrufen. Damit erreichen Sie, dass der Destruktor einer Basisklasse, der ja eventuell wichtige Aufräumarbeiten erledigt, auf jeden Fall aufgerufen wird, wenn das Objekt zerstört wird.

C++, C#, Java und Visual Basic .NET vererben Konstruktoren und Destruktoren nicht so, dass diese von außen verwendet werden können (ähnlich privaten Methoden und Eigenschaften). Wenn Sie in diesen Sprachen in einer Klasse keinen Konstruktor oder Destruktor einfügen, erledigt das der Compiler für Sie. Er erzeugt einen neuen Standardkonstruktor beziehungsweise Standarddestruktor in die Klasse. Wenn Sie in einer Basisklasse mehrere Konstruktoren

deklariert haben, sind diese in den Objekten einer abgeleiteten Klasse nicht verfügbar. Die abgeleitete Klasse besitzt nur den Standardkonstruktor beziehungsweise die Konstruktoren, die Sie eigens für die Klasse deklariert haben. Das folgende Beispiel demonstriert dieses Verhalten anhand eines Java-Quellcodes:

```
class K_D_Vererbung_Fehler
{
   public static void main(String[] args)
   {
      System.out.println("Beispiel für das Vererben von " +
          "Konstruktoren und Destruktoren");

      System.out.println("");
      BasisKlasse o1 = new BasisKlasse(); // OK
      BasisKlasse o2 = new BasisKlasse(123); // OK
      AbgeleiteteKlasse o3 = new AbgeleiteteKlasse(); // OK

      /* Versuch, ein Objekt der abgeleiteten Klasse
         zu erzeugen, wobei fälschlicherweise angenommen wird,
         dass der Konstruktor vererbt wird. Der Compiler meldet
         an dieser Stelle den Fehler, dass er das Symbol
         'AbgeleiteteKlasse(123)' nicht auflösen kann
         ("Cannot resolve symbol") */
      AbgeleiteteKlasse o4 = new AbgeleiteteKlasse(123);
   }
}

class BasisKlasse
{
   /* Der erste Konstruktor */
   public BasisKlasse()
   {
      System.out.println(" - Der erste Konstruktor der " +
         "Basisklasse wird aufgerufen.");
   }

   /* Der zweite Konstruktor */
   public BasisKlasse(int i)
   {
      System.out.println(" - Der zweite Konstruktor der " +
         "Basisklasse wird aufgerufen.");
   }
}

class AbgeleiteteKlasse extends BasisKlasse
{
   /* die abgeleitete Klasse definiert keinen eigenen Konstruktor */
}
```

Ein Konstruktor kann in diesen Klassen jedoch die Konstruktoren der Basisklasse aufrufen. Wenn Sie selbst keinen Aufruf eines Basisklassen-Konstruktors einfügen, erledigt das ebenfalls der Compiler ganz automatisch. Er ruft dann allerdings den parameterlosen Standardkonstruktor auf (den ja jede Klasse besitzt). Sie können sich also darauf verlassen, dass zumindest alle Standardkonstruktoren aller Basisklassen auch in größeren Vererbungshierarchien aufgerufen werden.

Destruktoren werden in allen im Artikel behandelten Sprachen, außer Object Pascal, ebenfalls nicht vererbt. Das ist aber nicht besonders schlimm, weil Destruktoren ja sowieso nicht überladen werden können. In Object Pascal müssen Sie den geerbten Destruktor immer an letzter Stelle eines Destruktors in einer abgeleiteten Klasse aufrufen, um sicherzustellen, dass alle Destruktoren von Basisklassen auch aufgerufen werden. In den anderen Sprachen müssen Sie den Destruktor der Basisklasse nicht unbedingt aufrufen, der Compiler erledigt dies automatisch für Sie. Die korrekte Aufrufreihenfolge von Destruktoren ist damit sichergestellt.

In den Beispielen zu diesem Artikel finden Sie im Ordner *Konstruktor und Destruktor* den Ordner *Vererbung beim Konstruktor und Destruktor* mit Demonstrationen für die Vererbung von Konstruktoren und Destruktoren für die einzelnen Sprachen.

3.1.10 Konstruktoren und Destruktoren bei der Vererbung in den verschiedenen Sprachen

C#

In C# werden Konstruktoren und Destruktoren zwar vererbt, aber nicht so, dass diese in abgeleiteten Klassen nach außen sichtbar sind. Im Prinzip verhält sich ein Konstruktor bzw. Destruktor bei der Vererbung wie ein geschütztes (protected) Klassenelement. Sie müssen also alle Konstruktoren der Basisklasse neu implementieren, wenn Sie diese in der abgeleiteten Klasse zur Verfügung haben wollen.

Enthält ein Konstruktor keinen Aufruf eines geerbten Konstruktors, fügt der Compiler automatisch einen Aufruf des parameterlosen Standardkonstruktors als erste Anweisung in den Programmcode ein. Wenn Sie selbst einen geerbten Konstruktor aufrufen wollen, verwenden Sie den `base`-Operator gefolgt von der eventuellen Argumentliste. Diese Anweisung muss allerdings direkt hinter dem Kopf des Konstruktors, durch einen Doppelpunkt getrennt angegeben werden:

```
class BasisKlasse
{
    /* Konstruktor */
    public BasisKlasse(string Info)
    {
       System.Console.WriteLine(" - Konstruktor der Basisklasse " +
          "wird mit Argument " + Info + " aufgerufen.");
    }

    /* Destruktor */
    ~BasisKlasse()
    {
      System.Console.WriteLine(" - Destruktor der Basisklasse " +
         "wird aufgerufen.");
    }
}

class AbgeleiteteKlasse: BasisKlasse
{
  /* Konstruktor mit Aufruf des geerbten Konstruktors */
  public AbgeleiteteKlasse(string Info):base(Info)
  {
     System.Console.WriteLine(" - Konstruktor der abgeleiteten " +
        "Klasse wird mit Argument " + Info + " aufgerufen.");
  }

  /* Destruktor */
  ~AbgeleiteteKlasse()
  {
     System.Console.WriteLine(" - Destruktor der abgeleiteten " +
        "Klasse wird aufgerufen.");
  }
}
```

Im Destruktor richtet der Compiler ebenfalls automatisch einen Aufruf des geerbten Destruktors als letzte Anweisung ein. Damit können Sie sich darauf verlassen, dass auch in größeren

Vererbungshierarchien alle Konstruktoren und Destruktoren der Basisklassen in der korrekten Reihenfolge aufgerufen werden.

Enthält eine Klasse keinen Konstruktor bzw. Destruktor fügt der Compiler automatisch einen parameterlosen Standardkonstruktor bzw. einen Destruktor in die Klasse ein. In diesen erzeugt der Compiler – im Gegensatz zum sonst ähnlichen C++ – automatisch einen Aufruf des geerbten Standardkonstruktors bzw. des geerbten Destruktors.

C++

C++ verhält sich bei der Vererbung von auf Konstruktoren und Destruktoren prinzipiell – bis auf den unten beschriebenen Unterschied – wie C#. Deshalb folgt hier zunächst nur der Quellcode des Beispiels. Beachten Sie, dass C++-Konstruktoren immer virtuell sein sollten (siehe Abschnitt 2.5.3):

```
class BasisKlasse
{
  public:
    /* Konstruktor */
    BasisKlasse(string Info)
    {
       cout << " - Konstruktor der Basisklasse wird mit Argument "
         << Info << " aufgerufen." << endl;
    }

    /* Destruktor */
    virtual ~BasisKlasse()
    {
       cout << " - Destruktor der Basisklasse wird aufgerufen." << endl;
    }
};

class AbgeleiteteKlasse: public BasisKlasse
{
  public:
    /* Konstruktor mit Aufruf des geerbten Basisklassen-Konstruktors */
    AbgeleiteteKlasse(string Info):BasisKlasse(Info)
    {
       cout << " - Konstruktor der abgeleiteten Klasse wird mit " <<
         "Argument " << Info << " aufgerufen." << endl;
    }

    /* Destruktor */
    virtual ~AbgeleiteteKlasse()
    {
       cout << " - Destruktor der abgeleiteten Klasse wird aufgerufen."
         << endl;
    }
};
```

Einen Unterschied zu C# sollten Sie beachten:

> Enthält eine Klasse keinen Konstruktor bzw. Destruktor fügt der Compiler wie bei C# automatisch einen parameterlosen Standardkonstruktor und einen Destruktor in die Klasse ein. In diesen erzeugt der Compiler aber keinen Aufruf des geerbten Standardkonstruktors bzw. des geerbten Destruktors. Wenn Sie also keine eigenen Konstruktoren und Destruktoren erzeugen, werden geerbte Konstruktoren/Destruktoren nicht aufgerufen.

Java

In Java werden Konstruktoren und Destruktoren (wie bei den anderen Sprachen außer Object Pascal) zwar vererbt, aber nicht so, dass diese in abgeleiteten Klassen nach außen sichtbar sind. Sie müssen also alle Konstruktoren der Basisklasse neu implementieren wenn Sie diese in der abgeleiteten Klasse zur Verfügung haben wollen.

Enthält ein Konstruktor keinen Aufruf eines geerbten Konstruktors, fügt der Compiler automatisch einen Aufruf des parameterlosen Standardkonstruktors als erste Anweisung in den Programmcode ein. Wenn Sie selbst einen geerbten Konstruktor aufrufen wollen, verwenden Sie den `super`-Operator gefolgt von der eventuellen Argumentliste. Diese Anweisung muss die erste Anweisung im Konstruktor sein.

> Wenn Sie einen eigenen Destruktor in eine abgeleitete Klasse einfügen, baut der Compiler bei Java keinen impliziten Aufruf des geerbten Destruktors in diesen Destruktor ein. Sie sollten den geerbten Destruktor also mit `super.finalize()` in der letzten Anweisung des Destruktors selbst aufrufen um abzusichern, dass die geerbten Destruktoren immer aufgerufen werden. Wenn Sie keinen eigenen Destruktor in Ihre Klasse einfügen, ruft der Compiler allerdings den geerbten Destruktor automatisch auf, wenn das Objekt zerstört wird.

```
class BasisKlasse
{
   /* Konstruktor */
   public BasisKlasse(String Info)
   {
      System.out.println(" - Konstruktor der Basisklasse wird mit " +
        "Argument " + Info + " aufgerufen.");
   }

   /* "Destruktor" */
   protected void finalize()
   {
      System.out.println(" - Destruktor der Basisklasse wird aufgerufen.");
   }
}

class AbgeleiteteKlasse extends BasisKlasse
{
   /* Konstruktor */
   public AbgeleiteteKlasse(String Info)
   {
      /* Aufruf des geerbten Konstruktors */
      super(Info);

      System.out.println(" - Konstruktor der abgeleiteten Klasse " +
        "wird mit Argument " + Info + " aufgerufen.");
   }

   /* "Destruktor" */
   protected void finalize()
   {
      System.out.println(
        " - Destruktor der abgeleiteten Klasse wird aufgerufen.");
```

```
      /* In Java-Destruktoren muss der geerbte Destruktor
       * explizit aufgerufen werden, der Compiler erledigt
       * dies nicht automatisch */
      super.finalize();
   }
}
```

Object Pascal (Delphi/Kylix)

Die Vererbung von Konstruktoren und Destruktoren für Object Pascal wurde bereits im Abschnitt 3.1.9 behandelt.

Visual Basic 6

Visual Basic 6 kennt keine echte Vererbung und keine Konstruktoren und Destruktoren.

Visual Basic .NET

In Visual Basic .NET werden Konstruktoren und Destruktoren wie in C# nur so vererbt, dass diese in abgeleiteten Klassen nach außen nicht sichtbar sind. Sie müssen also alle Konstruktoren der Basisklasse neu implementieren wenn Sie diese in der abgeleiteten Klasse zur Verfügung haben wollen.

Enthält ein Konstruktor keinen Aufruf eines geerbten Konstruktors, fügt der Compiler automatisch einen Aufruf des parameterlosen Standardkonstruktors als erste Anweisung in den Programmcode ein. Wenn Sie selbst einen geerbten Konstruktor aufrufen wollen, verwenden Sie den `MyBase`-Operator gefolgt von `.New` und der eventuellen Argumentliste. Diese Anweisung muss (wie bei den anderen Sprache auch) die erste Anweisung im Konstruktor sein.

> Wenn Sie einen eigenen Destruktor in eine abgeleitete Klasse einfügen, baut der Compiler (wie der Java-Compiler) keinen impliziten Aufruf des geerbten Destruktors in diesen Destruktor ein. Sie sollten den geerbten Destruktor also mit `MyBase.Finalize()` in der letzten Anweisung des Destruktors selbst aufrufen um abzusichern, dass die geerbten Destruktoren immer aufgerufen werden. Wenn Sie keinen eigenen Destruktor in Ihre Klasse einfügen, ruft der Compiler allerdings den geerbten Destruktor automatisch auf, wenn das Objekt zerstört wird.

```
Class BasisKlasse
  ' Konstruktor
  Public Sub New(Info As String)
    System.Console.WriteLine(" - Konstruktor der Basisklasse " & _
      "wird mit Argument " + Info + " aufgerufen.")
  End Sub

  ' Destruktor
  Protected Overrides Sub Finalize()
    System.Console.WriteLine(" - Destruktor der Basisklasse " & _
      "wird aufgerufen.")
  End Sub
End Class

Class AbgeleiteteKlasse
  Inherits BasisKlasse

  ' Konstruktor
  Public Sub New(Info As String)
    MyBase.New(Info)
    System.Console.WriteLine(" - Konstruktor der abgeleiteten  " & _
       "Klasse wird mit Argument " + Info + " aufgerufen.")
  End Sub

  ' Destruktor
```

```
  Protected Overrides Sub Finalize()
    System.Console.WriteLine(" - Destruktor der abgeleiteten " & _
       "Klasse wird aufgerufen.")

    ' VB ruft (wie Java) im Destruktor nicht den geerbten Destruktor
    ' automatisch auf, das müssen Sie selbst erledigen.
    MyBase.Finalize()
  End Sub
End Class
```

3.1.11 Methoden-Überladung

Die Programmiersprachen C++, C#, Java, Object Pascal (Delphi/Kylix) und Visual Basic .NET erlauben Ihnen, in einer Klasse mehrere Methoden mit demselben Namen, aber mit unterschiedlicher Signatur[14] zu definieren. Mit dieser *Überladung* von Methoden erhalten Sie eine einfache Möglichkeit, Methoden zu entwickeln, die auf unterschiedliche Weise aufgerufen werden können. Das folgende Beispiel verdeutlicht dies anhand einer Klasse zur Speicherung von XY-Koordinaten (Punkten):

```
unit Point_Class;

interface

type TPoint = class
  public
    X: integer;
    Y: integer;

    { Die Methode Add wird mehrfach überladen }
    procedure Add(X: Integer); overload;
    procedure Add(X, Y: Integer); overload;
    procedure Add(p: TPoint); overload;
end;

implementation

procedure TPoint.Add(X: Integer);
begin
  self.X := self.X + X;
end;

procedure TPoint.Add(X, Y: Integer);
begin
  self.X := self.X + X;
  self.Y := self.Y + Y;
end;

procedure TPoint.Add(p: TPoint);
begin
  self.X := self.X + p.X;
  self.Y := self.Y + p.Y;
end;
end.
```

In Object Pascal müssen Sie, wie Sie dem Beispiel entnehmen können, das `overload`-Schlüsselwort an die Deklaration der Methode anhängen, damit der Compiler erkennt, dass Sie diese überladen wollen. In den anderen genannten Sprachen ist dazu kein spezielles Schlüsselwort notwendig.

---

[14] Der Begriff „Signatur“ steht bei Funktionen, Prozeduren und Methoden für die Argumentliste und den Rückgabedatentyp derselben. Die Übergabearten (»By Value« oder »By Reference«) und die Datentypen der einzelnen Argumente und der Datentyp des Rückgabewerts (falls es sich um eine Funktion handelt) definieren also die Signatur.

Die überladenen Methoden besitzen denselben Namen, müssen sich aber in der Anzahl der Argumente oder dem Datentyp derselben unterscheiden. Beim Aufruf der Methode entscheidet der Compiler anhand der Argumentliste, welche Methode er aufrufen muss:

```
program Methoden_Ueberladung;

uses SysUtils,
  Point_Class in 'Point_Class.pas';

{$APPTYPE CONSOLE}

var p1, p2: TPoint;

begin
  writeln('Beispiel für Methoden-Überladung');

  { Erzeugen der Objekte }
  p1 := TPoint.Create;
  p2 := TPoint.Create;

  { Initialisieren der Objekte }
  p1.X := 10;
  p1.Y := 20;
  p2.X := 15;
  p2.Y := 25;

  { Aufruf der Add-Methode mit verschiedenen Signaturen }

  { Nur X wird übergebenen }
  p1.Add(10);

  { X und Y wird übergebenen }
  p1.Add(10, 15);

  { Ein anderer Punkt wird übergeben }
  p1.Add(p2);

  { Ausgabe der Koordinaten von p1 }
  writeln('p1: ' + IntToStr(p1.X) + ', ' + IntToStr(p1.Y));

  readln;

  { Objekte freigeben }
  p1.Free;
  p2.Free;
end.
```

Der Compiler muss die Chance haben, die aufzurufende Methode zu erkennen. Die einzelnen Methoden müssen sich deshalb entweder in der Anzahl der Argumente oder, wenn die Anzahl der Argumente zweier überladener Methoden gleich ist, in den Datentypen der Argumente unterscheiden.

Ist die Methode eine Funktion, reicht ein Ändern des Rückgabedatentyps nicht zur Überladung aus.

In C# und Visual Basic .NET werden Methoden anhand des Namens überladen und nicht, wie in anderen Sprachen, anhand der Signatur einer Methode. Wenn in C# und Visual Basic .NET in einer Basisklasse mehrere überladene Methoden implementiert sind und Sie erzeugen in einer abgeleiteten Klasse eine einzige neue gleichnamige Methode, werden alle überladenen Methoden der Basisklasse durch diese neue Methode verborgen. Mit einer Instanz der abgeleiteten Klasse können Sie nur noch Ihre neue Methode aufrufen. Innerhalb der Methoden der abgeleiteten Klasse können Sie die geerbten Methoden jedoch wie gewohnt noch über den `base`-Operator aufrufen.

3.1.12 Überladen von Konstruktoren

Programmiersprachen, die das Überladen von Methoden erlauben, ermöglichen Ihnen auch, Konstruktoren zu überladen. Damit erhalten Sie eine einfache Möglichkeit, das Initialisieren von Objekten möglichst flexibel zu gestalten. Ein Personobjekt könnte zum Beispiel nur mit dem Vornamen und dem Nachnamen der Person oder auch mit der Straße und dem Ort initialisiert werden:

```
unit Person_Class;
{ Beispiel für Konstruktor-Überladung }

interface

type TPerson = class
  public
    { Die Eigenschaften der Klasse }
    Vorname: string;
    Nachname: string;
    Strasse: String;
    Ort: string;

    { Der Konstruktor wird mehrfach überladen }
    constructor Create(Vorname, Nachname: String); overload;
    constructor Create(Vorname, Nachname, Ort: String); overload;
    constructor Create(Vorname, Nachname, Strasse, Ort: String); overload;
 end;

implementation

{ Der erste Konstruktor }
constructor TPerson.Create(Vorname, Nachname: String);
begin
  { Aufruf des Standardkonstruktors, damit dieser
    ein Objekt der Klasse erzeugt }
  inherited Create;

  { Setzen der übergebenen Eigenschaften }
  self.Vorname := Vorname;
  self.Nachname := Nachname;
  self.Ort := 'unbekannt';
  self.Strasse := 'unbekannt';
end;

{ Der zweite Konstruktor }
constructor TPerson.Create(Vorname, Nachname, Ort: String);
begin
  { Aufruf des Standardkonstruktors, damit dieser
    ein Objekt der Klasse erzeugt }
  inherited Create;

  { Setzen der übergebenen Eigenschaften }
  self.Vorname := Vorname;
```

```
  self.Nachname := Nachname;
  self.Ort := Ort;
  self.Strasse := 'unbekannt';
end;

{ Der dritte Konstruktor }
constructor TPerson.Create(Vorname, Nachname, Strasse, Ort: String);
begin
  { Aufruf des Standardkonstruktors, damit dieser
    ein Objekt der Klasse erzeugt }
  inherited Create;

  { Setzen der übergebenen Eigenschaften }
  self.Vorname := Vorname;
  self.Nachname := Nachname;
  self.Strasse := Strasse;
  self.Ort := Ort;
end;

end.
```

Bei der Erstellung eines Objekts dieser Klasse können Sie nun einen dieser Konstruktoren verwenden:

```
program Konstruktor_Ueberladung;

uses SysUtils,
  Person_Class in 'Person_Class.pas';

{$APPTYPE CONSOLE}

var p1, p2, p3: TPerson;

begin
  writeln('Beispiel fuer Konstruktor-Ueberladung');

  { Erzeugen der Objekte mit den verschiedenen
    Konstruktoren }
  p1 := TPerson.Create('Smilla', 'Jaspersen');
  p2 := TPerson.Create('Merril', 'Overturf', 'Wien');
  p3 := TPerson.Create('Donald', 'Duck', 'Eierweg 10', 'Entenhausen');

  ...
```

3.1.13 Abstrakte Basisklassen

Eine abstrakte Klasse ist eine Klasse, aus der entweder gar keine Objekte erzeugt werden können oder bei der das Erzeugen von Objekten keinen Sinn macht. Um die eventuelle Verwirrung direkt aufzulösen: Die Klasse *TKonto* in unserer Kontenverwaltung wird eigentlich nur als Basisklasse für die Klassen *TSparkonto* und *TGirokonto* verwendet. Aus dieser Klasse sollen keine Objekte erzeugt werden können. Diese Klasse sollte abstrakt sein.

Eine Klasse ist immer dann abstrakt, wenn mindestens eine Methode der Klasse nicht oder nicht sinnvoll verwendet werden kann. Diese in der Basisklasse sinnlosen Methoden werden dann einfach nicht implementiert. Es kann aber auch sein, dass eine abstrakte Klasse Methoden besitzt, die auch in abgeleiteten Klassen sinnvoll verwendet werden können, und die deswegen implementiert sind. In der Kontoklasse könnte es zum Beispiel eine Methode *getStand* geben, die den Kontostand zurückgibt, und eine Methode *Einzahlen*, über die Geld eingezahlt werden kann. Diese Methoden können sinnvoll in den abgeleiteten Klassen verwendet werden und werden deswegen schon in TKonto implementiert. Die *Abheben*-Methode wird allerdings in allen abgeleiteten Klassen neu implementiert und sollte deswegen abstrakt sein. Damit kann kein Programmierer ein Objekt der Klasse TKonto sinnvoll verwenden, was ja gewünscht ist.

Abstrakte Klassen können Sie im Prinzip in jeder Programmiersprache erzeugen. Manche Sprachen bieten Ihnen dazu spezielle Schlüsselworte an, bei anderen Sprachen können Sie

einfach die Implementierung einer abstrakt geplanten Methode weglassen oder in dieser eventuell sogar einfach einen Laufzeitfehler erzeugen, damit eine Verwendung eines Objekts dieser Klasse unmöglich ist.

In Object Pascal können Sie einzelne Methoden über das Schlüsselwort `abstract` zur abstrakten Methoden machen. Für diese Methoden muss sichergestellt werden, dass der Compiler auf jeden Fall immer die in der abgeleiteten Klassen überschriebene Methode aufruft, weswegen Sie zusätzlich noch das Schlüsselwort `virtual` angeben müssen. Die Bedeutung von *virtuellen Methoden* erkläre ich allerdings erst in Kapitel 4. Merken Sie sich hier einfach nur, dass Sie (auch in anderen Sprachen) abstrakte Methoden immer auch als virtuelle Methode kennzeichnen müssen (sofern die Programmiersprache nicht sowieso schon ausschließlich virtuelle Methoden verwendet, wie es bei den moderneren Sprachen der Fall ist).

```
{ Die Basisklasse Konto }
type TKonto = class
  private
    Nummer: integer;

  protected
    Stand: double;

  public
    { Die Konstruktoren }
    constructor Create(Nummer: integer); overload;
    constructor Create(Nummer: integer; Stand: double); overload;

    { Methode zum Einzahlen }
    procedure Einzahlen(Betrag: double);

    { Die Methode zum Abheben wird abstrakt deklariert
      und muss deswegen in dieser Klasse nicht
      implementiert werden, was ja auch nicht sinnvoll
      ist, da TKonto nur die Basisklasse für die
      anderen Konten ist }
    procedure Abheben(Betrag: double); virtual; abstract;

    { Methode zum Lesen der Kontonummer }
    function getNummer: integer;

    { Methode zum Lesen des Kontostands }
    function getStand: double;
end;
```

Die Abheben-Methode wird jetzt in dieser Klasse gar nicht mehr implementiert. In den abgeleiteten Klassen TGirokonto und TSparkonto wird sie wie gewohnt überschrieben.

In Object Pascal können Sie nun zwar noch ein Objekt dieser Klasse erzeugen, beim Versuch, die Abheben-Methode zur verwenden, erzeugt das Programm allerdings eine Exception:

```
k := TKonto.Create(1001); { das geht noch }
k.Abheben(1000); { das geht nicht mehr }
```

Besonders fein reagiert der Object Pascal-Compiler hier nicht, andere Compiler lassen das Kompilieren erst gar nicht zu, wenn versucht wird, abstrakte Klassen zu instanzieren.

3.2 Polymorphismus

Polymorphismus bedeutet übersetzt »Vielgestaltigkeit«. Damit ist gemeint, dass ein Objekt in mehreren Gestalten vorkommen kann. Eigentlich ist diese Aussage nicht ganz korrekt, denn ein Objekt gehört immer einer Klasse an und besitzt deswegen auch nur eine Gestalt, nämlich die, die durch die Klasse festgelegt wird. Aber eine Klasse kann ja, wie Sie bereits wissen, von einer Basisklasse abgeleitet werden und erbt damit alle Eigenschaften und Methoden dieser Klasse. In der abgeleiteten Klasse ist es möglich, geerbte Methoden mit einer neuen Programmierung zu überschreiben. Dabei darf allerdings nicht die Signatur der Methode (d. h. der Methodenkopf mit dem Namen, den Argumenten und dem eventuellen Rückgabewert der Methode) verändert werden. Die Programmierung innerhalb der Methode kann jedoch vollkommen anders ausgeführt sein als die der geerbten Methode. Polymorphismus bedeutet nun, dass ein Programmteil, der eine Referenz auf ein Objekt der Basisklasse besitzt, über diese Referenz auch ein Objekt einer davon abgeleiteten Klasse bearbeiten kann. Über die Vererbung besitzt ein Objekt der abgeleiteten Klasse ja auf jeden Fall die Eigenschaften und Methoden der Basisklasse, und zwar in genau derselben Form (d. h. mit derselben Signatur). Das Ganze ist am Anfang ziemlich schwierig zu verstehen, wenn Sie Polymorphismus allerdings einmal am praktischen Beispiel angewendet haben, ist dieser nicht mehr ganz so mysteriös.

Ein gutes Beispiel für Polymorphismus in der Praxis ist ein Spiel. Für ein objektorientiert programmiertes Schachspiel im Computer werden zum Beispiel mehrere Objekte benötigt, das Schachbrett und je ein Objekt für die einzelnen Figuren. Dem Schachbrett werden die Figuren in Form von zwei Objekt-Auflistungen zugeordnet, eine für die schwarzen und eine für die weißen Figuren. Um das Spiel einfach programmierbar zu machen, wird das Schachbrett über eine Methode *Ziehen* gesteuert, der als Argument die Position der Figur und die Zielposition übergeben wird. Es wäre jetzt genial, wenn das Schachbrett bei einem Zug der Figur auf der angegebenen Position einfach mitteilen würde, dass diese ihre Position zum Ziel hin wechseln soll. Das würde ganz einfach über eine Methode *GeheZu* der Figur geschehen, der die Zielposition übergeben wird. Die Figur überprüft, ob ihr dieser Zug überhaupt möglich ist (Sie wissen ja: Ein Turm darf zum Beispiel nur waagerecht ziehen, ein Springer nur diagonal usw.), und führt den Zug aus, wenn dies der Fall ist. Den Erfolg des Zuges kann die Methode *GeheZu* über einen boolschen Rückgabewert melden. Das Schachbrett kann bei einem erfolgreichen Zug überprüfen, ob auf der Zielposition eine gegnerische Figur steht, und diese dann aus dem Spiel entfernen.

Warum ich in diesem Beispiel die Figuren entscheiden lasse, ob der Zug möglich ist, und nicht das Schachspiel, erkläre ich erst später. Jetzt soll erst einmal geklärt werden, wie es dem Schachspiel möglich ist, die Methode *GeheZu* für ganz unterschiedliche Figuren aufzurufen. Hier kommt Polymorphismus ins Spiel: Für die Figuren wird eine Basisklasse erstellt, die die benötigte Methode *GeheZu* in abstrakter Form enthält. In dieser Klasse wird diese Methode noch gar nicht ausprogrammiert, weil ja erst die Figuren entscheiden können, ob der gewünschte Zug möglich ist. Die Klasse wird zusätzlich noch mit einer Eigenschaft ausgerüstet, der später eine Referenz auf das Schachbrett übergeben wird, damit die Figur erkennen kann, ob auf der Zielposition evtl. eine eigene andere Figur steht:

```
TFigur = class
  private
    Schachbrett: TSchachbrett;
    Position: string;

  public
    constructor Create(Schachbrett: TSchachbrett);
    function GeheZu(Ziel: string): boolean; virtual; abstract;
end;
```

Die Methode *GeheZu* ist abstrakt gekennzeichnet, was bedeutet, dass diese Methode erst in der abgeleiteten Klasse ausprogrammiert wird. In Object Pascal werden abstrakte Methoden mit dem Schlüsselwort `abstract` deklariert. Hinzu kommt, dass solche Methoden virtuell sein müssen, weswegen das Schlüsselwort `virtual` mit angegeben ist. Die Bedeutung von virtuellen Methoden erkläre ich in Kapitel 4.

Von dieser Klasse werden Klassen für die einzelnen Figurentypen abgeleitet:

```
TTurm = class(TFigur)
  public
    function GeheZu(Ziel: string): boolean; virtual;
end;

TSpringer = class(TFigur)
  public
    function GeheZu(Ziel: string): boolean; virtual;
end;
```

In diesen Klassen werden die Methoden *GeheZu* schließlich programmiert, nämlich so, dass die Figur überprüft, ob der Zug überhaupt möglich ist, und den erfolgreichen Zug durch die Rückgabe von `True` bestätigt.

Die Schachbrett-Klasse erhält jeweils ein Array von *TFigur*-Objekten für die weißen und die schwarzen Figuren:

```
TSchachbrett = class
  private
    WeisseFiguren: Array[1..16] of TFigur;
    SchwarzeFiguren: Array[1..16] of TFigur;
  public
    constructor Create;
    function Ziehen(Quelle, Ziel: String): boolean;
end;
```

Der Konstruktor dieser Klasse erzeugt die einzelnen Figuren, wobei die erste Figur ein Turm ist, die zweite Figur ein Springer usw. Wenn gespielt wird, ermittelt die Methode *Ziehen* des Schachbretts die Figur, die auf der angegebenen Position steht, und ruft die Methode *GeheZu* dieser Figur auf. Egal welche Figur nun tatsächlich auf der Position stand, das Schachbrett kann ganz einfach die Methode *GeheZu* aufrufen, weil diese Methode in der Basisklasse der Figuren definiert ist. Das ist Polymorphismus.

Ich wollte noch erklären, warum überhaupt die Figuren selbst entscheiden, ob ein Zug möglich ist. Die Antwort ist: Erweiterbarkeit. Wenn die Figuren selbst für ihre Aktionen zuständig sind, ist es kein Problem, für ein erweitertes Schachspiel – ein »Space-Schach« mit 64 Figuren vielleicht – neue Figuren ins Spiel zu bringen, die vollkommen anderen Regeln gehorchen. Dazu müssen Sie lediglich diese neuen Figuren von der Basisklasse ableiten und in deren Methoden *GeheZu* deren ganz eigenes Verhalten implementieren. Das Schachspiel selbst muss nur an sehr wenigen Stellen geändert werden. Eventuell ist es erforderlich, in der Schachbrett-Klasse das Array für die Figuren zu erweitern, wenn sich die Anzahl der Figuren geändert hat. Außerdem muss der Konstruktor des Schachbretts die neuen Figuren erzeugen und deren Position setzen. Aber der Aufwand für diese Änderungen ist minimal im Vergleich zu dem Aufwand, der nötig wäre, wenn das Schachbrett das Verhalten der Figuren selbst kontrollieren würde.

Ich will diese eher theoretischen Ausführungen aber auch an einem nachvollziehbaren Beispiel erläutern. Ein Programm soll die Daten mehrerer unterschiedlicher Grafikobjekte (Kreise, Rechtecke, Dreiecke) speichern und auf Anforderung die Gesamtfläche aller gespeicherten Grafikobjekte berechnen können. Zur Vereinfachung werden die einzelnen Objekte in einem statischen Array gespeichert (in der Praxis würde man dazu wohl eher eine Auflistung verwenden). Der Benutzer soll die Möglichkeit haben, unterschiedliche Objekte in diesem Array abzulegen. Das Array kann also nicht wissen, welche Objekte tatsächlich abgelegt werden. Hier ist Polymorphismus gefragt. Das Array speichert Referenzen vom Typ einer Basisklasse, von der die Grafikobjekte abgeleitet werden.

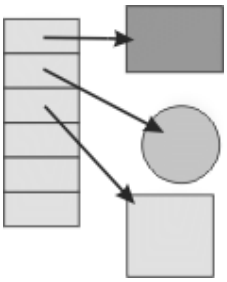

Abbildung 3.2: Eine Array speichert Referenzen auf unterschiedliche Objekte

Die Funktion zur Berechnung der Gesamtfläche kann nun nicht auf die unterschiedlichen Eigenschaften der einzelnen Objekte zurückgreifen (ein Kreis besitzt einen Radius, ein Rechteck und ein Dreieck eine Höhe und eine Breite). Deshalb wird in der Basisklasse eine Methode definiert, die die Fläche des Objekts berechnet. Diese Methode kann in der Basisklasse abstrakt sein und wird in den abgeleiteten Klassen implementiert:

```
unit GraphicObject_Classes;

interface

{ Abstrakte Basisklasse }
type TGraphicObject = class
 public
   { Abstrakte Methode zur Berechnung der Fläche }
   function CalcSurface: double; virtual; abstract;
end;

{ Klasse für einen Kreis }
type TCircle = class(TGraphicObject)
  private
    Radius: double;

  public
    { Der Konstruktor }
    constructor Create(Radius: double);

    { Die Methode CalcSurface wird überschrieben }
    function CalcSurface: double; override;
end;

{ Klasse für ein Rechteck }
type TRectangle = class(TGraphicObject)
  private
    Width: double;
    Height: double;

  public
    { Der Konstruktor }
    constructor Create(Width, Height: double);

    { Die Methode CalcSurface wird überschrieben }
    function CalcSurface: double; override;
end;


implementation

{ Der Konstruktor der TCircle-Klasse }
constructor TCircle.Create(Radius: double);
begin
  inherited Create;
  self.Radius := Radius;
end;

{ Die Methode zur Berechnung der Fläche der TCircle-Klasse }
function TCircle.CalcSurface: double;
```

```
begin
  CalcSurface := Radius * Radius * 3.1415927 / 4;
end;

{ Der Konstruktor der TRectangle-Klasse }
constructor TRectangle.Create(Width, Height: double);
begin
  inherited Create;
  self.Width := Width;
  self.Height := Height;
end;

{ Die Methode zur Berechnung der Fläche der TRectangle-Klasse }
function TRectangle.CalcSurface: double;
begin
  CalcSurface := Width * Height;
end;

end.
```

Das Beispielprogramm ermöglicht dem Benutzer nun, fünf Kreise oder Rechtecke zu erzeugen, und berechnet daraus dann die Gesamtfläche:

```
program Polymorphismus;

{$APPTYPE CONSOLE}

uses SysUtils, GraphicObject_Classes;


{ Ein Array, das Grafikobjekte speichert }
var GraphicObjects: array[1..5] of TGraphicObject;

{ Einige Hilfsvariablen }
var i: integer; input: string; r,w,h: integer;
var SurfaceSum: double;

begin
  writeln('Beispiel für Polymorphismus');

  { Der Benutzer erzeugt 5 beliebige Objekte }
  for i := 1 to 5 do
     begin
     repeat
        write('Objekt ' + IntToStr(i) +
              ': Welches Objekt wollen Sie erzeugen ' +
              '(k=Kreis, r=Rechteck): ');
        readln(input);
        if Input = 'k' then
           begin
           { Kreis erzeugen, dazu den Radius eingeben lassen }
           write('Radius: ');
           readln(r);
           GraphicObjects[i] := TCircle.Create(r);
           end
        else if input = 'r' then
           begin
           { Rechteck erzeugen, dazu Breite und Höhe eingeben lassen}
           write('Breite: ');
           readln(w);
           write('Hoehe: ');
           readln(h);
           GraphicObjects[i] := TRectangle.Create(w, h);
           end
        else
           begin
           writeln('Ungueltige Auswahl');
           end;
       until (Input = 'k') or (Input = 'r')
     end;
```

```
  { Gesamtfläche der Objekte berechnen }
  SurfaceSum := 0;
  for i := 1 to 5 do
     begin
     { Hier wird Polymorphismus angewendet.
       Unabhängig davon, welches Objekt tatsächlich
       in dem jeweiligen Array-Element
       gespeichert ist, kann das Programm die Methode
       CalcSurface aufrufen, weil diese in der Basisklasse
       deklariert ist und in den abgeleiteten Klassen
       überschrieben wird }
     SurfaceSum := SurfaceSum + GraphicObjects[i].CalcSurface;
     end;

  writeln;
  writeln('Die Gesamtsumme der Flaeche aller ' +
          'gespeicherten Objekte ist: ' +
          FloatToStr(SurfaceSum));
  readln;

  { Objekte freigeben }
  for i := 1 to 5 do
     begin
     GraphicObjects[i].Free;
     end;
end.
```

Ich hoffe, Sie haben den Polymorphismus mit diesen Erläuterungen schneller verstanden als ich damals, als ich diesem »Phänomen« begegnet bin ...

Polymorphismus kann man übrigens, außer durch Vererbung, auch über so genannte Schnittstellen erreichen. Schnittstellen beschreibe ich in Kapitel 4.

4 OOP-Specials

Dieses Kapitel beschreibt einige schon komplexere Techniken der OOP, die Sie in der Praxis manchmal benötigen.

4.1 Statische und virtuelle Methoden

Statische und virtuelle Methoden besitzen sehr viel Bedeutung bei der Vererbung, besonders im Zusammenhang mit dem Polymorphismus.

Statische Methoden sind eigentlich recht einfach erklärt. Dazu sollten Sie wissen, wie ein Programm Methoden nun wirklich im Arbeitsspeicher ablegt. Eine Methode ist ja, genau wie eine Prozedur oder Funktion, nichts anderes als eine Folge von Betriebssystem- oder Maschinensprachebefehlen. Der Compiler sorgt dafür, dass nicht jedes Objekt einer Klasse seine Methoden separat speichert. Das würde viel zu viel Arbeitsspeicher verbrauchen. Deshalb werden die Methoden einer Klasse ganz einfach separat von irgendwelchen Objekten im Arbeitsspeicher abgelegt, sobald das Programm eine Klasse benutzt. Den dazu notwendigen Programmcode baut der Compiler automatisch in das Programm ein. Abbildung 4.1 zeigt, wie das für drei Objekte einer Klasse (zur Speicherung von Kreisobjekten) im Arbeitsspeicher aussehen könnte.

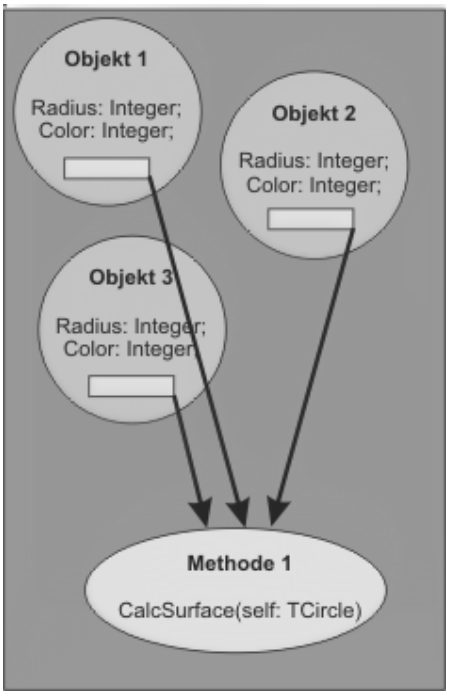

Abbildung 4.1: Methoden werden vom Compiler grundsätzlich separat von irgendwelchen Objekten im Arbeitsspeicher angelegt.

Bei statischen Methoden speichert das Objekt nun einen festen Zeiger auf den Speicherbereich, in dem die Methode angelegt ist (also die Adresse der Methode). Der Compiler erweitert die Methode um ein zusätzliches Argument, das Sie schon kennen, nämlich den Operator, der auf das Objekt zeigt (in Object Pascal: `self`; in Visual Basic: `Me`; in C++ und Java: `this`). Wenn Sie eine Methode einer Klasse von außen, über eine Instanz der Klasse, aufrufen, kennt der Compiler die Referenz auf das Objekt und kann diese an die Methode übergeben. Wenn Sie eine Methode einer Klasse innerhalb einer anderen Methode einer Klasse aufrufen, ergänzt der Compiler den Aufruf automatisch mit der Referenz auf das Objekt. Deshalb gilt eine Methode

immer für das Objekt, das die Methode aufgerufen hat. Diese Aufrufart wird im Allgemeinen auch als »frühe Bindung« bezeichnet.

Wie Sie Abbildung 4.1 entnehmen können, werden Eigenschaften übrigens in jedem Objekt separat gespeichert. Mit Eigenschaften treten also keine Probleme auf, weil jedes Objekt seine eigenen Daten verwaltet.

Mit statischen Methoden kann es dagegen schon einmal Probleme geben, nämlich dann, wenn Polymorphismus im Spiel ist. Angenommen, einer Variablen (allgemein: einer Referenz) vom Typ einer Basisklasse ist ein Objekt einer abgeleiteten Klasse zugewiesen (eben das ist ja Polymorphismus). Wenn das Programm über diese Referenz nun eine statische Methode aufruft, die in der abgeleiteten Klassen überschrieben wurde, führt das dazu, dass nicht die neue Methode der abgeleiteten Klasse aufgerufen wird, sondern die der Basisklasse. Die Variable besitzt ja lediglich eine Referenz auf die Basisklasse und in dieser ist die Methode (weil statisch) mit einer festen Adresse eingetragen. Das folgende Beispiel verdeutlicht dieses Problem:

```
unit Employee_Classes;

interface

{ Eine Basisklasse }
type TEmployee = class
  public Name: string;

  { Eine statische Methode, die das Gehalt
    des Mitarbeiters berechnet }
  function CalcSalary(Hours: integer): double;
end;

{ Eine abgeleitete Klasse }
type TChief = class(TEmployee)
  { Die geerbte statische Methode
    wird überschrieben }
  function CalcSalary(Hours: integer): double;
end;

implementation

function TEmployee.CalcSalary(Hours: integer): double;
begin
  { ein einfacher Mitarbeiter verdient 100 EUR pro Stunde }
  result := Hours * 100;
end;

function TChief.CalcSalary(Hours: integer): double;
begin
 { ein Chef verdient 200 EUR pro Stunde }
 result := Hours * 200;
end;

end.
```

Werden Referenzen auf die tatsächliche Klasse der Objekte verwendet, funktioniert die Berechnung korrekt. Werden dagegen allerdings Referenzen auf die Basisklasse zur Speicherung der Objekte verwendet, ist die Berechnung nicht mehr korrekt:

```
program Problem;

{$APPTYPE CONSOLE}

uses SysUtils, Employee_Classes;

var e: TEmployee; c: TChief;
var emps: array[1..2] of TEmployee;

begin
  writeln('Beispiel für das Problem, das statische Methoden bei der ' +
```

```
    'Vererbung verursachen, wenn Polymorphismus im Spiel ist');

  writeln;

  { Mit direkten Referenzen auf die Klasse
    werden die Gehälter korrekt berechnet }

  { Einen einfachen Mitarbeiter erzeugen }
  e := TEmployee.Create;

  { Das Gehalt berechnen }
  writeln('Der einfache Mitarbeiter verdient in ' +
    '100 Stunden ' + FloatToStr(e.CalcSalary(100)) + ' Euro');

  writeln;

  { Einen Chef erzeugen }
  c := TChief.Create;

  { Das Gehalt berechnen }
  writeln('Der Chef verdient in ' + '100 Stunden ' +
    FloatToStr(c.CalcSalary(100)) + ' Euro');

  { Mit Referenzen auf die Basisklasse werden die Gehälter
    nun nicht mehr korrekt berechnet: der Chef erhält
    dasselbe Gehalt wie der einfache Mitarbeiter }
  writeln;
  emps[1] := TEmployee.Create;
  writeln('Der einfache Mitarbeiter verdient in ' +
    '100 Stunden ' + FloatToStr(emps[1].CalcSalary(100)) + ' Euro');
  writeln;

  emps[2] := TChief.Create;

  { Hier tritt der Fehler auf: Der Compiler ruft nicht die
    in TChief implementierte Methode, sondern die Methode
    aus TEmployee auf, weil diese statisch deklariert ist }
  writeln('Der Chef verdient in ' + '100 Stunden ' +
    FloatToStr(emps[2].CalcSalary(100)) + ' Euro');

  readln;

  { Objekte freigeben }
  e.Free;
  c.Free;

end.
```

Abbildung 4.2 zeigt, was dabei passiert.

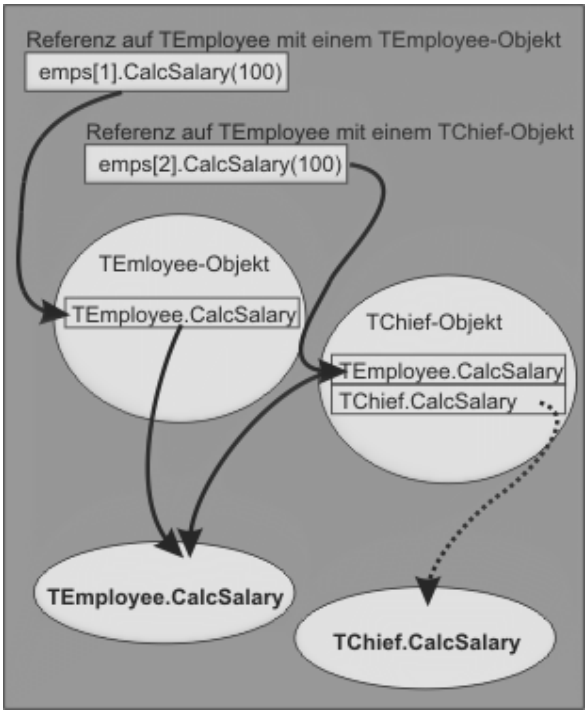

Abbildung 4.2: Das Problem mit statischen Methoden

Wenn Sie selbst kein Chef sind und dem Kommunismus anhängen, werden Sie vielleicht jetzt denken: »Oh, schöne Lösung, so verdienen Chefs immer dasselbe wie einfache Mitarbeiter.« Auch wenn dieses in manchen Fällen erstrebenswert ist, verursachen statische Methoden durch dieses Phänomen sehr komplizierte und schwierig zu findende Fehler.

4.2 Virtuelle Methoden und die VTABLE

Die Lösung des Problems ist eine virtuelle Methode. Für virtuelle Methoden speichert der Compiler nicht die Adresse der Methode aus dem Arbeitsspeicher in dem jeweiligen Objekt, sondern er verwaltet diese Methoden in einer separaten Tabelle, der so genannten *VTABLE*, die auch als »VMT« (Virtual Method Table) bezeichnet wird. Im Objekt wird lediglich die Adresse dieser VTABLE gespeichert. Der Compiler sorgt dafür, dass bei der Ausführung des Programms für jede Klasse eine eigene VTABLE im Speicher angelegt wird. Alle virtuellen Methoden dieser Klasse werden in dieser Tabelle, zusammen mit deren Adresse, eingetragen. An der Stelle des Aufrufs virtueller Methoden erzeugt der Compiler Programmcode, der in der VTABLE nachschaut, welche Methode für das jeweilige Objekt nun tatsächlich aufgerufen werden soll. Dabei wird nicht der Datentyp der Referenz, sondern der Datentyp des Objekts verwendet um die VTABLE zu identifizieren. Der Aufruf der Methode führt also über die VTABLE immer zu der korrekten Methode. Diese Aufrufart wird übrigens auch als »späte Bindung« bezeichnet, weil das Programm erst in der Laufzeit die Adresse der Methode ermittelt.

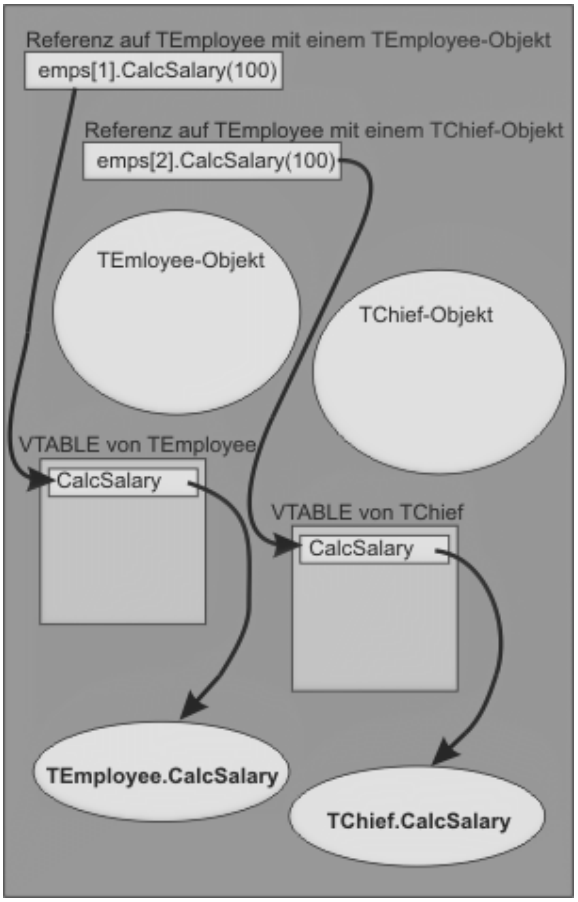

Abbildung 4.3: Aufruf von virtuellen Methoden über die VTABLE

Wenn Sie das Problem einmal verstanden haben, ist die Lösungen sehr einfach: Deklarieren Sie die Methoden, die in abgeleiteten Klassen überschrieben werden sollen, einfach virtuell. Programmiersprachen bieten Ihnen dazu ein Schlüsselwort, meist ist dies `virtual`, an. In Java brauchen Sie sich übrigens über virtuelle Methoden keine Gedanken zu machen, dort sind alle Methoden automatisch virtuell.

Das Problem mit der Berechnung des korrekten Gehalts für einen Chef aus dem vorigen Beispiel ist also schnell gelöst:

```
{ Eine Basisklasse }
type TEmployee = class
  public Name: string;

  { Eine virtuelle Methode, die das Gehalt
    des Mitarbeiters berechnet }
  function CalcSalary(Hours: integer): double; virtual;
end;

{ Eine abgeleitete Klasse }
type TChief = class(TEmployee)
  { Die geerbte statische Methode
    wird überschrieben }
  function CalcSalary(Hours: integer): double; override;
end;
```

Nun wird das Gehalt immer korrekt berechnet.

Beachten Sie, dass Sie in Object Pascal das `override`-Schlüsselwort angeben müssen, wenn Sie virtuelle Methoden überschreiben wollen. Vergessen Sie dieses Schlüsselwort in der abgeleiteten Klasse, wird die Methode nicht überschrieben, sondern durch eine neue, wieder statische Methode ersetzt. Der Fehler würde dann wieder auftreten, weil der Compiler die VTABLE umgeht und die Methode der Basisklasse aufruft. Der Compiler warnt Sie in diesem Fall, dass diese Methode die virtuelle Methode »verbirgt«. In anderen Sprachen ist allerdings kein weiteres Schlüsselwort notwendig, wenn Sie virtuelle Methoden überschreiben wollen.

Der Aufruf virtueller Methoden benötigt durch das notwendige Nachschauen in der VTABLE etwas mehr Zeit als der Aufruf statischer Methoden. Wenn Sie sehr schnelle Programme entwickeln müssen, sollten Sie sich also gut überlegen, welche Methoden virtuell sein müssen. Die Probleme, die statische Methoden verursachen können, haben aber in meinen Augen wesentlich mehr Gewicht als die Performance. Deklarieren Sie also Methoden, die in abgeleiteten Klassen überschrieben werden sollen, grundsätzlich virtuell.

4.3 Statische Klassenelemente

Im vorigen Abschnitt habe ich bereits die Funktionsweise von statischen Methoden beschrieben. Wie Sie darin gesehen haben, werden Methoden grundsätzlich unabhängig von den Objekten einer Klasse im Arbeitsspeicher angelegt. »Normale« Methoden können Sie nur über eine Instanz der Klasse (also über ein Objekt) aufrufen. Das zusätzliche Schlüsselwort `static` bzw. `class` (Object Pascal) bei der Deklaration einer Methode bewirkt nun, dass Sie diese Methode auch ohne eine Instanz der Klasse aufrufen können. Solche Methoden werden im Allgemeinen als *Klassenmethoden* bezeichnet.

4.3.1 Klassenmethoden

Klassenmethoden werden (sehr selten) verwendet, um Informationen über die Klasse zurückzugeben oder um irgendwelche Aktionen auszuführen, die alle Instanzen einer Klasse betreffen. In echt objektorientierten Programmiersprachen können Sie mit Klassenmethoden auch die prozedurale Programmierung simulieren. Sie entwickeln dazu Klassen mit Klassenmethoden für allgemeine Aufgaben, die nicht sinnvoll an Objekte gebunden werden können. Dazu gehören z. B. allgemeine mathematische Berechnungen wie die Berechnung eines Mittelwerts oder eines Maximalwerts.

Klassenmethoden werden aufgerufen, indem ihnen der Name der Klasse vorangestellt wird. Sie können solche Methoden allerdings auch über eine Instanz der Klasse aufrufen.

Ein Beispiel: Die Klassen unserer Kontoverwaltung sollen zurückmelden, ob bestimmte Operationen (Einzahlen, Auszahlen, Überweisen, Überziehen) mit den Objekten dieser Klasse möglich sind:

```
type TKonto = class
  public
    { Eine Klassenmethode, die zurückmeldet,
      ob bestimmte Operationen mit den
      Objekten der Klasse möglich sind }
    class function Supports(Action: string): boolean;
end;
```

```
type TGiroKonto = class(TKonto)
  public
    class function Supports(Action: string): boolean;
end;

class function TKonto.Supports(Action: string): boolean;
begin
  if (Action = 'Einzahlen') or
     (Action = 'Auszahlen') then
     Supports := true
  else
     Supports := false;
end;

class function TGiroKonto.Supports(Action: string): boolean;
begin
  if (Action = 'Einzahlen') or
     (Action = 'Auszahlen') or
     (Action = 'Überweisen') or
     (Action = 'Überziehen') then
     Supports := true
 else
     Supports := false;
end;
```

Sie können die Klassenmethode nun verwenden, ohne ein Objekt dieser Klasse zu erzeugen:

```
  { Aufruf der statischen Klassenmethode über den Klassenbezeichner }
  if TGirokonto.Supports('Überweisen') then
     begin
     k1 := TGirokonto.Create;

     { ... }

     end
  else
     begin
     writeln('Fehler: Das Girokonto unterstützt keine Überweisungen');
     end;
```

Sie können aber auch eine Instanz der Klasse verwenden:

```
 { Aufruf der statischen Klassenmethode über ein Objekt dieser Klasse }
 k1 := TGirokonto.Create;
 if k1.Supports('Überweisen') then
    begin
    { ... }
    end
 else
    begin
    writeln('Fehler: Das Girokonto unterstützt keine Überweisungen');
    end;
```

Die Frage, ob Klassenmethoden nun für diese klassische Anwendung Sinn machen oder nicht, lasse ich einmal im Raum stehen. Möglicherweise bringen diese Methoden in großen Klassenhierarchien mit langen Vererbungslinien etwas Übersicht und Information für den anwendenden Programmierer.

> Da Klassenmethoden unabhängig von Instanzen der Klasse verwendet werden, können Sie in diesen nicht auf die normalen Methoden und Eigenschaften der Klasse zurückgreifen (diese gehören ja immer zu einem Objekt). Umgekehrt geht das schon: In normalen Methoden der Klasse können Sie auf statische Klassenelemente zurückgreifen. Das Java-Beispiel im nächsten Abschnitt demonstriert dies. Hier wird eine statische Eigenschaft im Konstruktor und im Destruktor der Klasse verwendet.

Echt objektorientierte Programmiersprachen wie C# und Java nutzen Klassenmethoden sehr exzessiv für ihre Funktionsbibliotheken, so dass für den Aufruf einfacher Funktionen nicht immer erst ein Objekt instanziert werden muss. Für diese Art der Anwendung sind Klassenmethoden in echten OOP-Sprachen (die ja keine Module kennen) sehr gut geeignet.

4.3.2 Statische Eigenschaften

Genau wie Methoden können Sie in den meisten Sprachen auch Eigenschaften statisch deklarieren. Eine statische Eigenschaft verhält sich wie eine globale Variable, die für alle Instanzen einer Klasse gilt. Schreibt die eine Instanz einen Wert in eine solche Eigenschaft, kann die andere Instanz diesen Wert auslesen. Solche Eigenschaften werden also immer dann verwendet, wenn Daten über alle Instanzen einer Klasse global gespeichert werden sollen. In Programmiersprachen, die keine echt globalen Variablen kennen, können Sie diese damit simulieren.

Ein Beispiel: In einer Personenverwaltung soll die Anzahl der gespeicherten Personen verwaltet werden. Der Konstruktor der Klasse erhöht dazu einen statischen Personenzähler um 1, der Destruktor reduziert diesen wieder. Da Object Pascal leider keine statischen Eigenschaften kennt, habe ich dieses Beispiel in Java entwickelt:

```
public class Static_Demo
{
   public static void main(String[] args)
   {
      System.out.println("Java-Beispiel fuer statische "
        "Klassenelemente");
      Person p1 = new Person("Zaphod");
      Person p2 = new Person("Trillian");
      Person p3 = new Person("Ford");

      /* Ausgabe der Anzahl über die Klasse */
      System.out.println("Das Programm speichert " +
        Person.getCounter() + " Personen.");

      /* Ausgabe der Anzahl über eine Instanz der Klasse */
      System.out.println("Das Programm speichert " +
        p1.getCounter() + " Personen.");
   }
}

class Person
{
   public String Name;

   /* Statische Eigenschaft, die die Anzahl
      der aktuell existierenden Personen
      speichert */
   private static int Counter;

   /* Klassenmethode zum Lesen des Zählers */
   public static int getCounter()
   {
      return Counter;
   }

   /* Der Konstruktor */
   public Person(String Name)
   {
      this.Name = Name;
      Counter = Counter + 1;
   }

   /* Der Destruktor */
```

```
   public void finalize()
   {
      Counter = Counter - 1;
   }
}
```

4.4 Schnittstellen

Schnittstellen (engl.: Interfaces) sind ein weiteres wichtiges Konzept der modernen OOP. Als Schnittstelle werden bei der OOP im Allgemeinen die öffentlichen (Public-)Methoden und Eigenschaften einer Klasse bezeichnet. Jede Klasse besitzt damit mindestens eine Schnittstelle. In modernen Programmiersprachen können Klassen jedoch auch mehrere Schnittstellen enthalten. Jede dieser Schnittstellen besitzt einen eigenen Satz einen Eigenschaften und Methoden. Einer Klasse eine oder mehrere zusätzliche Schnittstellen hinzuzufügen ist sehr einfach (das *Verständnis*, welchen Sinn Schnittstellen haben, ist vielleicht etwas schwieriger, aber das erkläre ich Ihnen auch noch). Sie müssen die Schnittstelle als solche separat deklarieren und können diese dann der Klasse hinzufügen.

> Object Pascal erlaubt leider nicht die Deklaration von einfachen Schnittstellen. Schnittstellen basieren hier immer auf dem Microsoft-COM-Modell (Component Object Model), das die Kommunikation zwischen Anwendungen und Objekten regelt, wobei die Klassen der Objekte auch in kompilierten und auf dem System registrierten so genannten Komponenten gespeichert sein können. Das COM-Modell ist nun leider etwas komplex und erwartet einige besondere Methoden in einer Schnittstelle. Diese Methoden würden Sie wahrscheinlich ein wenig verwirren. Außerdem reicht der Platz in diesem Artikel nicht aus, um das COM-Modell zu beschreiben. Glücklicherweise müssen Sie die COM-Standardmethoden nicht selbst implementieren. Sie leiten Ihre Klasse in Object Pascal einfach von der Basisklasse *TInterfacedObject* ab, wenn Sie der Klasse Schnittstellen hinzufügen wollen. Diese Klasse implementiert die COM-Standardmethoden für Sie.

```
unit Interface_Demo_Classes;

interface

uses SysUtils;

{ Deklaration der Schnittstelle }
type IPrint = interface(IUnknown)
  procedure Print;
end;

{ Deklaration einer Personenklasse, die die
  Schnittstelle implementiert }
type TPerson = class(TInterfacedObject, IPrint)
  public
    Vorname: string;
    Nachname: String;
    Geburtsdatum: TDateTime;

    { Die Deklaration der Schnittstelle muss auch in
      die Klasse eingefügt werden }
    procedure Print;
end;

{ Deklaration einer Klasse zum Speichern von Auto-Daten }
type TAuto = class(TInterfacedObject, IPrint)
  public
    Typbezeichnung: string;
    Farbe: string;
    Baujahr: integer;

    { Die Deklaration der Schnittstelle muss auch in
      die Klasse eingefügt werden }
    procedure Print;
end;

implementation

{ Die Schnittstelle wird für jede Klasse,
  die die Schnittstelle geerbt hat, implementiert }
procedure TPerson.Print;
begin
  writeln('Ich bin eine Person. Mein Name ist ' +
    Vorname + ' ' + Nachname + '.');
  writeln('Ich bin am ' + DateToStr(Geburtsdatum) + ' geboren.');
end;

procedure TAuto.Print;
begin
  writeln('Ich bin ein Auto vom Typ ' + Typbezeichnung + '.');
  writeln('Meine Farbe  ist ' + Farbe + '.');
  writeln('Mein Baujahr ist ' + IntToStr(Baujahr) + '.');
end;

end.
```

Wie Sie dem Beispiel entnehmen können, werden die Methoden der Schnittstelle erst in den Klassen implementiert, die die Schnittstelle einbinden. Innerhalb einer Schnittstelle ist (normalerweise[15]) keine Implementierung von Methoden möglich. Aus einer Schnittstelle können Sie also selbst keine Objekte erzeugen. Die Schnittstelle wird – mit einer für die

[15] In allen mir bekannten Sprachen gibt es keine Möglichkeit, Schnittstellen so zu deklarieren, dass in diesen auch die Methoden implementiert werden. Und doch gibt es im COM-Modell einige Basisschnittstellen (wie z. B. *IShellLink*), die bereits auch eine Implementierung enthalten. Von diesen Schnittstellen kann man dann direkt auch Objekte erzeugen.

einzelnen Sprachen unterschiedlichen Technik – in die Klassen eingebunden, die diese Schnittstelle implementieren sollen. In Object Pascal geben Sie die Schnittstelle einfach hinter dem Klassenbezeichner an, von dem Sie Ihre Klasse ableiten. Die Methoden der Schnittstelle müssen dann auch noch in der Klasse deklariert und natürlich auch implementiert werden.

Jetzt, wo Sie wissen, wie Sie Schnittstellen in verschiedene Klassen implementieren können, werden Sie wahrscheinlich wissen wollen, wozu man diese überhaupt benötigt. Die Antwort ist eigentlich relativ einfach: Schnittstellen werden verwendet, um Polymorphismus zu realisieren. Die Grundlagen des Polymorphismus habe ich ja bereits in Kapitel 3 beschrieben. Den »normalen« Polymorphismus erreichen Sie über einfache Vererbung. Polymorphismus über Schnittstellen ist eine besondere Art, hat aber dieselbe Grundbedeutung.

Ein Programm, das *Personen*- und *Autoobjekte* instanziert, kann z. B. mit einer Referenz auf die Schnittstelle (!) die Methoden und Eigenschaften derselben verwenden:

```
{ Dieses Programm demonstriert die Anwendung
  von Schnittstellen }
program Schnittstellen;

uses SysUtils,
  Interface_Demo_Classes in 'Interface_Demo_Classes.pas';

{$APPTYPE CONSOLE}

var person: TPerson; auto: TAuto;

procedure PrintDataOfObject(ip: IPrint);
begin
  { Diese Prozedur bekommt eine Referenz auf die Schnittstelle übergeben
    und kann deswegen alle in dieser Schnittstelle deklarierten
    Eigenschaften und Methoden verwenden }
  ip.Print;
end;

begin
  writeln('Beispiel-Programm zur Demonstration von Schnittstellen');
  writeln;

  { Eine Person erzeugen }
  person := TPerson.Create;
  person.Vorname := 'Eddi';
  person.Nachname := 'The Cat';
  person.Geburtsdatum := StrToDate('12.3.1992');

  { Ein Auto erzeugen }
  auto := TAuto.Create;
  auto.Typbezeichnung := 'Ford Puma';
  auto.Baujahr := 1998;
  auto.Farbe := 'Rot';

  { Diese beiden Objekte werden nun an eine Prozedur übergeben, die als
    Argument einer Referenz auf die Schnittstelle besitzt }
  PrintDataOfObject(auto);
  writeln;
  PrintDataOfObject(person);

  readln;

  { Objekte freigeben }
  auto.Free;
  person.Free;
end.
```

Erkennen Sie den Unterschied zum normalen Polymorphismus?

> Schnittstellen ermöglichen Polymorphismus auch für Klassen, die nicht von einer gemeinsamen Basisklasse abgeleitet werden. Mit Schnittstellen können Sie Polymorphismus sogar für Klassen implementieren, die Sie von unveränderbaren Basisklassen abgeleitet haben.

Das ist der eine große Vorteil von Schnittstellen: Sie können mit Hilfe von diesen Polymorphismus auch dann für verschiedene Klassen ermöglichen, wenn Sie keine Möglichkeit haben, diese Klassen von einer gemeinsamen Basisklasse abzuleiten. Das ist immer dann der Fall, wenn zwei Klassen eigentlich keine Gemeinsamkeiten besitzen, wie im obigen Beispiel eine Person und ein Auto, die Objekte dieser Klassen aber trotzdem polymorph sein sollen. Diese Technik ermöglicht Ihnen sogar Polymorphismus für Klassen, die Sie von den in der Regel unveränderlichen Systemklassen Ihrer Programmiersprache abgeleitet haben. So könnte man sich zum Beispiel eine Anwendung vorstellen, die zwei verschiedene Windows-Fenster verwendet, eines z. B. für die Ausgabe von Texten und eines für die Ausgabe von Grafiken. Die Anwendung soll (ähnlich wie Word oder Excel) dem Anwender ermöglichen, mehrere Texte und mehrere Bilder gleichzeitig zu bearbeiten. Dazu erzeugen Sie in Ihrem Programm jeweils ein Fenster, das Sie von der Systemklasse für Windows-Fenster ableiten und das Sie mit der benötigten Funktionalität erweitern. Im Programm wird jeweils, auf Anforderung des Anwenders, ein neues Bild- oder Textfenster erzeugt, wobei die Referenz auf dieses Fenster vielleicht in einem Gerät gespeichert wird. Die Anwendung soll nun die Möglichkeit haben, den Text beziehungsweise das Bild des aktuell angezeigten Fensters ausdrucken. Um dieses zu ermöglichen, sind Sie quasi gezwungen, in die beiden Klassen für das Bild beziehungsweise den Text eine Schnittstelle zu implementieren, da Sie die Basisklasse nicht verändern können.

Eine Besonderheit von Schnittstellen gegenüber dem normalen Polymorphismus ist noch, dass eine Klasse auch ohne Probleme *mehrere* Schnittstellen implementieren kann. Deklarieren Sie dazu einfach die benötigten Schnittstellen und geben Sie diese bei der Deklaration der Klasse durch Kommata getrennt an.

5 Index