PROGRAMMER'S CHOICE

Guido Krüger
Thomas Stark

Handbuch der Java Programmierung

Standard Edition Version 6,
5. Auflage

DVD

- Alle Beispiele aus dem Buch
- Die HTML-Ausgabe
- Java 6 SDK und Eclipse 3.3
- Ausgewählte Video-Trainings
- eBook »Masterclass Java EE 5«

ADDISON-WESLEY

Teil III Objektorientierte Programmierung

7 OOP I: Grundlagen

7.1 Konzepte objektorientierter Programmiersprachen

7.1.1 Einführung

Objektorientierte Programmierung (kurz *OOP*) ist *das* Programmierparadigma der 90er Jahre. Viele der heute verwendeten Programmiersprachen sind entweder von Grund auf objektorientiert (Java, Eiffel, SmallTalk) oder wurden im Laufe der Zeit mit objektorientierten Erweiterungen versehen (Basic, Pascal, ADA). Selbst manche Scriptsprachen erlauben den Zugriff auf (mitunter vordefinierte) Objekte oder besitzen objektorientierte Eigenschaften (JavaScript, Python). Die objektorientierte Programmierung war eine der »Silver Bullets«, die die Software-Industrie aus ihrer Krise führen und zu robusteren, fehlerärmeren und besser wartbaren Programmen führen sollte.

Was sind nun die Geheimnisse der objektorientierten Programmierung? Was verbirgt sich hinter dem Begriff, und welches sind seine wichtigsten Konzepte? Wir wollen uns zunächst mit den Grundideen objektorientierter Programmierung auseinandersetzen und dann in diesem und den nächsten Kapiteln Schritt für Schritt erläutern, wie sie in Java umgesetzt wurden.

7.1.2 Abstraktion

Eine der wichtigsten Ideen der objektorientierten Programmierung ist die Trennung zwischen *Konzept* und *Umsetzung*, etwa zwischen einem *Bauteil* und seinem *Bauplan*, einer *Speise* und dem für die Zubereitung erforderlichen *Rezept* oder einem *technischen Handbuch* und der *konkreten Apparatur*, die dadurch beschrieben wird.

Diese Art von Unterscheidung ist in der wirklichen Welt sehr bedeutsam. Wer weiß, wie man *einen einzigen* Lichtschalter bedient, kann andere, gleichartige Schalter ebenfalls bedienen. Wer ein Rezept für eine Sachertorte besitzt, ist in der Lage, diese zu backen, selbst wenn er ansonsten über keine Koch- oder Backkünste verfügt. Wer einen Führerschein gemacht hat, kann ein Auto fahren, ohne im Detail über das komplizierte Innenleben desselben unterrichtet zu sein.

In der objektorientierten Programmierung manifestiert sich diese Unterscheidung in den Begriffen *Objekt* und *Klasse*. Ein Objekt ist ein tatsächlich existierendes »Ding« aus der Anwendungswelt des Programms. Es spielt dabei keine Rolle, ob es sich um die programmierte Umsetzung eines konkret existierenden Gegenstands handelt oder ob »nur« ein abstraktes Konzept modelliert wird. Eine »Klasse« ist dagegen die Beschreibung eines oder mehrerer ähnlicher Objekte. »Ähnlich« bedeutet dabei, dass eine Klasse nur Objekte eines bestimmten Typs beschreibt. Diese müssen sich zwar nicht in allen Details gleichen, aber doch in so vielen von ihnen übereinstimmen, dass eine gemeinsame Beschreibung angebracht ist. Eine Klasse beschreibt mindestens drei wichtige Dinge:

- Wie ist das Objekt zu bedienen?
- Welche Eigenschaften hat das Objekt und wie verhält es sich?
- Wie wird das Objekt hergestellt?

Ähnlich wie ein Rezept zur Herstellung von Hunderten von Sachertorten verwendet werden kann, ermöglicht eine Klasse das Erzeugen einer prinzipiell beliebigen Anzahl von Objekten. Jedes hat dabei seine eigene Identität und mag sich in gewissen Details von allen anderen unterscheiden. Letztlich ist das Objekt aber immer eine *Instanz* der Klasse, nach der es modelliert wurde. In einem Haus kann es beispielsweise fünfzig Lichtschalter-Objekte geben. Sie alle sind Instanzen der Klasse »Lichtschalter«, lassen sich in vergleichbarer Weise bedienen und sind identisch konstruiert. Dennoch unterscheiden wir sehr wohl zwischen dem Lichtschalter-Objekt, das die Flurbeleuchtung bedient, und jenem, das den Keller erhellt. Und beide wiederum unterscheiden sich eindeutig von allen anderen Lichtschalterinstanzen im Haus.

INFO

Es ist übrigens kein Zufall, dass wir die Begriffe »Instanz« und »Objekt« synonym verwenden. In der objektorientierten Programmierung ist das sehr verbreitet (auch wenn Puristen zwischen beiden Begriffen noch Unterschiede sehen) und wir wollen uns diesem Wortgebrauch anschließen.

Diese Unterscheidung zwischen Objekten und Klassen kann als *Abstraktion* angesehen werden. Sie bildet die erste wichtige Eigenschaft objektorientierter Sprachen. Abstraktion hilft, Details zu ignorieren, und reduziert damit die Komplexität des Problems. Die

Fähigkeit zur Abstraktion ist eine der wichtigsten Voraussetzungen zur Beherrschung komplexer Apparate und Techniken und kann in seiner Bedeutung nicht hoch genug eingeschätzt werden.

7.1.3 Kapselung

In objektorientierten Programmiersprachen wird eine Klasse durch die Zusammenfassung einer Menge von Daten und darauf operierender Funktionen (die nun *Methoden* genannt werden) definiert. Die Daten werden durch einen Satz Variablen repräsentiert, der für jedes instanziierte Objekt neu angelegt wird (diese werden als *Attribute*, *Membervariablen*, *Instanzvariablen* oder *Instanzmerkmale* bezeichnet). Die Methoden sind im ausführbaren Programmcode nur einmal vorhanden, operieren aber bei jedem Aufruf auf den Daten eines ganz bestimmten Objekts (das Laufzeitsystem übergibt bei jedem Aufruf einer Methode einen Verweis auf den Satz Instanzvariablen, mit dem die Methode gerade arbeiten soll).

Die Instanzvariablen repräsentieren den *Zustand* eines Objekts. Sie können bei jeder Instanz einer Klasse unterschiedlich sein und sich während seiner Lebensdauer verändern. Die Methoden repräsentieren das *Verhalten* des Objekts. Sie sind – von gewollten Ausnahmen abgesehen, bei denen Variablen bewusst von außen zugänglich gemacht werden – die einzige Möglichkeit, mit dem Objekt zu kommunizieren und so Informationen über seinen Zustand zu gewinnen oder diesen zu verändern. Das Verhalten der Objekte einer Klasse wird in seinen Methodendefinitionen festgelegt und ist von dem darin enthaltenen Programmcode und dem aktuellen Zustand des Objekts abhängig.

Diese Zusammenfassung von Methoden und Variablen zu Klassen bezeichnet man als *Kapselung*. Sie stellt die zweite wichtige Eigenschaft objektorientierter Programmiersprachen dar. Kapselung hilft vor allem, die Komplexität der Bedienung eines Objekts zu reduzieren. Um eine Lampe anzuschalten, muss man nicht viel über den inneren Aufbau des Lichtschalters wissen. Sie vermindert aber auch die Komplexität der Implementierung, denn undefinierte Interaktionen mit anderen Bestandteilen des Programms werden verhindert oder reduziert.

7.1.4 Wiederverwendung

Durch die Abstraktion und Kapselung wird die *Wiederverwendung* von Programmelementen gefördert, die dritte wichtige Eigenschaft objektorientierter Programmiersprachen. Ein einfaches Beispiel dafür sind *Collections*, also Objekte, die Sammlungen anderer Objekte aufnehmen und auf eine bestimmte Art und Weise verarbeiten können. Collections sind oft sehr kompliziert aufgebaut (typischerweise zur Geschwindigkeitssteigerung oder Reduzierung des Speicherbedarfs), besitzen aber in aller Regel eine einfache Schnittstelle. Werden sie als Klasse implementiert und werden durch die Kapselung der Code- und Datenstrukturen die komplexen Details »wegabstrahiert«, können sie sehr einfach wiederverwendet werden. Immer, wenn im Programm eine entsprechende Collection benötigt

wird, muss lediglich ein Objekt der passenden Klasse instanziert werden, und das Programm kann über die einfach zu bedienende Schnittstelle darauf zugreifen. Wiederverwendung ist ein wichtiger Schlüssel zur Erhöhung der Effizienz und Fehlerfreiheit beim Programmieren.

7.1.5 Beziehungen

Objekte und Klassen existieren für gewöhnlich nicht völlig alleine, sondern stehen in Beziehungen zueinander. So ähnelt ein Fahrrad beispielsweise einem Motorrad, hat aber auch mit einem Auto Gemeinsamkeiten. Ein Auto ähnelt dagegen einem Lastwagen. Dieser kann einen Anhänger haben, auf dem ein Motorrad steht. Ein Fährschiff ist ebenfalls ein Transportmittel und kann viele Autos oder Lastwagen aufnehmen, genauso wie ein langer Güterzug. Dieser wird von einer Lokomotive gezogen. Ein Lastwagen kann auch einen Anhänger ziehen, muss es aber nicht. Bei einem Fährschiff ist keine Zugmaschine erforderlich und es kann nicht nur Transportmittel befördern, sondern auch Menschen, Tiere oder Lebensmittel.

Wir wollen ein wenig Licht in diese Beziehungen bringen und zeigen, wie sie sich in objektorientierten Programmiersprachen auf wenige Grundtypen reduzieren lassen:

- »is-a«-Beziehungen (Generalisierung, Spezialisierung)
- »part-of«-Beziehungen (Aggregation, Komposition)
- Verwendungs- oder Aufrufbeziehungen

Generalisierung und Spezialisierung

Zuerst wollen wir die »is-a«-Beziehung betrachten. »is-a« bedeutet »ist ein« und meint die Beziehung zwischen »ähnlichen« Klassen. Ein Fahrrad ist kein Motorrad, aber beide sind Zweiräder. Ein Zweirad, und damit sowohl das Fahrrad als auch das Motorrad, ist ein Straßenfahrzeug, ebenso wie das Auto und der Lastwagen. All diese Klassen repräsentieren Transportmittel, zu denen aber auch die Schiffe und Güterzüge zählen.

Die »is-a«-Beziehung zwischen zwei Klassen *A* und *B* sagt aus, dass »*B* ein *A* ist«, also alle Eigenschaften von *A* besitzt, und vermutlich noch ein paar mehr. *B* ist demnach eine Spezialisierung von *A*. Andersherum betrachtet, ist *A* eine Generalisierung (Verallgemeinerung) von *B*.

»is-a«-Beziehungen werden in objektorientierten Programmiersprachen durch *Vererbung* ausgedrückt. Eine Klasse wird dabei nicht komplett neu definiert, sondern von einer anderen Klasse *abgeleitet*. In diesem Fall erbt sie alle Eigenschaften dieser Klasse und kann nach Belieben eigene hinzufügen. In unserem Fall wäre also *B* von *A* abgeleitet. *A* wird als *Basisklasse* (manchmal auch als *Vaterklasse*), *B* als *abgeleitete Klasse* bezeichnet.

Vererbungen können mehrstufig sein, d.h., eine abgeleitete Klasse kann Basisklasse für weitere Klassen sein. Auf diese Weise können vielstufige *Vererbungshierarchien* entstehen,

die in natürlicher Weise die Taxonomie (also die gegliederte Begriffsstruktur) der zu modellierenden Anwendungswelt repräsentieren. Vererbungshierarchien werden wegen ihrer Baumstruktur auch als *Ableitungsbäume* bezeichnet. Sie werden meist durch Graphen dargestellt, in denen die abgeleiteten Klassen durch Pfeile mit den Basisklassen verbunden sind und die Basisklassen oberhalb der abgeleiteten Klassen stehen. Für unsere Fahrzeugwelt ergäbe sich beispielsweise folgender Ableitungsbaum:

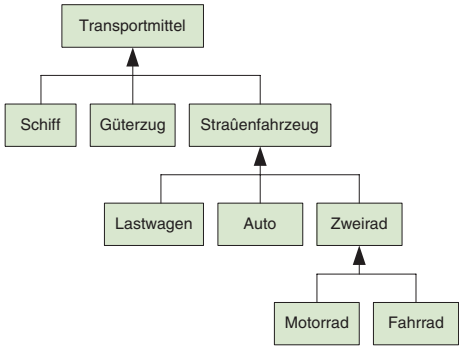

Abbildung 7.1: Vererbungshierarchie für Transportmittel

Als Eigenschaften der Basisklasse *Transportmittel* könnten etwa dessen Anschaffungskosten, seine Lebensdauer oder die Transportgeschwindigkeit angesehen werden. Sie gelten für alle abgeleiteten Klassen. In der zweiten Ableitungsebene unterscheiden wir nach der Art der Fortbewegung (wir hätten allerdings ebensogut nach der Farbe, dem Verwendungszweck oder einem beliebigen anderen Merkmal unterscheiden können). In der Klasse *Wasserfahrzeug* könnten nun Eigenschaften wie Verdrängung, Hochseetauglichkeit und erforderliche Besatzung festgehalten werden. Das Fährschiff schließlich fügt seine Transportkapazitäten für Autos, Lastwagen und Personen hinzu, gibt die Anzahl der Kabinen der unterschiedlichen Kategorien an und definiert, ob es im RORO-Verfahren be- und entladen werden kann oder nicht.

INFO

In manchen objektorientierten Programmiersprachen kann eine abgeleitete Klasse mehr als eine Basisklasse besitzen (z.B. in C++ oder Eiffel). In diesem Fall spricht man von *Mehrfachvererbung*. Die Vererbungshierarchie ist dann nicht mehr zwangsläufig ein Baum, sondern muss zu einem gerichteten Graph verallgemeinert werden. In Java gibt es allerdings keine Mehrfachvererbung und wir wollen daher nicht weiter auf die Besonderheiten dieser Technik eingehen.

Aggregation und Komposition

Der zweite Beziehungstyp, die »part-of«-Beziehungen, beschreibt die *Zusammensetzung* eines Objekts aus anderen Objekten (dies wird auch als *Komposition* bezeichnet). So besteht beispielsweise der Güterzug aus einer (oder manchmal zwei) Lokomotiven und einer großen Anzahl Güterzuganhänger. Der Lastwagen besteht aus der LKW-Zugmaschine und eventuell einem Anhänger. Ein Fahrrad besteht aus vielen Einzelteilen. Objektorientierte Sprachen implementieren »part-of«-Beziehungen durch Instanzvariablen, die Objekte aufnehmen können. Der Güterzug könnte also eine (oder zwei) Instanzvariablen vom Typ *Lokomotive* und ein Array von Instanzvariablen vom Typ *Güterzuganhänger* besitzen.

»part-of«-Beziehungen müssen nicht zwangsläufig beschreiben, *woraus* ein Objekt zusammengesetzt ist. Vielmehr können sie auch den allgemeineren Fall des *einfachen Aufnehmens* anderer Objekte beschreiben (was auch als *Aggregation* bezeichnet wird). Zwischen dem Motorrad, das auf dem Lastwagenanhänger steht, oder den Straßenfahrzeugen, die auf einem Fährschiff untergebracht sind, besteht zwar eine »part-of«-Beziehung, sie ist aber nicht *essentiell* für die Existenz des aufnehmenden Objekts. Der Anhänger existiert auch, wenn kein Motorrad darauf plaziert ist. Und das Fährschiff kann auch leer von Kiel nach Oslo fahren.

Während bei der objektorientierten Modellierung sorgsam zwischen beiden Fällen unterschieden wird (Komposition bezeichnet die strenge Form der Aggregation auf Grund einer existentiellen Abhängigkeit), behandeln objektorientierte Programmiersprachen sie prinzipiell gleich. In beiden Fällen gibt es Instanzvariablen, die Objekte aufnehmen können. Ist ein optionales Objekt nicht vorhanden, wird dies durch die Zuweisung eines speziellen *null-Objekts* ausgedrückt. Für die semantischen Eigenschaften der Beziehung ist die Klasse selbst verantwortlich.

Verwendungs- und Aufrufbeziehungen

Die dritte Art von Beziehungen zwischen Objekten oder Klassen hat den allgemeinsten Charakter. Benutzt beispielsweise eine Methode während ihrer Ausführung ein temporäres Objekt, so besteht zwischen beiden eine Verwendungsbeziehung: Objekt *x* verwendet eine Instanz der Klasse *Y*, um bestimmte Operationen auszuführen. Taucht in der Argumentliste einer Methode eine Objektvariable der Klasse *T* auf, so entsteht eine ähnliche Beziehung zu *T*. Zwar ist dies keine »part-of«-Beziehung und auch die Ableitungsbeziehung zwischen beiden Klassen spielt keine Rolle. Wenigstens muss aber die Methode die Argumentklasse kennen und in der Lage sein, Methoden darauf aufzurufen oder das Objekt an Dritte weiterzugeben.

Allgemeine Verwendungs- oder Aufrufbeziehungen finden in objektorientierten Programmiersprachen ihren Niederschlag darin, dass Objekte als lokale Variablen oder Methodenargumente verwendet werden. Sie werden auch mit dem Begriff *Assoziationen* bezeichnet.

INFO

In den vorangegangenen Abschnitten wurden mehrfach die Begriffe *Membervariable*, *Instanzvariable* und *Objektvariable* verwendet. Als *Objektvariable* bezeichnen wir stets eine Variable, die ein Objekt aufnehmen kann, also vom Typ einer Klasse ist. Das Gegenteil einer Objektvariable ist eine primitive Variable. Die Begriffe *Membervariable* und *Instanzvariable* werden synonym verwendet. Sie bezeichnen eine Variable, die innerhalb einer Klasse definiert wurde und mit jeder Instanz neu angelegt wird.

7.1.6 Polymorphismus

Als letztes wichtiges Konzept objektorientierter Programmiersprachen wollen wir uns mit dem *Polymorphismus* beschäftigen. Polymorphismus bedeutet direkt übersetzt etwa »Vielgestaltigkeit« und bezeichnet zunächst einmal die Fähigkeit von Objektvariablen, Objekte unterschiedlicher Klassen aufzunehmen. Das geschieht allerdings nicht unkontrolliert, sondern beschränkt sich für eine Objektvariable des Typs *X* auf alle Objekte der Klasse *X* oder einer daraus abgeleiteten Klasse.

Eine Objektvariable vom Typ *Straßenfahrzeug* kann also nicht nur Objekte der Klasse *Straßenfahrzeug* aufnehmen, sondern auch Objekte der Klassen *Zweirad*, *Vierrad*, *Anhänger*, *Motorrad*, *Fahrrad*, *Auto* und *Lastwagen*. Diese auf den ersten Blick erstaunliche Lässigkeit entspricht allerdings genau dem gewohnten Umgang mit Vererbungsbeziehungen. Ein *Zweirad* ist nunmal ein *Straßenfahrzeug*, hat alle Eigenschaften eines Straßenfahrzeugs und kann daher durch eine Variable repräsentiert werden, die auf ein Straßenfahrzeug verweist. Dass es möglicherweise ein paar zusätzliche Eigenschaften besitzt, stört den Compiler nicht. Er hat nur sicherzustellen, dass die Eigenschaften eines Straßenfahrzeugs vollständig vorhanden sind, denn mehr stellt er dem Programm beim Zugriff auf eine Variable dieses Typs nicht zur Verfügung. Davon kann er aber aufgrund der Vererbungshierarchie ausgehen.

INFO

Anders herum funktioniert Polymorphismus nicht. Wäre es beispielsweise möglich, einer Variable des Typs *Motorrad* ein Objekt des Typs *Zweirad* zuzuzuweisen, könnte das Laufzeitsystem in Schwierigkeiten geraten. Immer wenn auf der *Motorrad*-Variablen eine Eigenschaft benutzt würde, die in der Basisklasse *Zweirad* noch nicht vorhanden ist, wäre das Verhalten des Programms undefiniert, wenn zum Ausführungszeitpunkt nicht ein *Motorrad*, sondern ein Objekt aus der Basisklasse darin gespeichert wäre.

Interessant wird Polymorphismus, wenn die Programmiersprache zusätzlich das Konzept des *Late Binding* implementiert. Im Unterschied zum »Early Binding« wird dabei nicht bereits zur Compilezeit entschieden, welche Ausprägung einer bestimmten Methode aufgerufen werden soll, sondern erst zur Laufzeit. Wenn beispielsweise auf einem Objekt der Klasse *X* eine Methode mit dem Namen *f* aufgerufen werden soll, ist zwar prinzipiell

bereits zur Compilezeit klar, wie der Name lautet. Objektorientierte Programmiersprachen erlauben aber das *Überlagern* von Methoden in abgeleiteten Klassen, und da – wie zuvor erwähnt – eine Objektvariable des Typs *X* auch Objekte aus allen von *X* abgeleiteten Klassen aufnehmen kann, könnte *f* in einer dieser nachgelagerten Klassen überlagert worden sein. Welche konkrete Methode also aufgerufen werden muss, kann damit erst zur Laufzeit entschieden werden. Wir werden in Abschnitt 8.4, Seite 191 ein ausführliches Anwendungsbeispiel vorstellen.

Nun ist dieses Verhalten keinesfalls hinderlich oder unerwünscht, sondern kann sehr elegant dazu genutzt werden, automatische typbasierte Fallunterscheidungen vorzunehmen. Betrachten wir dazu noch einmal unsere Hierarchie von Transportmitteln. Angenommen, unser Unternehmen verfügt über einen breit gefächerten Fuhrpark von Transportmitteln aus allen Teilen des Ableitungsbaums. Als Unternehmer interessieren uns natürlich die Kosten jedes Transportmittels pro Monat, und wir würden dazu eine Methode *getMonatsKosten* in der Basisklasse *Transportmittel* definieren. Ganz offensichtlich lässt sich diese dort aber nicht *implementieren*, denn beispielsweise die Berechnung der monatlichen Kosten unseres Fährschiffs gestaltet sich ungleich schwieriger als die der drei Fahrräder, die auch im Fahrzeugfundus sind.

Anstatt nun in aufwändigen Fallunterscheidungen für jedes Objekt zu prüfen, von welchem Typ es ist, muss lediglich diese Methode in jeder abgeleiteten Klasse implementiert werden. Besitzt das Programm etwa ein Array von *Transportmittel*-Objekten, kann dieses einfach durchlaufen und für jedes Element *getMonatsKosten* aufgerufen werden. Das Laufzeitsystem kennt den jeweiligen konkreten Typ und kann die korrekte Methode aufrufen (und das ist die aus der eigenen Klasse, nicht die in *Transportmittel* definierte).

INFO

Falls es vorkommt, dass die Implementierung in einer bestimmten Klasse mit der seiner Basisklasse übereinstimmt, braucht die Methode nicht noch einmal überlagert zu werden. Das Laufzeitsystem verwendet in diesem Fall die Implementierung aus der Vaterklasse, die der eigenen Klasse am nächsten liegt.

7.1.7 Fazit

Objektorientierte Programmierung erlaubt eine natürliche Modellierung vieler Problemstellungen. Sie vermindert die Komplexität eines Programms durch Abstraktion, Kapselung, definierte Schnittstellen und Reduzierung von Querzugriffen. Sie stellt Hilfsmittel zur Darstellung von Beziehungen zwischen Klassen und Objekten dar und sie erhöht die Effizienz des Entwicklers durch Förderung der Wiederverwendung von Programmcode. Wir werden in den nächsten Abschnitten zeigen, wie die objektorientierte Programmierung sich in Java gestaltet.

7.2 Klassen und Objekte in Java

7.2.1 Klassen

Eine Klassendefinition in Java wird durch das Schlüsselwort `class` eingeleitet. Anschließend folgt innerhalb von geschweiften Klammern eine beliebige Anzahl an Variablen- und Methodendefinitionen. Das folgende Listing ist ein Beispiel für eine einfache Klassendefinition:

Listing 7.1: Eine einfache Klassendefinition

```
001 /* Auto.java */
002
003 public class Auto
004 {
005   public String name;
006   public int    erstzulassung;
007   public int    leistung;
008 }
```

Diese Klasse enthält keine Methoden, sondern lediglich die Variablen `name`, `erstzulassung` und `leistung`. Eine solche methodenlose Klassendefinition entspricht dem Konzept des Verbunddatentyps aus C oder PASCAL (`struct` bzw. `record`). Die innerhalb einer Klasse definierten Variablen werden wir im Folgenden (analog zu C++) meist als *Membervariablen* bezeichnen. Die in Abschnitt 7.1.3, Seite 157 erwähnten Begriffe *Instanzvariablen* oder *Instanzmerkmal* sind aber ebenso gültig.

7.2.2 Objekte

Um von einer Klasse ein Objekt anzulegen, muss eine Variable vom Typ der Klasse deklariert und ihr mit Hilfe des `new`-Operators ein neu erzeugtes Objekt zugewiesen werden:

Listing 7.2: Erzeugen eines Objekts mit new

```
001 Auto meinKombi;
002 meinKombi = new Auto();
```

Die erste Anweisung ist eine normale Variablendeklaration, wie sie aus Kapitel 4, Seite 95 bekannt ist. Anstelle eines primitiven Typs wird hier der Typname einer zuvor definierten Klasse verwendet. Im Unterschied zu primitiven Datentypen wird die Objektvariable `meinKombi` als *Referenz* gespeichert. Die zweite Anweisung generiert mit Hilfe des `new`-Operators eine neue Instanz der Klasse `Auto` und weist sie der Variablen `meinKombi` zu.

In Java wird jede *selbstdefinierte* Klasse mit Hilfe des `new`-Operators instanziert. Mit Ausnahme von Strings und Arrays, bei denen der Compiler auch *Literale* zur Objekterzeugung zur Verfügung stellt, gilt dies auch für alle vordefinierten Klassen der Java-Klassenbibliothek.

TIPP

Wie bei primitiven Variablen lassen sich beide Anweisungen auch kombinieren. Das nachfolgende Beispiel deklariert und initialisiert die Variable `meinKombi`:

Listing 7.3: Kombinierte Deklaration und Initialisierung einer Objektvariablen

```
001 Auto meinKombi = new Auto();
```

Nach der Initialisierung haben alle Variablen des Objekts zunächst Standardwerte. Referenztypen haben den Standardwert `null`, die Standardwerte der primitiven Typen können Tabelle 4.1, Seite 98 entnommen werden. Der Zugriff auf sie erfolgt mit Hilfe der Punktnotation `Objekt.Variable`. Um unser `Auto`-Objekt in einen 250 PS starken Mercedes 600 des Baujahrs 1972 zu verwandeln, müssten folgende Anweisungen ausgeführt werden:

Listing 7.4: Zuweisen von Werten an die Variablen eines Objekts

```
001 meinKombi.name = "Mercedes 600";
002 meinKombi.erstzulassung = 1972;
003 meinKombi.leistung = 250;
```

Ebenso wie der schreibende erfolgt auch der lesende Zugriff mit Hilfe der Punktnotation. Die Ausgabe des aktuellen Objekts auf dem Bildschirm könnte also mit den folgenden Anweisungen erledigt werden:

Listing 7.5: Lesender Zugriff auf die Variablen eines Objekts

```
001 System.out.println("Name........: "+meinKombi.name);
002 System.out.println("Zugelassen..: "+meinKombi.erstzulassung);
003 System.out.println("Leistung....: "+meinKombi.leistung);
```

7.3 Methoden

7.3.1 Definition

Methoden definieren das *Verhalten* von Objekten. Sie werden innerhalb einer Klassendefinition angelegt und haben Zugriff auf alle Variablen des Objekts. Methoden sind das Pendant zu den *Funktionen* anderer Programmiersprachen, arbeiten aber immer mit den Variablen des aktuellen Objekts. *Globale Funktionen*, die vollkommen unabhängig von einem Objekt oder einer Klasse existieren, gibt es in Java ebenso wenig wie globale Variablen. Wir werden später allerdings Klassenvariablen und -methoden kennenlernen, die nicht an eine konkrete Instanz gebunden sind.

Die Syntax der Methodendefinition in Java ähnelt der von C/C++:

```
{Modifier}
Typ Name([Parameter])
{
 {Anweisung;}
}
```

Nach einer Reihe von *Modifiern* (wir kommen in Abschnitt 8.2, Seite 183 darauf zurück) folgen der *Typ* des Rückgabewerts der Funktion, ihr *Name* und eine optionale *Parameter-*

liste. In geschweiften Klammern folgt dann der *Methodenrumpf*, also die Liste der Anweisungen, die das Verhalten der Methode festlegen. Die Erweiterung unserer Beispielklasse um eine Methode zur Berechnung des Alters des `Auto`-Objekts würde beispielsweise so aussehen:

Listing 7.6: Eine einfache Methode zur Altersberechnung

```
001 public class Auto
002 {
003   public String name;
004   public int    erstzulassung;
005   public int    leistung;
006
007   public int alter()
008   {
009     return 2000 - erstzulassung;
010   }
011 }
```

Hier wird eine Methode `alter` definiert, die einen ganzzahligen Wert zurückgibt, der sich aus der Differenz des Jahres 2000 und dem Jahr der Erstzulassung errechnet.

7.3.2 Aufruf

Der Aufruf einer Methode erfolgt ähnlich der Verwendung einer Instanzvariablen in Punktnotation. Zur Unterscheidung von einem Variablenzugriff müssen zusätzlich die Parameter der Methode in Klammern angegeben werden, selbst wenn die Liste leer ist. Das folgende Programm würde demnach die Zahl 10 auf dem Bildschirm ausgeben.

Listing 7.7: Aufruf einer Methode

```
001 Auto golf1 = new Auto();
002 golf1.erstzulassung = 1990;
003 System.out.println(golf1.alter());
```

TIPP

Wie an der Definition von `alter` zu erkennen ist, darf eine Methode auf die Instanzvariablen ihrer Klasse zugreifen, ohne die Punktnotation zu verwenden. Das funktioniert deshalb, weil der Compiler alle nicht in Punktnotation verwendeten Variablen `x`, die nicht lokale Variablen sind, auf das Objekt `this` bezieht und damit als `this.x` interpretiert.

Bei `this` handelt es sich um einen Zeiger, der beim Anlegen eines Objekts automatisch generiert wird. `this` ist eine Referenzvariable, die auf das aktuelle Objekt zeigt und dazu verwendet wird, die eigenen Methoden und Instanzvariablen anzusprechen. Der `this`-Zeiger ist auch *explizit* verfügbar und kann wie eine ganz normale Objektvariable verwendet werden. Er wird als versteckter Parameter an jede nichtstatische Methode übergeben. Die Methode `alter` hätte also auch so geschrieben werden können:

Listing 7.8: Verwendung von this

```
001 public int alter()
002 {
003   return 2000 - this.erstzulassung;
004 }
```


TIPP

Manchmal ist es sinnvoll, `this` explizit zu verwenden, auch wenn es nicht unbedingt erforderlich ist. Dadurch wird hervorgehoben, dass es sich um den Zugriff auf eine Instanzvariable, und nicht eine lokale Variable, handelt.

7.3.3 Parameter

Eine Methode kann mit Parametern definiert werden. Dazu wird bei der Methodendefinition eine Parameterliste innerhalb der Klammern angegeben. Jeder formale Parameter besteht aus einem Typnamen und dem Namen des Parameters. Soll mehr als ein Parameter definiert werden, so sind die einzelnen Definitionen durch Kommata zu trennen.

Alle Parameter werden in Java per *call by value* übergeben. Beim Aufruf einer Methode wird also der aktuelle Wert in die Parametervariable kopiert und an die Methode übergeben. Veränderungen der Parametervariablen innerhalb der Methode bleiben lokal und wirken sich nicht auf den Aufrufer aus. Das folgende Beispiel definiert eine Methode `printAlter`, die das Alter des Autos insgesamt `wieoft` mal auf dem Bildschirm ausgibt:

Listing 7.9: Eine Methode zur Ausgabe des Alters

```
001 public void printAlter(int wieoft)
002 {
003   while (wieoft- > 0) {
004     System.out.println("Alter = " + alter());
005   }
006 }
```

Obwohl der Parameter `wieoft` innerhalb der Methode verändert wird, merkt ein Aufrufer nichts von diesen Änderungen, da innerhalb der Methode mit einer Kopie gearbeitet wird. Das folgende Programm würde das Alter des Objekts `auto` daher insgesamt neunmal auf dem Bildschirm ausgeben:

Listing 7.10: Wiederholter Aufruf der Methode zur Ausgabe des Alters

```
001 ...
002 int a = 3;
003
004 auto.printAlter(a);
005 auto.printAlter(a);
006 auto.printAlter(a);
007 ...
```

Wie bereits erwähnt, sind Objektvariablen Referenzen, also Zeiger. Zwar werden auch sie bei der Übergabe an eine Methode per Wert übergeben. Da innerhalb der Methode aber der Zeiger auf das Originalobjekt zur Verfügung steht (wenn auch in kopierter Form),

wirken sich Veränderungen an dem Objekt natürlich direkt auf das Originalobjekt aus und sind somit für den Aufrufer der Methode sichtbar. Wie in allen anderen Programmiersprachen entspricht die *call by value*-Übergabe eines *Zeigers* damit natürlich genau der Semantik von *call by reference*.

INFO

Die Übergabe von Objekten an Methoden hat damit zwei wichtige Konsequenzen:

- Die Methode erhält keine Kopie, sondern arbeitet mit dem Originalobjekt.
- Die Übergabe von Objekten ist performant, gleichgültig wie groß sie sind.

Sollen Objekte kopiert werden, so muss dies explizit durch Aufruf der Methode `clone` der Klasse `Object` erfolgen.

ACHTUNG

Die Übergabe von Objekten und Arrays per Referenz kann leicht zu verdeckten Fehlern führen. Da die aufgerufene Methode mit dem Originalobjekt arbeitet, kann sie deren Membervariablen bzw. Elemente verändern, ohne dass der Aufrufer es merkt. Auch der `final`-Modifier (siehe Abschnitt 8.2, Seite 183) bietet dagegen keinen Schutz. Das unbeabsichtigte Ändern einer modifizierbaren Referenzvariablen bei der Übergabe an eine Methode kann nur durch vorheriges Kopieren verhindert werden.

7.3.4 Variable Parameterlisten

JDK 1.1–6.0

Seit Java 5 gibt es die Möglichkeit, variable Parameterlisten zu definieren, in denen ein formaler Parameter für eine beliebige Anzahl aktueller Argumente steht. Dazu kann der letzte Parameter einer Methode (und nur dieser) nach dem Typbezeichner mit drei Punkten versehen werden. So wird angezeigt, dass an dieser Stelle beim Aufruf eine beliebige Anzahl Argumente des passenden Typs übergeben werden darf:

```
001 public static void printArgs(String... args)
002 {
003   for (int i = 0; i < args.length; ++i) {
004     System.out.println(args[i]);
005   }
006 }
```

Listing 7.11: Eine Methode mit einer variablen Parameterliste

Technisch entspricht die Deklaration der eines Arrays-Parameters und so wird auch auf die Elemente zugegriffen. Die Vereinfachung wird sichtbar, wenn man sich den *Aufruf* der Methode ansieht. An dieser Stelle darf nämlich nicht nur ein einzelnes Array übergeben

werden, sondern die einzelnen Elemente können auch separat angegeben werden. Dabei erzeugt das Laufzeitsystem automatisch ein Array, in das diese Werte übertragen werden. Die beiden folgenden Aufrufe sind also gleichwertig:

```
printArgs(new String[]{"so", "wird", "es", "gemacht"});
printArgs("so", "wird", "es", "gemacht");
```

Nun wird auch deutlich, warum lediglich der letzte Parameter variabel sein darf. Andernfalls könnte der Compiler unter Umständen nicht mehr unterscheiden, welches aktuelle Argument zu welchem formalen Parameter gehört.

Praktischen Nutzen haben die variablen Parameterlisten bei Anwendungen, in denen nicht von vorneherein klar ist, wie viele Argumente benötigt werden. Tatsächlich wurde ihre Entwicklung durch den Wunsch motiviert, flexible Ausgabemethoden definieren zu können, wie sie etwa in C/C++ mit der printf-Familie zur Verfügung stehen (und seit der J2SE 5.0 mit der Klasse `java.util.Formatter`, die in Abschnitt 11.6, Seite 278 beschrieben wird). Sie können dann die in diesem Fall vielfach verwendeten überladenen Methoden ersetzen (siehe Abschnitt 7.3.6, Seite 170). Natürlich benötigt nicht jede Methode variable Parameterlisten, sondern ihre Anwendung sollte auf Spezialfälle beschränkt bleiben.

Wird eine Methode mit einem Parameter vom Typ `Object...` deklariert, entstehen in Zusammenhang mit dem ebenfalls seit der J2SE 5.0 verfügbaren Mechanismus des Autoboxing (siehe Abschnitt 10.2.3, Seite 228) Methoden, bei denen praktisch alle Typprüfungen des Compilers ausgehebelt werden. Da ein Element des Typs `Object` zu allen anderen Referenztypen kompatibel ist und primitive Typen dank des Autoboxing automatisch in passende Wrapper-Objekte konvertiert werden, kann an einen Parameter des Typs `Object...` eine beliebige Anzahl beliebiger Argumente übergeben werden.

Das folgende Listing zeigt eine Methode, die numerische Argumente jeweils so lange summiert, bis ein nichtnumerischer Wert übergeben wird. Dieser wird dann in einen String konvertiert und zusammen mit der Zwischensumme ausgegeben. Am Ende wird zusätzlich die Gesamtsumme ausgegeben. Nicht unbedingt eine typische Anwendung und erst recht kein empfehlenswerter Programmierstil, aber das Listing demonstriert, wie weitreichend die Möglichkeiten dieses neuen Konzepts sind:

Listing 7.12: Die Anwendung von Object...

```
001 /* Listing0712.java */
002
003 public class Listing0712
004 {
005   public static void registrierKasse(Object... args)
006   {
007     double zwischensumme = 0;
008     double gesamtsumme   = 0;
009     for (int i = 0; i < args.length; ++i) {
010       if (args[i] instanceof Number) {
011         zwischensumme += ((Number)args[i]).doubleValue();
012       } else {
```

```
013         System.out.println(args[i] + ": " + zwischensumme);
014         gesamtsumme += zwischensumme;
015         zwischensumme = 0;
016       }
017     }
018     System.out.println("Gesamtsumme: " + gesamtsumme);
019   }
020
021   public static void main(String[] args)
022   {
023     registrierKasse(
024       1.45, 0.79, 19.90, "Ware",
025       -3.00, 1.50, "Pfand",
026       -10, "Gutschein"
027     );
028   }
029 }
```

Listing 7.12: Die Anwendung von Object... (Forts.)

Die Ausgabe des Programms ist:

```
Ware: 22.14
Pfand: -1.5
Gutschein: -10.0
Gesamtsumme: 10.64
```

7.3.5 Rückgabewert

Jede Methode in Java ist typisiert. Der Typ einer Methode wird zum Zeitpunkt der Definition festgelegt und bestimmt den Typ des Rückgabewerts. Dieser kann von einem beliebigen primitiven Typ, einem Objekttyp (also einer Klasse) oder vom Typ `void` sein. Die Methoden vom Typ `void` haben gar keinen Rückgabewert und dürfen nicht in Ausdrücken verwendet werden. Sie sind lediglich wegen ihrer Nebeneffekte von Interesse und dürfen daher nur als Ausdrucksanweisung verwendet werden.

Hat eine Methode einen Rückgabewert (ist also nicht vom Typ `void`), so kann sie mit Hilfe der `return`-Anweisung einen Wert an den Aufrufer zurückgeben. Die `return`-Anweisung hat folgende Syntax:

```
return Ausdruck;
```

Wenn diese Anweisung ausgeführt wird, führt dies zum Beenden der Methode und der Wert des angegebenen Ausdrucks wird an den Aufrufer zurückgegeben. Der Ausdruck muss dabei zuweisungskompatibel zum Typ der Funktion sein. Die in Kapitel 5, Seite 115 erläuterte Datenflussanalyse sorgt dafür, dass hinter der `return`-Anweisung keine unerreichbaren Anweisungen stehen und dass jeder mögliche Ausgang einer Funktion mit einem `return` versehen ist. Der in C beliebte Fehler, einen Funktionsausgang ohne `return`-Anweisung zu erzeugen (und damit einen undefinierten Rückgabewert zu erzeugen), kann in Java also nicht passieren.

7.3.6 Überladen von Methoden

In Java ist es erlaubt, Methoden zu *überladen*, d.h. innerhalb einer Klasse zwei unterschiedliche Methoden mit demselben Namen zu definieren. Der Compiler unterscheidet die verschiedenen Varianten anhand der Anzahl und der Typisierung ihrer Parameter. Haben zwei Methoden denselben Namen, aber unterschiedliche Parameterlisten, werden sie als verschieden angesehen. Es ist dagegen nicht erlaubt, zwei Methoden mit exakt demselben Namen und identischer Parameterliste zu definieren.

Der Rückgabetyp einer Methode trägt nicht zu ihrer Unterscheidung bei. Zwei Methoden, die sich nur durch den Typ ihres Rückgabewerts unterscheiden, werden also als gleich angesehen. Da Methoden auch ohne die Verwendung ihres Rückgabewerts aufgerufen werden können (was typischerweise wegen ihrer Nebeneffekte geschieht), hätte weder der Compiler noch der menschliche Leser in diesem Fall die Möglichkeit, festzustellen, welche der überladenen Varianten tatsächlich aufgerufen werden soll.

TIPP

Das Überladen von Methoden ist dann sinnvoll, wenn die gleichnamigen Methoden auch eine vergleichbare Funktionalität haben. Eine typische Anwendung von überladenen Methoden besteht in der Simulation von variablen Parameterlisten (die als Feature erst seit der Version 5.0 zur Verfügung stehen). Auch, um eine Funktion, die bereits an vielen verschiedenen Stellen im Programm aufgerufen wird, um einen weiteren Parameter zu erweitern, ist es nützlich, diese Funktion zu überladen, um nicht alle Aufrufstellen anpassen zu müssen.

Das folgende Beispiel erweitert die Klasse `Auto` um eine weitere Methode `alter`, die das Alter des Autos nicht nur zurückgibt, sondern es auch mit einem als Parameter übergebenen Titel versieht und auf dem Bildschirm ausgibt:

Listing 7.13: Überladen einer Methode

```
001 public int alter(String titel)
002 {
003   int alter = alter();
004   System.out.println(titel+alter);
005   return alter;
006 }
```

Die Signatur einer Methode

Innerhalb dieser Methode wird der Name `alter` in drei verschiedenen Bedeutungen verwendet. Erstens ist `alter` der Name der Methode selbst. Zweitens wird die lokale Variable `alter` definiert, um drittens den Rückgabewert der parameterlosen `alter`-Methode aufzunehmen. Der Compiler kann die Namen in allen drei Fällen unterscheiden, denn er arbeitet mit der *Signatur* der Methode. Unter der Signatur einer Methode versteht man ihren *internen* Namen. Dieser setzt sich aus dem nach außen sichtbaren Namen plus

codierter Information über die Reihenfolge und Typen der formalen Parameter zusammen. Die Signaturen zweier gleichnamiger Methoden sind also immer dann unterscheidbar, wenn sie sich wenigstens in einem Parameter voneinander unterscheiden.

7.3.7 Konstruktoren

In jeder objektorientierten Programmiersprache lassen sich spezielle Methoden definieren, die bei der Initialisierung eines Objekts aufgerufen werden: die *Konstruktoren*. In Java werden Konstruktoren als Methoden ohne Rückgabewert definiert, die den Namen der Klasse erhalten, zu der sie gehören. Konstruktoren dürfen eine beliebige Anzahl an Parametern haben und können überladen werden. Die Erweiterung unserer `Auto`-Klasse um einen Konstruktor, der den Namen des `Auto`-Objekts vorgibt, sieht beispielsweise so aus:

Listing 7.14: Definition eines parametrisierten Konstruktors

```
001 public class Auto
002 {
003   public String name;
004   public int    erstzulassung;
005   public int    leistung;
006
007   public Auto(String name)
008   {
009     this.name = name;
010   }
011 }
```

Soll ein Objekt unter Verwendung eines parametrisierten Konstruktors instanziert werden, so sind die Argumente wie bei einem Methodenaufruf in Klammern nach dem Namen des Konstruktors anzugeben:

Listing 7.15: Aufruf eines parametrisierten Konstruktors

```
001 Auto dasAuto = new Auto("Porsche 911");
002 System.out.println(dasAuto.name);
```

In diesem Fall wird zunächst Speicher für das `Auto`-Objekt beschafft und dann der Konstruktor aufgerufen. Dieser initialisiert seinerseits die Instanzvariable `name` mit dem übergebenen Argument »Porsche 911«. Der nachfolgende Aufruf schreibt dann diesen Text auf den Bildschirm.

TIPP

Explizite Konstruktoren werden immer dann eingesetzt, wenn zur Initialisierung eines Objekts besondere Aufgaben zu erledigen sind. Es ist dabei durchaus gebräuchlich, Konstruktoren zu überladen und mit unterschiedlichen Parameterlisten auszustatten. Beim Ausführen der `new`-Anweisung wählt der Compiler anhand der aktuellen Parameterliste den passenden Konstruktor und ruft ihn mit den angegebenen Argumenten auf.

Wir wollen das vorige Beispiel um einen Konstruktor erweitern, der *alle* Instanzvariablen initialisiert:

Listing 7.16: Eine Klasse mit mehreren Konstruktoren

```
001 public class Auto
002 {
003   public String name;
004   public int    erstzulassung;
005   public int    leistung;
006
007   public Auto(String name)
008   {
009     this.name = name;
010   }
011
012   public Auto(String name,
013               int    erstzulassung,
014               int    leistung)
015   {
016     this.name = name;
017     this.erstzulassung = erstzulassung;
018     this.leistung = leistung;
019   }
020 }
```

Default-Konstruktoren

Falls eine Klasse überhaupt keinen *expliziten* Konstruktor besitzt, wird vom Compiler automatisch ein parameterloser *default*-Konstruktor generiert. Seine einzige Aufgabe besteht darin, den parameterlosen Konstruktor der Superklasse aufzurufen. Enthält eine Klassendeklaration dagegen nur *parametrisierte* Konstruktoren, wird kein *default*-Konstruktor erzeugt, und die Klassendatei besitzt überhaupt keinen parameterlosen Konstruktor.

Verkettung von Konstruktoren

Unterschiedliche Konstruktoren einer Klasse können in Java verkettet werden, d.h. sie können sich gegenseitig aufrufen. Der aufzurufende Konstruktor wird dabei als eine normale Methode angesehen, die über den Namen `this` aufgerufen werden kann. Die Unterscheidung zum bereits vorgestellten `this`-Pointer nimmt der Compiler anhand der runden Klammern vor, die dem Aufruf folgen. Der im vorigen Beispiel vorgestellte Konstruktor hätte damit auch so geschrieben werden können:

Listing 7.17: Verkettung von Konstruktoren

```
001 public Auto(String name,
002               int    erstzulassung,
003               int    leistung)
004 {
005   this(name);
006   this.erstzulassung = erstzulassung;
007   this.leistung = leistung;
008 }
```

INFO

Der Vorteil der Konstruktorenverkettung besteht darin, dass vorhandener Code wiederverwendet wird. Führt ein parameterloser Konstruktor eine Reihe von nichttrivialen Aktionen durch, so ist es natürlich sinnvoller, diesen in einem spezialisierteren Konstruktor durch Aufruf wiederzuverwenden, als den Code zu duplizieren.

ACHTUNG

Wird ein Konstruktor in einem anderen Konstruktor derselben Klasse explizit aufgerufen, muss dies als erste Anweisung innerhalb der Methode geschehen. Steht der Aufruf nicht an *erster* Stelle, gibt es einen Compiler-Fehler.

Es gibt noch eine zweite Form der Konstruktorenverkettung. Sie findet automatisch statt und dient dazu, abgeleitete Klassen während der Instanzierung korrekt zu initialisieren. In Abschnitt 8.1.4, Seite 181 werden wir auf die Details dieses Mechanismus eingehen.

Initialisierungsreihenfolge

Beim Instanzieren eines neuen Objekts werden die Initialisierungsschritte in einer genau festgelegten Reihenfolge ausgeführt:

- Zunächst werden die Superklassenkonstruktoren aufgerufen, so wie es im vorigen Abschnitt beschrieben wurde.
- Anschließend werden alle Membervariablen in der textuellen Reihenfolge ihrer Deklaration initialisiert.
- Schließlich wird der Programmcode im Rumpf des Konstruktors ausgeführt.

Wir wollen dies an einem Beispiel veranschaulichen:

Listing 7.18: Initialisierungsreihenfolge

```
001 /* Listing0718.java */
002
003 public class Listing0718
004 {
005   public static String getAndPrint(String s)
006   {
007     System.out.println(s);
008     return s;
009   }
010
011   public static void main(String[] args)
012   {
013     Son son = new Son();
014   }
015 }
016
017 class Father
```

Listing 7.18: Initialisierungsreihenfolge (Forts.)

```
018 {
019   private String s1 = Listing0718.getAndPrint("Father.s1");
020
021   public Father()
022   {
023     Listing0718.getAndPrint("Father.<init>");
024   }
025 }
026
027 class Son
028 extends Father
029 {
030   private String s1 = Listing0718.getAndPrint("Son.s1");
031
032   public Son()
033   {
034     Listing0718.getAndPrint("Son.<init>");
035   }
036 }
```

Im Hauptprogramm wird eine neue Instanz der Klasse `Son` angelegt. Durch die Konstruktorenverkettung wird zunächst zur Vaterklasse `Father` verzweigt. Darin wird die Membervariable `s1` initialisiert und anschließend der Rumpf des Konstruktors ausgeführt. Erst danach führt `Son` dieselben Schritte für sich selbst durch. Die Ausgabe des Programms ist demnach:

```
Father.s1
Father.<init>
Son.s1
Son.<init>
```

7.3.8 Destruktoren

Neben Konstruktoren, die während der Initialisierung eines Objekts aufgerufen werden, gibt es in Java auch *Destruktoren*. Sie werden unmittelbar vor dem Zerstören eines Objekts aufgerufen.

Ein Destruktor wird als geschützte (`protected`) parameterlose Methode mit dem Namen `finalize` definiert:

Listing 7.19: Die finalize-Methode

```
001 protected void finalize()
002 {
003   ...
004 }
```

INFO

Da Java über ein automatisches Speichermanagement verfügt, kommt den Destruktoren hier eine viel geringere Bedeutung zu als in anderen objektorientierten Sprachen. Anders als etwa in C++ muss sich der Entwickler ja nicht um die Rückgabe von belegtem Speicher kümmern; und das ist sicher eine der Hauptaufgaben von Destruktoren in C++.

Tatsächlich garantiert die Sprachspezifikation nicht, dass ein Destruktor überhaupt aufgerufen wird. Wenn er aber aufgerufen wird, so erfolgt dies nicht, wenn die Lebensdauer des Objekts endet, sondern dann, wenn der Garbage Collector den für das Objekt reservierten Speicherplatz zurückgibt. Dies kann unter Umständen nicht nur viel später der Fall sein (der Garbage Collector läuft ja als asynchroner Hintergrundprozess), sondern auch gar nicht. Wird nämlich das Programm beendet, bevor der Garbage Collector das nächste Mal aufgerufen wird, werden auch keine Destruktoren aufgerufen. Selbst wenn Destruktoren aufgerufen werden, ist die Reihenfolge oder der Zeitpunkt ihres Aufrufs undefiniert. Der Einsatz von Destruktoren in Java sollte also mit der nötigen Vorsicht erfolgen.

Zusammenfassung

In diesem Kapitel wurden folgende Themen behandelt:

- Die Grundlagen der objektorientierten Programmierung
- Die Bedeutung der Begriffe *Abstraktion*, *Kapselung* und *Wiederverwendung*
- Die folgenden Beziehungen zwischen Objekten und Klassen:
  - Generalisierung und Spezialisierung
  - Aggregation und Komposition
  - Verwendung und Aufruf
  - Polymorphismus und Late Binding
- Die Bedeutung von Klassen und Objekten
- Definition und Aufruf von Methoden
- Methodenparameter und die Übergabearten *call by value* und *call by reference*
- Variable Parameterlisten
- Der Rückgabewert einer Methode und die `return`-Anweisung
- Überladen von Methoden
- Konstruktoren, Default-Konstruktoren und die Verkettung von Konstruktoren
- Destruktoren und das Schlüsselwort `finalize`